Contents

9. JTAG and Multi-ICE 9-1
9.1. 實驗目的 9-1
9.2. 實驗原理 9-1
9.2.1. About AXD 9-1
9.2.2. Multi-ICE 9-3
9.3. 引導實驗 9-5
9.3.1. 實驗步驟 9-5
9.4. 實驗要求 9-21
9.5. 問題與討論 9-21
9.6. 參考文件及網頁 9-21

9. JTAG and MultiI-CE

9.1. 實驗目的

In this Lab the debugger target is Multi-ICE unit and an ARM Integrator board. You should have set up the hardware and the software of Multi-ICE unit and target board. The debugging skills you learn is the same as Lab3 except you do the debugging tasks with Multi-ICE. You will learn how to start-up the Multi-ICE server and debug program.

9.2. 實驗原理

9.2.1. About AXD

Debugger introduction:

A debugger is software that enables you to make use of a debug agent in order to examine and control the execution of software running on a debug target. ARM support two methods to do this work. One is AXD, the ARM eXtended Debugger. The other is armsd, the ARM Symbolic Debugger. They have the same function to debug the selected target. But AXD with GUI interface, is much easier to use it.
The debugger issues instructions that can:

- Load software into memory on the target
- Start and stop execution of that software
- Display the contents of memory, registers, and variables
- Enable you to change stored values.

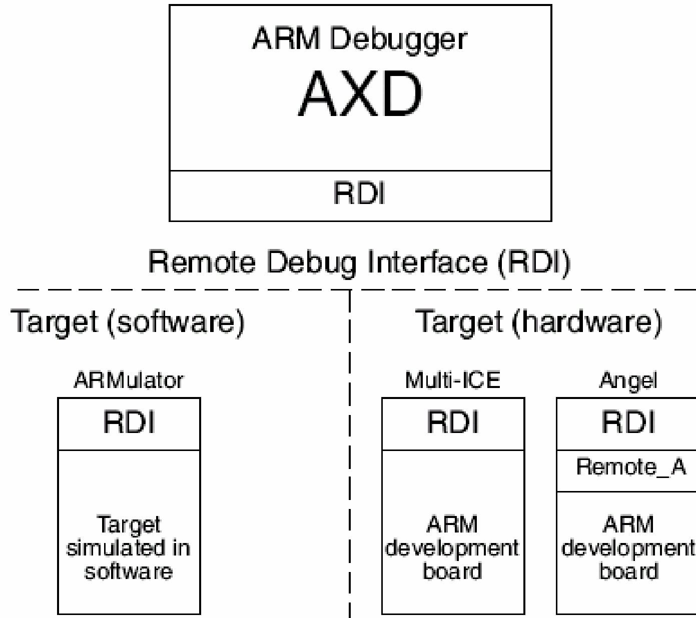


Figure 1 Debugger-Target Interface

Debug target:

Debug target can be classified as software and hardware target. In software simulation, the expected behavior of the product is simulated by software. Even though you might run this software on the same computer as the debugger, it is useful to think of the target as a separate piece of hardware. They support ARMulator as the software simulation tool.
In hardware simulation, the interfaces can be classified as Multi-ICE and Angel types. This means that debugger scope and control the hardware device though Multi-ICE (ARM support) or Real-Monitor (Angel support) interface, adding logic, and probe.
In our course, we suggest Multi-ICE as the basic interface connected between hardware and host computer.

Remote debug interface:

The Remote Debug Interface (RDI) is an ARM standard procedural interface between a debugger and the debug agent (see Figure 1-1 on page 1-6).
RDI gives the debugger a uniform way to communicate with:

- a debug agent running on the host (for example, ARMulator)
- a debug monitor running on ARM-based hardware accessed through a communication link (for example, Angel)
- a debug agent controlling an ARM processor through hardware debug support (for example, Multi-ICE).

