MCUez

Easy development software from the company that knows MCU hardware best

MCUez HC05/HC08 Debugger User's Manual
MCUEZDBG0508/D
Rev. 1

MCUEZDBG0508/D
Rev. 1

MCUez HC05/HC08 Debugger

User's Manual

MOTOROLA

Important Notice to Users

While every effort has been made to ensure the accuracy of all information in this document, Motorola assumes no liability to any party for any loss or damage caused by errors or omissions or by statements of any kind in this document, its updates, supplements, or special editions, whether such errors are omissions or statements resulting from negligence, accident, or any other cause. Motorola further assumes no liability arising out of the application or use of any information, product, or system described herein; nor any liability for incidental or consequential damages arising from the use of this document. Motorola disclaims all warranties regarding the information contained herein, whether expressed, implied, or statutory, *including implied warranties of merchantability or fitness for a particular purpose*. Motorola makes no representation that the interconnection of products in the manner described herein will not infringe on existing or future patent rights, nor do the descriptions contained herein imply the granting or license to make, use or sell equipment constructed in accordance with this description.

The computer program contains material copyrighted by Motorola, Inc., first published in 1997, and may be used only under a license such as the License For Computer Programs (Article 14) contained in Motorola's Terms and Conditions of Sale, Rev. 1/79.

Trademarks

This document includes these trademarks:

MCUez is a trademark of Motorola, Inc.

Microsoft Windows is a registered trademark of Microsoft Corporation.

WinEdit is a trademark of Wilson WindowWare.

List of Sections

List of Sections

Table of Contents

Section 1. General Information

Section 2. Getting Started and User Interface

Section 3. Component Windows

Section 4. Operating Procedures

Section 5. Command Reference

Appendix A. Register Description File

Appendix B. C Source-Level Debugging

Appendix C. MMEVS0508 Target

Appendix D. MMDS0508 Target

Index

Table of Contents

List of Figures

List of Figures

Figure	Title	Page

List of Figures

List of Tables

List of Tables

Section 1. General Information

1.1 Contents

1.2 Introduction

Motorola's MCUez debugger is a multipurpose tool used for various tasks in the embedded systems and industrial control world, such as emulation and/or cross-debugging of an embedded application or hardware design.

The debugger consists of the engine and a set of component windows bound to the task they perform (for example, a debugging session). The debugger engine is the heart of the system. It monitors and coordinates the tasks of the components. Each component has its own functionality (for example, source level debugging).

This manual describes how to use the MCUez debugger. A brief description of each section is provided here.

- **Section 1. General Information** provides document conventions, a functional description of operation, and support information.
- **Section 2. Getting Started and User Interface** provides information on the toolbar, status bar, object information bar, drag and drop operation, and the MCUez menus.
- **Section 3. Component Windows** provides detailed information on each basic component and how to use them.
- **Section 4. Operating Procedures** contains procedures on how to use the MCUez debugger.
- **Section 5. Command Reference** provides detailed information on all MCUez commands. An example is provided with each description.
- **Appendix A. Register Description File** defines the syntax for creating a register file.
- **Appendix B. C Source-Level Debugging** provides information on how to use the debugger when debugging C code at the source level.
- **Appendix C. MMEVS0508 Target** provides information about the target interfaces related to the MMEVS0508 modular evaluation system for the MC68HC05 and MC68HC08 MCU Families.
- **Appendix D. MMDS0508 Target** provides information about the target interfaces related to the modular development system for the MC68HC05 and MC68HC08 MCU Families.

1.3 Document Conventions

This section describes terms and styles used throughout the manual.

1.3.1 General Term

This general term is used in this document:

Key1 + Key2

The + (plus) sign means that Key1 is held down while Key2 is pressed.

1.3.2 Mouse Operations

This list describes the terminology used to define mouse operations.

- Click — Implies to click the left mouse button once
- Right click — This click operation is done with the right mouse button.
- Double click — This indicates to double click the left mouse button.
- Drag — Press and hold down the left mouse button while dragging the mouse. The object will move with the mouse cursor and drop when the mouse button is released.

1.3.3 Typographic Styles in This Manual

These typographic conventions are used in this manual:

- **Bold face** type is used for literal strings that must be used exactly as shown in the example and for the names of menus, windows, dialog boxes, icons, and buttons.
- `Courier` type face is used for all C-code program listings, command lines, and directories.
- *Italics* are used where the string is a place holder that may be substituted for a string of the user's own design.
- Variable user inputs are in *`Courier`* italics.
- Filenames are in italics with all lower case letters, for example, *proj.ext.*

These styles are used in this manual to define notational conventions:

- **Numeric constants —** Numeric constants are displayed in the C language format. Constants that are in the 0x format are hexadecimal. Constants that have no prefix are assumed to be decimal. The notation k, unless to denote a frequency setting in kilohertz, defines a number multiplied by 1024.
- **Function prototypes —** Structures and function call descriptions are given in terms of the C language. This does not limit the implementation of calling programs to C, but it is the calling routine’s responsibility to provide the correct link to these routines.

Section 2. Getting Started and User Interface

2.1 Contents

2.2 Introduction

This section describes how to connect the target hardware, start the debugger, and use the graphical user (GUI) interface environment.

This section also introduces target interfaces related to the modular development system for the MC68HC05 and MC68HC08 MCU Families.

The MCUez debugger uses Motorola's MMDS0508 or MMEVS0508 to communicate with a target system. The MMDS0508 is an emulator system that provides emulation memory and a bus state analyzer for MCUs with a CPU05 or CPU08. The MMEVS0508 is an emulator system designed by Motorola for MCUs that use a CPU05 or CPU08 to provide emulation memory.

Using the MMDS interface, an external target system can download an executable program, execute it, and relay the results of the target system to the debugger.

The debugger fully supervises and monitors the target system's MCU. It also controls CPU activities such as read and write in internal/external memory (even when the CPU is running) and single-step/run/stop processes.

2.3 MMDS/MMEVS Hardware Connection

The MMDS or MMEVS target is connected to the RS-232 serial port of the host computer. Use the serial link cables provided to connect the host to the target. The host computer is configured as a data terminal.

For additional information about the MMEVS and MMDS targets, refer to **Appendix C. MMEVS0508 Target** and **Appendix D. MMDS0508 Target**, respectively.

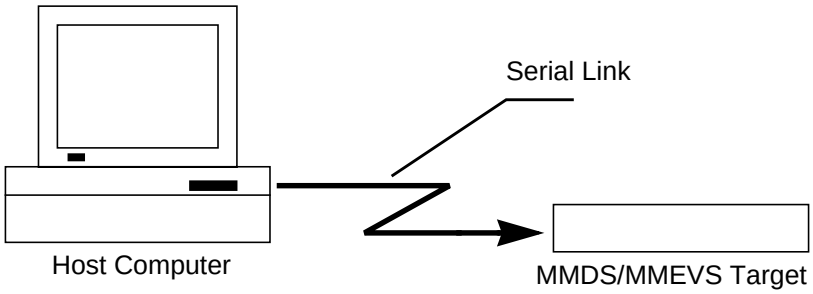


Figure 2-1. Hardware Connection

2.4 Starting the MCUez Debugger

After installing the MCUez software, start the **MCUez Shell**. Typically, this is done from the Windows **Start** menu. For example, select **Start** > **Programs** > **MCUez** > **MCUez Shell**.

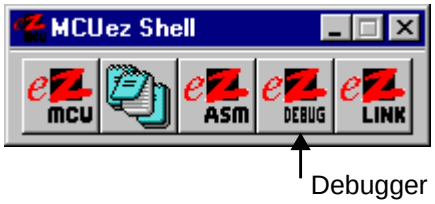


Figure 2-2. MCUez Shell

Before launching the debugger, power up the MMDS or MMEVS. Then click on the **ezDEBUG** icon in the shell. The debugger will load the target firmware and the memory map configuration file (*.mem*), also known as the personality file. A register description file (*.reg*) is also loaded. The *.mem* files are located in the default installation directory *\prog\mem* and *.reg* files are located in *\prog\reg*.

Memory map files are related to the specific MCU used in the MMDS or MMEVS target.

CAUTION: *If the corresponding memory file is missing or has been overwritten, the debugger will not connect with the target.*

The I/O (input/output) register definitions are loaded from a register description file (*.reg*) located in the *MCUez\prog\reg* directory. These files are related to the specific MCU used in the target. The filename is represented as *MCUxxxx.reg*, where *xxxx* is the MCU Id in hexadecimal representation. The register file enables register names to be used as command parameters or as expression operands. If no appropriate file is found, an error message is displayed, corresponding information will be missing, and related commands may not operate properly.

2.4.1 Configuration File

The *project.ini* configuration file located in the project directory contains information used to configure the debugger and specify a default target. Current target options are ESL (emulation server library) or D-Bug12. ESL is used for MMDS or MMEVS. Specifying ESL as the default target will load the emulator server library with functions used to operate with MMDS or MMEVS and cause the debugger to automatically detect the hardware target connection. If no target is specified as the default, the user must set a target from the **Component | Set Target...** menu option. D-Bug12 is used for HC12 target interfaces as described in the *MCUez HC12 Debugger User's Manual*, Motorola document order number MCUEZDBG12/D.

Example *project.ini* file:

```
[DEFAULTS]
Window0=Source     0   0  50  40
Window1=Assembly  50   0  50  40
Window2=Register  50  40  50  30
Window3=Memory    50  70  50  30
Window4=Data       0  40  50  25
Window5=Command    0  65  50  20
Window6=Module     0  85  50  15
Target=ESL
MainFrame=0,1,-1,-1,-1,-1,204,174,866,732

[Motorola ESL]
ServerType=0
MapFileName=C:\MCUEZ\PROG\MEM\00BFEV00.MEM
...
```

2.5 Loading a Target

If no default target is set in the *project.ini* file, select **Set Target...** from the **Component** menu and choose **ESL** from the list.

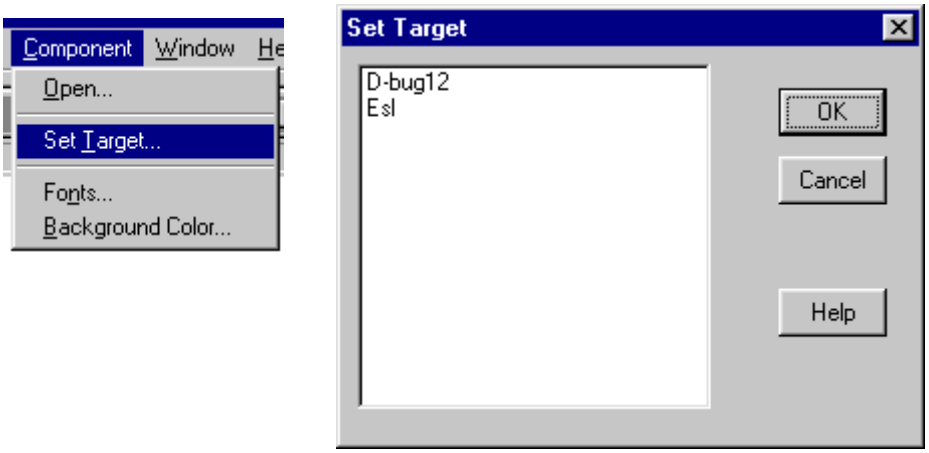


Figure 2-3. Setting a Target

The ESL driver automatically attempts to locate the target, for example, the MMDS0508 or MMEVS0508. If the target is successfully loaded, the related menu appears between the **Run** and **Component** menus, for example MMDS0508. The debugger status bar displays information related to the connected target.

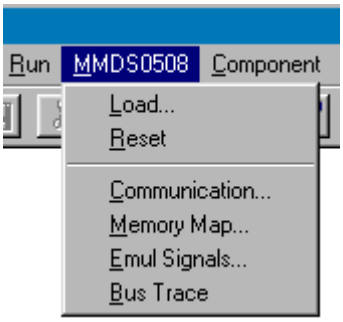


Figure 2-4. MMDS0508 Menu

If the ESL driver cannot locate the target, an error message and the **Communication** dialog box appear, indicating that the target is not connected.

The **ESL** menu replaces the **Target** menu. The **Communication** dialog box can also be displayed by selecting the **ESL | Connect** menu option.

Figure 2-5. Establishing a Connection

2.5.1 Communication Dialog Box

The default is COM1 at a baud rate of 9600. To change settings, enter another device name in the edit box, select a baud rate, and click **Connect**. After connecting with the target, new settings are saved in the *project.ini* file.

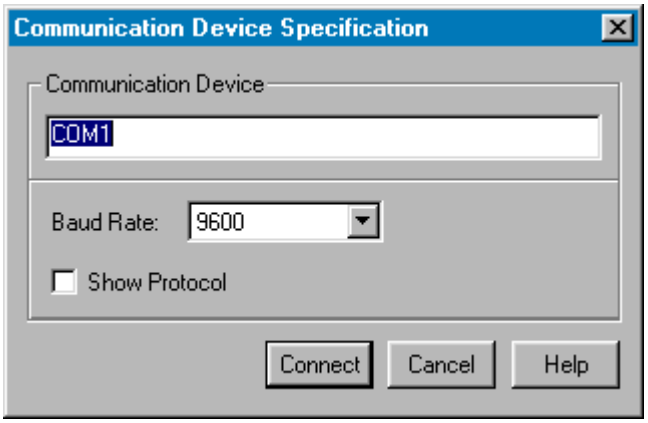


Figure 2-6. Communication Device Dialog Box

The MCUez debugger uses this data format:

- Eight data bits
- One stop bit, no parity
- Variable baud rate

2.6 Graphical Environment

The debugger consists of an arrangement of component windows that reside in the debugger main window. The main window provides a toolbar, status bar, object information bar, and menu bar for component windows (see **Figure 2-7**).

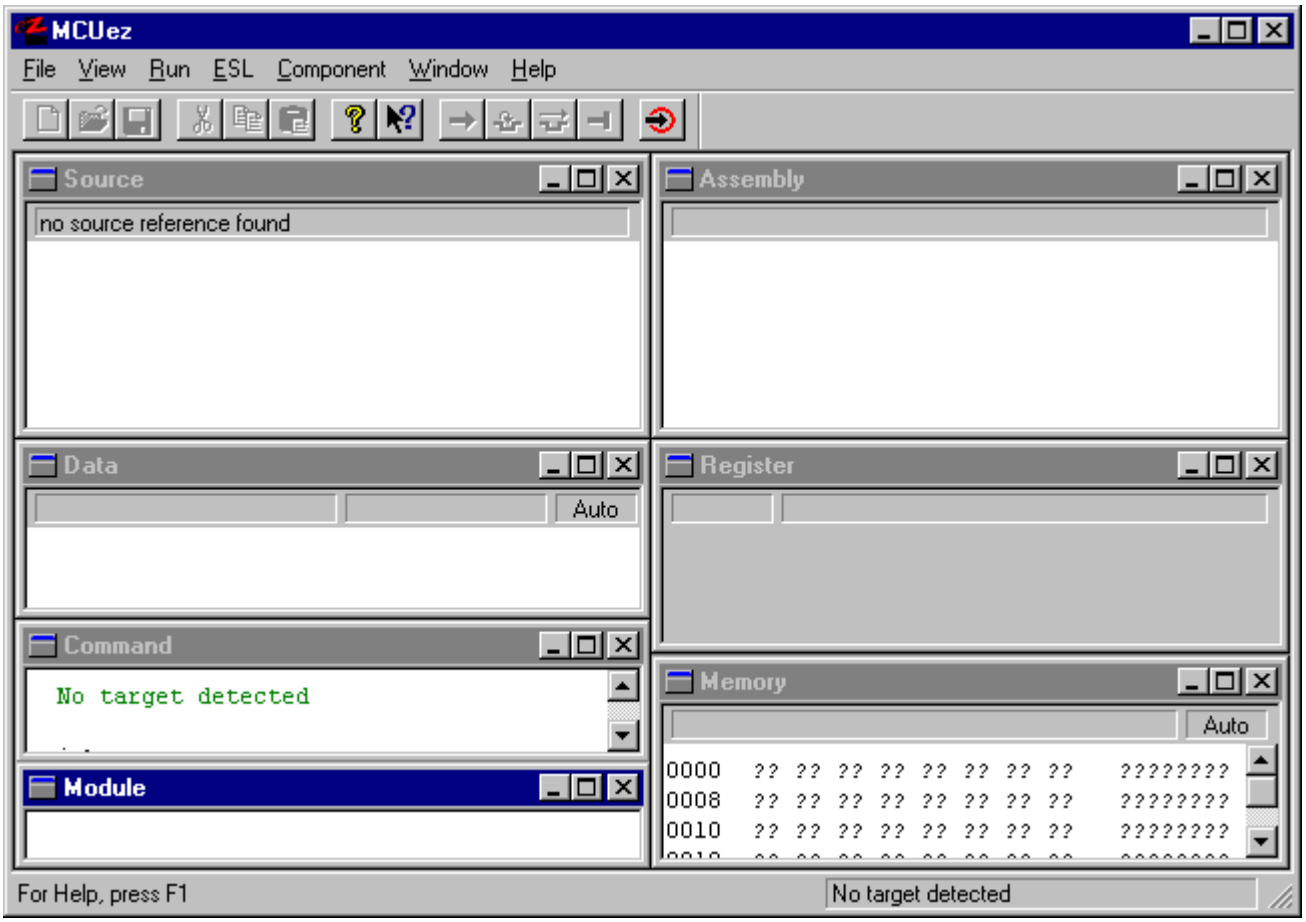


Figure 2-7. MCUez Debugger and Component Windows

The main window manages the layout of the different component windows. The component windows can be:

- Multiple windows tiled in the debugger main window
- Automatically resized with the main window
- Overlapped
- Minimized
- Maximized

2.6.1 Toolbar

A brief description is available when the mouse pointer is pointed at an icon. The toolbar is illustrated in **Figure 2-8**.

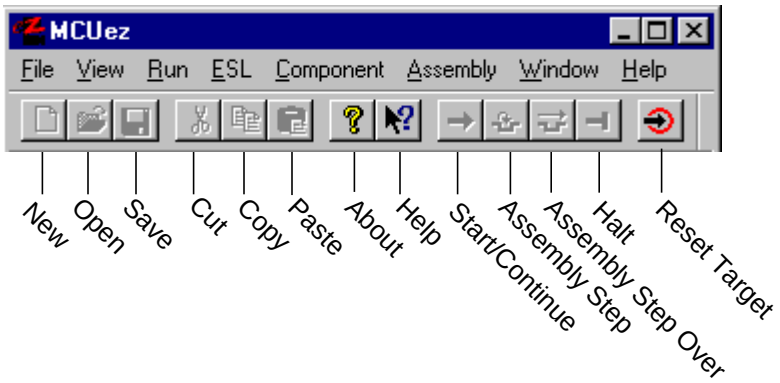


Figure 2-8. MCUez Debugger Toolbar

2.6.2 Status Bar

The status bar at the bottom of the debugger window (see **Figure 2-9**) contains a help line that displays a brief explanation when the mouse cursor is positioned over a button or menu item. Also shown is target-specific information.

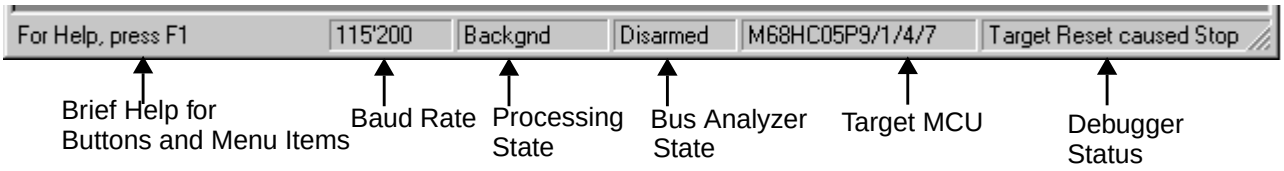


Figure 2-9. MCUez Debugger Status Bar

2.6.2.1 Debugger Status

Ready	The MCUez debugger is ready and waits until a new target or application is loaded. This message is generated once the debugger has been started.
Halt	Execution of the application has been stopped by a request of the application. The Motorola predefined macro HALT has been reached in the application during execution.
Running	The application currently is executing in the debugger.
Halted	Execution of the application has been stopped on user request. The menu entry **Run | Halt** or the **Halt** icon in the toolbar has been selected. A “Halted” message is also displayed when the first **Assembly Step** is executed either from the **Run** menu or push button. Subsequent **Assembly Steps** result in a “trace” message.
Stepped	Execution of the application has been stopped after a single step at source level. The menu entry **Run | Single Step** or the **Single Step** icon in the toolbar has been selected. This message is relevant only when debugging at C source level.
Stopped	Execution has been stopped after a step out from a function call. The menu entry **Run | Step Out** or the **Step Out** icon in the toolbar has been selected. This message is relevant only when debugging at C source level.
Traced	Execution of the application has been stopped after a single step at assembler level. The menu entry **Run | Assembly Step** or the **Assembly Step** icon in the toolbar has been selected. This message is relevant only when debugging at C source level.
Breakpoint	Application has been stopped because a breakpoint has been reached.

2.6.2.2 CPU Specific Messages

Some error messages depend on the CPU used. These are messages related to exceptions. The MCUez debugger makes a distinction between predefined exceptions (that have a specific meaning for all derivates in the CPU family) and user-defined exceptions. A user-defined exception can be freely configured by the user and does not have the same meaning for all derivates in the CPU family.

The exception message format is:

Exception `<string>|<number>`

where:

`string` describes the reason for the exception. This string is only specified when a predefined exception is detected.

`number` is the entry in the vector table that generates the exception. This number is only specified when a user-defined exception is detected.

Two exceptions are treated differently: An address error exception and a bus error exception for the target processor. Check the target hardware manual for an explanation of the exception.

Other exceptions can be generated for a vector that are not associated with an interrupt function. Possible reasons:

- An interrupt source was not disabled. Insert code to disable the interrupt source in the application.
- The corresponding entry in the vector table was not initialized with the address of the function associated with the interrupt. Initialize the vector table.

2.6.3 Object Information Bar

The object information bar provides immediate information about an object selected in a component window (**Figure 2-10**).

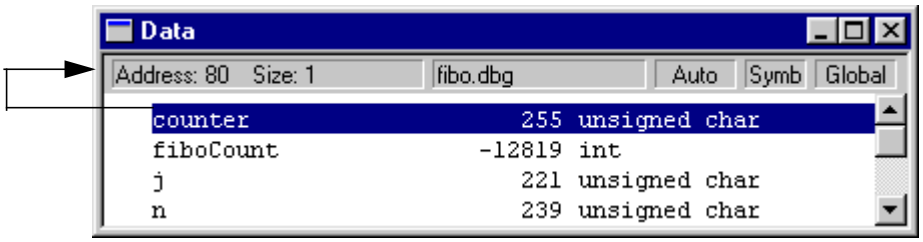


Figure 2-10. Debugger Object Information Bar

2.6.4 Menu Bar

Figure 2-11 shows an example of the debugger main menu with the MMDS0508 target menu.

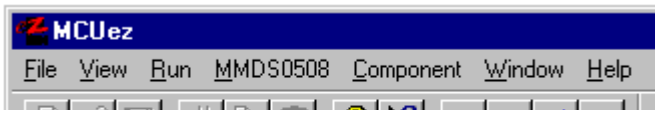


Figure 2-11. Debugger Main Menu

2.6.4.1 File Menu

The **File** menu is dedicated to the debugger. Use the **File** menu to exit the debugger, as shown in **Figure 2-12**.

Figure 2-12. File Menu

2.6.4.2 View Menu

The **View** menu contains two options:

- Check/uncheck toolbar to display or hide it
- Check/uncheck status bar to display or hide it

Figure 2-13. View Menu

2.6.4.3 Run Menu

The **Run** menu is used for debug operations. Table 2-1 defines all **Run** menu commands.

Table 2-1. Run Menu

Menu Entry	Description
Start/Continue	Starts or continues execution of the loaded application until a breakpoint is reached, a runtime error is detected, or the user stops the application using **Halt**
Halt	Interrupts and halts a running application enabling examination of the state of each variable in the application, setting of breakpoints, and inspection of the source code
Assembly Step	If the application is halted, this command performs a single step; for instance, execution continues for one CPU instruction from the point it was halted.
Assembly Step Over	Similar to the **Assembly Step** command, but does not step into called functions (steps over subroutine call instructions)
Breakpoints	Opens the **Breakpoints Setting** dialog, which lists breakpoints defined in the application and allows their properties to be modified. Breakpoints also can be set in the **Source** and **Assembly** component windows.

The menu entries (except breakpoints) have an associated toolbar button. **Figure 2-14** illustrates the **Run** menu.

Figure 2-14. Run Menu

2.6.4.4 Target Menu

The **Target** menu appears between the **Run** and **Component** menus when no target is specified as the default in the *project.ini* file and no target component has been set. **Select Target | Load...** to display the load executable file message. Click **Yes** to display the set target listbox, enabling selection of a target, for example ESL. After it is connected to a hardware target, the **Target** menu is replaced by a menu with the name of the hardware target, for example MMDS0508.

The user can also select **Component | Set Target...** to connect to a target.

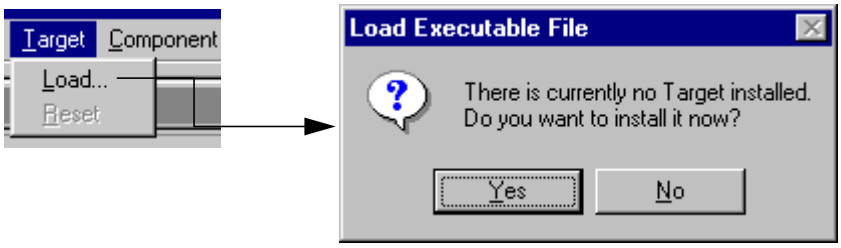


Figure 2-15. Target Menu

2.6.4.5 Component Menu

Open additional component windows by selecting the **Component | Open...** menu entry. Select a component window from the list of components and click **OK**. Component windows are described in **Section 3. Component Windows**. **Figure 2-16** illustrates the **Component** menu.

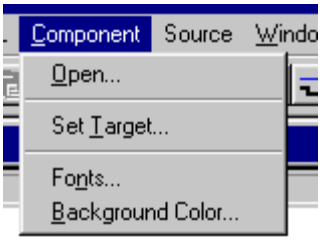


Figure 2-16. Component Menu

Select **Component | Set Target...** to set the preferred target.

Select **Component | Fonts...** to open a standard **Font Selection** dialog and select the font characteristics to use in the component windows.

Select **Component | Background Color...** to open the color selection dialog. This enables setting of the background color for component windows.

2.6.4.6 Window Menu

The **Window** menu (**Figure 2-17**) sets the general arrangement of component windows and loads or stores custom arrangements.

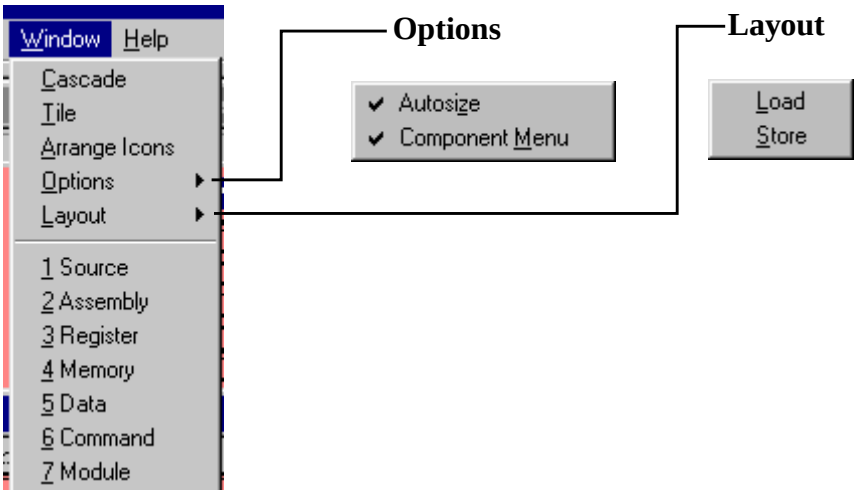


Figure 2-17. Window Menu

Check **Autosize** to automatically resize component windows when the debugger main window is resized. Check **Component Menu** to display the menu associated with the currently selected component window. For example, if the **Source** window is selected, the **Source** menu appears in the main menu.

Select **Window | Layout** to load or store arrangements in a *.hwl* file.

2.6.4.7 Help Menu

The **Help** menu (**Figure 2-18**) provides on-line help and version information.

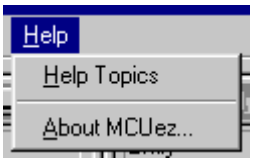


Figure 2-18. Help Menu

2.7 Drag and Drop

The user can drag objects from one component window to another. This is defined as “drag and drop.” For example, the user can display the memory layout corresponding to the address held in a register by dragging the address from the register component (for instance, EA) to the memory component. See the example in **Figure 2-19**.

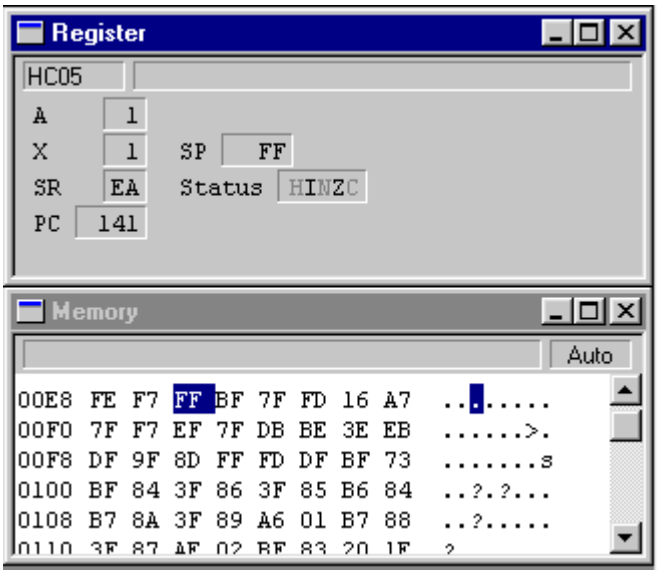


Figure 2-19. Component Drag and Drop Example

2.7.1 Drag and Drop an Object

To drag and drop, follow this step-by-step procedure:

1. Select the component window containing the object to drag.
2. Make sure the destination component window is visible.
3. Select and hold the object with the left mouse button.
4. Drag the object into the destination component window and release the mouse button.

The following sections describe the possible combinations of drag and drop between components and associated actions.

If the destination of a dragged item is not possible, the cursor is displayed as a circle with a line through it.

2.7.2 Dragging from the Assembly Component

The drag and drop actions shown in **Table 2-2** are possible from the assembly component.

Table 2-2. Dragging from the Assembly Component

Destination Component	Action
Command Line	The command line component appends the address of the pointed to instruction to the current command.
Memory	Dumps memory starting at the selected instruction PC. The PC location is selected in the memory component.
Register	Loads the destination register with the address of the selected instruction.
Source	Source component scrolls to the corresponding source statement and highlights it.

2.7.3 Dragging into the Assembly Component

The actions shown in **Table 2-3** occur when dragging and dropping into the assembly component.

Table 2-3. Dragging into the Assembly Component

Source Component	Action
Source	Displays disassembled instructions starting at the first high level language instruction selected. The assembler instructions corresponding to the selected high level language instructions are highlighted in the assembly component.
Memory	Displays disassembled instructions starting at the first address selected. Instructions corresponding to the selected memory area are highlighted in the assembly component.
Register	Displays disassembled instructions starting at the address stored in the source register. The instruction starting at the address stored in the register is highlighted.

2.7.4 Dragging from the Data Component

The actions shown in **Table 2-4** occur when dragging from the data component.

Table 2-4. Dragging from the Data Component

Destination Component	Action
Command Line	Dragging the name appends the address range of the variable to the current command in the command line window. Dragging the value appends the variable value to the current command in the command line window.
Memory	Dumps memory starting at the address where the selected variable is located. The memory area where the variable is located is selected in the memory component. Dragging from a data value in the memory component is not allowed.
Register	Dragging the name loads the destination register with the address of the selected variable. Dragging the value loads the destination register with the value of the variable.

NOTE: *The user can drag either a variable name or a variable value. Both operations are possible. Dragging the variable name drags the address of the variable.*

Expressions are evaluated at run time; therefore, they do not have a location address associated with them. Without a location address, the user cannot drag an expression name into another component, although expression values can be dragged to other components.

2.7.5 Dragging into the Data Component

Table 2-5 shows all options available when dragging into the data component.

Table 2-5. Dragging into the Data Component

Source Component	Action
Source	A selection in the source window is considered as an expression in the data window, as if it had been entered through the expression editor of the data component (refer to ***3.3.6.1 Expression Editor***).
Module	Displays global variables from the selected module in the data component

2.7.6 Dragging from the Source Component

Table 2-6 describes the actions taken when dragging from the source component.

Table 2-6. Dragging from the Source Component

Destination Component	Action
Assembly	Displays disassembled instructions starting at the first high level language instruction selected. The assembler instructions corresponding to the selected high level language instructions are highlighted in the assembly component
Register	Loads the destination register with the PC of the first instruction selected
Data	A selection in the source window is considered as an expression in the data window, as if it had been entered through the expression editor of the data component (refer to ***3.3.6.1 Expression Editor***).

2.7.7 Dragging into the Source Component

Table 2-7 decribes the action taken when dragging into the source component.

Table 2-7. Dragging into the Source Component

Source Component	Action
Assembly	Source component scrolls to the source statements corresponding to the pointed to assembly instruction and highlights it.
Memory	Displays high level language source code starting at the first address selected. The instructions corresponding to the selected memory area are greyed in the source component.
Module	Displays source code from the selected module

2.7.8 Dragging from the Memory Component

Table 2-8 describes the action taken when dragging from the memory component.

Table 2-8. Dragging from the Memory Component

Destination Component	Action
Assembly	Displays disassembled instructions starting at the first address selected. The instructions corresponding to the selected memory area are highlighted in the assembly component.
Command Line	Appends the selected memory range to the command line window
Register	Loads the destination register with the start address of the selected memory block
Source	Displays high level language source code starting at the first address selected. Instructions corresponding to the selected memory area are greyed in the source component.

2.7.9 Dragging into the Memory Component

Table 2-9 describes the action taken when dragging into the memory component.

Table 2-9. Dragging into the Memory Component

Source Component	Action
Assembly	Dumps memory starting at the selected instruction PC. The PC location is selected in the memory component.
Data	Dumps memory starting at the address of the selected variable. The memory area where the variable is located is selected in the memory component.
Register	Dumps memory starting at the address stored in the selected register. The corresponding address is selected in the memory component.
Module	Dumps memory starting at the address of the first global variable in the module. The memory area where this variable is located is selected in the memory component.

2.7.10 Dragging from the Register Component

Table 2-10 shows the options available when dragging from the register component.

Table 2-10. Dragging from the Register Component

Destination Component	Action
Assembly	Assembly component receives an address range, scrolls to the corresponding instruction, and highlights it.
Memory	Dumps memory starting at the address stored in the selected register. The corresponding address is selected in the memory component.
Command Line	The address stored in the pointed to register is appended to the current command.

2.7.11 Dragging into the Register Component

Table 2-11 shows the options available when dragging into the register component.

Table 2-11. Dragging into the Register Component

Source Component	Action
Assembler	Loads the destination register with the address of the selected instruction
Data	Dragging the name loads the destination register with the start address of the selected variable. Dragging the value loads the destination register with the value of the variable.
Source	Loads the destination register with the PC of the first instruction selected
Memory	Loads the destination register with the start address of the selected memory block

2.7.12 Dragging from the Module Component

Table 2-12 shows the options available when dragging from the module component.

Table 2-12. Dragging from the Module Component

Destination Component	Action
Data | Global	Displays the global variables from the selected module in the data component
Memory	Dumps memory starting at the address of the first global variable in the module. The memory area where this variable is located is selected in the memory component.
Source	Displays source code from the selected module

NOTE: *Nothing can be dragged into the module component.*

Section 3. Component Windows

3.1 Contents

3.2 Introduction

This section describes the various component windows and associated menus.

3.3 Components

The MCUez debugger operates in an environment constructed of windows that represent components of the target system. Component windows are applications loaded at run-time and have access to the target interface, the symbol table, and other global facilities. Additional component windows can be opened by selecting the **Component|Open...** menu option. **Figure 3-1** shows the **Open Window Component** dialog box. Component windows are implemented as dynamic link library files with the extension *.wnd*.

CPU (central processor unit) components handle processor-specific operations such as register naming, instruction decoding (disassembly), and stack tracing. CPU components are reflected in the **Register** and **Memory** component windows. Applicable CPU components are loaded when connection with the target is established.

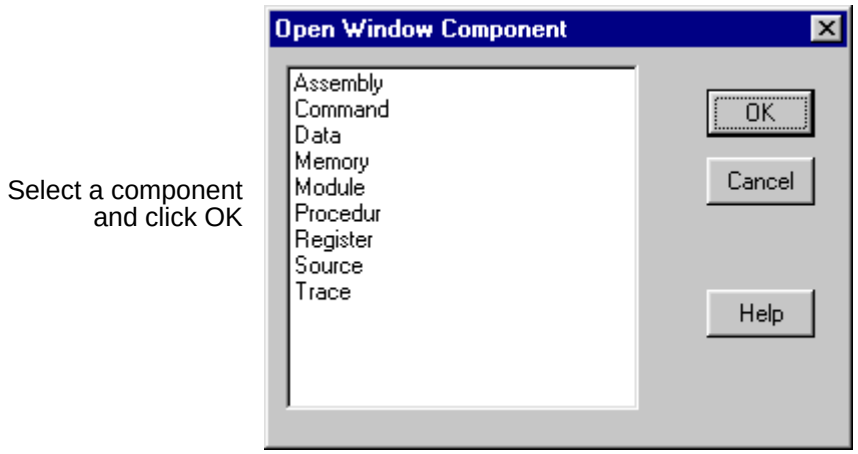


Figure 3-1. Open Window Component Dialog Box

3.3.1 Component Window Menu

Each component window has two menus. One menu can appear on the debugger main menu bar and the other is a pop-up menu that can be opened by clicking the right mouse button in the active component window.

