Table of Contents

dScope Functions

Error Messages

CPU Driver Files

Overview

What is dScope

dScope is a software debugger which simulates the hardware of the MCS 51, MCS 251 and 80C166 microcontroller family and can execute all machine instructions.   Simulation of the integrated peripherals is implemented by means of loadable drivers.   This makes dScope fully capable of simulating the integrated hardware of the various derivatives of the microcontrollers.   A corresponding driver exists for each controller type supported.   The software test is executed optionally either at the source level, assembler level, or a combination of both.

dScope Window Interface and Operation

dScope divides the screen into several fully user configurable areas.   Each window can be shown or hidden. A window is selected by clicking the mouse pointer   somewhere within the window or   by tabbing through the list of windows using **Ctrl+Tab**.   The following paragraphs include a short description for each of the dScope windows and their basic purpose:

Mainframe Window and Main Menu Line

The Mainframe window is the "parent" of all other dScope windows.   It hosts the main menu, a toolbar with shortcuts for showing and hiding the individual "child" windows.   A statusbar is located on the bottom of the mainframe.   It's purpose is to show a short description text for each menu entry.   The description is also available for the toolbar buttons.   To get it, select some button by moving the mouse pointer over a button.   Press the left mouse button,and keeping the left mouse button pressed, watch the text in the status bar.   Finally, move the mouse pointer out of the button and release the left mouse button.   Going this way, the command represented by one toolbar button is not executed.

The main menu line contains entries for sub menu selections.   A sub menu is selected either with the mouse or by simultaneously pressing the **Alt** key and the underlined character of the menu entry. Selection within a menu is also accomplished by using the mouse or the or ↓ keys followed by **Enter**, or by entering one of the underlined characters from the list of menu entries.

DEBUG Window

The DEBUG window is used to display: 1) source text of a loaded program, 2) the assembler instructions, or 3) a combination of both.   The display mode can be changed using the **Commands** menu entry from the menu line. Text can be scrolled upwards, downwards or horizontally by using the scrollbars or using the keys **PgUp**, **PgDn**, or the arrow keys. The **Commands** menu contains other commands dealing with trace recording, source module selection, text search and others.   The remaining menu entries are used to start and stop execution of the user program.   These entries do not have a pulldown menu   and are duplicated in the dialog bar to the right of the DEBUG window.   Therefore, it is possible to perform a single step, for example, by entering **Alt+S** or by clicking the **StepInto** entry of the menu line or by clicking the **StepInto** button from the dialog bar.

The status line of the DEBUG window shows the current display mode (HLL/MIXED/ASM) and the simulation engines status (RUN/STOP).   Also, when a menu command is selected, a brief description of the command is shown in the status bar.

COMMAND Window

The COMMAND window is used to enter command lines. The output from   most commands occur in the COMMAND window as well.   The advantage of the COMMAND window is that command lines can be put into disk files which can be read with the INCLUDE command.   This feature serves the purpose of setting up dScope without the need to activate different dialogs or windows for configuration of dScope.   The integrated dynamic syntax builder eases the

command entry by providing command context sensitive command help text.   Both the command output and the command lines entered are saved in two distinct history buffers, one for the output and one for the command lines entered for later recall.

REGISTER Window

The REGISTER window shows the values of the various CPU registers, the current program counter and executed cycles plus time information.   The register layout in the REGISTER window depends on the type of the CPU being simulated.   The register values are updated each time program execution is stopped or if the window is selected or double clicked with the left mouse button.

WATCH Window

The purpose of the WATCH window is to show the values of aggregate types (structures, unions, arrays) or simple scalar values.   The items to be displayed are defined or removed either by commands in the COMMAND window or with a specific dialog for simple operation of the WATCH window.   The WATCH window is updated after break out of execution but can be directed to update itself periodically while the user program is executed.

SERIAL Window

The serial outputs of the user program are displayed in the SERIAL window.   dScope traps read and write operations to the specific serial I/O registers of the CPU and directs the output to the SERIAL window and serial input to the CPU specific I/O register.   The details on serial I/O depends on the CPU type and is covered by the actually loaded CPU driver.   The serial output may also be directed to a disk file.   The SERIAL window maintains a history buffer which allows to view about 8K of serial output.

MEMORY Window

The MEMORY window is used to display a range of memory in hex bytes.   The range may be up to 64K in length.   The MEMORY window is updated after break out of execution but can be directed to update itself periodically while the user program is executed.

SYMBOL Browser Window

The SYMBOL browser window is used to show the complete symbolic debugging information of the loaded user program.   This information includes public symbols, functions with local symbols and line number information.   The output can be limited to symbols with a specific memory type or.   The SYMBOL browser also includes a simple but powerful regular expression search engine.

The SYMBOL browser supports the dragging and dropping of symbols to other dScope windows such as the COMMAND window and to the input fields of all dialogs which require expressions

for input such as define watch, inline assemble and others.

PERFORMANCE ANALYZER Window

The PERFORMANCE ANALYZER window shows the symbolic name or numeric address of up to 255 defined ranges and for each range a bar which indicates the execution time percentage consumed by each range.   The display gives a quick overview of the CPU time being consumed for each range.   The bottom of the window shows the minimum, maximum and medium time along with the invocation count of the selected range.

CALLSTACK Window

The CALLSTACK window shows the current function call nesting.   The caller of some function is shown on the bottom of the CALLSTACK window if a line from the display is selected.   The button labeled '**Refresh**' updates the function nesting output since the CALLSTACK window can be left open all the times even when the user program is executed.   Another button labeled '**Show invocation**' forces the DEBUG window to show the code which invoked the selected function.

CODE COVERAGE Window

The CODE COVERAGE window shows the functions based on the user selectable module together with the number of instructions of each function and the percentage value the instructions already executed.   Additional features include dynamic update while executing the user program and synchronizing the DEBUG window to the next executed or unexecuted instruction.

TOOLBOX Window

The TOOLBOX window contains user defined buttons which represent dScope commands.   Up to 15 command buttons can be defined, the first button labeled Reset is predefined and cannot be removed.

The Mainframe Window

The Mainframe is the "parent" of all other dScope windows.   It includes the main menu,   a   toolbar and a status bar.

see also:

The Toolbar Buttons

Obtaining Help for a Toolbar Button

Menu Commands

The Toolbar Buttons

The toolbar contains a number of buttons which duplicates commands of the **View** menu.   An exception is the combo box (the second item) which provides the list of currently available CPU driver DLL's.   The following list shows the toolbar buttons and the function associated with each button.

Starts the **Load Object File**dialog

CPU driver DLL selectionShows the currently selected driver DLL or blank, if none is selected.   To get the list of drivers, drop down the combo box (click at the arrow symbol) and select your requested driver.

Show/hide **COMMAND** window

Show/hide **DEBUG** window

Show/hide **REGISTER** window

Show/hide **SERIAL** window

Show/hide **WATCH** window

Show/hide **PERFORMANCE ANALYZER** window

Show/hide **MEMORY** window

Show/hide **SYMBOL BROWSER** window

Show/hide **STACK** window

Show/hide **CODE COVERAGE** window

Show/hide **TOOLBOX** window

Reset dScope

Show the **About** dialog

Obtaining Help for a Toolbar Button

To obtain a brief help text on some toolbar button, do the following:

- ◆ Click at the toolbar button which you want to get help for (left mouse button)

- ◆ *Keep the mouse button pressed* and watch the help text in the status bar.

- ◆ Move the mouse pointer outside the toolbar button.

- ◆ Release the left mouse button.

The third step cancels the help text in the status bar and together with the fourth step cancels the command for the selected toolbar button.

Menu Commands

The following sections describe the individual menu commands of the Mainframe window.

Load object file:

Opens the load file dialog.   Select the drive and path and finally the name of the file you want to load. dScope supports OMF and HEX files.   The dScope loader will automatically decide which type of object file so you don't have to distinguish between the different file formats.   If the loader detects an unknown or invalid file format, it will notify you by giving you a message box and canceling the load operation.

Load CPU driver

The **Load CPU driver** command drops down the toolbar combo box containing the list of currently available drivers.   Select the appropriate driver for the debugging of your application.

Exit dScope

This command exits dScope and returns control to Windows.   Note that dScope will cancel Exit if either the user program is still executed or a dScope user function is running.
To get out of this situation, close the message box, stop the execution of the user program (see DEBUG window) or any active signal functions (SIGNAL KILL command) and then repeat the Exit command.

View Menu Commands

The commands listed in the menu perform the actions listed as follows:

Toolbar	Show or hide the toolbar.
Status Bar	Show or hide the status bar.
Register window	Show or hide the REGISTER window.
Debug window	Show or hide the DEBUG window.
Serial window	Show or hide the SERIAL window.
Command window	Show or hide the COMMAND window
Watch window	Show or hide the WATCH window.
Performance Analyzer	Show or hide the PERFORMANCE ANALYZER window.
Memory window	Show or hide the MEMORY window.
Symbol window	Show or hide the SYMBOL browser window.
Code Coverage window	Show or hide the CODE COVERAGE window.
Toolbox	Show or hide the TOOLBOX window.
Call-Stack	Show or hide the CALL-STACK window.

Setup Menu Commands

The Setup menu contains commands to change dScope settings:

Colors and fonts...	Opens the color and font configuration dialog. Each of the individual dScope windows be configured with different background and text colors as well as fonts. See *dScope Dialogs* for details on how to setup colors and fonts.
Update Memory window	Enables/Disables periodic update of the MEMORY window contents. The periodic update is performed while executing the user program. If disabled, the memory window is updated only after breaking out of execution.
Update Watch window	Enables/Disables periodic update of the WATCH window contents. The periodic update is performed while executing the user program. If disabled, the memory window is updated only after breaking out of execution.
MCS 51 Registers	Configures the REGISTER window to show the MCS 51 registers.
MCS 251 Registers	Configures the REGISTER window to show the MCS 251 registers.
Breakpoints...	Activates the Breakpoint dialog (see *dScope Dialogs* for details).
Watchpoints...	Activates the Watchpoint dialog (see *dScope Dialogs* for details)..
Memory map...	Activates the Memory map dialog (see *dScope Dialogs* for details)..
Setup Performance Analyzer	Activates the Performance Analyzer setup dialog (see *dScope Dialogs* for details)..
Reset dScope	Resets dScope. This means that the current PC and the CPU driver is reset. This is equivalent to the Reset hardware signal to the CPU. All settings, memory map and debug information is still valid.

Peripheral Menu Commands

The Peripheral contains commands for invocation of various peripheral dialogs.   Such dialogs are defined by the CPU driver.   Each on chip peripheral is supported by dScope by one or more dialogs. Each dialog covers a specific peripheral function group such as timer-0, timer-1, interrupts or ports.

NOTE: the Peripheral menu is dynamically created when a CPU driver DLL is loaded.   If no driver has been loaded previously, then the Peripheral menu is invisible!

Help Menu Commands

Index	Start Winhelp, show the index of all items.
Commands	Start Winhelp with the dScope command list.
About dScope...	Shows the About dialog containing the dScope version number.

The DEBUG Window

The DEBUG window is the most frequently used window of dScope, since most program execution and trace functions are provided by this window.   The DEBUG window contains a dialog bar, status bar and a tool bar.   Both the dialog bar and the tool bar duplicate the most frequently used menu commands such as **Go**, **StepInto** or **View Trace**.   The Status bar has three panes.   The first pane displays help information when a menu command is to be selected.   The second pane displays the actual view mode (HLL/MIXED/ASM).   The third pane displays the current execution status: RUN or STOP.

see also:

The Toolbar Buttons

Scrolling

Set and Remove an Execution Breakpoint

DEBUG Menu Commands

Selecting a source module for display

Searching for text in the DEBUG window

Inline Assembler dialog

Trace recording

Logging DEBUG window contents to file

The Toolbar Buttons

If you want to have more horizontal space, you may turn off the dialog bar and turn on the tool bar. Toggle **Show Dialogbar** and **Show Toolbar** in the command menu.
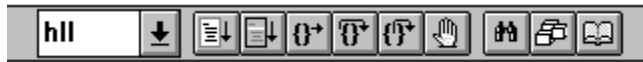
The meaning of the tool bar buttons (left to right) is:

- **Select display mode** (hll, mixed, asm)
- Go
- Go till cursor line
- StepInto
- StepOver
- StepOut
- Stop
- View trace records
- **Select source module** (activates dialog box)
- **Find text** (activates dialog box)

Scrolling

The DEBUG window supports both vertical and horizontal scrolling.   Scrolling can be performed either by using the keyboard or the scrollbars.   The keys **PgDn**, **PgUp** and the **cursor keys** may be used to scroll.

Scrolling with the scrollbars is 'as usual' in many Windows programs.   Clicking the area between the arrow symbols and the thumb tracker is equivalent to page up/down or 10 characters left/right.

NOTE
the vertical thumb is always positioned in the middle of the scroll bar.   Dragging it in either direction forces the selection of one out of 256 64K segments.   The disassembly is resyncronized to the newly selected segment.   Since dScope simulates up to 16M byte of memory, dragging the thumb to the top represents segment 0, whereas the bottom represents segment 255 (this is the default code segment for MCS 51 and MCS 251).   After releasing the dragged thumb, it will reposition itself into the middle of the scrollbar.

Set and Remove an Execution Breakpoint

You may set one or more execution breakpoints by double clicking the left mouse button at the desired source or assembly line in the DEBUG window.   The selected line will be redrawn using the BP-Highlight color and also receives a label reading **[BR n]** where n represents a the breakpoint number assigned by dScope.   Double clicking the left mouse button at a line with an execution break will remove that breakpoint.   If an attempt is made to set a breakpoint at a source line which generated no object code, dScope will beep in this case and not set a breakpoint.

DEBUG Menu Commands

The DEBUG window has it's own menu line containing the relevant commands that are most often required when debugging user programs.   The advantage here is that you can maximize the window to full screen size and still have the relevant commands for debugging available.   Most of the commands are self explaining.   The more complex commands are described in the following the command tables.

Commands   See next table.

Go!   Start execution of the user program from the current program counter (PC).

GoTilCurs!   Start execution of the user program from the current program counter (PC).   Execution is stopped if the current cursor line is reached.   The cursor line is selected by simply moving the mouse pointer to the desired line and then selecting it by a single click of the left mouse button.
             dScope provides an additional shortcut for the GoTilCurs command.   Double clicking the right mouse button at the desired line starts execution and stops, if the selected line is reached.

StepOut!   Start execution from the current program counter and stop if the current function is left and execution returns to the statement following the function invocation.   dScope internally maintains a list of currently nested function calls and therefore knows the code address where a function invocation took place.   If no function calls are present in the list, then the StepOut command is not performed.

StepInto!   This command forces execution of the next statement.   The definition of statement depends on the current view mode: in HLL mode, a statement defines a high level statement.   In ASM or MIXED mode, a statement defines a single assembler instruction.   In any case, the StepInto command does not treat a function call as one statement, therefore it steps into the function.

StepOver!   The StepOver forces execution of the next statement.   It works much the same as the StepInto command with the exception that a function call is treated as one statement. That means, if a function call is executed, the whole function plus possible nested function calls are taken as one statement.

Stop!   Stops execution of the user program.

Entries of the Commands menu item:

View High level	Switch view mode to HLL (high level language) mode. If the currently disassembled address range has associated line number information and the source/list file (depending on module type C/ASM) is accessible, then the source lines are displayed instead of plain assembly lines. If the before mentioned is not true, then the view mode is still changed, but the display itself will not change. In this case, the display will automatically change to HLL once an address range with sufficient line number information is disassembled.
View Mixed	Switch to MIXED mode. . If the currently disassembled address range has associated line number information and the source/list file (depending on module type C/ASM) is accessible, then the display will show source lines intermixed with assembly lines. This mode is intended for analyzing the compiler generated code for a specific source line, for example.
View Assembly	Switch to ASM mode. The display will show assembly lines only. This mode is the default if an address range with no line number information or inaccessible source or list file is disassembled.
View Trace Records	Switch to trace history mode. In order to do this, the Record trace command (also contained in this menu) has to be issued and then program execution must be started. Note that execution must be stopped before the trace history can be viewed.
	If the trace history is non empty, then the DEBUG window will show a view lines of the trace records on the upper half of the screen. Use the cursor keys or the vertical scrollbar to view deeper into the history. Note that the values in the register window are also changed if the cursor line is moved. dScope records the execution address along with the register values on execution of each assembly instruction.
Select source module	Displays a dialog box showing a list of currently available high level modules. The DEBUG window will get synchronized immediately if a new module is selected.
Inline assemble	Display the inline assembler dialog.
Find...	Displays the Find dialog box. The dialog is

modeless, therefore you can start searching text with program execution still running.

Show Dialogbar

Show or hide the dialog bar.   If the Dialog bar is hidden, then you must use the menu to issue Go or Step commands.

Show Statusbar

Show or hide the status bar.

Show Toolbar

Show or hide the tool bar.

Record trace

Enable/disable trace recording.

Set log start line

Set the starting line for later file log.

Perform File log

Log the area from the starting line up to the current cursor line to the currently active log file.

Selecting a source module for display

dScope normally displays the module which is derrived from the current program counter (PC).   You may change the module by using the Module-Selection dialog.   From the **Commands** menu, choose **Select source module**.   You will get a dialog box showing the names of all modules which have high level debugging information and whose source file is accessible.   Select some module and click the **Show selected module** button.   The display in the DEBUG window will immediately switch to and display the newly selected module.   If you are finished with module selection, use the system menu located to the top left of the dialog to close the dialog.

Searching for text in the DEBUG window

Within the DEBUG window, text can be searched for.   To do that, select the   **Commands** menu, then choose **Find**.   dScope displays the Find dialog:
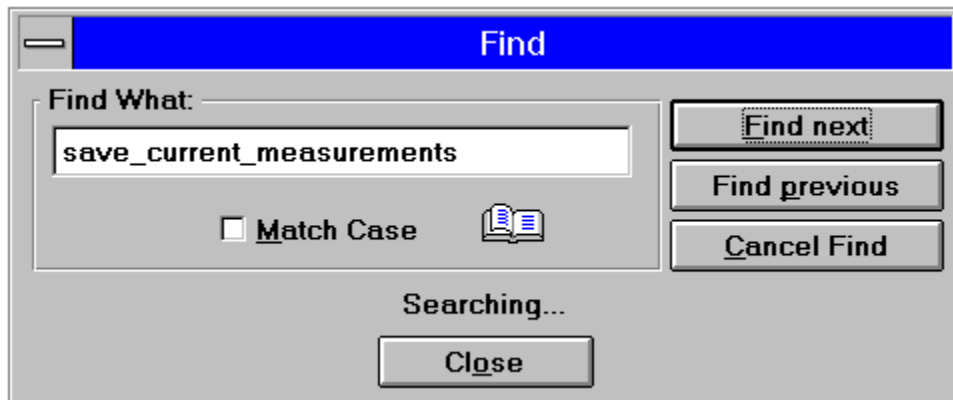
Enter the text you want to search for in the edit field.   If the text search is case sensitive, then check the **Match Case** checkbox.   Start the search by clicking the **Find next** button to search downwards or **Find previous** to search in reverse direction.   If the given text is found, then the DEBUG window will synchronize to the context where the text was found with the cursor line marking the text found.   You can start searching the text even if the user program is executing.

NOTE   the search may run for a long time, especially if the search runs over the entire 16M byte range. In this case, you may select the **Cancel** button at any time to stop searching.   If

Inline Assembler dialog

This command invokes the inline assembler dialog.   Refer to <u>Inline Assembler dialog</u>.

Trace recording

For finding errors in programs, it is often desired to view the 'near past' of the execution.   dScope maintains a trace history buffer which is capable of holding the most recently executed 512 assembler instructions.   After execution of each instruction, the following information is captured:

- ◆ the program counter
- ◆ the program status word(s)
- ◆ the content of all CPU registers

NOTE   the registers actually recorded by dScope depends on the selected CPU driver.

To get the trace history, perform the following steps:

- ◆ From the Commands menu, choose **Record trace**.   The menu line will show a check mark, if trace recording is enabled.   Otherwise no checkmark is shown.
- ◆ Perform some execution command: Go, Step, ...
- ◆ Stop the execution.   Note that execution must be stopped before the trace history can be viewed.
- ◆ Choose the **View trace** button from the dialog bar or the **View trace records** item from the Commands menu.

The upper half of the DEBUG window shows the most recently executed instructions.   The history lines are preceded by a minus sign with the numeric trace entry number.   The lines displayed following the history lines are the lines that will be executed upon next execution.   Use the cursor up and down keys or the vertical scroll bar to   scroll deeper into the history.   Note that the values in the REGISTER window will get updated when the cursor line is moved within the history lines.   This is because the register values are also recorded with each trace frame.

Note that any execution command such as Go, Step or Stepout automatically removes the history display before execution is started.   This is to avoid having outdated history lines on screen.

Logging DEBUG window contents to file

Sometimes you may want to write a range of code to a file.   This could happen when you want to analyze a piece of code which you don't have the source code for as is the case for example on third party libraries you linked into your application.   To do that, perform the following steps:

◆ In the COMMAND window. open a log file first by entering the command:
LOG >path and name of log file you want
for example:
LOG >C:\TMP\LOGFILE
this will create the file LOGFILE in path C:\TMP.   Instead of using the right angle   character > you can also use the double right angle >> to append to an existing file.

◆ In the DEBUG window move or scroll the cursor line to the desired starting line for log.

◆ From the Commands menu, choose 'Set log start line'.   dScope now captures the the given line.

◆ In the DEBUG window, move or scroll the cursor line to the last line which should be logged.

◆ From the Commands menu, choose 'Perform File log'.   This will actually write the text in the given bounds to the log file.

◆ In the COMMAND window, enter the command:
LOG OFF
This command flushes and then closes the log file.

dScope writes the text to the log file in plain ASCII format which can be processed by any text editor.

The Command Window

The COMMAND window is used to enter dScope commands.   The results of many dScope commands are displayed in the COMMAND window.   Although most dScope commands can be performed by use of various dialog boxes, direct command entry of almost all dScope commands in the COMMAND window is supported.

Commands can be put into disk files and included within dScope's COMMAND window.   Such files, called command files contain plain ASCII text lines.   Command files have the advantage that you can put the dScope commands such as -   load CPU-driver, map, load obj-file, set breaks or any other command into them.   Once a command file exists, use INCLUDE   to execute the commands of the file.
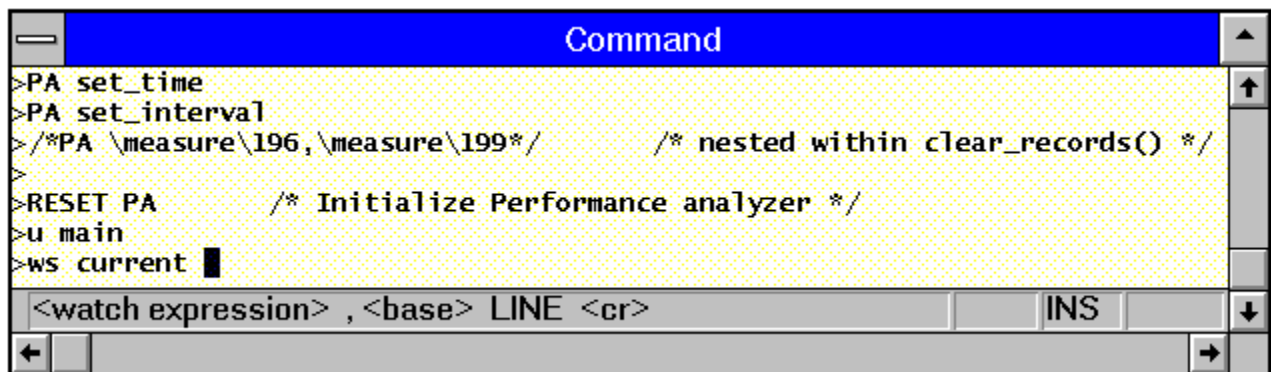
The COMMAND window maintains two history buffers: one for the command output and another one for command lines entered.   The command output buffer receives all characters that are output by dScope commands.   The command output buffer holds the most recent 32K characters of output.   The command line history buffer holds the most recently entered 64 input lines.   Any line of the command line history can be recalled and again used for command input.

see also:

Recalling a line from the command line history

Scrolling the COMMAND Window Content

Status Bar

Syntax Generator

Command Interpreter

Editing a Command Line

Comment Lines

Chaining Commands

Recalling a line from the command line history

Click into the COMMAND window (this moves the input focus).   Use the Cursor-Up and Cursor-Down keys to recall lines out of the history buffer.

Note that the Scroll Lock key must be disabled for recall of command lines.   If the Scroll Lock key is enabled, then the Cursor-Up and Cursor-Down key will scroll the command output history rather than the command line history.

Scrolling the COMMAND Window Content

Scrolling of the command output history can be performed by either using the scroll bars of the COMMAND window or by using the following keys:

- PgDn - scroll window contents on page downwards.
- PgUp - scroll window contents one page upwards.
- Cursor down - scroll window one line downwards.
- Cursor up - scroll window one line upwards.
- Cursor right - scroll window one character to the right.
- Cursor left - scroll window one character to the left.

The Cursor left and Cursor right keys work as described when the Scroll-lock key is enabled. If it is disabled, the cursor is moved within the boundaries of the current input line.

Status Bar

The Status Bar is displayed on the bottom of the COMMAND window.   It has four panes where each pane has a special function:

Pane #1   The built in syntax generator displays it's syntax help messages
           in this pane.

Pane #2   This pane displays 'TAB' if the user can view more command
           syntax options by entering TAB.   If no more command options
           are available, then Pane 2 is blank.

Pane #3   Displays INS or OVR.   This is the current writing mode for the
           command line editor, INSert or OVeRwrite.

Pane #4   Displays the current state of the Scroll Lock key. SRCL means
           Scroll Lock is enabled, a blank pane means Scroll Lock is
           disabled.

The HELP text is continuously displayed in the first status bar pane during the command entries.   This reflects the syntax of the command (syntax generator) and informs the user about command keywords, necessary parameters, and command options.   The command entry is simultaneously simplified, since options from the HELP line can be transferred very easily in the command line.

Syntax Generator

The command menu of the syntax generator is displayed in the first status bar pane.   During the command entry, the syntax generator displays possible commands, options, and, parameters.   The display of 'TAB' at the end of the line indicates that additional information is available by entering **Tab**.   Selection of a command is performed by entering the first command option appearing in the HELP line.   However, only uppercase letters for the option can be entered.   In the case of command entry, the HELP line is automatically reduced to the options still remaining.   If only one fixed option remains, the entry of a blank character is enough to enter the entire command option.   For example:

```
>E_
```

After the entry of the command, command options appear in the first pane.   The features named above also apply here.   After entering "E", the following help text line appears:

```
>E_
 Evaluate  EXIT  <CR>
```

The HELP line refers to the available parameters: "EVALuate", "EXIT" and finally "CR", which means **Enter**.   If "V" is entered next, the following entry also suffices:

```
>EV<space>   /* dScope expands to 'EVAL' */
>EVAL _
 <expression>
```

The syntax generator in the HELP line then refers to the parameter "expression" that has to be entered.   The syntax generator assists in learning dScope command syntax and helps to avoid errors.   If the syntax generator cannot follow the syntax of a command, the message '**no help available'** is issued in the status pane.   This generally indicates an improper or illegal entry.

Command Interpreter

dScope contains a command interpreter that executes all commands and functions.   The command entry always occurs in the last line of the COMMAND window.   dScope supports dynamic help text generation to ease command entry.   This line displays the possible commands, parameters, and options, used in connection with the syntax generator.

Editing a Command Line

Command lines can be manipulated using the following control keys:

Enter	Execute entire entry line.
Backspace	Delete character in front of the cursor.
Ctrl+D, Ctrl+F, Del	Delete character under the cursor.
Esc, Ctrl+C	Abort entry and start new entry line.
Home	Position cursor at the beginning of the entry line.
End	Position cursor at the end of the entry line.
Ins	Toggle between insert/overwrite mode.
Tab	Display additional information in the HELP line.
← †	Move cursor a position to the left.
→ †	Move cursor a position to the right.
†	Assume an prior entry line in the line editor.
↓ †	Assume a later entry line in the line editor.

† These control keys only work when   **Scroll Lock**   is disabled.

Comment Lines

dScope allows the entry of comments according to C or PL/M programming language conventions.   A comment cannot extend over more than one line in command mode.   For the definition of dScope functions, comments extending over several lines are permissible.

NOTE The syntax generator does not support multiline comments and commands chained after a comment.

Example:
```
>BS TEST      /* set a breakpoint at address 'TEST' */
>FUNC void abc (void)  {
>  1:   /* this is a
>  2:      multiline comment */
>  3:  }
```

Chaining Commands

Several commands can be entered in a dScope command line separated by a semicolon.   This is necessary, for example, when a function key, **F1** to **F10**, is to be assigned to several commands.

NOTE   The syntax generator supports only the first command entered.

Example:
```
>G , \MEASURE\228 ; OBJ CMDBUF   /* two commands */
>SET F2 = "OBJ current; D X:0x4000,X:0x4FFF"
```

The Watch Window

The WATCH window is used for permanent display of scalars, structures, unions or arrays. The WATCH window can be active at any time including while execution of the user program is performed.   The window contents are updated each time the execution is stopped, for example after a Single-Step, StepOut or if the Go command is stopped.   Also, the WATCH window can be set to 'periodic' update where the content is updated each time 1255 instructions have been executed.   The values changed since the last time of update will be highlighted:
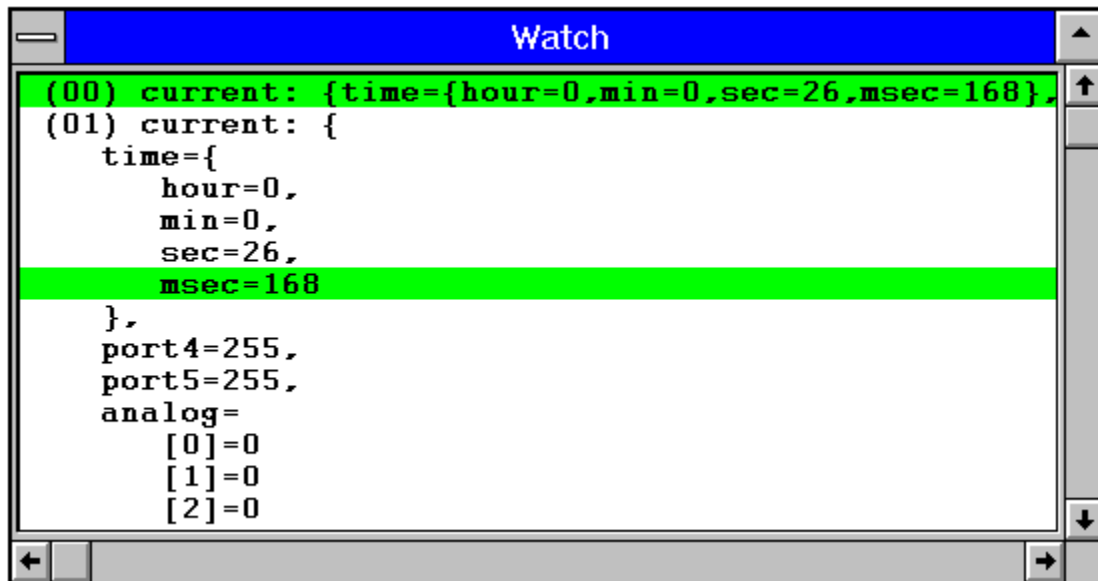
Watch expressions can be in single line or multiline mode.   In single line mode, the content of the given expression is displayed in one line, which may cause truncation of the output at about 128 characters.   In multiline mode, the components of for example a structure or an array are displayed in distinct line.   The example screen of the WATCH window shows two identical watch expressions.   The first one is displayed in single line mode, the second one uses multiline mode.   A single line watch is considered changed, if one component changes the value which means that the whole line is highlighted.

The WATCH window may be hidden by either using it's system menu or by using the main menu **View - Watch window** or the associated toolbar button.

see also:

Scrolling the Watch window

Defining a watch expression

Toggling periodic Watch update

Removing a watch expression

Scrolling the Watch window

The scroll bars of the WATCH window are enabled if the number of lines exceeds the client area height or some line exceeds the client area width.   Use the scroll bars to move vertically and horizontally.   The WATCH window contents will stay in the given position until either the window is resized or a watch point is added or removed.

Defining a watch expression

From the dScope main menu, choose **Setup - Watchpoints.** This command will bring up the **Watchpoints** dialog:
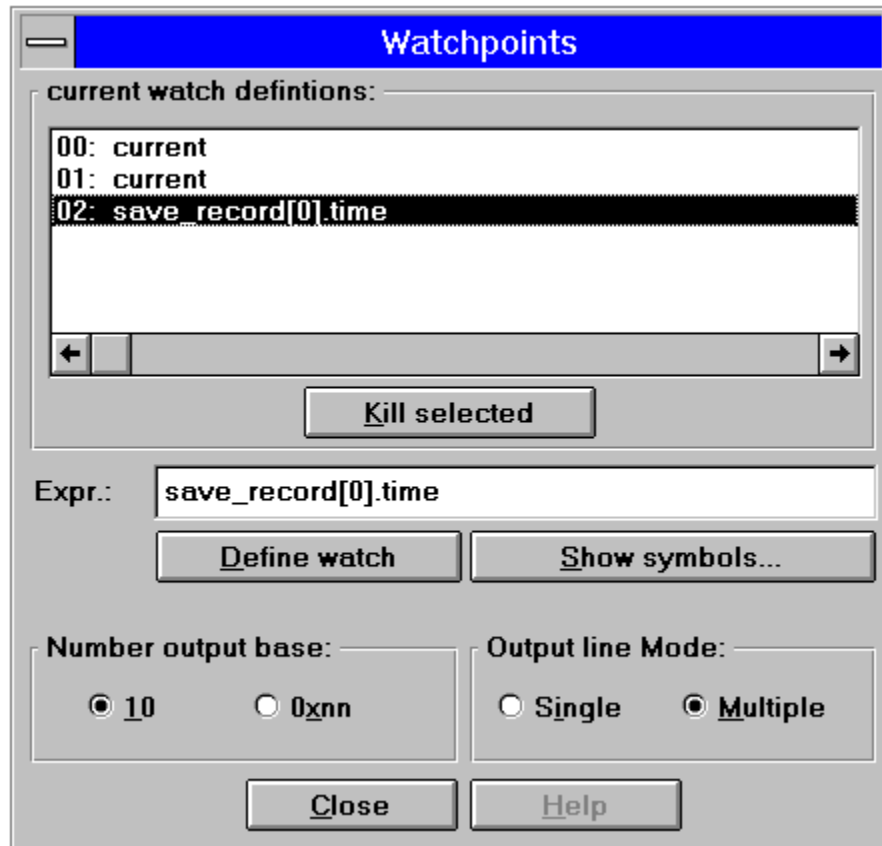
- ◆ In *Expression* field, enter the name of the symbol or an expression. To ease the entry of symbols, you may select the **Show symbols** button which opens the Symbol browser window (if not already open). From the Symbol browser, you can drag a symbol into the *Expression* field as follows:

    > Move the mouse pointer to the requested symbol.
    > Press the left mouse button, and don't release the button (the cursor changes).
    > Move the mouse pointer to the *Expression* field of the **Watchpoints** dialog
    > Release the mouse button. The qualified symbols name is then filled in.

- ◆ Select the desired *Number output base* , decimal or hex.

- ◆ Select the *Output line mode*, single or multiline mode. For single scalar values such as an integer, a byte, bit or float value, the output line mode has no influence since scalars are always displayed on a single line.

- Select the **<u>Define watch</u>** button to define a given watch expression.   If the given expression is correct, then the edit field will be cleared and the watch expression is transferred to the list box.   The leading number in front of the watch expression is a dScope assigned watch number which used to refer to watches when the command line (WK command) is used to delete watches.

NOTE Watch definitions can also by defined with the WS command and removed with the WK command in the command window.   Refer to dScope Commands for more information on Watch commands.