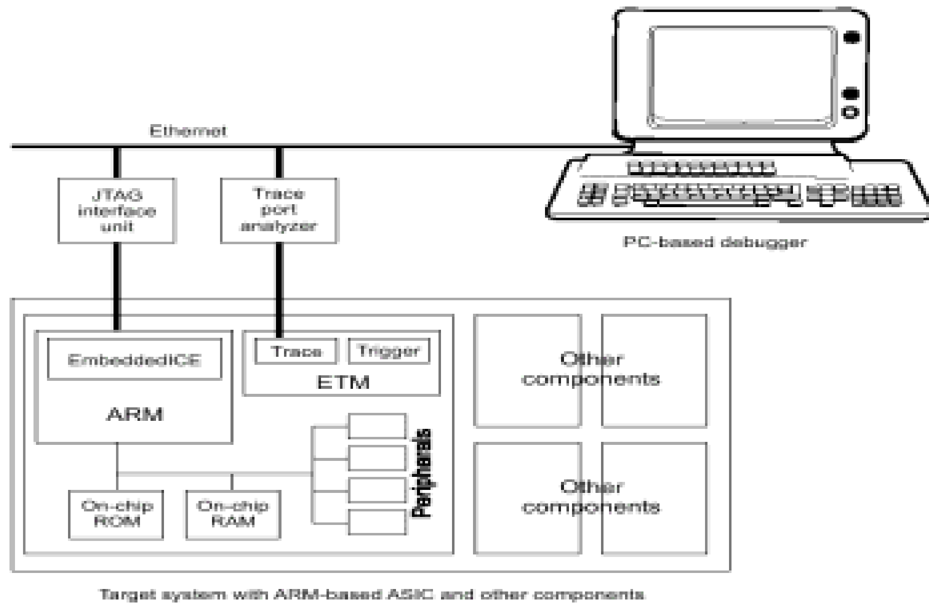


Figure 2 Debugging connection

9.2.2. Multi-ICE

What is Multi-ICE?

An interface unit that connects the parallel port of a workstation to the JTAG interfaces of an ASIC that includes debug and EmbeddedICE capability

Function of Multi-ICE:

1. Debug extensions to the ARM core
2. The EmbeddedICE logic
3. The ICE extension unit

The debug extensions consist of a number of scan chains around the processor core and some additional signals that are used to control the behavior of the core for debug purposes:

- BREAKPT: enables external hardware to halt processor execution for debug purposes. active high
- DBGRQ: is a level-sensitive input that causes the CPU to enter debug state when the current instruction has completed.
- DBGACK: is an output from the CPU that goes high when the core is in debug state.

The EmbeddedICE logic is the integrated onchip logic that provides JTAG debug support for ARM core, which can be accessed through the TAP controller on the ARM core using the JTAG interface.

The ICE Extension Unit(IEU) is a logic block that can be added to the EmbeddedICE logic when a processor is fabricated.