The active component menu appears between the **Component** and **Window** menus on the debugger main menu bar. For example, if the **Source** component window is selected, the **Source** menu will appear. If the **Data** component window is selected, the **Data** menu will appear as shown in **Figure 3-2**.

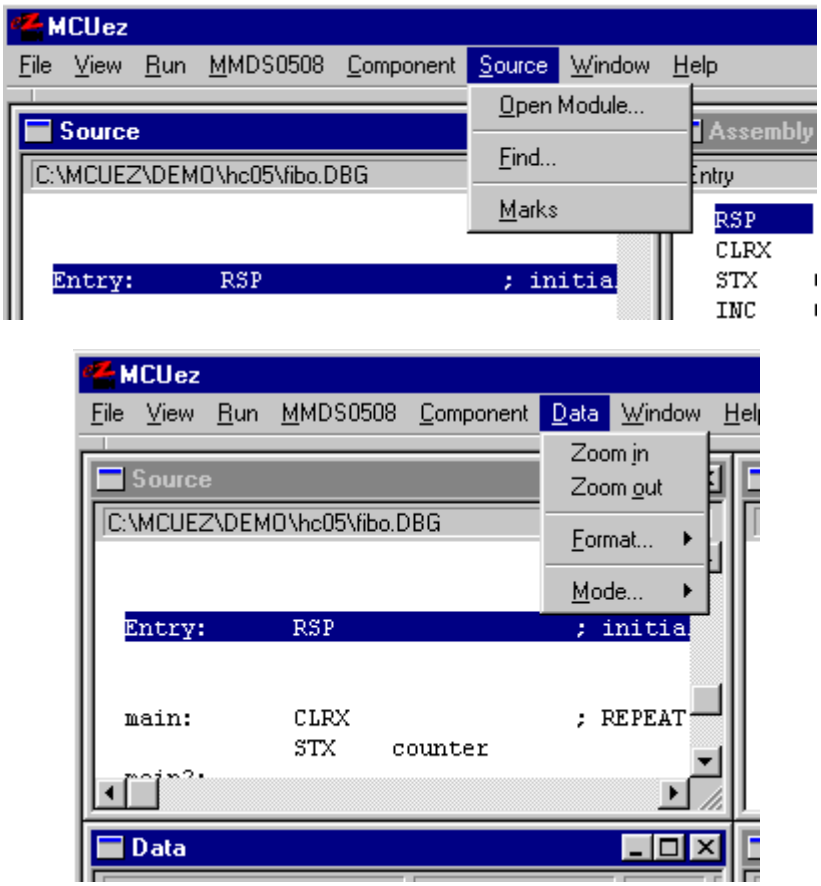


Figure 3-2. Active Component Menu

The pop-up menu is a dynamic context-sensitive menu. It contains entries for the currently active component. Pop-up menu entries differ to reflect operations that can be performed on the object pointed to by the mouse. For example, if a breakpoint is pointed to, menu options are available to delete, enable, or disable the breakpoint. **Figure 3-3** shows the **Source** component pop-up menu.

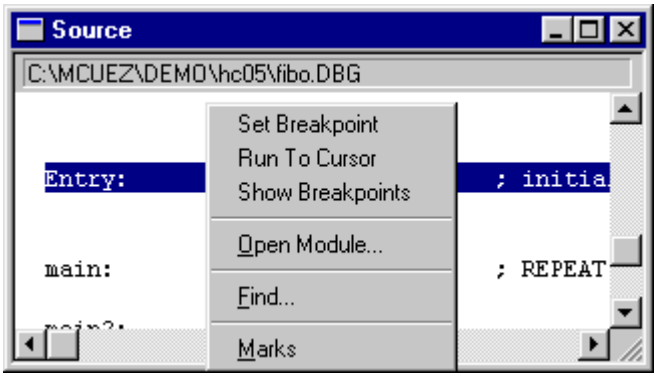


Figure 3-3. Source Component Pop-up Menu

3.3.2 Source Component Window

The **Source** component window displays the program source code (application file). It enables the user to view, change, monitor, and control the current execution location in the program. A word is selected by placing the mouse pointer on the word, then double clicking the left mouse button. A section of code is selected by holding down the left mouse button and dragging the mouse across the code to highlight the selected area.

NOTE: *Text displayed in the* **Source** *component window cannot be edited. The* **Source** *component window is a file viewer only.*

The range of source code covered by the current program counter is selected (for example, in a C source: `fib1=fib2;`). The selected source code range is highlighted to match the highlighted **Assembly** component instruction set.

If breakpoints have been set in the program, they will be marked with a special symbol to indicate the nature of the breakpoint. See **3.3.2.1 Breakpoints**.

If execution has stopped, the current position is marked in the **Source** component window by highlighting the corresponding statement. The complete path of the displayed source file is written in the object information bar. **Figure 3-4** shows the **Source** component window.

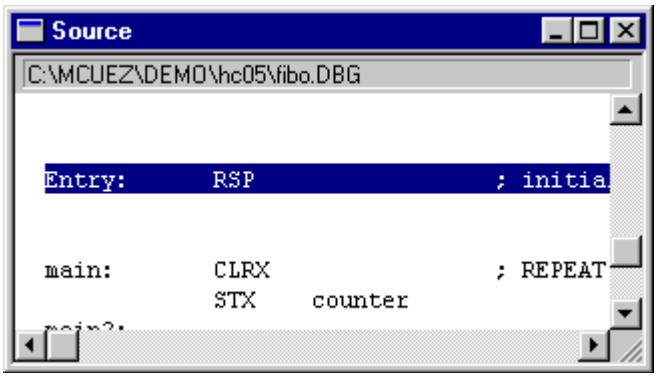


Figure 3-4. Source Component Window

Figure 3-5 shows the **Source** component menu and its associated pop-up menu obtained by clicking the right mouse button.

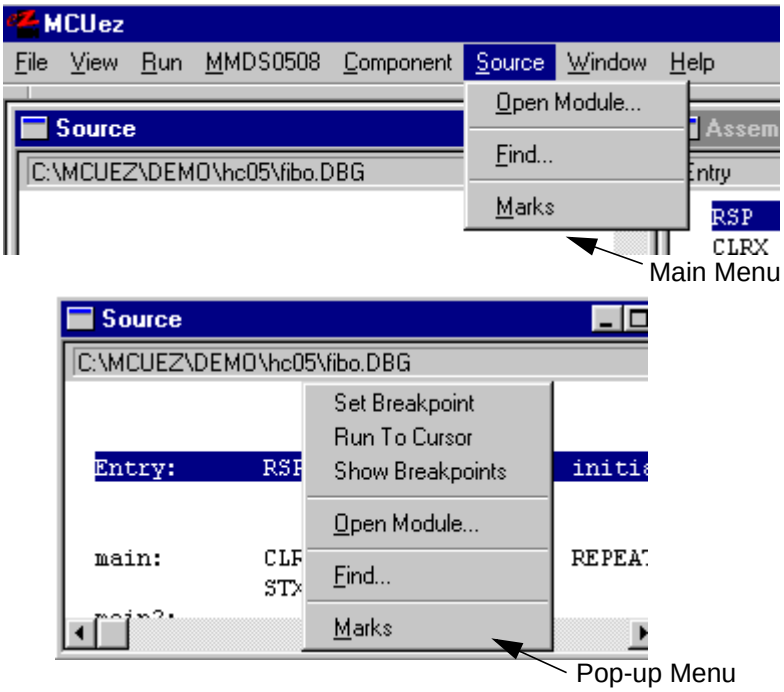


Figure 3-5. Source Component Main Menu and Pop-up Menu

The following describes **Source** component menu options.

- Menu bar
  - **Open Module** — Opens a dialog which lists all source files bound to the application currently loaded. Choose the module to be viewed in the source window.
  - **Find** — Opens a dialog box prompting for a string and then searches the file displayed in the source component for the specified string

    To start searching, click **Find Next** and the search begins at the current selection or first line visible in the source component. **Up/down** radio buttons enable the search backward or forward. If the string is found, it is highlighted. If the string is not found, a message will be displayed.
  - **Marks** — Displays upside down check marks at source lines where breakpoints may be set
- Pop-up menu
  - **Set Breakpoint** — Appears in the pop-up menu if no breakpoint is set or disabled. When selected, sets a permanent breakpoint at this position. If program execution reaches this instruction, the program is halted and the current program state is displayed in all component windows.
  - **Delete Breakpoint** — Appears in the pop-up menu for a breakpoint (pointed at with mouse cursor) that is set or disabled. Deletes a selected breakpoint
  - **Enable Breakpoint** — Appears in the pop-up menu for a pointed to breakpoint that is disabled. Enables a selected breakpoint.
  - **Disable Breakpoint** — Appears if a breakpoint is set. Disables a selected breakpoint
  - **Run to Cursor** — Sets a temporary breakpoint at the nearest code position and continues program execution. When program execution reaches this instruction, the program is halted and the current program state is displayed in all component windows. Temporary breakpoints are removed automatically as soon as they are reached.

NOTE: *If there is a disabled breakpoint at this same position, the temporary breakpoint will also be disabled and the program will not halt.*

- **Show Breakpoints** — Opens the **Breakpoints Setting** dialog box, which lists breakpoints defined in the application and allows properties to be modified
- **Open Module** — Opens a dialog that lists all source files bound to the application currently loaded. Select the module to be viewed in the source window.
- **Find** — Opens a dialog box prompting for a string and then searches the file displayed in the source component for the specified string
- **Marks** — Displays upside down check marks at source lines where breakpoints may be set

3.3.2.1 Breakpoints

If breakpoints have been set in the program, they will be marked with a specific symbol.

- Temporary breakpoint symbol :  A lighter color encased by a darker color with the lighter color usually being yellow and the darker being red
- Permanent breakpoint symbol:  Solid red
- Disabled breakpoint symbol:  Halftone and light red

If execution has stopped, the current position is marked in the source component by highlighting the corresponding statement.

3.3.2.2 Decoding Instructions

To disassemble code, select a range of instructions in the source component and drag it into the assembly component. The corresponding range of code is highlighted in the **Assembly** component window.

In the source component, place the mouse cursor on a line of code. Click and hold the left mouse button and press the R key. Release the R key, then release the left mouse button to highlight a code range in the assembly component that corresponds to the code pointed to in the source component. The code range in the source component is also highlighted. **Figure 3-6** illustrates this process.

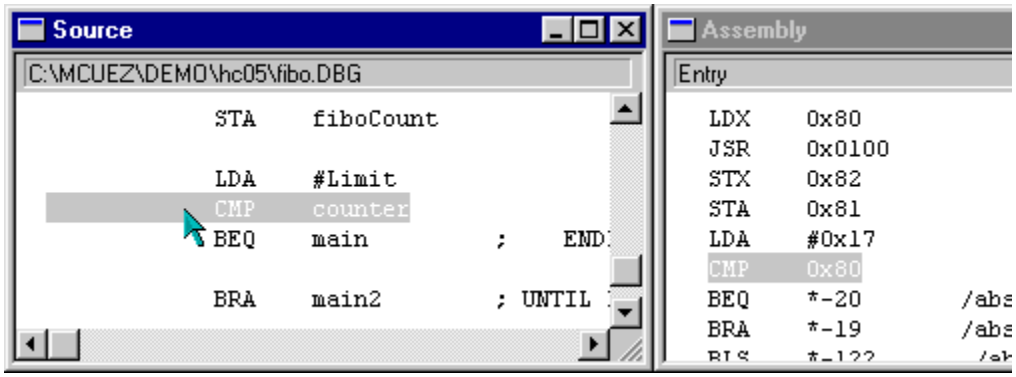


Figure 3-6. Online Disassembly

3.3.2.3 Find Dialog Box

Enter the string to search for in the **Find what** edit box. To start searching, click **Find Next**. The search begins at the current selection or first line visible in the source component when nothing is selected.

Figure 3-7 shows the **Find** dialog box.

Figure 3-7. Find Dialog Box

Find dialog box options are:

- **Match whole word only**: If this box is checked, only strings separated by special characters are recognized.
- **Match case**: If checked, the search is case sensitive.
- The **Up / Down** buttons will enable the search backward or forward. If the string is found, it is highlighted. If the string is not found, a message is displayed.

NOTE: *If an item (single word or source section) has been selected in the source component before opening the* **Find** *dialog, the first line of the selection will be automatically copied into the* **Find what** *edit box.*

3.3.3 Assembly Component Window

The assembly component displays program code in disassembled form. The assembly component function is similar to that of the source component, enabling the user to view, change, monitor, and control the current location of program execution.

The assembly component contains all disassembled instructions generated by the application. Each disassembled line shows this information:

- Address
- Machine code
- Instruction
- Absolute address for branch instruction

Per default, the instruction and absolute address for a branch instruction are visible. The program instruction and absolute address also can be viewed by selecting the corresponding menu entry.

Breakpoints set in the application are marked with a special symbol. If execution has stopped, the current position is marked by highlighting the corresponding instruction. The object info bar displays the procedure name that contains the currently selected instruction. **Figure 3-8** shows the assembly component window.

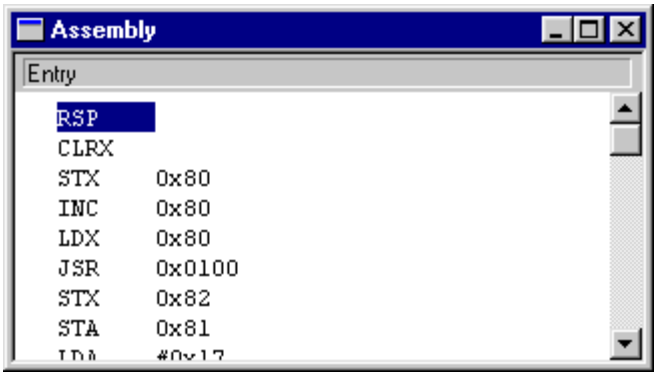


Figure 3-8. Assembly Component Window

Figure 3-9 shows the **Assembly** component main menu and associated pop-up menu. The paragraphs that follow describe each menu option.

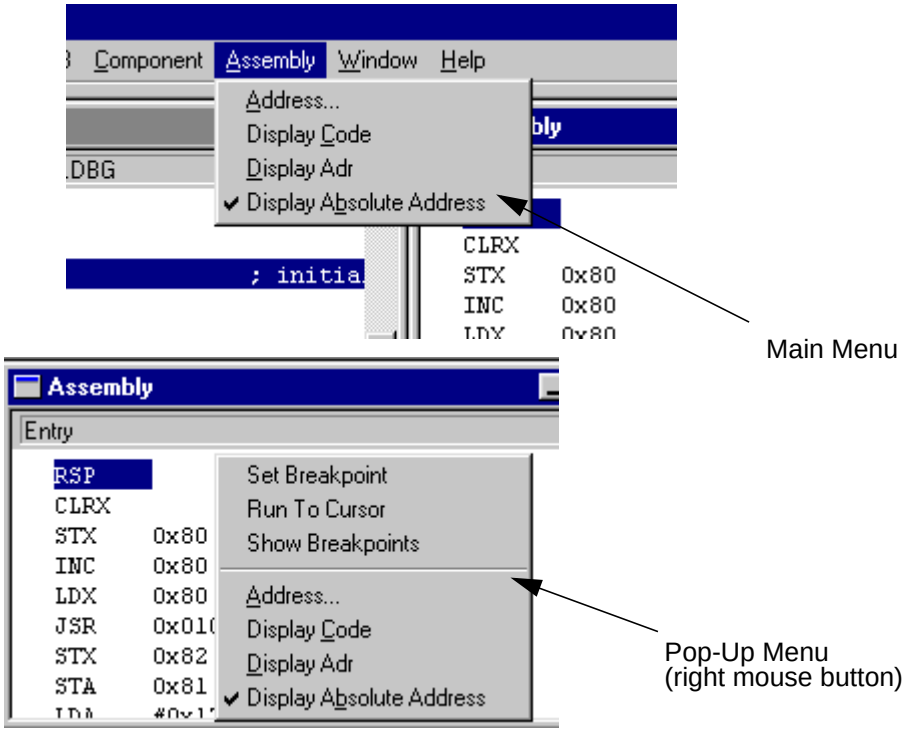


Figure 3-9. Assembly Component Main Menu and Associated Pop-up Menu

- Menu bar
  - **Address...** — Opens a dialog box prompting for an address. Enter a hexadecimal address to interpret and display the memory contents as assembler instructions starting at the specified address.
  - **Display Code** — Displays the machine code in front of each disassembled instruction
  - **Display Address** — Displays the location address at the beginning of each disassembled instruction. If both **Display Code** and **Display Address** are selected at the same time, the absolute address is displayed first, then the hexadecimal code, and finally the disassembled instruction.
  - **Display Absolute Address** — Displays the absolute address at the end of a disassembled branch instruction

- Pop-up menu
  - **Set Breakpoint** — Appears in the pop-up menu if no breakpoint is set or a breakpoint is disabled. Sets a permanent breakpoint on the instruction pointed at by the mouse cursor. When program execution reaches this instruction, the program is halted and the current program state is displayed in all component windows.
  - **Delete Breakpoint** — Appears in the pop-up menu for a pointed to breakpoint that is set or disabled. Deletes a selected breakpoint
  - **Enable Breakpoint** — Appears in the pop-up menu if a breakpoint is disabled. Enables a selected breakpoint
  - **Disable Breakpoint** — Appears in the pop-up menu if a breakpoint is set. Disables a selected breakpoint
  - **Run to Cursor** — When selected, sets a temporary breakpoint at the pointed to instruction and continues program execution. When program execution reaches this instruction, the program is halted and the current program state is displayed in all component windows. Temporary breakpoints are removed automatically as soon as they are reached.

NOTE: *If there is a disabled breakpoint at this same position, the temporary breakpoint will also be disabled and the program will not halt.*

  - **Show Breakpoints** — Opens the **Breakpoints Setting** dialog box and lists breakpoints defined in the application. Breakpoint properties can be modified.
  - **Address** — Opens a dialog box prompting for an address. Enter a hexadecimal address to interpret and display memory contents as assembler instructions starting at the specified address.
  - **Display Code** — Displays the machine code in front of each disassembled instruction
  - **Display Address** — Displays the location address at the beginning of each disassembled instruction. If both **Display Code** and **Display Address** are selected, the absolute address is displayed first, then the hexadecimal code, and finally the disassembled instruction.
  - **Display Absolute Address** — For a branch instruction, displays the absolute address at the end of the disassembled instruction

3.3.3.1 Retrieving Source Statements

Point to an instruction in the **Assembly** component window and drag and drop it into the **Source** component window. The source component scrolls to the source statement that generates this assembly instruction and highlights it.

Point to an instruction in the assembly component with the mouse cursor, click and hold the left mouse button, press the L key, then release the L key and mouse button. The code range in the source component is highlighted and the associated information in the memory component is highlighted.

3.3.4 Register Component Window

The register component window displays the content of registers and status register bits of the target processor. Register values are displayed in binary or hexadecimal format. All values can be edited. **Figure 3-10** shows the register component.

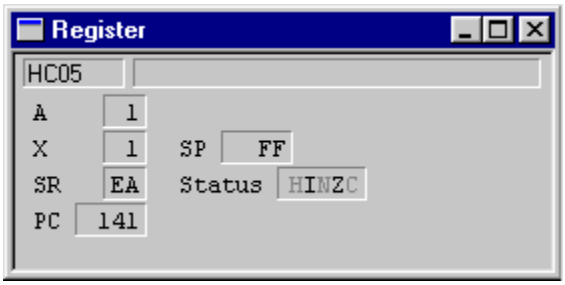


Figure 3-10. Register Component Window

3.3.4.1 Status Register Bits

All status register bits that are set, are displayed black (or dark if user has changed font color). All reset status register bits are displayed gray. A reset status register bit is toggled by placing the mouse pointer on the bit, then double clicking the left mouse button. Contents of registers that have changed since the last display refresh are shown in red (except for status register bits) during application execution.

3.3.4.2 Editing Registers

Double clicking on a register opens an edit box over the register enabling modification of the register value.

Press the Escape key (Esc) to retain the previous value of the register. Press the Enter key or select outside the edited register to validate a new value and change the register content.

If the Tab key is pressed, the register content is changed and validated, and the next register value is selected for modification.

Click and hold the left mouse button and press the A key to view the source code as well as the changed contents of the assembly and memory components. The source component shows the source code located at the address stored in the register. The assembly component shows the disassembled code starting at the address stored in the register. The memory component dumps memory starting at the address stored in the register.

3.3.4.3 Register Display Options Menu

The **Register Display Options** menu (**Figure 3-11**) provides the option to display code in either binary or hexadecimal format.

Figure 3-11. Register Display Options Menu

3.3.5 Memory Component Window

The memory component displays unstructured memory contents or memory dumps (continuous memory words without distinction between variables). Various word sizes (byte, word, double) and data formats (binary, octal, hexadecimal, decimal, unsigned decimal) can be specified for the memory display.

To specify the start address for a memory dump, use the **Address...** menu option. A memory area can be initialized with a fill pattern using the **Fill Memory** dialog box. An ASCII dump can be added/removed to the right side of the window by checking/unchecking ASCII in the **Display** menu option. The location address also can be added/removed on the left side of the window by checking/unchecking **Address** in the **Display** menu option.

NOTE: *Memory values that have changed since the previous program halt are displayed in red. If a memory item is edited or rewritten with the same value, the memory item display remains black.*

The object information bar contains the procedure or variable name, structure field, and memory range matching the first selected memory word. **Figure 3-12** shows the **Memory** component window.

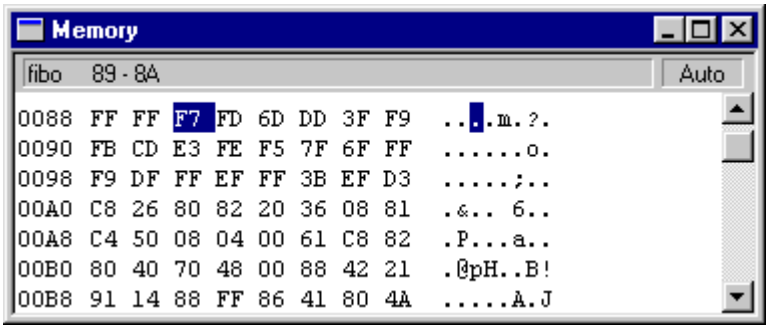


Figure 3-12. Memory Component Window

3.3.5.1 Memory Component Operations

Memory component operations are:

- Double click a memory position to edit it.
- Drag the mouse in the memory dump to select a memory range.
- Press and hold the left mouse button and press the A key to jump to a memory address. The selected value is interpreted as an address. The memory component dumps memory starting at this address.

3.3.5.2 Memory Component Pop-up Menu

The **Memory** component pop-up menu (**Figure 3-13**) is displayed by placing the cursor in the **Memory** component window and clicking the right mouse button.

Figure 3-13. Memory Component Pop-up Menu

Table 3-1 defines options in the **Memory** component pop-up menu.

Table 3-1. Memory Component Pop-up Menu

Menu Entry	Description
Word Size	Opens a submenu enabling the user to specify the display unit size. The three available sizes are byte, word (= 2 bytes), and longword (= 4 bytes).
Format	Selects the format in which the items are to be displayed. Available formats are hexadecimal, binary, octal, signed, and unsigned decimal.
Display	Opens a submenu enabling the user to toggle the display of addresses and ASCII dump
Mode	Switches between automatic, periodical, and frozen update mode. (See **3.3.5.3 Memory Update Mode**)
Address...	Opens a dialog box (**Figure 3-14**) and prompts for an address. The memory component dumps memory starting at the specified address.
Fill...	Opens the **Fill** dialog (**Figure 3-15**) to fill a memory range with a bit pattern

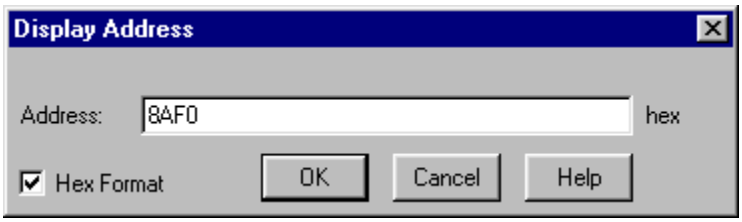


Figure 3-14. Memory Component Display Address

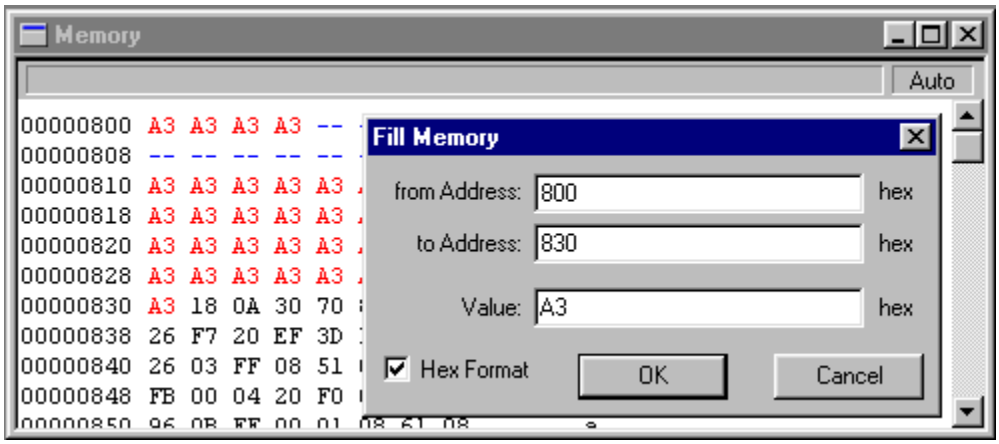


Figure 3-15. Memory Component Fill Memory Dialog Box

Click **OK** in the **Fill Memory** dialog to initialize all memory positions from $800 to $830 with the value $A3.

NOTE: *If* **Hex Format** *is checked, numbers and letters are considered to be hexadecimal numbers. Otherwise, expressions can be typed and hex numbers must be prefixed with Ox or $.*

3.3.5.3 Memory Update Mode

The memory component can be updated in three different modes:

1. In automatic mode (default), memory dump is updated when the target is stopped.
2. In frozen mode, memory dump displayed in the memory component is not updated when the target is stopped.
3. In periodical mode, memory dump is updated at regular time intervals when the application is running. The default update rate is 1 second, but it can be modified by steps up to 100 ms using the associated dialog box.

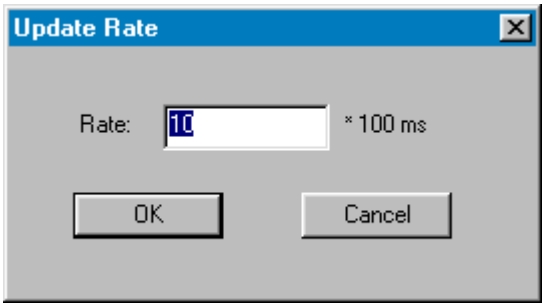


Figure 3-16. Update Rate: Memory Component

NOTE: *The periodic update mode available in the* **Data** *or* **Memory** *component menu is only available for MMDS0508 using dual-port RAM and is only possible for variables or memory positions located in the dual-port RAM area. Dual-port RAM allows the user to specify the base address and enable real-time memory. To enable dual port RAM, click on the* **MMDS0508 | Memory Map...** *menu option. To specify the base address, enter the desired value in the* **Base Address** *edit box. To enable real-time memory, check* **Enable**. *The real-time memory size is 1 Kbyte.*

3.3.6 Data Component Window

The data component contains the names, values, and types of variables. The **Data** component window shows all variables present in the current source module. Display formats, such as symbolic representation, (depending on the variable types), as well as hexadecimal, octal, binary, signed, and unsigned formats are selectable.

The object information bar contains the address and size of the selected variable. It also contains the module name where the displayed variables are defined, the display mode (automatic, frozen, etc.), and the display format (symbolic, hex, bin, etc.).

Values can be edited by double clicking on a value or the line containing a value. Arrays can be expanded by clicking on the plus (+) symbol preceding an array name. **Figure 3-17** shows the **Data** component window.

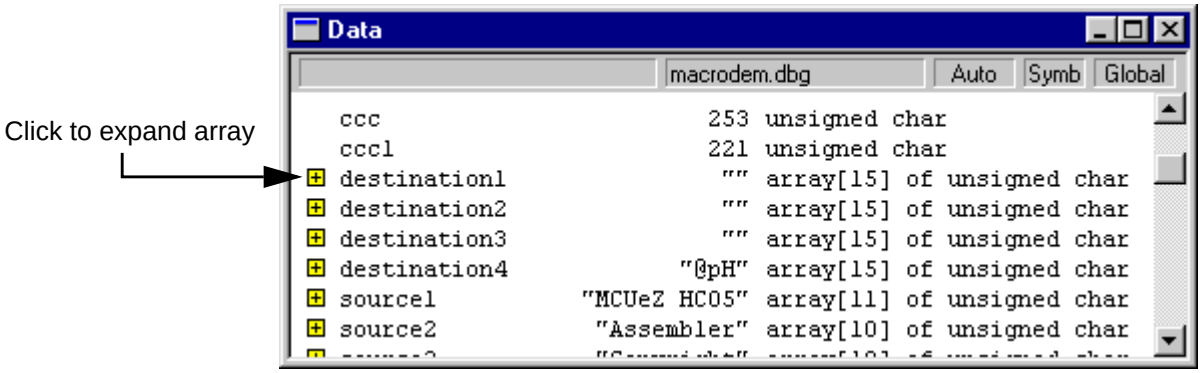


Figure 3-17. Data Component Window

3.3.6.1 Expression Editor

To add an expression, double click a blank line in the data component to open the **Expression Editor** dialog or point to a blank line, as shown in **Figure 3-18**, and right click to select **Add Expression...** in the pop-up menu.

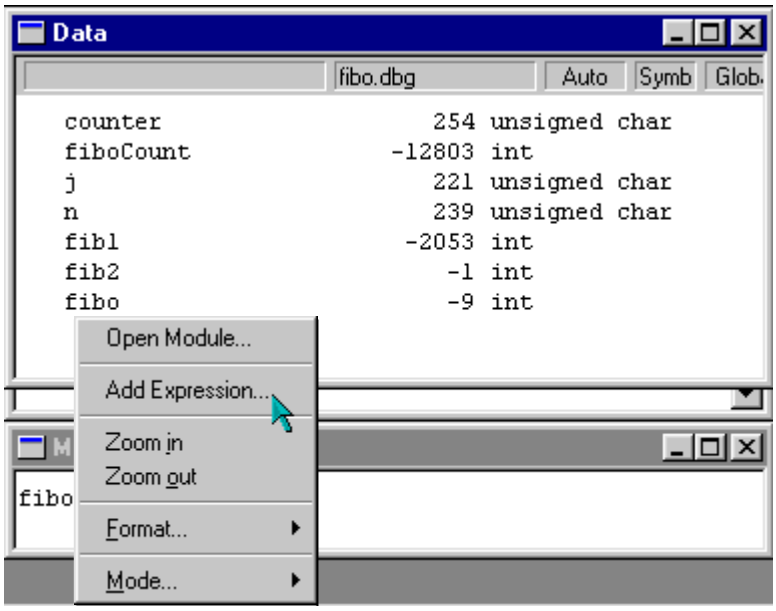


Figure 3-18. Accessing the Expression Editor

Enter a logical or numerical expression in the **Edit Expression** box using ANSI C syntax. This expression is a function of one or several variables from the current data component. **Figure 3-19** shows the expression editor.

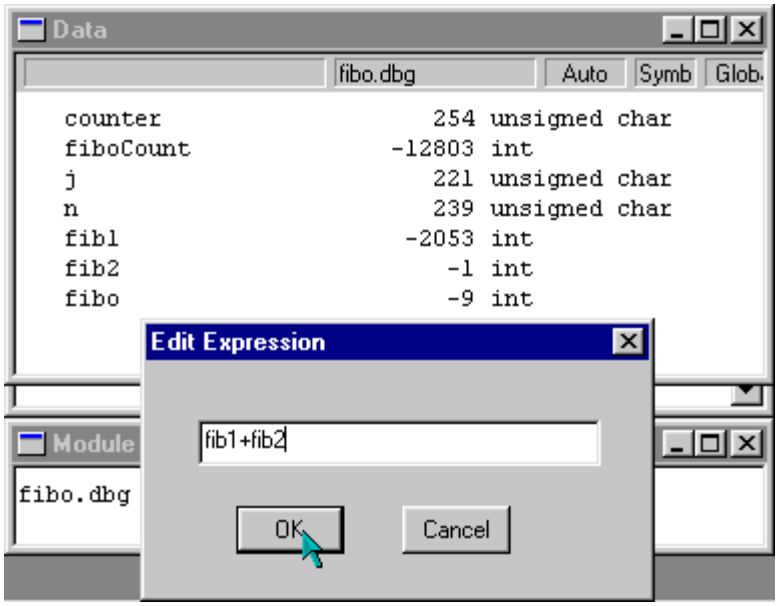


Figure 3-19. Using the Edit Expression Box

Example: With two variables `variable_1, variable_2;`

- Expression entered:
  `(variable_1<<variable_2)+ 0xFF) <= 0x1000`
  will result as a Boolean type
- Expression entered:
  `(variable_1>>~variable_2)* 0x1000`
  will result as an integer type

NOTE: *It is not possible to drag an expression defined with the expression editor.*

Operations:

- Double click a variable line to edit the value.
- Click the plus and minus signs next to each variable to fold or unfold the variable structure.
- Double clicking a blank line opens the expression editor to insert an expression in the data component.

3.3.6.2 Data Component Pop-up Menus

Figure 3-20 shows the **Data** component pop-up menus. Notice that additional options for the expression editor are displayed when pointing to a blank line or an existing expression.

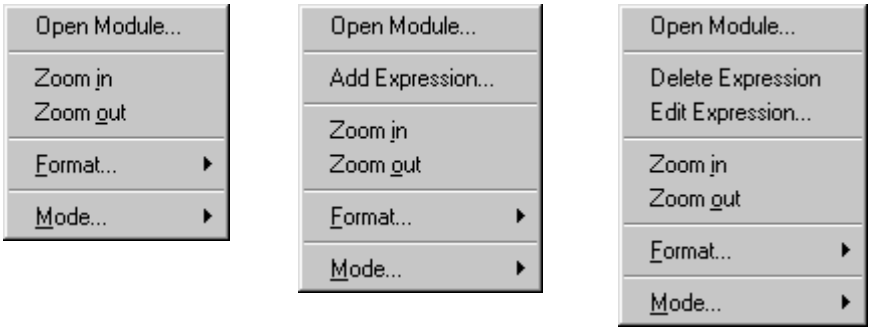


Figure 3-20. Data Component Pop-up Menus

Table 3-2 identifies all data component operations.

Table 3-2. Data Component Pop-up Menu

Menu Entry	Description
Open Module...	Opens a dialog that lists all source files bound to the application. The global variables from the selected module are displayed in the data component. This is only supported when the component is in global scope mode.
Zoom In	Expands the selected structure. For example, members of an array are displayed when selecting an array name and zooming in.
Zoom Out	Returns to the previous level
Format...	Switches between Symbolic (display format depends on the variable type), Hex (hexadecimal), Oct (octal), Bin (binary), Dec (signed decimal), UDec (unsigned decimal) display formats
Mode...	Switches between automatic, periodical, locked, and frozen update modes
Add Expression...	Appears only in the data pop-up menu when right clicking on an empty line. When selected, a user-defined expression can be added in the data component through the **Edit Expression** dialog.
Edit Expression...	Appears only in the data pop-up menu when right clicking on a line containing a user-defined expression. When selected, allows you to edit the pointed to user-defined expression through the **Edit Expression** dialog.
Delete Expression	Appears only in the pop-up menu when right clicking on a line containing a user-defined expression. When selected, it deletes the pointed to user-defined expression.

3.3.6.3 Data Update Mode

The data component can be updated in three different modes:

1. In automatic mode (default), variables are updated when the target is stopped. Variables from the currently executed module are displayed in the data component.
2. In locked and frozen mode, variables from a specific module are displayed in the data component. In locked mode, data component variable values are updated when the target is stopped. In frozen mode, variables are not updated when the target is stopped.
3. In periodical mode, variables are updated at regular time intervals when the target is running. The default update rate is 1 second. The update rate can be modified in steps of up to 100 ms.

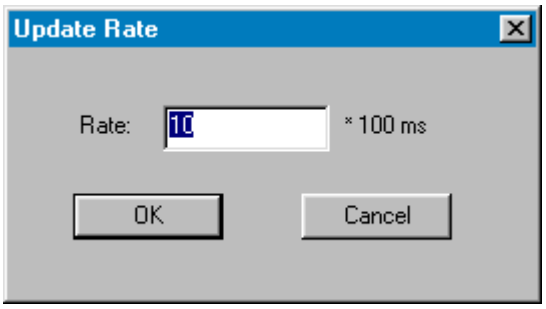


Figure 3-21. Update Rate: Data Component

3.3.7 Command Line Component Window

MCUez commands are entered on the right side of the `in>` terminal prompt in the **Command Line** component window. To recall a command, use the up arrow key, down arrow key, or special function key F3. The component executes the command entered and displays results or error messages. The 10 previous commands can be recalled using the up or down arrow keys. Commands are displayed in blue. Prompts and command responses are displayed in black. Error messages are displayed in red.

Enter **help** at the command prompt to display a list of commands.

NOTE: *Memory ranges, addresses, and values can be dropped into the command line component. Nothing can be dragged from the command line component. The command line component accesses corresponding items of the current command. Refer to* ***Section 5. Command Reference*** *for detailed information regarding MCUez commands.*

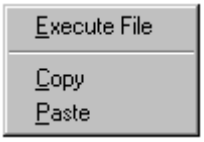


Figure 3-22. Command Line Component Menu

Table 3-3. Command Line Operations

Menu Entry	Description
Execute File	When selected, opens a dialog where the user can select a file containing MCUez commands to be executed
Copy	Appears only in the pop-up menu when something is selected in the command line component. When selected, copies the selected text to the clipboard.
Paste	Appears only in the pop-up menu when something is stored in the clipboard. When selected, the first line of text currently stored in the clipboard is appended to the current command.