Toggling periodic Watch update

Normally, dScope updates the WATCH window each time an execution command (StepInto, Go, ...) stops.   No update takes place while execution is running.   You can force periodic updates as follows:

- ◆ From dScopes main menu, choose **Setup - Update Watch window**.   This will enable periodic update of the WATCH window while execution is running.   Choosing **Setup-Update Watch window** a second time will toggle periodic update.

With this feature activated, the update will take place each time 1255 CPU instructions have been executed.

NOTE Periodic update with huge structures or arrays may slow down execution speed of the simulation engine.

Removing a watch expression

From the dScope main menu, choose **Setup - Watchpoints**.   This will bring up the <u>The Watchpoint Dialog</u> with the list box filled with the currently defined watch expressions.

- ◆ Select the watch expression you want remove.   Move the mouse pointer to the desired watch entry and click the left mouse button.
- ◆ Select the **<u>Kill watch</u>** button to remove the selected watch expression.

The Register Window

The REGISTER window is used to display the values of the CPU registers, additional information on cycles executed so far, and the computed total execution time based on the value of the current crystal frequency.   dScope for MCS 51, MCS 251 and 80C166 uses different layouts for displaying the content since the registers of these families are different. The appropriate layout for display is automatically selected if some CPU driver is loaded into dScope but can also be changed at any time using the **MCS 51 Registers** or **MCS 251 Registers** command from dScope's main menu.

NOTE The commands **MCS 51 Registers** and **MCS 251 Registers** are not available on dScope 166 for Windows.

If for example some MCS 51 CPU driver such as 8051FX.DLL or 80517.DLL   has been loaded, then the the REGISTER window looks like this:

```
        Regs              ▲
  A=00       B=01
 R0=47    R1=01
 R2=09    R3=05
 R4=00    R5=00
 R6=00    R7=00
DPTR: 09E4
  SP: 0079
 PSW: ---R0---
   $:      C:0D4C
 Cyc:  12369380
 Sec: 12.369380
```

The display contains the values for the register A, B, R0 to R7, DPTR, SP and PSW.   The line identified by **$** represents the current execution point, commonly known as program counter. The line starting with **Cyc** (Cycles) shows the number of cycles executed.   The last line marked **Sec** (Seconds) shows the total execution time so far.   The execution time is correlated to current value of the XTAL variable which holds the default value of 12000000, which means 12Mhz (default value for MCS 51 CPU drivers).   Register Values that changed since the last update are shown using the highlight color which can be configured using dScope's color configuration dialog.

The   MCS 251 register display is different from the MCS 51 registers:

```
┌──┬──────────────┬──┐
│──│    Regs      │▲ │
├──┴──────────────┴──┤
│   R0: 2B 37 00 04  │
│   R4: 00 04 00 02  │
│   R8: 00 00 00 2B  │
│  R12: 00 00 00 00  │
│ WR16: 0000  0000   │
│ WR20: 0000  0000   │
│ WR24: 0000  0000   │
│ WR28: 0000  0000   │
│ DPTR: 0001:10B5    │
│   SP: 0000:0080    │
│  PSW: C--R1---     │
│ PSW1: --------     │
│    $:     C:0113   │
│  Sts:  21095783    │
│  Sec: 21.095783    │
│                    │
└────────────────────┘
```

The first four lines show the values for the registers R0 to R15, where R11 represents the accumulator.   The second four lines show the values of the MCS 251 word registers.   Note that the registers WR0 to WR14 overlay the byte registers, so the values of R0 and R1 (0x2b37) represent the value of WR0.   The DPTR value is shown 32 Bit format since the segment value of the DPTR may refer to any segment of the 16M byte address range.   The program status word is split into two lines which reflect the implementation of the MCS 251 program status words.

see also:

Register Window update

Changing Register values

Register Window update

The window contents are updated each time the execution is stopped, for example after a Single-Step, StepOut or if the Go command is stopped.   The window gets updated if it is selected (by clicking somewhere within the REGISTER window, for example) or by double clicking the left mouse button somewhere within the REGISTER window.

Changing Register values

Register values can be changed in the Command window by simply entering the register name followed by an assignment or any valid C style expression.   To do that, select the Command window and enter one or more assignment expressions.   The paragraph below shows a few examples of register assignments:

```
A = 0xFE                /* assign 0xFE to Accumulator */
DPTR = 0x1234              /* assign 0x1234 to DPTR */
WR20 += --DPTR            /* note C style operators */
R0=3,R1=4,R2=5        /* comma separated expressions */
R7=current.time.sec--       /* program symbols used */
DR56=0x20000        /* set MCS 51 xdata segment to 2 */
```

The Serial Window

The SERIAL window emulates a serial terminal connected to the MCS 51/251 or 80C166 special function registers or some other special function registers (ie. a parallel port.)   In the case of a running 8051 program, all key entries, except for dScope's control keys, are passed to the user program using the serial interface.

```
+-------------------------------- Serial I/O --------------------------------+
| and the voltage on the four analog inputs AN0 through AN
+ command -+ syntax -----+ function --------------------
| Read       | R [n]        | read <n> recorded measurements
| Display    | D            | display current measurement val
| Time       | T hh:mm:ss   | set time
| Interval   | I mm:ss.ttt  | set interval time
| Clear      | C            | clear measurement records
| Quit       | Q            | quit measurement recording
| Start      | S            | start measurement recording
+----------+------------+----------------------------------

Command: d
```
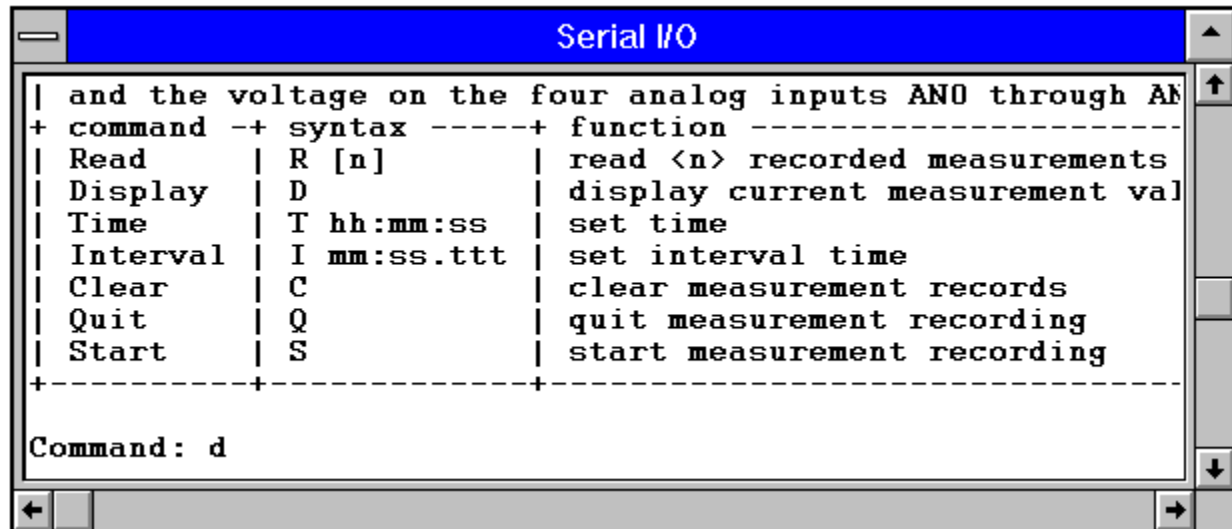
Like any other dScope window, the SERIAL window may be minimized and restored using the button on the top right of the window.   The window can be hidden using dScope's main menu, the toolbar, or with the **Close** command in the system menu of the SERIAL window.

see also:

SERIAL Window Commands

Scrolling the SERIAL Window

SERIAL Window Commands

The SERIAL window can display the output in either ASCII format or as hex bytes.   The format is changed by a simple click with the right mouse button within the SERIAL window.   The command menu will then appear:

```
┌────────────────────────────── Serial I/O ─────────────────────────┬─┬─┐
│                                                                    │▲│
├────────────────────────────────────────────────────────────────┬──┼─┤
│| Display   | D            | display current measurement val     │  │↑│
│| Time      | T hh:mm:ss   | set time                            │  │ │
│| Interval  | I mm:ss.ttt  | set interval time                   │  │ │
│| Clear     | C            | clear measurement records           │  │ │
│| Quit      | Q            | quit measurement recording          │  │ │
│| Start     | S            | start measurement recording         │  │ │
│+---------+------------+----------------------------------------  │  │ │
│                                                                  │  │ │
│Command: |  ┌─────────────────────┐                              │  │↓│
├──────────│  Hex output          │───────────────────────────────┼──┼─┤
│←│        │√ Ascii output        │                             │→│
│ │        ├─────────────────────┤                             │ │
│         │  Clear Window        │
│         └─────────────────────┘
```
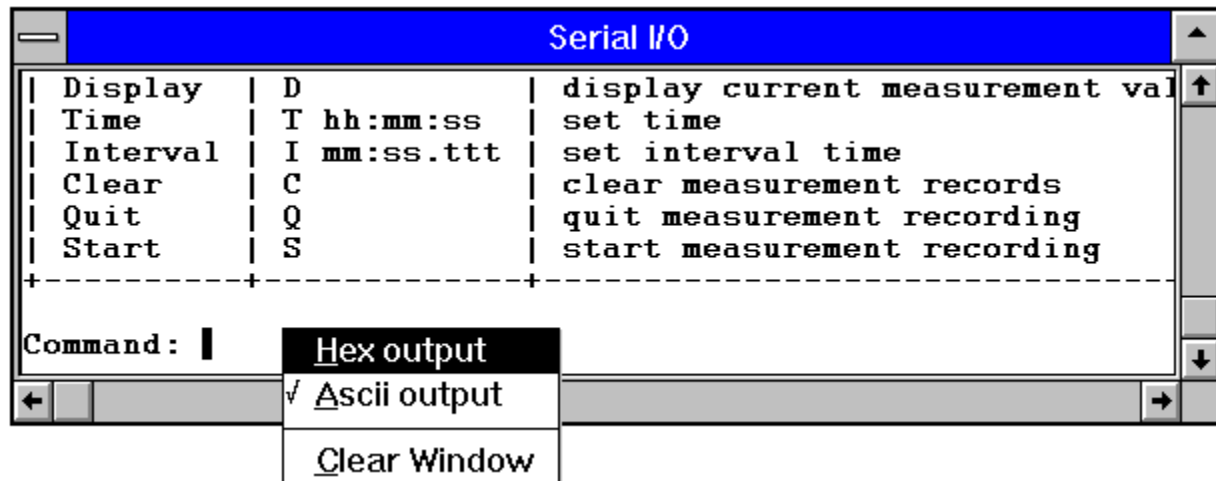
Select the requested mode by a click with the left mouse button.   The text currently displayed in the SERIAL window is not affected by the mode change.   The future serial output will by displayed using the new mode.   The contents of the SERIAL window may be cleared by selecting the **Clear Window** command.   Note that the serial output mode does not affect the characters entered in the SERIAL window.   If a character is entered, it's hex value (a byte) becomes available in the serial input buffer.

The SERIAL window maintains a history buffer capable of holding the most recent 8K of serial data.   A single line may be up to 128 characters in length.   If a line containing more than 128 characters is encountered, dScope will break the line by inserting a new line character.

Scrolling the SERIAL Window

The SERIAL window maintains a history buffer capable of holding the most recent 8K of serial data.   A single line may be up to 128 characters in length.   If a line containing more than 128 characters is encountered, dScope will break the line by inserting a new line character.   The SERIAL window can be scrolled vertically or horizontally using either the vertical and horizontal scrollbar or one of the keys listed in the following table:

PgDn	Scroll window contents one page downwards.
PgUp	Scroll window contents one page upwards.
↓	Scroll window contents one line downwards.
	Scroll window contents one line upwards.
→	Scroll window contents one characters to the right.
←	Scroll window contents one characters to the left.

NOTE If the vertical boundaries of the history buffer are reached, dScope responds with a keyboard beep reminder.

The Performance Analyzer Window

The PERFORMANCE ANALYZER window (or PA for short) displays the results of the execution time recorded for each defined function or address range.   The results are displayed as bar graphs.   Also, more range information such as invocation count or maximum execution time for a selected range is displayed.   The PA supports   up to 256 address ranges.   The PA window contains a ruler for quick percentage overview, a status bar with help for the PA menu commands, and a bar for display of the times:

Performance Analyzer
Commands
0% 10 20 30 40 50 60 70 80 90 100%
<unspecified>:
main:
timer0:
clear_records:
measure_display:
save_current_measurements:
read_index:

min time:	max time:	avg time:	total time:	%	count:
0.000033	0.000262	0.000086	0.921753	34.3	10750
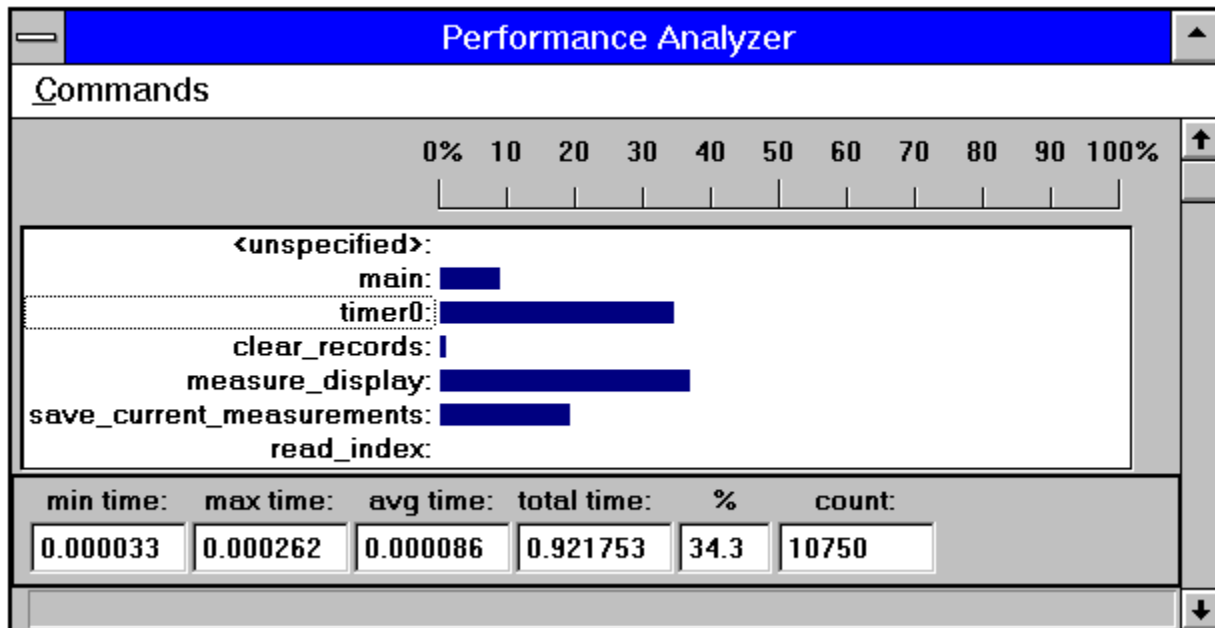
The client part of the PA window contains the starting address (here in symbolic form) of each address range.   The bars following the symbols give an overview of the percentage of execution time.   The first line is predefined by dScope and is always present and cannot be removed. It represents a container which receives the execution time not consumed by any defined address range.

A specific line can be selected by clicking at the desired line.   This line then becomes the selected address range, which the additional time information on the bottom of the PA window is shown for:

min time	Is the minimum time an invocation of the range took.
max time	Is the maximum time an invocation of the range took.
avg time	Displays the average time each invocation of the range took.
total time	Gives the total execution time in seconds consumed by the given address

	range.
%	Gives the percentage value of the total execution time consumed by the given address range.
count	Displays the invocation count.

You can display the timing information for any other range by simply selecting another range. For the predefined **<unspecified>** range, only the total time and percentage field contain information, all other fields are blank.

see also:

The Performance Analyzer Command menu

The PA window contains a menu with the following commands:

- ◆ Reset PA - Resets the performance analyzer by clearing the recorded time and invocation information of any currently defined range.

- ◆ Activate PA - Enables or disables the time recording.   If disabled, the display will be frozen since time gathering for all address ranges is turned off.

- ◆ Show times - Show or hide the additional time information fields.

Scrolling the Performance Analyzer Window

The PA window can be scrolled vertically by either the vertical scrollbar or one of the keys listed in the following table:

PgDn	Scroll window contents one page downwards.
PgUp	Scroll window contents one page upwards.
↓	Scroll window contents one line downwards.
	Scroll window contents one line upwards.

NOTE If the vertical boundaries are reached, dScope responds with a keyboard beep reminder.

Defining a PA Address range

In order to enable performance analysis, PA ranges have to be defined first.   A PA range is an address range with a *unique*   entry and exit point.   Such a range can be defined either using the PA command in the Command window or   'The Performance Analyzer Setup Dialog':
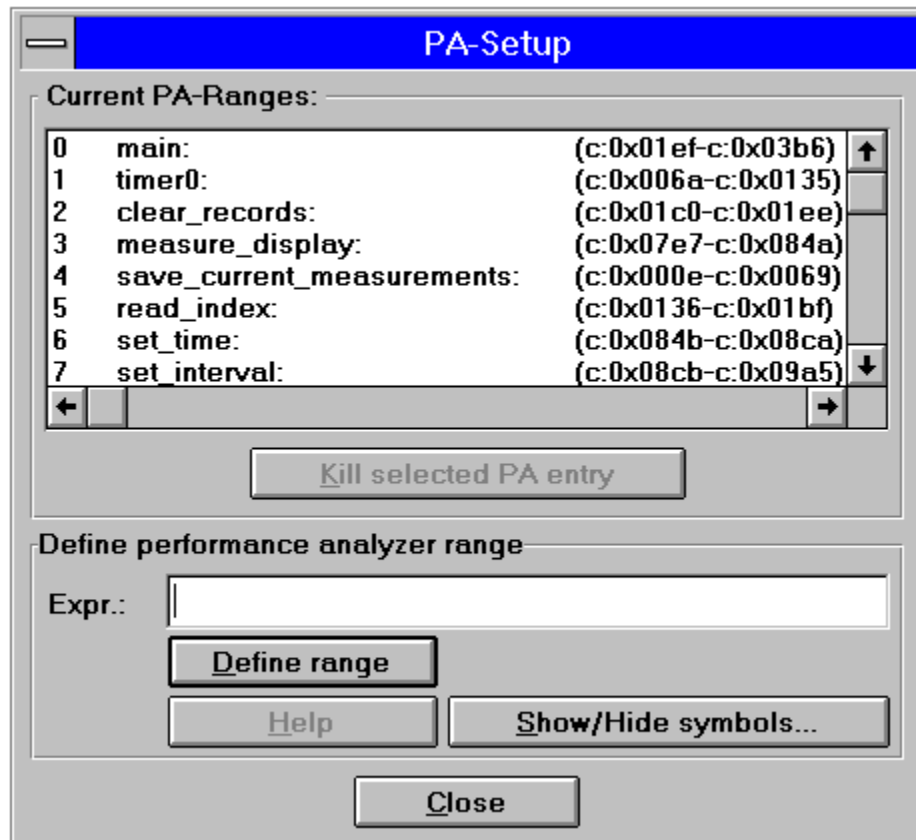
```
PA-Setup
┌─Current PA-Ranges:────────────────────────────────┐
│ 0    main:                        (c:0x01ef-c:0x03b6)  ↑ │
│ 1    timer0:                      (c:0x006a-c:0x0135)    │
│ 2    clear_records:               (c:0x01c0-c:0x01ee)    │
│ 3    measure_display:             (c:0x07e7-c:0x084a)    │
│ 4    save_current_measurements:   (c:0x000e-c:0x0069)    │
│ 5    read_index:                  (c:0x0136-c:0x01bf)    │
│ 6    set_time:                    (c:0x084b-c:0x08ca)    │
│ 7    set_interval:                (c:0x08cb-c:0x09a5)  ↓ │
│ ←  ▱                           →  │
│           Kill selected PA entry          │
└───────────────────────────────────────────────────┘
┌─Define performance analyzer range─────────────────┐
│ Expr.:   │                                    │      │
│          │ Define range │                            │
│          │    Help    │  │ Show/Hide symbols... │     │
└───────────────────────────────────────────────────┘
              │    Close    │
```

In the *Expression*   field, enter the range.   A range can be a function name of the user program or two numeric expressions which represent the starting and ending address of the address range.   If only a function name is given, dScope will derive the ending address using the high level debug information loaded with the users program.   If such debug information is not available, then an error message will be displayed.   When the entry in the *Expression*   field is finished, hit the Enter key or click the **Define range** button to add the range to the list of existing ranges.

To ease the entry of symbols, you may select the **Show symbols** button which opens the Symbol browser window (if not already open).   From the Symbol browser, you can drag a symbol   into the *Expression*   field as follows:

- ◆ Move the mouse pointer to the requested symbol.
- ◆ Press the left mouse button, and don't release the button (the cursor changes).

- Move the mouse pointer to the *Expression* field of the **Watchpoints** dialog

- Release the mouse button. The qualified symbols name is then filled in.

NOTE PA range definitons can alternatively by defined and removed with the PA command in the command window.   Refer to dScope Commands for more information on PA commands.

Removing a PA Address range

From the list of defined PA ranges, select the desired line.   The **Kill selected PA entry** button will get enabled, click it.   The range is immediately removed both from the list and internally from the list of active PA ranges.

Identifying valid Address Ranges

A valid PA address range must have a unique entry and exit point.   Within dScope, you have two choices on how to find out the address ranges currently available.   The first choice is to use the **Scope** command in the command window.   This commands displays address range information about all currently defined blocks.   The following example shows part of the output created by the **Scope** command with the MEASURE sample application loaded into dScope:

```
MEASURE
  {CvtB} RANGE: 0xFF03B7-0xFF07E5
  {CvtB} RANGE: 0xFF000B-0xFF000D
  SAVE_CURRENT_MEASUREMENTS RANGE: 0xFF000E-0xFF0069  /* valid */
  TIMER0 RANGE: 0xFF006A-0xFF0135                     /* valid */
  _READ_INDEX RANGE: 0xFF0136-0xFF01BF               /* valid */
  CLEAR_RECORDS RANGE: 0xFF01C0-0xFF01EE             /* valid */
  MAIN RANGE: 0xFF01EF-0xFF03B6                       /* valid */
MCOMMAND
  {CvtB} RANGE: 0xFF09A6-0xFF0A23
  MEASURE_DISPLAY RANGE: 0xFF07E7-0xFF084A           /* valid */
  _SET_TIME RANGE: 0xFF084B-0xFF08CA                 /* valid */
  _SET_INTERVAL RANGE: 0xFF08CB-0xFF09A5             /* valid */
GETLINE
  _GETLINE RANGE: 0xFF0A24-0xFF0A87                  /* valid */
?C_FPADD
?C_FPMUL
```

The indented identifiers represent functions and their address range in memory.   You can use all ranges not starting with '**{**' and with the starting and ending address present.   The ranges named **{CvtB}** can't be used for performance analysis.   Such ranges result from functions which do not have sufficient debug information, usually code from libraries or modules which were not compiled for debug.

The second choice is to use the Symbol browser window to view or select a function for a PA range.   Choose the **Publics** or **Locals** mode, check **Options** and turn all memory spaces filters except code off.   With the MEASURE application loaded, the Symbol browser window will show something like this:
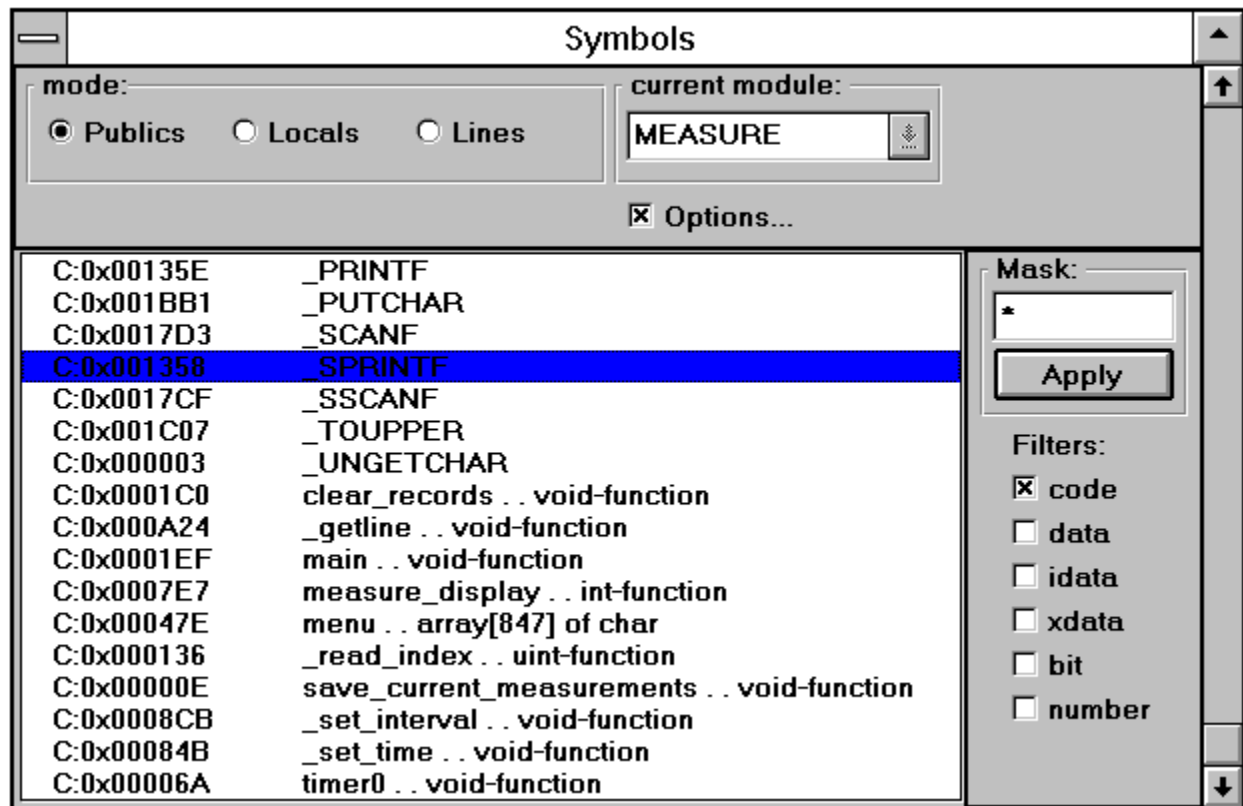
The client area shows some symbols with ' **... function** ' appended.   These symbols identify function symbols which may be dragged into the PA-Setup dialog for definition of PA range without having to specify the functions end address.

The Memory Window

The MEMORY window is used to display memory areas in both hexadecimal and ASCII format. The range to be displayed must not cross 64K boundaries.

```
┌────────────────────────────── Memory ──────────────────────────────┐ ▲
│ Addr   00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 0123456789ABCDEF │ ↑
│ X:4000: 00 00 09 03 59 FF FF 00 00 00 00 00 00 09 03 5A ....Y.........Z │
│ X:4010: FF FF 00 00 00 00 00 00 09 03 5B FF FF 00 00 00 ..........[..... │
│ X:4020: 00 00 00 09 03 5C FF FF 00 00 00 00 00 00 09 03 .....\.......... │
│ X:4030: 5D FF FF 00 00 00 00 00 00 09 03 5E FF FF 00 00 ]..........^.... │
│ X:4040: 00 00 00 00 09 03 5F FF FF 00 00 00 00 00 00 09 ......_......... │
│ X:4050: 03 60 FF FF 00 00 00 00 00 00 09 03 61 FF FF 00 .`..........a... │
│ X:4060: 00 00 00 00 00 09 03 62 FF FF 00 00 00 00 00 00 .......b........ │
│ X:4070: 09 03 63 FF FF 00 00 00 00 00 00 09 03 64 FF FF ..c..........d.. │
│ X:4080: 00 00 00 00 00 00 09 03 65 FF FF 00 00 00 00 00 ........e....... │ ↓
└─────────────────────────────────────────────────────────────────────┘
```

The address range to be displayed is defined with the **Display memory** command **D**. This command is to be entered in the Command window, as shown in the following examples:

```
>D X:0x00,X:0xffff /* 64K of xdata memory */
>D I:0x00,I:0xFF   /* the internal data memory */
>D current         /* 256 bytes starting from &current */
>D save_record     /* 256 bytes starting at &save_record */
```

The Display memory command normally takes 2 parameters: the first representing the display start address and the second giving the display end address.   If the second parameter is missing from the command, then   the number of bytes shown will be either 256 or the remaining bytes in the given memory space up to the upper limit of that memory space (0xFF on Idata, for example).

If the Display command is given and the MEMORY window is hidden, then the output will be directed to the COMMAND window rather than the MEMORY window.

NOTE dScope will truncate the address range for display to be within one 64K segment.   If for example a range from 0x000000 to 0x1FFFF is given, dScope will truncate the range to 0x00000 to 0x0FFFF.

see also:

Scrolling the Memory Window

Memory Window update

Scrolling the Memory Window

The MEMORY window can be scrolled vertically or horizontally by either using the scrollbar or one of the keys listed in the following table:

PgDn	Scroll window contents one page downwards.
PgUp	Scroll window contents one page upwards.
Cursor down	Scroll window contents one line downwards.
Cursor up	Scroll window contents one line upwards.

NOTE If the address boundaries are reached, dScope responds with a keyboard beep reminder in this case.

Memory Window update

The contents of the MEMORY window are updated each time the execution is stopped, for example after a Single-Step, StepOut or if the Go command is stopped.   Also, the window gets updated if it is selected (by clicking somewhere within the MEMORY window, for example) or by double clicking the left mouse button somewhere within the MEMORY window.

You can also have the MEMORY window being updated periodically while execution is running. This feature can be activated by selecting   **Setup - Update Memory window**   in the dScopes main menu.

The Symbol Browser Window

The Symbol Browser window displays the currently defined symbols or line numbers of the loaded user program and the CPU specific symbols introduced by the CPU driver. This includes public symbols as well as symbols locally defined in any functions.

Symbols
mode: ○ Publics (selected) ○ Locals ○ Lines   **current module:** MEASURE
☒ Options...

Address	Symbol		Mask:
D:0x000036	intcycle . . uchar		*
D:0x000037	interval . . struct interval		**[Apply]**
C:0x0001EF	main . . void-function		
B:0x000109	mdisplay . . bit		**Filters:**
C:0x0007E7	measure_display . . int-function		☒ code
B:0x00010A	measurement_interval . . bit		☒ data
C:0x00047E	menu . . array[847] of char		☒ idata
C:0x000136	_read_index . . uint-function		☒ xdata
C:0x00000E	save_current_measurements . . void-function		☒ bit
X:0x004000	save_record . . array[744] of struct mrec		☒ number
D:0x000034	savefirst . . uint		
C:0x0008CB	_set_interval . . void-function		
C:0x00084B	_set_time . . void-function		
D:0x000030	setinterval . . struct interval		
D:0x00002E	sindex . . uint		
B:0x000108	startflag . . bit		
C:0x00006A	timer0 . . void-function		

The Symbol Browser window contains different controls for symbol qualification:

◆ Mode selection:

Client mode selection.   The mode can be **Publics**, **Locals** or **Lines**:
Publics: All public symbols of the user program are shown.   Public symbols are those symbols which have application wide scope rather than module or function scope.   The CPU specific symbols such as SFR names and bit names are also considered public symbols.

Locals: Shows the local symbols of the functions from the current module. The current module is selected with the current module   combo box.   The locally defined symbols in each function are indent on the display.

Lines: Forces the line number and associated addresses from the current module to be displayed.   The current module is selected with the current module   combo box.

◆ Current module:

The **current module** combo box is used to select the current module.   The combo box is enabled only if the mode selection is **Locals**.   The combo box contains the names of all modules.   Note that some modules may not contain local symbols or line information, especially on modules which were not compiled for with the debug compiler option or on library modules.

◆ Options:

Check this button to show the dialog bar containing the memory space filters and the symbol search qualification.

Filters - With the memory space filters, the symbols being displayed can be restricted to symbols which match the selected memory spaces.   The memory space filters are extended on MCS 251, since there are additional memory spaces, for example **edata** or **ebit**.   The memory space filters are disabled in **Lines** mode, since line number corresponds to code space.

Mask - With the **Mask** control, you can select specific symbols to be displayed by means of very simple regular expressions.

see also:
Searching for specific symbol names
Scrolling the Symbol Browser Window
Dragging Symbols

Searching for specific symbol names

The symbol output may be restricted to symbols with specific names by entering a name mask into the **Mask** field in the dialog bar and then selecting the **<u>Apply</u>** button to start the window rebuild.   A name mask can consist of any alphanumeric characters with the special function mask characters explained below.   The **Mask** field applies to **Publics** and **Locals** mode only, it is *disabled* in **Lines** mode.

Special mask characters:

        #   matches a digit (0...9)
        $ matches any character
        *   matches zero, one or more character occurrences.

Name mask examples:

*	Matches any symbol. This is the default mask in the Symbol Browser.
#	Matches any symbol. This is the default mask in the Symbol Browser.
*##	Matches any symbol which ends with two adjacent digits.
_a$#*	Matches any symbol starting with an underline character followed by the lowercase letter 'a' followed by any character followed by a digit followed by anything or nothing. For example, _ab1 or _a10value would match here.
_*ABC*#	Matches any symbol that starts with the an underline character, followed by zero or more characters up to the string ABC, followed by zero or more characters and finally ends with a digit.

The alpha characters of a search mask are treated case sensitive.   Note that a search mask cannot have two adjacent star characters.   Besides taking the search mask for selecting symbol names, the memory space (specified using the Options dialog bar) is also taken into account for matching symbols.

Scrolling the Symbol Browser Window

The window contents can be scrolled vertically by either the scrollbar or one of the keys listed in the following table:

PgDn	Scroll window contents one page downwards.
PgUp	Scroll window contents one page upwards.
Cursor down	Scroll window contents one line downwards.
Cursor up	Scroll window contents one line upwards.

NOTE If the vertical boundaries are reached, dScope responds with a keyboard beep reminder.

Dragging Symbols

The Symbol Browser window supports Drag & Drop of symbols to the Command window or any dialog which requires numeric expressions for input.   Such dialogs are the inline assembler dialog, the watchpoint dialog and others.   To move a symbol   into some other context, simple select the desired symbol by clicking on it with the left mouse button (don't release the button) and move it to the target location.   Watch the cursor, it will change on areas where drop is not supported and again change on areas where symbol drop is supported.   Release the mouse button, this will insert the symbolic name.   If a local symbol is dragged to some window or edit control, then the fully qualified name will be inserted, for example:

dragged symbol:	**buffer** in block _**SET_TIME**
result:	**\mcommand_set_time\buffer**
dragged line:	line **#90** of module MEASURE
result:	**\measure\90**

A fully qualified local symbol consists of a module name, a function name and the local symbol name.   The components in a qualified name are delimited by a backslash character.   If a line number is dragged, the fully qualified line will consist of the module name and the line number. Note that line numbers are associated to a module, not to functions.