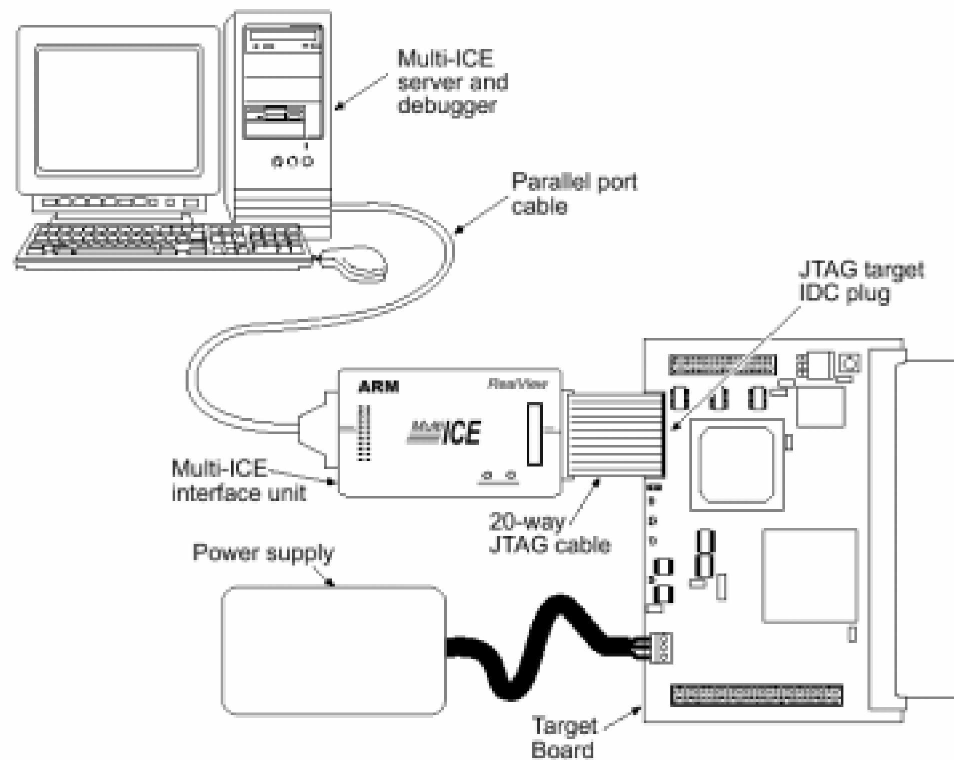


Figure 3 Multi-ICE connection

Feature of Multi-ICE:

For the real time debug, a system may have many different devices operating at the same time. If they all connect to just one host computer to control and to trace these devices, it is impossible. So they separate this work into many computer, and then use an application so called “port mapper” to synchronize this signal though network connectivity.

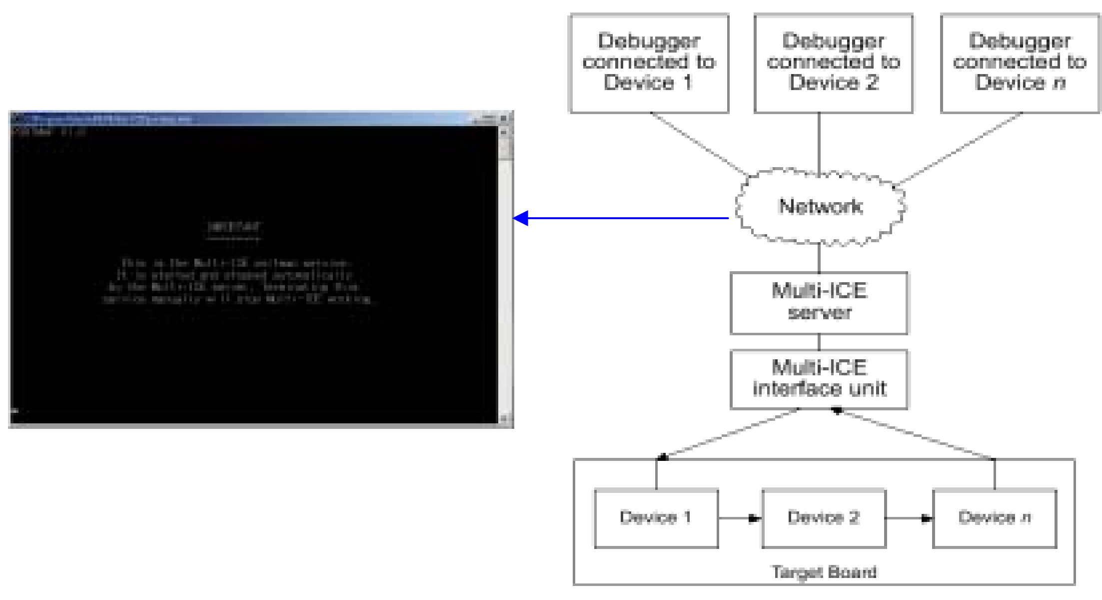


Figure 4 Debug and Timing synchronization from network connection

9.3. 引導實驗

9.3.1. 實驗步驟

The following instructions are based on the demonstration program that runs the Dhrystone test software. For details of the Dhrystone test program, please refer to the readme.txt file and the various source files in its subdirectory. (e.g., C:\Program Files\ARM\ ADSv1_1\Examples\dhryansi\)