Selected text from the command line component can also be copied to the clipboard using the standard window key combination CTRL + C.

The first line of text currently stored in the clipboard can be appended to the current command using the standard window key combination CTRL + V.

3.3.8 Module Component Window

The **Module** component window provides an overview of source modules used to build the application. It displays all source files (source modules) bound to the application. The module component displays all modules in the order they appear in the absolute file (*.abs*). **Figure 3-23** shows the module component.

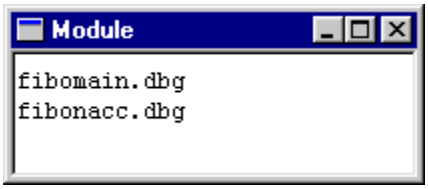


Figure 3-23. Module Component Window

Double clicking a module name forces all open windows to display information about the module. The source component shows the module's source and the data component displays the module's global variables.

NOTE: *The module component has no associated menu.*

Section 4. Operating Procedures

4.1 Contents

4.2 Introduction

This section provides procedures on:

- Configuring the debugger
- Configuring the debugger for use with various editors
- Starting the debugger from the shell
- Configuring a default layout
- Loading, starting, and stopping an application
- Defining and setting breakpoints
- Stepping through code within an application
- Working with variables
- Working on registers
- Accessing memory contents

4.3 Configuring the MCUez Debugger

The debugger must be associated with a project directory to find all requested configuration and component files.

The project (working) directory is defined in the *mcutools.ini* file located in the Windows® directory (for example, *C:\winnt*). The working directory (including path) is defined in the environment variable `DefaultDir` in the `[Options]` group or `WorkDir` in the `[WorkingDirectory]` group.

4.3.1 Configuring for Use with Editors

Perform the following steps to define an editor. Additional information about the **MCUez Shell** and **Configuration** settings are explained in the *MCUez Installation and Configuration User's Manual*, Motorola document order number MCUEZINS/D.

1. Start the **MCUez Shell**.
2. Click on the **ezMCU** icon (first icon) in the shell. The **Configuration** dialog is displayed.
3. Click **Change ...** to open the **Current Configuration** dialog.
4. Select the **Editor** tab.
5. In the **Editor** list box, select an editor.
6. For Codewright from Premia Corporation or WinEdit™, enter a filename in the **Project File** edit box. Codewright project files have the extension *.pjt* and WinEdit files have the extension *.wpj*. The editor project file is created automatically in the project directory.
7. In the **Executable** edit box, enter or browse to the editor's executable file.

4.4 Automating the MCUez Startup Process

Often the same tasks have to be performed after starting the debugger. Tasks can be automated by writing a command file that contains all commands to be executed after startup. Most target components will execute the command file *startup.cmd* once the target component is loaded and initialized. By inserting the call command in the startup command file (for example, *call init.cmd*), the user-defined command file (*init.cmd*) also will be executed when the target component is loaded.

Example of *startup.cmd* file:

```
call init.cmd
...
```

Example of *init.cmd* file:

```
load fibo.abs
bs &main t
g
```

The *init.cmd* file will load the application *fibo.abs*, then set a temporary breakpoint at the start of the function `main` and start the application. The application will then stop at `main` after executing the startup and initialization code.

4.5 Configuring the Default Layout

The default layout is activated when the debugger is started. This layout is defined in the *project.ini* file located in the project directory. **Figure 4-1** is an example of a *project.ini* file.

```
[DEFAULTS]
Window0=Source      0   0  50  40
Window1=Assembly   50   0  50  40
Window2=Register   50  40  50  30
Window3=Memory     50  70  50  30
Window4=Data        0  40  50  25
Window5=Command     0  65  50  20
Window6=Module      0  85  50  15
Target=ESL
MainFrame=149 54 752 583
[Motorola ESL]
ServerType=1
COMDEV=COM2
BAUDRATE=57600
SHOWPROT=0
```

Figure 4-1. Example of project.ini File

Elements of the *project.ini* file:

- `Target:` Specifies the default target component used when starting the debugger, for example `Target=ESL`
- `Layout:` Specifies a default layout file for component windows, for example `Layout=mylayout.hwl`. Layout files are created by selecting the **Window | Layout | Store** menu option in the debugger.

NOTE: *The complete path to the layout file must be specified if the file is not stored in the project directory.*

NOTE: *Other elements of the project.ini file are defined automatically and written to the file after performing operations such as using the mouse to resize windows or entering information in dialog boxes.*

4.6 Loading an Application

Follow these steps to load an application:

1. Select **Load** in the target menu (for example, MMDS0508) to open the **Load Executable File** dialog box (**Figure 4-2**). The target menu is located between the **Run** and **Component** menus.
2. Select an application (for example, *fibo.abs*).
3. Click **OK** to close the dialog box and load the application into the debugger.

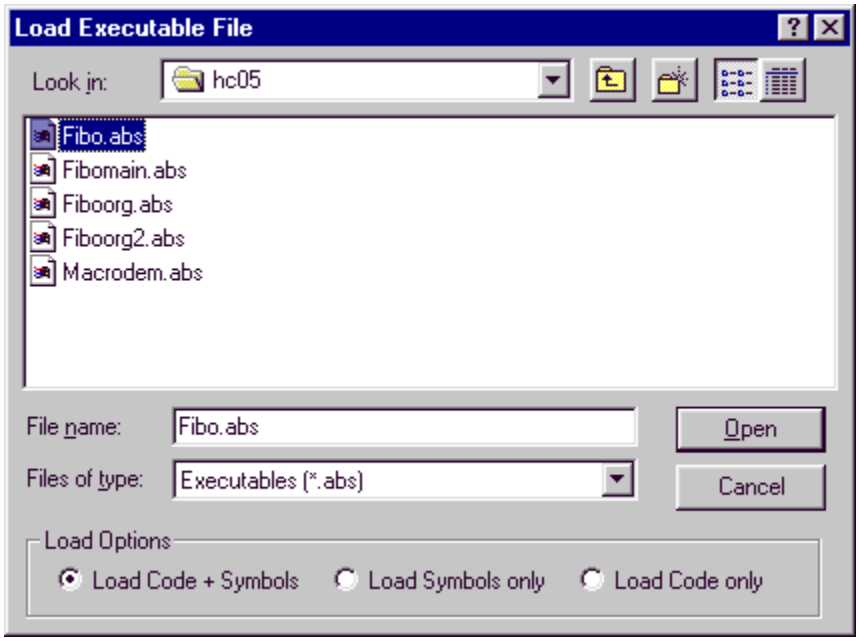


Figure 4-2. Load Executable File Dialog Box

Figure 4-3 illustrates an executable file loaded in the debugger. The source component displays the source from the module containing the entry point for the application. The highlighted statement is the entry point.

The assembly component contains the corresponding disassembled code. The highlighted statement is the entry point from the application. The code is disassembled directly from the target board memory.

The global data component contains the list of global variables defined in the module containing the application entry point.

The PC (program counter) in the register component is initialized with the PC value from the application entry point.

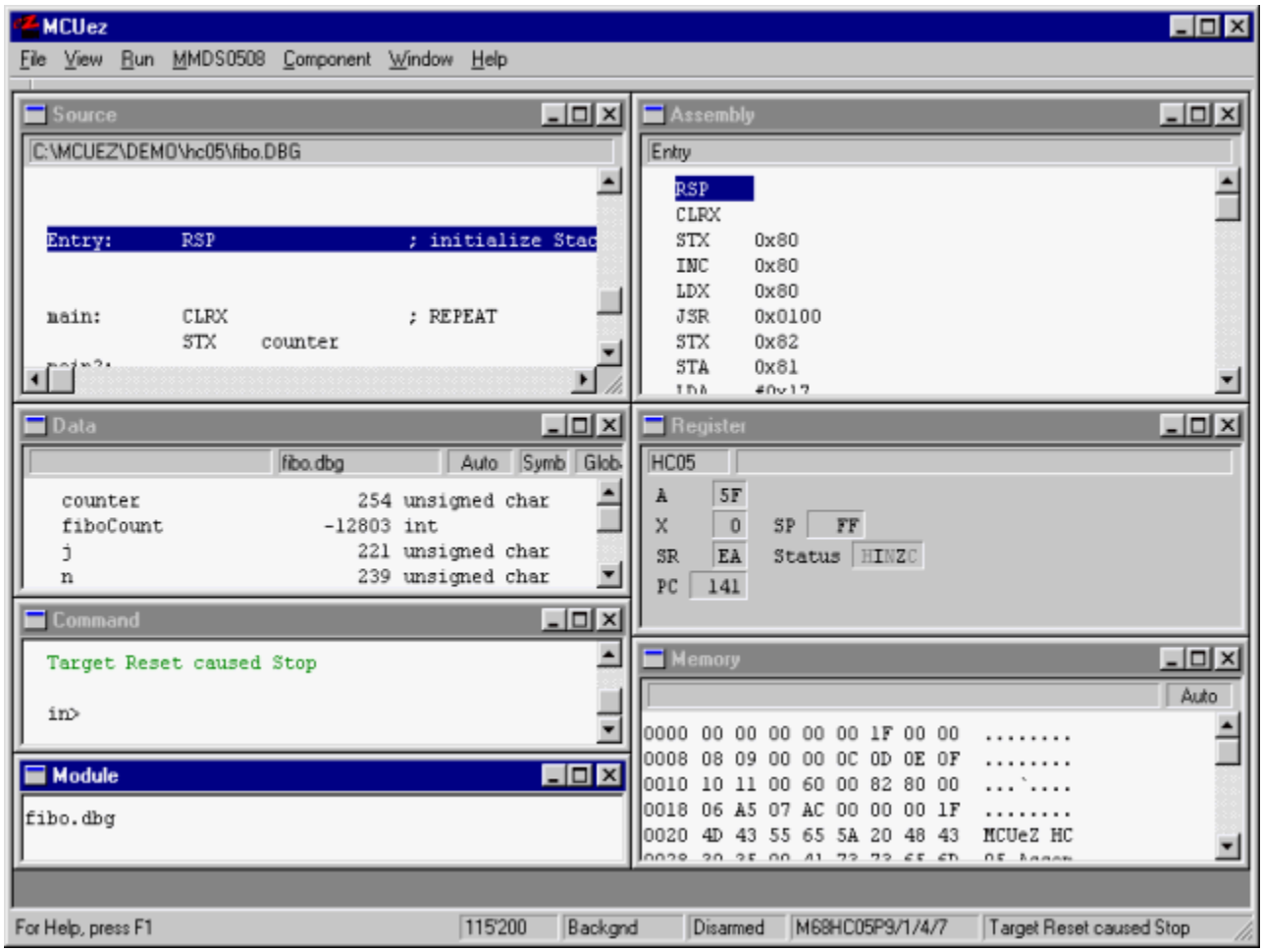


Figure 4-3. Loading an Application

4.7 Starting an Application

The two ways to start an application are:

1. Select the **Run| Start/Continue** menu option.
2. Click the **Start/Continue** button in the MCUez toolbar  .

The message `RUNNING` in the status bar indicates that the application is running.

The application will continue until:

1. It is manually halted.
2. A breakpoint has been reached.
3. An exception has been detected.

4.8 Stopping an Application

The two ways to stop an application are:

1. Choose **Run | Halt**.
2. Click on the **Halt** button in the MCUez toolbar  .
   `HALTED` is displayed in the status bar to indicate that program execution has stopped.

The highlighted line in the source component is the source statement at which the program was stopped (for example, the next statement that will be executed).

The highlighted line in the assembly component is the assembler statement at which the program was stopped (for example, the next assembler instruction that will be executed).

The data component displays the names and values of global variables defined in the currently executed module. The name of the module is specified in the data component information bar.

4.9 Breakpoints

Breakpoints are control points associated with a program counter (PC) value (for instance, program execution is stopped as soon as the PC reaches the value defined in a breakpoint). The MCUez debugger supports different types of breakpoints:

- Run-to-cursor breakpoints (temporary breakpoints), which are deleted as soon as they are reached. This type of breakpoint is activated the next time the corresponding instruction is executed.
- Set breakpoints (permanent breakpoints), which remain active until the user deletes them. This type of breakpoint will be activated each time the corresponding instruction is executed.

Breakpoints can be set in either a source or assembly component. **Figure 4-4** shows the **Breakpoints Setting** dialog window.

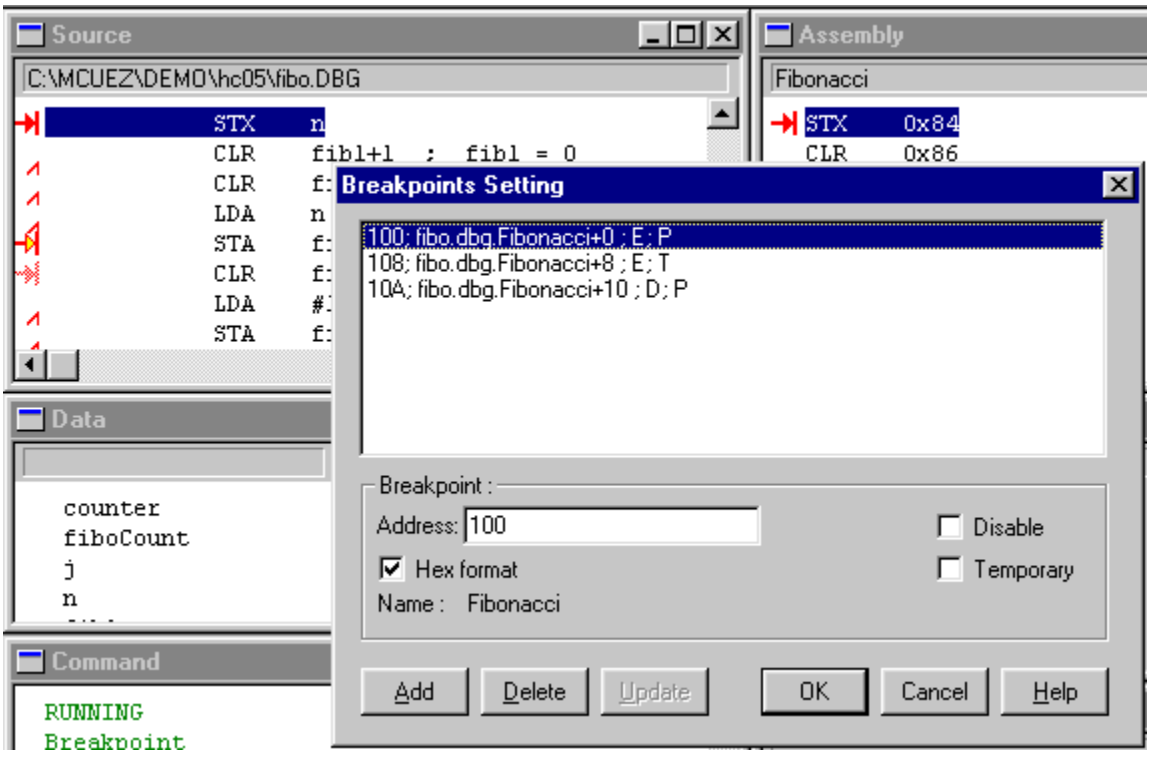


Figure 4-4. Breakpoints Setting Dialog Window

The **Breakpoint Setting** dialog consists of:

- A list box which displays a list of currently defined breakpoints
- **Breakpoint:** group box which displays the address of the currently selected breakpoint, name of executable file, name of procedure in which the breakpoint has been set, the state of the breakpoint (disable or not) and type of breakpoint (temporary or permanent).
- **Add** button: Adds a new breakpoint at the address specified in the Address edit box
- **Delete** button: Click **Delete** to remove the currently selected breakpoint.
- **Update** button: Updates changes made to an existing breakpoint
- **OK** button: Click **OK** to accept all modifications.
- **Cancel** button: Click **Cancel** to retain all previous settings.
- **Help** button: Click **Help** to open the help file and associated information.

The list box allows multiple consecutive breakpoints to be selected. Select a breakpoint, then hold the Shift key, and select another breakpoint.

Select multiple non-consecutive breakpoints by selecting a breakpoint, then holding the control (Ctrl) key, and selecting other breakpoints.

When selecting multiple breakpoints, the name of the group box **Breakpoint:** changes to **Selected breakpoints:** and the associated controls **Address** (hex) and **Name:** are disabled.

4.9.1 Breakpoint Symbols

A temporary breakpoint has this symbol :  .

A permanent breakpoint has this symbol:  .

A disabled breakpoint has this symbol:  .

4.9.2 Identifying All Positions to Define a Breakpoint

Some compound statements (a statement that can be split into several base instructions) can be generated when using a high level language. The debugger helps detect all positions where a breakpoint can be set when these steps are followed:

1. Right click in the source component to display the source pop-up menu.
2. Select **Marks** from the pop-up menu. All statements where a breakpoint can be set are identified by a special mark: . See **Figure 4-5**.
3. To remove the breakpoint marks, right click in the source component and uncheck **Marks**.

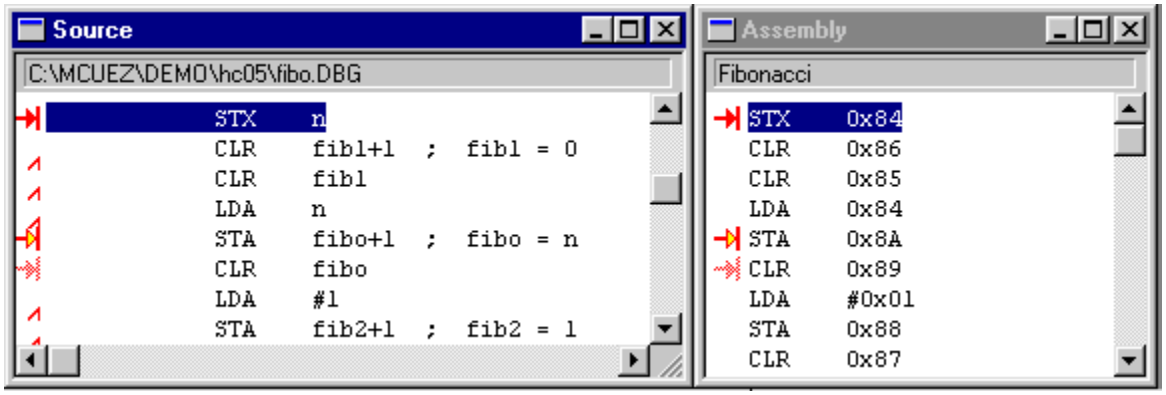


Figure 4-5. Identifying Breakpoint Positions

4.9.3 Defining a Breakpoint

The debugger provides two ways to define a breakpoint:

1. Using the pop-up menu:
   - Point at a statement in the **Source** or **Assembly** component window and click the right mouse button to display the pop-up menu.
   - Select **Set BreakPoint** or **Run to Cursor** from the pop-up menu. A breakpoint mark is displayed in front of the selected statement.
2. Using the keyboard:
   - Point at a statement in the **Source** or **Assembly** component window, hold the left mouse button, and press the P key for a permanent breakpoint or T for a temporary breakpoint.
   - A breakpoint mark is displayed in front of the selected statement.

Once a breakpoint has been defined, program execution can continue. The application stops before executing the statement. Permanent breakpoints remain active until they are disabled or deleted.

4.9.4 Deleting a Breakpoint

The debugger provides four ways to delete a breakpoint:

1. Using **Delete Breakpoint** from pop-up menu:
   - In the source or assembly component, point at a statement where a breakpoint has been defined and right click.
   - Select **Delete Breakpoint** from the pop-up menu.
2. Using the keyboard:
   - In the source or assembly component, point at a statement where a breakpoint has been defined, hold down the left mouse button, and press the D key.

3. Select **Show Breakpoints...** from **Source** or **Assembly** pop-up menu:
   - Place the mouse pointer in the source or assembly component window and right click.
   - Select **Show Breakpoints** from the pop-up menu. The **Breakpoints Setting** dialog is displayed.
   - In the list of defined breakpoints, select the breakpoint to delete.
   - Click **Delete**. The selected breakpoint is removed from the list of defined breakpoints.
   - Click **OK** to close the **Breakpoints Setting** dialog box.
4. Select **Run | Breakpoints ...**:
   - Choose **Run | Breakpoints ...** The **Breakpoints Setting** dialog is displayed.
   - Select the breakpoint to delete.
   - Click **Delete**.
   - Click **OK** to close the **Breakpoints Setting** dialog box.

4.9.5 Breakpoints Menu

Figure 4-6 shows the breakpoint options in the pop-up menu.

Figure 4-6. Breakpoints

Table 4-1 defines all breakpoint menu options in the pop-up menu.

Table 4-1. Breakpoint Pop-Up Menu Definitions

Menu Entry	Description
Set Breakpoint	Selects the permanent breakpoint option
Run To Cursor	When selected, sets a temporary breakpoint at the nearest code position and continues execution of the program immediately. If a disabled breakpoint is at this position, the temporary breakpoint will also be disabled and the program will not halt. Temporary breakpoints are removed automatically when they are reached.
Show Breakpoints	Opens the **Breakpoints Setting** dialog box and allows the user to consult the list of breakpoints defined in the application and to modify their properties
Marks	Toggles the display of source positions where breakpoints may be set. If this switch is on, source positions are marked by upside down check marks.

NOTE: *If some statements do not show marks although the mark display is switched on, the following may be at fault:*

- *The statement did not produce code due to compiler optimization.*
- *The entire procedure or section is not linked in the application because of smart linking.*

4.10 Stepping in the Application

The debugger provides stepping functions at the assembler level. Implementing stepping functions is described in the following subsections.

4.10.1 Stepping on Assembly Level

The debugger provides two ways of stepping to the next assembler instruction:

1. Select **Run | Assembly Step**.
2. Click the **Assembly Step** button on the toolbar:  .

The message, TRACED, in the status line indicates that the application is stopped by an assembly step function. The application stops at the next assembler instruction.

The display in the source component is always synchronized with the display in the assembly component. The highlighted instruction in the source component is the source instruction that has generated the highlighted instruction in the assembly component.

Elements displayed in red in the register, memory, or data components are values that have changed during execution of the assembly instruction.

4.10.2 Stepping Over a Function Call (Flat Step)

The debugger provides two ways of stepping over a function call:

1. Select **Run | Step Over**.
2. Click the **Step Over** button on the tool bar  .

The message, STEPPED OVER, in the status line indicates that the application is stopped by a step over function. If the application was stopped previously on a function invocation (a JSR (jump to subroutine) or BSR (branch to subroutine) instruction), a **Step Over** stops the application on the source instruction following the function invocation.

Elements displayed in red in the register, memory, or data components are values that have changed when the **Step Over** function was invoked.

4.11 Working with Variables

The following sections describe how to work with variables in the debugger.

4.11.1 Displaying Global Variables from a Module

The debugger provides two ways to view a list of global variables defined in a module:

1. Using drag and drop:
   - Drag a module name from the module component to a data component.
2. Using pop-up menu:
   - Place the mouse pointer in the data component and right click.
   - Select **Open Module** in the pop-up menu. A dialog box that contains the list of all modules used to build the application opens.
   - Double click on a module name.

The data component displays the list of global variables and their values that are defined in the selected module.

4.11.2 Changing the Variable Value Display Format

The debugger can display variable values in different formats. The **Format** entry in the pop-up menu provides several options. The selected format affects all data component variables. See **Table 4-2**.

Table 4-2. Changing the Variable Value Display Format

Menu Entry	Description
Hex	Variable values are displayed in hexadecimal format.
Oct	Values are displayed in octal format.
Dec	Values are displayed in signed decimal format.
UDec	Values are displayed in unsigned decimal format.
Bin	Values are displayed in binary format.
Symbolic	Displayed format depends on the variable type. Values for character variables are displayed in ASCII character and decimal format. Values for other variables are displayed in signed or unsigned decimal format depending on the variable being signed or not.

4.11.3 Modifying a Variable Value

The debugger allows variable values to be changed. Double click on a variable. The current value is highlighted and can be edited (**Figure 4-7**).

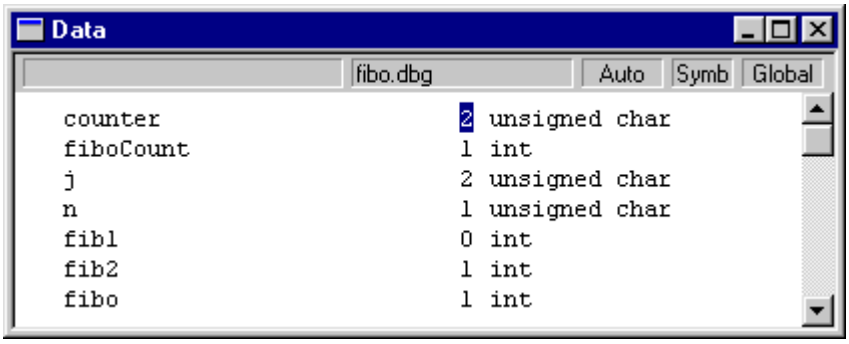


Figure 4-7. Modifying a Variable Value

The following procedure describes how to modify a variable value:

1. Formats for the input value follow the rule for ANSI C constant values. Values are prefixed by `0x` for a hexadecimal value or `0` for octal values. All other values are treated as decimal values. For example, if the data component is in decimal format and the variable input value is 0x20, then the variable is initialized with 32. If a variable input value is 020, the variable is initialized with 16.
2. To restore the previous value, press the escape (Esc) key or select another variable.
3. To validate the new value, press either the Enter key or Tab key.
4. If an input value has been validated by pressing the Tab key, the next variable value in the component is highlighted automatically.

4.11.4 Displaying an Allocated Variable Address

The debugger displays the start address and variable size in the data information bar when a variable name is clicked.

4.11.5 Loading an Address Register with a Variable Address

To load a register with the address of a variable, drag a variable name from the data component to the register component. The destination register is updated with the start address of the selected variable.

4.12 Working with Registers

The following sections describe how to work with registers.

4.12.1 Changing the Register Display Format

The debugger allows the register content to be displayed in hexadecimal or binary format. To do so:

1. Right click in the register component to display the pop-up menu.
2. Choose **Options ...**
3. Select either binary or hexadecimal format.

4.12.2 Modifying the Content of an Index or Accumulator Register

The following procedure describes how to modify the content of an index or accumulator register. The register window changes to reflect the MCU used in the system.

1. Double click on a register. The current register content is highlighted (**Figure 4-8**) and can be edited.

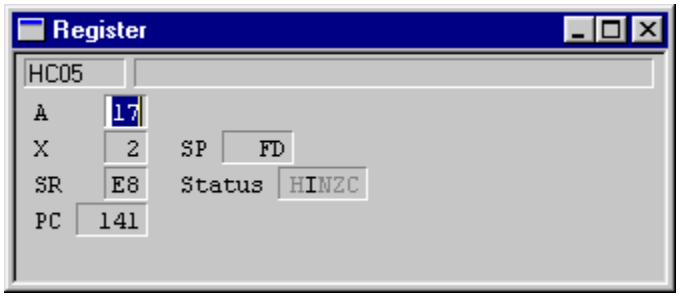


Figure 4-8. Modifying the Content of an Index or Accumulator Register

2. The format of the input value depends on the format selected for the register component. If the format is hex, the input value is treated as a hex value. If the input value is 10, the variable will be set to `0x10 = 16`.
3. To restore the previous register content, press the Esc key.
4. To validate the input value, press either the Enter key or Tab key or select another register.
5. If an input value has been validated by pressing the Tab key, the content of the next register is highlighted automatically.

4.12.3 Modifying Bit Register Contents

In a bit register, each bit has a specific meaning, for example, a status register (SR) or condition code register (CCR) bit from a processor. Mnemonic characters for bits set to 1 (one) are displayed in black. Mnemonic characters for bits that are reset to 0 (zero) are displayed in grey. Single bits inside the bit register are toggled by double clicking on the corresponding mnemonic character.

4.12.4 Retrieving a Memory Dump Starting at a Register-Indicated Address

The MCUez debugger provides two ways to dump the memory starting at the address a register is pointing to:

1. Using drag and drop:
   - Drag a register from the register component to the memory component.
2. Select **Address...** menu entry:
   - Right click in the memory component to display the pop-up menu.
   - Select **Address ...** to open the **Memory** dialog box.
   - Enter the register content in the **Address:** field and select **OK** to close the dialog box.

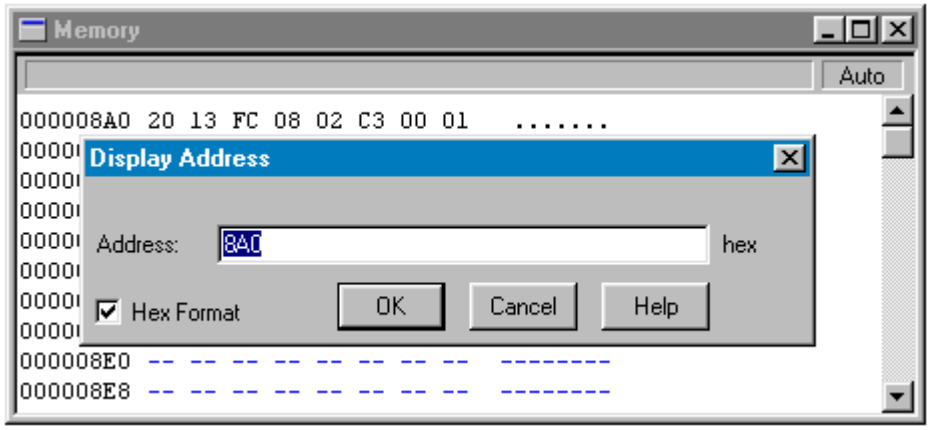


Figure 4-9. Choosing a Memory Address

The memory component scrolls until it reaches the address specified in the dialog box. This feature allows the user to display a memory dump from the application stack.

NOTE: *If **Hex Format** is checked, numbers and letters are treated as hexadecimal numbers. Otherwise, expressions can be typed and hex numbers should be prefixed with `0x` or `$`.*

4.13 Working with Memory

These sections describe how to work with and modify memory content.

4.13.1 Changing the Memory Display Format

The debugger allows the user to define the format to use in the memory component. The **Format** menu entry in the **Memory** component pop-up menu provides several options. Table 4-3 defines the format options.

Table 4-3. Memory Display Format Options

Menu Entry	Description
Hex	Memory dump is displayed in hexadecimal format.
Dec	Memory dump is displayed in signed decimal format.
UDec	Memory dump is displayed in unsigned decimal format.
Oct	Memory dump is displayed in octal format.
Bin	Memory dump is displayed in binary format.

4.13.2 Modifying Memory Address Content

The debugger allows the content of a memory address to be changed. To do so:

1. Double click the memory location to be modified. The content of the memory location is highlighted and can be edited.
2. The format for the input value depends on the format selected for the memory component. If the format is hex, the input value is treated as a hex value. For example, if the input value is 10, the memory location will be set to 0x10 = 16.
3. To restore the previous memory value, press Esc.
4. To validate the new value, press Enter or Tab, or select another memory position.
5. Once a value has been allocated to a memory word, it is validated and the next memory address is selected automatically.

Section 5. Command Reference

5.1 Contents

5.2 Introduction

This section provides a detailed list of all MCUez debugger commands. All commands and component names are case insensitive. The EBNF (Extended Backus-Naur Form) command syntax is:

```
[<component name> [:<component number>] < ] <command>
```

where `component name` is the name of the component window.

The `component number` is the number of the component. This number does not exist in the component window title if only one component of this type is open. When two instances of the data component are open, each data component is titled numerically as in `Data:1` and `Data:2`. A number is automatically associated with a component if more than one instance of the component exists.

The redirect left symbol (<) redirects a command to a specific component. Some commands are valid for several or all components. If the command is not redirected to a specific component, all concerned components will be affected. Also, a mismatch could occur due to the fact that command parameters could differ from one component to another for the same command name.

5.3 List of Available Commands

This section lists and defines each available MCUez debugger command.

5.3.1 Kernel Commands

Kernel commands are used to build command files. Therefore, they can be used only in an MCUez command file. The command line component accepts one command at a time. It is possible to build powerful programs, combining kernel commands with target commands and component commands.

Command Syntax	Short Description
`AT time`	Sets a time condition for a command execution
`ELSE`	Alternate operation associated with `IF` command
`ELSEIF condition`	Alternate conditional operation associated with `IF`
`ENDFOCUS`	Resets the current focus (see `FOCUS` command)
`ENDFOR`	Exits a `FOR` loop
`ENDIF`	Exits an `IF` condition
`ENDWHILE`	Exits a `WHILE` loop
`FOCUS componentName`	Sets the focus on a specified component
`FOR[variable =]range [“,” step]`	`FOR` loop instruction
`GOTO label`	Unconditional branch to a label in a command file
`GOTOIF condition Label`	Conditional branch to a label in a command file
`IF condition`	Conditional execution
`REPEAT`	`REPEAT` loop instruction
`RETURN`	Returns from a `CALL` command
`UNTIL condition`	Condition of a `REPEAT` loop
`WAIT [time] [;s]`	Command file execution pause
`WHILE condition`	`WHILE` loop instruction

5.3.2 Target Commands

Target commands are used to monitor the MCUez target execution. Target input/output files, target execution control, direct memory editing, breakpoint management, and CPU register setup are handled by these commands. Target commands are executed independent of open components.

Command Syntax	Short Description
`BC <address | * >`	Deletes a breakpoint
BD	Displays a list of all breakpoint
`BS <address> [P | T]`	Sets a breakpoint
`CALL [filename] [;C] [;NL]`	Executes a command file
`CD [path]`	Changes the current working directory
`CF [filename] [;C] [;NL]`	Executes a command file
`CR [filename][;A]`	Opens a record file
`DASM [address|range][;OBJ]`	Disassembles
`DB [address|range]`	Displays memory bytes
`DEFINE symbol [=]`	Defines a user symbol expression
`DL [address|range]`	Displays memory bytes as longwords
`DW [address | range]`	Displays memory bytes as words
`G [address]`	Starts execution of the loaded application
`LF [filename][;A]`	Opens a log file
`LOG type [=] state {[,] type [=] state}`	Sets options for the log file
`LS [symbol | *][;(C | S)]`	Displays the list of symbols
`MEM`	Displays the memory map
`MS range list`	Sets memory bytes
`NB [base]`	Sets the base for arithmetic operations
`NOCR`	Closes the record file
`NOLF`	Closes the log file
`P [address]`	Single assembly steps into the program
`RD [list | *]`	Displays content of registers
`RS register[=]value {[,]register[=]value}`	Sets a register
`S`	Stops application execution

Command Syntax	Short Description
SAVE range filename [offset][;A]	Saves a memory block in S-record format
SREC filename [offset]	Loads a memory block in S-record format
T [address][,count]	Traces instructions at specified address
UNDEF symbol | *	Undefines a user symbol
WB range list	Writes bytes
WL range list	Writes longwords
WW range list	Writes words

5.3.3 Component Commands

Component commands monitor the debugger environment, component windows, component window layouts, and load component windows and user applications.

Command Syntax	Short Description
ACTIVATE <component name>	Activates a component window
ATTRIBUTES list	Sets up display inside component
AUTOSIZE on|off	Autosizes windows in main window
BCKCOLOR color	Sets the background color
CLOSE <component name> | *	Closes a component
E <expression> [;(O|D|X|C|B)]	Evaluates a given expression
FILL <range> <value>	Fills a memory range with a value
FIND <string> [;B] [;MC] [;WW]	Finds and highlights a pattern
FONT 'fontName' [size][color]	Sets text font
HELP	Displays a list of available commands
LOAD applicationName	Loads user's application
OPEN <component name> [x y width height][;i|;max]	Opens a component window
SLAY <filename>	Saves the general window layout
SMEM range	Shows a memory range
SMOD module	Shows a module
SPC address	Shows specified address in window
SPROC level	Shows information associated with procedure

Command Syntax	Short Description
`UPDATERATE rate`	Sets the data update mode
`VER`	Displays version number of components and MCUez
`ZOOM <address in | out>`	Zooms in/out on an array

5.4 Definition of Terms

A definition and explanation of how certain words are used in command syntax descriptions follow.

address — A number matching a memory address. This number can be specified in the ANSI C format (for instance, `0x` for hexadecimal value, O for octal) or in the MCUez assembler format ($ for hexadecimal, @ for octal, % for binary).

Example: `255, 0377, 0xFF, $FF`

NOTE: **address** *can also be an "expression" if "constant address" is not specifically mentioned in the command description. An "expression" can be: Global application variables, I/O register definitions defined in* `DEFAULT.REG`*, definitions in the command line, and numerical constants.*

Example:
```
DEFINE IO_PORT = 0x210
  WB IO_PORT 0xFF
```

range — A composition of two addresses that define a memory address range. The syntax is shown as:

```
address..address
```
or
```
address, size
```

where `size` is an ANSI format numerical constant.

Example:
```
0x2F00..0x2FFF
```

Refers to a memory range starting at `0x2F00` and ending at `0x2FFF` (256 bytes)

Example:
```
0x2F00,256
```

Refers to a memory range starting at `0x2F00`, which is 256 bytes wide. Both previous examples are equivalent.

filename — A DOS filename and path that identifies a file and its location. The command interpreter does not assume the filename extension. Use backslash (\) or slash (/) as a directory delimiter. The parser is case insensitive. If no path is specified, it looks for the file or writes the file into the current project directory. For example, when no path is specified, the default directory is the project directory.

component — Name of a component window

Example:

```
Memory
```

5.5 Expressions

Many commands accept expressions as parameters. Expression syntax and semantics are described here.