The Call Stack Window

The Call Stack window displays the list of currently nested function calls or interrupt procedures being executed.   Each line in the display starts with a number indicating the nesting followed by the numeric address of the invoked function and the symbolic name of the function, if available:

```
Call-Stack

4:    C:0x112B    ?C_COPY
3:    C:0x006A    timer0
2:    C:0x135E    _PRINTF
1:    C:0x07E7    measure_display
0:    C:0x01EF    main

        Refresh          Show invocation

Invocation:

\measure\timer0

             Close
```
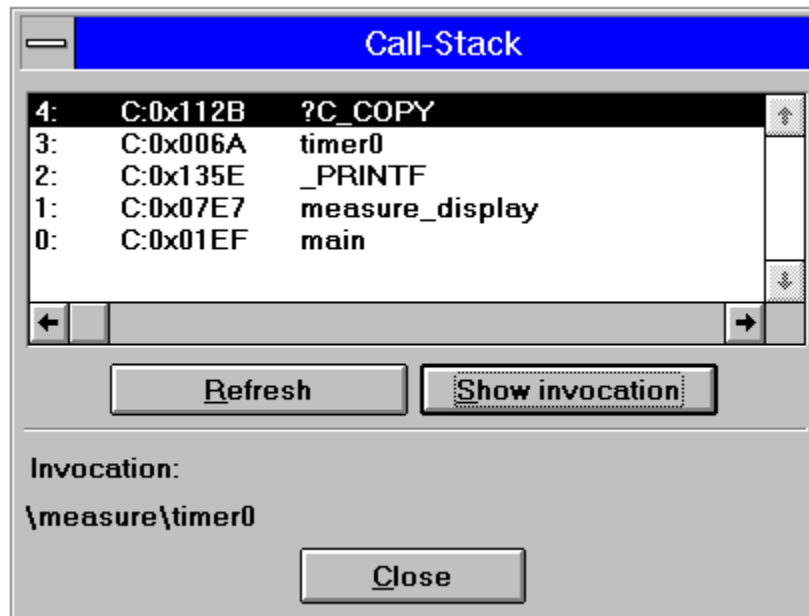
The Call Stack window can be left open at any time.   The **Refresh** button forces the display being updated to reflect the current function nesting which changes as the user program is executed.   The window also shows the caller of the selected line near to the bottom of the window.   The selected line designates the highlighted line in the list box.

The **Show invocation** button forces the DEBUG window to show the code where the function was invoked.   Depending on the view mode of the DEBUG window, the invocation code can be viewed in high level, mixed or assembly mode.

The Code Coverage Window

The Code Coverage window contains information to find out about executed and non executed areas of the user program.   With the code coverage feature of dScope, unexecuted code can be discovered.   Such areas of code may exist in case of programming errors or simply dead code which could be eliminated.   When the user program is executed by any execution command such as **StepInto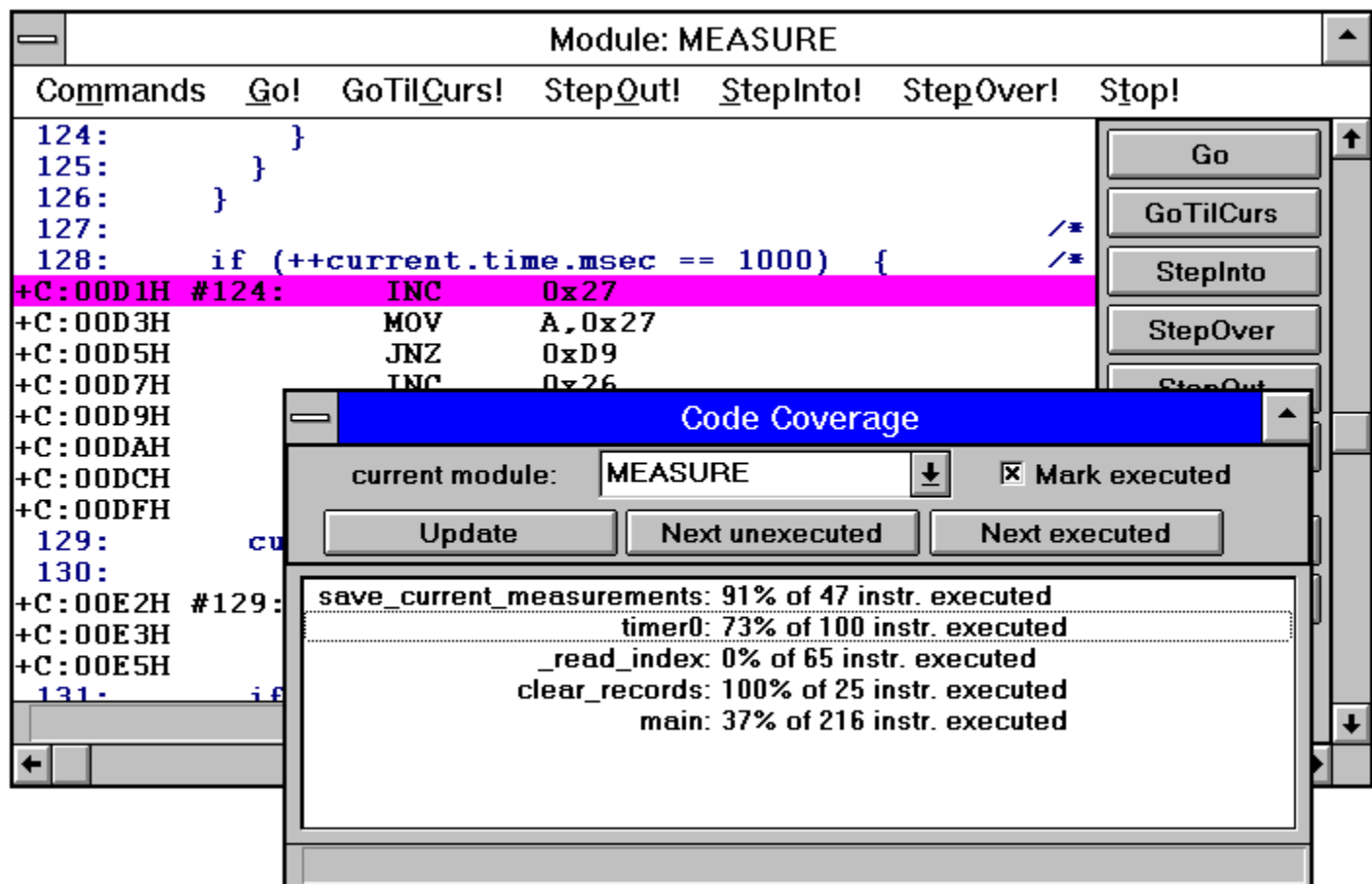** or **Go**, dScope marks every code address where an instruction was executed.   The Code Coverage window displays this type of information in a module based manner:

```
┌──────────────────────────────────────────────────────────────┐
│ �一            Code Coverage                                ▲  │
├──────────────────────────────────────────────────────────────┤
│  current module:   │MEASURE            ↓│  ☒ Mark executed    │
│  ┌─────────────────┬───────────────────┬──────────────────┐  │
│  │     Update      │  Next unexecuted  │  Next executed    │  │
│  └─────────────────┴───────────────────┴──────────────────┘  │
│ ┌──────────────────────────────────────────────────────────┐ │
│ │ save_current_measurements: 100% of 47 instr. executed    │ │
│ │            timer0: 78% of 100 instr. executed            │ │
│ │        _read_index: 0% of 65 instr. executed             │ │
│ │     clear_records: 100% of 25 instr. executed            │ │
│ │          main: 37% of 216 instr. executed                │ │
│ └──────────────────────────────────────────────────────────┘ │
│ ┌──────────────────────────────────────────────────────────┐ │
│ └──────────────────────────────────────────────────────────┘ │
└──────────────────────────────────────────────────────────────┘
```
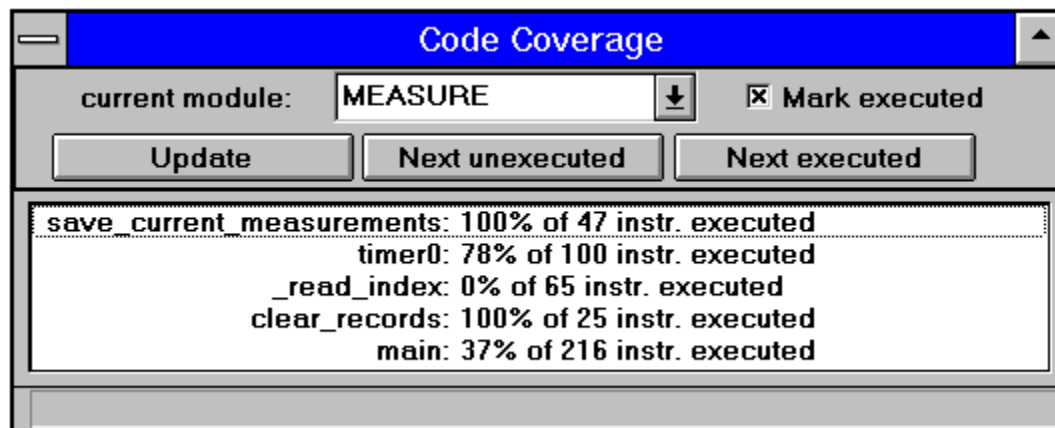
The client area contains the names of all functions which correspond to the selected module. The module can be changed by selecting a new one from the **current module** list box.   Each line starts with the name of a function followed by the percentage value of executed instructions out of the total number of instructions belonging to the function.   The **Update** button forces the percentage value to be updated when the user program is executing.   Selecting the **Next unexecuted** button will resynchronize the DEBUG window to the next instruction of the given function not already executed.   Selecting the **Next executed** button will resynchronize the DEBUG window to the next instruction of the given function already executed.   If next executed or non executed instruction fails for the given function, dScope will automatically do the search on the next function from the list of functions.

The **Mark executed** checkbox forces all executed lines in the DEBUG window marked with a plus sign:

```
─  Module: MEASURE                                          ▲

Commands   Go!   GoTilCurs!   StepOut!   StepInto!   StepOver!   Stop!

 124:           }                                                        ┌──────────┐  ↑
 125:         }                                                          │    Go    │
 126:       }                                                            └──────────┘
 127:                                                          /*        ┌──────────┐
 128:       if (++current.time.msec == 1000)   {               /*        │ GoTilCurs │
+C:00D1H #124:      INC        0x27                                      └──────────┘
+C:00D3H            MOV        A,0x27                                     ┌──────────┐
+C:00D5H            JNZ        0xD9                                       │ StepInto │
+C:00D7H            INC        0x26                                       └──────────┘
+C:00D9H                                                                  ┌──────────┐
+C:00DAH                                                                  │ StepOver │
+C:00DCH          ┌──────────────────────────────────────────────┐      └──────────┘
+C:00DFH          │ ─          Code Coverage                    ▲ │      ┌──────────┐
 129:        cu   │                                               │      │ StepOut  │
 130:             │  current module:  ┌MEASURE────────┐ ↓  ⊠ Mark executed │
+C:00E2H #129:    │                   └───────────────┘            │
+C:00E3H          │  ┌──────Update──────┐┌─Next unexecuted─┐┌─Next executed─┐│
+C:00E5H          │  └──────────────────┘└─────────────────┘└───────────────┘│
 131·        if   │ ┌────────────────────────────────────────────┐ │      ↓
                  │ │save_current_measurements: 91% of 47 instr. executed│
 ←┌─┐             │ │            timer0: 73% of 100 instr. executed      │
  └─┘             │ │     _read_index: 0% of 65 instr. executed          │
                  │ │  clear_records: 100% of 25 instr. executed         │
                  │ │        main: 37% of 216 instr. executed            │
                  │ │                                                    │
                  │ └────────────────────────────────────────────────────┘
                  └──────────────────────────────────────────────────────┘
```

In the previous screen example, the **Next executed** button has been pressed a few times, synchronizing the DEBUG window to the next executed instruction and highlighting that line. Repeated selection of Next (un)executed commands will walk to the list of functions and start all over again with the first function when the end of the function list is reached.

see also:

Scrolling the Code Coverage Window

Scrolling the Code Coverage Window

If the Code Coverage window contains more function entry lines than can fit on the client area of the window, the vertical scroll bar will be visible, otherwise it gets removed.   The window contents can be scrolled by either the vertical scrollbar or one of the keys listed in the following table:

PgDn   Scroll window contents one page downwards.

PgUp   Scroll window contents one page upwards.

Cursor down   Scroll window contents one line downwards.

Cursor up   Scroll window contents one line upwards.

NOTE If the vertical boundaries are reached, dScope responds with a keyboard beep reminder.

The Toolbox Window

The Toolbox window is a user configurable, modeless dialog.   It may contain up to 16 command buttons where the first command **Reset** is predefined and can't be removed.   The button command gets executed when a button is pressed.   This can be done at any time including when the user program is executed.   Also, commands can also be assigned to function keys F1 ... F12.   Pressing a function key then has the same effect as pressing a Toolbox button.   Refer to **dScope Command SET** for details.

Note: The function key F10 is not available for assigning a command to it.

see also:

Creating a Toolbox Button

Removing a Toolbox Button

Creating a Toolbox Button

A command button is defined using the Define Button command in the COMMAND window. The general syntax is:

DEFINE BUTTON "button_label", "button_command"

Both parameters to the command are required to be C style strings.   The first parameter "button_label" defines the name displayed as the button label.   The second parameter "button_command"   must be a valid dScope command which is executed when the button is pressed.   The following examples show the define commands used to create the buttons shown in the Toolbox picture:

```
>Define Button "clr dptr", "dptr=0"
>Define Button "show main()", "u main"
>Define Button "show r7", "printf (\"R7=%02XH\n\",R7)"
```

NOTE the second parameter to the last example reading - **printf   (\"R7=%02XH\n\",R7)"** introduces nested strings.   Since dScope's printf command requires a format string for it's first parameter and the whole command must be string, strings are getting nested.   The double quote characters of the nested string must be escaped - \" - in order to avoid   syntax errors.

Every time a syntactically correct Define Button command has been entered, the button is immediately added to the Toolbox, and the height of the Toolbox is automatically enlarged. Each button receives a dScope assigned button number which is displayed just before the button.   This number is the handle to identify buttons to be removed.

All button commands created are saved in the dScope's INI file on exit from dScope.   The next time dScope is invoked, the button commands are automatically restored and are available in the Toolbox again.

Removing a Toolbox Button

A Toolbox button is removed by entering the **KILL BUTTON** command in the COMMAND window.   The parameter to the command must be the dScope assigned button number:

```
>Kill Button 3  /* kill 'show r7' button */
>Kill Button 2  /* kill 'show main()' button */
```

NOTE The first Toolbox button, '**Reset**' is predefined an cannot be removed from the Toolbox.

The Breakpoint Dialog

The Breakpoint dialog assists in managing breakpoints. Breakpoints are used to halt a program run at a certain code addresses or under certain conditions.   dScope supports up to 40 simultaneously active breakpoints.

The following paragraphs describes the Breakpoint dialog and it's controls.
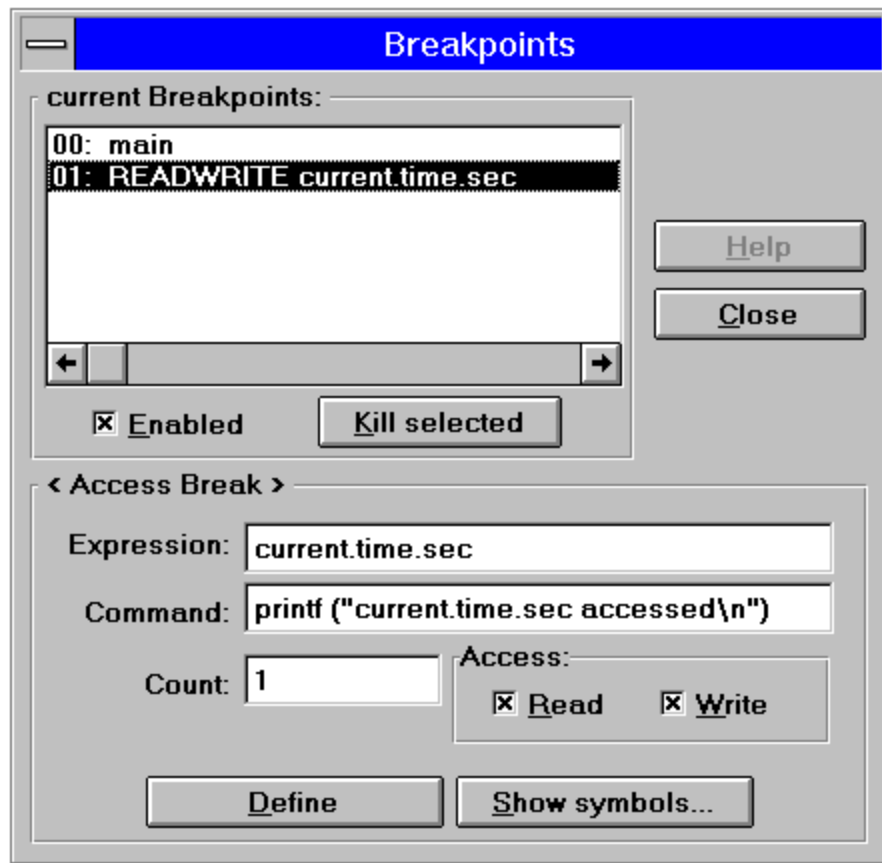
Description of the dialog controls:

- ◆ **Current Breakpoints listbox**

  The current breakpoints listbox shows the currently defined breakpoints.   If you select a line by a mouse click, then the specific information for that breakpoint is displayed using the controls located on the lower half of the dialog.   This is to view the details of the selected breakpoint.   The bounding rectangle of the entry fields will show the breakpoint class, if a breakpoint is selected.   The class may be **<Execution Break>,** <**Conditional Break**> or <**Access Break**>, see Breakpoint Classes for more information on this.

◆ **'Enabled' Checkbox**

This checkbox shows the state of the selected breakpoint from the list.   Select a line from the list of breakpoints.   The checkbox will then be checked, if the breakpoint is enabled, otherwise it is unchecked.   You can change the state of the breakpoint by checking or unchecking the control.   The selected breakpoint will be immediately enabled or disabled, depending on state of the checkbox.

◆ **Kill button**

The kill button is used for removal of the selected line from the list of breakpoints. If no line is selected from the list, then the kill button is disabled, otherwise it is enabled.

◆ **Expression**

The Expression field is used to enter the breakpoint expression, usually some symbol from the user program.   If a breakpoint is to be defined, the Expression field must contain an expression.

◆ **Command**

The Command field allows entry of a dScope command or an expression.   The command is executed, or the expression is calculated when the breakpoint is reached during execution of the user program.   Note that input to the Command field is optional, and can be omitted.   The program execution is NOT stopped when a command string exists.   Instead, the program execution is continued after the command string is executed.   If the program run is to be stopped, the variable _BREAK_ must be set to "1.

◆ **Count**

The Count field can be given a number which specifies the number of occurrences of the breakpoint during execution until the breakpoint actually stops execution.   Entry to the Count field is optional, if no value is given, it defaults to one.

◆ **Access: Read, Write**

The Access check buttons are used to create access breakpoints.   An access breakpoint must specify an address of some data item of the user program. During execution of the user program, dScope will catch, read, or write accesses to the specific address and decide whether to continue or stop execution.   This finally depends on the parameters given to Count or Command field.

◆ **Define button**

With the Define button, the definition of a breakpoint is initiated, if at least the Expression field contains some input.

◆ **Show symbols button**

The Show symbols button opens or closes the Symbol Browser window.   For simple entry of symbols into the Expression field, you can drag a symbol from the browser window into the entry fields.

- **Help button**

    Open WinHelp with on-line help for the breakpoint context

- **Close button**

    Close the dialog.   This can be done with the Close entry from the system menu also.   Closing the breakpoint dialog does not change any breakpoints.

see also:

Breakpoint Classes

Defining an Execution Breakpoint

Defining a Conditional Breakpoint

Defining an Access Breakpoint

Breakpoint Classes

dScope maintains three different classes of breakpoints, each class with specific advantages and disadvantages on utility and execution speed to meet the common situations when debugging a program.   The breakpoint classes and the specific features are:

- ◆ **Execution Breaks**

  An Execution breakpoint is simply an address where execution is stopped when the specified address is reached.   The execution speed is not affected when execution breaks are defined.   Execution breakpoints are the most simple. They can be used to check control flow of a program.
  The user must guarantee that the address specifies an opcode address (e.g. the address of the first byte of an MCS 51/251 instruction).   An execution breakpoint for a certain code address can only be specified once; multiple definitions are not permitted.

- ◆ **Conditional Breaks**

  A conditional (or complex) breakpoint is intended for those situations, where a problem cannot be simply described by an execution address or a simple overwrite of some location.   Given the situation 'variable **sindex** gets a bad value'.   If the source of the problem is not known, the potential problem may be in any function of the application.   In order to find the problem, we have to check the value of the variable '**sindex**' each time an assembler instruction of the user program has been executed.   Checking a variable for a certain numeric range introduces a lot of overhead and may slow down execution speed considerably. On the other hand, this may be the only way to catch fuzzy program errors.

- ◆ **Access Breaks**

  Access breakpoints are used to catch memory overwrites or accesses to illegal memory addresses.   An access break is created if either a READ or WRITE is selected.   The speed loss is only minimal, because the expression is evaluated when the specified access event occurs, only.

dScope will classify the breakpoint expression automatically, that is, if the access specifiers Read/Write are present.   Otherwise analysis of the expression will decide whether an Execution breakpoint or Conditional breakpoint was given.   If the expression specifies a code address such as an address of a function or a line number, then it is an Execution breakpoint, otherwise it is a Conditional breakpoint.

Defining an Execution Breakpoint

- ◆ In the Expression field, enter the name of a function or a qualified line number.
- ◆ In the Command field, enter a command to be executed if the breakpoint becomes active (optional)
- ◆ In the Count field, enter the number of times the address must be reached in order for the breakpoint to become active.   This is optional, no entry means one.
- ◆ Select the Define button.   The breakpoint gets defined and displayed in the list of breakpoints and all fields are cleared.

Make sure, that the Access controls *Read* and **Write** are unchecked. Otherwise an access breakpoint that doesn't make sense would be created.

Example-1:

Expression:	**main**
Command:	**printf ("main has been reached\n")**
Count:	**1**

This example will stop execution if the address of 'main()' is reached the first time.   Since a command was entered also, it will get executed.   The output of the command is directed to the COMMAND window.

Example-2:

Expression:	**\MEASURE\110**
Command:	
Count:	**1000**

This example will stop execution if the address of the line number 110 in module MEASURE has been executed 1000 times.   Since no command was entered, none is executed.   Note after the breakpoint has been active, the count value will show 1.

Defining a Conditional Breakpoint

- ◆ In the Expression field, enter any expression.   The expression should describe the event to occur for the breakpoint to become active.
- ◆ In the Command field, enter a command to be executed if the breakpoint becomes active (optional)
- ◆ In the Count field, enter the number of times the event must occur for the breakpoint to become active   (optional).
- ◆ Select the Define button.   The breakpoint gets defined and displayed in the list of breakpoints and all fields are cleared.

Make sure that the Access controls *Read* and **Write** are unchecked. Otherwise an access breakpoint will be created.

Example-1:

Expression:	**(sindex > 0x0A && sindex < 0x25) || sindex == 0x133**
Command:	**eval sindex**
Count:	**1**

This example will stop execution the value of the variable sindex is in range 0x0B to 0x24 or has the value 0x133.   The command 'eval sindex' will then be executed, evaluating the value of sindex and displaying the result to the COMMAND window in different number bases.

Example-2:

Expression:	**$ == timer0 && sindex > 5**
Command:	
Count:	**1000**

This example will stop execution if execution enters function 'timer0' ($ represents the current program counter) and at the same time, the variable 'sindex' contains a value greater than 5.

WARNING

Avoid assignments to the current program counter, such as $=timer0, within a breakpoint expression, since dScope repeatedly redraws the DEBUG window on changes of $.   dScope will react considerably delayed to commands and button selections.   If you run into this situation by accident, then change into the COMMAND window and enter BK *, which removes all breakpoints.   Then hit the Stop button in the DEBUG window.

Defining an Access Breakpoint

◆ In the Expression field, enter an expression which fits the following.

The expression must represent a memory address and memory type.   This is the case, if for example the expression is the name of a scalar.   Some extensions that behave according to the following rules are allowed:

1. The result of the expression must have a unique memory type.   This means that only one name of an object may occur.
2. Only the operators &, &&, <, <=, >, >=, ==, and != are permitted.   An expression according to rule 1 must exist to the right of one of these operators.   An expression the left of the operator can be of any type or complexity, rule 1 does not apply to an expression right to the operator.

If an expression conforming to rule 2) is entered, dScope uses the left of the operator for the memory address and space.   If on execution an access to that address is detected, dScope evaluates the expression right to the operator and break, if the result is TRUE (assuming Count = 1).

◆ In the Command field, enter a command to be executed if the breakpoint becomes active (optional)

◆ In the Count field, enter the number of times the event must occur for the breakpoint to become active   (optional).

◆ Select the access specification, Read or Write or both, depending on the type of access you want to catch to the given memory object.

◆ Select the Define button.   The breakpoint gets defined and displayed in the list of breakpoints and all fields are cleared.

Example-1:

We want to break execution if the variable 'sindex', which is of type 'int',   is written to and the value of sindex equals 0x133.   The required entries are:

Expression: **sindex && sindex == 0x133**
Command:
Access*:* **Write**
Count: **1**

Although the entries are looking good, the breakpoint will probably never become active.   The reason for this is that the variable '**sindex**' has type '**int**', which means it uses two bytes in memory.   The high order byte is at some memory address followed by the low order byte at the next consecutive address.   The break expression specifies '**sindex && ..**', which specifies that accesses to the high order byte are catched.   It should be noted that addresses for access breakpoints should specify the low order byte of a scalar.   In our example, that would be **'&sindex+1'** for a 16-bit variable such as int/short or unsigned int/short ('&sindex+3' would be the correct address for sindex assuming to be a long or

float variable, for example).   This does not apply to char/unsigned char or byte scalars, since they use one byte of memory only..

Corrected Example-1:
Expression: **&sindex+1   &&   sindex == 0x133**
Command:
Access*:*   **Write**
Count: **1**

The expression **'&sindex + 1'** to the left of the **&&** operator now specifies the low order byte of 'sindex'.   The idea behind all of this is, that the low byte gets written to in any case, whereas the high byte might get written to only if an overflow on the low byte occurs.   In any case, the example will break if the write access takes place and the value 0x133 matches with no events lost.

NOTE:

Unlike C expressions, where   &sindex + n means address of 'sindex' plus n times the size of 'sindex', dScope's address expressions do not scale the offset in this case.   That means, &sindex+1 always targets the address of 'sindex' plus one byte, regardless of the type of the variable.   Using the C rules of scaling, it would not be possible to address part of a memory object.

Example-2:

We want to break execution if the structure '**curren**t' is written to.   Since '**current**' contains 11 bytes, the method used in Example-1 does not address the need for this example. One opportunity would be setting an access breakpoint on each byte address of the structure.   This makes sense on small structures only. On bigger ones, you would have to set dozens of breakpoints which is clearly a bad choice.   It turns out that we cannot address the requirement with access breakpoints either, so we have to choose another opportunity for this case, the MAP command.   With the MAP command we can map memory ranges with specific access permissions.   The following example shows the appropriate map command:

MAP &current, &current + sizeof (current) - 1 READ

The whole structure 'current' is mapped for READ access, that is, if any write to the structure takes place, then dScope will stop execution because of the access mode violation.   In the COMMAND window, and error message notifying the violation will also be displayed.

The Watchpoint Dialog

The Watchpoint dialog assists in managing watchpoint expressions. Watchpoints are used to show the content of a variable or a whole structure or and array in the WATCH window.   The WATCH window is updated each time execution of the user program is stopped but can also be updated periodically while execution runs, by selecting the command '**Update Watch window'** from dScope's main menu.

The following paragraphs contains the description of the Watchpoint dialog and it's controls.

Description of the dialog controls:

- ◆ Current watch definitions listbox

    The current watch definitions listbox shows the currently defined watch expressions.   If you select a line by a mouse click, then the specific information for that watch is displayed using the controls located on the lower half of the dialog.   This is to view the details of the selected watch expression.

- ◆ Kill selected button

The kill button is used for removal of the selected line from the list of watch expressions.   If no line is selected from the list, then the kill button is disabled, otherwise it is enabled.

◆ Expr.

The Expression field is used to enter the watch expression.   If a watch is to be defined, the Expression field must contain an expression.

◆ Define watch button

If the Expression feild contains some input, the definition of a watch is initiaited with the Define button.

◆ Show symbols button

The Show symbols button opens or closes the Symbol Browser window.   For simple entry of symbols into the Expression field, you can drag a symbol from the browser window into the entry fields.

◆ Number output base: 10, 0xnn

Determines the number base used for displaying byte, word, dword or pointer values.   The output number base can be either decimal (10) or hex (0xnn).   The number base does not apply to float values which are always displayed using the float output format.

◆ Output line Mode: Single, Multiple

Decides between single line and multiline modeFor single scalars, the output mode is irrelevant since scalars are displayed in single line mode anyway, regardless of the mode selection.   For aggregate types such as arrays, structures, or unions, however, the output line mode comes into play.   In single line mode the content of the whole aggregate is displayed into one line.   Since a single line is limited to 128 characters, the output may get truncated to 128 characters, therefore loosing part of the output.   In multiline mode however, each component of the aggregate is displayed in a separate line.

◆ Help button

Open WinHelp with online help for the watchpoint context

◆ Close button

Close the dialog, this can be done with the Close entry from the system menu also.   Closing the dialog does affect the currently defined watch expressions.

The list of watch expressions starts with a dScope assigned number starting from zero.   This number is the watch number.   It is the handle to reference a specific watch for use in the command line form of the watch commands in the COMMAND window.

NOTE avoid defining huge arrays of structures with thousands of members since the overhead added to process this amount of information periodically will considerably slow down the execution speed of dScope.   Decide which parts of such an array are really of interest to you and qualify them.   For example, in the MEASURE application, save_record[] is an array of structures resulting in thousands of lines being output to the WATCH window.   It is very unlikely that such an amount of output is really valuable.   Instead, select the structure or structures out of the array and enter 'save_records[4]' for example instead of 'save_records' which selectes the whole array.

see also:

Defining   struct/union/array type Watch Expressions

Defining Pointer type Watch Expressions

Defining struct/union/array type Watch Expressions

In the Expression field, enter the name of the aggregate typed object.

Examples:
```
current                  /* a struc */
current.time             /* a struc nested within 'current' */
save_record[sindex].time  /* a struc out of struc array 'save_record' */
```

Defining Pointer type Watch Expressions

In the Expression field, the name of a possibly dereferenced pointer name should be entered. To illustrate the difference between dereferenced and unreferenced pointer, assume the given declarations:

```
struct node  {                  /* a struct */
  struct node      *next;       /* a ptr to a struct */
  unsigned char       op;
  unsigned char     type;
};
struct node nodes[10];          /* an array of [10] struct node */
struct node pN1 = &nodes[0];  /* node pointer */
```

If just the pointer name '**pN1**' is entered,   the result yields the pointer value to be displayed in the WATCH window.   If the pointer name is entered in dereferenced   format: ***pN1**, then the whole object referenced by the pointer is displayed, in our case the complete 'node' structure.

Examples:
```
      pN1           /* display the pointer value */
      *pN1          /* display the node structure */
      pN1->next     /* display pointer value of node.next ref'd via pN1 */
      *pN1->next    /* display linked node struct 'pN1->next' */
```

Note that the application must been have built with full debug information, that is, using the DEBUG or DEBUG/OBJECTEXTEND controls, depending on the compiler.   dScope requires full type information in order to support aggregate references.   If the debug information is insufficient, an error message will be displayed since, for example, dereferences are not valid to a type other than a pointer type.   The same is true for structures also.

The Memory Map Dialog

The Memory map dialog assists in managing mapping of memory. The following paragraphs contains the description of the Memory map dialog and it's controls.
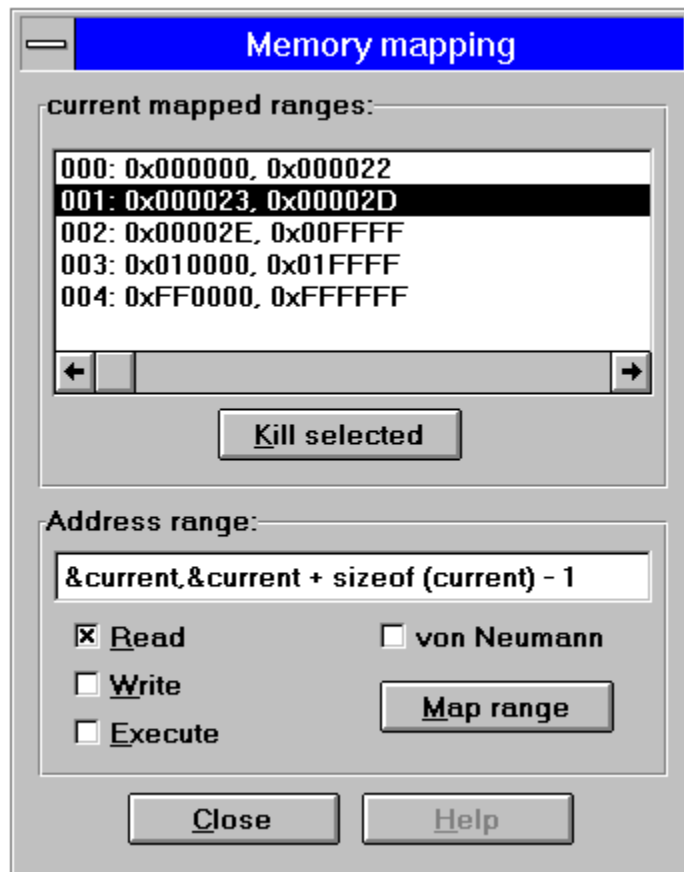
Description of the dialog controls:

◆ Current mapped ranges listbox

> The current mapped ranges lists all memory ranges currently mapped. By selecting a line from the list, the specific information for that mapped range is displayed using the controls located on the lower half of the dialog. The addresses shown in each line correspond to the scheme dScope uses to map logical to physical segments: 0x00nnnn := data/idata/edata, 0x01nnnn := xdata, 0xFFnnnn := code.

◆ Kill selected button

> The kill button is used for removal of the selected memory range from the list of mapped ranges. If no line is selected from the list, then the kill button is

disabled, otherwise it is enabled.   If a range is killed, then it gets physically removed which means accessing an address out of a non existing range causes an error message 'access violation' to be displayed in the COMMAND window.

◆ Address range expression entry

The address range may contain one or two address expressions.   If only one expression is entered, then a single byte is mapped.   If two expressions, separated by a comma are entered, then the memory range starting from expr1 to expr2 is mapped.   The expressions can be numeric constants as well as symbolic addresses as shown in the dialog picture.

◆ Access permission controls: Read, Write, Execute

The access permissions are used to restrict access to the mapped memory range to Read, Write, Execute or a combination of them.   Memory ranges which contain executable code have the permissions Execute/Read.   This is because read accesses may take place in MCS 51/251 application due to the MOVC A,@A+PC instruction, which reads from the code memory.

Note: the access permission of the internal data memory space (0x00 ... 0xFF) is ignored. This memory is always read-write which corresponds to the behaviour of the MCS 51/251 architecture.

◆ von Neumann control (applies to MCS 51/251):

The von Neumann control activates von Neumann mapping for the given range, if the control is checked.   This has the effect of directing accesses to the memory range given by the expression(s) to the code segment (0xFFnnnn).   The overall effect on this is that a write to a von Neumann mapped range physically writes into the code segment.   The von Neumann option identifies the specified memory area as memory type "von Neumann".   This causes an intentional overlapping of external data memory and code memory of the CPU.   The consequence of this is that write accesses to external data memory also change the code memory.   Note that memory ranges mapped as von Neumann must not cross a 64K boundary and the range must not be a range from the code segment, for example 0xFF8000,0xFFFFFF.

Make sure that the range you are about to map von Neumann has **Read and/or Write** attributes set.

◆ Map range button

With the Map range button, the given range is mapped, if the address range entry field contains some input.