Starting the Multi-ICE server

To start the Multi-ICE server

1. Ensure that:
   - The Multi-ICE interface unit is plugged into the workstation
   - The Multi-ICE interface unit is plugged into the target JTAG connector
   - The target is powered up
   - The green power light on the interface unit is glowing brightly
2. Select **Start** -> **Program** -> **ARM Multi-ICE v2.2** -> **Multi-ICE Server.** The software displays the Multi-ICE server window, shown in Figure 1. The portmap application might also be started and minimized, depending on the host computer configuration.

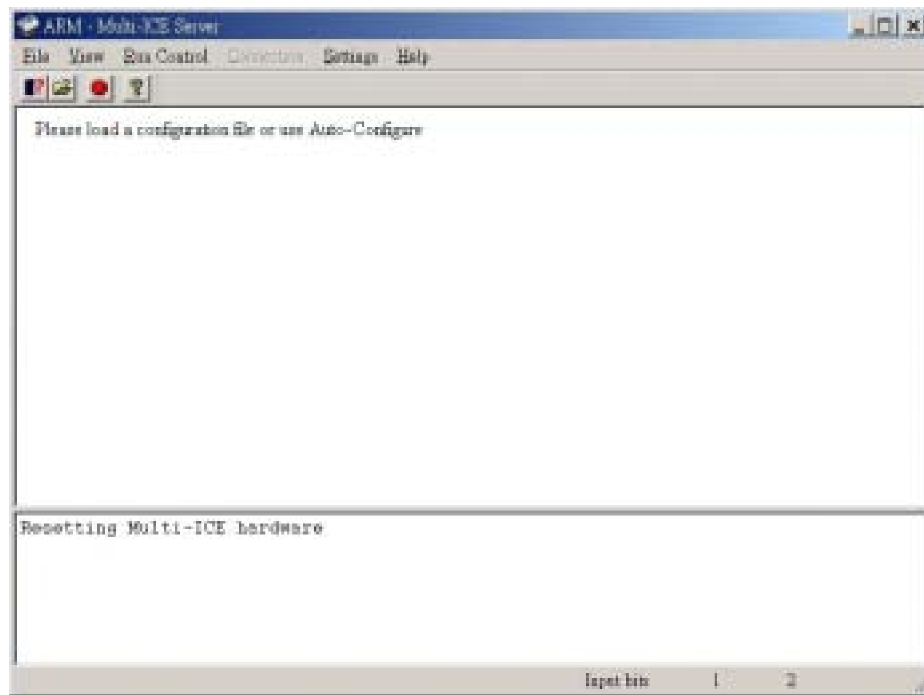
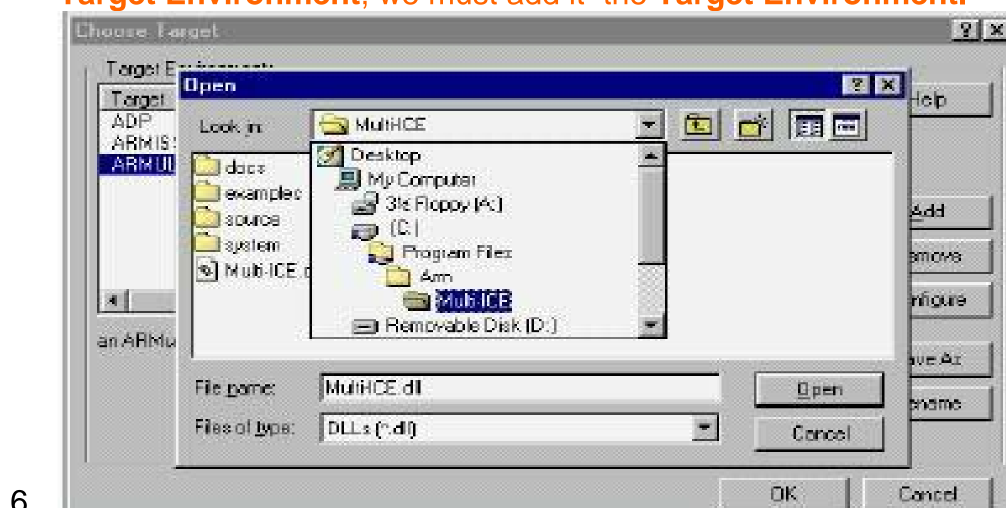