5.5.1 Expression Definition in EBNF

Example:

```
expression = lorExpr
lorExpr    = landExpr {"||" landExpr} // logical OR
landExpr   = orExpr {"&&" orExpr}     // logical AND
orExpr     = xorExpr {"|" xorExpr}    // bitwise OR
xorExpr    = andExpr {"^" andExpr}    // bitwise XOR
andExpr    = eqExpr {"&" eqExpr}      // bitwise AND
eqExpr     = relExpr {("==" | "!=") relExpr}
relExpr    = shiftExpr {("<" | ">" | "<=" | ">=") shiftExpr}
shiftExpr  = addExpr {("<<" | ">>") addExpr}
addExpr    = mulExpr {("+" | "-") mulExpr}
MulExpr    = castExpr {("*" | "/" | "%") castExpr
castExpr   = ["~" | "!" | "+" | "-" ] parenExpr
parenExpr  = "(" expression ")"
               | cObject
               | symbol
               | register
               | variable
               | string
               | number
symbol   defined with the DEFINE command
register   = IOReg
variable   = ObjectReg
ObjectReg  = ["OBJPOOL::"] ObjectSpec
ObjectSpec = ObjectName ["." FieldName].
ObjectName = ident [":" Index].
FieldName  = IdentNum ( [".." IdentNum] | ["." Size] ).
IdentNum   = ident | "#" HexNumber.
Size       = "B" | "W" | "L".
ident      is an identifier as defined in ANSI C

IOReg      = ["IOREG::"] group | regName
group      Refer to the Motorola I/O register file definition
           in Appendix A. Register Description File.
regName    Refer to the Motorola register name definition
           in Appendix A. Register Description File.

itemName   = module |[[module] ":"] procedure |
           [[module] ":" [procedure] ":"] variable
variable   = ident { "." ident | number }
module     = ident ["." extension]
procedure  = ident
extension is an identifier as defined in ANSI-C
number is a number as defined in ANSI-C
ident is an identifier as defined in ANSI-C
```

Module names can have an extension. If no extension is specified, the parser will look for the first module that has the same name (without extension).

5.5.2 Semantics

A scope represents either a module or procedure. A scope is recognized by the presence of the double colon which terminates the scope. If the scope identification contains at least one colon, it is assumed to represent a procedure; otherwise, it represents a module.

Empty module or procedure names represent the current module or procedure, respectively. The current procedure is the procedure that the program counter of the debugger points to. The current module is the module that contains the current procedure.

Items are identified as absolute or relative, corresponding to the presence or absence of a scope.

An item is identified as absolute by specifying its scope, for instance, the module and/or procedure where the item is located.

An item is identified as relative if a scope is omitted. In this case, the item is assumed to be located in the current procedure.

5.5.3 Scope Examples

`fibo.dbg:Fibonacci:fib1`	Matches the local variable `fib1` of the procedure `Fibonacci` in the module `fibo.dbg`
`:main`	Matches the procedure `main` in the current module
`start12.c:_Startup`	Matches the procedure `_Startup` in the module `start12.c`
`::counter`	Matches the global variable `counter` of the current module
`:Fibonacci:fib1`	Matches the local variable `fib1` of the procedure `Fibonacci` of the current module
`fibo.dbg::counter`	Matches the global variable `counter` of the module `fibo`

`fib1`	Matches the local variable of the current procedure or a global variable of any module
`startupData.flags`	Matches the field flags of the local or global variable `startupData` (which is a structure) of the current module or procedure

5.5.4 Constant Standard Notation

Inside an expression, the ANSI C standard notation for constant is supported. This means that independent of the current number base, hexadecimal or octal constants can be specified using standard ANSI C notation.

Example:	**Notation**	**Meaning**
	`0x----`	Hexadecimal constant
	`0----`	Octal constant

Similarly, the assembler notation for constant is supported. This means that independent of the current number base, hexadecimal, octal, or binary constants can be specified using the assembler prefixes.

Example:	**Notation**	**Meaning**
	`$----`	Hexadecimal constant
	`@`	Octal constant
	`%`	Binary constant

When the default number base is 16, constants starting with a letter `A`, `B`, `C`, `D`, `E`, or `F` must be prefixed by `0x` or `$`. Otherwise, the command line detects a symbol and not a number.

Example:	**Notation**	**Meaning**
	`5AFD`	Hexadecimal constant `$5AFD`
	`AFD`	Symbol name
	`$AFD`	Hexadecimal constant

5.6 Kernel Commands

Kernel commands are commands that build command programs. They are used as a command line component in an MCUez debugger command file. Each command line component accepts one command at a time. The user can build programs and combine kernel, base, common, and component-specific commands.

AT

Short description:

Time delay for executing a command in a command file

Syntax:

```
At <time>
```

Argument:

`time` Expression interpreted in milliseconds

Description:

The `AT` command temporarily suspends a command from executing for a specified delay in milliseconds. The delay is measured from the time the command file is started. In the event that command files are chained (one calling another), the delay is measured from the time the first command file is started. This command can be executed only from a command file. The time specified is relative to the start of the command file.

Example:

```
AT 10 OPEN Command
```

This command opens a command line component 10 ms after execution of the command file.

CALL

Short description:

Executes a command file

Syntax:

```
CALL [FileName] [;C][;NL]
```

Description:

The `CALL` command is an alias of the `CF` command. Refer to the **`CF`** command in **5.3.2 Target Commands** for a description and examples.

DEFINE

Short description:

Defines a user symbol

Syntax:

`DEFINE symbol [=]` *`expression`*

Arguments:

`symbol`	User-defined name
`expression`	User-defined expression assigned to symbol name

Description:

The `DEFINE` command creates a symbol and associates the value of an expression with the symbol. Arithmetic expressions are evaluated when the command is interpreted. The symbol represents the expression until the symbol is redefined or undefined using the `UNDEF` command. A symbol is a maximum of 31 text characters. In a command line, all symbol occurrences (after the command name) are substituted by their values before processing. A symbol cannot represent a command name. A symbol definition precedes (and therefore conceals) a program variable with the same name. Defined symbols remain valid when a new application is loaded.

Use this command to assign meaningful names to expressions that are used in other commands. This increases the readability of command files and avoids re-evaluation of complex expressions. An application variable or I/O register can be overwritten with a `DEFINE` command.

Example:

```
DEFINE addr $1000
DEFINE limit = addr + 15
```

First `addr` is defined as a constant equivalent to `$1000`. Then `limit` is defined and affected with the value (`$1000 + 15`). Define a symbol (on the command line) using the `DEFINE` command. The symbol defined in the application is not accessible until an `UNDEF` is issued on the symbol name.

Example:

```
/* Define symbol testCase */
DEFINE testCase = $800
/* Display value stored at address $800 */
DB testCase
/* Undefine symbol testCase */
UNDEF testCase
```

ELSE

Short description:

Alternate operation associated with `IF` command

Syntax:

`ELSE`

Description:

The `ELSE` keyword is associated with the `IF` command.

ELSEIF

Short description:

Alternate conditonal operation associated with `IF` command

Syntax:

`ELSEIF condition`

Argument:

`condition` Defined as in C language syntax

Description:

The `ELSEIF` keyword is associated with the `IF` command.

ENDFOCUS

Short description:

Resets the current focus (refer to **FOCUS** command)

Syntax:

`ENDFOCUS`

Description:

The `ENDFOCUS` command resets the current focus. It is associated with the `FOCUS` command. The following commands are broadcast to all currently open components. This command is only valid in a command file.

Example:

```
FOCUS Assembly
ATTRIBUTES code on
ENDFOCUS
FOCUS Source
ATTRIBUTES marks on
ENDFOCUS
```

The `ATTRIBUTES` command is first redirected to the assembly component by the `FOCUS Assembly` command. The code is displayed next to assembly instructions. Then the assembly component is released by the `ENDFOCUS` command and the second `ATTRIBUTES` command is redirected to the source component by the `FOCUS Source` command. Marks are displayed in the source window.

ENDFOR

Short description:

End of a FOR loop

Syntax:

```
ENDFOR
```

Description:

The ENDFOR keyword is associated with the FOR command and terminates a FOR loop.

ENDIF

Short description:

End of an IF condition

Syntax:

```
ENDIF
```

Description:

The ENDIF keyword is associated with the IF command and terminates a conditional block.

ENDWHILE

Short description:

End of a WHILE loop

Syntax:

```
ENDWHILE
```

Description:

The ENDWHILE keyword is associated with the WHILE command and terminates a WHILE loop.

FOCUS

Short description:

Sets the focus on a specified component

Syntax:

`FOCUS component`

Argument:

`component` Component window

Description:

The `FOCUS` command sets the given component window as the destination for all subsequent commands up to the next `ENDFOCUS` command. The focus command eliminates having to repeatedly specify the same command redirection, especially in the case where command files are edited manually. It is not possible to visually notice that a component is the focus for commands. Use the `ACTIVATE` command to activate a component window. This command is valid only in a command file.

Example:

```
FOCUS Assembly
ATTRIBUTES code on
ENDFOCUS
FOCUS Source
ATTRIBUTES marks on
ENDFOCUS
```

The `ATTRIBUTES` command is first redirected to the assembly component by the `FOCUS Assembly` command. The code is displayed next to assembly instructions. Then the assembly component is released by the `ENDFOCUS` command and the second `ATTRIBUTES` command is redirected to the source component by the `FOCUS Source` command. Marks are displayed in the source window.

NOTE: *The* `FOCUS` *command cannot be nested.*

FOR

Short description:

`FOR` loop command

Syntax:

```
FOR [variable =]range [“,” step]
```

Arguments:

`variable`	A symbol or program variable. During execution of the loop, the iteration value is stored in `variable`.
`range`	Integer that specifies start and end condition for the loop
`step`	Constant number defining the increment for the iteration value

Description:

The `FOR` loop allows all commands to be executed, up to the trailing `ENDFOR`, a predefined number of times. The bounds of the range and optional steps are evaluated only at the beginning. A variable (either a symbol or program variable) may be optionally specified, which is assigned to all values in the range during execution of the `FOR` loop.

Assignment happens immediately before comparing the iteration value with the upper boundary. The variable is a copy of the internal iteration value. Modifications on the variable do not impact the number of iterations.

This command is halted by pressing the Esc key.

Example:

```
DEFINE loop = 0
FOR loop = 1..6,1
T
ENDFOR
```

The `T` (trace) command is performed six times.

GOTO

Short description:

Unconditional branch to a label in a command file

Syntax:

```
GOTO Label
```

Argument:

`Label` User-defined label used to mark a place in code

Description:

The `GOTO` command diverts command file execution to the command line that follows `Label`. `Label` must be defined in the current command file. The `GOTO` command fails if `Label` is not found. A label can be followed on the same line only by a comment.

No MCUez command is allowed on the same line as a label.

Example:

```
GOTO MyLabel
...
...
MyLabel:  // comments
```

When the instruction `GOTO MyLabel` is reached, the program pointer jumps to `MyLabel` and follows program execution from this position.

GOTOIF

Short description:

Conditional branch to a label in a command file

Syntax:

`GOTOIF <condition> <Label>`

Arguments:

`condition`	Defined as in C language syntax
`Label`	User-defined label used to mark a place in code

Description:

The `GOTOIF` command diverts execution of the command file to the command line that follows the label if the condition is true; otherwise, execution continues on the next line in the command file. The `GOTOIF` command fails if the condition is true and the label is not found.

A label can be followed on the same line only by a comment. No MCUez command is allowed on the same line as a label. Labels must be defined in the current command file.

Example:

```
DEFINE jump = 0
...
DEFINE jump = jump + 1
...
GOTOIF  jump == 10 MyLabel
T
...
MyLabel:  // comments
```

The program pointer jumps to `MyLabel` only if `jump == 10`. Otherwise, the `T` (trace) command is executed.

IF

Short description:

Conditional execution

Syntax:

`IF condition`

Argument:

`condition` Defined as in C language syntax

Description:

The conditional commands (for example, the `IF`, `ELSEIF`, `ELSE`, and `ENDIF` subcommands) allow different command sections to be executed depending on the result of the corresponding conditions.

Conditional blocks may be nested. A conditional block can be specified to start inside an `IF`, `ELSEIF`, or `ELSE` command block.

The conditions of the `IF` and `ELSEIF` commands, respectively, guard all commands up to the next `ELSEIF`, `ELSE`, or `ENDIF` command on the same nesting level. The `ELSE` command guards all commands up to the next `ENDIF` command on the same nesting level. Any occurrence of a subcommand not in sequence of `"IF`, zero or more `ELSEIF`, zero or one `ELSE`, `ENDIF"` is an error.

Example:

```
DEFINE jump = 0
...
DEFINE jump = jump + 1
...
IF  jump == 10
T
DEFINE jump = 0
ELSEIF jump == 100
DEFINE jump = 1
ELSE
DEFINE jump = 2
ENDIF
```

The `jump == 10` condition is evaluated as in ANSI C and depending on the test result, the `T` (trace) instruction is executed or the `ELSEIF jump == 100` test is evaluated.

REPEAT

Short description:

REPEAT loop command

Syntax:

REPEAT

Description:

The REPEAT command enables a command sequence to be executed until a specified condition is true. REPEAT blocks can be nested. A REPEAT block can be started inside a REPEAT block.

Example:

```
DEFINE var = 0
...
REPEAT
  DEFINE var = var + 1
  ...
UNTIL var == 2
```

The REPEAT-UNTIL loop is identical to the ANSI C loop. The operation DEFINE var = var + 1 is executed twice, then var == 2 is executed and the loop exits.

This command can be halted by pressing the Esc key.

RETURN

Short description:

Returns from a CALL or CF command

Syntax:

```
RETURN
```

Description:

The RETURN command terminates the current command processing level. If executed within a command file, control is returned to the caller of the command file (for example, the first instance which did not chain execution).

Example in file *d:\demo\cmd1.txt*:

```
...
CALL d:\demo\cmd2.txt
T
...
```

Example in file *d:\demo\cmd2.txt*:

```
...
...
RETURN // returns to the caller
```

The command file *cmd1.txt* calls a second command file *cmd2.txt*. It is necessary to insert the RETURN instruction to return to the caller file. Then the T (trace) command is executed.

UNDEF

Short description:

Undefines a user-defined symbol

Syntax:

```
UNDEF symbol | *
```

Arguments:

`symbol`	User-defined symbol
`*`	Undefines all symbols previously defined with the `DEFINE` command

Description:

The `UNDEF` command removes a symbol definition from the symbol table. `UNDEF` does not undefine symbols that have been defined in the loaded application.

Program variables whose names were redefined using the `DEFINE` command become visible again.

Undefining an undefined symbol is not considered an error.

Example:

```
DEFINE test = 1
...
UNDEF test
```

When the `test` variable is no longer needed, it can be undefined and removed from the symbol table. After `UNDEF test`, the `test` variable cannot be used unless it is redefined.

UNTIL

Short description:

Specifies the condition of a REPEAT loop

Syntax:

```
UNTIL condition
```

Argument:

`condition`	Defined as in C language syntax

Description:

The UNTIL command is associated with the REPEAT command.

Refer to the **REPEAT** command.

WAIT

Short description:

Insert a pause in a command file execution

Syntax:

```
WAIT [time] [;s]
```

Arguments:

`time`	Specifies time interval to pause command execution
`;s`	Pauses command execution until the target is halted (on a breakpoint, exception, etc.). If the target is already halted, execution of the command continues immediately.

Description:

The `WAIT` command pauses command execution for a specified `time` in tenths of a second or pauses until the target is halted when the option `;s` is set.

When no parameter is specified, the command pauses for 50 tenths of a second (5 seconds).

If `time` and `;s` are specified, `time` is used as a timeout. Command execution waits until the target is halted. If within the specified time interval the target is not halted, execution continues as soon as the specified `time` elapses. If the target is already halted, command execution continues immediately.

Example:

```
WAIT 100
T
...
```

Pauses for 10 seconds before executing the `T` (trace) instruction

WHILE

Short description:

`WHILE` loop instruction

Syntax:

```
WHILE condition
```

Argument:

`condition` Defined as in C language syntax

Description:

The `WHILE` command executes a sequence of commands as long as `condition` is true.

`WHILE` blocks can be nested.

This command can be stopped by pressing the Esc key.

Example:

```
DEFINE jump = 0
...
WHILE jump < 100
  DEFINE jump = jump + 1
ENDWHILE
T
...
```

While `jump < 100`, the jump variable is incremented by the expression: `DEFINE jump = jump + 1`. When the loop is exited, the `T` (trace) instruction is executed.

5.7 Target Commands

The target commands monitor MCUez target execution. Target input/output files, target execution control, direct memory editing, and CPU register setup are handled by these commands. Target commands are executed independently of any components that are active.

BC

Short description:

Deletes a breakpoint. BC stands for breakpoint clear.

Syntax:

```
BC [address | * ]
```

Arguments:

`address`	Specifies the address of the breakpoint to be deleted. This address must be in the ANSI format or MCez assembler format. `address` can be replaced by *expression* as shown in the next example.
`*`	Deletes all breakpoints

Description:

`BC` deletes a breakpoint at the specified address. When `*` is specified, `BC` deletes all breakpoints.

Equivalent operation:

Point the breakpoint directly in the **Assembly** or **Source** component window, right click, and choose **Delete Breakpoint** in the pop-up menu, or open the **Breakpoint Setting** dialog by selecting **Show Breakpoint**, then select the breakpoint and click **Delete**.

Example:

```
BC 0x8000
```

This command deletes the breakpoint set at the address `0x8000`. The breakpoint symbol is removed in the **Source** and **Assembly** window. The breakpoint is removed from the breakpoint list.

NOTE: *Correct module names (such as, `FIBO.DBG`) are displayed in the* **Module** *component window.*

```
BC &FIBO.DBG:Fibonacci
```

In this example, an *expression* replaces the address. `FIBO.DBG` is the module name and `Fibonacci` is the function where the breakpoint is cleared.

This example deletes the breakpoint set at the start address of the symbol `Fibonacci` defined in the module `fibo.dbg`.

BD

Short description:

Displays all breakpoints currently defined

Syntax:

`BD`

Description:

In the command line component, the `BD` command displays the list of all breakpoints with addresses and types (temporary, permanent).

For each breakpoint, this information is displayed:

```
<SymbolName> <address> <type>
```

`SymbolName` is the symbol (or function) where the breakpoint is defined.

`address` is where the breakpoint is set.

`type` represents type of breakpoint. `T` stands for temporary breakpoints and `P` for permanent breakpoints.

Example:

```
in>BD
Fibonacci 0x805c T
Fibonacci 0x8072 P
Fibonacci 0x8074 T
main 0x8099 T
```

Currently, one permanent and two temporary breakpoints are set in the function `Fibonacci`, and one temporary breakpoint is set in the function `main`.

NOTE: *This list will not display whether a breakpoint is disabled or active.*

BS

Short description:

Sets a breakpoint. BS stands for breakpoint set.

Syntax:

```
BS address [P|T]
```

Arguments:

`address`	Indicates the address where the breakpoint is to be set. This address can be specified in the ANSI C format or in MCUez assembler format (see **5.5.4 Constant Standard Notation**). `address` can also be replaced by an *expression* as shown in the next example.
`P`	Specifies a permanent breakpoint
`T`	Specifies a temporary breakpoint. A temporary breakpoint is deleted after the debugger stops on it.

Description:

`BS` sets a temporary (`T`) or permanent (`P`) breakpoint at the specified address. If `P` or `T` is not specified, the default is a permanent breakpoint.

Equivalent operation:

Point to a statement in the **Assembly** or **Source** component window, right click, and select **Set Breakpoint** in the pop-up menu.

Example:

```
BS 0x8000 T
```

This command sets a temporary breakpoint at the address `0x8000`.

```
BS $8000 P
```

This command sets a permanent breakpoint at the address `0x8000`.

```
BS &FIBO.DBG:Fibonacci
```

In this example, an *expression* replaces the address. `FIBO.DBG` is the module name and `Fibonacci` is the function where the breakpoint is set.

NOTE: *Module names (for example,* `fibo.dbg`*) are displayed in the* **Module** *component window.*

The previous example sets a breakpoint on the symbol `Fibonacci` defined in `fibo.dbg`.

CD

Short description:

Changes the current working directory

Syntax:

```
CD [path]
```

Argument:

`path` Changes path to another working directory

Description:

The `CD` command changes the current working directory to the directory specified in `path`. When the command is entered with no parameter, the current directory is displayed.

The directory specified must be a valid directory. It should exist and be accessible. When specifying a relative path, make sure the path is relative to the current project directory.

Example:

```
in> cd
C:\mcuez\demo
in> cd ..\prog
C:\mcuez\prog
```

The new project directory is `C:\mcuez\prog`

CF

Short description:

Executes a command file

Syntax:

```
CF [filename] [;C] [;NL]
```

Arguments:

`filename`	Name of file to be executed
`;C`	Chains the command file. Useful in a nested command file.
`;NL`	Specifies that commands from the called file are not displayed in the **Command Line** window or written to the log file.

Description:

The `CF` command reads the commands in the specified command file, which are then executed by the command interpreter. The command file contains ASCII text commands.

Command files can be nested. `CF` (or `CALL`) can be used in a command file to start another command file. By default, once execution from the called command file is complete, the remaining commands in the calling file are executed.

When the `;C` option is entered in the calling file, the command interpreter terminates the calling file after executing the called file.

If the option is omitted, execution of the remaining commands in the calling file is resumed after the commands in the called file have been executed.

Any error halts execution of `CF` file commands.

When the command is entered with no parameter, the **Open File** dialog is displayed. Use this dialog to select the command file to execute.

The `CALL` command is a synonym for the `CF` command.

Example:

Without `;C` option:

if *command1.txt* file contains:

```
bckcolor green
cf command2.txt
bckcolor white
```

if *command2.txt* file contains:

```
bckcolor red
```

Execution:

```
in> cf command1.txt
executing command1.txt
!bckcolor green
!cf command2.txt
executing command2.txt
!bckcolor red
done command2.txt
!bckcolor white
done command1.txt
```

With `;C` option:

if *command1.txt* file contains:

```
bckcolor green
cf command2.txt ;C
bckcolor white
```

if *command2.txt* file contains:

```
bckcolor red
```

Execution:

```
in> cf command1.txt
executing command1.txt
!bckcolor green
!cf command2.txt ;C
executing command2.txt
!bckcolor red
done command2.txt
done command1.txt
```

CR

Short description:

Opens a record file

Syntax:

```
CR [filename][;A]
```

Arguments:

`filename`	User-defined file. If `filename` is not specified, a standard **Open File** dialog is displayed.
`;A`	Opens the specified file in append mode. Records are appended at the end of an existing record file.

Description:

The `CR` command records commands in an external file. Recording continues until a close record file command (`NOCR`) is executed.

If the `;A` option is omitted and `filename` is an existing file, the file contents are erased before records are written to it.

Example:

```
in>cr /mcuez/demo/myrecord.txt ;A
```

The *myrecord.txt* file is opened in append mode.

NOTE: *If no path is specified, MCUez assumes the path to be the current working directory.*

DASM

Short description:

Disassembles source code

Syntax:

```
DASM [<address>|<range>][;OBJ]
```

Arguments:

`address`	Constant expression indicating the address where disassembly begins
`range`	Address range that specifies addresses to be disassembled. When `range` is omitted, a maximum of 16 instructions are disassembled.
`;OBJ`	Displays assembler code in hexadecimal format

Description:

The `DASM` command displays the disassembled code of an application, starting from the address given as a parameter.

When `address` and `range` are omitted, disassembly begins at the address of the instruction that follows the last instruction that was disassembled by the most recent `DASM` command. If this is the first `DASM` command of a session, disassembly begins at the current address in the program counter.

Press the Esc key to stop the command.

Equivalent operation:

Right click in the **Assembly** component window, select **Address...** and enter the address to start disassembling in the **Show PC** dialog.

Example:

```
in>dasm   0xf04b
00F04B    LDHX     #0x0450
00F04E    TXS
00F04F    CLRH
00F050    CLRX
00F051    STX      0x80
00F053    INC      0x80
00F055    LDX      0x80
00F057    JSR      0xF000
00F05A    STX      0x82
00F05C    STA      0x81
00F05E    LDA      #0x17
00F060    CMP       0x80
00F062    BEQ      *-20     /abs = F050
00F064    BRA      *-19     /abs = F053
00F066    DECX
00F067    DECX
```

Disassembled instructions are displayed in the **Command Line** component window.

DB

Short description:

Displays memory bytes

Syntax:

```
DB [<address>|<range>]
```

Arguments:

`address`	Address constant expression indicating the address of the first byte to be displayed
`range`	Address range constant indicating the range of addresses to be displayed. If range is omitted, one byte is displayed.

If address and range are omitted, the first byte displayed is taken from the address following the last memory position displayed by the most recent `DB`, `DW`, or `DL` command or from address `0x0000` (for the first `DB`, `DW`, `DL` command of a session).

Description:

The `DB` command displays hexadecimal and ASCII byte values in a specified range of memory. The command displays one or more lines, depending on the address or range specified. Each line shows the address of the first byte displayed in the line, followed by the number of specified hexadecimal byte values.

The hexadecimal byte values are followed by the corresponding ASCII characters, separated by spaces. Between the eighth and ninth values, a hyphen (-) replaces the space as the separator. Each non-displayable character is represented by a period (.).

The address can be an expression. If an I/O (input/output) register is specified, the `DB` command will use the value of the register as an address. To use the address of the I/O register, specify the ampersand (&) character followed by the register name.

This command can be stopped by pressing the Esc key.

Example:

```
in> DB 0x8000..0x800F
8000: FE 80 45 FD 80 43 27 10-35 ED 31 EC 31 69 70 83
in> DB 0x8000,8
8000: FE 80 45 FD 80 43 27 10
```

Memory bytes are displayed with matching ASCII characters in the command line component.

The following code example displays the byte at the address of the TCR I/O register. I/O registers are defined in a *.reg* file.

```
in> DB &TCR
0012: 5A Z
```

DL

Short description:

Displays memory bytes as longword

Syntax:

```
DL [<address>|<range>]
```

Arguments:

`address`	Address constant expression indicating the address of the first longword to be displayed
`range`	Address range constant indicating the range of addresses to be displayed. If range is omitted, one longword is displayed. If range is omitted, the first longword displayed is taken from the address following the last longword displayed by the most recent `DB`, `DW`, or `DL` command or from address `0x0000` (for the first `DB`, `DW`, or `DL` command of a session).

Description:

The `DL` command displays the hexadecimal values of the longwords at a specified address or in a specified range of memory. The command displays one or more lines, depending on the address or range specified. Each line shows the address of the first longword displayed in the line, followed by the number of specified hexadecimal longword values.

The address can be an expression. If an I/O register is specified, the `DB` command will use the value of the register as an address. To use the address of the I/O register, specify the ampersand (&) character followed by the register name.

This command can be stopped by pressing the Esc key.

Example:

```
in> DL 0x8000,2
8000: FE8045FD 80432710
```

The content of two longwords starting at 0x8000 is displayed as longword values (4 bytes).

```
in> DL 0x100 + 2
```

Displays memory dump of address $102

DW

Short description:

Displays a word

Syntax:

```
DW [address | range]
```

Arguments:

`address`	Address constant expression indicating the address of the first word to be displayed
`range`	Address range constant indicating the range of addresses to be displayed. If range is omitted, one word is displayed

If `address` and `range` are omitted, the first word displayed is taken from the address following the last word displayed by the most recent `DB`, `DW`, or `DL` command or from address `0x0000` (for the first `DB`, `DW`, or `DL` command of a session).

Description:

The `DW` command displays the hexadecimal values of the words in a specified range of memory. The command displays one or more lines, depending on the address or range specified. Each line shows the address of the first word displayed in the line, followed by the number of specified hexadecimal word values.

The address can be an expression. If an I/O register is specified, the `DB` command will use the value of the register as an address. To use the address of the I/O register, specify the ampersand (&) character followed by the register name.

This command can be stopped by pressing the Esc key.

Example:

```
in> dw 0x8000..0x8007
8000: FE80 45FD 8043 2710
```

Contents of the memory range starting at `0x8000` and ending at `0x8007` is displayed as word values (2 bytes).

```
in> DW 0x8000,4
8000: FE80 45FD 8043 2710
```

The content of four words starting at `0x8000` is displayed as word values.

E

Short description:

Evaluates a given expression

Syntax:

```
E <expression>[;O|;D|;X|;C|;B]
```

Arguments:

`expression`	String or arithmetic expression to be evaluated
`;O`	Displays value of expression as octal (base 8) number
`;D`	Displays value of expression as decimal (base 10)
`;X`	Displays value of expression as hexadecimal (base 16)
`;B`	Displays value of expression as binary (base 2)
`;C`	Displays value of expression as an ASCII character. That is, the remainder resulting from dividing the number by 256 is displayed.

All values are displayed in the current font. Control characters (<32) are displayed as decimal.

Description:

The `E` command evaluates an expression and displays the result in the **Command Line** component window. If the expression is the only parameter entered (no option specified), the value of the expression is displayed in the default number base. The result is displayed as a signed number in decimal format and as an unsigned number in all other formats.

Example:

```
in>define a=0x12
in>define b=0x10
in>e a+b = 34
```

The addition operation of the two previously defined variables `a` and `b` is evaluated and the result is displayed in the **Command Line** window.

G

Short description:

Begins execution of the currently loaded application

Syntax:

```
G [address]
```

Argument:

`address`	Address expression. This value is loaded into the program counter before execution starts. When no address is entered, the address in the program counter is not altered and execution begins at the address in the program counter.

Description:

The `G` command starts executing code in the emulated system at the current address in the program counter or at the specified address. The user can specify the program entry point, skipping execution of the previous code.

Example:

```
G 0x8000
```

Program execution is started at address `0x8000`. *RUNNING* is displayed in the status bar. The application runs until a breakpoint is reached or manually stopped.

LF

Short description:

Opens a log file

Syntax:

```
LF [filename][;A]
```

Arguments:

`filename`	DOS filename that identifies the file or device to which the log is written. The command interpreter does not assume a filename extension.
`;A`	Opens the file in append mode. Logged lines are appended to the end of the existing log file.

If the `;A` option is omitted and the specified file exists, the file contents are cleared before logging begins.

Description:

The `LF` command logs commands and responses to an external file or device. While logging remains in effect, any line that is displayed in the **Command Line** window is also written to the log file. Logging continues until a close log file (`NOLF`) command is executed. When the `LF` command is entered with no filename, the **Open File** dialog is displayed.

Use the `LOG` command to specify the type of information to be logged.

Example:

```
LF LOGFILE.TXT ;A
```

The file *logfile.txt* is opened as a log file in append mode. This example assumes the file is in the working directory.

LOG

Short description:

Enables or disables logging of specific information

Syntax:

```
LOG type [=] state {[,] type [=] state}
```

Arguments:

`type` is one of the following types:

`CMDLINE`	Commands entered on the command line
`CMDFILE`	Commands read from a file
`RESPONSES`	Command output responses
`ERRORS`	Error messages
`NOTICES`	Asynchronous event notices, such as breakpoints
`state`	Value is `on` or `off` to enable or disable logging of the type.

Description:

The `LOG` command enables or disables logging of user-specified information in the command line component (and to the log file, when it as been opened with an `LF` command).

Example:

```
LOG ERRORS = OFF, CMDLINE = on
```

Error messages are not logged. Commands entered in the command line component are logged.

By default, when the `LOG` command is not specified, all types are logged.

Responses are results of commands. For examble, for the `DB` command, the displayed memory dump is the response of the command.

Errors are displayed in red in the command line component. Protocol messages are not errors.

Notices are displayed in green on the command line. They denote status information returned asynchronously by the target.

Logging of IF, FOR, WHILE, and REPEAT

When logging is enabled for commands executed from a command file (a command file executed with the CF or CALL command without the NL option and with the CMDFILE type of the LOG command set to ON), all commands executed in an IF block are logged. All commands in a block that are not executed because the corresponding condition is not verified are also logged but preceded with a dash (-).

Example:

When executing this command file:

```
define truth = 1
IF truth
  bckcolor blue
  at 2000 bckcolor white
else
  bckcolor yellow
  at 1000 bckcolor white
ENDIF
```

This log file is generated:

```
!define truth = 1
!IF truth
!  bckcolor blue
!  at 2000 bckcolor white
!else
!-  bckcolor yellow
!-  at 1000 bckcolor white
!ENDIF
```

When logging is enabled for commands executed from a command file, all commands executed in the FOR loop are logged each time they have been executed.

Example:

When executing this file:

```
define i = 1
FOR i = 1..3
   ls
ENDFOR
```

This log file is generated:

```
!define i = 1
!FOR i = 1..3
!   ls
i             0x1 (1)
!ENDFOR
!   ls
i             0x2 (2)
!ENDFOR
!   ls
i             0x3 (3)
!ENDFOR
```

When logging is enabled for commands executed from a command file, all commands executed in the `WHILE` loop are logged each time they are executed.

Example:

When executing this file:

```
define i = 1
WHILE i < 3
   define i = i + 1
ls
ENDWHILE
```

This log file is generated:

```
!define i = 1
!WHILE i < 3
!   define i = i + 1
! ls
i             0x2 (2)
!ENDWHILE
!   define i = i + 1
! ls
i             0x3 (3)
!ENDWHILE
```

LS

Short description:

Displays the list of symbols

Syntax:

```
LS [symbol | *][;(C |S)]
```

Arguments:

`symbol`	A restricted regular expression that specifies the symbol whose values are to be listed. This argument is case sensitive.
`*`	Lists all symbols
`;C`	Lists symbols in accepted format, which consists of a `DEFINE` command for each symbol
`;S`	Lists symbol table statistics following the list of symbols

Description:

In the command line component, the `LS` command lists the values of symbols defined in the symbol table and also by the user. There is no limit to the number of symbols that can be listed. The size of memory determines the symbol table size. Use the `DEFINE` command to define symbols and the `UNDEF` command to delete symbols. Symbols listed with the `LS` command are split in two parts: applications symbols (symbols defined in the application currently loaded) and user symbols (symbols defined on the command line using the `DEFINE` command). For application symbols, `LS` displays the address of the symbol. For user-defined symbols, `LS` displays the value of the symbol.

Example:

```
in> ls
User Symbols:
j               0x2      (2)
Application Symbols:
counter         0x80     (128)
fiboCount       0x81     (129)
j               0x83     (131)
n               0x84     (132)
fib1            0x85     (133)
fib2            0x87     (135)
fibo            0x89     (137)
Fibonacci       0xF000   (61440)
Entry           0xF041   (61505)
```

When an LS is performed on a single symbol that is an application variable as well as a user symbol, the application variable is displayed.

Example with `j` being an application symbol as well as a user symbol:

```
in>ls j
Application Symbol:
j            0x83 (131)
```

MEM

Short description:

Displays the memory map

Syntax:

`MEM`

Description:

The `MEM` command displays a representation of the current memory map of the system and the lower and upper boundaries of the internal module that contain the MCU registers.

Example:

```
in>mem
Type         Addresses                  Comment
-------------------------------------------------------------------------
IO               0..3F        PRU or TOP TOP board resource or the PRU

NONE            40..4F        NONE
RAM             50..64F       RAM
NONE           650..7FF       NONE
EEPROM         800..A7F       EEPROM
NONE          A80..3DFF       NONE
ROM          3E00..FDFF        ROM
IO           FE00..FE1F        PRU or TOP TOP board resource or the PRU
NONE         FE20..FFDB        NONE
ROM          FFDC..FFFE        ROM
COP          FFFF..FFFF        special ram for cop
RT MEM           0..3FF        (enabled)
-------------------------------------------------------------------------
```

MS

Short description:

Sets memory bytes

Syntax:

```
MS range list
```

Arguments:

`range`	Address range constant that defines the block of memory to be set to the values of the bytes in the list
`list`	List of byte values to be stored in the block of memory

Description:

The `MS` command initializes a specified block of memory to a specified list of byte values. When the `range` is wider than the `list` of byte values, the list is repeated as many times as necessary to fill the memory block.

When the `range` is not an integer multiple of the length of the `list`, the last copy of the list is truncated appropriately. This command is identical to the write bytes (`WB`) command.

Example:

```
MS 0x1000..0x100F 0xFF
```

The memory range between addresses `0x1000` and `0x100F` is filled with the `0xFF` value.

NB

Short description:

Sets the number base for expression values in arguments.

Syntax:

```
NB [base]
```

Argument:

`base` Decimal integer specifying the number base (2, 8, 10, or 16)

Description:

The `NB` command changes or displays the default number `base` for the constant values in expressions. The initial default number base is 10 (decimal) and can be changed to 16 (hexadecimal), 8 (octal), 2 (binary), or reset to 10 with this command.

If base is omitted, the current default number base is displayed in the command line window.

Independent of the default base number, the ANSI C standard notation for constant is supported inside of an expression. That means that independent of the current number base hexadecimal or octal constants can be specified using standard ANSI C notation.

Notation	Meaning
`0x----`	Hexadecimal constant
`0----`	Octal constant

Example:

```
0x2F00, /* Hexadecimal Constant */
043,    /* Octal Constant */
```

In the same way, the assembler notation for constant is also supported. That means that independent of the current number base, hexadecimal, octal, or binary constants can be specified using the assembler prefixes.

Notation	Meaning
`$----`	Hexadecimal constant
`@`	Octal constant
`%`	Binary constant

Example:

```
$2F00,  /* Hexadecimal Constant */
@43,    /* Octal Constant */
%10011  /* Binary Constant */
```

When the default number base is 16, constants starting with a letter A, B, C, D, E, or F must be prefixed either by 0x or by $. Otherwise, the command line interpreter cannot detect if an integer constant or a symbol is specified.

Notation	Meaning
`5AFD`	Hexadecimal constant `$5AFD`
`AFD`	In this the Hexadecimal constant $AFD or the symbol, which name is AFD?

Example:

```
in>NB 16
```

The number base is hexadecimal.

NOCR

Short description:

Closes the record file

Syntax:

```
NOCR
```

Description:

The NOCR command closes the current record file. The record file is opened with the CR command.

Example:

```
NOCR
```

The current record file is closed.

NOLF

Short description:

Closes the log file

Syntax:

```
NOLF
```

Description:

The NOLF command closes the current log file. The log file is opened with the LF command.

Example:

```
NOLF
```

The current log file is closed.

P

Short description:

Steps into the program using assembly step over

Syntax:

```
P [address]
```

Argument:

`address`	Address constant expression indicating where execution begins. If `address` is omitted, execution begins at the instruction pointed to by the program counter.