◆ Help button

Open WinHelp with online help for the Memory-Map context

◆ Close button

Close the dialog, this can be done with the Close entry from the system menu also.   Closing the dialog does not change any mapped ranges.

dScope's memory map feature supports one byte granularity.   This means, you can map single bytes without being limited to a minimum block size.   After invocation of dScope, the following memory ranges are mapped with the following access permissions by default:

0x000000 - 0x00FFFF read write (the MCS 51/251 DATA space)
0x010000 - 0x01FFFF read write (the MCS 51/251 XDATA space)
0xFF0000 - 0xFFFFFF exec read   (the MCS 51/251 CODE space)

dScope supports up to 16M bytes of memory available for user programs, that is 256 segments of 64K bytes each.   The default MCS 51 and MCS 251 memory spaces are assigned by dScope to the segments with the numbers listed in the following table:

dScopes segment mapping scheme:

0x00 (D:)	maps to data segment starting at 0x00:0x0000 ... 0x00:0x00FF (0x00:0xFFFF on MCS 251)
0x01 (X:)	maps to default xdata segment (0x01:0x0000...0x01:0xFFFF)
0x80...0x9F	maps to banked code segments (0x80 is bank-0, 0x81 is bank-1,...)
0xFF (C:)	maps to the default code memory of the MCS 51 and MCS 251 (0xFF:0000...0xFF:0xFFFF).

If a user program is loaded into dScope, segments will be mapped as required by the user program.   This is true for both banked and non banked applications.   If MCS 51 user programs are loaded,   memory mapping commands are almost never required except for the special cases where the access permissions of a specific memory range is changed to catch illegal writes to some location.   The same is true for MCS 251 user programs with the exception of dynamic memory pools where dScope does not know about.

Although dScope supports 16M byte of user program memory, only the memory ranges required by the user program should be mapped, if mapping is performed by Map commands. Depending on the amount of memory available to dScope, mapping huge amounts of memory may slow down the execution speed of dScope, since a lot of disk swapping may take place. Any block of memory is allocated twice: the first block is the segment actually used for read, write and execute, the second block holds the specific attributes such as access permissions and information for code coverage and performance analysis.

Examples:

Map the address range starting at 0x20000 and ending at 0x2FFFF for read and write access:
Range: **0x20000,0x2FFF**
Permission: **Read Write**

Map the range 0x000000 to 0x00FFFF for read and write access.   This example makes sense for MCS 251 derrivatives which support 64K of data instead of the 128/256 bytes on the MCS 51 controllers:
Range: **0x000000,0x00FFFF**
Permission: **Read Write**

Given the previous mapped range, we want to detect write accesses to a structure named 'current' which resides in the data segment.   The command to do this is as follows:
Range: **&current, &current + sizeof(current) - 1**
Permission: **Read**

The first expression specifies the start of the range, the second expression the end of the range, by adding the size of the structure minus one to the starting address of the range.   That memory range is mapped for read only access.   The mapping of the memory ranges around the new range are not affected by this map command.   If the program attempts to write to the given range, dScope will give an 'Access violation' error message.   This example provides an alternative to overcome the one byte access limitation of Access breakpoints.

The following example maps xdata range 0x8000...0xFFFF (logical addresses 0x18000...0x1FFFF in dScope) to code address 0x8000...0xFFFF (logical addresses 0xFF8000...0xFFFFFF in dScope).   The range should have read/write permissions:
Range: **X:0x8000,X:0xFFFF**
Permission: **Read Write vonNeumann**
The result of this mapping is, if a program does a write to xdata address 0x8000, for example, it actually writes to 0x8000 in code memory.   On the other hand, reading from location xdata 0x8000 (0x18000) reads the code memory location 0xFF8000.   Xdata accesses to locations 0x0000 to 0x7FFF are directed into the xdata segment, not the code segment.

The Performance Analyzer Setup Dialog

The PA setup dialog assists in managing Performance analyzer ranges (PA range).   A   PA range is an entity by which dScope records the number of invocations and the amount of execution time taken for them.   A PA range is normally represented by a function, for example main() or timer0().   Such a range has a unique entry point at the starting address of the range and a unique exit point, the last address of the range.   dScope supports up to 255 PA ranges.
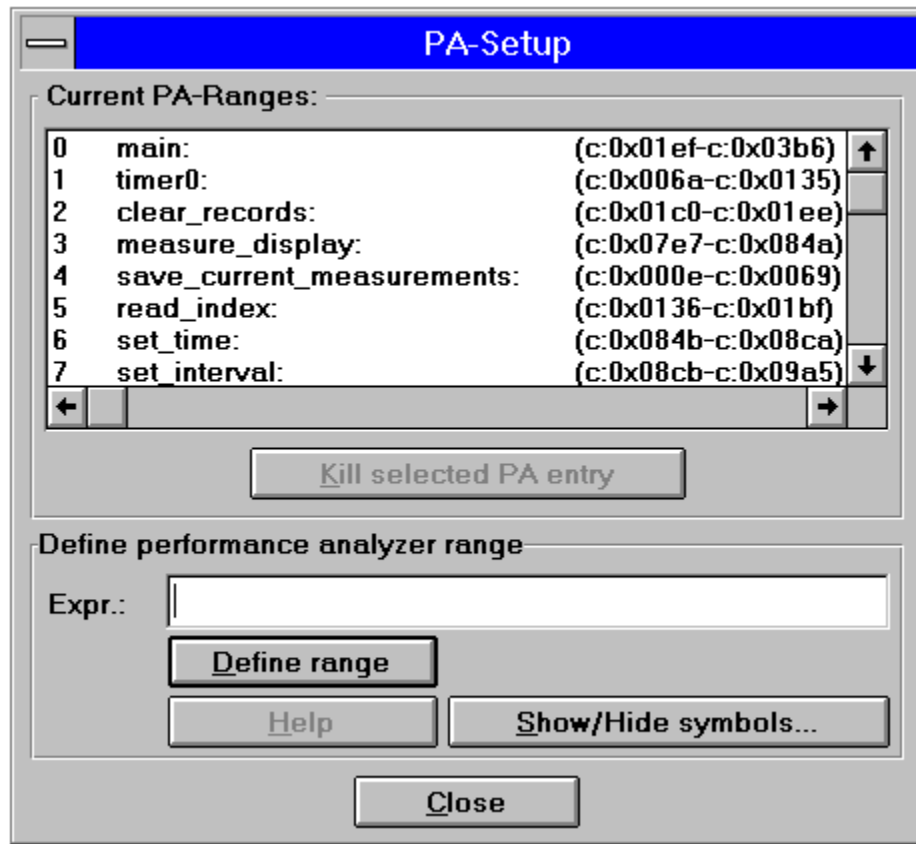
see also:

Defining a PA Address range

Identifying valid Address Ranges

Defining a PA Address range

In order to enable performance analysis, PA ranges have to be defined first.   A PA range is an address range with a *unique*   entry and exit point.   Such a range can be defined either using the PA command in the Command window or the **PA-Setup** dialog.

Description of the dialog controls:

◆ Current PA-Ranges listbox

> The current PA ranges lists all currently define PA ranges.   By selecting a line from the list,   the Kill selected button is enabled for removal of a PA range.   The display of each line starts with a dScope assigned PA number followed by a symbolic or numeric value identifying the range start.   The last item displayed is the PA address range.

◆ Kill selected PA entry button

> The Kill selected PA button is used for removal of the selected PA range from the list of ranges.   If no line is selected from the list, then the kill button is disabled, otherwise it is enabled.

◆ Address range expression entry

> In the *Expression* field, enter the range.   A range can be a function name of the user program or two numeric expressions which represent the starting and ending address of the address range.   If only a function name is given, dScope will derive the ending   address using the high level debug information loaded with the users program.   If such debug information is not available, an error message will be displayed.   When the entry in the *Expression* field is finished, click at the **Define range** button to add the range to the list of existing ranges.

> To ease the entry of symbols, you may select the **Show symbols** button which opens the Symbol browser window (if not already open).   From the Symbol browser, you can drag a symbol   into the *Expression* field as follows:

>> Move the mouse pointer to the requested symbol.
>> Press the left mouse button, and don't release the button (the cursor changes).
>> Move the mouse pointer to the *Expression* field of the **Watchpoints** dialog
>> Release the mouse button. The qualified symbols name is then filled in.

NOTE   PA range definitions can alternatively be defined and removed with the PA command in the command window.   Refer to dScope Commands for more information.

◆ Define range button

> With the Define range button, the given PA range is created and added to the list of ranges.

◆ Show symbols button

> The Show symbols button opens or closes the Symbol Browser window.   For simple entry of symbols into the Expression field, you can drag a symbol from the browser window into the entry fields.

◆ Help button

> Open WinHelp with on-line help for the Performance Analyzer context

◆ Close button

> Close the dialog, this can be done with the Close entry from the system menu also.   Closing the dialog does not affect currently defined PA ranges.
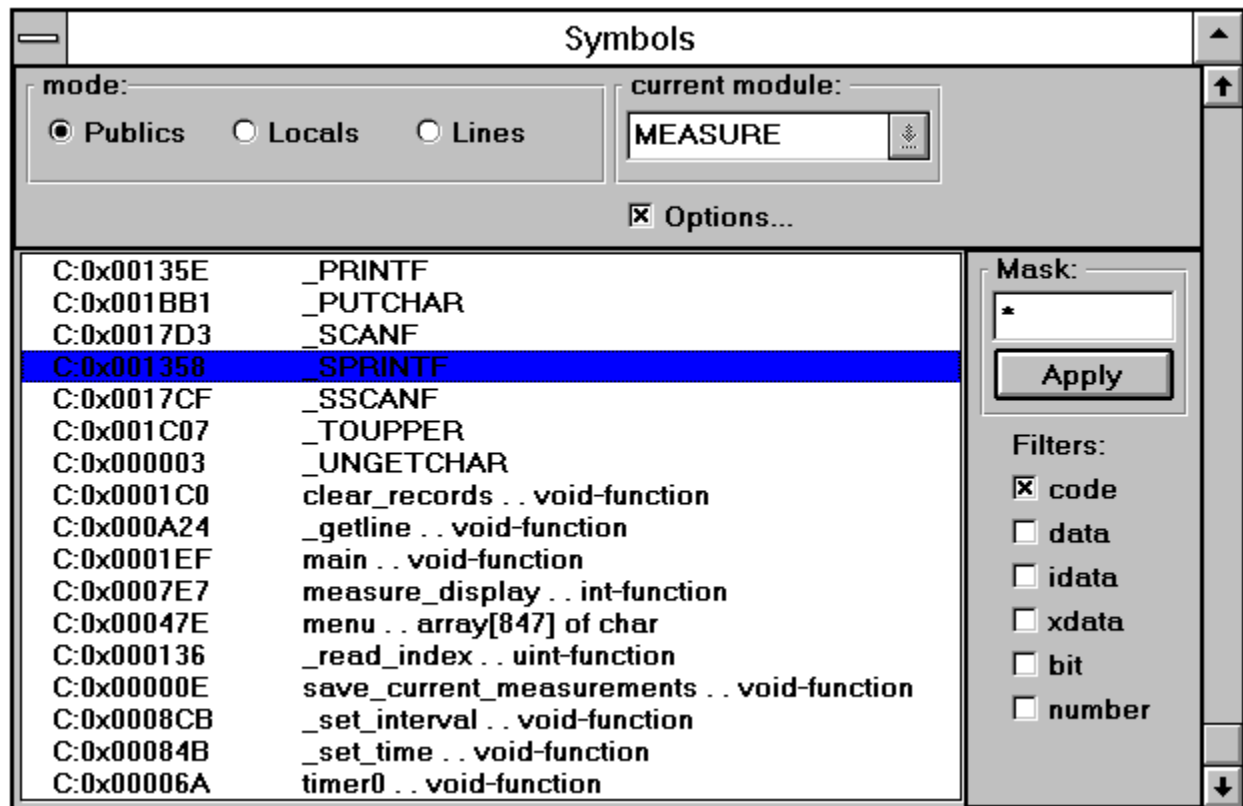
Identifying valid Address Ranges

A valid PA address range must have a unique entry and exit point.   Within dScope, you have two choices to find out the currently available address ranges.   The first one is to use the **Scope** command in the command window.   This command displays address range information about all currently defined blocks.   The following example shows part of the output created by the **Scope** command with the MEASURE sample application loaded into dScope:

```
MEASURE
  {CvtB} RANGE: 0xFF03B7-0xFF07E5
  {CvtB} RANGE: 0xFF000B-0xFF000D
  SAVE_CURRENT_MEASUREMENTS RANGE: 0xFF000E-0xFF0069  /* valid */
  TIMER0 RANGE: 0xFF006A-0xFF0135                     /* valid */
  _READ_INDEX RANGE: 0xFF0136-0xFF01BF               /* valid */
  CLEAR_RECORDS RANGE: 0xFF01C0-0xFF01EE             /* valid */
  MAIN RANGE: 0xFF01EF-0xFF03B6                      /* valid */
MCOMMAND
  {CvtB} RANGE: 0xFF09A6-0xFF0A23
  MEASURE_DISPLAY RANGE: 0xFF07E7-0xFF084A           /* valid */
  _SET_TIME RANGE: 0xFF084B-0xFF08CA                 /* valid */
  _SET_INTERVAL RANGE: 0xFF08CB-0xFF09A5             /* valid */
GETLINE
  _GETLINE RANGE: 0xFF0A24-0xFF0A87                  /* valid */
?C_FPADD
?C_FPMUL
```

The indented identifiers represent functions and there address range in memory.   You can use all ranges not starting with '**{**' and with the starting and ending address present.   The ranges named **{CvtB}** can't be used for performance analysis.   Such ranges result from functions which do not have sufficient debug information, usually code from libraries or modules which were not compiled for debug.

The second choice is to use the Symbol browser window to view or select a function for a PA range.   Choose the **Publics** or **Locals** mode, check **Options** and turn all memory spaces filters but code off.   With the MEASURE application loaded, the Symbol browser window will show something like this:

Symbols

mode:
- ● Publics   ○ Locals   ○ Lines

current module:
MEASURE

☒ Options...

C:0x00135E	_PRINTF
C:0x001BB1	_PUTCHAR
C:0x0017D3	_SCANF
C:0x001358	_SPRINTF
C:0x0017CF	_SSCANF
C:0x001C07	_TOUPPER
C:0x000003	_UNGETCHAR
C:0x0001C0	clear_records . . void-function
C:0x000A24	_getline . . void-function
C:0x0001EF	main . . void-function
C:0x0007E7	measure_display . . int-function
C:0x00047E	menu . . array[847] of char
C:0x000136	_read_index . . uint-function
C:0x00000E	save_current_measurements . . void-function
C:0x0008CB	_set_interval . . void-function
C:0x00084B	_set_time . . void-function
C:0x00006A	timer0 . . void-function

Mask:
`*`

Apply

Filters:
- ☒ code
- ☐ data
- ☐ idata
- ☐ xdata
- ☐ bit
- ☐ number

The client area shows some symbols with ' **... function** ' appended.   These symbols identify functions symbols which may be dragged into the PA-Setup dialog for definition of PA range without having to specify the functions end address.

The Inline Assembler Dialog

The Inline assembler dialog is provided for entry and direct assembly of assembler instructions. The following paragraphs describes the dialog and it's controls.
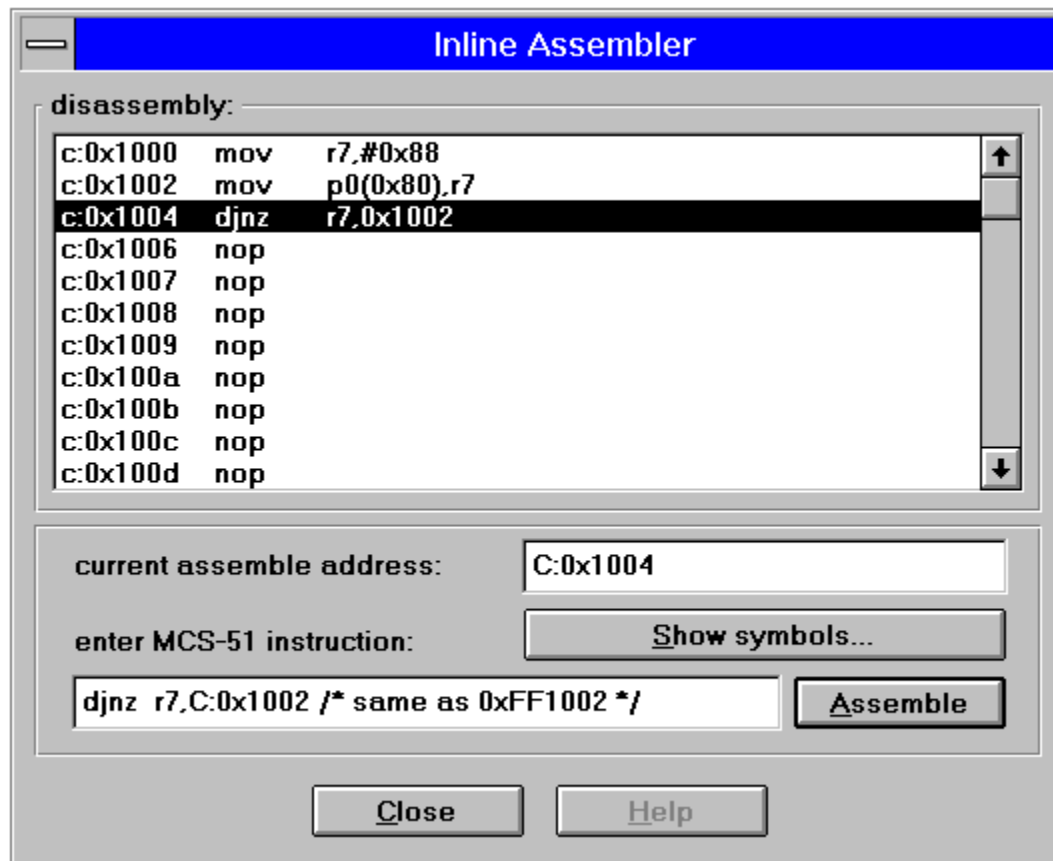
Description of the dialog controls:

- Disassembly listbox

    The disassembly list box displayes about 256 assembler instructions.   The address of the first instruction is C:0, if the dialog is invoked the first time. Otherwise, it will start at the address from a previous invocation of the dialog. You may change the address however by entering a new one in the current assemble address field.   Double clicking some line will change the current assemble address to be the address of the selected line.

- current assemble address entry field

    Shows the current assemble address, which is the address in memory where the next assembled instruction will be stored.   You can change this address by entering an address expression followed by ENTER.

◆ Instruction entry field

> This entry field is used to enter the assembly language instruction.   The valid assembly language instructions depend on the CPU driver currently loaded.   If for example the driver 80251S.DLL is loaded, the instructions for the 80251 will be accepted.   Be careful however, when entering the target for call and jump instructions.   Entering 'SJMP 0' for example when the 80251S CPU driver is loaded, will have the jump target to be 0x000000 which may be out of range due to the segment being different since the default code segment is located at 0xFFnnnn.   In this case, enter '**SJMP 0xFF0000** or **SJMP C:0'**.

> If the entry of the instruction is finished, press the Assemble button or hit the Enter key.   If something is wrong with the instruction, an error message will be displayed in the COMMAND window.

> The operands to instructions can be symbols or line numbers, such as 'JMP \MEASURE\210' or 'MOV sindex,A'.   To ease the entry of symbols and for automatic scope resolve, you may select the **Show symbols** button which opens the Symbol browser window.   From the Symbol browser, you can drag a symbol into the *Expression*   field as follows:

>> Move the mouse pointer to the requested symbol.
>> Press the left mouse button, and don't release the button (the cursor changes).
>> Move the mouse pointer to the *Expression*   field of the **Watchpoints** dialog
>> Release the mouse button. The qualified symbols name is then filled in.

◆ Assemble button

> With the Assemble button, the given instruction is assembled and the resulting opcodes are stored at the current inline assemble address and the next consecutive addresses.   After that, the current assemble address is incremented by the number of opcodes stored.

◆ Show symbols button

> The Show symbols button opens or closes the Symbol Browser window.   For simple entry of symbols into the Expression field, you can drag a symbol from the browser window into the entry fields.

◆ Help button

> Open WinHelp with Inline-Assembly context.

◆ Close button

> Closes the dialog.

The Color & Font Dialog

Each dScope window can be given individual colors and fonts.   For color and font setup, it is a good idea to load some application, (the MEASURE program, for example) before doing the setup.   In order to setup the MEASURE application and providing output to almost any dScope window, follow these steps:

> In the COMMAND window, enter the command   -
> **INCLUDE C:\C51\EXAMPLES\MEASURE\MEASURE.INI**
> Note that the actual path may be different depending in your installation.   Supply your drive and root path (C:\C51 is assumed here).   The include file contains all commands to be ready for running the MEASURE sample program.

> In the DEBUG window, click the Go button.   dScope will execute the sample program.   Let the program run for a few seconds, the click the Stop button. Now, every dScope window contains some output, regardless of whether a particular window is visible or hidden.

> In the COMMAND window, enter the command -
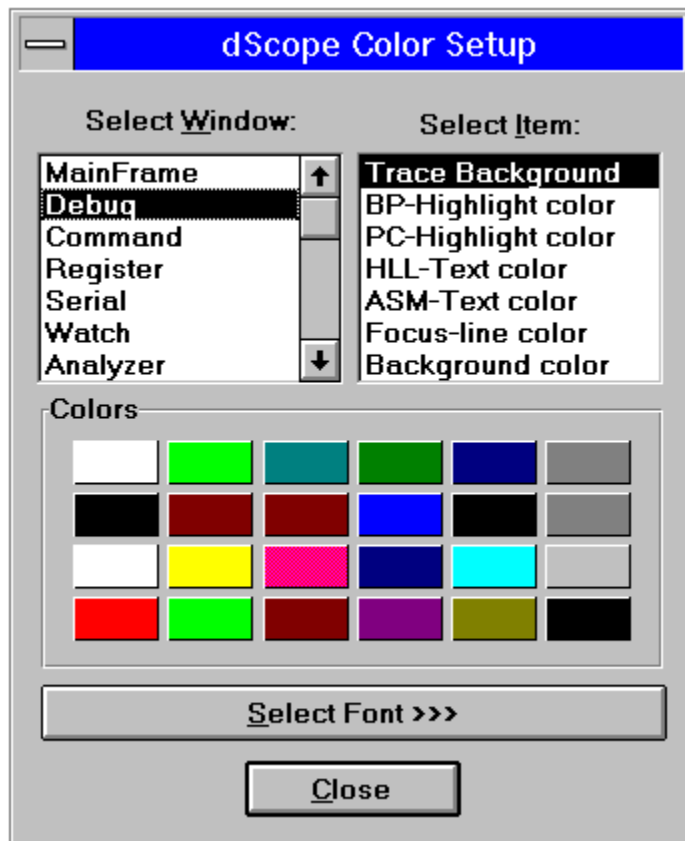>
> **U main**
> This will show the source code for function main() in module 'measure'.

> In the DEBUG window, select 'View Mixed' from the 'Commands' menu.   This changes the view mode to mixed.   You will get source lines intermixed with assembly lines.   Since the color for source lines and assembly lines can be distinct, this step makes sense.

After these steps, each window contains some output.   Since colors and fonts can be setup individually for each window, some output is required for the windows to be setup.

To configure a specific window, follow the guidelines listed below:

- ◆ Open the window you want to configure.
- ◆ From the **Setup** menu, choose **Colors and fonts**.   dScope will show the color dialog box:

dScope Color Setup

Select **W**indow:

- MainFrame
- Debug
- Command
- Register
- Serial
- Watch
- Analyzer

Select **I**tem:

- Trace Background
- BP-Highlight color
- PC-Highlight color
- HLL-Text color
- ASM-Text color
- Focus-line color
- Background color

Colors

Select Font >>>

Close

The dialog contains a list named 'Select Window', which contains the names of all dScope windows that can be setup.   If you select one entry,   the second listbox will show the associated items which can be setup for the given window. If the window to be configured allows for font changes, then the button labeled **'Select Font>>'** will be shown, otherwise it is hidden.

◆ Select the **Window** you want to configure, for example the DEBUG window.   For configuration of the colors, you should see something like this:
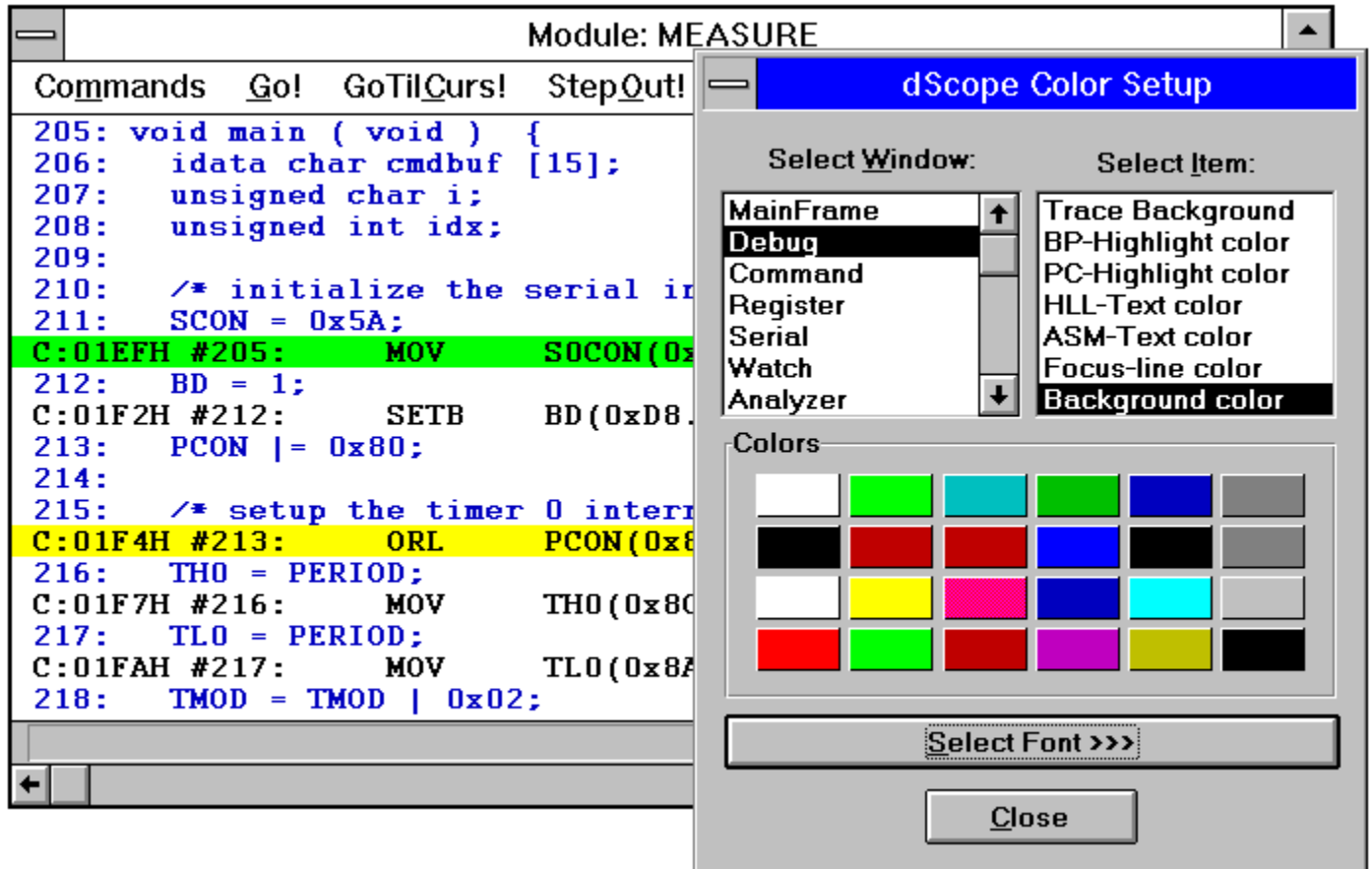
- Choose an **Item** from the listbox, for example **Background color**.   From the color palette, select some color.   The DEBUG window background will change immediately.   If you want to set the PC-Highlight color, make sure the current PC is within the visible area of the DEBUG window.   This can be achieved in our example by double clicking the right mouse button at line #205 which means go till selected cursor line.   If   you want to set the BP-Highlight color (Breakpoint mark color) then double click with the left mouse button at some assembly line.   After that, you can choose new colors.   The color change will take place immediately.

- You may also change the font to match your taste and the screen capabilities.   Choose the **Select Font** button to open the choose font dialog:
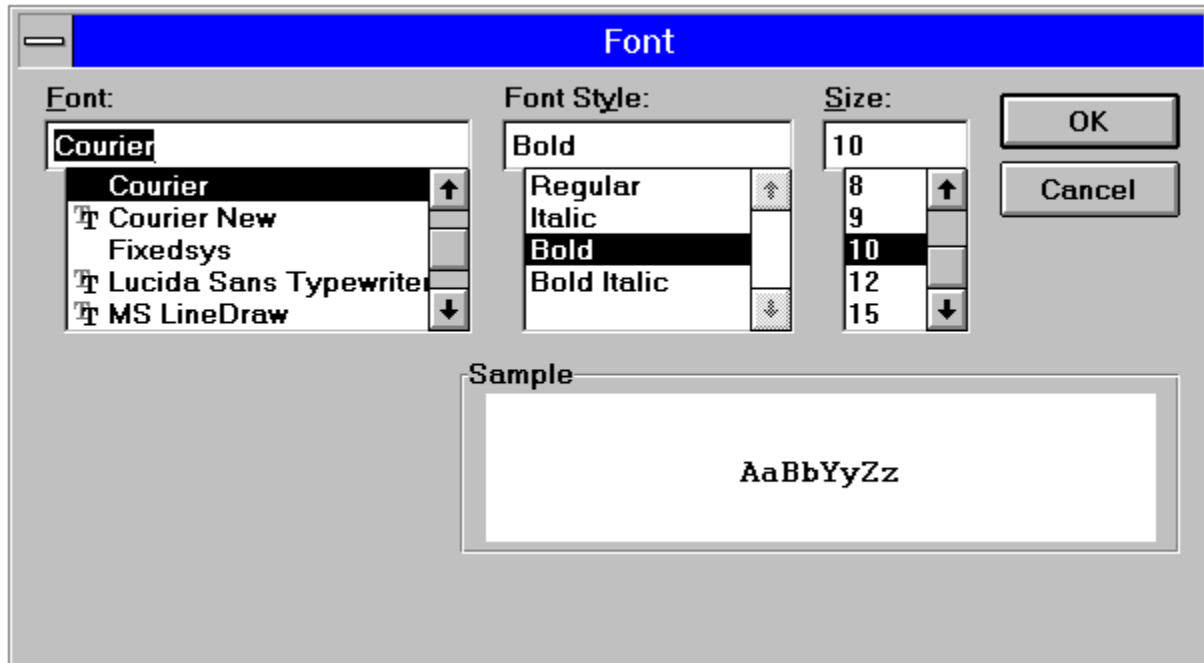
- ◆ Choose some font, font style and font size and click the OK button.   The DEBUG window will get repainted using the new font.   If the result seems not OK to you, then repeat the font selection step.

NOTE The actual content of the Font listbox may be different on your computer.   This depends on the fonts available.   If you have some font package installed on your computer, the Font listbox may show more fonts available than on a computer with standard Windows 3.1 installation.

- ◆ When you are finished, choose another window for color setup or close the color setup dialog.

The configuration of dScope windows other than the DEBUG window is straightforward.   If you want to setup another window, make sure it is visible.   After that, select the window   in the 'Select Window' listbox from the color setup dialog.   The 'Select Item' listbox will display the items which can be setup for the given window.

dScope will save the color and font settings as well as the window sizes and positions on Exit in file *DSW51.INI*   located in the path where the dScope executable **DSW51.EXE** exists.

Introduction

Most of the dScope commands contain numeric expressions as parameters.   A numeric expression is, in the simplest case, a number or a complex expression that contains numbers, debug objects or operands.   dScope entry lines that do not contain commands are automatically handled as expressions.   The assignments, for example "R7 = --ACC", are immediately executed here.   Entry lines without assignments display the calculated result of the expression.   The operators available are identical with those of the C language.

Components of an Expression

An expression can consist of the following components:

- Constants

  Are fixed numeric values or character strings.

- System Variables

  Are predefined variables within dScope.

- Variables (symbols)

  Refer to variable names addressed by symbolic names.   The variable names refer to the names of objects in a loaded user program.

- Operators

  Identify operations that are to be executed with the subexpressions.   The possible operators correspond to the conventions of the C language.

- Line Numbers

  Refer to the code addresses of executable 8051 programs.   Line numbers are generated by the compilers "C51" and "PL/M-51" during the compilation and stored in the object file.

- Bit Addresses

  Are address specifications that refer to the bit addressable data memory of the MCS 51/251 microcontrollers.

- Memory Space

  The memory type allows the assignment of an expression to a physical address space of the 8051 microcontroller.

- Type Specifications

  Used for the type adoption of expressions and subexpression.

see also:
Constants
HEX Constants in C Notation
Constants with a Specified Number Base
Floating Point Numbers (float)
Character Strings

Constants

dScope understands hexadecimal constants that are written using the C conventions as well as numbers, with a base specification suffix.   If the base specification is missing, the number base defaults to decimal.

HEX Constants in C Notation

0xnnnn or 0Xnnnn, i.e.   0xFF,   0xab04, 0xFF0123

Constants with a Specified Number Base

Hexadecimal: i.e.   1234H
Octal: i.e.   777Q or 777O
Decimal: i.e.   1234T or simply 1234
Binary: i.e.   11111111Y

For better readability, numbers can be grouped with the dollar character (this also applies for float and HEX constants in C notation):

1111$1111y is the same as 11111111y

Numbers must begin with a leading zero when the next character is a HEX character in range A ... F.   If a number contains a HEX character but not a base specification, the base HEX is automatically assumed.

Each number besides floating point numbers can be assigned a long suffix.   This allows the calculation of a value as a long (32-bit integer) value and not as an 16-bit value (int):


```
0x1234L, 1234TL, 1255HL
```


When, during the lexical analysis, a number is determined that exceeds the value of the area for int (0..65535), the type of the number is automatically changed to long int.   This avoids the truncating of significant digits.

Floating Point Numbers (float)

decimal value   .   decimal value
decimal value E [+/-   ] decimal value
decimal value   .   decimal value   [E [+/-] decimal value]

Examples:
`4.12, 0.1e3, 12.12e-5`

In comparison to the writing conventions in the C language, a floating point number must begin with the digit before a decimal point; for example, the display of .12 is not allowed and must be entered as 0.12.   This is to avoid confusion with bit addresses.

Character Strings

The same rules which apply for the C language also apply to the characters .   Embedded Escape sequences and numeric values are also therefore supported.   In comparison to C, successive character strings are not chained to one character string.

Examples:
```
"string\x007\n"
"value of %s = %04XH\n"
```


In some cases, it is necessary to have strings nested, for example when defining a Toolbox button.   Since such a command itself must be string, strings may get nested:

> ```
> "printf ("hello world!\n")"
> ```

If the string is written as shown, a syntax error will occur.   The correct way of writing such strings is:

> ```
> "printf (\"hello world!\n\")"
> ```

Note the escaped double colons that enclose the nested string.

Character Constants

The same rules apply for character constants as for the C language.   Therefore, Escape sequences are also supported.   The following Escape sequences are processed within character constants and character chains:

\\	Backslash, literalized
\a	Alert, bell
\b	Backspace
\f	Formfeed
\n	Newline
\r	Carriage return
\t	Tabulator
\ooo	Octal constant
\Xhhh	HEX constant

Examples:

```
'a', '1', '\n', '\v', '\x0FE', '\015'
```

System Variables

System variables within dScope are predefined numeric variables. They can be contained in expressions. The variables can be entered in both uppercase and lowercase. The following system variables are defined:

- ◆ CYCLES:

    The variable "cycles" always shows the current state of the cycle counter. This is increased during the program execution. "Cycles" is a 32-bit value of the type "unsigned long". The value of the variable "cycles" is displayed in the REGISTER window and allows the determination of the program execution times.

- ◆ RAMSIZE:

    RAMSIZE indicates the size of the internal, direct-accessible data memory. The default setting is 0x100 (256) bytes; this agrees, for example, for the 80C515/517 CPU. For CPUs with less internal data memory (i.e. 8051, 128 bytes), the variable must be changed accordingly:
    ```
    RAMSIZE = 0x80      /* Set RAMSIZE for the  8051 */
    RAMSIZE             /* Interrogation             */
    0x80                /* Output                    */
    ```
    RAMSIZE has the type "unsigned int".

- ◆ RADIX:

    The RADIX variable determines the number output base for numeric values. It has the default setting 0x10 for hexadecimal output. Any other value from RADIX is interpreted as decimal:
    ```
    RADIX = 0x0A        /* Decimal output base */
    RADIX = 0x10        /* Hexadecimal output base */
    RADIX = 3           /* Interpreted as decimal output */
    ```

- ◆ _IIP_:

    "Interrupt In Progress" status indicates the current interrupt nesting. The type of _IIP_ is "unsigned char".

- ◆ $:

    $ represents the actual state of the program counter (PC) and is of the "unsigned long" type. The program counter is output in the REGISTER window.

- ◆ ITRACE:

    ITRACE indicates whether trace recording will be performed during the program. It is executed when the value is not equal to zero, otherwise it is not executed.
    ```
    ITRACE = 1          /* Trace recording enabled */
    ITRACE = 0          /* Trace recording disabled */
    ITRACE              /* Output of the current value of ITRACE */
    ```

- ◆ _BREAK_:

  If the value of the system variable "_BREAK_" is set to "1", the execution of the running 8051 program is interrupted.   The program run can, for example, be stopped from within a dScope debug function.

  ```
  _BREAK_ = 1          /* Stop program execution of the 8051 CPU */
  ```

- ◆ _MODE_:

  MODE indicates the CPU mode when simulating MCS 251 code.   It's value may be 0, which specifies the **binary mode**   of operation or 1, which specifies the **source mode**   of operation.   If the user program loaded has been created with the A251/C251/L251 tools, then dScope automatically recognizes the mode of operation the application runs with.   The _MODE_ variable is not relevant for non MCS 251 applications and should be zero.

  ```
  _MODE_ = 1           /* Assume source mode of MCS 251 operation */
  _MODE_ = 0           /* Assume binary mode of MCS 251 operation */
  _MODE_               /* display the current value of _MODE_ */
  ```

- ◆ _FRAMESIZE_:

  The _FRAMESIZE_ variable indicates the frame size on execution of interrupt procedures.   The value of _FRAMESIZE_ may be either 0, which specifies 2-Byte frames or 1, which specifies 4-Byte frames.   The frame size comes into play when simulating MCS 251 applications.   On interrupt, the MCS 251 first saves the PSW, then the 24 bit return address on stack since interrupt procedures may reside at any address of the 16M byte address space.

  _FRAMESIZE=0:   16 bit return address (default for MCS 51)
  _FRAMESIZE=1:   PSW, 24 bit return address

  ```
  _FRAMESIZE_ = 1      /* Assume MCS 251 interrupt frame */
  ```

Variables (Symbols)

Variables are objects that are addressed with a symbolic name.   Symbol names represent numeric values and addresses.   Symbols are stored in object files as debug information by language compilers (C51, A51, PL/M-51 or 251 tools).   These are available for the symbolic testing of programs.

The conventions for symbol names are shown below:

- Maximum of 31 characters

- First character 'A'-'Z', 'a'-'z' or '_' or '?'

- Following character 'A'-'Z', 'a'-'z', '0'-'9', '_', '?'

When a symbol begins with a question mark and when the conditional operator "?" follows, a blank character must be entered as a delimiter:

```
r5 = acc ??symbol : r7          /* Wrong, no blank space */
r5 = acc ? ?symbol : r7         /* Correct */
```

This means that symbols can begin with the question mark within dScope and in assembler programs but not, however, in the C language.   dScope does not differentiate between uppercase and lowercase and knows two classes of symbols:

- Reserved Words

  Reserved words are all predefined symbols including dScope commands, command options and system variables.

- Program Symbols (Objects)

  Program symbols are stored by loading an object file, when the object program provides debug information.   This is the case when the run of the C51 compiler or A51 assembler was performed with the invocation option "DEBUG".

Reserved Words

◆ Register names and names that appear in the assembler language.

The names can be used in expressions and represent the contents of the named register (except AB):

R0 to R7 Specifies the working register of the current register bank of the 8051
A   Accumulator of the 8051
C   Carry bit of the 8051 program status word
AB   For the MUL and DIV instruction in the inline assembler
DPTR   16-bit data pointer
WRn   MCS 251 word registers WR0, WR2, ... WR30
DRn   MCS 251 dword registers DR0,DR4, DR8, ... DR60

◆ Predefined CPU symbols.   These symbols are defined even when no user program is loaded but a CPU driver was loaded:

These symbols define the special function register names and their associated special function bits.   The names of the CPU symbols are different among the different CPU drivers.

◆ Data type names:

VOID, BIT, CHAR, UCHAR, INT, UINT, LONG, ULONG, FLOAT, PTR, BYTE, WORD, DWORD, REAL

◆ Commands and options as well as reserved words of the integrated procedure language:

The identifiers are the names of dScope commands as well as command option names.

Literalization

Command words and command options cannot be a part of a numeric expression.   When the name of a symbol from a user program duplicates the name of a reserved word, the symbol must be literalized in this case.

Example:

Set breakpoint at the address of the function "switch" in the current module.   Since the word 'switch' is reserved in dScope, a syntax would result in the following example:

```
BS switch     /* gives an error */
BS `switch    /* Correct, since literalized with ` */
```

Literalization allows the symbol to avoid being recognized as a reserved word.   Instead, the symbol table built on the load of the user program is searched to resolve the symbol.

Searching for fully Qualified Symbols

A fully qualified symbol contains up to three components:

- **The module name -** Identifies the module name where a user symbol is defined in.
- **The function name -** Identifies the function name within the module, where a local symbol is defined.
- **The symbol name -** Identifies the name of the symbol to be searched for.

The third component may be missing, that is, if the second component specifies a symbol in a module rather than a function.   It is also possible to specify qualified line numbers.   In this case, the second component specifies the line number with the third component omitted from the qualification.

Examples of qualified symbols and lines:

\MEASURE\clear_records\idx	Identifies symbol '**idx**' in function '**clear_records**' which is in module '**MEASURE**'.
\MEASURE\MAIN\cmdbuf	Identifies symbol '**cmdbuf**' in function '**MAIN**' which is in module '**MEASURE**'
\MEASURE\sindx	Identifies symbol '**sindx**' in module '**MEASURE**'
\MEASURE\225	Identifies line **225** in module '**MEASURE**'.
\MCOMMAND\82	Identifies line **82** in module '**MCOMMAND**'.
\MEASURE\TIMER0	Identifies symbol '**TIMER0**' in module '**MEASURE**'. This symbol may be a function or non-function symbol.

If a local symbol to a function is to be searched for, then the module name and the function name must be part of the qualification.   If a line number is to be searched for, then the module name and the line number must be given in the qualification.

What is a Module Name ?

A module name is a name assigned to an entity of the user program.   For example, if the user program contains the code of a compiled source module named **MCOMMAND.C**, then the module name is **MCOMMAND**.   That is, for C language modules, the module name is always the name of the C language source file without file path and file name extension.

For modules compiled with the PL/M51 compiler, the module name is constructed differently.   A PL/M module always starts with an identifier followed by a colon followed by the DO statement:

```
MY_MODULE:  DO; /* the first statement in a PLM module */
  /* other PLM statements */
END;            /* each PLM module is terminated by END; */
```

The colon separated identifier starting each PLM module designates the module name of a PLM module.

Making Symbolic Information available

The table of modules names and their associated symbols (such as functions and local symbols of the function) is created when the object file is loaded.   This assumes that the object file contains debug information.   This is produced by the "A51" and "C51" compilers using the DEBUG control option.   In order to maintain type information, the OBJECTEXTEND control option is used for the C51 compiler (with A51, this is not possible).   dScope can then recognize the type of objects including structures and their elements.

Searching for Non-Qualified Symbols

During the analysis of an expression, names which occur are searched according to certain rules.   The rules depend on the context the expression is analyzed:

- ◆   Non-qualified symbol in function definition mode
- ◆   Non-qualified symbol in command mode

Qualified symbols are searched for in the qualified path.   If such a symbol cannot be found by dScope, an error occurs.   The following search sequence, in contrast, applies for non-qualified symbols:

1. **A dScope function is defined at the time:**
   The search is conducted in the table of the local symbols of the function currently contained in the definition.

2. **when 1. is not true or fails:**
   The search is performed in the table of symbols in the current module.   The current module depends on the value of the program counter ($).   Symbols in the current module represent symbols that have been defined within the module, but outside any functions (file scoped symbols or module STATIC's).

3. **When 2. fails:**
   The search is performed in the table of symbols created by 'DEFINE <type> name' commands.   Such symbols are created within dScope and are not part of the user's program.

4. **When 3. fails:**
   The symbol table of the GLOBAL symbols (PUBLIC) is searched.

5. **When 4. fails:**
   The table of the PIN register is searched.   This table is available when a CPU driver (i.e.   80517.DLL) is loaded.   The peripheral register names are summarized in a table provided by the CPU driver.

6. **When 5. fails,** it concerns an unknown symbol**.**

Line Numbers

Line numbers are produced by the compilers for testing purposes.   This gives a relation between a statement of a source program and the physical address of the first assembler instruction that was produced by the compiler for the statement.   Line numbers are stored in the object file and are available for the symbolic debug of a program.   For high-level language debugging, line numbers are vital because the option is granted to display the original lines of a source program or listing (possibly mixed with the assembler instructions), instead of pure assembler mnemonics.

dScope allows the use of line numbers within expressions.   A line number represents an address in code memory.   The syntax is as follows:

```
\integer_constant
\modulename\integer_constant
```

In the first case, the line number must exist in the current module.   In the second case, it must exist in the specified module.   If the specified module or line number does not exist, an error message is issued.

Examples:
```
\measure\108    /* Line 108 in module "MEASURE" */
\143            /* Line 143 in the current module */
```

Bit Addresses

A bit address represents the address of a bit in bit-addressable memory or in one of the Special Function Registers (SFR) of the microcontroller.

```
expression . bit_position
```

The value of <expression> depends on the CPU being simulated.   For MCS 51 style controllers,   the result of <expression> must be a numeric value between 0x20 and 0x2F or 0x80 and 0xFF.   For MCS 251 style controllers, the result of <expression> must be a value in range 0x20 to 0xFF.   Bit_position must evaluate to an integer constant in the range 0 to 7.

Examples:

```
20H.0        /* Bit 0 */
0x2F.7       /* Bit 127 */
ACC.0        /* Bit 0 of the accumulator (0xE0) */
```

Memory Space Prefix

The memory space prefix is used to assign a memory type to an expression with a constant. For example, this is necessary when an expression is used as an address for an output command.   Symbolic names normally have an assigned memory space so that the specification of the memory space can be omitted.   The following MCS 51 memory spaces known:

C: Code memory (CODE)

D: Internal, direct-addressable RAM memory of the 8051 (DATA)

I: Internal, indirect-addressable RAM memory of the 8051 (IDATA)

X: External RAM memory (XDATA)

B: Bit-addressable RAM memory

P: Peripheral memory (VTREGS - 80x51 pins)

EB: extended Bit-addressable RAM memory (EBIT, MCS 251)

ED: extendend data RAM memory (EDATA, MCS 251)

The prefix **P:** represents a special case.   P: must always be followed by a name which is then searched in a special symbol table.   This table contains the names of the pin registers that are available only if a CPU driver (i.e.   80517.DLL) has been loaded.

Besides the symbol memory space prefixes, the prefix may also be a numeric constant in range 0x00 to 0xFF which identifies the segment being referenced. The relationship between the symbolic and numeric prefix is shown in the following table:

0x00	maps to data segment starting at 0x00:0x0000 ... 0x00:0x00FF (0x00:0xFFFF on MCS 251). This is equivalent to D:, I:, ED:
0x01	maps to default xdata segment (0x01:0x0000...0x01:0xFFFF). This equivalent to X:
0xEF...0xFE	maps to banked code segments (0xFE is bank-0, 0xFD is bank-1,...)
0xFF	maps to the default code memory of the MCS 51 and MCS 251 (0xFF:0000...0xFF:0xFFFF). This is equivalent to C:

Examples:

```
C:0x100                 /* Address 0x100 in code memory */
I:100                   /* Address 0x64 in internal RAM of the 8051 */
X:0FFFFH                /* Address 0xFFFF in the external data memory */
B:0x7F                  /* Bit address 127 or 2FH.7 */
ACC                     /* Address 0xE0, memory type = D:   */
C                       /* Address 0xD7 (PSW.7), memory type = B:   */
0xFF:0x0100             /* same as C:0x100 */
0x01:0x1234             /* same as X:0x1234 */
0x00:0x20               /* same as D:0x20 or I:0x20 */
```

Type Specifications

Type specifications change the type of the expression or subexpression.   dScope automatically performs type adaptations within an expression.   The user can, however, perform explicit type adoptions.   The methods for this correspond basically to the C language with the exception of type adoptions to pointers which are not possible.   The type names are simplified, in comparison to C (i.e.   uint for unsigned int) and extended to be of equal use for PL/M-51 users:

```
bit            /* Type bit */
char           /* Type signed char */
uchar          /* Type unsigned char */
int            /* Type signed int */
uint           /* Type unsigned int */
long           /* Type signed long */
ulong          /* Type unsigned long */
float          /* Type float */
byte           /* corresponds to uchar */
word           /* corresponds to uint */
dword          /* corresponds to ulong */
real           /* corresponds to float */
ptr            /* C51 generic Pointer */
```

Examples:

```
(float) 0x100       /* Produces 256 from type float */
float (c:0x100)     /* Reads float value from addr 0x100 in code mem */
float(c:0x100)=3.1  /* Assignment */
float(c:0x100)      /* Produces 3.1 */
acc                 /* Predefined:  Accum.  (type uchar, addr.  0xE0) */
D:0xE0 = 0x00       /* Output of the result value */
acc = 5             /* Assignment */
(float) acc         /* Read accum.  (uchar) and conv.  to type float */
```

Operators

dScope supports all operators of the C language.   The operators have the same meaning, unless otherwise described.   Some operators and operations are to be considered as special extensions.   The following table lists all operators and operands in BNF representation with descending priority:

- ◆ primary:

```
const                        /* Constants (int, long, float) */
string_const                 /* Character string constants, i.e. "ds51\n" */
system_variable              /* i.e. CYCLES, _IIP_ */
register_name                /* i.e. R0, R7, A, C */
type(mspace:const)           /* i.e. int (x:0x12) */
mspace primary               /* i.e. C:0x1000 */
pspace identifier            /* i.e. P:port5, P:ain0 */
id (expr)                    /* Invocation of DS funcs, printf("end\n") */
id                           /* Non-qualified symbol */
\mod\func\id                 /* Fully-qualified symbol */
\mod\id                      /* Partially-qualified symbol */
\const                       /* Line number */
\mod\const                   /* Qualified line number */
( expr )                     /* Parenthesis */
```

- ◆ postfix:

```
primary
postfix++/--                 /* Post inc/dec, i.e.  R0--, CYCLES++ */
postfix.id                   /* Structure, i.e.  interval.msec */
postfix->id                  /* Structure pointer, i.e.  pint->msec */
postfix.const                /* Bit address, i.e ACC.7, 0x20.5 */
postfix [expr]               /* Array access, i.e.  save_record[1] */
```

- ◆ unary:

```
postfix
++/--postfix                 /* Pre inc/dec, i.e.  --R0, ++CYCLES */
&unary                       /* Address, i.e.  &ACC, &R0 */
*unary                       /* Reference, i.e.  *read_index.buffer */
+unary                       /* Unary plus, i.e.  +ACC */
-unary                       /* Unary minus, i.e.  -0x10, -R7 */
~unary                       /* Not bit, i.e.  ~ACC, ~R4 */
!unary                       /* Not logical, i.e.  !R0 */
sizeof (unary)               /* Size of operator, i.e.  sizeof(R5) */
sizeof (type)                /* i.e.  sizeof (float) */
```

- ◆ expr:

```
unary
(type) expr                  /* Type conversion, i.e. (long) charval */
expr * expr                  /* Multiplication, i.e. ACC * R5 */
expr / expr                  /* Division, i.e. CYCLES / 2 */
expr % expr                  /* Module, i.e. ACC % 7 */
expr + expr                  /* Addition, i.e. CYCLES + R5 */
expr - expr                  /* Subtraction, i.e. CYCLES - R7 */
expr >> expr                 /* Right writing, i.e. ACC >> 3 */
```

```
expr << expr              /* Left writing, i.e. ACC << 3 */
expr < expr               /* Less than, i.e. R5 < R7 */
expr > expr               /* Greater than, i.e. R5 > R0 */
expr >= expr              /* Greater or equal to, i.e. DPTR >= 200 */
expr <= expr              /* Less than or equal to, i.e. DPL < DPH */
expr == expr              /* Equal, i.e. DPL == (DPH + ACC) */
expr != expr              /* Unequal, i.e. DPL != ACC */
expr & expr               /* AND, i.e. ACC & 7 */
expr ^ expr               /* XOR, i.e. DPH ^ ACC */
expr | expr               /* OR, i.e. DPTR | 0x1 */
expr && expr              /* Logical AND, i.e. ACC==3 && !R7 */
expr || expr              /* Logical OR, i.e. ACC==3 || R5==R2 */
expr, expr                /* Expression list, i.e. R5,R4,-1 */
expr ? expr : expr        /* Condition: ACC==1?R5:R7 */
expr asnop expr           /* Assignments, i.e. DPTR = -1 */
```

◆ asnop:

```
=                 /* Simple assignment, i.e. ACC = R7 */
+=                /* Addition assignment, i.e. R5 += 5 */
-=                /* Subtraction assignment, i.e. ACC -= R3 */
*=                /* Multiplication assignment, i.e. DPL *= 4 */
/=                /* Division assignment, i.e. ACC /= 2 */
%=                /* Modulo assignment, i.e. DPH %= 7 */
<<=               /* Left writing assignment, i.e. ACC <<= 1 */
>>=               /* Right writing assignment, i.e. R5 >>= ACC */
&=                /* Bit AND assignment, i.e. R5 &= 1 */
|=                /* Bit OR assignment, i.e. ACC |= 2 */
^=                /* Bit XOR assignment, i.e. DPL ^= ACC */
```

Address Expressions

Some commands contain one or more expressions as parameters that are to be interpreted as addresses; i.e.   with display memory command:

```
D 0x100        /* Display memory command */
```

The illustrated command outputs the memory contents in byte representation.   After it gives several address spaces in the 8051, it is not clear to the display command which address spaces are concerned.   An error message is issued and the command is not executed.

```
D C:0x100      /* Display memory command */
```

The illustrated command only functions correctly because the start address as well as the memory type is known in the case of the code memory.

Another case represents the predefined dScope function "memset()".   This function is used to initialize the memory area with a value:

```
MEMSET (startaddress, endaddress, value)
i.e.:  MEMSET (X:0x100, X:0x1000, ACC)
```

In this case, "memset" only knows the address area and the value with which   the area should be written (each of the accumulators), but not the memory space.   The function invocation is therefore aborted with an error message.   "memset" must recognize a memory space from the expression parameters:

```
MEMSET (X:0, X:0x7FFF, -1)              /* XDATA from 0 to 32767 */
MEMSET (ACC, 0xFF, 0)                   /* DATA ( ACC !) from 0xE0 to 0xFF */
```

It is important to specify a memory space for an expression that contains only constants.   The other case is when the expression contains a symbolic name of an object whose memory space can be derived from that expression:

```
D ACC
```

ACC is a predefined symbol for the accumulator that lies in internal RAM of the 8051 at address 0xE0.   This satisfies all conditions for the "Display" command.
The entry of ACC without specification displays the current contents of the accumulator as a result.   ACC as an address expression uses, however, only the address of the accumulator and not its contents.   This corresponds to the syntax:

```
D &ACC         /* Address of accumulator */
```

NOTE For address expressions, dScope automatically adds the address-of operator (& prefix),

even when this was not explicitly specified.   This prevents the contents of the accumulator from being used as the address as in the prior case.

Differences Between dScope and C Expressions

dScope expressions contain some differences in comparison to C expressions, as described below:

- dScope does not differentiate between uppercase and lowercase for symbolic names and command names.
- No type conversion exists to a type pointer (i.e. char *). Pointer types are obtained from the type information when an object file is loaded and cannot be created later on.
- Function calls in the form printf("hello\n") refer to dScope functions. No functions can be invoked from the loaded object program because of this reason.
- Structure assignments are not supported as in C.

    Pointers to structures and union's can be differentiated with the content operator '*'. For this reason, the contents of a complete structure or array can be output by using, for example, the "OBJect" command or in watch expressions. In the C language, these types of operations are not possible:

    Example:

    A C program contains the following structure declaration:
    ```
    struct time { char hour; char min; char sec; } time, *ptime;
    ```

    Use within dScope:
    ```
    obj *ptime          /* Output the structure that points to the ptime */
    ```

    If the "*" operator is used on a structure pointer, it is ignored when the expression is not used in an "OBJ" command or when it is not used as a watchpoint expression. In this case, only the pointer value is output:

    ```
    *ptime              /* Outputs only the pointer value */
    OBJ *ptime          /* Outputs structure */
    WS *ptime           /* Define watchpoint, structure output */
    ```

Examples with Expressions

A number of examples with expression are summarized below.   The entries are represented in bold print, the outputs are displayed in normal print.

Preparation:


```
>LOAD 80517.DLL
80C517/80C537 PERIPHERALS for dScope for Windows V1.0

>LOAD C:\C51\EXAMPLES\MEASURE\MEASURE
```


Examples:
```
>0x1234                             /* Simple constant */
0x1234                             /* Output */
>EVAL 0x1234
4660T 11064Q 1234H '...4'          /* Output in several number bases */
>ACC                               /* Interrogate value of the accum */
D:0xE0 = 0x00                      /* Address from ACC = 0xE0, mem type = D:  */
>ACC = --R7                        /* Set accum and R7 equal to value R7-1  */
>ACC,R7                            /* Interrogation, comma exp.  are valid */
D:0xE0 = 0xFF                      /* Output for ACC */
D:0x7 = 0xFF                       /* Output for R7 */
                                   /* String constant within printf() */
>printf ("dScope is coming now!\n")
dScope is coming now!              /* Output */
>main                              /* Get addr of main() from MEASURE.C */
0xFF01EF                           /* Reply, means C:0x01EF */
>&main                             /* Same as before */
0xFF01EF
>d main                            /* Display: address = main, mem type = C: */
FF:0231    75 98 5A D2 DF 43 87-80 75 8C 06 75 8A 06 43 u.Z.C.u.u.C
FF:0240  89 02 D2 8C D2 A9 D2 AF-12 01 F3 75 4C 05 75 4D ......uL.uM
FF:0250  04 75 4E FD 12 14 17 75-4C 05 75 4D 04 75 4E 57 uN.uL.uM.uNW
FF:0260  12 14 17 75 40 69 75 41-0F 12 0A D1 E4 F5 3D 74 .u@iuA...=t
>main + 0x10                       /* Address calculation */
0x241
>uchar (C:0x1EF)                   /* Read byte from code addr 0x01EF */
0x75                               /* Reply */
>dir \measure\main                 /*Output sym from main() as mod MEASURE */
FUNCTION:  MAIN                    /* Output */
  I:0x000067H .  .  .  . cmdbuf .  .  array[15] of char
  D:0x00003CH .  .  .  . i .  .  uchar
  D:0x00003DH .  .  .  . idx .  .  uint
>$ = main                          /* Set program counter to main()  */
>dir                               /* points to local mem sym.  from main() */
FUNCTION:  MAIN                    /* Output */
  I:0x000067H .  .  .  . cmdbuf .  .  array[15] of char
  D:0x00003CH .  .  .  . i .  .  uchar
  D:0x00003DH .  .  .  . idx .  .  uint
>cmdbuf                            /* Interrogate address from cmdbuf */
I:0x0067                           /* Output of addr due to aggr type (Array)*/
>cmdbuf[0]                         /* Output cont.  of first array element */
0x00
>OBJ cmdbuf                        /* Output the contents of entire array */
"\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"
>i                                 /* Output contents from i  */
0x00
>idx                               /* Output contents from idx  */
0x0000
>idx = $                           /* Set contents from index equal to the PC*/
>idx                               /* Output contents from idx  */
0x01EF
```

```
>\211                                  /* Address of the line number #104 */
0xFF01EF                               /* Reply */
>main                                  /* \211 is like main */
0xFF01EF
>
>`BD                                   /* BD bit literalized, else conflict with */
                                       /* BreakpointDisable ! */
0                                      /* Reply */
>PCON                                  /* Output contents of the PCON register  */
0x00
>\MCOMMAND#91                          /* A line number of module "MCOMMAND" */
0xFF08CB
>set_interval                          /* Output a value of the PUBLIC variable */
0xFF08CB
>--ACC                                 /* Auto-decrement also for 8051 registers */
0xFE
>mdisplay                              /* Output a PUBLIC bit variable */
0
>mdisplay = 1                          /* Change */
>mdisplay                              /* Check result */
1
>ACC == 0xFE ? R7 :  $                 /* Conditional operation */
0xFF                                   /* Condition "ACC==0xFE" is true */
>--ACC
0xFD
>ACC == 0xFE ? R7 :  $                 /* Condition no longer true --ACC */
0x231
>ACC == 0xFD && R7 == 0xFF && $ == main     /* Logical expression */
0x01                                   /* 0x01 = TRUE, 0x00 = FALSE */
>save_record[0]                        /* Address of a record */
X:0x4000
>obj save_record[0]                    /* Output complete record contents */
{time={hour=0x00,min=0x00,sec=0x00,msec=0x0},
port4=0x00,port5=0x00,analog={0x00,0x00,0x00,0x00}}
>
>save_record[0].time.hour = R7         /*Change struct element of records
change */
>save_record[0].time.hour              /* Interrogation */
0xFF
>save_record[1].time.hour = save_record[0].time.hour
>save_record[0].time.hour, save_record[1].time.hour
0xFF
0xFF
>menu                                  /* Address of an array */
0x04FE
>
>menu [0], menu[1], menu[2], menu[3]  /* Output array elements*/
0x0A
0x2B
0x2A
0x2A
>
>ACC /= 2                              /* C short form for  "ACCU = ACCU / 2" */
0x7E                                   /* Output */
>ACC /= 2
0x3F
>ACC /= 2
0x1F>(!R0)                             /* Parenthesis , else DOS cmd! */
0x01
>DPTR--                                /* C short form for  "DPTR = DPTR - 1" */
0x0
>++DPTR                                /* C short form for  "DPTR = DPTR + 1" */
0xFFFF
>ACC << 3                              /* ACC SHL 3 */
0xF8
>printf ("RegBank = %d\n", (PSW & 0x18) >> 3)
RegBank = 0
>PSW |= 0x18                           /* Adjust register bank */
0x19
>printf ("RegBank = %d\n", (PSW & 0x18) >> 3)
RegBank = 3
```

```
>--\measure\main\idx                    /* Auto INC/DEC valid for qual symbol */
0x01EF
>++\measure\main\idx
0x01F0
>++\measure\main\idx
0x01F1
>A <<= B + 2                            /* C short form for  "A = A SHL (B+2)" */
0x7C
>intcycle
0x00
>intcycle |= ~1                         /* C short for */
0xFE                                    /*"intcycle = intcycle OR NOT 1"*/
>intcycle
0xFE
>                                       /*example useful in DS51 function*/
>interval.min = getint ("enter integer:  ")
enter integer:  21
>interval.min                           /* Interrogation */
0x15
>obj interval                           /* Output structure contents */
{min=0x15,sec=0x00,msec=0x0}
>obj interval,10t                       /* Output struct contents, decimal base */
{min=21,sec=0,msec=0}
>                                       /* Change structure elements */
>interval.min = interval.sec = interval.msec = A / 2
>obj interval                           /* Control */
{min=0x3E,sec=0x3E,msec=0x3E}
```

Overview

The dScope operating commands are described in this section.   The commands are organized into the following sections:

- <u>Display and Change Memory Commands</u>

    This section comprises the commands to display and change memory contents in various data formats.

- <u>Program Execution Commands</u>

    Concerns the commands "Go" and "TraceStep" and the Performance Analyzer commands.   These are used to execute a loaded program and for analyzing function related execution times and invocation counts.

- <u>Breakpoint Commands</u>

    Breakpoints are used for the debugging of programs in which the program execution is halted at certain addresses, or under certain conditions.   For the definition and management of breakpoints, a series of commands are available.

- <u>General Commands</u>

    This group of commands comprises "supported" commands such as loading a program or peripheral driver, displaying symbols or lines, and others.

Display and Change Memory Commands

ASM Command

Syntax:

ASM [address]   Show current assemble address if address parameter is missing, otherwise set assemble address to given address.

ASM instruction   increased by the number of the Assemble given instruction and store resulting opcodes in memory starting at the current assemble address.   The assemble address is then automatically opcodes created.

The "ASM" command is used to display or set the current assembly address and for assembly of CPU instructions.   When instructions are assembled, the contents of the code memory is changed.

The inline assembler accepts valid assembler mnemonics.   The set of valid mnemonics depend on the CPU driver currently loaded.   If a MCS 51 driver such as 8051FX.DLL or 80517.DLL is loaded, the inline assembler accepts MCS 51 mnemonics.   If a MCS 251 driver such as 80251.DLL is loaded the inline assembler accepts a superset of the MCS 51 instructions, that is, all MCS 51 instructions plus the extended MCS 251 instructions.

Example:
```
>ASM C:0x0000        /* set assemble address to C:0x0000 */
>ASM mov a,#12
>ASM mov r0,#0x20
>ASM movx @r0,A
>ASM inc r0
>ASM movx @r0,A
>ASM jmp C:0x8000
>ASM C:0020H         /* set assemble address to C:0x0020 */
>ASM CLR A
```

ASSIGN Command

<u>**Syntax:**</u>

ASSIGN   Show current assemble address if address parameter is missing, otherwise set assemble address to given address.

ASSIGN channel <inreg >outreg

increased by the number of the Assemble given instruction and store resulting opcodes in memory starting at the current assemble address.   The assemble address is then automatically opcodes created.

The "ASSIGN" command is used to assign the serial interface(s).   The specification "channel" is WIN for the SERIAL window.

The names for "inreg" and "outreg" are defined by the CPU driver.   As long as no CPU driver was loaded, an "ASSIGN" command is not possible (and makes no sense).   A loaded CPU driver routinely assigns the serial interface of the 8051 (two for 80C517) to the SERIAL window. Note that this is true for ony CPU driver.

The names of the serial interfaces can be determined using the command "DIR VTREG".

Example:

```
>LOAD 80517.DLL            /* In order for  S0IN and S0OUT can be known */
>ASSIGN                    /* Interrogate assignment */
 WIN: <S0IN >S0OUT
>
>ASSIGN WIN <S0IN >S0OUT   /* Assign S0IN/S0OUT to the SERIAL window */
>ASSIGN
 WIN: <S0IN >S0OUT
```

Note that the names of the serial I/O registers depend on the CPU being simulated.   For the 80517, they are S0IN and S0OUT, for the 8051/52 the names are SIN and SOUT.   Use the command 'DIR VTREG' to get the names for the serial I/O registers.

DEFINE Command

Syntax:

DEFINE <type> <identifier>   Define a symbol of type <type> and name <identifier>.

The "DEFINE" command is used to create dScope symbols.   Such symbols serve the purpose of value holders, for example to receive the return value of a dScope function.   In general, DEFINE symbols do not refer to any of the memory spaces, they are just symbolic names for values of given type.   A symbol created by DEFINE can be used just like any other public symbol.   The only restriction to program symbols is that they don't have a memory space.

The <type> parameter can be one of the following types shown:

```
BIT                     /* Type bit */
CHAR                    /* Type signed char */
UCHAR or BYTE           /* Type unsigned char */
INT                     /* Type signed int */
UINT or WORD            /* Type unsigned int */
LONG                    /* Type signed long */
ULONG or DWORD          /* Type unsigned long */
FLOAT or REAL           /* Type float */
```

The <identifier> parameter defines the name of the symbols.   It must conform to the rules for variables or symbols.

Examples:

```
>DEFINE BYTE TmpByte        /* define TmpByte to be a byte value */
>DEFINE FLOAT TmpFloat      /* define TmpFloat to be a float value */
>TmpFloat = 3.14159         /* give it a value */
>TmpFloat                   /* display the value of TmpFloat */
```

DISPLAY Command

Syntax:

D [startaddr [, endaddr.]]   Output memory in byte format

The "Display" command is used to output a memory area in bytes.   If the address specifications are omitted, continuation is resumed in the memory type and the address where a prior "Display" command was ended.   The address parameter for the start address, if specified, must contain a unique memory space in order for one of the different address spaces to be selected.

An output line of the Display command consists of the leading address, that is the address of the first byte, a maximum of 16 hex bytes and a maximum of 16 ASCII bytes.   Non-printable ASCII values are printed as dots.

When a start address as well as an end address were specified, the specified area is output according to current size of the COMMAND window.   If the MEMORY window is visible at the time the Display command is entered, the output will be directed to the MEMORY window rather than the COMMAND   window.

Example:
```
>D main                                 /* Output beginning at address "main"*/
>D X:0,0x100                            /* Output 256 bytes of the external data */
>                                       /*memory beginning at address 0 */
>D menu                                 /* Output beginning at address "menu" */
>D save_record, save_record + 0x2F
01:4000  62 62 62 62 62 62 62 62-62 62 62 62 62 62 62 62  bbbbbbbbbbbbbbbb
01:4010  62 62 62 62 62 62 62 62-62 62 62 62 62 62 62 62  bbbbbbbbbbbbbbbb
01:4020  62 62 62 62 62 62 00 00-00 00 00 00 00 00 00 00  bbbbbb..........
```

The display shows the segment value separated by a colon.   The segment values correspond to the memory spaces as follows:

0x00	maps to data segment starting at 0x00:0x0000 ... 0x00:0x00FF (0x00:0xFFFF on MCS-251)
0x01	maps to default xdata segment (0x01:0x0000...0x01:0xFFFF)
0x80...0x9F	maps to banked code segments (0x80 is bank-0, 0x81 is bank-1,...)
0xFF	maps to the default code memory of the MCS 51 and MCS 251 (0xFF:0000...0xFF:0xFFFF).

ENTER Command

Syntax:

E <type> address = expr [, expr [, ...]]   Enter values of given type into memory

The "Enter" command is used to interactively change memory contents.   The data type is given as the first parameter to the Enter command.   It may be one of the following types:

```
BIT             /* Type bit */
CHAR            /* Type signed char */
UCHAR or BYTE   /* Type unsigned char */
INT             /* Type signed int */
UINT or WORD    /* Type unsigned int */
LONG            /* Type signed long */
ULONG or DWORD  /* Type unsigned long */
FLOAT or REAL   /* Type float */
PTR             /* Type C51 generic pointer */
```

The second parameter is the target location of the value of the expression preceded by the assignment operator.   After the assignment operator, one or more comma separated expressions may follow.   The value of each expression is converted to fit the requested Enter type and finally stored in consecutive addresses.

Example:

```
>E CHAR x:0 = 1,2,"-dScope-"          /* Enter Character */
>E FLOAT x:0x2000 = 3,4,15.2,0.33     /* Enter Float */
>EP x:0x1000 = main,timer0, &current  /* Enter Pointer */
>
```

MAP Command

<u>**Syntax:**</u>

    RESET MAP   Reset Map assignments.
    MAP startaddr, endaddr [READ WRITE EXEC] [VNM]   Map memory.

Using the "MAP" command, the memory for the given address range is allocated.   The memory to be mapped may receive additional flags such as READ, WRITE or EXEC or any combination of them.

dScope's memory map feature supports one byte granularity.   This means, on the extreme, each byte of 16M byte range can be mapped with different access permission flags (assuming your computer has sufficient memory to map the whole range).

The VNM option identifies the specified memory area as memory type "von Neumann".   This causes an intentional overlapping of external data memory and code memory of the CPU. Because of this, write accesses to external data memory also change the code memory.   Note that memory ranges with the VNM flag must not cross a 64K boundary and the range must not be a range from the code segment, for example 0xFF8000,0xFFFFFF.