Description:

The P command executes a CPU instruction either at a specified address or current instruction (pointed to by the program counter). This command traces through subroutine calls, software interrupts, and operations involving the following instructions:

- Branch to SubRoutine (BSR)
- Jump to SubRoutine (JSR)
- SoftWare Interrupt (SWI)

For example, if the current instruction is a BSR instruction, the subroutine is executed, and execution stops at the first instruction after the BSR instruction. For instructions that are not in the above list, the P and T commands are equivalent.

When the instruction specified in the P command has been executed, the software displays the content of the CPU registers, the instruction bytes at the new value of the program counter, and a mnemonic disassembly of that instruction.

Example:

```
in>p
A=0x0 HX=0x450 SR=0x70 PC=0xF04E SP=0xFF
00F04E 94    TXS
```

Register contents are displayed and the current instruction is disassembled.

RD

Short description:

Displays register contents

Syntax:

```
RD [<list>| *]
```

Arguments:

`list`	List of registers to be displayed. Registers to be displayed are separated by a space. When `RD CPU` is specified, all CPU registers are displayed. If no CPU is loaded, No CPU loaded is displayed as an error message.
`*`	Lists the content of register file currently loaded. The address and size of each register is displayed. If no register file is loaded, the following error message is displayed: No register file loaded.

If `list` is omitted, the list and any other parameters of the previous `RD` command are used. For the first `RD` command of a session, all CPU registers are displayed.

Description:

The `RD` command displays the content of specified registers. The display for a register includes both the mnemonic and the hexadecimal value of the register. If the specified register is not a CPU register, it is considered to be an I/O register. The debugger looks for the specified register in the loaded register file. This file is called *MCUxxxx.reg* (where *xxxx* is a number related to the MCU).

Example:

```
in>RD A X
A = 0x1
X = 0xF
```

Contents of registers A and X are displayed.

Example:

```
in>RD CPU //
```

will display all CPU registers

RS

Short description:

Sets a register

Syntax:

```
RS <register>[=]<value> { [,]<register>[=]<value>}
```

Arguments:

`register`	Specifies the name of a CPU register or register in the register file to be changed
`value`	Integer constant expression in ANSI C or MCUez assembler format representation

Description:

The `RS` command places specified values into specified registers.

An equal sign (=) may be used to separate the register name from the value assigned to the register; otherwise, they must be separated by a space. The contents of any number of registers may be set using a single `RS` command. If the specified register is not a CPU register, it is considered to be an I/O register. The debugger looks for the specified register in the loaded register file. This file is called *MCUxxxx.reg* (where *xxxx* is a number related to the MCU).

Example:

```
in>rs a=0xff hx=0x7fff
```

The new content of register `a` is `0xff` and register `hx` is `0x7fff`. The **Register** component window is updated with the new values.

S

Short description:

Stops execution of loaded application

Syntax:

`S`

Description:

The `S` command stops execution of the application. Use the `Go` (`G`) command to start or continue execution.

Example:

```
in>s
STOPPING
HALTED
```

The current application is halted.

SAVE

Short description:

Saves a memory block in S-record format

Syntax:

```
SAVE <range> <filename> [<offset>][;A]
```

Arguments:

`range`	Address range constant that defines the block of memory to be saved in a Motorola S-record file
`filename`	Specified file where the records are written
`offset`	Optional offset to add or subtract from addresses when writing S-records. The default is 0.
`;A`	Appends the saved S-records to the end of an existing file. If this option is omitted, and the specified file exists, the file contents are cleared before saving the S-records.

Description:

The SAVE command saves a specified block of memory to a specified file in Motorola S-record format. The memory block can be reloaded later using the load S-record (SREC) command.

Example:

```
SAVE 0x1000..0x2000 DUMP.SX ;A
```

The memory range `0x1000` to `0x2000` is appended to the *dump.sx* file.

NOTE: *If no path is specified, the path is assumed to be the current working directory.*

SET

Short description:

Sets a new target

Syntax:

```
SET targetName
```

Argument:

`targetName`	Name of target (without extension) to be set

Description:

Sets a new target for the debugger. The file *targetName.tgt* must be available in the *PROG* directory.

Example:

```
SET ESL
```

The ESL target is set in the MCUez debugger.

SREC

Short description:

Loads the content of an S-record file in memory

Syntax:

```
SREC <filename> [offset]
```

Argument:

`filename`	Specified file containing records
`offset`	A signed value added to the addresses stored in the file when loading the file contents

Description:

The `SREC` command loads Motorola S-records from a specified file.

Example:

```
SREC DUMP.SX
```

The *dump.sx* file is loaded into memory.

NOTE: *If no path is specified, the path is assumed to be the current working directory.*

T

Short description:

Traces program instructions. Program trace begins at a specified address.

Syntax:

```
T [address][,count]
```

Arguments:

`address`	Address constant expression indicating where execution begins. If `address` is omitted, the instruction pointed to by the program counter is the first instruction traced.
`count`	Integer constant expression [1, 65,535], that specifies the number of instructions to be traced. If `count` is omitted, one instruction is traced.

Description:

The `T` command executes one or more instructions starting at a specified address or at the current instruction (address in program counter). The `T` command traces into subroutine calls and software interrupts. For example, if the current instruction is a branch to subroutine instruction (BSR), the BSR is traced, and execution stops at the first instruction of the subroutine. After executing the last (or only) instruction, the `T` command displays the contents of the CPU registers, the instruction bytes at the new address in the program counter, and a mnemonic disassembly of the current instruction.

This command can be stopped by pressing the Esc key.

Example:

```
in>T 0xF030
TRACED
A=0x0 HX=0x7F02 SR=0x62 PC=0xF032 SP=0x44D
00F032 B787   STA 0x87
```

Contents of registers are displayed and the current instruction is disassembled.

WB

Short description:

Sets a specified block of memory to a specified list of byte values

Syntax:

```
WB range list
```

Arguments:

`range`	Address range constant that defines the block of memory to be initialized to the values of the bytes in the list
`list`	List of byte values to be stored in the block of memory

Description:

The `WB` command sets a specified block of memory with a specified list of byte values. When the range is wider than the list of byte values, the list of byte values is repeated as many times as necessary to fill the memory block. This command is identical to the memory set (`MS`) command.

Example:

```
WB 0x0401 0x19 0x69
```

This command fills up the memory range `0x0401..0x0402` with the byte values `0x19` and `0x69`.

Example:

```
WB 0x0205..0x0220 0xFF
```

This command fills up the memory range `0x0205..0x0220` with the byte value `0xFF`.

WL

Short description:

Sets a specified block of memory to a specified list of longword values

Syntax:

```
WL range list
```

Arguments:

`range`	Address range constant that defines the block of memory to be set to the longword values in the list
`list`	List of longword values to be stored in the block of memory

Description:

The `WL` command initializes a specified block of memory with a specified list of longword values. When the range is wider than the list of longword values, the list of longword values is repeated as many times as necessary to fill the memory block. When the range is not an integer multiple of the length of the list, the last copy of the list is truncated accordingly.

Example:

```
WL 0x2000 0x0FFFFF0F
```

This command fills up the memory address `0x2000..0x2003` with the longword value `0x0FFFFF0F`.

WW

Short description:

Sets a specified block of memory to a specified list of word values

Syntax:

```
WW range list
```

Arguments:

`range`	Address range constant that defines the block of memory to be set to the word values in the list
`list`	List of word values to be stored in the block of memory

Description:

The `WW` command sets a specified block of memory with a specified list of word values. When the range is wider than the list of word values, the list of word values is repeated as many times as necessary to fill the memory block. When the range is not an integer multiple of the length of the list, the last copy of the list is truncated accordingly.

Example:

```
WW 0x2000..0x200F 0xAF00
```

This command fills up the memory range `0x2000..0x200F` with the word value `0xAF00`.

5.8 Component Commands

The commands listed in this section monitor the MCUez debugger environment and manage component windows.

ACTIVATE

Short description:

Activates a component window

Syntax:

```
ACTIVATE <component window>
```

Description:

Activates a component window. The window is displayed in the foreground and its title bar is highlighted.

Example:

```
ACTIVATE Memory
```

This command will make the memory component the active window.

ATTRIBUTES

Short description:

Sets the display and formatting attributes for a component window. Usually, this command is not specified interactively by the user. However, this command can be written in a session record file or in a script command file to save and reload component window layouts. An equivalent operation is typically possible, using MCUez menus and drag and drop, as described under the equivalent operations headings that follow.

In the Assembly Component

Syntax:

```
ATTRIBUTES list
   list=command{,command}
command= ADR ON|OFF | SMEM range | SPC address
|CODE(ON|OFF) | ABSADR (ON|OFF) | TOPPC address
```

Arguments:

`address`	Address to be located
`range`	Memory range to be located
`module`	Specified module
`CODE on`	Switches on the display of the machine code
`CODE off`	Switches off the display of the machine code
`ADR on`	Switches on the display of the addresses in front of the disassembly instruction
`ADR off`	Switches off the display of the addresses in front of the disassembly instruction
`ABSADR on`	Switches on the display of the absolute address for destination of branch instructions
`ABSADR off`	Switches off the display of the absolute address for destination of branch instructions
`SPC address`	PC address location
`TOPPC address`	Address location of the first line of the PC

Description:

The `ATTRIBUTES` command sets the display and state options for the **Assembly** component window.

The `ADR` command displays or hides the address of a disassembled instruction. `ON OFF` is used to switch the display of the address in front of disassembled instructions on or off.

The `SMEM` (show memory range) and `SPC` (show PC address) scroll the assembly component to the corresponding address or range code location and select/highlight the corresponding assembler instructions or set of instructions.

The `CODE` command displays or hides the machine code of the disassembled instruction. `ON OFF` is used to switch the machine code on or off.

The `ABSADR` command shows or hides the display of the destination address in a disassembled instruction like branch to. `ON OFF` is used to switch the display of the destination absolute address on or off.

The `TOPPC` command specifies the program counter of the first visible line.

Equivalent operations:

`ATTRIBUTES ADR`	Select menu **Assembly	Display Adr**.
`ATTRIBUTES SMEM`	Select a range in **Memory** component window and drag it to the **Assembly** component window.
`ATTRIBUTES SPC`	Drag a register to the **Assembly** component window.
`ATTRIBUTES CODE`	Select menu **Assembly	Display Code**.

Example:

```
Assembly < ATTRIBUTES ADR ON,CODE ON, SMEM 0x800,16
```

Addresses and hexadecimal codes are displayed in the **Assembly** component window, and assembly instructions at addresses 0x800,16 are highlighted.

In the Register Component

Syntax:

`ATTRIBUTES list`

`list=command{,command})`

`command=FORMAT(hex|bin) | VSROLLPOS vposition | HSCROLLPOS hposition`

Where `vposition =` *expression*

Where `hposition =` *expression*

Arguments:

`VSCROLPOS` vposition = 1->	The second line of registers is on top of the register component.
`VSCROLPOS` vposition = 0 ->	Returns to the default display. The first line of registers is on top of the register component.
`HSCROLPOS` hposition = 1->	The second column of registers is on the left hand side of the register component.

HSCROLPOS hposition = 0->	Returns to the default display. The first column of registers is on the left hand side of the register component. The HSCROLLPOS command sets the position (in column: a column is about the tenth of the greatest register or bitfield width) of the horizontal scroll box.
hex	Sets format representation to hexadecimal
bin	Sets the format representation to binary

Description:

The ATTRIBUTES command sets the display and state options of the **Register** component window. The FORMAT command sets the display format of register values. The VSCROLLPOS command sets the position (in line: each register and bit-field have the same height, which is the height of a line) of the vertical scroll box.

The attribute VSCROLLPOS enables scrolling vertically in the register component. The expression specified is an absolute and positive value for scrolling. This command is effective only, when a vertical scroll bar is present on the right of the register component.

The attribute HSCROLLPOS enables horizontal scrolling in the register component. The expression specified is an absolute and positive value for scrolling.

Equivalent operations:

ATTRIBUTES FORMAT	Select menu option ***Register*** **|** ***Options***.
ATTRIBUTES VSCROLLPOS	Scroll vertically in the **Register** component window.
ATTRIBUTES HSCROLLPOS	Scroll horizontally in the **Register** component window.

Example:

```
Register < ATTRIBUTES FORMAT BIN
```

Contents of registers are displayed in binary format in the **Register** component window.

```
Register < ATTRIBUTES VSCROLLPOS 3
```

Scrolls three positions down. The fourth line of registers is displayed on the top of the register component.

```
Register < ATTRIBUTES VSCROLLPOS 0
```

Returns to the default display. The first line of registers is displayed on the top of the register component.

```
Register < ATTRIBUTES HSCROLLPOS 5
```

Scrolls five positions right. The sixth column of registers is displayed on the left of the register component.

```
Register < ATTRIBUTES HSCROLLPOS 0
```

Returns to the default display. The first column of registers is displayed on the left of the register component.

In the Source Component

Syntax:

```
ATTRIBUTES list

   list=command{,command}

   command= SPC address | SMEM range | SMOD module
   | SPROC numberAssociatedwithProcedure | MARKS
   (ON|OFF)
```

Arguments:

`address`	Address to be located
`range`	Memory range to be located
`module`	Specified module
`MARKS on`	Displays breakpoint marks
`MARKS off`	Hides breakpoint marks

Description:

The ATTRIBUTES command sets the display and state options of the **Source** component window.

The SMEM (show memory range) command and SPC (show PC address) command displays the corresponding module's source text, scrolls to the corresponding text range location or text address location, and highlights the corresponding statements.

The SMOD (show module) command displays the corresponding module's source text. If the module is not found, a message is displayed in the **Command Line** component window. The SPROC (show procedure) command loads the corresponding module's source text, scrolls to the corresponding procedure, and selects (highlights) the statement in the procedure chain of this procedure.

The SPROC command is applicable only for C source-level debugging. The numberAssociatedwithProcedure is the level of the procedure in the procedure chain.

The MARKS command displays or hides the marks. Marks are visible in the **Source** component window.

Example:

```
Source < ATTRIBUTES MARKS ON
```

Equivalent operations:

ATTRIBUTES SPC	Drag and drop from register component to source component
ATTRIBUTES SMEM	Drag and drop from memory component to source component
ATTRIBUTES SMOD	Drag and drop from module component to source component
ATTRIBUTES SPROC	Drag and drop from procedure component to source component
ATTRIBUTES MARKS	Select menu **Source | Marks**.

In the Data Component

Syntax:

```
ATTRIBUTES list
   list=command{,command}
   command=FORMAT(bin | oct | hex | signed | unsigned
   | symb) | MODE(automatic |periodical | locked |
   frozen) | SMOD module | UPDATERATE rate
```

Arguments:

`hex`	Sets format representation to hexadecimal
`oct`	Sets format to octal
`bin`	Sets format to binary
`symb`	Sets format as a symbol
`signed`	Displays value in signed decimal format
`unsigned`	Displays value in unsigned decimal format
`periodical`	Set `Data` component to periodical update mode.
`locked`	Set `Data` component to locked update mode.
`frozen`	Set `Data` component to frozen update mode.
`automatic`	Set `Data` component to automatic update mode.
`module`	Specified module
`rate`	Update rate in tenth of second. Valid value is 1 to 600.

Description:

The `ATTRIBUTES` command sets the display and state options of the **`Data`** component window.

The `FORMAT` command selects how the list of variables is represented. The format can be one of the following: binary, octal, hexadecimal, signed decimal, unsigned decimal, or symbolic.

The `MODE` command selects the display mode for variables.

In automatic mode (default), variables are updated when target is stopped. Variables from the currently executed module or procedure are displayed in the data component.

In locked and frozen mode, variables from a specific module are displayed in the data component.

In locked mode, variable values displayed in the data component are updated when the target is stopped.

In frozen mode, variable values displayed in the data component are not updated when the target is stopped.

In periodical mode, variables are updated at regular time intervals when the target is running. The default update rate is 1 second, but it can be modified by steps of 100 ms using the associated dialog box or the UPDATERATE command.

The UPDATERATE command sets the update rate for the data component. This command is relevant only when the update mode for the data component is set to periodical.

The SMOD command displays global variables for the corresponding module.

Equivalent operations:

`ATTRIBUTES FORMAT`	Select menu option **Data** | **Format...**
`ATTRIBUTES MODE`	Select menu option **Data** | **Mode...**
`ATTRIBUTES SMOD`	Drag and drop from module component to data component.
`ATTRIBUTES UPDATERATE`	Select menu option **Data** | **Mode** | **Periodical**.

Example:

```
Data < ATTRIBUTES MODE FROZEN
```

In the data component, the update mode for updating global variables is frozen mode. Variables are not refreshed when the application is halted.

In the Memory Component

Syntax:

```
ATTRIBUTES list
   list=command{,command})
   command=FORMAT(bin|oct|hex|signed|unsigned)
   | WORD number | ADR (ON|OFF) | ASC (ON|OFF)
   | ADDRESS address | SPC address | SMEM range
   | SMOD module | MODE (Automatic | Periodical
   | Frozen) | UPDATRATE rate
```

Arguments:

`hex`	Sets format representation to hexadecimal.
`oct`	Sets format to octal
`bin`	Sets format to binary
`signed`	Displays value in signed decimal format
`unsigned`	Displays value in unsigned decimal format
`number`	Requested word size. The word size can be 1, 2, or 4 bytes.
`address`	Memory address to be located
`range`	Memory range to be located
`module`	Specified module
`periodical`	Set data component to periodical update mode.
`frozen`	Set data component to frozen update mode.
`automatic`	Set data component to automatic update mode.
`rate`	Update rate in tenth of a second. Valid values are 1 to 600.

Description:

The `ATTRIBUTES` command sets the display and state options of the **Memory** component window.

The `FORMAT` command selects the diplay format in the memory window. Format can be set to binary, octal, hexadecimal, signed decimal, unsigned decimal, or symbolic.

The `WORD` command selects the word size of the **Memory** window. The word size can be 1, 2, or 4 bytes.

The `ADR` command displays or hides the address in front of the memory lines.

The `ASC` command displays or hides the ASCII dump at the end of the memory lines.

The `ADDRESS` command scrolls the **Memory** component window to the specified address and displays the corresponding memory address.

The `SPC` (show PC), `SMEM` (show memory) commands scroll the **Memory** component window to the specified address or range of memory.

The `SMOD` (show module) commands scroll the **Memory** component window to the address of the first global variable in the specified module.

The MODE command selects the memory component display mode.

In automatic mode (default), the memory component is updated when the target is stopped.

In frozen mode, memory displayed in the **Memory** component window is not updated when the target is stopped.

In periodical mode, the content of the **Memory** component window is updated at regular time intervals when the target is running. The default update rate is 1 second, but it can be modified by steps of 100 ms using the associated dialog box or the UPDATERATE command.

Equivalent operations:

`ATTRIBUTES FORMAT`	Select menu option **Memory** | **Format**.
`ATTRIBUTES WORD`	Select menu option **Memory** | **Word Size**.

`WORD 1` — The content is displayed in byte format.

`WORD 2` — The content is displayed in word (2-byte) format.

`WORD 4` — The content is displayed in long (4-byte) format.

`ATTRIBUTES ADR`	Select menu option **Memory** | **Display** | **Address**.
`ATTRIBUTES ASC`	Select menu option **Memory** | **Display** | **ASCII**.
`ATTRIBUTES ADDRESS`	Select menu option **Memory** | **Address...**
`ATTRIBUTES SMEM`	Drag and drop from data component (variable) to memory component.
`ATTRIBUTES SMOD`	Drag and drop from source component to memory component.

Example:

```
Memory < ATTRIBUTES ASC OFF, ADR OFF
```

ASCII dump and addresses are removed from the memory component.

AUTOSIZE

Short description:

Autosizes component windows in the main window

Syntax:

`AUTOSIZE on|off`

Description:

`AUTOSIZE` enables or disables window autosizing.

When `on`, component windows are automatically adapted to the MCUez main window when it is resized.

Example:

```
AUTOSIZE off
```

Window autosizing is disabled.

BCKCOLOR

Short description:

Sets the background color

Syntax:

```
BCKCOLOR color
```

Argument:

`color` Available colors: `BLACK, GREY, LIGHTGREY, WHITE, RED, YELLOW, BLUE, CYAN, GREEN, PURPLE, LIGHTRED, LIGHTYELLOW, LIGHTBLUE, LIGHTCYAN, LIGHTGREEN, LIGHTPURPLE`

Description:

`BCKCOLOR` sets the background color. Ensure that the text will be visible by using different colors for the font and background. Do not use colors that have a specific meaning in the **Command Line** window. These colors are:

- Red — To display error messages
- Blue — To echo commands
- Green — To display asynchronous events

When `WHITE` is specified as an argument, the default background color is set for all component windows.

Example:

```
BCKCOLOR LIGHTCYAN
```

The background color for all currently opened windows will be set to `LIGHTCYAN`.

CLOSE

Short description:

Closes a component window

Syntax:

```
CLOSE <component> | *
```

Arguments:

`component`	Component window in the debugger environment
`*`	Closes all component windows, including command line component

Description:

`CLOSE` closes the specified component window or all windows.

Example:

```
CLOSE Memory
```

The **Memory** component window is closed.

FILL

Short description:

Fills a memory range with a value

Syntax:

```
FILL range value
```

Description:

In the memory component, the `FILL` command fills a corresponding range in memory component with the defined value. The value must be a single byte pattern (higher bytes ignored).

Equivalent operation:

The **Fill Memory** dialog is available from the **Memory** pop-up menu by selecting **Fill...** or selecting the **Memory | Fill...** menu option.

Examples:

```
in>FILL 0x8000..0x8008 0xFF
```

The memory range `0x8000..0x8008` is filled with the value `0xFF`.

FIND

Short description:

Finds and highlights a pattern

Syntax:

```
FIND <“string”> [;B] [;MC] [;WW]
```

Arguments:

`string`	Pattern to match
`;B`	Searches backward, default is forward
`;MC`	Match case
`;WW`	Match whole word

Description:

In the source component, the `FIND` command is used to search a specified pattern in the source file currently loaded. If the pattern has been found, it is highlighted. The search is forward (default), backward (`;B`), match case sensitive (`;MC`), or match whole word (`;WW`). The operation starts from the currently highlighted statement or from the beginning of the file (if nothing is highlighted). If the item is found, the **Source** window is scrolled to the position of the item and the item is highlighted.

Equivalent operation:

Select **Source | Find...** or open the **Source** pop-up menu and select **Find...** to open the **Find** dialog.

Example:

```
in>FIND “this” ;B ;WW
```

The `“this”` string (considered to be a whole word) is searched in the **Source** component window. The search is performed backward.

FONT

Short description:

Sets the text font

Syntax:

```
FONT ‘FontName’ [size][color]
```

Arguments:

`FontName`	Name of a valid font installed on the system. If the specified font is not found, Windows will try to find an available font that best fits the specification.
`size`	Defines the font size. This parameter must be a positive numerical constant representing a point size.
`color`	Defines the font color: `BLACK, GREY, LIGHTGREY,WHITE, RED, YELLOW, BLUE, CYAN, GREEN, PURPLE, LIGHTRED, LIGHTYELLOW, LIGHTBLUE, LIGHTCYAN, LIGHTGREEN, LIGHTPURPLE`

Description:

The `FONT` command changes the font type, color, and size.

The color specified in the `FONT` command applys to all text displayed in the component windows. Do not use the same color for the font and background; otherwise, text will not be visible.

Font color exceptions — Exceptions apply to the command line component and source component. Colors used in the command line component are fixed and cannot be changed. The prompt and responses are displayed in black, commands in blue, and error messages in red.

Colors used in the source component are also fixed. ANSI C keywords are displayed in blue, comments in green, and strings in red. The rest of the code is displayed in the color specified in the `FONT` command.

Equivalent operation:

The **Component | Fonts...** menu option will display the **Font** dialog.

Example:

```
FONT ‘Arial’ 8 BLUE
```

Text is written in blue using the Arial 8 point font.

HELP

Short description:

Displays a list of available commands

Syntax:

`HELP`

Description:

In the command line component, the `HELP` command displays all available primary commands. Subcommands for the `ATTRIBUTES` command are not listed. Component specific commands for components that are not opened will also not be listed.

Example:

```
in>help
 MCUez:
   VER            Shows version information
   AUTOSIZE       Selects window sizing mode
   OPEN           Opens a component window
   SET            Loads a target component
 ...
 ...
```

LOAD

Short description:

Loads an application file

Syntax:

```
LOAD [applicationName]
```

Argument:

`applicationName` Name of application. If the *.abs* file is not located in the project directory, the complete path must be specified.

Description:

Loads an application file for debugging. If no parameter is specified, the **Load Executable File** dialog is opened.

Example:

```
LOAD FIBO.ABS
```

Loads the application *fibo.abs*

OPEN

Short description:

Opens a component window

Syntax:

```
OPEN componentName [x y width height ][;I]
```

Arguments:

`componentName`	Name of the component window to be opened
`x`	X axis of the component window from upper-left corner
`y`	Y axis of the component window from upper-left corner
`width`	Component window width
`height`	Component window height
`I`	Opens the component window as an icon

`x, y, width,` and `height` are specified as a percentage of the MCUez main window.

Description:

`OPEN` opens a component window. If `I` is set, the component window is opened as an icon.

Example:

```
OPEN Source 0 78 60 22
```

The **Source** component window is opened at the specified position, width, and height.

SLAY

Short description:

Saves the general window layout

Syntax:

```
SLAY <filename>
```

Argument:

`filename`	Name of file (with full path) where component window layout is saved

Description:

The `SLAY` command is used to save the layout of all component windows available in the main window.

Example:

```
slay /mcuez/demo/mylayout.hwl
```

The current layout is saved in the file *mylayout.hwl*.

NOTE: *Layout files usually have an .hwl extension. However, any file extension can be specified. If no path is specified, the destination directory is the current project directory.*

SMEM

Short description:

Shows a memory range

Syntax:

```
SMEM <range>
```

Argument:

`range`	Address range constant that defines the block of memory to be displayed

Description:

This command applies to the **Source**, **Assembly**, and **Memory** component windows.

In the source component, the `SMEM` command displays the corresponding module's source text, scrolls to the corresponding text location (the code address), and highlights the statements which correspond to this code address range.

In the assembly component, the `SMEM` command scrolls the assembly component, shows the location (assembler address), and highlights the assembler lines of the address range.

In the memory component, the `SMEM` command scrolls the memory dump component and shows the locations (memory address) of the address range given as the range argument.

Example:

```
in>Source < SMEM 0x8000..0x8008
```

The **Source** component window scrolls to the source code corresponding to the instruction located at address `0x8000`. The source code generating code between `0x8000` and `0x8008` is highlighted.

```
in>Memory < SMEM 0x8000,8
```

The **Memory** component window scrolls to the address `0x8000` and the memory range `0x8000..0x8007` is highlighted.

```
in> SMEM 0xF040..0xF050
```

The **Source**, **Assembly**, and **Memory** component windows scroll to `0xF040` and the range `0xF040` to `0xF050` is highlighted.

SMOD

Short description:

Shows a module

Syntax:

```
SMOD <module>
```

Argument:

`module`	Name of a module bound to the application. The module name should contain no path. The module extension (*.dbg* for assembly sources or *.c* for C sources, etc.) must be specified.

Description:

This command applies to the **Source**, **Data** and **Memory** component windows.

In the source component, the `SMOD` command displays the corresponding module's source text. If the module is not found, a message is displayed in the command line window.

In the data component, the `SMOD` command displays the corresponding module's global variables.

In the memory component, the `SMOD` command scrolls the memory component and highlights the first global variable.

The module is searched for in the directories specified in the `GENPATH` environment variable. An error message is displayed:

- If the specified module is not bound to the application currently loaded
- If no application is loaded

Example:

```
in>Data < SMOD fibo.dbg
```

Global variables defined in the *fibo.dbg* module are displayed in the **Data** component window.

NOTE: *The module parameter must be a module filename listed in the* **Module** *component window.*

SPC

Short description:

Shows the specified address in a component window

Syntax:

```
SPC <address>
```

Description:

This command applies to the **Source**, **Assembly**, and **Memory** component windows.

In the source component, the SPC command loads the corresponding module's source text, scrolls to the corresponding text location (the code address), and highlights the statement that corresponds to this code address.

In the assembler component, the SPC command scrolls the assembly component, shows the location (assembler address), and highlights the assembler instruction of the specified address.

In the memory component, the SPC command shows the location (memory address) of the specified address.

Example:

```
in>Assembly < SPC 0x8000
```

The **Assembly** component window scrolls to the address 0x8000 and the instruction is highlighted.

SPROC

Short description:

Shows information associated with the specified procedure. This command is available only when performing C source debugging using the HIWARE compiler. Refer to **Appendix B. C Source-Level Debugging** for information on C source debugging.

UPDATERATE

Short description:

Sets the data update mode

Syntax:

```
UPDATERATE rate
```

Argument:

`rate`	Constant number representing tenths of a second, between 1 and 600 tenths of a second (for example, 0.1 to 60 seconds)

Description:

In the data and memory components, the UPDATERATE command is used to set the data refresh update rate. The UPDATERATE command has effect only when the data or memory component is set to periodical mode.

Example:

```
in>Memory <updaterate 30
```

This rate sets memory to update every three seconds.

VER

Short description:

Displays the MCUez debugger and component windows' version numbers

Syntax:

VER

Example:

```
in>ver
MCUez              1.0.10
MCUez Engine       1.0.22
Source             1.0.9
Assembly           1.0.8
Register           1.0.5
Memory             1.0.9
Data               1.0.11
Command Line       1.0.7
Module             1.0.3
MotoESL Target     1.0.15
  Probe ID         0x3C1
  BOOT ROM         V-1.01
  EEPROM           V-2.34
  Protocol         V-2.41
  Server           V-2.20
```

ZOOM

Short description:

Zooms in/out on an array or variable

Syntax:

```
ZOOM (address in | address out)
```

Arguments:

`address`	Address of a structure or pointer variable that should be zoomed-in or zoomed-out

Description:

In the data component, the `ZOOM in` command is used to display the member fields of structures by revealing the structure. Member fields are not expanded in place. The display of the member fields replaces the previous view. The `ZOOM out` command is used to return to the nesting level indicated by the given identifier. Addresses are not needed to zoom out. Simply type `ZOOM out`.

Example:

```
in>ZOOM 0x1FE0 in
```

The variable structure located at address `0x1FE0` is zoomed in.

NOTE: *This command is relevant when C-source debugging.*

```
in>zoom &_StartupData
```

The previous code example zooms in on the `_StartupData` structure (`&_StartupData` is the address of the `_StartupData` structure).

5.9 Command Files

The command files *startup.cmd*, *reset.cmd*, *preload.cmd*, and *postload.cmd* are MCUez system command files.

- *startup.cmd* is executed when a target is loaded (the target defined in the *project.ini* file or by selecting the **Component | Set Target** menu option).
- *reset.cmd* is executed when the **Target Name | Reset** menu option is selected (**Target Name** represents the actual target, for example, MMDS0508, SDI, etc.).
- *preload.cmd* is executed before loading an *.abs* application file (select **Target Name | Load...** to load an executable file).
- *postload.cmd* is executed after loading an *.abs* application file.

Appendix A. Register Description File

A.1 Contents

A.2 Introduction

When loading an MCUez target, definitions of the I/O (input/output) registers are loaded from a *.reg* file. This allows the names of these registers to be used as parameters for commands or as operands in an expression. The syntax of the file is defined in **A.4 Description Using Extended Backus–Naur Form (EBNF)**.

There may be several different files depending on the MCU used. The name of the correct file is derived from the MCU identification number (MCU Id) in the following way:

MCUxxxx.REG

where *xxxx* is the MCU Id in hexadecimal representation. This file is expected to be found in the directory where the program files are located (for example, *..\PROG\REG*). If this file is not found, the *default.reg* file is searched for and loaded. If this file is not found, corresponding information will be missing and related commands may not deliver the complete results.

A.3 File Format

A header contains the name, identification number, and location of the register block of the MCU. The header is followed by a list of module descriptors. Each descriptor contains register definitions and (optionally) a memory map

specification. The register definitions can be grouped under a group name. Each register definition defines the name, address, and size of an I/O register. The memory map specification is used by the MEM command to display the configured memory of that module.

A.4 Description Using Extended Backus–Naur Form (EBNF)

The syntax of the register file is described here in EBNF.

```
MCUDescription          =  Header {Module}.
Header                  =  "MCU" McuName McuId RegBase
                           RegSize.
Module                  =  "MODULE" ModuleName {RegDef}
                           {GroupDef | MapDef}.
GroupDef                =  "GROUP" GroupName {RegDef}.
RegDef                  =  RegName RegOffset Size.
MapDef                  =  "MEMMAP" BlkName BaseMapDef
                           {MapSecifier}.
BaseMapDef              =  "BASE" Exp "SIZE" Exp
                           "ENABLED" Exp.
MapSpecifier            =  "SPECIFIER" [Label] Exp.
Exp                     =  CExpression | SwitchExpr.
SwitchExpr              =  CExpression ":" {CaseSpec}.
CaseSpec                =  "[" ConstValue ":"
                           (CExpression | StringDef) "]".
McuName                 =  StringDef.//name of the MCU
McuId                   =  ConstValue.//identification
                           number of the MCU
RegBase                 =  ConstValue.//base address of
                           the registers after reset
ModuleName              =  Name.//name of the module
GroupName               =  Name.//name of a group of
                           registers
RegName                 =  Name.//name of the register
RegOffset               =  ConstValue.//offset from the
                           register base address
Size                    =  ConstValue.//size of the
                           register in bits
BlkName                 =  Name.//name of the memory
                           block
Label                   =  StringDef.//name to be used to
                           label the specifier
CExpression             =  // expression defined in
                           ANSI-C that contains integers
ConstValue              =  // constant value as defined
                           in ANSI-C
Name                    =   // identifier as defined
                           in ANSI-C
StringDef               =  // any number of printable
                           characters in double quotes
```

[1] Evaluation of expressions are done with signed 32-bit arithmetic.
[2] Non-printable characters are interpreted as white spaces.

Example: This example describes a hypothetical MCU. It contains the modules ABC, SQIM, and FLASH. The SQIM module has two groups of registers, PORTS and CHIPSELECTS.

```
MCU "MY_MCU"  0x07A5  0xFFF000  0x1000
  MODULE ABC
    ABCMCR     0x700    16
    PORTABC    0x706    16
  MODULE SQIM
    SQIMCR     0xA00    16
    SYNCR      0xA04    16
    GROUP PORTS
      PORTA      0xA10     8
      PORTB      0xA11     8
    GROUP CHIPSELECTS
      CSPAR0     0xA44    16
      CSBARA     0xA60    16
      CSORA      0xA62    16
    MEMMAP CSA
      BASE (CSBARA & 0xFFF8) << 8
      SIZE CSBARA & 7 :
           [0:0x800] [1:0x2000] [2:0x4000]
          [3:0x10000] [4:0x20000] [5:0x40000]
           [6:0x80000] [7:0x80000]
         ENABLED (CSPAR0 & 3) >= 2
        SPECIFIER "ACCESS" (CSORA >> 11) & 3
:
                    [0:"None"][1:"Read"]
                    [2:"Write"][3:"Both"]
         SPECIFIER "BYTE" (CSORA >> 13) & 3 :
                    [0:"None"][1:"Lower"]
                    [2:"Upper"][3:"Both"]
         SPECIFIER (CSORA >> 4) & 3 :
                    [0:"None"][1:"Lower"]
                    [2:"Upper"][3:"Both"]
  MODULE FLASH
    FEEMCR     0x820    16
    FEEBAH     0x824    16
    FEEBAL     0x826    16
    MEMMAP FLASH
      BASE (FEEBAH << 16)
      SIZE  0x8000
      ENABLED (FEEMCR & 0x8000) == 0
<eof>
```

Register Description File

Appendix B. C Source-Level Debugging

B.1 Contents

B.2 Introduction

This appendix provides information on performing C source-level debugging with the the MCUez debugger. The C source-level debugging process is applicable only for applications that are compliant with the *ELF/DWARF 2.0* object format standard.

NOTE: *A license key is required to activate this feature. Contact HIWARE AG for information on CSLD pricing and how to obtain the license key.*

B.3 Source Component

The **Source** component window displays the source code of the program (application file). It enables the user to view, change, monitor, and control the current execution location in the program. The text displayed in the **Source** component window is chroma-coded. Language, keywords, comments, and strings are emphasized with different colors (respectively blue, green, red). A word is selected by double clicking it.

Select a section of code by holding the left mouse button and dragging the mouse over the appropriate code range. If the selected source code range matches the highlighted instruction set in the assembly component, both code groups are highlighted. Marks are displayed at all locations where breakpoints can be set. If execution has stopped, the current position is marked in the source component with the corresponding statement in the source component highlighted.

NOTE: *The text visible in the* **Source** *component window cannot be edited. The source component is a file viewer only.*

B.4 Procedure Component

The procedure component displays the list of procedure or function calls that were performed up to the moment the program was halted. This is the procedure chain, also known as the call chain. Entries in the procedure chain are displayed in reverse order from the last call (most recent on top) to the first call (on bottom).

Procedure parameter values and types can be displayed. The object information bar contains the source module and the address of the selected procedure.

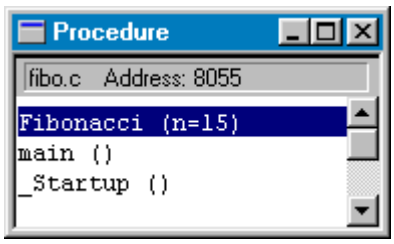


Figure B-1. Procedure Component Window

B.4.1 Operations

Double clicking on a procedure name forces all open windows to display information about that procedure. The **Source** component window shows the procedure's source. The **Data** component window displays the local variables and parameters of the selected procedure. Figure B-2 shows the **Procedure** component window menu.

- **Show Values** — Displays function parameter values in the procedure component
- **Show Types** — Displays function parameter types in the procedure component

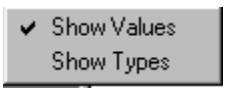


Figure B-2. Procedure Component Window Menu

B.4.2 Drag Out

Table B-1 shows the drag and drop actions possible from the procedure component.