After invocation of dScope, the following memory areas are mapped with the following access permissions by default:

        0x000000 - 0x00FFFF   read write (the MCS 51/251 DATA space)
        0x010000 - 0x01FFFF   read write (the MCS 51/251 XDATA space)
        0xFF0000 - 0xFFFFFF   exec read   (the MCS 51/251 CODE space)

dScope supports up to 16M bytes of memory available for user programs, that is 256 segments of 64K bytes each.   The default MCS 51 and MCS 251 memory spaces are assigned by dScope to the segments with the numbers listed in the following table:

If a user program is loaded into dScope, segments will be mapped as required by the the user program.   This is true for both banked and non banked applications.   If MCS 51 user programs are loaded,   memory mapping commands are almost never required except for the special cases where the access permissions of a specific memory range is changed to catch illegal writes to some location.   The same is true for MCS 251 user programs with the exception of dynamic memory pools where dScope does not know about.

Although dScope supports 16M byte of user program memory, only the memory ranges required by the user program should be mapped, if mapping is performed by explicit map commands. Depending on the amount of memory available to dScope, mapping huge amounts of memory may slow down the execution speed of dScope, since a lot of disk swapping may take place. Any block of memory is allocated twice: the first block is the segment actually used for read, write and execute, the second block holds the specific attributes such as access permissions and information for code coverage and performance analysis.

The **RESET MAP** command clears all mapped segments and restores the mapping to the default state, that is, the default mapping after invoking dScope.   Note that a user program as well as the complete debug information (symbols, lines and types) is killed after performing RESET MAP.

Examples:

```
MAP 0x10000,0x1FFFF read write        /* Enable 64K XDATA RAM */
RESET MAP                             /* Reset MAP */
MAP 0xFF0000,0xFFFFFF exec read       /* Enable default code segment */
MAP 0x1800,0x1FFFF read write VNM     /* Enable "von Neumann" memory type */
```

OBJECT Command

<u>**Syntax:**</u>

OBJ expression [, numberbase] [LINE]   Output contents of an object

The "OBJect" command is used to output complete structures or array contents (aggregates). In the case of simple scalars, the command has no effect; the output is the same as when "OBJ" is not specified before the expression.   The optional specification of the number base determines the number base for outputting numeric values.   It must be either a number base of 16 (hexadecimal) or 10 (decimal).   The LINE parameter forces the object to be output in single line mode with the output truncated at 128 characters.   If the LINE parameter is omitted, then multiline is used by default.

NOTE   Complex data types in expressions are only permissible when appropriate type information exists in the loaded program.   Actually, this information is only produced by the C51 compiler when the module is translated with the options DEBUG and OBJECTEXTEND.   If the type information is omitted, an error message is issued.

Example:
```
>OBJ current LINE       /* single line mode */
{time={hour=0x00,min=0x00,sec=0x00,msec=0x0000},port4=0x00, ...
>
>OBJ current,10         /* multiline, radix 10 */
{
  time={                /* a nested structure */
    hour=0,             /* the structure members ... */
    min=0,
    sec=0,
    msec=0
  },
  port4=0,              /* a scalar */
  port5=0,              /* another scalar */
  analog=               /* an array ...*/
    [0]:  0
    [1]:  0
    [2]:  0
    [3]:  0
}
```

The examples have been created using the MEASURE sample application.

UNASSEMBLE Command

Syntax:

U [address]   Disassemble code memory

The "Unassemble" command is used to disassemble code memory.   The output occurs here in the DEBUG window.   The displayed depends either on the selected mode (High Level, Mixed, Assembler), or on the maximum possible mode of the disassembled area:

◆ High Level Language Mode

> The original statements from a source or list   file are output.   If this mode is no longer possible, because an assembler module with no line information available is disassembled, dScope automatically switches to the assembler mode. dScope always attempts to perform the output in high level mode.

◆ Mixed Mode

> In mixed mode, the original statements from a source or list file as well as the statements of the resulting assembler instructions are output.   If the high-level language output is no longer possible, dScope automatically switches to the assembler mode.

◆ Assembler Mode

> In assembler mode, only the assembler instructions are output.   The operands are symbolically output, whenever possible.
> The output of high-level language statements requires a module that contains line number information and also the associated source or listing file can be accessed.

Example:
```
U main          /* Disassemble starting at address "main" */
U               /* Continue where the previous U stopped */
U C:0x0         /* Disassemble from address C:0x0000 */
```

Watchpoint Kill Command

Syntax:

WK number [, number [, ...]]   Delete specified watchpoints
WK *   Delete all watchpoints

The "WatchpointKill" command is used to delete watchpoint definitions.   The number of the watchpoints to be deleted must be specified as a parameter.   The wildcard parameter "*" causes all watchpoints to be deleted.   The WATCH window is automatically reduced to the required size, or removed when no more active watchpoints are defined.

Example:

```
WK 0,2          /* Delete watchpoints 0 and 2 */
WK *            /* Delete all watchpoints */
```

Watchpoint Set Command

Syntax:

WS expression [, numberbase] [LINE]   Define a watch expression

The "WatchpointSet" command is used to define watch expressions.   These are displayed in the WATCH window together with their results.   The expressions can contain optional operands and operators.   After each "STEP" and "Go" command, the current results of the watch expressions are output.   The optional specification of the number base determines the output base for numeric values.   It must either be a number base of 16 hexadecimal output or 10 for decimal output.   The LINE parameter is optional.   It forces the watch expression to be displayed in single line mode, that is, all values displayed on a single line.   Depending on the amount of output, a line may get truncated after 128 characters.   For single scalar values, the LINE parameter has no meaning.   If the LINE parameter is omitted, then multiline output is used in the WATCH window.

Aggregates are allowed as expressions.   These are structures, arrays, and pointers to structures.   The result of such an expression is always the contents of the aggregate.   In the case of structure pointers, the access operator (*) must be added as a prefix for outputting the complete structure contents.

The following objects are declared in a C program:

```
struct time  { char hour;
               char  min;
               char  sec; } time, *ptime;
```

The command "WS time LINE" produces the following output:

```
00:  time:  {hour=0x00,min=0x00,sec=0x00}
```

The command "WS *ptime LINE" produces the following output:

```
01:  *ptime:  {hour=0x00,min=0x00,sec=0x00}
```

If the specification of the operator * is omitted, the pointer value from ptime is merely output and not the object which ptime references.

The command "WS ptime->hour" produces the following output:

```
02:  ptime->hour:  0x00
```

Each definition of a watchpoint contains an internal reference number.   The reference number is intended to delete a watchpoint in the "WatchpointKill" command.   The output contains the

reference number and the original text of the watchpoint definition.   Afterwards, the result is displayed.   An output line is truncated to 128 characters.   This is to be considered when arrays with many elements or structures with many members are output in LINE mode.

Example:
```
>WS interval,0x0a LINE
>WS save_record[0].analog
>WS save_record[0]
>WS sindex
```

Program Execution Commands

GO Command

<u>**Syntax:**</u>

     G [startaddr.] [, stopaddr.]   Start program execution

The "Go" command starts a program execution.   If the specification of the start address is omitted, the current state of the program counter ($) is used as the default start address.   If the stop address is omitted, the program execution is either stopped when a breakpoint is reached, or by pressing the <u>**Stop**</u> button in the DEBUG window.

When conditional breakpoints are defined, the program run is internally executed in signal step mode, despite Go.   This is because after each step, the break condition(s) must be checked.

After the program execution is aborted, various screen windows (REGISTER, WATCH, DEBUG and others) are updated.

The specification of the start address is generally not necessary after the current program counter ($) is used as the start address.   The specification of the stop address defines a temporary breakpoint which is deleted again when reached.

Example:

```
>G,main      /* Run starting at $ up to address "main" */
>G           /* Start at $.  Break with Ctrl+C or breakpoint */
```

Trace Step Command

      T [expression]   Execute one step or "expression" number of steps
      P [expression]   Execute one pstep or "expression" number of psteps

The "TraceStep" and "P" command are used to execute one or several program steps.

The difference between "T" and "P" is in the fact that "P" considers CALL instructions to be one step.   Therefore, a subprogram is executed by means of the "Go" command.   In comparison, the command "TraceStep" continues to execute the single step mode even in the subroutines.

The definition of a program step depends on the current display mode in the DEBUG window, where the following display modes are possible:

◆ Assembler Display Mode:

      In assembler display mode, the "T" step corresponds to an assembler instruction. A "P" step also includes the "CALL" instructions, and thereby subroutines, as one step.

◆ Mixed Display Mode:

      In the mixed display mode, the same applies as for assembler display mode.

◆ High Level Display Mode:

      In the high level display mode, a step corresponds to a program statement.   This can be several CPU instructions.   If a statement contains a call of another function, then a T step causes the invoked function to be stopped, whereas a P step considers function calls as one step analog to the assembler display mode.

Example:

```
>T 100        /* Execute 100 steps */
>P            /* Execute a step, ignore CALLs */
```

Performance Analyzer (PA) Commands

    PA   Display currently defined PA ranges
    PA KILL *   Kill all currently defined PA ranges
    PA KILL item [, item [,...]]   Kill one or more PA ranges
    PA start [, end]   Define a new PA range
    PA RESET   Reset performance analyzer

The Performance Analyzer commands (PA commandsfor short) are intended for definition or removal of PA ranges.   A PA range is an address range, normally an address where some function starts in code memory and the address of the last opcode belongs to that function. When dScope executes the user program, the values on time consumption and invocation count are collected for each PA range.   The results are displayed in the Performance Analyzer window.   The output from the PA commands however, are directed to the COMMAND window.

The PA commands and their functions are as follows:

- **PA:**

    displays the currently define PA address ranges along with the collected min/max execution times and the invocation counts.

- PA KILL *

    Kills all currently existing PA ranges.

- PA KILL item_list

    Kills the PA ranges given in item_list.   dScope assignes a number to each PA range on definition.   The number is the reference handle to a specific PA range.

- PA start [, end]

    Create a new PA range starting at address 'start'.   The second parameter, 'end' specifies the ending address for the address range.   It can be omitted however, if dScope can find out the ending address by itself.   This is the case, if 'start' specifies a function type expression where dScope can figure out the length of the function.   An address range must specify a range with a unique entry and exit point.   Intermediate return instructions within the range are not permitted. PA ranges also must be distinct, that is, a new PA range must not overlap an already existing range. If it does, an error message will be displayed.   Note that an application should be compiled with full debug information (C51: DEBUG OBJECTEXTEND, A51: DEBUG LINES, C251: DEBUG, A251: DEBUG LINES).

- PA RESET

Clears the recorded execution time and invocation count of all currently defined PA ranges.

Examples:

```
>PA main              /* define a PA range for main() */
>PA timer0            /* and for timer0() ... */
>PA clear_records     /* and some more ... */
>PA measure_display
>PA save_current_measurements
>PA read_index
>PA set_time
>PA set_interval
>
>PA                        /* display all PA ranges */
  0: main: (FF01EF-FF03B6)    /* means C:0x01EF-C:0x03B6 */
  1: timer0: (FF006A-FF0135)
  2: clear_records: (FF01C0-FF01EE)
  3: measure_display: (FF07E7-FF084A)
  4: save_current_measurements: (FF000E-FF0069)
  5: read_index: (FF0136-FF01BF)
  6: set_time: (FF084B-FF08CA)
  7: set_interval: (FF08CB-FF09A5)
/* After execution of the user program ... */
>PA   /* display all PA ranges and the recorded information */
  0: main: (FF01EF-FF03B6)
     count=1, min=-1, max=0, total=167589
  1: timer0: (FF006A-FF0135)
     count=2828, min=33, max=254, total=226651
  2: clear_records: (FF01C0-FF01EE)
     count=1, min=27086, max=27086, total=27086
  3: measure_display: (FF07E7-FF084A)
     count=10, min=19495, max=19503, total=185027
  4: save_current_measurements: (FF000E-FF0069)
     count=491, min=205, max=209, total=100665
  5: read_index: (FF0136-FF01BF)
  6: set_time: (FF084B-FF08CA)
  7: set_interval: (FF08CB-FF09A5)
>
>PA KILL 7           /* Kill a few PA ranges */
>PA KILL 6
>PA KILL 5
>PA
  0: main: (FF01EF-FF03B6)
     count=1, min=-1, max=0, total=167589
  1: timer0: (FF006A-FF0135)
     count=2828, min=33, max=254, total=226651
  2: clear_records: (FF01C0-FF01EE)
     count=1, min=27086, max=27086, total=27086
  3: measure_display: (FF07E7-FF084A)
     count=10, min=19495, max=19503, total=185027
  4: save_current_measurements: (FF000E-FF0069)
     count=491, min=205, max=209, total=100665
>
>PA RESET      /* clear all recorded information */
>PA
  0: main: (FF01EF-FF03B6)
  1: timer0: (FF006A-FF0135)
  2: clear_records: (FF01C0-FF01EE)
  3: measure_display: (FF07E7-FF084A)
```

Breakpoint Commands

Breakpoint Disable Command

Syntax:

BD number [, number [, ...]]   Disable Breakpoints with the specified numbers
BD *   Disable all breakpoints

Breakpoint definitions can be disabled, using the "BreakpointDisable" command.   The breakpoint definition, however, remains.   A list with the numbers of the breakpoints to be disabled or a wildcard (*) is expected as a   parameter.   The command "BreakpointList" indicates the "disabled" breakpoints.

Example:
```
>BD 0,1 /* Disable breakpoints 0 and 1 */
>BD *   /* Disable all breakpoints */
>BL            /* Show current breakpoints */
 0: (E C: 0xFF01EF) 'main', CNT=1, disabled
 1: (E C: 0xFF006A) 'timer0', CNT=10, disabled
     exec ("MyRegs()")
 2: (C) 'sindex == 8', CNT=1, disabled
 3: (C) 'save_record[5].time.sec > 5', CNT=3, disabled
 4: (A RD 0x000037) 'READ interval.min == 3', CNT=1, disabled
 5: (A WR 0x000034) 'WRITE savefirst==5 && acc==0x12', CNT=1, disabled
```

Breakpoint Enable Command

Syntax:

>**BE number [, number [, ...]]**   Enable Breakpoints with the specified numbers
>**BE ***   Enable all breakpoints

Breakpoint definitions can be enabled, using the "BreakpointEnable" command.   A list with the numbers of the breakpoints to be enabled or a wildcard (*) is expected as a parameter.   The "BreakpointList" command indicates the "enabled" breakpoints.

Example:
```
>BE 0,1         /* Enable breakpoints 0 and 1 */
>BE *           /* Enable all breakpoints */
>BL                   /* Show current breakpoints */
 0: (E C: 0xFF01EF) 'main', CNT=1, enabled
 1: (E C: 0xFF006A) 'timer0', CNT=10, enabled
    exec ("MyRegs()")
 2: (C) 'sindex == 8', CNT=1, enabled
 3: (C) 'save_record[5].time.sec > 5', CNT=3, enabled
 4: (A RD 0x000037) 'READ interval.min == 3', CNT=1, enabled
 5: (A WR 0x000034) 'WRITE savefirst==5 && acc==0x12', CNT=1, enabled
```

Breakpoint Kill Command

BK number [, number [, ...]]   Delete Breakpoints with the specified numbers
BK *   Delete all breakpoints

Breakpoint definitions are deleted, using the "BreakpointKill" command..   A list with the numbers of the breakpoints to be deleted or a wildcard (*) is expected as a parameter.

Example:
```
>BK 0,1          /* Delete breakpoints 0 and 1 */
>BK *            /* Delete all breakpoints */
```

Breakpoint List Command

BL List breakpoint definitions

All currently defined breakpoint definitions are listed, using the "BreakpointList" command. The output is to be interpreted as follows:

- Consecutive Number (0 to 39)

    Each breakpoint contains an internal number that must be used to delete or freeze a certain breakpoint.

- Breakpoint Class

    The breakpoint class indicates the type of breakpoint it concerns:
    E: Execution breakpoint
    The code address is also output; i.e. (A C:0x231)
    C: Conditional breakpoint
    Output of (C).
    A: Access breakpoint
    The access mode and the data address of the breakpoint are output. The access mode can be RD for "READ", WR for "WRITE" or RW for "READWRITE".

- Original Text of the Break Definition

    A copy of the entry text of the break definition is contained in this field for easy identification.

- CNT (count) Output

    This field indicates the current pass counter value of the breakpoint. It always displays one when no value was specified for the definition or when the value already has expired (permanent breakpoint).

- enabled/disabled Field

    This field indicates if the breakpoint definition is enabled or disabled.

- exec Field

    The command string is displayed in the exec field which is executed after a breakpoint is reached.

Examples:
```
>BL                /* Show current breakpoints */
```

```
0:  (E C: 0xFF01EF) 'main', CNT=1, enabled
1:  (E C: 0xFF006A) 'timer0', CNT=10, enabled
     exec ("MyRegs()")
2:  (C) 'sindex == 8', CNT=1, enabled
3:  (C) 'save_record[5].time.sec > 5', CNT=3, enabled
4:  (A RD 0x000037) 'READ interval.min == 3', CNT=1, enabled
5:  (A WR 0x000034) 'WRITE savefirst==5 && acc==0x12', CNT=1, enabled
```

Breakpoint Set Command

<u>Syntax:</u>

BS expr. [, count [, "cmd"]]	Define Execution or Conditional break
BS READ expr. [, count [, "cmd"]]	Define 'READ' access break
BS WRITE expr. [, count [, "cmd"]]	Define 'WRITE' access break
BS READWRITE expr. [,count [, "cmd"]]	Define 'READ & WRITE' access break

Breakpoints are used to halt a program execution at certain code addresses or under certain conditions.   dScope supports up to 40 simultaneously active breakpoints.

Following is a description of the parameters of a Breakpoint Specification:

- ◆ expression

     The expression can be an address specification or an arbitrary expression. dScope determines which class of breakpoints that an expression belongs to through analysis.

- ◆ count

     The parameter "count" is optional and can be an expression.   Determination can be made the number of times the breakpoint specification must be occur at the program runtime before the breakpoint is active.   If the "count" specification is omitted, the value 1 is assumed as default.

- ◆ "cmd"

     Each breakpoint specification can contain a command string as a parameter.   It is executed when the breakpoint is active.   The program execution is NOT stopped when a command string exists.   Instead, it is continued after the command string is executed.   If the program run is to be stopped, the variable _BREAK_ must be set to "1".   If a command string is specified, a count specification must be made prior:
     Correct:
     `BS main, 1, "D X:0x4000"`
     Wrong:
     `BS main, , "D X:0x4000"`

<u>dScope recognizes three type of break specifications:</u>

- ◆ Execution Breaks

     Execution breaks stop the program execution when the specified code address is reached.   The execution speed is not affected when execution breaks are defined.

The user must guarantee that the address specifies an opcode address (e.g. the address of the first byte of an MCS 51/251 instruction).   An execution breakpoint for a certain code address can only be specified once; multiple definitions are not permitted.

- Conditional Breaks

    For conditional breaks, the specified expression is recalculated after the execution of each assembler instruction.   If the result is 0, the program run is continued, otherwise a breakpoint stops the program execution.   This breakpoint type is the most flexible of all, because optional conditions can be calculated by the expression.   On the other hand, the program execution time may slow down considerably, depending on the number and complexity of the breakpoints.

- Access Breaks

    An access break is identified by one of the leading options: READ, WRITE or READWRITE.   Access breakpoints can be used to stop the program execution when certain addresses are accessed, or when certain values are read or written. The speed loss is only minimal.   This is because the expression is evaluated only when the specified access event occurs.   The expression must be reducible to a memory address and memory type.   Some extensions that accord to the following rules are allowed:

    1. The result of the expression must have a unique memory type.   This means that only one name of an object may occur.

    2. Only the operators &, &&, <, <=, >, >=, ==, and != are permitted.   An expression according to rule 1 must exist to the right of one of these operators.   An optional expression can appear to the left of the operator (rule 1 does not apply to an expression right to the operator).   Through the appropriate bracketing, the subexpression furthest left must corresponds to rule 1 above.

Example:

Following declarations in a C program:

```
struct time { char hour; char min; char sec; } time;
int    i0, i1;
```

Use in Qualified Abort Conditions:

```
BS WRITE time.sec                      /* Correct, address + offset */
BS WRITE time.sec + i0                 /* Wrong, mem type not unique */
BS WRITE time.sec == i0                /* Correct */
BS WRITE time.sec != i0*i              /* Correct */
BS WRITE time.sec && (ACC==5 && i1 =! i0)  /* Correct */
```

Example 2 is wrong since the addition of two values does not result in a memory type. The other examples satisfy the rules; they represent a unique address of the subexpression furthest to the left.

dScope analyses the context of the expression and attempts to assign it one of three breakpoint classes:

- When an access mode is specified before the expression (READWRITE, READ or WRITE), the breakpoint specification is assigned the class of the access breakpoints.

- If the expression is a simple address, it is assigned the class of the execution breaks, if no access (READ, ...) exists.

- If the expression is not reducible to an address, it is assigned the class of the conditional breakpoints.

Each breakpoint contains an internal number that is output together with the breaks using the "BreakpointList" command. This number must be used as a reference to freeze or delete of a breakpoint.

Examples:

```
>BS main                           /* Example-1 */
>BS timer0,10,"MyRegs()"           /* Example-2 */
>BS sindex == 8                    /* Example-3 */
>BS save_record[5].time.sec > 5, 3 /* Example-4 */
>BS READ interval.min == 3         /* Example-5 */
>BS WRITE savefirst ==5 && acc == 0x12  /* Example-6 */
```

Example 1: an "execution break" is set to the address of the function "main".

Example 2: an "execution break" is set to the address of the function "timer0". The breakpoint is active only after the 10th call from "timer0". Afterwards, the command string is executed. In this example, it represents an invocation of a dScope function. After the execution of the function, the program processing is continued.

Example 3: conditional break. After each executed assembler instruction of the program, the condition "sindex == 8" is checked. If this condition is true, the breakpoint will fire, otherwise program execution continues.

Example 4: conditional break. The same applies here as for example 3, except that the condition "save_record[5].time.sec > 5" must be recognized three times.

Example 5: access break. The expression "interval.min == 3" is checked if the structure "min" is read-accessed, and when "min" contains the value 3 at the same time.

Example 6:   access break.   The expression is checked if the symbolic address "savefirst" is write-accessed.   The program run is stopped when "savefirst" contains the value 5 and the accumulator "acc" contains the value 0x12 after the write process.

General Commands

DEFINE BUTTON Command

Syntax:

> **DEFINE BUTTON "label", "command"**   **Define a Toolbox command button**

This command is used to create a command button which is then added to the Toolbox window.

Both parameters to the command are required to be C style strings.   The first parameter "label" defines the name displayed as the button label.   The second parameter "command"   must be a valid dScope command which is executed when the button is pressed.

Examples:

```
>DEFINE BUTTON "clr dptr", "dptr=0"
>DEFINE BUTTON "show main()", "u main"
>DEFINE BUTTON "show r7", "printf (\"R7=%02XH\n\",R7)"
```

> **NOTE** the second parameter to the last example reading -
> "printf   (**\"**R7=%02XH\n**\"**,R7)"
> introduces nested strings.   Since dScope's printf command requires a format string for it's first parameter and the whole command must be string, strings are getting nested.   The double quote characters of the nested string must be escaped - \" - in order to avoid   syntax errors.

All button commands created are saved in the dScope's INI file on exit from dScope.   The next time dScope is invoked, the button commands are automatically restored and are available in the Toolbox again.

DIR Command

Syntax:

DIR PUBLIC	show public names
DIR VTREG	show names of virtual registers
DIR DEFSYM	show all created with 'define <type><name'
DIR [\module\func]	show symbols of the current or specified scope
DIR [\module] LINE	show line number of current or specified scope
DIR \module	show symbols of specified module
DIR \module\function	show symbols of specified scope
DIR FUNC	show names of dScope functions
DIR UFUNC	show names of user defined dScope functions
DIR BFUNC	show names of all functions (user + predefined)
DIR SIGNAL	show names of all signal functions

The "DIR" command is used to output symbol names of various types.   If "DIR" is invoked without an additional specification, the symbol names of the current valid area are output.

The current module is that module whose address space is shown by the program counter ($). dScope determines the address areas assigned to the module during the loading of an OMF-51/251 object file.   This achieves an independent, automatic selection of the currently valid symbol names and line numbers from the state of the program counter.   dScope maintains various internal symbol tables whose contents can be output by command options to the "DIR" command:

- ◆ PUBLIC

   Output of all global symbol names.   These are each object that have the attribute "PUBLIC" in A51 and PL/M-51, or not the attribute "STATIC" in C51.

- ◆ VTREG

   Using this option, the output of the PIN register names is permitted.   These are, however, only available when an IOF driver was previously loaded.   The symbols and the number of symbols that are defined depends on the type of the IOF driver.

- ◆ DEFSYM

   Using this option, the output of the symbols created by the '*DEFINE <type><name>*'   command is permitted.

- ◆ LINE or \module LINE

   The line numbers of the current or specified module are output.

- ◆ module\func

    The symbols of the specified function are output.   The function must exist in the specified module.

- ◆ FUNC

    The names and the prototype of all currently defined dScope functions are output.   This includes predefined functions, user-defined functions and signal functions.   This concerns functions that can be defined and invoked within dScope, not functions within a loaded user program.

- ◆ BFUNC

    The names of predefined dScope functions are output.   These functions are always available and cannot be deleted or redefined (i.e.   printf (char *, ...)).

- ◆ SIGNAL

    The names of signal functions are output.   Signal functions are user functions that process in the background.   These are used to produce signal forms for the port inputs.

Example:

The entries are bold in the following examples.   The outputs are only shown in extracts. The examples have been created using the driver 80517.DLL and the sample application MEASURE.

```
>DIR MODULE        /* all module names */
  MEASURE
  MCOMMAND
  GETLINE
  ?C_FPADD
  ?C_FPMUL
  ...
>DIR \MEASURE     /* module 'MEASURE' */
 MODULE: MEASURE
    C:0x000000 . . . . _ICE_DUMMY_  . . uint
     FUNCTION: {CvtB} RANGE: 0xFF03B7-0xFF07E5
      C:0x000000 . . . . _ICE_DUMMY_  . . uint
     FUNCTION: {CvtB} RANGE: 0xFF000B-0xFF000D
      C:0x000000 . . . . _ICE_DUMMY_  . . uint
     FUNCTION: SAVE_CURRENT_MEASUREMENTS RANGE: 0xFF000E-0xFF0069
     FUNCTION: TIMER0 RANGE: 0xFF006A-0xFF0135
      D:0x00000F . . . . i . . uchar
     FUNCTION: _READ_INDEX RANGE: 0xFF0136-0xFF01BF
      D:0x00003F . . . . buffer . . ptr to char
      D:0x000042 . . . . index . . int
      D:0x000007 . . . . args . . uchar
     FUNCTION: CLEAR_RECORDS RANGE: 0xFF01C0-0xFF01EE
      D:0x000006 . . . . idx . . uint
     FUNCTION: MAIN RANGE: 0xFF01EF-0xFF03B6
      I:0x000067 . . . . cmdbuf . . array[15] of char
      D:0x00003C . . . . i . . uchar
      D:0x00003D . . . . idx . . uint
>DIR \MEASURE LINE     /* Lines of module 'MEASURE' */
 MODULE: MEASURE
C:0x000E . . . . #87
```

```
C:0x000E . . . . #88
C:0x003A . . . . #89
C:0x0049 . . . . #90
...
C:0x03B6 . . . . #291
C:0x03B6 . . . . #292
>DIR PUBLIC            /* all PUBLIC symbols */
  B:0x000640 . . . . T2I0 . . bit
  B:0x000641 . . . . T2I1 . . bit
  ...
  D:0x000023 . . . . current . . struct mrec
  C:0x0007CD . . . . ERROR . . array[16] of char
  X:0x004000 . . . . save_record . . array[744] of struct mrec
  C:0x00000E . . . . save_current_measurements . . void-function
  C:0x0001EF . . . . main . . void-function
  C:0x00047E . . . . menu . . array[847] of char
  D:0x000030 . . . . setinterval . . struct interval
  ...
  B:0x000601 . . . . IEX2 . . bit
  B:0x000600 . . . . IADC . . bit
>DIR VTREG            /* Show Pin-Registers and Values */
    PORT0:  uchar, value = 0xFF
    PORT1:  uchar, value = 0xFF
    PORT2:  uchar, value = 0xFF
    PORT3:  uchar, value = 0xFF
    PORT4:  uchar, value = 0xFF
    PORT5:  uchar, value = 0xFF
    PORT6:  uchar, value = 0xFF
    PORT7:  uchar, value = 0x00
    PORT8:  uchar, value = 0x00
     AIN0:  float, value = 0
     AIN1:  float, value = 0
     AIN2:  float, value = 0
     AIN3:  float, value = 0
     AIN4:  float, value = 0
     AIN5:  float, value = 0
     AIN6:  float, value = 0
     AIN7:  float, value = 0
     AIN8:  float, value = 0
     AIN9:  float, value = 0
    AIN10:  float, value = 0
    AIN11:  float, value = 0
     S0IN:   uint, value = 0x0000
    S0OUT:   uint, value = 0x0000
     S1IN:   uint, value = 0x0000
    S1OUT:   uint, value = 0x0000
    VAGND:  float, value = 0
    VAREF:  float, value = 5
     XTAL:  ulong, value = 0xB71B00
   PE_SWD:  uchar, value = 0x00
    STIME:  uchar, value = 0x00
>$ = MAIN     /* set current execution point to main() */
>DIR          /* now, the main() symbols are preselected */
 FUNCTION: MAIN RANGE: 0xFF01EF-0xFF03B6
  I:0x000067 . . . . cmdbuf . . array[15] of char
  D:0x00003C . . . . i . . uchar
  D:0x00003D . . . . idx . . uint
>DIR DEFSYM   /* those created by 'DEFINE <type> <name>' */
            word00:   uint, value = 0x0000
            byte00:  uchar, value = 0x00
           dword00:  ulong, value = 0x0
           float00:  float, value = 0
>DIR FUNC     /* predefined dScope functions */
predef'd:   void  MEMSET (ulong, ulong, uchar)
predef'd:   void  TWATCH (ulong)
predef'd:    int  RAND (uint)
predef'd:  float  GETFLOAT (char *)
predef'd:   long  GETLONG (char *)
predef'd:    int  GETINT (char *)
predef'd:   void  EXEC (char *)
```

```
predef'd:   void  PRINTF (char *, ...)
```

DOS Command

<u>**Syntax:**</u>

!   Open a DOS window and stay there.   The DOS window is closed by entering 'EXIT' at the DOS command level

The exclamation point forces a DOS box to be opened.   Within the DOS box, any dos command can be executed.   The DOS box is closed by entering EXIT at the DOS command level.

Example:
!

The exclamation point is also a valid C operator (logical NOT).   An expression that is not part of the dScope command (because no keyword in front) can therefore not begin with the "!" operator.   If this is necessary, the expression must be placed in parenthesis:

```
!dptr        /* Interpreted as a "open DOS box" command */
(!dptr)      /* Is interpreted as expression and calculated */
```

EVAL Command

<u>**Syntax:**</u>

    EVAL expression   Show result of the expression in four number bases

The "EVAL" command calculates the specified expression and outputs the result in decimal, octal, hexadecimal and, when possible, in ASCII format.   The same expression without preceding "EVAL" outputs only the result in the current number base, set under "RADIX".   The expression can contain several subexpressions, separated by a comma.

Example:

```
>eval -1
16777215T 77777777Q 0xFFFFFF '....'
>eval intcycle
0T 0Q 0x0 '....'
>intcycle = 0x12
>eval intcycle
18T 22Q 0x12 '....'
>eval 'a'+'b'+'c'
294T 446Q 0x126 '...&'
>eval main
16712175T 77600757Q 0xFF01EF '....'
>eval save_record[1].time
81931T 240013Q 0x1400B '..@.'
>eval save_record[1].time.sec
0T 0Q 0x0 '....'
>save_record[1].time.sec = 1
>eval save_record[1].time.sec
1T 1Q 0x1 '....'
>eval save_record[1].time.sec = 0
0T 0Q 0x0 '....'
>
```

EXIT Command

EXIT   Exit dScope

The "EXIT" command causes all open files to be closed, and return is made to Windows.   The "EXIT" command cannot be contained in an INCLUDE file.   Moreover, it is not allowed as an argument of an exec() call.   The EXIT command will be cancelled if dScope is still executing the user program or if a dScope function is still active.   In any case, kill the active function or break out of execution first, then reenter the EXIT command.

INCLUDE Command

INCLUDE [path]filename   Read dScope commands from a file

Using the "INCLUDE" command, the command file is read and passed line by line to dScope for execution.   An INCLUDE file can, on the other hand, contain nested INCLUDE commands.
The maximum nesting level comprises 4 (when no other command levels are executing).   Note that an INCLUDE command requires any Go or Step commands to stopped before INCLUDE is issued, otherwise an error will occur with the INCLUDE command being canceled.

Example:


```
INCLUDE measure.ini
```

KILL Command

<u>**Syntax:**</u>

KILL FUNC *funcname*	Delete a user defined dScope function
KILL FUNC *	Delete all user defined dScope functions
KILL BUTTON number	Remove one or more Toolbox buttons

dScope functions can be deleted using the command "KILL FUNC".   The first form of the "KILL FUNC" command deletes the specified user function.   The second form deletes all user functions.   User functions include signal functions as well as normal functions.   Predefined functions, however, are not included, since they cannot be deleted.   If an active signal function is deleted, it is automatically removed from the list of active signal functions.

The third syntax of the KILL command is used to remove Toolbox buttons.   The parameter to the KILL BUTTON command is the button number.   The button number is displayed in the Toolbox window ahead of the buttons (range 1 ... 16).

Example:

```
>KILL FUNC ANALOG      /* Delete user function "analog" */
>KILL FUNC myregs      /* Delete user function "myregs" */
>KILL FUNC *           /* Delete all user functions */
>KILL BUTTON 3         /* Kill Toolbox Button number 3 */
>KILL BUTTON 1         /* Kill Toolbox Button number 1 */
```

Note: predefined dScope functions such as memset() cannot be killed.

LOAD Command

LOAD [path]filename   Load on object or HEX (HEX386) file.   Alternatively, a CPU driver such as 8051FX.DLL may be loaded.   In this case, a path must not be entered, the driver name sufficies.   dScope loades drivers always from the path, where dScope itself is installed.

The "LOAD" command is used to load a file of the following type:

◆ File in Intel HEX/HEX386 Format

This format is produced by the Object to Hex Converter program and contains no symbolic debugging information.   The testing of user programs is therefore only possible at the assembler level (no display of the source and no type information).

◆ File in Intel Object Format OMF51

This format is produced by the A51 assembler, the C51 compiler and L51 linker/locater.   Files of this type contain complete symbolic debug and type information, and line numbers.   Type-specific high-level language debugging is fully supported by OMF51 files.

◆ File in Intel Object Format OMF-251

This format is produced by the A251 assembler, the C251 compiler and L251 linker/locater.   Files of this type contain complete symbolic debug and type information, and line numbers.   Type-specific high-level language debugging is fully supported by OMF-251 files.

◆ File in Object Format OMF-166 (dScope-166 for Windows)

This format is produced by the 80C166 tools.   Files of this type contain complete symbolic debug and type information, and line numbers.   Type-specific high-level language debugging is fully supported by OMF-166 files.

◆ Driver for Simulation for the Peripheral Components (dScope)

dScope analyzes the file to be loaded using the contents (not the name or the extension) and attempts to assign it one of the classes mentioned above.   If this is not possible, it is an unknown format.   A load procedure is not performed in this case, and an error message is issued.

Loading a File in Intel HEX Format

This file format does not allow symbolic test options at all because of the non-existing debug information.   The Intel HEX format is a general format that can be processed by most PROM programmers.