Table B-1. Procedure Component Drag and Drop Actions

Destination Component	Action
Data | Local	Displays local variables from the selected procedure in the data component
Source	Displays source code of the selected procedure. Current instruction inside the procedure is highlighted in the source component.
Assembly	The current assembly statement inside the procedure is highlighted in the assembly component.

B.4.3 Drop Into

Nothing can be dropped into the **Procedure** component window.

B.5 Data Component

The **Data** component window contains the names, values, and types of global or local variables. The **Data** component window (in Figure B-3) shows all variables that are present in the current source module or procedure.

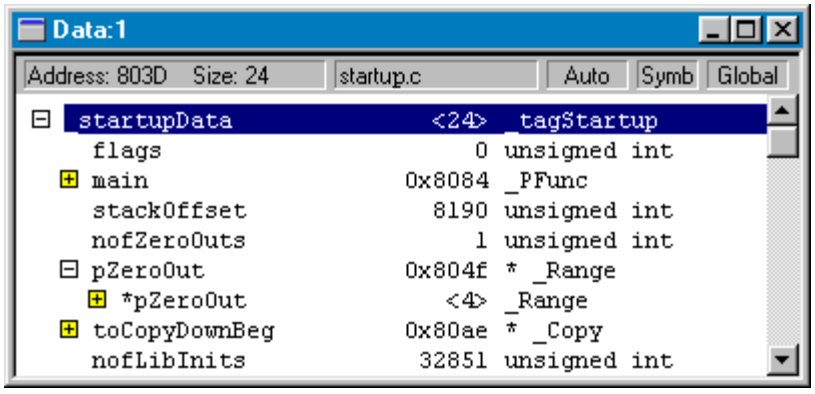


Figure B-3. Data Component Window

The object info bar contains the address and size of the selected variable. It also contains the module name or procedure name where the displayed variables are defined, the display mode (automatic, locked, etc.), the display format (symbolic, hex, bin, etc.), and the current scope (global or local).

Various display formats such as symbolic representation, hexadecimal, octal, binary, signed, and unsigned can be selected. Structures can be unfolded to display their member fields. Pointers can be traversed to display the data they are pointing to. **Table B-2** lists the menu options for the data component.

Table B-2. Data Component Menu Options for C Source-Level Debugging

Menu Entry	Description
Zoom in	Develops the selected structure. The member field of the structure replaces the variable list.
Zoom out	Returns to previous level of development
Scope...	Switches between local or global variable display
Format...	Switches between Symbolic (display depends on type of variable), Hex (hexadecimal), Oct (octal), Bin (binary), Dec (signed decimal), UDec (unsigned decimal) display format.
Mode...	Switches between automatic, periodical, locked, or frozen update mode

In automatic mode (default), variables are updated when the target is stopped. Variables from the currently executed module or procedure are displayed in the data component.

In locked and frozen mode, variables from a specific module are displayed in the data component. In that case, the same variables are always displayed in the data component.

In locked mode, variable values displayed in the data component are updated when the target is stopped.

In frozen mode, values displayed in the data component are not updated when the target is stopped.

In periodical mode, variables are updated at regular time intervals when the target is running. The default update rate is 1 second.

B.6 Breakpoints Setting Dialog

The **Breakpoints** setting dialog box consists of:

- A list box that displays currently defined breakpoints
- A **Breakpoint:** group box that displays the address of the currently selected breakpoint, name of the procedure in which the breakpoint has been set, state of the breakpoint (disabled or not), and type of breakpoint (temporary or permanent)
- A **Counter:** group box that displays the current value and interval value of the counter. This group allows the user to define a counting breakpoint. (See **B.10 Stopping an Application**.)
- A **Delete** button to remove the currently selected breakpoint
- **OK** button to validate all modifications
- **Cancel** button to disregard all modifications and retain previous values
- **Help** button to open the help file

See **Figure B-4**.

B.7 General Rule for Halting on a Control Point

Counting Control Point: If the interval is greater than one (1), a counting control point has been defined. When the application is running, its current value is decremented each time the control point is reached. The application will halt on this control point if the value is equal to zero (0). If a command has been defined and enabled, the command is executed when the application is halted on the control point.

B.8 Configuring the Default Layout

This line must be in the *project.ini* file to establish a default layout for the debugger:

```
Window2=Procedure   0  30  50  15
```

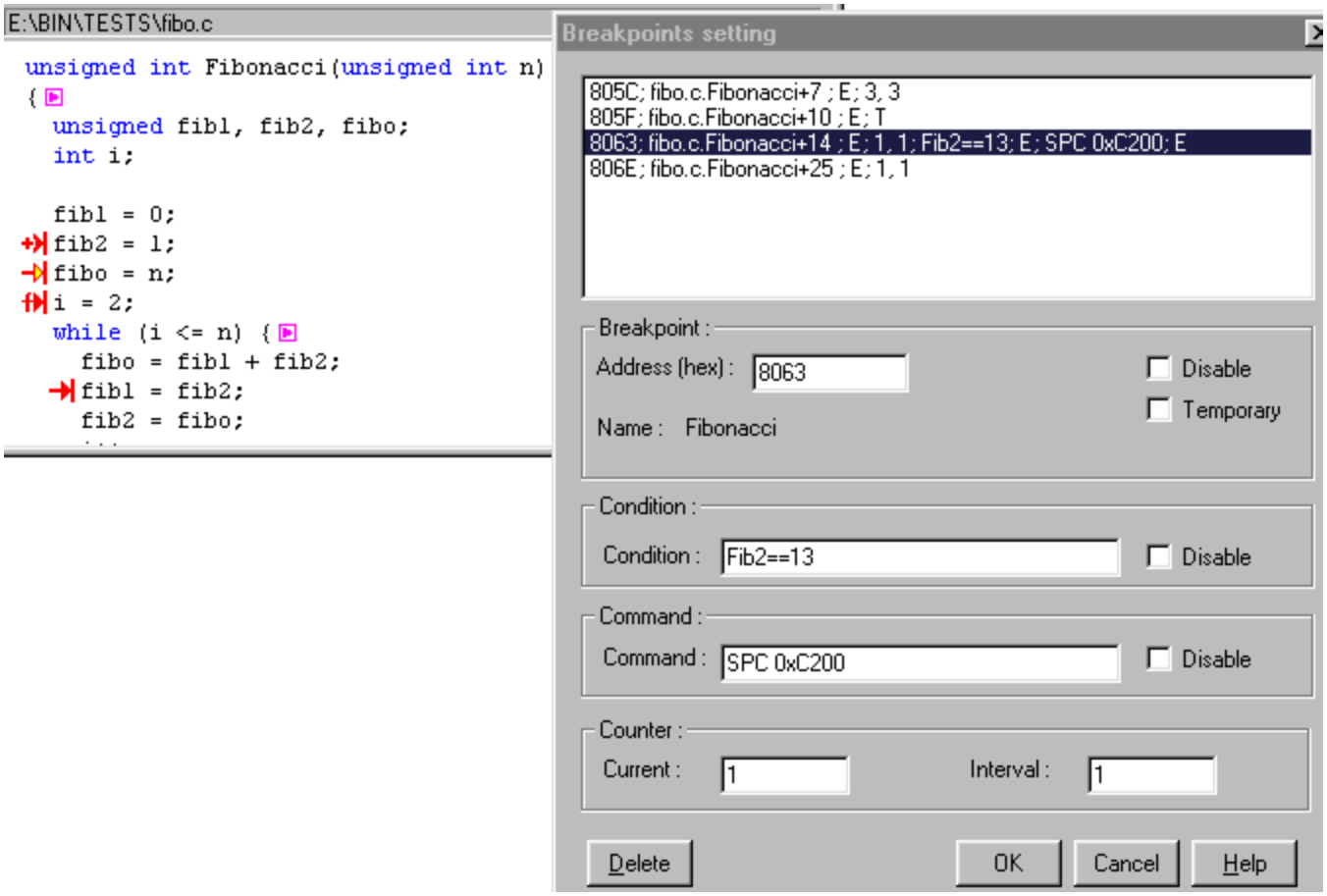


Figure B-4. Breakpoints Setting Dialog Window

B.9 Loading an Application

The global data component contains the list of global variables defined in the module that contains the application entry point. The local data component is empty.

B.10 Stopping an Application

The **Data** component window (assigned the global attribute) displays the name and value of all global variables defined in the module where the currently executed procedure is implemented. The module name is specified in the data component information bar. The **Data** component window (assigned the local attribute) displays the name and value of the local variables defined in the current procedure. The name of the procedure is specified in the data information bar.

B.11 Defining Counting Breakpoints

Counting breakpoints are activated after the instruction has been executed a specified number of times. This section describes breakpoint operations.

A counting breakpoint is recognized by this icon: .

A counting breakpoint is set by the **Breakpoints Setting** dialog. To access this dialog box:

- Point to a C statement in the source component window, hold the left mouse button, and press the S key.
- Point at a C statement in the **Source** component window and click the right mouse button to open the **Source** pop-up menu, then select **Set BreakPoint** or **Show BreakPoints**.
- Select **Run | Breakpoints ....** from the main menu bar.

If program execution continues, the **Current** field is decremented each time the instruction containing the counting breakpoint is reached. When **Current** is equal to 0, the application stops. If the check box **Temporary** is not checked (not a temporary breakpoint), **Current** is reloaded with the value stored in **Interval** to enable the counting breakpoint again.

B.12 Stepping in the Application

The MCUez debugger provides stepping functions at the application source level and assembler level.

B.12.1 Stepping at Source Level

Figure B-5 shows a typical stepping-at-source-level operation.

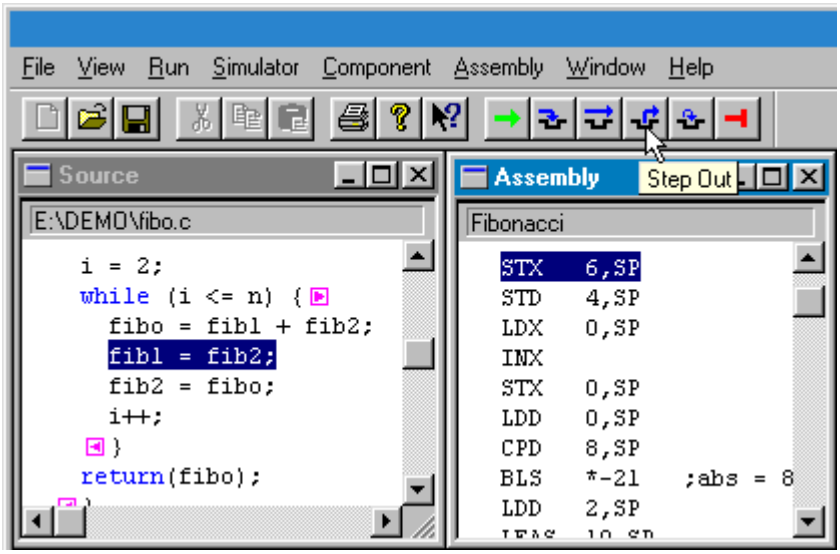


Figure B-5. Stepping-at-Source Level Window

The debugger provides two ways of stepping to the next source instruction:

1. Select **Run | Single Step**.
2. Click the **Single Step** icon on the debugger toolbar  .

STEPPED displayed in the status line indicates that the application is stopped by a step function.

If the application was previously stopped on a function invocation, a **Single Step** stops the application at the beginning of the invoked function.

The display in the assembly component is always synchronized with the display in the source component. The highlighted instruction in the assembly component is the first assembler instruction generated by the highlighted instruction in the source component.

Elements from the register, memory, or data components that are displayed in red are the register, memory position, and local or global variables. The indicated variables are those whose values have changed during execution of the source statement.

B.12.2 Stepping Over a Function Call (Flat Step)

The debugger provides two ways of stepping over a function call:

1. Select **Run | Step Over**.
2. Click the **Step Over** icon on the toolbar  .

STEPPED OVER displayed in the status line indicates that the application was stopped by a step over function.

If the application was previously stopped on a function invocation, a **Step Over** halts the application on the source instruction directly following the function invocation.

The display in the assembly component is always synchronized with the display in the source component. The highlighted instruction in the assembly component is the first assembler instruction generated by the highlighted instruction in the source component.

Elements from the register, memory or data components that are displayed in red are the register, memory position, and local or global variables. The indicated variables are those whose values have changed during execution of the source statement.

B.12.3 Stepping Out of a Function Call

The debugger provides two ways of stepping out of a function call:

1. Select **Run | Step Out**.
2. Click the **Step Out** icon on the toolbar  .

STOPPED displayed in the status line indicates that the application is stopped by a step out function.

If the application was previously stopped on a function invocation, a **Step Over** halts the application on the source instruction directly following the function invocation.

B.13 Displaying a Local Variable from a Function

The debugger provides two methods to view the local variable list defined in a function.

1. Using drag and drop — Drag a function name from the procedure component to a data component with attribute local.
2. Using double click — Double click a function name in the procedure component.

The data component (with attribute local that is neither frozen nor locked) displays the list of variables (with their values and type) defined in the selected function.

B.14 Miscellaneous C Source-Level Commands

This section describes all debugger commands associated with C source-level debugging.

SPROC

Short description:

Shows information associated with the specified procedure

Syntax:

```
SPROC level
```

Description:

In the data component, the SPROC command shows local variables of the corresponding procedure stack level.

In the source component, the SPROC command shows the corresponding module's source text, scrolls to the corresponding procedure, and highlights the statement that is in the procedure chain.

`level = 0` is the current procedure level. `level = 1` is the caller stack level and so on.

Data component example:

```
in>Data:2 < SPROC 0
```

This command displays the local variables defined in the caller function number 1 in the call chain.

Source component example:

```
in>Source < SPROC 1
```

ATTRIBUTES

Short description:

Sets the display inside a component window

In the Procedure Component

Syntax:

```
ATTRIBUTES list
```

Arguments:

```
list=command{,command}
```

command=`VALUES (ON|OFF)|TYPES (ON|OFF)`

Description:

The `ATTRIBUTES` command sets the display and state options of the procedure component window.

The `VALUES` and `TYPES` command `ON` or `OFF` indicates if the values or types should be displayed in the **Procedure** window. This command is applicable for the procedure component only when performing C source-level debugging.

Example:

```
Procedure < ATTRIBUTES VALUES ON,TYPES ON
```

Argument types and values are displayed in the **Procedure** component window.

In the Data Component

Syntax:

```
ATTRIBUTES list
```

Arguments:

```
list=command{,command})
  command=FORMAT(bin | oct | hex | signed
  | unsigned | symb) | MODE(automatic |periodical
  | locked | frozen) | SCOPE (global | local)
  | SPROC module | SMOD module | UPDATERATE rate
```

Description:

The `ATTRIBUTES` command sets the display and state options of the **Data** component window.

The `FORMAT` command indicates how variables will be represented. Display formats are binary, octal, hexadecimal, signed decimal, unsigned decimal, or symbolic.

The `SCOPE` command selects and displays global or local variables.

The `MODE` command selects the display mode of variables.

In automatic mode (default mode), variables are updated when the target is stopped. Variables from the currently executed module or procedure are displayed in the data component.

In locked and frozen mode, variables from a specific module are displayed in the data component.

In locked mode, values from variables are updated when the target is stopped.

In frozen mode, values are not updated when the target is stopped.

In periodical mode, variables are updated at regular time intervals when the target is running. The default update rate is 1 second.

The `UPDATERATE` command sets the update rate for the data component. This command is only relevant when the update mode for the data component is set to periodical.

The `SMOD` (show module) command displays global variables of the corresponding module.

The SPROC (show procedure) command displays local variables of the procedure.

Arguments:

hex	Sets format representation to hexadecimal
oct	Sets format to octal
bin	Sets format to binary
symb	Sets format as a symbol
signed	Displays value in signed decimal format
unsigned	Displays value in unsigned decimal format
periodical	Set data component to periodical update mode
locked	Set data component to locked update mode
frozen	Set data component to frozen update mode
automatic	Set data component to automatic update mode
module	Specified module
rate	Update rate in tenths of a second. Valid values for the rate are 0 to 600.

Equivalent operations:

ATTRIBUTES FORMAT	elect menu entry **Data | Format...**.
ATTRIBUTES MODE	Select menu entry **Data | Mode...**.
ATTRIBUTES SCOPE	Select menu entry **Data | Scope...**.
ATTRIBUTES SPROC	Drag and drop from procedure component to data component.
ATTRIBUTES SMOD	Drag and drop from module component to data component.
ATTRIBUTES UPDATERATE	Select menu entry **Data | Mode | Periodical**.

Example:

```
Data < ATTRIBUTES MODE FROZEN
```

In the data component, the mode for updating global variables is set to frozen. Variables are not refreshed when the application is halted.

Appendix C. MMEVS0508 Target

C.1 Contents

C.2 Overview

This appendix introduces the MMEVS0508 modular evaluation system for the MC68HC05 and MC68HC08 MCU Families.

The MMEVS0508 is an emulator system designed by Motorola for MCUs that use a CPU05 or CPU08 to provide emulation memory. MMEVS interface allows an external target system based on a Motorola MCU to download an executable program from the MCUez environment, execute it, and relay the results of the real target system behavior to the MCUez debugger.

The debugger fully supervises and monitors the target system's MCU. It also controls CPU functions such as read and write in internal/external memory (even when the CPU is running) and single-step/run/stop processes.

C.3 Loading the MMEVS0508 Target

From the debugger, load the MMEVS0508 target by selecting **Set Target...** from the **Component** menu (**Figure C-1**) and selecting ESL (emulator server library) from the list of targets.

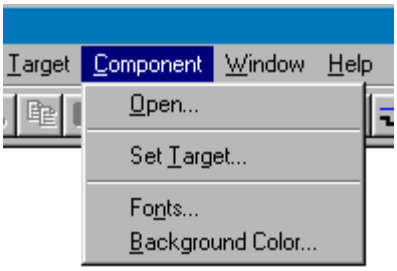


Figure C-1. Component Menu

The ESL driver automatically attempts to locate the MMEVS0508. If the target is successfully loaded, the **MMEVS0508** menu replaces the **Target** menu (**Figure C-2**) and the debugger status bar displays information shown in **Figure C-3**.

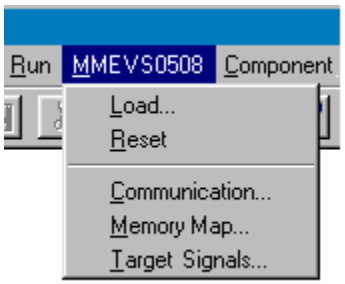


Figure C-2. MMEVS0508 Menu

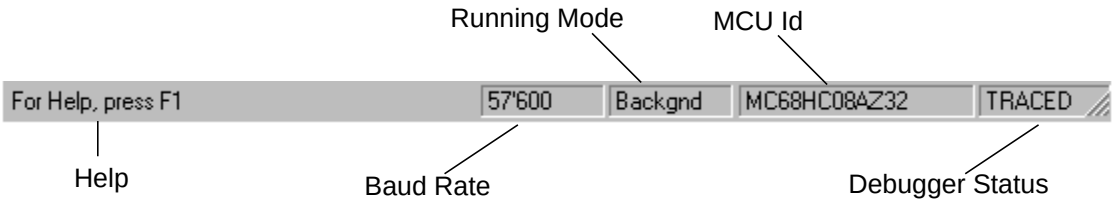


Figure C-3. Debugger Status Bar

Displayed from left to right after the help line are the communication baud rate, debugger run mode, name of connected MCU (MCU Id), and debugger status.

If the driver cannot locate the target, an error message and the **Communication** dialog box appear indicating that the target is not connected. This dialog also can be opened by selecting **Connect...** in the **ESL** menu.

C.3.1 MMEVS Target Startup File

The startup command file (*startup.cmd*) is executed by the debugger immediately after the MMEVS target driver has been loaded. This file must be located in the current project directory. Any debugger command can be used in this file. Commands are described in **Section 5. Command Reference**.

Example of a *startup.cmd* file:

```
wb 0x0035   0x00
wb 0x0012   0x11
```

C.4 MMEVS0508 Target Menu

The MMEVS0508 **Target** menu is displayed in **Figure C-4**.

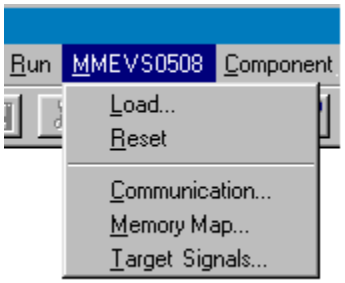


Figure C-4. MMEVS0508 Target Menu

C.4.1 Load...

Select **Load...** to load an executable file containing an *.abs* extension.

C.4.2 Reset

Select **Reset** to reset the target MCU. After selecting this option, the commands specified in the *reset.cmd* script file are executed. This file should be located in the current project directory. Any debugger command can be used in this file. Commands are described in **Section 5. Command Reference**.

C.4.3 Communication...

Select **Communication...** from the **MMEVS0508** menu to display the **Communication** dialog box. If the target is not connected, enter the port that connects the serial cable to the target. Select the maximum baud rate the host will support or 115,200.

The maximum baud rate depends on the speed and interrupt load of the host computer. The default is 9600.

Check the **Show Protocol** box to display the communication protocol in the command line component of the debugger. This feature should be used only for advanced debugging issues.

C.4.4 Memory Map...

Choose **Memory Map...** from the menu to open the **Memory Configuration** dialog (**Figure C-5**). This dialog displays the current memory map.

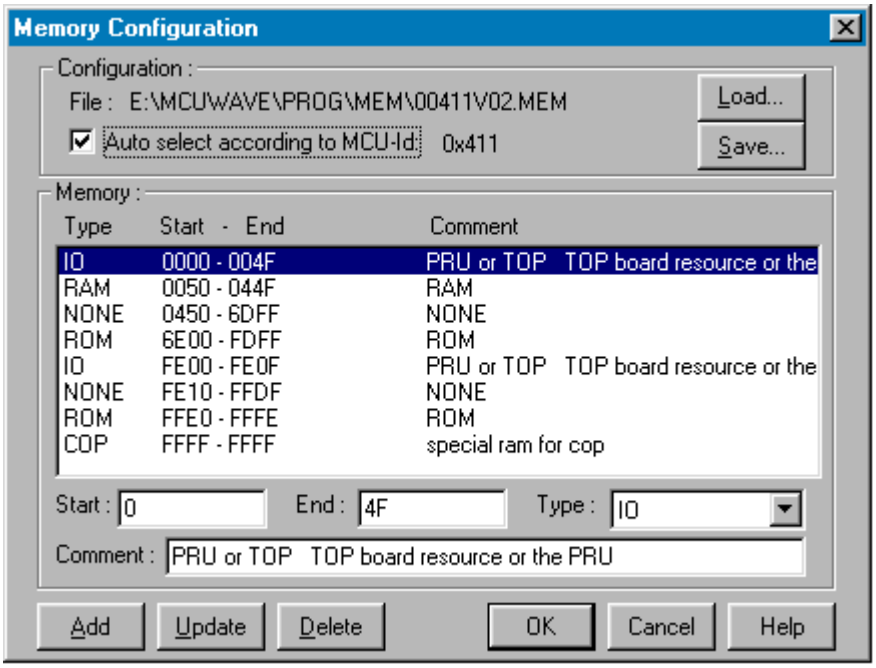


Figure C-5. Memory Configuration Dialog Box

C.4.4.1 Configuration

Auto select according to MCU-Id: Upon starting the debugger, the target memory setup is automatically loaded. The debugger loads the memory map file (*.mem*) related to the MCU Id. If not checked, the previously opened or saved memory map file is automatically loaded.

Load... Opens a dialog box (**Figure C-6**) to locate and load a different memory file. If a file is not found or is invalid, an error message is displayed (**Figure C-7**). Memory (personality) files can also be loaded with the `LOADMAP` command line command. To work properly, the ESL target must load the memory file that matches the connected emulation module.

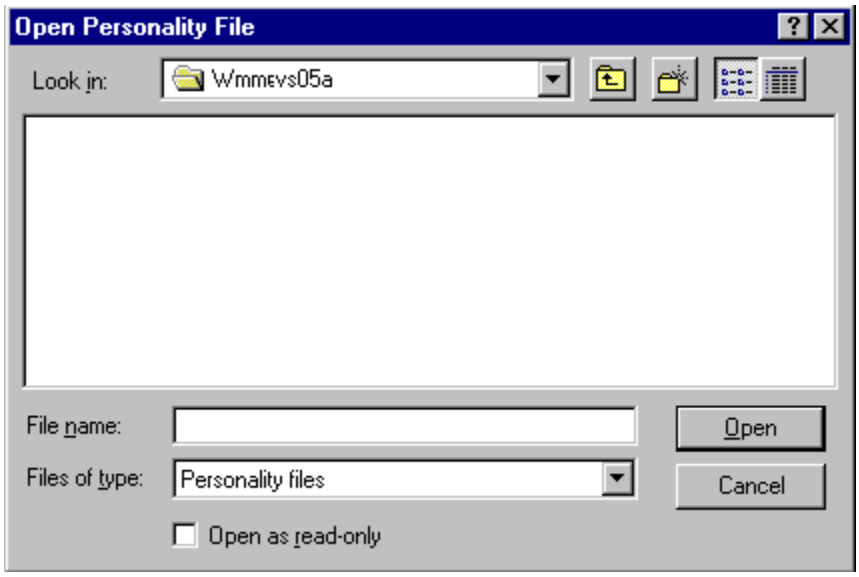


Figure C-6. Open Personality File Dialog Box

Figure C-7. Memory Map Error Message

The *.mem* filename is constructed like this: `0nnnnVxx.mem`
nnnn: MCU Id (4-digit hex value)
xx: 2-digit version number

Select **Retry** to open a memory file. If the selected file is valid, it is loaded and copied into the *\PROG\MEM* directory. Select **Cancel** to open the **Open Personality File** dialog box.

Save... Opens a dialog box to save new user-defined memory locations in a memory configuration file

WARNING: *Do not save a new configuration to a provided .mem file. This would overwrite the original .mem file and could prohibit the MMEVS from booting up.*

C.4.4.2 Memory

The memory group box in the **Memory Configuration** dialog allows the real-time memory to be specified. The user can specify the start and end location for a selected memory type and enter a brief comment. The MMEVS0508 real-time memory consists of dual-ported memory that can be assigned to any valid RAM or ROM memory address. While the MMEVS0508 is running, the debugger can display and modify the real-time memory. If part of the memory overlays the MCU I/O, RAM, or EEPROM, that portion can only be displayed, not monitored.

C.4.5 Target Signals

Select **Target Signals...** from the **MMEVS0508** menu to open the **Target Signals** dialog shown in **Figure C-8**. Settings are read from the MMEVS0508. If the user changes the settings and selects **OK**, the values are written to the MMEVS0508. If **Save and Reload** is checked, the configuration is reloaded when the debugger is started.

This dialog allows the MCU clock speed to be set and the target signal connection to be reset.

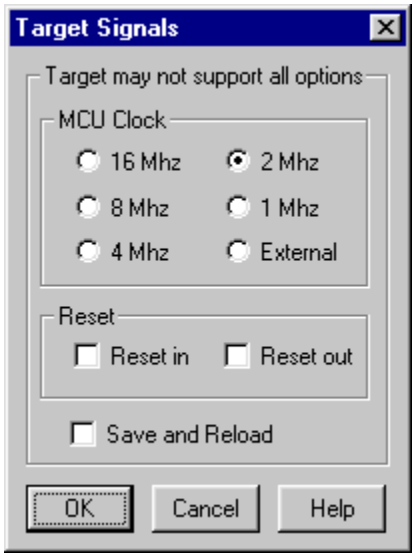


Figure C-8. Target Signals Dialog Box

C.5 MMEVS Commands

This section describes MMEVS commands. These commands can be typed in the debugger command line component or inserted in a command file.

BAUD — Serial Communication Baud Rate

Description:

The BAUD command sets the baud rate for communication between the system controller and host computer. For maximum performance, the baud rate should be set as high as the host computer can accommodate. The maximum rate is 115,200, the default is 9600.

If the system does not support a specified baud rate, an error message is displayed.

Syntax:

```
BAUD [rate]
```

Argument:

`rate`	1200, 2400, 4800, 9600, 19,200, 28,800, 38,400, 57,600, 115,200

Example:

```
BAUD 57600
```

Exit — Terminate Session

Description:

The EXIT command terminates the host session with the target.

Syntax:

```
EXIT
```

LOADMAP — Load Memory Map

Description:

The LOADMAP command allows the user to load a memory map from a file. If the specified file starts with a number, it should be preceded by a period and back slash (.\). For example, `LOADMAP .\00123V22.mem`.

Syntax:

```
LOADMAP <filename> | <MCU ID>
```

Arguments:

`filename`	Specifies a memory file that defines a memory map
`MCU ID`	MCU identifier. If specified, the command will load the memory file that matches the MCU Id.

Example:

`LOADMAP 0x3C1 /*`	Loads memory map from a file that matches the MCU Id */

OSC — Select Emulator Clock Frequency

Description:

The `OSC` command selects the emulator clock frequency. Six clock frequencies are available. Five internally generated clock frequencies are available: 16 MHz, 8 MHz, 4 MHz, 2 MHz, and 1 MHz. An external clock source is also available. The default emulator clock rate is set by the firmware and adapted to the current frequency. Enter this command without parameters to open the equivalent **Target Signals** dialog box.

Syntax:

```
OSC [<rate> | <source>]
```

Arguments:

`rate`	`OSC1MHZ`	Selects the 1 MHz oscillator
	`OSC2MHZ`	Selects the 2 MHz oscillator
	`OSC4MHZ`	Selects the 4 MHz oscillator
	`OSC8MHZ`	Selects the 8 MHz oscillator
	`OSC16MHZ`	Selects the 16 MHz oscillator
`source`	`EXT`	Selects an external clock source

Example:

```
OSC osc8Mhz   /* Use the 8 MHz internal emulator clock */
```

Disables the real-time memory block at base address $1000

PROTOCOL — Protocol Command

Description:

The PROTOCOL command allows the user to display the communication protocol between the debugger and the server in the **Command Line** window.

Syntax:

```
PROTOCOL [ON | OFF]
```

Arguments:

ON	Protocol is displayed in the **Command Line** window (default).
OFF	Protocol is not displayed in the **Command Line** window or logged in the log file.

RESET — Reset Command

Description:

The RESET command resets the target MCU. RESET GO performs a reset of the EM (emulator) and automatically starts execution of the code starting at the address stored in the reset vector. RESET STOP performs a reset and waits for user commands.

NOTE: *This command does not execute the reset command file (reset.cmd).*

Syntax:

```
reset [GO | STOP]
```

Arguments:

GO	Resets the MCU and continues at address stored in reset vector
STOP	Resets the MCU and stops (default)

Example:

```
RESET GO  /* Resets the MCU and continues */
RESET  /* Resets the MCU and stops (default) */
```

SIG — Target Signal Emulation Command

Description:

The SIG command enables or disables control signals from the target MCU and controls emulation. If a signal or list of signals is entered without specifying ENABLE or DISABLE, the signals are enabled. If no signal is specified, the **Target Signal** dialog box appears.

Syntax:

```
SIG [ [ENABLE] <signal> {<signal>}]
[ DISABLE <signal> {<signal>}]
```

Arguments:

signal	The signal to be enabled or disabled. Available signals are RESETIN and RESETOUT.
ENABLE	Connect the signal from the target system.
DISABLE	Disconnect signal from the target system.

Example:

```
SIG ENABLE RESETIN
```

Enables the RESET IN signal from the target system

Appendix D. MMDS0508 Target

D.1 Contents

D.2 Overview

MCUez debugger uses Motorola's MMDS0508 to communicate with a target system. The MMDS0508 is an emulator system that provides emulation memory and a bus state analyzer for MCUs with a CPU05 or CPU08.

Using the MMDS interface, an external target system can download an executable program, execute it, and relay the results of the target system to the debugger.

The debugger fully supervises and monitors the target system's MCU. It also controls CPU activities such as read and write in internal/external memory (even when the CPU is running) and single-step/run/stop processes.

D.3 Loading the MMDS0508 Target

From the MCUez debugger, load the MMDS0508 target by selecting **Set Target...** from the **Component** menu (**Figure D-1**) and selecting **ESL** from the list.

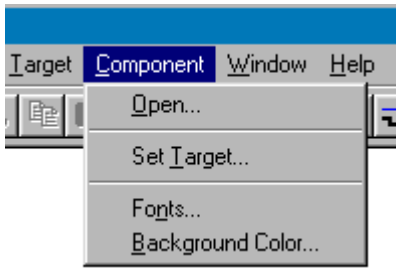


Figure D-1. Component Menu

The ESL driver automatically attempts to locate the MMDS0508. If the target is successfully loaded, the **MMDS0508** menu replaces the **Target** menu and the debugger status bar displays information shown in **Figure D-2**.

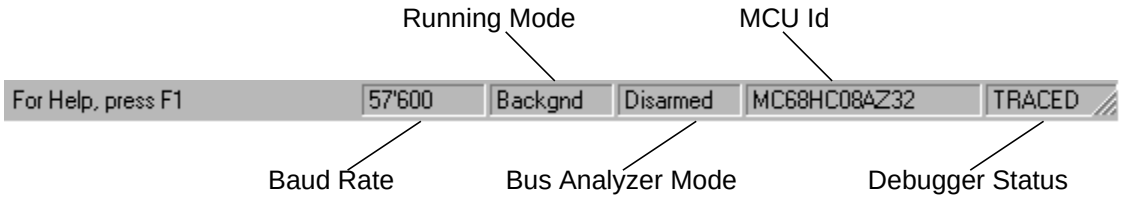


Figure D-2. MCUez Status Bar

Displayed from left to right after the help line are: the communication baud rate, debugger run mode, bus analyzer mode, name of connected MCU, and debugger status.

If the driver cannot locate the target, an error message and the **Communication** dialog box appear, indicating that the target is not connected. The **ESL** menu replaces the **Target** menu. The **Communicaton** dialog can also be opened by selecting **Connect...** in the **ESL** menu.

D.3.1 MMDS Target Startup File

The startup command file (*startup.cmd*) is executed by the debugger immediately after the MMDS target driver has been loaded. This file must be located in the working directory. Any debugger command can be used in this file. Commands are described in **Section 5. Command Reference**.

Example of a *startup.cmd* file:

```
wb 0x0035   0x00
wb 0x0012   0x11
```

D.4 MMDS0508 Target Menu

The **MMDS0508** target menu is displayed in **Figure D-3**.

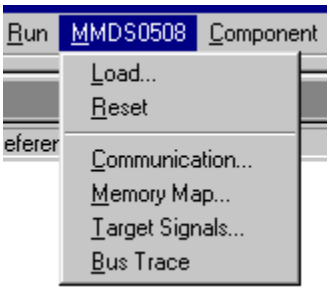


Figure D-3. MMDS0508 Target Menu

D.4.1 Load...

Select **Load...** to load an executable file containing an *.abs* extension.

D.4.2 Reset

Select **Reset** to reset the target MCU. After selecting this option, the commands specified in the *reset.cmd* script file are executed. This file should be located in the current project directory. Any debugger command can be used in this file. Commands are described in **Section 5. Command Reference**.

D.4.3 Communication...

Select **Communication...** to display the **Communication** dialog box. If the target is not connected, enter the port that connects the serial cable to the target. Select the maximum baud rate the host will support or 115,200.

The maximum baud rate depends on the speed and interrupt load of the host computer. The default is 9600.

Check the **Show Protocol** box to display the communication protocol in the command line component of the debugger. This feature should be used only for advanced debugging issues.

D.4.4 Memory Map...

Choose **Memory Map...** from the menu to open the **Memory Configuration** dialog window (**Figure D-4**). This dialog displays the current memory map.

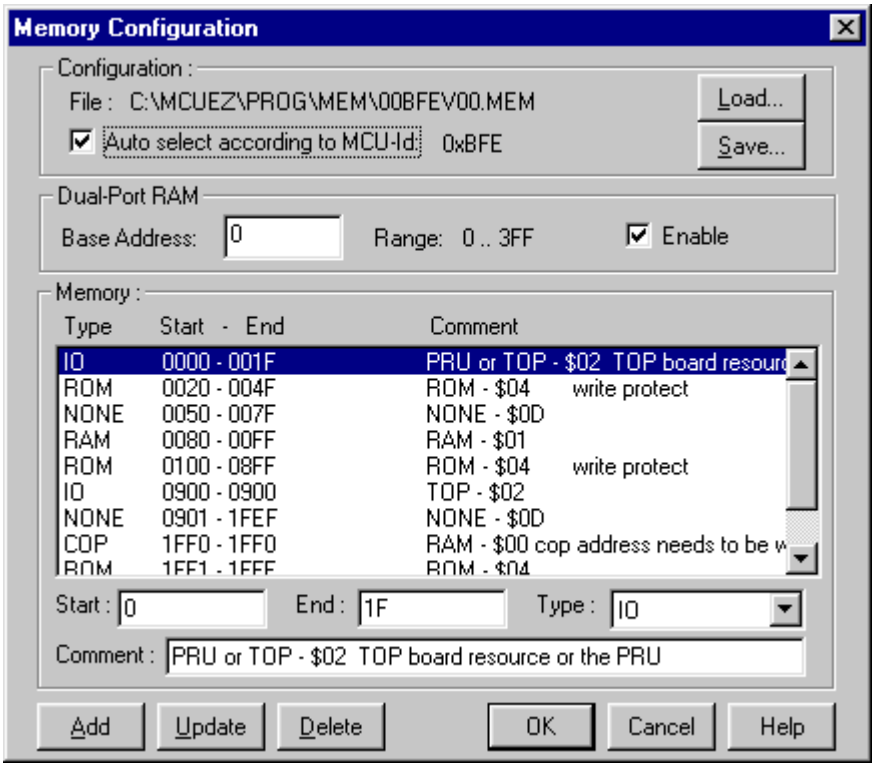


Figure D-4. Memory Configuration Dialog Box

D.4.4.1 Configuration

Auto select according to MCU-Id: Upon starting the debugger, the target memory setup is automatically loaded. The debugger loads the memory map file (*.mem*) related to the MCU Id. If not checked, the previously opened or saved memory map file is automatically loaded.