Example:
```
LOAD MON51.HEX
```

Loading an Object File

The testing of a program at the high language level requires that this program is in OMF-51/251 object format with line number information.   This format is produced by the C51 compiler for further processing by L51.   The following table illustrates which translator is available in the associated debug information order.   If the call option DEBUG is omitted, the translated module generally contains no debug information, as default.

A51	DEBUG	no	no	no
C51	DEBUG	no	yes	yes
	DEBUG OE	yes	yes	yes
PL/M-51	DEBUG	no	yes	yes
A251	DEBUG	yes, for scalar types	yes	yes
C251	DEBUG	yes	yes	yes

NOTES

INCLUDE files from C sources (i.e.   #include <myfile.h>) should not contain statements that produce executable code.   Otherwise, the high-level language display is incorrect.   The reason for this behavior is that a module can only be assigned one source or list file.   Assignments of more than one file are not supported by the object format OMF-51.   For object files generated by the MCS 251 tools, no such restriction exists.

The C51 call option "OE" is an abbreviation for "OBJECTEXTEND".   It is only effective when the option "DEBUG" was also specified.   The C51 compiler should always be operated with the options "DEBUG" and "OBJECTEXTEND" when dScope is subsequently used for testing.

PL/M-51 refers to the Intel PL/M-51 compiler only.

The debug information of a prior "LOAD" command (symbol and line numbers) is deleted.

The output of lines from the source program is always possible when the loaded object contains line number information and when the associated source or list file can be found.   Type-specific testing is only possible for modules of the C51 compiler (when the call options "DEBUG" and "OBJECTEXTEND" were used).   Another advantage is that the names of the C source files

(without path) are stored in the object file and can be clearly known.

In most conventional programs, a combination of C51 and A51 modules often occurs.   The previous specifications each refer to a module, not the entire program.

After the loading, dScope searches for the source or list files in the specified path of the "LOAD" command or in the current path, when a specification is omitted.   Other paths can be defined using the "SET" command.

For PL/M-51 modules, the listing of a translation is used for the high-level language output. Since the module name does not absolutely agree with the file name (without extension), and no reference to the source or list file exists in the object file, the file name of the list file must be assigned to the PL/M-51 module using the "SETMOD" command.   The path can be determined using the command "SET SRC" or can be a part of the SETMOD specification as shown below:

```
SET SRC = C:\P51
SETMOD \PLMMOD PLMMOD.LST
        - or
SETMOD \PLMMOD = C:\P51\PLMMOD.LST
```

Example:
```
LOAD C:\DSW\SAMPL51\MEASURE\MEASURE
```

Loads the program "MEASURE" from the path C:\DSW\SAMPL51\MEASURE.   The C source files are also searched in this path.

Loading and Debugging a banked application

dScope supports loading and debugging of banked applications in OMF-51 format with up to 32 memory banks.   A banked application is loaded as usual with the LOAD command or via the load file dialog.   The object file loader recognizes banked files and maps the different banks into segments as follows:

 ◆ Bank-0 code is loaded at address 0x80:nnnn

 ◆ Bank-1 code is loaded at address 0x81:nnnn

 ◆ Bank-n code is loaded at address 0x80+n:nnnn

Since dScope supports up to 16 MB of memory, it uses segments 0x80 ... 0x9F to store code banks.   The common code is loaded into all active banks, this means it is available in any code bank.   Note that dScope automatically allocates the segments required to store the code banks, you need not map memory before loading the banked application.

Notes and Hints on debugging banked applications

 ◆ dScope extracts the bank switch function symbols out of the object file to find out the bank switch code address entries.   The bank switch function symbols must be named **? B_SWITCHnn** where nn is the bank number.   This compatible to the method used to create banked applications with C51, A51 and BL51.   If dScope executes an address which denotes a bank switch, it will switch to the appropriate segment containing the target code bank.   The code contained in the bank switch function is of no interest to dScope.

NOTE

dScope requires that banked applications are created with the banked linker BL51 V3.5 or newer.   If you have used an older version of BL51, relink with the new version, otherwise, dScope will not handle bank switching correctly.

 ◆ If an address breakpoint is set into the common area of the application, that breakpoint is set in the common area of every present bank.   An address breakpoint somewhere in the banked area is not duplicated into the other banks.

 ◆ Addresses in different banks can be qualified using the double backslash notation, for example **\\3\modx1\funcx1** denotes **funcx1** in module **modx1** contained in **bank 3**.   This is a fully qualified address which can be used whenever an address parameter for a command is required.

 ◆ The DEBUG window changes the output of the address in mixed or asm mode to reflect

the code bank number, not just the plain address.

Load a dScope CPU Driver

The last category of loadable files concerns the CPU drivers.   The CPU drivers are Windows dynamic link libraries or DLL's for short.

The DLL's exist in the path where the dScope executable is located.   The path however depends on the path you selected on installation of dScope for Windows.   In any case, do not enter a path for the driver, simply enter the driver name, for example 8051FX.DLL.   dScope will automatically search for the driver in the dScope executable path.

If no specific CPU driver has been loaded, a standard 8051 can be simulated.   However, none of the on-chip peripherals (timer, ports, A/D converter, etc.) would be active or have any significance.   Use or initialization of any of the control or Special Function Registers (SFR) would also have no affect.   The names of the I/O ports will not be recognized, and communication with the outside world (via the serial port) is not possible.

NOTE   If a CPU driver is loaded when a user program and/or another driver is already loaded, dScope kills all currently available debug information.   This state is comparable to the state immediately after dScope was invoked.

Example:
Load the CPU Driver 80517.DLL
>LOAD 80517.DLL

The driver signs on with the control type and the version:

80C517/80C537 PERIPHERALS for dScope-251/W V1.0

Upon loading, the driver automatically defines specific I/O names and performs an initialization. This makes dScope capable of completely simulating all of the peripherals of the 80517 CPU. The I/O names   and the current values and types can be listed using the command "DIR VTREG".

LOG Command

<u>**Syntax:**</u>

        LOG > [path]filename   Create log file
        LOG >> [path]filename   Append to existing log file or create new
        LOG   Interrogate log status
        LOG OFF   Close log file and deactivate

The "LOG" command is used to create a file which receives the outputs displayed in the COMMAND window.   If the file name is introduced with a right angle, the file is created new, a previously existing file with the same is overwritten.   If the name is introduced with double right angles, the existing file is merely appended.   If the specified file does not exist, then it is created.

The file name can contain a drive and/or a path specification.   File names can also be entered as character strings; i.e.   "c:\usr\tmp\logfile".

Example:
```
LOG >C:\TMP\dslog                /* Create log file */
LOG                              /* Interrogate log status */
 command log file: C:\TMP\dslog  /* Reply */
LOG OFF                          /* Close log file */
```

RESET Command

RESET   perform dScope reset
RESET MAP   Reset dScope's memor ymapping to default state
RESET var   Reset SET variable

The "RESET" command without additional specifications resets dScope.   This is comparable to a processor reset ($=C:0x0000, P0=0xFF etc.).   A loaded program including debug information remains. Possibly active signal functions are, however, deactivated.

"RESET MAP" resets the MAP assignments and kills all currently available debug information as well as a loaded user program.

The third option of the "RESET" command is used to reset the assignment of a SET variable and is described in this section under the "SET/RESET" command.

SAVE Command

SAVE [path]filename addr1,addr2   The address range specified by addr1 up to addr2 is save to filename using HEX386 format.

The "SAVE" command is used to save a memory image to a file in HEX386 format.   Such a file can be loaded by dScope with the LOAD command.   A file in HEX386 format does not contain debug information.

SLOG Command

<u>**Syntax:**</u>

     SLOG > [path]filename Create a log file for serial output
     SLOG >> [path]filename   Append to existing log file or create a new one
     SLOG   Interrogate log status
     SLOG OFF   Close log file and turn of logging

The "SLOG" command is used to create a file which receives the inputs and outputs displayed in the SERIAL window.   If the file name is introduced with a right angle, the file is created, a previously existing file with the same is overwritten.   If the name is introduced with double right angles, the existing file is merely appended.   If the specified file does not exist, then it is created.   The file name can contain a drive and/or a path specification.   File names can also be entered as character strings; i.e.   "c:\usr\tmp\logfile".

Example:
```
SLOG >C:\TMP\dslog                /* Create serial log file */
SLOG                              /* Interrogate slog status */
 serial log file: C:\TMP\dslog    /* Reply */
SLOG OFF                          /* Close serial log file */
```

SCOPE Command

Syntax:

      SCOPE [\module [\function]]   Show address range of scope block

The "SCOPE" command outputs the address assignment of modules and functions of a loaded program.   When a program is loaded in OMF-51 or OMF-251 format with debug information, dScope internally creates a table that contains the address assignment.   The symbolic information is also set with relation to the runtime addresses.   This results in the automatic selection of a symbol table for non-qualified symbol specifications to be performed.

Without the specification of a module and optional function name, the address assignment of all modules is output.   If a module name is specified, only the address assignment of the corresponding module is output.

The indentation of the lines reflect the layout of the source program for a certain module.

Examples:
```
>scope \measure\main /* show scope range of main() */
MAIN RANGE: 0xFF01EF-0xFF03B6
>
>scope \measure      /* show scope ranges for module 'measure' */
MEASURE
  {CvtB} RANGE: 0xFF03B7-0xFF07E5  /* dScope dummy scope block */
  {CvtB} RANGE: 0xFF000B-0xFF000D
  SAVE_CURRENT_MEASUREMENTS RANGE: 0xFF000E-0xFF0069
  TIMER0 RANGE: 0xFF006A-0xFF0135
  _READ_INDEX RANGE: 0xFF0136-0xFF01BF
  CLEAR_RECORDS RANGE: 0xFF01C0-0xFF01EE
  MAIN RANGE: 0xFF01EF-0xFF03B6
>
>scope               /* show all scope ranges */
MEASURE
  {CvtB} RANGE: 0xFF03B7-0xFF07E5
  {CvtB} RANGE: 0xFF000B-0xFF000D
  SAVE_CURRENT_MEASUREMENTS RANGE: 0xFF000E-0xFF0069
  TIMER0 RANGE: 0xFF006A-0xFF0135
  _READ_INDEX RANGE: 0xFF0136-0xFF01BF
  CLEAR_RECORDS RANGE: 0xFF01C0-0xFF01EE
  MAIN RANGE: 0xFF01EF-0xFF03B6
MCOMMAND
  {CvtB} RANGE: 0xFF09A6-0xFF0A23
  MEASURE_DISPLAY RANGE: 0xFF07E7-0xFF084A
  _SET_TIME RANGE: 0xFF084B-0xFF08CA
  _SET_INTERVAL RANGE: 0xFF08CB-0xFF09A5
GETLINE
  _GETLINE RANGE: 0xFF0A24-0xFF0A87
?C_FPADD
?C_FPMUL
?C_FPDIV
?C_FPCMP
...
```

The scope blocks named **{CvtB}** are blocks created by dScope.   Such scope blocks are the result of insufficient debug information available.   This may be the case on modules linked in from libraries without debug information or from Assembly language modules with no debug information.   The same is true for blocks with a valid name but no scope range afterwards.

SET/RESET Command

Syntax:

SET var = "string"	Assign a string to some variable
SET var	Show assignment of a variable
RESET var	Reset variable assignment

The commands "SET var" and "RESET var" are used to assign a string to a predefined variable or to reset the variable.   The following variables are allowed:

◆ SRC

> Path(s) to search for the search of source or list files for high-level language outputs.   SRC is the only variable that contains several assignments (up to 10).

◆ F1 to F9, F11 and F12

> The function keys **F1** to **F9**, **F11** and **F12** (F10 is not available).   Command strings can be assigned to function keys so frequently used commands can be executed with a single keystroke.

Example:
```
>SET F5="LOAD \\OBJS\\80517.IOF"              /* F5 assignment */
>SET F5                                       /* Interrogation */
F5                                            /* Press F5 */
80C517/80C537 PERIPHERALS for dScope-251/W V1.0

>LOAD \OBJS\MEASURE                           /* Load mod.  "MEASURE" */
>SET SRC                                      /* Output of SRC assign*/
\objs                                         /* Load sets SRC path */
>RESET F5                                     /* Reset F5 key */
>SET F5                                       /* F5 interrogation */
F5 =                                          /* Unassigned */
```

NOTE When a string contains an additional string or a backslash (i.e.   for specifications of path and file names), the characters   and \ must be literalized by a preceding backslash. Otherwise, an error message is issued.

Wrong:
```
>SET F5 = "load \objs\measure"
```
Correct:
```
>SET F5 = "load \\objs\\measure"
```

A special case is assignment of a command to the F1 function key.   Since F1 is a public symbol (if a MCS 51 style CPU driver is loaded) designating a bit from the program status word (PSW), using

SET F1   =   "dir"

will lead to a syntax error.   You can avoid this by writing the name of the function key within
double quotes:

                    SET   "F1"   =   "dir"

SETMOD Command

Syntax:

SETMOD \module = hll_file Assign source or list file to given module
SETMOD [\module] Show the current assignment

Using the "SETMOD" command, an assignment between a module of a loaded program and a source or list file is produced.   This, as a rule, is necessary for modules that were created with the PL/M-51 compiler.   For C modules, the module name corresponds to the file name or the C source file without path and file extension.   For PL/M modules, the module is not produced from the file name, but from the first source statement; i.e.   "PLMMODULE: DO;".   In this case, dScope has no possibility to find the PL/M listing without an explicit file assignment.

Example:

An application consists of two modules, a C module with the name **CA.C** and a PL/M module with the name **PLM.P51**.   The high-level language output should also occur for the PL/M module.


```
/*******************/
/* C MODULE "CA.C" */
/*******************/
/* Module name = CA  */
extern alien char PLMFUNC (char x, char y, int z);
char  a,x,c;

main()  {
  a = 2;
  x = 4;
  c = 5;
  c = PLMFUNC (a, x, c);
  if (c == a) --x;
  else        ++a;
  while (1);
}

alien char cafunc (char q1, char q2) { /* Called from PLMMODULE */
  return (q1 + q2 + 2);
}

/*****************************/
/* PL/M-51 MODULE:  "PLM.P51" */
/*****************************/
PLMMODULE:  DO;    /* Module name = "PLMMODULE" */
cafunc:  procedure (q0, q1) byte external;  /* In CA.C */
  declare (q0, q1) byte;
end cafunc;
plmfunc:  procedure (x, y, z) byte public;
  declare (x, y) byte;
  declare z      word;
  z = cafunc (x, y);
  z = z + (x * y);
  return (z);
end plmfunc;
END PLMMODULE;
```

Program Generation:
```
PLM51 PLM.P51 DEBUG CODE
C51 CA.C CODE DEBUG OBJECTEXTEND
L51 CA.OBJ, PLM.OBJ
```

The linked program is stored under the name "CA".

```
/* invoke dScope */
>LOAD CA                                    /* Load program */
>SETMOD                                     /* Output module assignment */
 Module CA, Src/Lst: CA.C
 Module PLMMODULE, Src/Lst: <none>
```

The module "CA" has an assignment of a file name; for module "PLMMODULE".   It is, however, omitted.   Because of this, the output of listing statements is not possible for the module "PLMMODULE".

```
>SETMOD \PLMMODULE=PLM.LST                  /* Perform assignment */
>SETMOD                                     /* Output assignment */
 Module CA, Src/Lst: CA.C
 Module PLMMODULE, Src/Lst: PLM.LST
>U main                                     /*Disassemble main from CA.C*/
>U plmfunc                                  /*Disassemble plmfunc of PLM.P51*/
```

NOTES

For C modules that were created with the C51 compiler, the C source file is used for the high-level language display.   Modules that were translated using the options DEBUG and OBJECTEXTEND contain the names of the source file without the path.

For PL/M modules that were created with the Intel PL/M-51 compiler, a list file is used for the high-level language display.   The use of the PL/M source file is not possible because PL/M processes with statement numbers (C processes with line numbers).   The high-level language output is therefore not correct.

dScope always tries to find a source or list file.   For this purpose, attempts are performed with 1) the path of a preceding "LOAD" command, 2) possible additional paths (SET src+xxx), 3) and the current directory.   The attempts are further performed with the extension .LST and .C when the first attempt with the unchanged name did not succeed.

When a source or list file can be opened, dScope determines by means of the first bytes of the file if it is a list file or not.

High-level display mode is only possible for modules with line numbers.

INCLUDE files of C sources (i.e.   #include <myfile.h>) should not contain statements that produce executable code.   Otherwise, the high-level language display mode is incorrect.   The reason for this behavior is that a module can only be assigned one source or list file.   Assignments of more than one file is not supported by the OMF-51.   For OMF-251, no such restriction exists.

SIGNAL Command

SIGNAL KILL funcname Deactivate given signal function
SIGNAL STATE   Show names of currently active signal functions

The first form of the "SIGNAL" command is used to remove an active signal function from the list of active signal functions.   The name of an active signal function must be specified as a parameter.   Afterwards the signal function is available thereafter for a further call.

Example:
```
>SIGNAL KILL ANALOG0       /* Deactivate signal function "analog0" */
```

Using the command "SIGNAL STATE", the list of the active signal functions is output.   The output has the following format:

```
>SIGNAL STATE
 0  idle     Signal = ANALOG0 (line 10)
```

The leading number is an internal code followed by the state of the signal function.   This can be idle or running.   The name of the signal function follows and the line number of the last executed statement within a signal function.

Introduction

A dScope function is a function defined by the user or a predefined function within dScope that may return a value and may receive actual parameters.

> **Tip: dScope** functions are not to be confused with the functions of a user program. They represent their own class of functions.

The language in which the functions are written is a subset of the C programming language.   All control flow statements such as IF, ELSE, WHILE, DO WHILE and SWITCH on CASE, BREAK, CONTINUE and GOTO exist as in the C language, and they function in an identical manner. The declaration of local scalars is permitted.   The use and understanding of control structures is assumed and no further clarification is given in the following sections.

Procedure for Creating Functions

◆ A function is created using the text editor and is stored in a file for later (re)use.   A file can contain numerous functions.

> A function is typically entered by hand in command mode.   This, as a rule, is not very useful because the function text is not available after the end of the session. Consequently, no changes or corrections can be made.

◆ A file containing functions is passed to dScope for analysis by means of the "INCLUDE" command.

> dScope analyzes the functions using an integrated function compiler and generates an internal intermediate code in order to achieve high execution performance.   If no error occurs during the compilation, the name of the function is added to the internal list of functions.   At this time, the function can be invoked directly from the command line or from within other functions.   If an error occurs during compilation, the function is ignored.   In this case, the erroneous function can be corrected with the text editor and re-submitted to dScope for compilation.

◆ The function compiler performs strict type testing.   For example, the number of parameters, the return type and generally the compatibility of the types is tested.

◆ Functions that are not needed can be deleted.

◆ dScope commands can be executed from within functions by means of the "exec()" function.   The "exec()" function is one of many predefined functions.

Function Classes

dScope utilizes three classes of functions:

◆ User Functions

> User functions can be used to extend the dScope command scope by a number of commands, or to chain frequently used command sequences or recurring expressions in a function.   User functions as well as the signal functions can process the same expressions that are permissible at the dScope command level.   Using the built-in function exec(), dScope commands can be executed even from functions.   Only expressions are allowed within functions.   This is why commands must be passed as string parameters to the "exec()" function.

◆ Predefined Functions (built-ins)

> These function classes are predefined by dScope.   Predefined functions can neither be deleted nor redefined.   Such attempts result in an error message.

◆ Signal Functions

> Signal functions are used to simulate the behavior of the signal generator.   This allows signal forms to be produced.   These signals can, for example, be applied on the input lines of the CPU (virtual).   The maximum time resolution compromises one cycle, or 1us at a nominal 12 MHz clock frequency.   Signal functions run in the background during the program execution and are coupled via dScope's cycle counter.   A maximum of 10 signal functions can be active at the same time.

User Functions

The definition of a function can be made only from the dScope command level:

```
FUNC <return_type> <fname> ( <parameter_list> ) {
  /* Statements */
  }
```

e.g.   for signal functions:

```
SIGNAL VOID <fname> ( <parameter_list> ) {
  /* Statements */
  }
```

A function consists of the following components:

- Keyword "func" or "signal".   This opens a definition.   "func" opens a neutral function; SIGNAL opens a signal function.

- <return_type> is the type of the return value of the function.   If the specification of the type is omitted, the type int (integer) is explicitly assumed.   Return types can be char, int, long, float and void.   The type void indicates that the function returns no value to the caller.   dScope checks for functions with a return type not equal to void to see if at least a return statement with an expression exists.   In the case of void functions, the return statements must not have an expression.   Note that dScope does not catch the cases where some control path returns without having a return expression.

- <fname> is the name of the function.   dScope functions can be invoked only be their names.

- (<parameter_list>) contains the names of the parameters and their types.   The parameter declarations are to be separated by commas when more than one parameter exists.   dScope permits up to eight parameters.   If no parameters are specified, no actual parameters may be passed during the invocation.   This applies the same as with a void parameter list; test() and test(void) are therefore indifferent.

  Parameter lists examples:
  ```
        func void test () { ; }          /* Empty list (void) */
        func void test (void) { ; }      /* Empty list (void) */
        func void test (int pa1) { ; }
        func void test (float fp1, long lp2, int x1) { ; }
  ```

- **{** the open curly bracket.   The function definition is complete when the number of open brackets is balanced with the number of the closing (}) brackets.

NOTE

The first curly brace '{' must be on the same line which starts the function definition.   A syntax error will occur otherwise.   This applies to functions entered in command mode as well as functions defined via include files !

Correct function definition:

```
func void MyFunc (void)  {  /* { on first line */
  /* ... statements */
}
```

Illegal function definition:

```
func void MyFunc (void)  /* { not on first line ! */
{
  /* ... statements */
}
```

User functions may not invoke signal functions or the "twatch()" function.   The value of a local object is undefined as long as no explicit assignment to an object was made.

Example:

Definition of a Function for the output of the register contents.   Switch to DOS, invoke some text editor capable of creating plain ASCII files and edit a file named MYREGS.FNC to have the content as shown:

```
/* Function MyRegs() shows Registers R0...R7 */
FUNC void MyRegs (void)  {
  printf ("-------- MyRegs() --------\n");
  printf (" R0 R1 R2 R3 R4 R5 R6 R7\n");
  printf (" %02X %02X %02X %02X %02X %02X %02X %02X\n",
          R0,  R1,  R2,  R3,  R4,  R5,  R6,  R7);
  printf ("------------------------\n");
}
```

After exiting the editor,   switch back to dScope's COMMAND window and enter:

```
>INCLUDE MYREGS.FNC
```

Depending on the drive and path you used when creating the file, specify the correct path in the INCLUDE command.   The "INCLUDE" command reads the entry file line-by-line; each line is echoed in the COMMAND window.

The function is compiled and its name is entered in the internal table of the functions.   The contents can be listed using the command "**DIR UFUNC**" (UNFUNC stands for *U*ser *FUNC*tion):

```
>DIR UFUNC
user:  void  MyRegs ( void )
```

Using the command "**DIR BFUNC**", the names of all predefined functions are listed (BFUNC stands for **B**uilt-in **FUNC**tion):

```
>DIR BFUNC
predef'd:  void       MEMSET ( ulong, ulong, uchar )
predef'd:  void       TWATCH ( long )
```

```
predef'd:  int RAND ( uint )
predef'd:  uchar      TIMEWAIT ( uint )
predef'd:  void       KEYWAIT ( char * )
predef'd:  float      GETFLOAT ( char * )
predef'd:  long       GETLONG ( char * )
predef'd:  int GETINT ( char * )
predef'd:  void       EXEC ( char * )
predef'd:  void       PRINTF ( char *, ...  )
```

Using the command "**DIR FUNC**", all names of all built-in functions, user functions and signal functions are listed:

```
>DIR FUNC
    user:  void  MYREGS ( void )
predef'd:  void  MEMSET ( ulong, ulong, uchar )
predef'd:  void  TWATCH ( long )
   .
   .
   .
predef'd:  long  GETLONG ( char * )
predef'd:  int   GETINT ( char * )
predef'd:  void  EXEC ( char * )
predef'd:  void  PRINTF ( char *, ...  )
>
```

The function "MyRegs()" can be invoked directly from the COMMAND window:
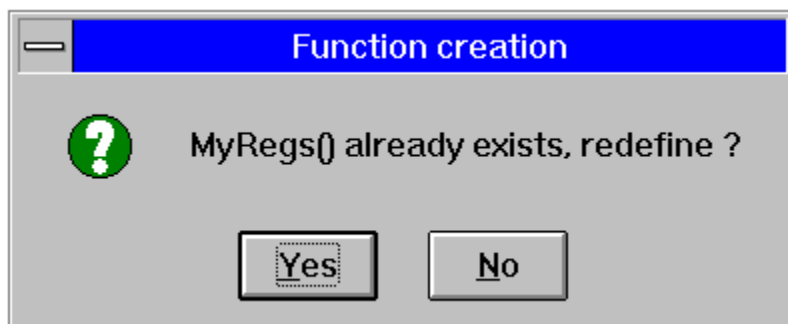
```
>MyRegs()
```

The output of the function "MyRegs()" is as follows:

```
----------------- MyRegs() --------------------
 R0=00 R1=00 R2=00 R3=00 R4=00 R5=00 R6=00 R7=00
-----------------------------------------------
```

If the function 'MyRegs()' is redefined later on for example by re-including the file MYREGS.FNC, a message box will be displayed which queries for either defining or ignoring the new function definition:

The function "MyRegs()" may be removed by the command:

```
>KILL FUNC myreg
```

Predefined Functions

dScope provides a series of functions that are always defined and can neither be redefined by the user nor deleted.   These are "support" functions for writing individual dScope functions. The following table lists the predefined functions:

void	printf	("string", ...)
void	exec	("command_string")
int	getint	("prompt_string")
int	getlong	("prompt_string")
int	getfloat	("prompt_string")
int	rand	(int seed)
void	twatch	(ulong cycles)
void	memset	(ulong start, ulong end, uchar val)

- ◆ **void printf ("format_string", ...)**

  formatted output can be done using the function "printf()".   The first argument must be a format string, the following optional parameters can be expressions or even strings.   The conventional C specifications apply as format specifications:

  Examples:
  ```
  >printf ("random number = %04XH\n", rand(0))
  random number = 1014H
  >printf ("random number = %04XH\n", rand(0))
  random number = 64D6H
  >printf ("%s-%d %s\n", "dScope",51, "Windows")
  dScope-51 Windows
  >printf ("%lu\n", (ulong) -1)
  4294967295
  >printf ("%u\n", (ulong) -1)
  65535
  >printf ("%ld\n", (ulong) -1)
  -1
  ```

- ◆ **int getint ("prompt_string")**

  The request string is output, a dialog box is displayed with an edit field to enter the value (see getfloat() for an example).   In the entry line, a numeric value must appear (operators are not allowed).   The value is calculated, adapted to the type "int", and returned to the caller.

  Example:
  ```
  >getint ("enter integer value:")
  ```

- ◆ **int getlong ("prompt_string")**

  The request string is output, a dialog box is displayed with an edit field to enter the value (see getfloat() for an example).   In this entry line, a numeric value must appear (operators are not allowed).   The value is calculated, adapted to the type

"long", and returned to the caller.
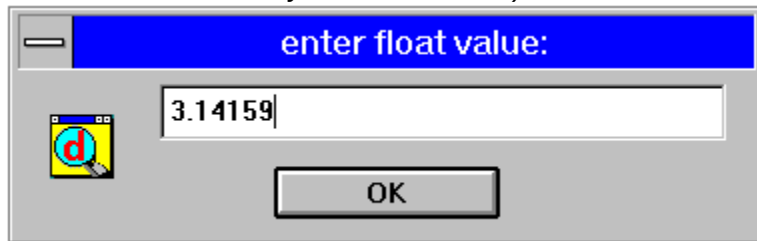**Example:**
```
>getlong ("enter long value:")
```

◆ **float getfloat ("prompt_string")**

The request string is output, a dialog box is displayed with an edit field to enter the value.   In this entry line, a numeric value must appear (operators are not allowed).   The value is recalculated, adapted to type "float" and returned to the caller.

Example:
```
>getfloat ("enter float value:")
```

A dialog box is displayed with string parameter given in the get-value command being the dialog box title.   In the Edit field, enter the value, the hit return or press the OK button.   The dialog will then be closed with the entered value (or zero, if no entry has been made) is returned to the caller:

◆ **void exec ("command_string")**

The string parameter is passed to dScope for the execution.   The string must therefore be a valid dScope command.

Examples:
```
>exec ("DIR PUBLIC; eval dptr + r7")
>exec ("BS timer0")
>exec ("BK *")
```

The command string can contain several commands separated by a semicolon.

The Following Commands are Not Allowed by exec():
```
EXIT
FUNC/SIGNAL (Introduce function definition)
ASM/ENTER
RESET
LOAD
```

Expressions that are valid in a dScope command line can also be contained in a function.   It is therefore not necessary to calculate expression via "exec()".

◆ **int rand (int seed)**

"rand()" returns a random number in the area -32768 to +32767.   If a value not

equal to 0 is passed a parameter, the random number generator is initialized with the value.

Examples:
```
>rand (0x1234)         /* Initialize random generator with 0x1234 */
0x3B98
>rand (0)              /* No initialization */
0x64BD
>rand (0)
0x12B5
```

- **void twatch (long cycles)**

  "twatch()" sets a timer breakpoint.   The parameter "cycles" determines how many cycles must   elapse beginning with the current state of the cycle counter until the timer breakpoint is active.   dScope updates the cycle counter during the program run, when a "Go" or "STEP" command runs.

  "twatch()" is allowed only within Signal Functions!

  For exact use, see Signal Functions.

  amples:
```
>twatch (4096)      /* allowed only from within signal functions */
 twatch (4096)      /* Invalid at the command level */
_^
ERROR 145: TimeWatch(): not within signal()
```

- **void memset (ulong startaddr, ulong endaddr, uchar value)**

  "memset" is used to fill a memory range with the value "value".   The first parameter must return a unique memory space so that "memset" knows which memory space is concerned.

  Examples:
```
>MEMSET (X:0, X:0xFFFF, 'a') /* Write XDATA RAM with "a" */
>MEMSET (C:0x0, C:0x1FFF, 0) /* clear code memory range */
>MEMSET (D:0x30, D:0x7F, r7) /* DATA area 0x30-0x7F, value from R7 */
```

Signal Functions

Signals and pulse forms can be generated with signal functions.   The time interval between the signal changes is produced using the predefined function "twatch()".   A signal function must begin with the keyword "SIGNAL".   User functions begin with "FUNC".

The following guidelines apply for signal functions:

- The return type must be void.   A maximum of eight function parameters are permissible.

- A signal function can invoke other predefined functions and user functions, but no other signal functions.   User functions cannot invoke signal functions nor "twatch()".

- At least one call of the predefined function "twatch()" must exist.   Signal functions without "twatch()" result in nonsense because no relation to the execution time of the program exists, and for the other functions, **Ctrl+C** cannot be used to abort.   dScope can therefore enter an infinite loop.

If a signal function is invoked, dScope performes the following:

- The signal function is included in the queue of active signal functions.   An already active signal function cannot be active a multiple number of times.   The list of active signal functions can be output using the command "SIGNAL STATE".

- The function is marked as running.

- Execution of the function is started and stopped as soon as a call to function "twatch()" has been executed.

- After execution of a "twatch()" call, the signal function is marked as idle.   The signal function is therefore frozen for the number of processor cycles defined with "twatch()".

- After the number of "twatch()" cycles have elapsed, the signal function is reactivated, and marked as running.   Execution continues at the statement after the preceding "twatch()" call, until a new "twatch()" call is executed.

- If the signal function is terminated, for example by a return statement or end of the function, it is automatically removed from the queue of active signal functions.

Active signal functions can be removed from the queue of active signal functions using the command:

```
SIGNAL KILL signalfunction_name
```

The name of active signal function must be specified here as a parameter.   Afterwards, the signal function is available again for another call.

The command "SIGNAL STATE" lists the active signal functions.   It has the following format:

```
>SIGNAL STATE
 0  idle      Signal = ANALOG0 (line 10)
```

The leading number is an internal code followed by the status of the signal function.   This can be idle or running.   Afterwards, the name of the signal function and the line number within the signal function follow, by which the execution is continued.

Signal Function Example:

A signal function is created in the following example which applies a step-forming voltage on the analog input 0 of the 80517 in the area between 0 V and an upper value limit.   The voltage is increased in intervals of 5 V.   After the voltage reaches the peak, the backwards stepping to 0 V is produced in 0.5 V intervals.   The entire procedure should be continued indefinitely.   The time interval between the voltage changes should comprise 25 ms in reference at 12 MHz clock; this corresponds to a cycle number of 25000.   The voltage is to be changed every 25000 cycles.

The text of the function was created by means of an editor and stored in the file "ANALOG". The function requires that the 80517.DLL is loaded, since the symbol "AIN0" is   defined from the CPU driver 80517.DLL.   "AIN0" is the name if the first analog input of the 80C517 controller.

```
>
>INCLUDE ANALOG
>SIGNAL void analog0 (float limit)  {
    2:
    3:  float volts;
    4:
    5:  printf ("ANALOG0 (%f) ENTERED\n", limit);
    6:  while (1)  {             /* Indefinite */
    7:   volts = 0;
    8:   while (volts <= limit)  {
    9:    ain0 = volts;        /* Analog input 0 */
   10:    twatch (25000);      /* 25000 cycles time beak */
   11:    volts += 0.5;                /* Increase voltage */
   12:   }
   13:   volts = limit-0.5;
   14:   while (volts >= 0.5)  {
   15:    ain0 = volts;
   16:    twatch (25000);              /* 25000 cycles time break */
   17:    volts -= 0.5;                /* Decrease voltage */
   18:   }
   19:  }
   20: }
>
>DIR SIGNAL                             /* Output names for the signal funct*/
  signal:  void  ANALOG0 ( float )
```

The signal function "analog0" can then be invoked:

```
>ANALOG (5.0)                      /* Start of 'ANALOG()' */
ANALOG0 (5.000000) ENTERED
```

The command "SIGNAL STATE" is used to determine the current state of the function "analog0":

```
>SIGNAL STATE
 0  idle     Signal = ANALOG0 (line 10)
```

The output means that "analog0" executed the "twatch()" call up to line 10 and is contained in the idle state.   After completion of 25000 cycles by the start of a "Go" or T STEP command, the execution in line 11 is continued until the next "twatch()" call in line 10 or line 16, when the first WHILE loop has elapsed.

"analog0" can be removed from the queue of active signal functions when desired.

```
>SIGNAL KILL ANALOG0
```

Deviations of dScope Functions from C Language

◆ As opposed to the C language, dScope does not differentiate between uppercase and lowercase.   The names of objects or the control statements (FOR ...) can therefore be written in either uppercase or lowercase.

◆ There is no preprocessor.   Preprocessor statements like "#define", "#include", "#ifdef", etc.   are therefore not supported.

◆ There are no global declarations.   Declarations of scalars must   be within functions. Otherwise, the function cannot access the complete dScope symbol table.   You can however, define symbols with the '**DEFINE <type><name>'** command and use such variables for value place holders to substitute global declarations.

◆ There is no initialization for the declaration of a scalar.   If an initial value is desired, an explicit assignment must be made.

◆ Only scalar types can be declared as local objects within a function.   Structures, arrays and pointers are not allowed.   This applies for the function return type as well as parameters also.

◆ Functions can only return scalar types to the caller.   Pointers and structures are not allowed.

◆ Functions cannot be called recursively, neither directly nor indirectly.   During the execution of functions, dScope recognizes violations to this rule and aborts the function execution.

◆ Functions can only be activated with their name.   Function calls of an indirect nature via pointers are not supported.