Load... Opens a dialog box (**Figure D-5**) to locate and load a different memory file. If a file is not found or is invalid, an error message is displayed (**Figure D-6**). Memory files (blocks) can also be loaded with the `LOADMAP` command line command. To work properly, the ESL target must load the memory file that matches the connected emulation module.

The *.mem* filename is constructed like this: `0`*`nnnn`*`V`*`xx`*`.mem`
nnnn: MCU Id (4-digit hex value)
xx: 2-digit version number

Select **Retry** to open a memory file. If the selected file is valid, it is loaded and copied into the *\PROG\MEM* directory. Select **Cancel** to open the **Open Personality File** dialog box.

Save... Opens a dialog box to save new user-defined memory locations in a memory configuration file.

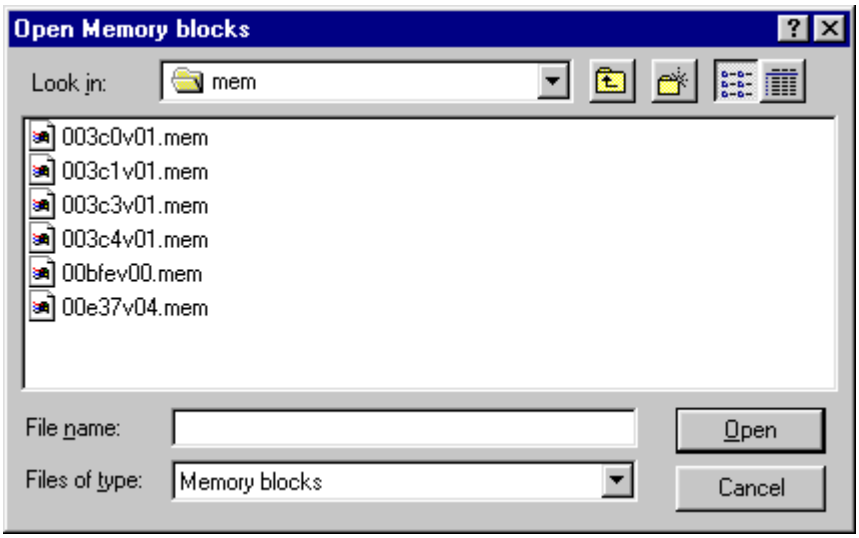


Figure D-5. Open Memory Blocks Dialog Box

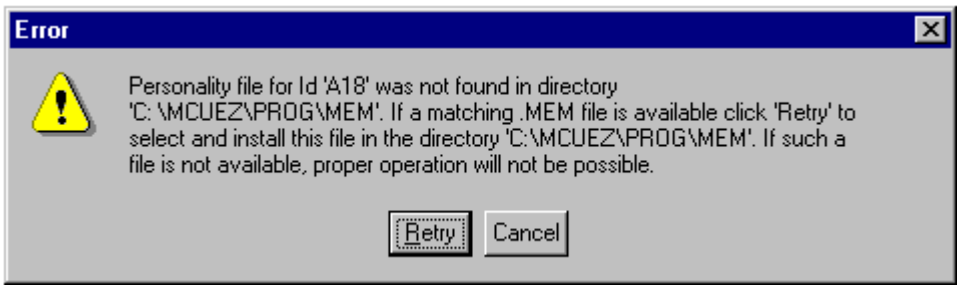


Figure D-6. Memory Map Error Message

WARNING: *Do not save a new configuration to a provided .mem file. This would overwrite the original .mem file and could prohibit the MMDS from booting up.*

D.4.4.2 Dual-Port RAM

Dual-port RAM allows the base address and enable real-time memory to be specified.

To specify the base address, enter the desired value in the **Base Address** edit box. To enable real-time memory, check **Enable**. The real-time memory size is 1 Kbyte.

NOTE: *The periodic update mode available in the debugger* **Data** *or* **Memory** *component menu is only possible for variables or memory positions located in the dual-port RAM area.*

D.4.4.3 Memory

The memory group box in the **Memory Configuration** dialog allows the real-time memory to be specified. The user can specify the start and end location for a selected memory type and enter a brief comment. The real-time memory consists of dual-ported memory that can be assigned to any valid RAM or ROM memory address. While the MMDS0508 is running, the debugger can display and modify the real-time memory. If part of the memory overlays the MCU I/O, RAM, or EEPROM, that portion can only be displayed, not monitored.

D.4.5 Target Signals...

Select **Target Signals...** from the **MMDS0508** menu to open the **Target Signals** dialog shown in **Figure D-7**. Settings are read from the MMDS0508. Change settings and select **OK** to write values to the MMDS. Check **Save and Reload** to reload the configuration when the debugger is started.

This dialog allows the MCU clock speed to be set and the target signal connection to be reset.

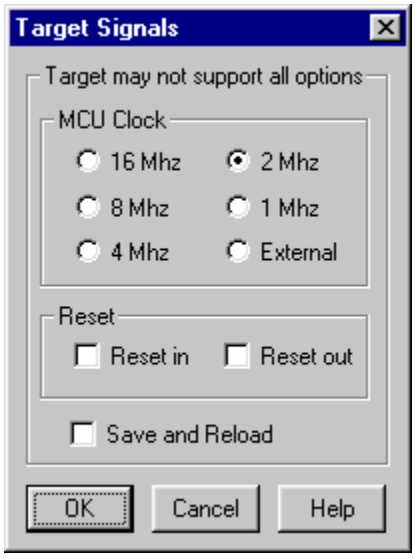


Figure D-7. Target Signals Dialog Box

D.4.6 Bus Trace

Select **Bus Trace** from the **MMDS0508** menu to run the bus analyzer. This will open the **Trace** component window and insert the **Trace** menu between the **Component** and **Window** menus.

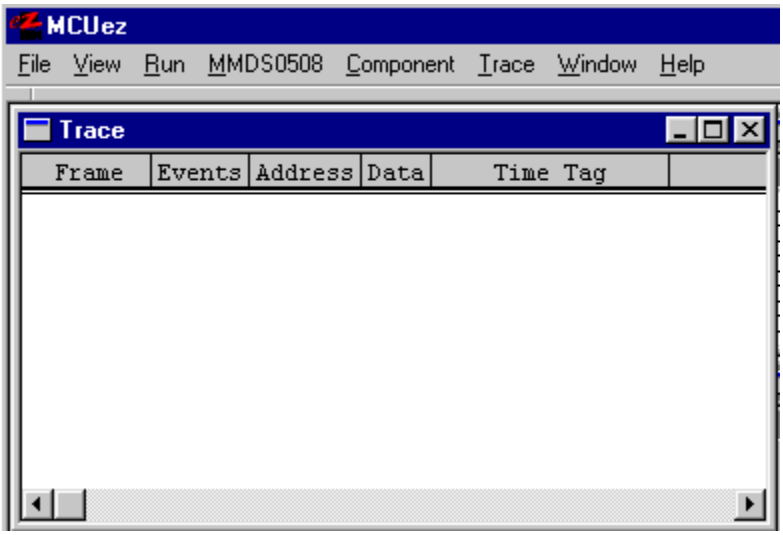


Figure D-8. Trace Component and Menu

D.5 Bus Analyzer

Except for emulation of the target system MCU, the most important feature offered by a microcontroller development tool is an instrument to analyze program execution activities on the target MCU bus. This analysis allows the user to determine what is occurring in a system without actually affecting it.

NOTE: *The bus analyzer in the MMDS0508 shows the logical state of the MCU bus. It does not show signal hold and setup times.*

D.5.1 Trace Buffer

The trace buffer contains 8,192 entries, or frames, each of which stores 96 bits. When the bus analyzer is activated and the emulator is running, a frame of the selected type is strobed into the trace buffer for each bus cycle. When the event(s) to trigger the analyzer has occurred in the specified sequence, only the specified number of additional frames are stored.

D.6 Using the Bus Analyzer

The bus analyzer **Trace** component window has its own menu to control the features of the MMDS0508 bus analyzer hardware. The bus analyzer window is called **Trace**. The **Trace** component window (**Figure D-9**) can be loaded by selecting **Open...** in the **Component** menu and selecting **Trace** from the list or by selecting **Bus Trace** in the **MMDS0508** menu.

From the user's perspective, using the bus analyzer requires three steps:

1. Defining the data collection parameters
2. Collecting the desired bus data (running the program)
3. Viewing the collected data

The bus analyzer allows these parameters to be defined:

- Triggers
- Pattern
- Sequencer
- Time tag clock

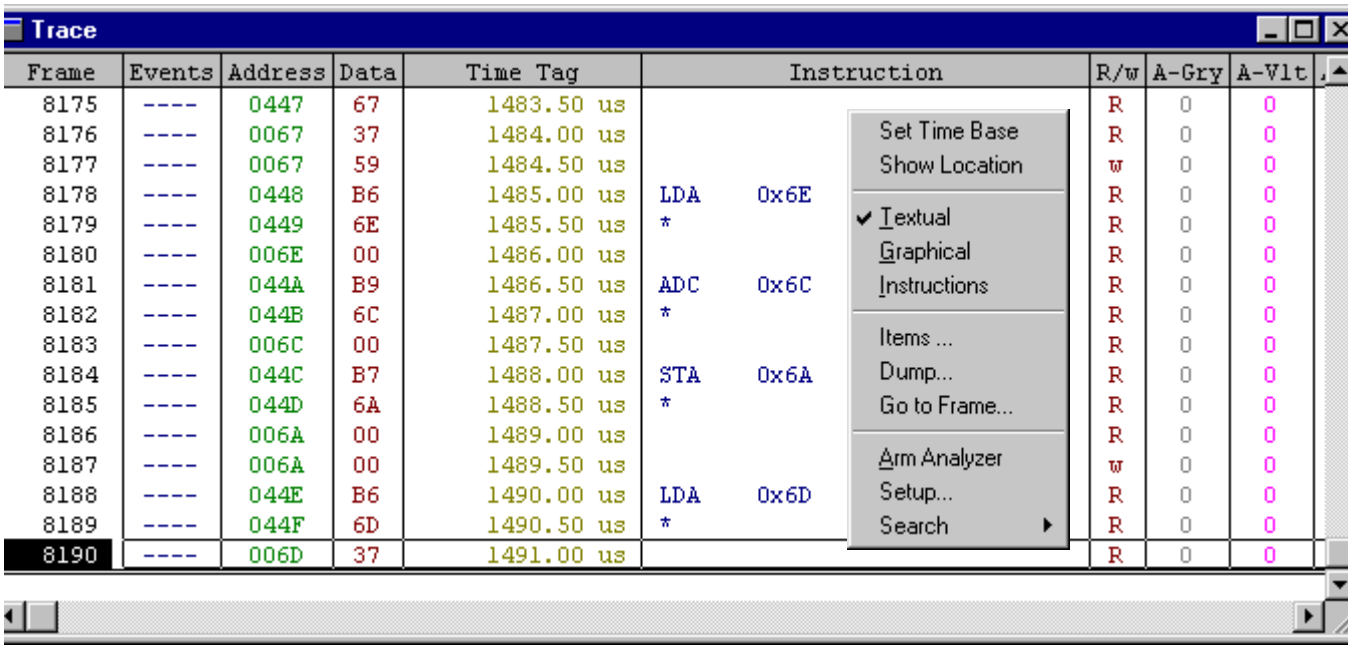


Figure D-9. Trace Component Window and Pop-up Menu

D.6.1 Trigger Setup

To set up the bus analyzer, select **Trace | Setup...** to display the **Bus Analyzer Configuration** dialog box (**Figure D-10**). This dialog box allows the user to define symbolic names for address values. The address and symbolic name values must match. If they are inconsistent, the user will be prompted to:

- Use the address and remove the symbol
- Replace the address with the symbol address
- Fix the inconsistency by returning to the dialog box and clicking the symbol button

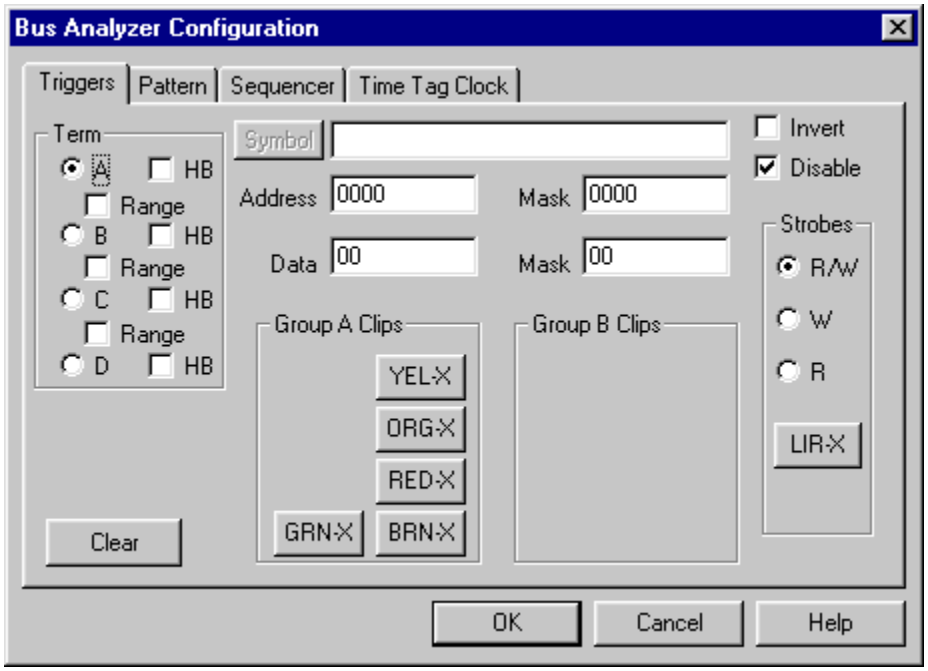


Figure D-10. Bus Analyzer Configuration Dialog Box

In the **Triggers** tab, define a trigger in terms of one to four events. For each term, the user can define the read-write actions and clips.

The user can use these terms as ranged or non-ranged triggers. When using ranges, the user can use A and B together or C and D together to give two different ranges or B and C together for one range.

The address masks and data masks can be edited to provide a don't care on any address or data signal. For example, if a trigger is set up for an address of 0x1000, but the corresponding mask is 0xFFFE, then the trigger will be qualified when the address is 0x1000 or 0x1001.

Terms	Specify the term to be displayed for editing. The term is used to identify an event (a frame which satisfies the term is called an event).
Address	Specify the trigger address.
Data	Specify the trigger data.
Strobes	Specify the state of R/W on which to trigger.
Group A Clips	Toggle to specify each logic clip as high (H), low (L) or don't care (X).

The clips buttons show the Group A logic clips with their respective colors.

Logic clips are used to trace signals in the target system. When a trigger occurs, a breakpoint is provided that shows the states of significant logic signals before, at, and after the breakpoint.

Invert	Check **Invert** to specify that a term, previously defined as triggered within a range, is to be triggered outside that range.
Disable	Check **Disable** to disable the trigger for a specific term.
Clear	Select the **Clear** button to clear all changes.
HB	**Hardware Breakpoints** make use of the bus analyzer triggering circuit. Users can break on any combination of bus signals, including address and data values, instruction fetch, read/write, and logic clips. Four independant hardware breakpoints are available (Terms A, B, C, or D).

D.6.2 Pattern Search Setup

To search for a frame with a specific pattern, define the search pattern to find all matched frames in the trace buffer. The pattern consists of an address, a data word, logic clips, and four miscellaneous signals.

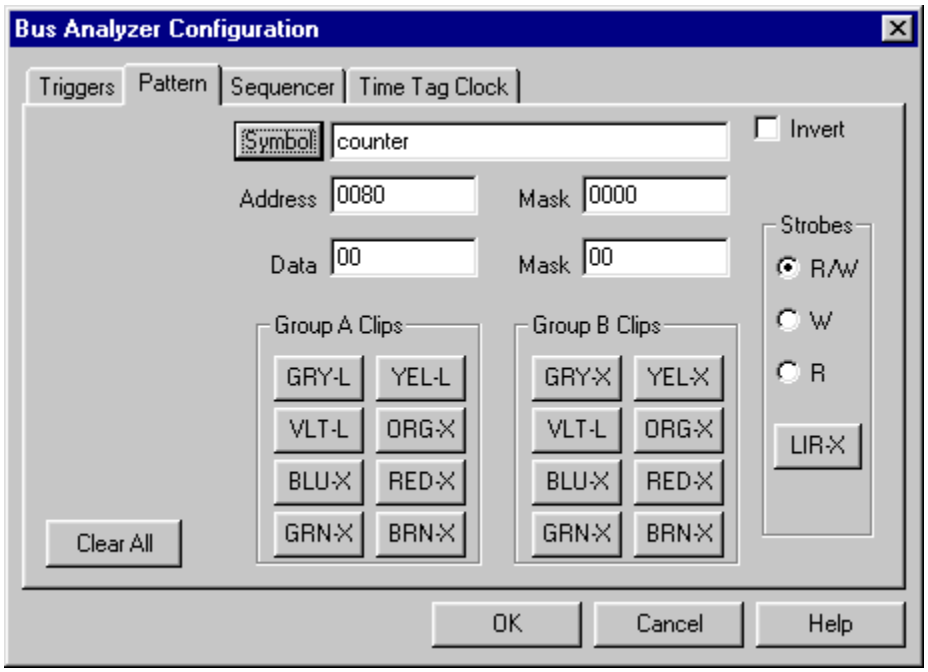


Figure D-11. Search for Pattern

In the dialog box shown in **Figure D-11**, the search pattern can be defined the same as it is defined in the **Trigger** tab of the **Bus Analyzer Configuration** dialog box.

The bus analyzer will search for the defined pattern in the trace buffer, then highlight and display the matched frame in the **Trace** window.

Address	Specify the **Address** to be matched.
Data	Specify the **Data** value to be matched.
Strobes	Specify the state of R/W and LIR-X to be matched.
Group A/B Clips	Toggle to specify each logic clip as high (H), low (L), or don't care (X).

The clips buttons show Group A and B logic clips with their respective colors.

Invert	Select **Invert** to find any frame that does not match the defined pattern.
OK Button	Click **OK** to store the specified values.
Cancel Button	Click **Cancel** to retain previous settings.

D.6.3 Sequencer Setup

The bus analyzer can operate in different modes to gather pertinent bus data. The various trace modes of the bus analyzer make it possible to choose which actions to take when a certain pattern (event) or sequence of patterns appear on the bus. To trigger the bus analyzer, define the desired bus state(s) as terms and select the desired sequence of terms as a trigger event.

To select a recording mode, click on the **Sequencer** tab in the **Bus Analyzer Configuration** dialog box.

In continuous and counted modes (non-triggered modes), collection does not stop until the analyzer is terminated. The analyzer stops when the application stops or the bus analyzer is disarmed.

In sequential (triggered) modes, the defined terms are used to track the occurrence of events. Data collection is stopped based on certain combination of events. Each term has an associated pre-event count that will count events for that term. The sequencer condition will use that term when the count is reached.

An event is a pattern of bus signals (which can include addresses and data values) to which the analyzer is connected by logic clips and miscellaneous MCU signals. An event can also be the negation of a defined pattern. Each signal can be defined as asserted, negated, or ignored (don't care).

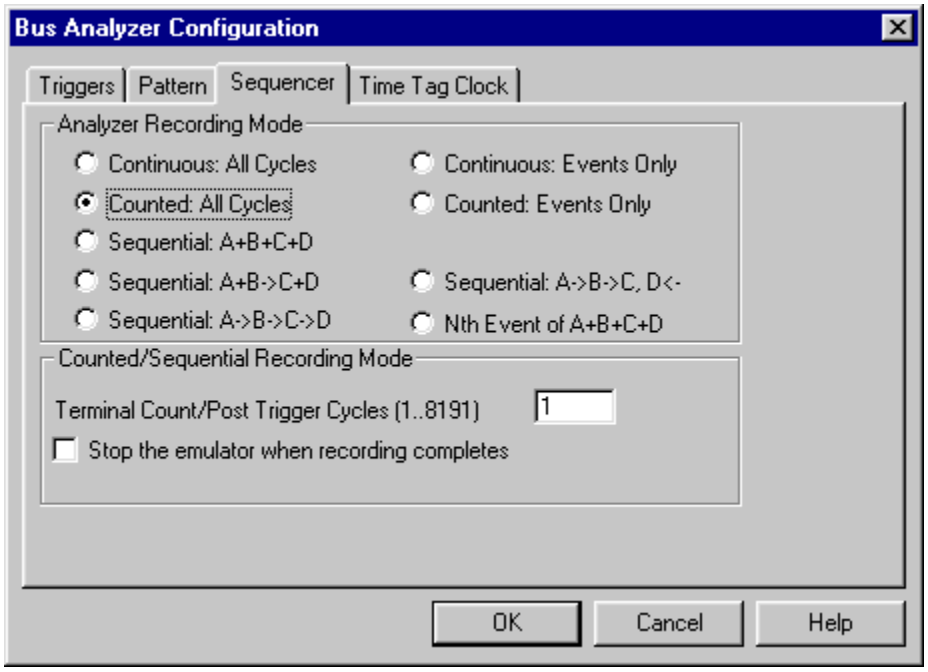


Figure D-12. Bus Analyzer Configuration (Sequencer Tab)

Click on one of the nine option buttons to select the recording mode.

D.6.3.1 Continuous (Non-Triggered) Modes

Continuous: All Cycles

Provides a real-time, non-invasive trace of MCU bus activities. The bus analyzer stores all cycles. When used in this way, the bus analyzer continuously records bus data in the trace buffer whenever the user target system is being emulated. No qualifications for triggering or halting data collection can be defined.

NOTE: *The user must stop the application to view the collected frames.*

Continuous: Events Only

Stores all events. Events are defined by the terms set up in the **Triggers** tab.

D.6.3.2 Counted (Non-Triggered) Modes

Counted: All Cycles

Configures the bus analyzer to record a specified number of cycles

Counted: Events Only

Collects all events until the specified count is reached, then collection stops

D.6.3.3 Sequential (Triggered) Modes

Sequential: A + B + C + D

Select this option for the bus analyzer to start recording after event A, B, C, or D. Frame collection will terminate after the specified number of post-trigger cycles.

Sequential: A + B -> C + D

Select this option to start the bus analyzer on event A or B followed by events C or D. This sequencer can be simplified to involve fewer than four events by defining all signals of unused events as ignored. Frame collection ends after the specified number of post-trigger cycles.

Sequential: A -> B -> C -> D

Select this option to start the bus analyzer after four events: A then B then C then D, occurring in sequence. Frame collection ends after the specified number of post-trigger cycles.

Sequential: A -> B -> C, D<-

Select this option to start the bus analyzer on cycles of three events in sequence, A then B then C, provided that the fourth event (D) remains false. When the fourth event occurs, the sequence starts again with the first event. This sequence can be used as a 3-event sequence by leaving event D defined with all signals ignored. Frame collection ends after the specified number of post-trigger cycles.

Nth Event of A+B+C+D

Select this option for the bus analyzer to begin storing data that matches events A, B, C, and D, until nth event is stored. Then the next 4096 cycles are stored. This allows for a maximum of 4096 events to be stored (including the nth event), followed by 4096 cycles.

For all sequential modes, data storage ends after the specified number of post-trigger cycles.

D.6.3.4 Counted/Sequential Recording Mode

Terminal Count/Post Trigger Cycles (1..8191)

Enter a number in the range of 1 to 8191. If one of the counted modes was selected, this number represents the number of bus cycles to trace. If one of the sequential modes was selected, this is the number of cycles to trace after the trigger sequence occurs. If one of the continuous recording modes was selected, this value is ignored.

Stop the emulator when recording completes

Check this box to stop the emulator after the collection of bus cycles has stopped.

D.6.4 Time Tag Clock Setup

The bus analyzer uses a clock for the time tag, which provides a time reference value (timestamp) in each frame of the trace buffer. To change the clock, select the **Time Tag Clock** tab in the **Bus Analyzer Configuration** dialog box.

Time Tag Clock Frequency

Select one clock frequency to be used, either an internal oscillator at 1, 2, 4, 8, or 16 MHz, a bus clock, or a programmable clock. In general, faster clock rates provide higher resolution and are appropriate for faster emulator clock rates.

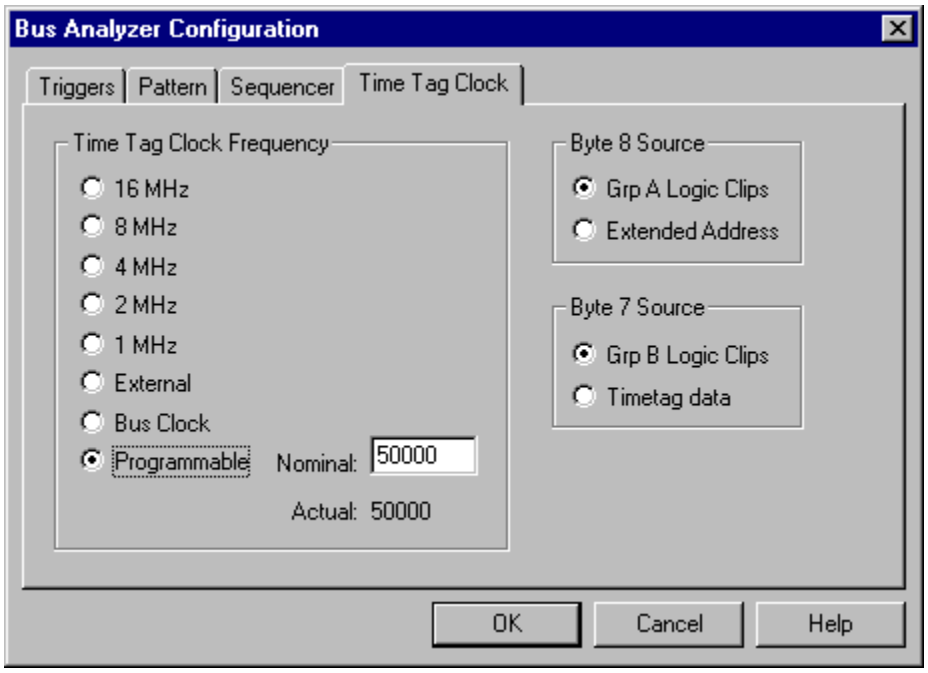


Figure D-13. Bus Analyzer Configuration (Time Tag Clock)

The programmable clock has to be programmed in a range of 50 to 50,000 Hz. Entering a nominal value causes the closest actual value to be calculated and used. If the nominal value is an integer that is a quotient of dividing 500,000 Hz by an integer, the nominal value is also the actual value. If this is not the case, the nominal value is rounded up to the next valid frequency and that frequency is displayed as the actual value.

Since the MMDS0508 supports the HC08 (up to 16 M of data space), the trace buffer will store the additional addresses (XA16-23). To allow for the additional storage, the logic clips on group A can be multiplexed with the expanded addresses. Currently, there is no EM (emulation) board supporting extended address mode.

The bus analyzer can also collect data from two groups of logic clips, Group A and Group B. Time tag can also be expanded to a 24-bit counter to allow for more time measurement.

Click the **OK** button to accept changes and close the dialog box.

D.7 Collecting Data Frame

When the emulator and bus analyzer have been initialized and the logic clips in the defined events that are used (if any) have been connected, emulation can begin.

D.7.1 Arming the Analyzer

The bus analyzer has to be armed to collect data. Select the menu entry **Trace | Arm Analyzer** to arm the bus analyzer.

D.7.2 Disarming the Analyzer

To stop the analyzer, select the menu entry **Trace | Disarm Analyzer**. Disarming the analyzer stops data collection without stopping emulation.

D.7.3 Start Emulation

To begin emulation, select the menu entry **Run | Start/Continue** in the **MCUez Debugger** main menu or click the **Start/Continue** button. Emulation continues until stopped by either a bus analyzer breakpoint or manually. When emulation stops, data in the **Bus Analyzer** window is updated.

D.7.4 Status Bar

When the bus analyzer is activated, the status bar displays **Armed**. The bus analyzer is ready to collect data. When emulation begins, the status bar displays **Running**. When the bus analyzer collects data, the status bar displays **Analyzing**. When data collection has stopped, the bus analyzer state changes to **Disarmed**.

D.7.5 Recording Bus Data

When data collection begins, the bus analyzer records bus data into the buffer as frames. When the end of the buffer is reached, the bus analyzer wraps around to the first frame in the buffer and continues recording. This process continues

until the bus analyzer is manually disarmed, the specified number of frames has been recorded, or the specified number of post-trigger cycles following the trigger event has been recorded.

D.7.6 Trigger Event

When a trigger event is detected, the event cycle is latched into the buffer and the bus analyzer continues recording data until the specified number of post-trigger cycles has been collected. When the required post-trigger cycles have been collected, the bus analyzer stops collecting data.

With the first post-trigger cycle, the bus analyzer automatically begins searching for the next trigger event.

If other events occur while collecting the post-trigger cycles for the first event, the bus analyzer marks those event cycles while continuing to collect post-trigger cycles.

D.8 Viewing Collected Data

D.8.1 View Cycles

When the desired cycles have been collected, the bus analyzer provides a variety of methods to view those cycles. At this point, the trace buffer contains up to 8192 of the most recently stored frames. The higher-numbered frames are usually the post-trigger frames. The lower-numbered frames are those frames stored before the trigger occurred, if any were stored.

When the bus analyzer is deactivated, data is displayed in the **Trace** window.

D.8.2 Textual, Graphical, or Instructions

Select one of these entries in the **Trace** pop-up menu to set the bus data display.

NOTE: *If the sequencer is set up to not collect all frames (for example, event-only modes), the instructions may not be displayed.*

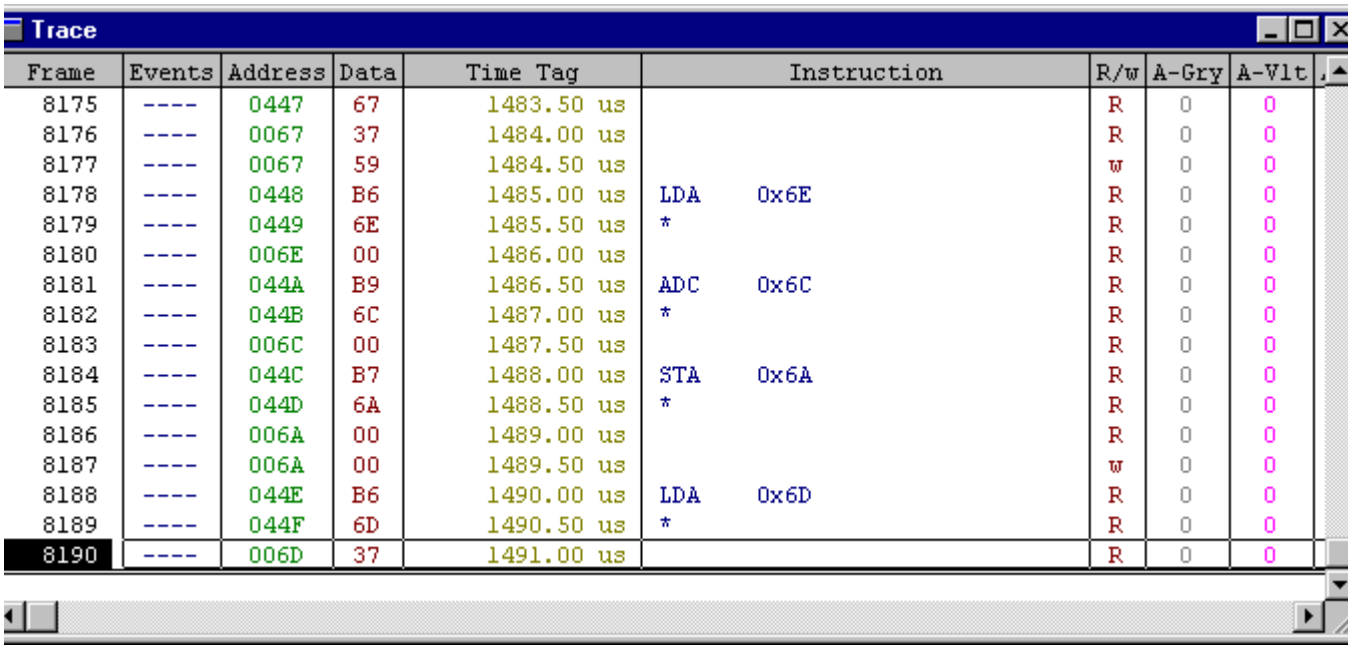
The bus analyzer can display data either textually or graphically. The user can also specify which items to be displayed by selecting the **Trace | Items...** menu entry.

D.8.3 Textual Display

In the textual representation, all frames or just the frames where an instruction starts can be displayed. Use the right side scroll bar to display other frames. Use the bottom scroll bar to display other signals.

Two horizontal lines (double bar) are used to mark a specific frame. The frame number of the marked frame is inverted.

The **Trace** window contains the following items. It is possible to add or remove any item by selecting the **Trace | Items...** menu entry.



Frame	Events	Address	Data	Time Tag	Instruction	R/w	A-Gry	A-Vlt
8175	----	0447	67	1483.50 us		R	0	0
8176	----	0067	37	1484.00 us		R	0	0
8177	----	0067	59	1484.50 us		w	0	0
8178	----	0448	B6	1485.00 us	LDA 0x6E	R	0	0
8179	----	0449	6E	1485.50 us	*	R	0	0
8180	----	006E	00	1486.00 us		R	0	0
8181	----	044A	B9	1486.50 us	ADC 0x6C	R	0	0
8182	----	044B	6C	1487.00 us	*	R	0	0
8183	----	006C	00	1487.50 us		R	0	0
8184	----	044C	B7	1488.00 us	STA 0x6A	R	0	0
8185	----	044D	6A	1488.50 us	*	R	0	0
8186	----	006A	00	1489.00 us		R	0	0
8187	----	006A	00	1489.50 us		w	0	0
8188	----	044E	B6	1490.00 us	LDA 0x6D	R	0	0
8189	----	044F	6D	1490.50 us	*	R	0	0
8190	----	006D	37	1491.00 us		R	0	0

Figure D-14. Trace Window Elements

Frame	The cycle or frame number is 0–8191, which identifies the frame. The most recently stored frame is frame 8191 (or the highest-numbered frame that has been stored when the trace buffer is full).
Events	The defined trigger identifier is A, B, C, or D. When the corresponding data in the frame matches the data defined for an event, the identifier of that event is displayed in the event column.
Address	The address bus value is stored in the frame, displayed as four hexadecimal digits. This is the address on the address bus when the frame is strobed into the trace buffer.
Data	The data bus value is stored in the frame, displayed as two hexadecimal digits. This is the value on the data bus when the frame is strobed into the trace buffer.
Time Tag	Contains a representation of the time tag count stored when the frame is strobed into the trace buffer. When the bus clock is the time tag clock, the time tag is the number of time tag clock cycles. When a clock other than the bus clock is chosen, the time tag is displayed as a number of seconds or fractions of second.
Control Signals	The remaining fields contain values of the control signals or the two groups of logic clips.

D.8.4 Instructions Display

If the instructions format is chosen, only instructions are displayed in the trace buffer. The instructions format is not possible in the events only recording mode.

D.8.5 Graphical Display

Figure D-15 and Figure D-16 show examples of graphically displaying bus analyzer data. A graphical representation gives a better overview than the textual display. The leftmost section presents a textual description of the current frame with information about frame number, events, values on the data and address bus, time tag value, etc.

In the graphical display, it is possible to zoom in or zoom out to see more detail or get a better general view. **Zoom in / Zoom out** are available in the **Trace** pop-up menu by clicking the right mouse button inside the **Trace** component window.

To zoom in, select **Trace | Zoom In** or press I on the keyboard.

To zoom out, select **Trace | Zoom Out** or press O on the keyboard.

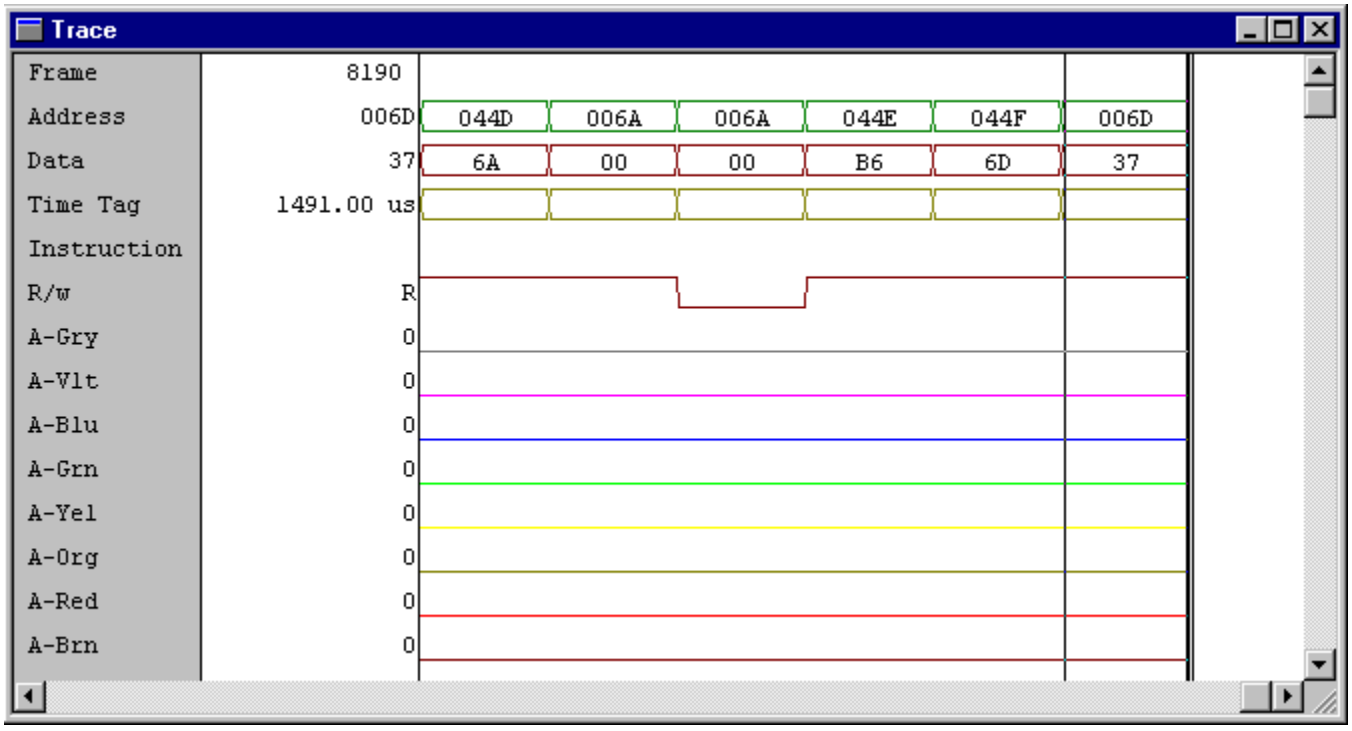


Figure D-15. Trace Window Graphical Display (Zoom In)

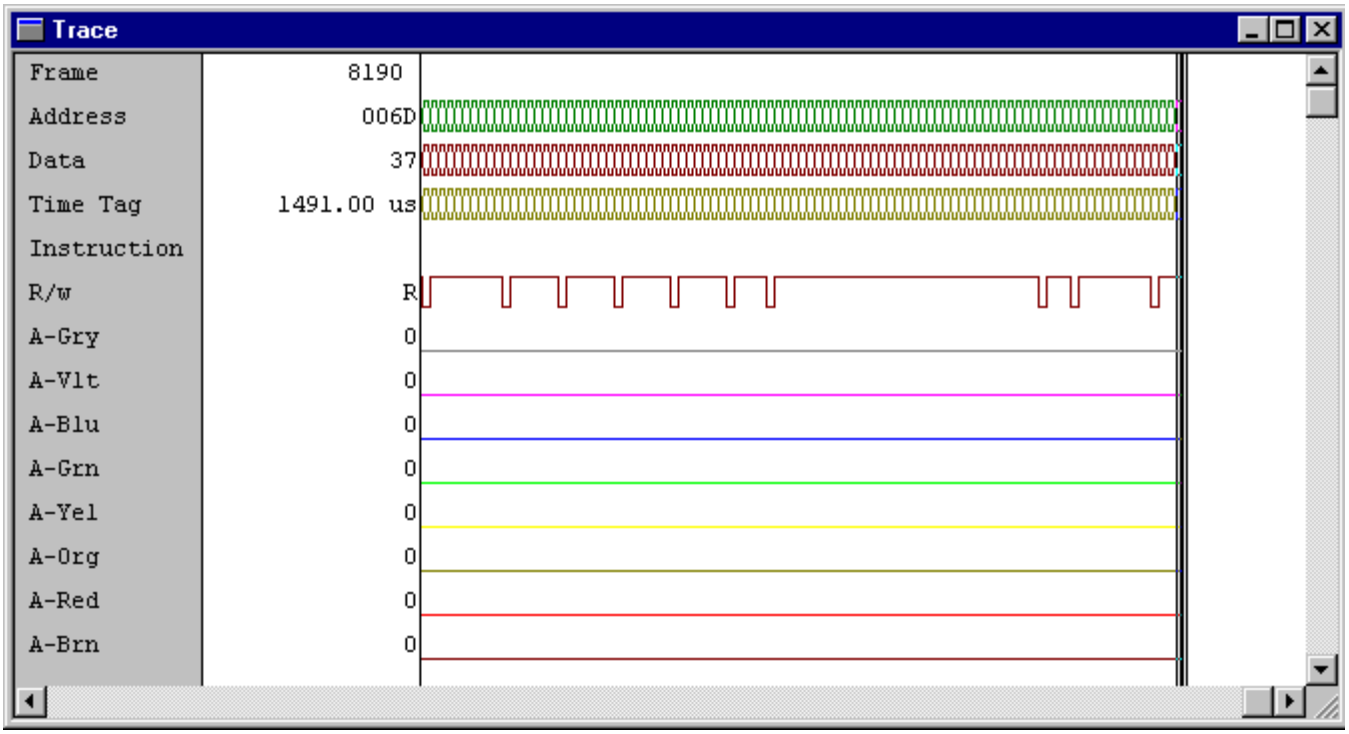


Figure D-16. Trace Window — Zoom Out

Dragging the marker over the bus analyzer data display may also generate updates in component windows, for example, the **Source** and **Assembly** windows. Thus the bus analyzer data can be examined in the continually updated windows as the marker is moved over the graphic or text display of the bus analyzer data.

In graphic mode, the vertical bar indicates the marker position and displays all information for the current frame. In textual or instruction mode, the horizontal bar indicates the position of the marker and displays information for the current frame.

D.8.6 Set Time Base

To reference a frame with a tag value of 0, hold the mouse cursor over a frame and click the right mouse button. The **Trace** pop-up menu appears. Select **Set Time Base** and the time tag values will be displayed relative to the selected frame.

D.8.7 Show Location

Select a frame and click the right mouse button to display the **Trace** menu, then select **ShowLocation**. The marker can be positioned with a left mouse click. The **Source** and **Assembly** windows are updated automatically.

D.9 Add/Remove Items in the Trace Window

In the default configuration, the following items are displayed in the window: **Frame**, **Events**, **Address**, **Data**, **Time Tag**, **Instruction**, and **R/W**. It is possible to add or remove items and rearrange their order by using the buttons provided in this dialog box.

To edit an item (set its color or assign a name) select **More...**.

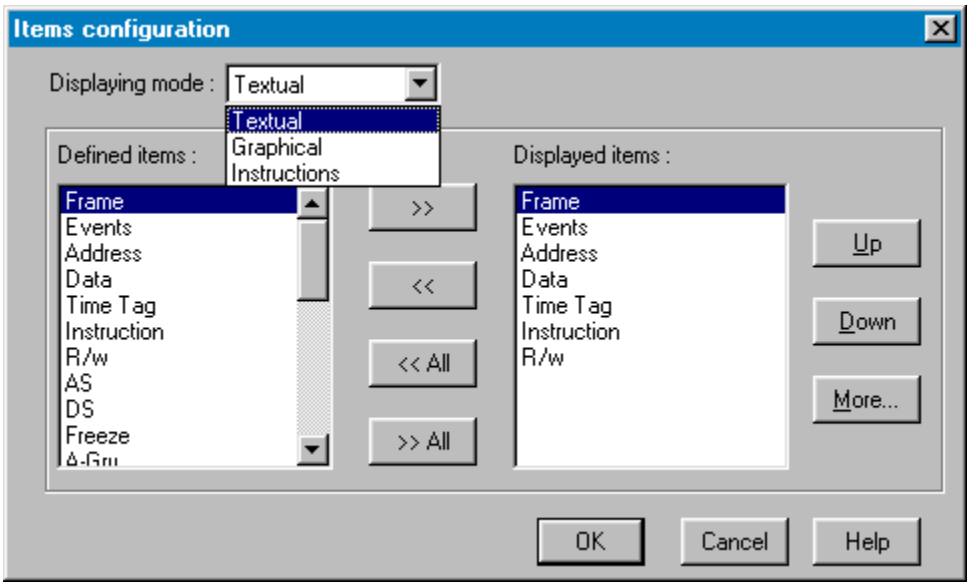


Figure D-17. Items Configuration Dialog Box

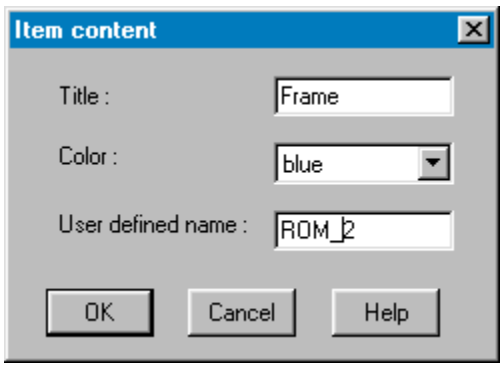


Figure D-18. Item Content Dialog Box

D.10 Searching

It is possible to search for a specific trace buffer frame. It is also possible to search for one or more events defined to trigger the bus analyzer or search for a specific pattern.

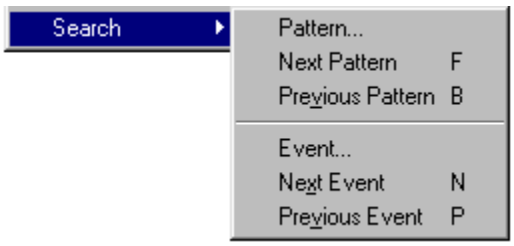


Figure D-19. Search for Event or Pattern

D.10.1 Search for a Frame

Select **Trace | Go to Frame...** and enter the desired frame number to search for a frame. If the frame is found, it is selected. If the selected frame is not visible, the **Bus Analyzer** window will scroll to the frame. If the frame number is larger than the last frame stored in the trace buffer, the bus analyzer will scroll to the last frame.

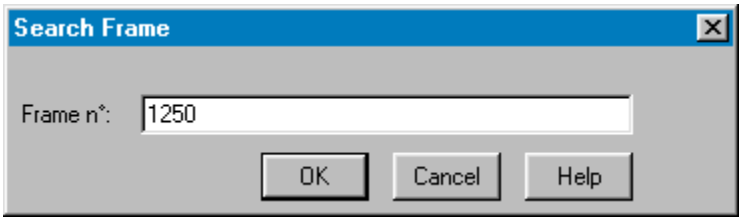


Figure D-20. Search Frame Dialog Box

D.10.2 Search for Events

To search for an event, select one or more events in the **Search Event Specification** dialog box.

OK Button	Click **OK** to store the selected events and close the dialog box.
Forward	Searches forward to find the next frame that matches the specified event
Backward	Searches backward to find the previous frame that matches the specified event

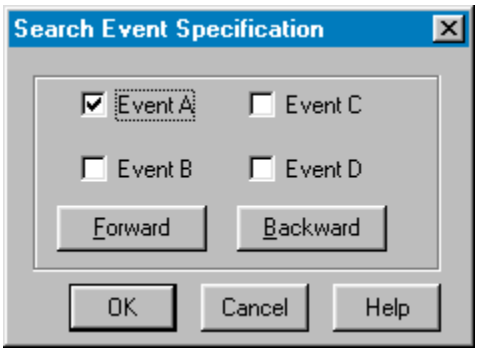


Figure D-21. Search Event Specification Dialog Box

D.10.3 Next Event

In the **Trace** pop-up menu, select **Search | Next Event** to search forward for the next frame that matches the specified event. The user can also press the N key while the **Trace** window is active to find the next event.

D.10.4 Previous Event

Open the **Trace** pop-up menu and select **Search | Previous Event** to find the previous frame that matches the specified event. The user can also press the P key while the **Trace** window is active.

D.10.5 Search for a Pattern

To search for a frame with a specific pattern, select **Trace | Search | Pattern...**. Define the search pattern to find all stored bus cycles that match the pattern. The pattern consists of an address, a data word, logic clips, and four miscellaneous signals.

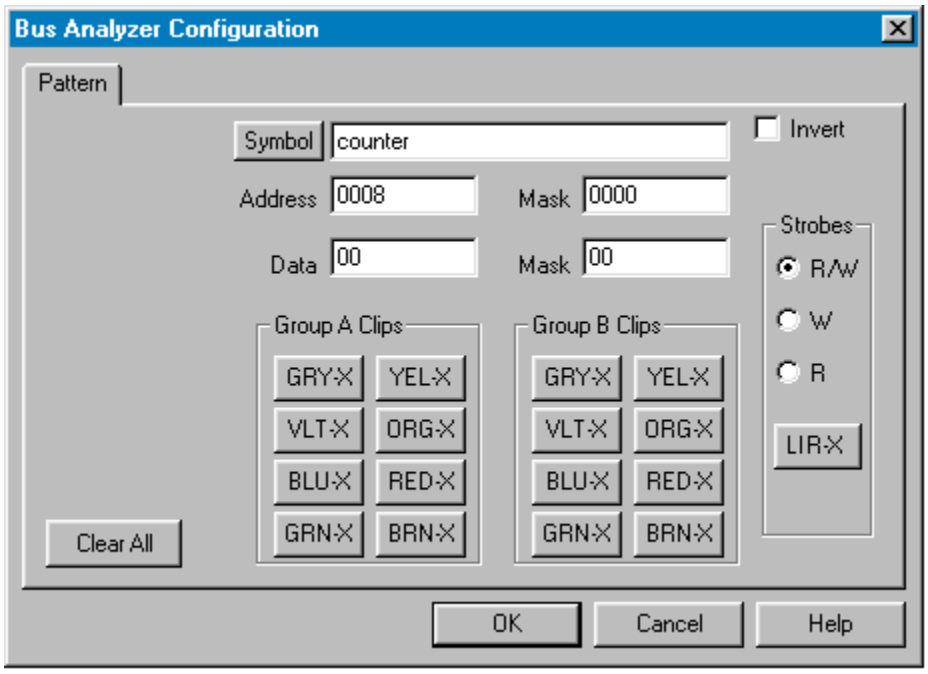


Figure D-22. Search for Pattern

The bus analyzer will search for the defined pattern in the trace buffer, then highlight and display the matched frame in the **Trace** window.

Address	Specify the **Address** to be matched.
Data	Specify the **Data** value to be matched.
Strobes	Specify the state of R/W and LIR-X to be matched.
Group A/B Clips	Toggle to specify each logic clip as high (H), low (L), or don’t care (X).

The clips buttons show Group A and B logic clips with their respective colors.

Invert	Select **Invert** to find any frame that does not match the defined pattern.
OK Button	Click **OK** to store the specified values.
Cancel Button	Click **Cancel** to retain previous settings.

D.10.6 Next Pattern

Select the menu entry **Trace | Search | Next Pattern** to search for the next occurrence of the pattern. If the pattern is not found, an error message is displayed.

D.10.7 Previous Pattern

Select **Trace | Search | Previous Pattern** to search for the previous occurrence of the pattern. If the pattern is not found, an error message is displayed.

D.11 Dumping Bus Analyzer Data to a File

To dump bus analyzer data to a file, open the **Trace** pop-up menu and select **Dump...**.

Dump File	Enter the name of the file to which data is to be written.
Select	Click **Select** to open a standard open file dialog box.
Frames to Dump	Enter the frames to be dumped to the file.
All	Select **All** to dump all frames within the specified range to the file.
Instructions	Select **Instructions** to dump just the frames in the specified range that contain an instruction.
OK Button	Click **OK** to dump the bus analyzer data to the file.
Cancel Button	Click **Cancel** to cancel the operation and no data will be dumped.

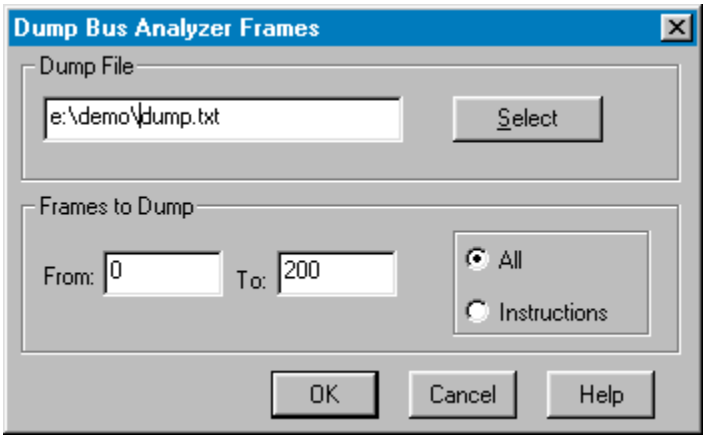


Figure D-23. Dump Bus Analyzer Frames Dialog Box

D.12 MMDS Commands

This section describes MMDS commands. These commands can be typed in the debugger command line component or inserted in a command file.

ARM — Arm Bus Analyzer

Description:

The ARM command arms the bus analyzer. When armed, the analyzer records bus cycles when the emulator is executing user code. Arming the analyzer clears the current contents of the trace buffer.

Syntax:

```
ARM
```

BAUD — Baud Rate

Description:

The BAUD command sets the baud rate for communication between the system controller and host computer. For maximum performance, the baud rate should be set as high as the host computer can accommodate. The maximum rate is 115,200, the default is 9600. Enter the command without a rate to display the **Communication Device** dialog box.

If the system does not support a specified baud rate, an error message is displayed.

Syntax:

```
BAUD [rate]
```

Argument:

`rate`	Specifies the baud rate; either: 1200, 2400, 4800, 9600, 19,200, 28,800, 38,400, 57,600, or 115,200

Example:

```
BAUD 57600
```

Changes the communication baud rate to 57,600

CT — Clear Trigger

Description:

The CT command clears the values of specified bus analyzer triggers (events), A, B, C, and/or D. The command also disables the cleared triggers.

Syntax:

```
CT <list> | *
```

Arguments:

list	List of trigger identifiers
*	All triggers (A, B, C, and D)

Examples:

```
CT A B
```

Clears triggers A and B

```
CT *
```

Clears all triggers

When clearing a trigger that is part of a range, the second trigger in the range is cleared at the same time. For example, if a range is defined between triggers C and D, `CT C` clears trigger C and D.

DARM — Disarm Bus Analyzer

Description:

DARM disarms the bus analyzer. When disarmed, the analyzer does not record bus cycles.

Syntax:

```
DARM
```

EXIT — Terminate Session

Description:

The `EXIT` command terminates the host session with the target.

Syntax:

```
EXIT
```

GE — Go to Event

Description:

The `GE` command searches forward or backward in the analyzer trace buffer for a frame that matches the search event defined with this command. For a forward search, the search begins at the frame immediately following the current frame; a backward search begins at the frame immediately preceding the current frame.

Syntax:

```
GE <list> | * [;B]
```

Arguments:

`list`	List of events (`A`, `B`, `C`, and `D`) separated by a space character or a comma
*	All events (`A`, `B`, `C`, and `D`)
`;B`	Specifies backward search. When this option is omitted, a forward search is performed.

Example:

```
GE A B
```

Moves the cursor to the next frame that contains the event A and/or B

GF — Go to Frame

Description:

`GF` moves the cursor to a specified trace buffer frame. When the number of the specified frame is greater than the number of frames currently stored in the buffer, the command moves to the last frame.

Syntax:

`GF <frame>`

Argument:

`frame`	Specifies a frame number in the range of 0...8190. The frame is always specified as a decimal number, regardless of the current default number base.

Examples:

```
GF 4096
```

Moves the cursor to frame 4096

```
GF 32768
```

Moves the cursor to the last frame in the buffer

GP — Go to Analyzer Search Pattern

Description:

GP searches forward or backward in the analyzer trace buffer for a frame that matches the search pattern defined with the SP command. A forward search begins at the frame following the current frame. A backward search begins at the frame preceding the current frame. When the search finds a frame that matches the search pattern, the line cursor is vertically positioned on the matching frame in the center of the screen. If a matching frame is not found, the line cursor is not moved. When no search pattern is defined, the command searches for any pattern, for example, the line cursor moves to the next line (or preceding line).

Syntax:

```
GP [;B]
```

Argument:

;B	Specifies a backward search. When this option is omitted, a forward search is performed.

Examples:

```
GP
```

Searches forward for the next frame that matches the search pattern

```
GP ;B
```

Searches backward for the previous frame that matches the search pattern

LF — Log File

Description:

The LF command is used to specify a log file to store contents of the bus analyzer trace buffer.

Syntax:

```
LF <filename>
```

Example:

```
LF mylog.txt /* Create log file to obtain trace buffer data*/
```

LOADMAP — Load Memory Map

Description:

The LOADMAP command allows the user to load a memory map from a file. If the specified file starts with a number it should be preceded by a period and slash (.\) (for example, `LOADMAP .\00123V22.mem).`

Syntax:

```
LOADMAP <filename> | <MCU ID>
```

Arguments:

`filename`	Specifies a memory file that defines a memory map
`MCU ID`	MCU identifier. If specified, the command will load the memory file that matches the MCU Id.

Example:

```
LOADMAP 0x3C1 /* Loads memory map from a file that matches the MCU Id */
```

LT — Log Trace

Description:

The `LT` command copies the contents of the bus analyzer trace buffer to the log file. The trace buffer is logged in the current view format. An `LF` (log file) command must be executed before entering the `LT` command to initiate logging. The target must be stopped to execute an `LT` command.

Syntax:

```
LT [<range>]
```

Argument:

`range` Specifies a range of frames to be copied. If range is omitted, all frames in the trace buffer are copied. The default number base for range is decimal.

Examples:

```
LF logfile   /* Open the log file */
LT 10..20 /* Copy frames 10 to 20 to log file */
NOLF   /* Close the log file */
```

OSC — Select Emulator Clock Frequency

Description:

The `OSC` command selects the emulator clock frequency. Six clock frequencies are available. Five internally generated clock frequencies are available: 16 MHz, 8 MHz, 4 MHz, 2 MHz, and 1 MHz. An external clock source is also available. The default emulator clock rate is set by the firmware and adapted to the current frequency. Enter this command without parameters to open the equivalent **Target Signals** dialog box.

Syntax:

```
OSC [<rate> | <source>]
```

Arguments:

`rate`	`OSC1MHZ`	Selects the 1-MHz oscillator
	`OSC2MHZ`	Selects the 2-MHz oscillator
	`OSC4MHZ`	Selects the 4-MHz oscillator
	`OSC8MHZ`	Selects the 8-MHz oscillator
	`OSC16MHZ`	Selects the 16-MHz oscillator
`source`	`EXT`	Selects an external clock source

Example:

```
OSC osc8Mhz   /* Use the 8 MHz internal emulator clock */
```

Disables the real-time memory block at base address $1000

PROTOCOL — Communication Protocol Command

Description:

The `PROTOCOL` command allows the user to display the communication protocol between the debugger and the server in the **Command Line** window.

Syntax:

```
PROTOCOL [ON | OFF]
```

Arguments:

`ON`	Protocol is displayed in the **Command Line** window (default).
`OFF`	Protocol is not displayed in the **Command Line** window or logged in the log file.

RESET — Reset Command

Description:

The RESET command resets the target MCU. RESET GO performs a reset of the EM (emulator) and automatically starts execution of the code starting at the address stored in the reset vector. RESET STOP performs a reset and waits for user commands.

Syntax:

```
reset [GO | STOP]
```

Arguments:

GO	Resets the MCU and does a Go from Reset.
STOP	Resets the MCU and stops (default).

Examples:

```
Reset Go
```

Resets the MCU and does a Go from Reset.

```
Reset
```

Resets the MCU and stops (default setting).

NOTE: *This command does not execute the reset command file (reset.cmd). The* **MMDS0508 | Reset** *menu option executes the reset.cmd file.*

RTMEM — Real-Time Memory

Description:

The `RTMEM` command allows the user to enable or disable the real-time memory.

Syntax:

```
RTMEM [<address>][;E | ;D]
```

Arguments:

`address`	The base address of the real-time memory block to be enabled or disabled. If not specified, it is set to 0.
`;E`	Enables the real-time memory block (default)
`;D`	Disables the real-time memory block

Example:

```
RTMEM 0x1000 ;D
```

Disables the real-time memory block at base address $1000

SC — Set Analyzer Clock

Description:

The SC command sets the source for the time tag clock, which increments the trace buffer time tag. Alternately, if no parameters are specified, the command interpreter displays the **Bus Analyzer Configuration** dialog box. Then the user can select the **Time Tag Clock** tab and set the clock source.

Syntax:

```
SC [<timetag> [<frequency>]]
```

Arguments:

`timetag` Specifies the source for the time tag clock. Sources are:

`OSC1MHZ`	Selects the 1-MHz oscillator
`OSC2MHZ`	Selects the 2-MHz oscillator
`OSC4MHZ`	Selects the 4-MHz oscillator
`OSC8MHZ`	Selects the 8-MHz oscillator
`OSC16MHZ`	Selects the 16-MHz oscillator
`EXT`	Selects the external clock
`BUS`	Selects the bus clock

`PROGRAM` Selects the programmable clock. When the programmable clock is selected and the frequency argument is omitted, the default frequency is 50 Hz.

`frequency` Specifies the time tag clock frequency for the programmable clock, which is in the range of 50 Hz to 50 kHz. The frequency is entered as a decimal hertz value. The programmable clock source has a fundamental frequency of 500,000 Hz, and the frequency should be the quotient of 500,000 Hz divided by an integer. Thus, 50,000 is valid, but 49,000 is invalid because an invalid frequency is rounded up to the next higher frequency.

Examples:

```
SC PROGRAM 100
```

Sets the programmable clock (running at 100 Hz) as the time tag clock source

```
SC OSC8MHz
```

Sets the 8-MHz oscillator as the time tag clock source

SIG — Set Emulator Signal

Description:

The SIG command enables or disables control signals from the target MCU and controls emulation. If a signal or list of signals is entered without specifying ENABLE or DISABLE, the signals are enabled.

Syntax:

```
SIG [ [ENABLE] <signal> {<signal>}]
[ DISABLE <signal> {<signal>}]
```

Arguments:

signal	Signal to be enabled or disabled; available signals are RESETIN and RESETOUT.
ENABLE	Connect signal from the target system.
DISABLE	Disconnect signal from the target system.

Example:

```
SIG ENABLE RESETIN
```

Enables the RESET IN signal from the target system

ST — Trigger Command

Short description:

Set trigger

Syntax:

```
ST[<id> [[!] [(<address> |<address range> | ,)
[(<data> | <data range> | ,)
[(<clips> | ,)[LIR= (X | H | L)]]]] [;R | ;W| ;RW]
[;D] ]]
```

Description:

The ST command sets the value of one of the four bus analyzer triggers. If a value is set or only the triggered ID is entered, the command interpreter displays the **Trigger** tab of the **Bus Analyzer Configuration** dialog box for the user to set a trigger value. Do not include a space character when entering the command and ID on the command line. For example, `in>sta`

`id`	Specifies the trigger ID of the analyzer trigger: A, B, C, or D.
`!`	The inversion operator applies to the entire trigger. When `!` is specified and an address, data value, and clip value are specified, the trigger occurs when the address, data value, or clip value is not the specified value. When `!` and a range are specified, the trigger occurs at values outside the range and at the lowest value in the range.
`address`	An address to which a trigger is set. The address is specified with an address constant, as follows: `<address>[:<mask>]`

When a mask is entered, only the bits of the address that correspond to 1s in the mask are used in the comparison.

`address range` A range of addresses within which a trigger is set. The address range can be specified with a start and end address constant or with a start address and a length value, as follows:

`<start-address>`[:`<mask>`][..`<end-address>`]
or
`<start-address>`[:`<mask>`][::`length`]

When a mask is entered, only the bits of the address that correspond to one bits (1s) in the mask are used in the comparison. When a length and mask are entered, the length is added to the start address and the mask is applied to the start address and sum to obtain the end address.

`,` The comma indicates that the address, address range, data, data range, clips, or clips range has been omitted. The omitted item is ignored in the trigger.

`data` A data value that defines the trigger. The value is specified as:
<value>[:`<mask>`]

When a mask is entered, only bits of the value that correspond to 1s in the mask are used in the comparison.

`data-range` A range of data values that define the trigger. The data range can be specified with a start and end value or with a start value and a length, like this:

`<start-value>`[:`<mask>`][..`<end-value>`]
or
`<start-value>`[::`<length>`]

When a mask is entered, only bits of the value that correspond to one bits (1s) in the mask are used in the comparison. When a length and a mask are entered, the length is added to the start value. The mask is applied to the start value and to “start value + length” to obtain the end address.

`clips` A 5-bit value that defines logic clip signals on the MMDS0508 analyzer for the trigger. The value is specified like this:

```
<clips>[:<mask>]
```

When a mask is entered, only bits of the value that correspond to 1s in the mask are used in the comparison. For example, the code `0x1F:0x1F` sets all clips of Term A to H. If a mask is not specified, the value `0x1F` will be assigned by default.

Each trigger clip line has three options:

1. H — High
2. L — Low
3. X — Don’t care

The bits of the clips and mask words are:

Group A	Bit	Signal
	0	BRN
	1	RED
	2	ORG
	3	YEL
	4	GRN
	6	LIR (active low)

`;R` Trigger on a read bus cycle only.

`;W` Trigger on a write bus cycle only.

`;RW` Trigger on a read or write bus cycle.

`;D` Disable trigger; for example, set trigger value and disable trigger.

`LIR` Trigger on a specific value for LIR signal.

LIR = H Triggers when LIR is high
LIR = L Triggers when LIR is low
LIR = X Triggers when LIR is high or low

When the user specifies a range for one of the address or data options, a second option is also seen as a range. For example, the command `STC 8 20..40` is interpreted as `STC 8.. 8 20..40`, and the command `STC 8..9 20` is interpreted as `STC 8..9 20..20`.

If neither `R` nor `W` is specified, the trigger defaults to a read/write bus cycle.

Set the LIR signal with the clips (bit 6) or the LIR option.

The address, data, or clips will be ignored if the matching mask is specified as 0.

If a bit is set in both the clips and masks, the trigger is set to high (-H). If a bit is not set in clips, but is set in masks, the trigger is set to low (-L). If a bit is not set in the mask, the trigger does not depend on the state of the clip.

Examples:

```
STA 0x1000
```

Sets analyzer trigger A to match accesses at address $1000

```
STB , 4
```

Sets analyzer trigger B to match accesses with a value of 4, at any address

```
STC 8 20..40
```

Sets analyzer triggers C and D to match accesses using a value from 20 to 40 at address 8

```
STC 8..10 20
```

Sets analyzer triggers C and D to match accesses using value 20 at an address from 8 to 10

NOTE: *Bits set to 0 are don't care bits. Bits set to 1 are bits that expect the value for Address, Data, or Clip. Bits set to 0 are in a position that accepts 0 or 1. Bits set to 1 are in a position that only accepts values from the address fields. For the code* `Address 0xC000 Mask 0xFFFC`*, the trigger is detected for* `0xC000` *and* `0xC001`*,* `0xC002`*, or* `0xC003` *is loaded in the address bus. For the code* `Address 0x00B0 Mask 0x00F0`*, the trigger is detected when an address in the range [*`0xB0..0xBF`*] is loaded.*

SP — Set Analyzer Search Pattern

Short description:

Set analyzer search pattern

Syntax:

```
SP [!] [(<address> | ,) [(<data> | ,)
[(<clips> | ,)[LIR = (X | L  | H)]]][;R | ;W| ;RW]
```

Description:

The SP command defines a pattern for searching the bus analyzer trace buffer. The goto search pattern command (GP) performs the search for the pattern defined by the most recent SP command. If no parameter is specified, the **Bus Analyzer Search Pattern** dialog box is displayed.

`!` — The inversion operator applies to a specified single address or data value. When `!` is specified, the command sets a pattern on addresses and/or data values other than the specified address or data value.

`address` — An address to which a pattern is set. The address is specified with a constant:

```
<address> [:mask]
```

When a mask is entered, only the bits corresponding to one bits (1s) in the mask are useful in the pattern.

`,` — The comma indicates that the address, data, or clips byte has been omitted. The omitted item matches any value of the omitted type (don’t care).

`data` — Data value that defines the data pattern. The value is specified as:

```
<value> [:mask]
```

When a mask is entered, only the bits corresponding to one bits (1s) in the mask are useful in the pattern.

`clips` — 16-bit value that defines logic clip signals for the search pattern. The value is specified as:

```
<clips> [:mask]
```

When a mask is entered, only the bits corresponding to one bits (1s) in the mask are useful in the pattern.

The bits of the clips are:

Bit		Signal	Bit		Signal
0	**Group A**	BRN	8	**Group B**	BRN
1		RED	9		RED
2		ORG	10		ORG
3		YEL	11		YEL
4		GRN	12		GRN
5		BLU	13		BLU
6		LIR	14		VLT
7		GRY	15		GRY

`;R` Search for a read bus cycle only.

`;W` Search for a write bus cycle only.

`;RW` Search for a read or write bus cycle.

`LIR` Search for a specific value for LIR signal.

LIR = H LIR is high.
LIR = L LIR is low.
LIR = X LIR is high or low.

If a bit is set in both clips and masks, the pattern is detected when the bit is high (-H). If a bit is set in masks but not in clips, the pattern is detected when the bit is low (-L). If a bit is not set in masks, the pattern does not depend on the state of the clip.

Examples:

```
SP 0x1000
```

Sets analyzer search pattern to search for accesses at address $1000

```
SP , 4
```

Sets analyzer search pattern to search for accesses with a data value of 4, at any address

```
SP 8 20
```

Sets analyzer search pattern to search for accesses using a value of 20 at address 8

NOTE: *Bits set to 0 are don't care bits. Bits set to 1 are bits that expect the value for Address, Data, or Clip. Bits set to 0 accept 0 or 1. Bits set to 1 only accept values from address fields. For* `Address 0xC000 Mask 0xFFFC`*, the search pattern is detected for* `0xC000` *and* `0xC001`*,* `0xC002`*, or* `0xC003` *is loaded in the address bus. For* `Address 0x00B0 Mask 0x00F0`*, the search pattern is detected when an address in the range [*`0x00B0..0xFFBF`*] is loaded.*

SQ — Set Sequencer

Description:

The SQ command sets the analyzer sequencer mode. Alternately, if no parameters are specified, the command interpreter displays the **Bus Analyzer** configuration dialog. Then the user can select the **Sequencer** tab and program the sequencer.

Syntax:

```
SQ [<mode> [<count>] [;S] ]
```

Arguments:

`mode` Selects the sequencer mode. Modes are:

`ALL`	Records all bus cycles
`EVENT`	Records events only
`SEQ0`	Sequential recording mode A+B+C+D
`SEQ1`	Sequential recording mode A+B->C+D
`SEQ2`	Sequential recording mode A->B->C D<-
`SEQ3`	Sequential recording mode A->B->C ->D
`SEQ4`	nth event of A+B+C+D; After collecting nth events, then 4095 more cycles are collected in the trace buffer.

`count` Specifies terminal count for the Counted: ALL cycles and Counted: Events Only modes as a decimal value. Causes frames of the appropriate type to be recorded. Then the analyzer stops recording and is disarmed. If count is omitted, the count defaults to one (1). Specifies post-trigger count (decimal) for the sequential modes. Causes count bus cycles to be recorded after the trigger point. If count is omitted, the default is one. This also specifies n of the nth event of A+B+C+D sequencer.

`;S` Stops the emulator when recording stops. Thus the analyzer can be used as a sequenced breakpoint machine.

Examples:

```
SQ ALL 100 /*Specifies 100 bus cycles. After 100 cycles, the analyzer disarms and stops recording.*/

SQ EVENT 10 ;S /* Records 10 event cycles, then stops the emulator. */

SQ ALL /* Sequencer is set to Continuous: All Cycles */
```

TD — Trigger Disable

Description:

The `TD` command disables specified triggers.

Syntax:

```
TD <list> | *
```

Arguments:

`list`	List of triggers to disable; each trigger is separated by a comma or space character.
*	All triggers (A, B, C, and D)

Examples:

```
TD A,B
```

Disables triggers A and B

```
TD *
```

Disables all triggers

TE — Trigger Enable

Description:

The TE command enables specified triggers.

Syntax:

```
TE <list> | *
```

Arguments:

list	List of triggers to enable; each trigger is separated by a comma or a space.
*	All triggers (A, B, C, and D)

Examples:

```
TE A B
```

Enables triggers A and B

```
TE *
```

Enables all triggers

TT — Display Time Tag Difference

Description:

The `TT` command displays the difference between the time tags in the two trace frames of the analyzer display. When frame numbers are not entered, the command displays the difference between the beginning and ending frames.

Syntax:

```
TT [<sframe> [<eframe>]]
```

Arguments:

`sframe`	Starting frame number
`eframe`	Ending frame number. When `eframe` is omitted, the end frame in the trace buffer is used to calculate the difference.

Examples:

```
TT
```

Displays the difference between the beginning and ending frames in the trace buffer

```
TT 80
```

Displays the difference between frame 80 and the ending frame in the trace buffer

```
TT 10 40
```

Displays the difference between frames 10 and 40 in the trace buffer

VA — Analyzer View

Description:

The VA command selects the display format of the analyzer trace buffer.

Syntax:

```
VA [MODE=(MIX | INS | GRAPH)]
```

Arguments:

MODE=	Followed by one of the three options indicates the bus analyzer display mode.
MIX	Mixed view
INS	Instruction view
GRAPH	Graphical view

Example:

```
VA MODE=MIX
```

Sets mixed view mode

MMDS0508 Target

Index

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

Z

Index

Need to know more? That's ez, too.

Technical support for MCUez development tools is available through your regional Motorola office or by contacting:

Motorola, Inc.
6501 William Cannon Drive West
MD:OE17
Austin, Texas 78735
Phone (800) 521-6274
Fax (602) 437-1858
CRC@CRC.email.sps.mot.com

How to reach us:
USA/EUROPE/Locations Not Listed: Motorola Literature Distribution; P.O. Box 5405, Denver, Colorado 80217, 1–800–441–2447 or 1-303-675-2140.
Customer Focus Center: 1–800–521–6274
JAPAN: Motorola Japan Ltd.; SPD, Strategic Planning Office, 141, 4–32–1, Nishi–Gotanda, Shinagawa–ku, Tokyo, Japan, 03–5487–8488
ASIA/PACIFIC: Motorola Semiconductors H.K. Ltd., Silicon Harbour Centre, 2 Dal King Street, Tai Po Industrial Estate, Tai Po, New Territories, Hong Kong, 852–26668334
Mfax™, Motorola Fax Back System: RMFAX0@email.sps.mot.com; http://sps.motorola.com/mfax/; TOUCHTONE, 1-602–244–8609; US & Canada ONLY, 1–800–774–1848
HOME PAGE: http://motorola.com/sps/

Mfax is a trademark of Motorola, Inc.

MOTOROLA
Semiconductor Products Sector