◆ For the declaration of functions with parameters, only the new ANSI form is supported; the old K&R format is not supported:

```
func test (int pa1, int pa2)  {   /* ANSI type, correct */
  /* ...  */
}
func test (pa1, pa2)              /* Old K&R style */
int pa1, pa2;                     /* not supported !!! */
{
  /* ...  */
```

}

Error message format

dScope issues a numbered error message when an error is encountered.   The point where the error was recognized may be marked also, depending on context.   For example:

```
acc + r0 + r500
-----------^
ERROR 125:  symbol or line not found
load c:\objs\measure
-----^
ERROR 109:  file does not exist
```

List of Error Numbers and Messages

100 -   illegal digit in number
A illegal digit in a number was detected.   Make sure the digits of the number are conforming to the number base, for example, digits > 0-7 in octal numbers, 0-9 in decimal numbers etc.

101 -   illegal scope qualifier
A scope qualifier consists of a module name specifier, a function name specifier or line number, and in case of function name an optional local symbol specifier such as **\MEASURE\MAIN\cmdbuf** or **\MEASURE\225** or **\MEASURE\sindex** but not **\MEASURE\225\cmdbuf**, for example.

102 -   illegal bank specifier
A bank specifier is used to specify a symbol in some code bank for use in memory banked applications.   The bank specifier should have the following format:
> **\\BankNumber\SymbolName**
> **\\BankNumber\ModuleName\Line**
> **\\BankNumber\ModuleName\LocalSymbolName**

Example: **\\5\Bank5Module\buffer**

103 -   incomplete bank specifier
A bank specifier must include the bank specification (for example \\3) followed by a scope qualifier. See also Error 102.

104 -   too many qualifiers
A scope qualifier may consist of at most three components: a module, a function or line number and a local symbol in case the second component specifies a function.   See also Error 101.

105 -   invalid object file
An attempt was made to load an object file which is either corrupted or does not conform to the OMF-51 or OMF-251 format specification.

106 -   more than 16000 lines in one module
A single module of the user program must not contain more than 16000 line numbers.   If this error occurs, you should break such a huge module into two modules.

107 -   invalid hex file
An attempt was made to load a hex file which is either corrupted or otherwise does not conform to the HEX, HEX86 or HEX386 format specification.

108 - **checksum error in hex file**
The hex file to be loaded contains a checksum error.

109 - **file does not exist**
The file name given in a LOAD or INCLUDE command specifies a non existing file.

110 - **literal expected**
A literal was expected but not found.   A literal is normally used to enter a file name with an optional path specification.   A character string, for example, can also be used as a literal; i.e.   for the "LOAD" command:

LOAD "c:\\user1\\project.90\\newcmd"/* String literal */
LOAD c:\user1\project.90\newcmd   /* Equivalent   */

111 - **illegal character constant**
The specified character constant is not correct.   Either the constant is not closed with ', or it contains an ESCAPE sequence for a number whose value is greater than 0xFF (255).

112 - **out of range escape value**
An escape sequence in a string or character constant must evaluate to a value in range 0 to 0xFF, greater values are not permitted.

113 - **unclosed string**
The specified string is unclosed.   The string must be enclosed with double quote characters, for example "string".   If strings are nested, then the double quotes of the inner string must be escaped, for example "printf (\"hello\n\")".

114 - **syntax error**
Another term is expected at the marked position than found.

115 - **illegal register combination**
The tokens found do not form a valid register name for the inline assembler, for example @DPTR+A instead of @A+DPTR or @R4 where only @R0 and @R1 are valid.

116 - **unclosed comment**
A non-closed comment was discovered during the analysis of a function. Note that a comment must be closed within one line.   Multiline comments are not permitted.

117 - **expression too complex**
This error occurs when very complex expressions (possibly separated by comma) are analyzed.   In command mode, expressions can be simplified or divided into several command lines.   In function definition mode, comma expressions should be avoided.   Separate statements should be used

instead.

118 - **')' expected**
A closing parenthesis was expected but some other token was found.

119 - **'(' expected**
A n opening parenthesis was expected but some other token was found.

120 - **';' expected**
A semicolon was expected in a function definition but some other token was found.

121 - **'}' expected**
A curly right closing bracket to complete a statement block is expected.   The error mark shows the approximate position where the bracket is expected.

122 - **']' expected**
A] bracket after an index expression was expected but some other token was found.

123 - **identifier expected**
An identifier was expected but some other token was found.

124 - **':' expected**
A colon was expected, for example in a case expression in a function but some other token was found.

125 - **symbol or line not found**
The given symbol or line number does not exist.

126 - **'void' expected**
One of the following is required when starting a function definition:
Parameter list (i.e.   func void test (int i0, long l1))

Empty list (i.e.   func void test ())

Void list (i.e.   func void test (void))

127 - **illegal expression token**
An expression contains a token not valid in expression context.

128 - **illegal type 'void'**
A local variable within a function cannot be declared to be of type void.

129 - **illegal type**
An invalid type was used in the given context, for example 'DEFINE void var' or an a void type subexpression within an expression.

local variable within a function cannot be declared to be of type void.

130 -   illegal or unknown memory space
An expression did not yield a valid memory space where a memory space is required,   for example an expression in a Display or breakpoint expression.

131 -   illegal type conversion
An inadmissible type combination of the operands was discovered during the analysis of an expression.   The expression is rejected.

132 -   unknown struct/union member
The name of the given structure member is undefined.

133 -   undefined peripheral I/O location
The name given in an I/O register specifier 'P:name' is undefined.   Use the DIR VTREG command to get the I/O register names currently available.

134 -   operator requires lvalue
The address of operator must have an expression to the right side which represents the address of an object:

```
&ACC            /* Valid */
&(ACC + R0)     /* Invalid, no address because + */
```

135 -   illegal bit position
The bit position in a bit address must evaluate to a constant value in range 0 to 7.
136 -   invalid base for bit type expression
Specification of the byte base in a bit address must be in the area 0x20 to 0x2F or in the area 0x80 to 0xFF and be divisible by 8 without a remainder.   MCS 251 bit addresses allow the base to be any value in range 0x20 to 0xFF without restrictions.

137 -   left side of '.id' requires struct/union
The expression to the left of   dot operator for accessing structures must contain the type structure or union, respectively.
138 -   left side of '->id' requires struct/union pointer
The expression left to the pointer operator for accessing structures via pointers must contain be of type 'pointer to structure or union'.

139 -   [dim] applied to non array
The expression left to the array reference does not yield a pointer or array type expression.

140 -   '&' requires lvalue
The address of operator cannot be applied to constants, for example.

141 -   '*':   invalid indirection
The indirection with the asterisk operator requires that the expression right of the asterisk contains the type 'pointer to ...'.   Typeless pointers cannot be dereferenced.

142 - bad op in float type expression
The "NOT" operator '~' for example can't be applied to operands of float type .

143 - invalid left hand side of assignment expression
The left side of an assignment operator must contain an expression whose address represents a lvalue.   Assignments to constants or functions for example are not possible.

144 - call to undefined function
The given function call does not refer to a currently defined dScope function.

145 - TimeWatch():   not within Signal()
The call to the predefined function "twatch()" can only occur from within a signal function ("SIGNAL void sig(void)").

146 - can't activate Signal() from user function
A signal function cannot be invoked from any other function.   This applies the both signal and non-signal functions.   Signal functions can be activated at the command level only.

147 - incompatible parameter type
Some of the predefined functions (i.e.   "printf()" or "getint()") expect a character string as the actual parameter.   Everything else besides a string causes an error message.

148 - missing function parameter
Too few actual parameters were specified in a function call.   The number of parameters must agree to the function definition.

149 - too many actual parameters in function call
Too many actual parameters were specified in a function call.   The number of parameters must agree to the function definition.

150 - improper operand
The specified operand does not correspond to the requirements of the assembler instruction.

151 - div/mod by zero error
A division or modulo by zero has been detected in an expression.

152 - function difference encountered
This error occurs under the following conditions:

A function was defined; i.e.   "test1()".

A function (i.e.   "test2()") was defined that contains a call to "test1()".

The parameter number or parameter types or the return type of the function "test1()" is subsequently changed.

The function "test2()" is invoked.

It is determined during the execution of "test2()" that the function "test1()" does not use the same number of parameters or deviating types.   This is inadmissible.   In

this case, the invoked function must be adapted.

153 -   **too many parameters**
More actual parameters were passed to a dScope function than declared in that function, this is inadmissible.

154 -   **recursive user function activation**
Calls to dScope functions must not be recursive.   Recursions are recognized at the runtime of functions; the execution is aborted in that case.

155 -   **GetNumber-functions: invalid number**
The predefined dScope functions "getint()", "getlong()" and "getfloat()" expect a numeric parameter without any operators to be input.

156 -   **unknown identifier**
An identifier from a scope qualifier is undefined.

157 -   **undefined line number**
An identifier line number from a scope qualifier has been detected.

158 -   **'{', scope stack overflow (16)**
The maximum nesting level of statement blocks within a function definition comprises 16.

159 -   **invalid 'break/continue'**
Break and continue statements are only permissible according to the rules of the C language.   Valid enclosing statements are FOR, WHILE, DO and SWITCH/CASE.

160 -   **'case/default':   missing enclosing switch**
A case or default statement requires an enclosing switch statement.

161 -   **more than one 'default'**
Only one default statement can exist in one switch statement level.

162 -   **duplicate label**
A label must not be defined more than once in one function.   Use another name for the label to avoid redefinition.

163 -   **missing return expression**
A function with any return type but void must contain at least one return statement with an expression.   Functions that do not have an explicit type specified contain the type "int" and must therefore also contain a return statement with an expression.

164 -   **return value on void function**
A void function cannot return a value to the caller.   The return statement must therefore not contain an expression.

165 -   undefined label
A label referenced by a GOTO statement is undefined.

166 -   not an integer constant expression
The expression of a CASE statement must be of the type (u)char or (u)int.   Type
float or other types are inadmissible.

167 -   invalid type on controlling expression
The type of an expression that control a loop (FOR, WHILE, DO) must be an integral
type; e.g.   type structure/union/array is not allowed.

168 -   duplicate identifier (parm or local)
The name of a parameter or local object must not appear more than once, otherwise
a redefinition error occurs.

169 -   more than 8 parameters
The number of parameters to a single function is limited to 8.

170 -   undefined function
The given function call does not refer to a currently defined dScope function.

171 -   can't redefine built-in function
A predefined function such as printf() or memset() cannot be redefined.

172 -   can't remove built-in function
A predefined function such as printf() or memset() cannot be removed.

173 -   signal function: can't receive or return value(s)
A signal function cannot receive or return a value, it must therefore be a void function,
i.e.   signal void sig1 (void).

174 -   duplicate case label
The constant expression of a CASE statement may only appear once in one switch
statement level.

175 -   insufficient memory
dScope has run out of memory while allocating memory for finalizing a function body.

176 -   function too big
The function being compiled is too big.   This is the case if a function contains more
than 512 labels or the dScope internal code size for that function exceeds 32K bytes.

177 -   signal function must contain a twatch() call
At least one call of the function "twatch()" must be contained in a signal function.
Otherwise, a signal function makes no sense.   Signal functions cannot be aborted
with **Ctrl+C** when these are in an infinite loop.   Such functions should be killed using
the SIGNAL KILL command.

178 -   internal: function execution error
During the execution of a user function, an internal error occurred.   Please contact your local distributor with details on the function that caused the error.

179 -   this signal() already activated
Attempt was made to reactivate a previously active signal function.   This is not permissible without removing the function from the queue of active signal functions using the command "SIGNAL KILL <func>".

180 -   too many signal functions
The maximum number of concurrently running signal functions is 10.

181 -   no such signal function
The function specified in a "SIGNAL KILL" command is not in the queue of active signal functions.

182 -   limit exceeded: function nesting (20)
The maximum nesting level of 20 was exceeded during the execution dScope functions.   The execution is aborted.

183 -   Command not allowed now
The given command cannot be executed now.   This can be the case for example, when the command 'RESET MAP' is entered while dScope is executing the user program.

184 -   Include nesting limit exceeded
The maximum nesting level of 3 or 4, depending on command context was exceeded.   The execution is aborted.

185 -   invalid restricted access break expression
During the specification of a breakpoint with an access specification (READ, WRITE or READWRITE), one of the following rules were neglected:

A)   The address part of an expression must have a unique memory type.   In other words, only a name of an object may occur.   For the specification of a structure element, an constant offset is included here.

B)   Only the operators &, &&, <, <=, >, >=, ==, and != are permissible.   An expression corresponding to rule A) must exist to the right of one of these operators. An optional expression can exist to the left of the   operator.   Rule A) above does not apply to the expression right of the operator.

Example with Following Declarations in a C Program:

struct time { char hour; char min; char sec; } time;
int i0, i1;

Use in Qualified Abort Conditions:

BS WRITE time.sec   /* Correct */
BS WRITE time.sec + i0   /* Wrong, */

```
                                           /* mem type not unique/
BS WRITE time.sec == i0                    /* Correct */
BS WRITE time.sec != i0*i1                 /* Correct */
BS WRITE time.sec && (ACC==5 && i1!=i0)    /* Correct */
```

186 -   invalid item number

The specified number to delete a watchpoint or breakpoint does not identify a valid watchpoint or breakpoint.

187 -   access out of bounds (address)

An attempt was made to access an invalid location in memory.   This is the case if an unmapped memory address is accessed.

188 -   invalid or out of range number base

The specification of the number base for use in a watch expression was not 16 (hexadecimal) or 10 (decimal).   Bases other than 0x0a and 0x10 are not permissible.

189 -   address value out of range

An expression representing some memory address for a command was not in range 0x000000 to 0xFFFFFF (0 to 16M-1).

190 -   exit-address required for PA-range

The PA range definition requires one more address expression specifying the end of the range.   Since dScope cannot derive that address from the range start address, you must supply it.

191 -   PA-range overlaps an existing PA-range

A PA address range being defined must not overlap an already existing PA-range (neither partial nor total overlap).

192 -   unknown environment variable

The environment variable given in a SET command is undefined.   The valid environment variables are F0 ... F12 and SRC.

193 -   can't redefine an activated function

An already invoked and still running dScope function can't be redefined.   Wait for completion of the function or in case of a signal function kill it first.

194 -   Access violation at <address>

While executing a user program, a memory access violating the access permissions of the target address has been detected.   This can happen on erroneous programs or if the memory map command(s) are simply incorrect or incomplete.

195 -   redefinition

An attempt was made to define an already existing value symbol with the DEFINE <type><name> command.

196 -   log file already active

A LOG or SLOG command attempted to open a log file while a log file already was active.

197 -   can't create (or append to) logfile
The file name specified in a LOG or SLOG command designates a file which can't be opened or written to.   This may happen if the file name specifies a directory or a read only file.

198 -   too many operands
Inline assembly: the given assembler instruction contains too many operands, for example a NOP instruction may not have any operands.

199 -   number of operands does not match instruction
Inline assembly: the number of operands given in the assembler instruction does not match the requested number of operands for that instruction, for example 'CALL 0x1000,DPTR' where CALL requires one parameter only.

200 -   illegal type override
An operand to an assembler instruction used an illegal type override, for example 'VOID DATA 0x20' or 'PTR XDATA 0x20'.   The types void and ptr are not allowed in type overrides, BIT , BYTE, WORD and DWORD types are valid.

201 -   illegal operand: '/register or '#register'
An illegal register operand has been detected in an inline assembler instruction. Register expressions may not be prefixed by '/' (not bit) or '#' (immediate).

202 -   register: illegal type- or space override
A register operand cannot be preceded by a memory space or type override.

203 -   instruction does not match cpu-type
An assembler instruction not available on the current CPU has been detected.

204 -   branch target out of range
The branch target address given in a jmp or call instruction is out of range.   The target for MCS 51 conditional jump's must be within +127/-128 bytes relative to the current execution point.

205 -   illegal register operand
An invalid register operand was detected in an inline assembler instruction, for example 'MOV A,R11'   where only registers R0 to R7 are allowed for the second operand.

206 -   invalid short value
A short constant value for an MCS 251 'INC/DEC Rn,#short const' is out of range. Such a short constants must be of value 1, 2 or 4.

207 -   invalid instruction operand
An operand given in an inline assembler instruction does not match the requested class of operands, for example an #immediate operand instead of a register operand.

208 -   RESTRICTED VERSION: code size limit exceeded

You have tried to load a user program which violates the code size limits of your restricted dScope software.
Consult your local dealer for information on how to get the   unrestricted dScope for Windows version.

209 -   illegal von Neumann map command
The address range in a von Neumann map command must not cross a 64K boundary and it must not be an address range from the code segment (0xFF0000 to 0xFFFFFF).

210 -   too many items
An attempt was made to define more than 255 PA ranges.

211 -   Access to non existing SFR (0xnnnn)
A non existing SFR has been accessed.   This kind of access check is peformed on derivatives only which do not allow access to non existing special function registers.

212 -   command not supported in target mode
The command just executed is not allowed in target mode.   dScope is in target mode if a target driver such as MON51.DLL or MON251.DLL has been loaded.   In this case, it is not possible to map memory or use the performance analyzer, for example.

213 -   invalid break address
The breakpoint address is not valid.   This applies to target mode if a break address overlaps a break address range of an existing breakpoint.

214 -   unsupported breakpoint type
The given breakpoint type is not supported by the target driver.   For example, the MON251.DLL target driver does not support **Access** type breakpoints.

215 -   premature end of file
This error signals the end of an include file which contains a definition of a dScope function which is not yet completed on end of include file.   Function definitons cannot cross file boundaries.

Purpose of a CPU Driver

Using a CPU Driver dScope is a general purpose MCS 51/251/80C166 family debugger/simulator.   In order to accommodate the various members of the 8051 family of microcontrollers, each iteration of dScope is "customized" with loadable I/O drivers.   These drivers are Windows DLL's (dynamic link libraries).   The CPU drivers contain chip specific and peripheral specific configuration information for the dScope program.   The various members of the MCS 51 family (8052, 80515, 80517, etc.) or the MCS 251 family typically differentiate in the peripherals integrated in the chip.   In order to use dScope for simulating most members of the MCS 51/251 family, the simulation definitions of device specific peripherals was placed in separate driver programs.   This guarantees that in the future, new microcontrollers expanding the microcontroller family can also be supported.

With over 140 members, it is not possible to have an CPU driver file for each derivative.   In most cases it is enough to be able to simulate most of any given chip, in that software can be written with conditional variations to accommodate limitations in current testing methodology.   For instance, it is not possible to effectively simulate I2C Bus hardware.   While certainly not many, the prudent user should be aware of these limitations, and must test around them.

The dScope debugger/simulator is intended to simulate the "instruction engine" portion of your CPU, it is not intended to be a "sub-system" or "system" simulator.   Generally accepted 'C' language programming techniques encourage "modular" software design.   For instance, if your design is to be used on a chip not currently completely supported with a unique driver, you can easily test one portion (one or more modules) of your code with one driver, and another portion of your code with a different driver.

Another effective technique often used by advanced developers to test code intended for chips not yet released, is to use the function definition capability of dScope to write specific peripheral device routines. In that way when specific programs or data locations are reached, or specified conditions achieved, a function is called to provide the appropriate response, activity, or result.

If no specific driver has been loaded, a standard 8051 can be simulated.   However, none of the on-chip peripherals (timer, ports, A/D converter, etc.) would be active or have any significance.   Use or initialization of any of the control or Special Function Registers (SFR) would have no affect.   The names of the I/O ports will not be recognized, and communication with the outside world would not be possible.

List of available drivers

80251S.DLL

8051.DLL

8052.DLL

8051FX.DLL

80515.DLL

80515A.DLL

80517.DLL

80517A.DLL

80552.DLL

80751.DLL

80410.DLL

80781.DLL

80320.DLL

80251S.DLL

Driver file for the 80251SA, SB SQ and SP chips.   The 80251S.DLL driver file simulates the logical and timing behavior of the following hardware components:

- ◆ Timer 0
- ◆ Timer 1
- ◆ Timer 2 with Reload/Capture register
- ◆ PCA Timer/Counter with 5 16-bit Capture/Compare modules
- ◆ Serial Interface
- ◆ Interrupt System
- ◆ Power Saving Modes (Idle, Power-down)

NOTE   Timer 2 in Clock-Out Mode, is not supported.

80251SB.DLL defines the following VTREG symbols (peripheral registers):

PORT0:	digital I/O lines of PORT 0 (8-bit)
PORT1:	digital I/O lines of PORT 1 (8-bit)
PORT2:	digital I/O lines of PORT 2 (8-bit)
PORT3:	digital I/O lines of PORT 3 (8-bit)
SIN:	serial input for SERIAL CHANNEL 0 (9-bit)
SOUT:	serial output for SERIAL CHANNEL 0 (9-bit)
XTAL:	Oscillator frequency
STIME:	serial Timing enable

8051.DLL

Driver file for the 8051, 8031, 80C51, 80C31, 80C52T2 and other similar chips.   The 8051.DLL driver simulates the logical and timing behavior of the following hardware components:

- ◆ Timer 0
- ◆ Timer 1
- ◆ Serial Interface
- ◆ Interrupt System
- ◆ Power Saving Modes (Idle, Power-down)

8051.DLL defines the following VTREG symbols (peripheral registers):

PORT0:	digital I/O lines of PORT 0 (8-bit)
PORT1:	digital I/O lines of PORT 1 (8-bit)
PORT2:	digital I/O lines of PORT 2 (8-bit)
PORT3:	digital I/O lines of PORT 3 (8-bit)
SIN:	serial input for SERIAL CHANNEL 0 (9-bit)
SOUT:	serial output for SERIAL CHANNEL 0 (9-bit)
XTAL:	Oscillator frequency
STIME:	serial Timing enable

8052.DLL

Driver file for the 8052, 8032, 80C52, 80C32, and similar chips.   The 8052.DLL driver file simulates the logical and timing behavior of the following hardware components:

◆ Timer 0

◆ Timer 1

◆ Timer 2 with Reload/Capture register

◆ Serial Interface

◆ Interrupt System

◆ Power Saving Modes (Idle, Power-down)

8052.DLL defines the following VTREG symbols (peripheral registers):

PORT0:	digital I/O lines of PORT 0 (8-bit)
PORT1:	digital I/O lines of PORT 1 (8-bit)
PORT2:	digital I/O lines of PORT 2 (8-bit)
PORT3:	digital I/O lines of PORT 3 (8-bit)
SIN:	serial input for SERIAL CHANNEL 0 (9-bit)
SOUT:	serial output for SERIAL CHANNEL 0 (9-bit)
XTAL:	Oscillator frequency
STIME:	serial Timing enable

8051FX.DLL

Driver file for the 8051FA, 8051FB, 8051FC, and similar chips.   The 8051F.DLL driver file simulates the logical and timing behavior of the following hardware components:

- Timer 0
- Timer 1
- Timer 2 with Reload/Capture register
- PCA Timer/Counter with 5 16-bit Capture/Compare modules
- Serial Interface
- Interrupt System
- Power Saving Modes (Idle, Power-down)

NOTE   Timer 2 in Clock-Out Mode, is not supported.

8051F.DLL defines the following VTREG symbols (peripheral registers):

PORT0:	digital I/O lines of PORT 0 (8-bit)
PORT1:	digital I/O lines of PORT 1 (8-bit)
PORT2:	digital I/O lines of PORT 2 (8-bit)
PORT3:	digital I/O lines of PORT 3 (8-bit)
SIN:	serial input for SERIAL CHANNEL 0 (9-bit)
SOUT:	serial output for SERIAL CHANNEL 0 (9-bit)
XTAL:	Oscillator frequency
STIME:	serial Timing enable

80515.DLL

Driver file for the 80515, 80C515, 80512, and similar chips.   The 80515.DLL driver file simulates the logical and timing behavior of the following hardware components:

◆ Timer 0

◆ Timer 1

◆ Timer 2 with Reload/Capture register

◆ A/D Converter

◆ Watchdog

◆ Serial Interface

◆ Interrupt System

◆ Power Saving Modes (Idle, Power-down)

80515.DLL defines the following VTREG symbols (peripheral registers):

PORT0:	digital I/O lines of PORT 0 (8-bit)
PORT1:	digital I/O lines of PORT 1 (8-bit)
PORT2:	digital I/O lines of PORT 2 (8-bit)
PORT3:	digital I/O lines of PORT 3 (8-bit)
PORT4:	digital I/O lines of PORT 4 (8-bit)
PORT5:	digital I/O lines of PORT 5 (8-bit)
AIN0:	analog input line AIN0 (float value)
AIN1:	analog input line AIN1 (float value)
AIN2:	analog input line AIN2 (float value)
AIN3:	analog input line AIN3 (float value)
AIN4:	analog input line AIN4 (float value)
AIN5:	analog input line AIN5 (float value)
AIN6:	analog input line AIN6 (float value)
AIN7:	analog input line AIN7 (float value)
VAGND:	analog reference voltage VAGND (float value)
VAREF:	analog reference voltage VAREF (float value)
SIN:	serial input for SERIAL CHANNEL 0 (9-bit)
SOUT:	serial output for SERIAL CHANNEL 0 (9-bit)
XTAL:	Oscillator frequency
STIME:	serial Timing enable

80515A.DLL

Driver file for the 80C515A and similar chips.   The 80515A.DLL driver file simulates the logical and timing behavior of the following hardware components:

◆ Timer 0

◆ Timer 1

◆ Timer 2 with Reload/Capture register

◆ A/D Converter

◆ Watchdog

◆ Serial Interface

◆ Interrupt System

◆ Power Saving Modes (Idle, Power-down)

80515A.DLL defines the following VTREG symbols (peripheral registers):

PORT0:	digital I/O lines of PORT 0 (8-bit)
PORT1:	digital I/O lines of PORT 1 (8-bit)
PORT2:	digital I/O lines of PORT 2 (8-bit)
PORT3:	digital I/O lines of PORT 3 (8-bit)
PORT4:	digital I/O lines of PORT 4 (8-bit)
PORT5:	digital I/O lines of PORT 5 (8-bit)
AIN0:	analog input line AIN0 (float value)
AIN1:	analog input line AIN1 (float value)
AIN2:	analog input line AIN2 (float value)
AIN3:	analog input line AIN3 (float value)
AIN4:	analog input line AIN4 (float value)
AIN5:	analog input line AIN5 (float value)
AIN6:	analog input line AIN6 (float value)
AIN7:	analog input line AIN7 (float value)
VAGND:	analog reference voltage VAGND (float value)
VAREF:	analog reference voltage VAREF (float value)
SIN:	serial input for SERIAL CHANNEL 0 (9-bit)
SOUT:	serial output for SERIAL CHANNEL 0 (9-bit)
XTAL:	Oscillator frequency
STIME:	serial Timing enable

80517.DLL

Driver file for the 80C517, and similar chips.   The 80517.DLL driver file simulates the logical and timing behavior of the following hardware components:

- ◆ Timer 0
- ◆ Timer 1
- ◆ Timer 2 with Reload/Capture register
- ◆ A/D Converter
- ◆ Watchdog
- ◆ Two Serial Interfaces
- ◆ Arithmetic Unit
- ◆ 8 Data Pointers
- ◆ Interrupt System
- ◆ Power Saving Modes (Idle, Power-down)

NOTE   the following hardware components are not simulated:
Exact timing of the Arithmetic Unit.
Digital Input Lines of Port 7 and Port 8.
Oscillator Watchdog.

80517.DLL defines the following VTREG symbols (peripheral registers):

PORT0:	digital I/O lines of PORT 0 (8-bit)
PORT1:	digital I/O lines of PORT 1 (8-bit)
PORT2:	digital I/O lines of PORT 2 (8-bit)
PORT3:	digital I/O lines of PORT 3 (8-bit)
PORT4:	digital I/O lines of PORT 4 (8-bit)
PORT5:	digital I/O lines of PORT 5 (8-bit)
PORT6:	digital I/O lines of PORT 6 (8-bit)
PORT7:	digital I/O lines of PORT 7 (8-bit)
PORT8:	digital I/O lines of PORT 8 (8-bit)
AIN0:	analog input line AIN0 (float value)
AIN1:	analog input line AIN1 (float value)
AIN2:	analog input line AIN2 (float value)
AIN3:	analog input line AIN3 (float value)
AIN4:	analog input line AIN4 (float value)
AIN5:	analog input line AIN5 (float value)
AIN6:	analog input line AIN6 (float value)
AIN7:	analog input line AIN7 (float value)
AIN8:	analog input line AIN8 (float value)
AIN9:	analog input line AIN9 (float value)
AIN10:	analog input line AIN10 (float value)
AIN11:	analog input line AIN11 (float value)
VAGND:	analog reference voltage VAGND (float value)
VAREF:	analog reference voltage VAREF (float value)
S0IN:	serial input for SERIAL CHANNEL 0 (9-bit)
S0OUT:	serial output for SERIAL CHANNEL 0 (9-bit)

S1IN:	serial input for SERIAL CHANNEL 1 (9-bit)
S1OUT:	serial output for SERIAL CHANNEL 2 (9-bit)
XTAL:	Oscillator frequency
STIME:	serial Timing enable

80517A.DLL

Driver file for the 80C517, and similar chips.   The 80517.DLL driver file simulates the logical and timing behavior of the following hardware components:

◆ Timer 0

◆ Timer 1

◆ Timer 2 with Reload/Capture register

◆ A/D Converter

◆ Watchdog

◆ Two Serial Interfaces

◆ Arithmetic Unit

◆ 8 Data Pointers

◆ Interrupt System

◆ Power Saving Modes (Idle, Power-down)

NOTE   the following hardware components are not simulated:
            Exact timing of the Arithmetic Unit.
            Digital Input Lines of Port 7 and Port 8.
            Oscillator Watchdog.

80517.DLL defines the following VTREG symbols (peripheral registers):

PORT0:	digital I/O lines of PORT 0 (8-bit)
PORT1:	digital I/O lines of PORT 1 (8-bit)
PORT2:	digital I/O lines of PORT 2 (8-bit)
PORT3:	digital I/O lines of PORT 3 (8-bit)
PORT4:	digital I/O lines of PORT 4 (8-bit)
PORT5:	digital I/O lines of PORT 5 (8-bit)
PORT6:	digital I/O lines of PORT 6 (8-bit)
PORT7:	digital I/O lines of PORT 7 (8-bit)
PORT8:	digital I/O lines of PORT 8 (8-bit)
AIN0:	analog input line AIN0 (float value)
AIN1:	analog input line AIN1 (float value)
AIN2:	analog input line AIN2 (float value)
AIN3:	analog input line AIN3 (float value)
AIN4:	analog input line AIN4 (float value)
AIN5:	analog input line AIN5 (float value)
AIN6:	analog input line AIN6 (float value)
AIN7:	analog input line AIN7 (float value)
AIN8:	analog input line AIN8 (float value)
AIN9:	analog input line AIN9 (float value)
AIN10:	analog input line AIN10 (float value)
AIN11:	analog input line AIN11 (float value)
VAGND:	analog reference voltage VAGND (float value)
VAREF:	analog reference voltage VAREF (float value)
S0IN:	serial input for SERIAL CHANNEL 0 (9-bit)
S0OUT:	serial output for SERIAL CHANNEL 0 (9-bit)

S1IN:	serial input for SERIAL CHANNEL 1 (9-bit)
S1OUT:	serial output for SERIAL CHANNEL 2 (9-bit)
XTAL:	Oscillator frequency
STIME:	serial Timing enable

80552.DLL

Driver file for the 80552 and similar chips.   The 80552.DLL driver file simulates the logical and timing behavior of the following hardware components:

- Timer 0
- Timer 1
- Timer 2 with Capture/Compare Registers
- A/D Converter
- Watchdog
- PWM Outputs
- Serial Interface 0
- Interrupt System

The following hardware components are not simulated:

- Serial Interface 1 (I2C Bus)

80552.DLL defines the following VTREG symbols (peripheral registers):

PORT0:	digital I/O lines of PORT 0 (8-bit)
PORT1:	digital I/O lines of PORT 1 (8-bit)
PORT2:	digital I/O lines of PORT 2 (8-bit)
PORT3:	digital I/O lines of PORT 3 (8-bit)
PORT4:	digital I/O lines of PORT 4 (8-bit)
PORT5:	digital I/O lines of PORT 5 (8-bit)
AIN0:	analog input line AIN0 (float value)
AIN1:	analog input line AIN1 (float value)
AIN2:	analog input line AIN2 (float value)
AIN3:	analog input line AIN3 (float value)
AIN4:	analog input line AIN4 (float value)
AIN5:	analog input line AIN5 (float value)
AIN6:	analog input line AIN6 (float value)
AIN7:	analog input line AIN7 (float value)
VAGND:	analog reference voltage VAGND (float value)
VAREF:	analog reference voltage VAREF (float value)
SIN:	serial input for SERIAL CHANNEL 0 (9-bit)
SOUT:	serial output for SERIAL CHANNEL 0 (9-bit)
XTAL:	Oscillator frequency
STIME:	serial Timing enable

80751.DLL

Driver file for the 80C750, 80C751, 80C752, and similar chips.   The 80751.DLL driver file simulates the logical and timing behavior of the following hardware components:

- ◆ Timer 0
- ◆ Timer 1
- ◆ Interrupt System

The following hardware components are not simulated:

- ◆ I2C Bus

80751.DLL defines the following VTREG symbols (peripheral registers):

          PORT0:   digital I/O lines of PORT 0 (8-bit)
          PORT1:   digital I/O lines of PORT 1 (8-bit)
          PORT2:   digital I/O lines of PORT 2 (8-bit)
           XTAL: Oscillator frequency

80410.DLL

Driver file for the 80CL410 and similar chips.   The 80410.DLL driver file simulates the logical and timing behavior of the following hardware components:

- ◆ Timer 0
- ◆ Timer 1
- ◆ Interrupt System
- ◆ Idle and Power-down Modes

The following hardware components are not simulated:

- ◆ Serial Interface (I2C Bus)

80410.DLL defines the following VTREG symbols (peripheral registers):

<div align="center">

PORT0:	digital I/O lines of PORT 0 (8-bit)
PORT1:	digital I/O lines of PORT 1 (8-bit)
PORT2:	digital I/O lines of PORT 2 (8-bit)
PORT3:	digital I/O lines of PORT 3 (8-bit)
XTAL:	Oscillator frequency

</div>

80781.DLL

Driver file for the 80CL781 and similar chips.   The 80781.DLL driver file simulates the logical and timing behavior of the following hardware components:

◆ Timer 0

◆ Timer 1

◆ Timer 2 with Reload/Capture register

◆ Serial Interface 0

◆ Interrupt System

The following hardware components are not simulated:

◆ Serial Interface 1 (I2C Bus)

80781.DLL defines the following VTREG symbols (peripheral registers):

PORT0: digital I/O lines of PORT 0 (8-bit)
PORT1: digital I/O lines of PORT 1 (8-bit)
PORT2: digital I/O lines of PORT 2 (8-bit)
PORT3: digital I/O lines of PORT 3 (8-bit)
SIN: serial input for SERIAL CHANNEL 0 (9-bit)
SOUT: serial output for SERIAL CHANNEL 0 (9-bit)
XTAL: Oscillator frequency
STIME: serial Timing enable

80320.DLL

Driver file for Dallas 80C320, 80C520 and similar chips.   The 80320.DLL driver file simulates the logical and timing behavior of the following hardware components:

◆ Timer 0

◆ Timer 1

◆ Timer 2 with Reload/Capture register

◆ Watchdog

◆ Serial Interface 0

◆ Serial Interface 1

◆ Interrupt System

◆ Power Saving Modes (Idle, Power-down)

80320.DLL defines the following VTREG symbols (peripheral registers):

PORT0:	digital I/O lines of PORT 0 (8-bit)
PORT1:	digital I/O lines of PORT 1 (8-bit)
PORT2:	digital I/O lines of PORT 2 (8-bit)
PORT3:	digital I/O lines of PORT 3 (8-bit)
S0IN:	serial input for SERIAL CHANNEL 0 (9-bit)
S0OUT:	serial output for SERIAL CHANNEL 0 (9-bit)
S1IN:	serial input for SERIAL CHANNEL 1 (9-bit)
S1OUT:	serial output for SERIAL CHANNEL 1 (9-bit)
XTAL:	Oscillator frequency
S0TIME:	Timing enable for SERIAL CHANNEL 0
S1TIME:	Timing enable for SERIAL CHANNEL 1