Figure 5Unconfigured Multi-ICE server window

3. If a dialog box appears informing you that the Multi-ICE hardware cannot be found, click on OK and recheck the items listed in step 1.
4. Configure the server. This can usually be done using the Autoconfiguration command. Select **File** -> **Auto-configure** and wait until the server has examined the target. If the configuration works, the screen looks similar to Figure 2.
5. Set AXD Configure Target. Open AXD, and select **Options->Configure Target,** and to choose Multi-ICE as target. If Multi-ICE is not listed in the **Target Environment**, we must add it the **Target Environment.**
6. 

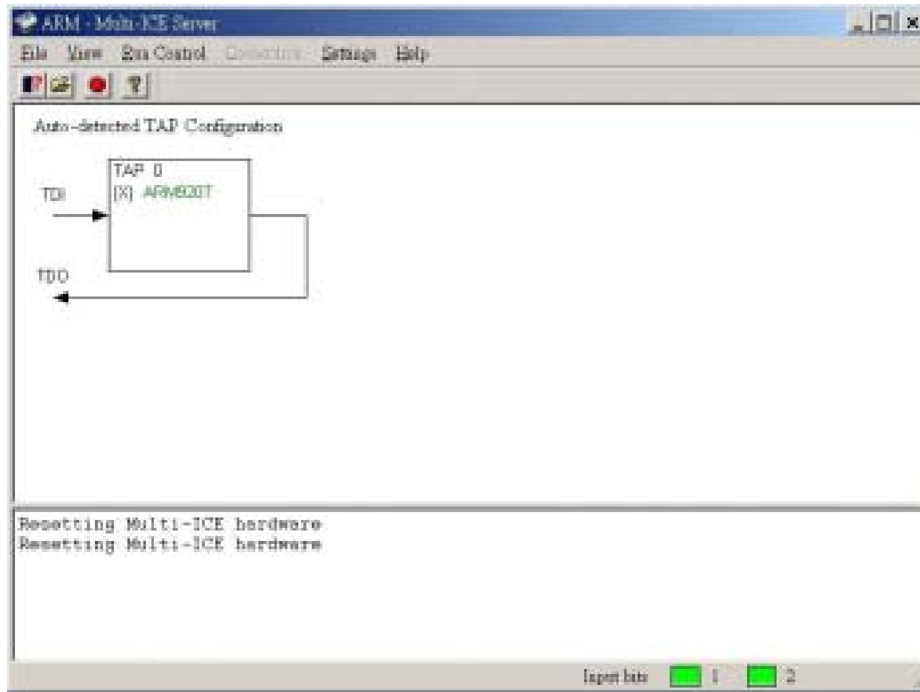


Figure 6 Multi-ICE server window configured for an ARM920T

----------Note----------
You can select **Settings** -> **Start-up Options** and click **Auto-Configure** in the **Start-up Configuration box**. This option automatically creates a configuration files naming all devices found as described in Automatic device configuration when you start Multi-ICE server next time.

Perform a variety of debugging tasks

1. Start CodeWarrior IDE.
2. Select **File** > **New** to create a new project.
   (1) Select **ARM Executable Image** under the Project stationary.
   (2) Click the **Set…** button next to the Location field.
   (3) Navigate to the directory where you want to save the project and enter a project name, for example My_Project. This lab uses this directory C:\My Documents\ARMLAB\<your id no.>\My_Project

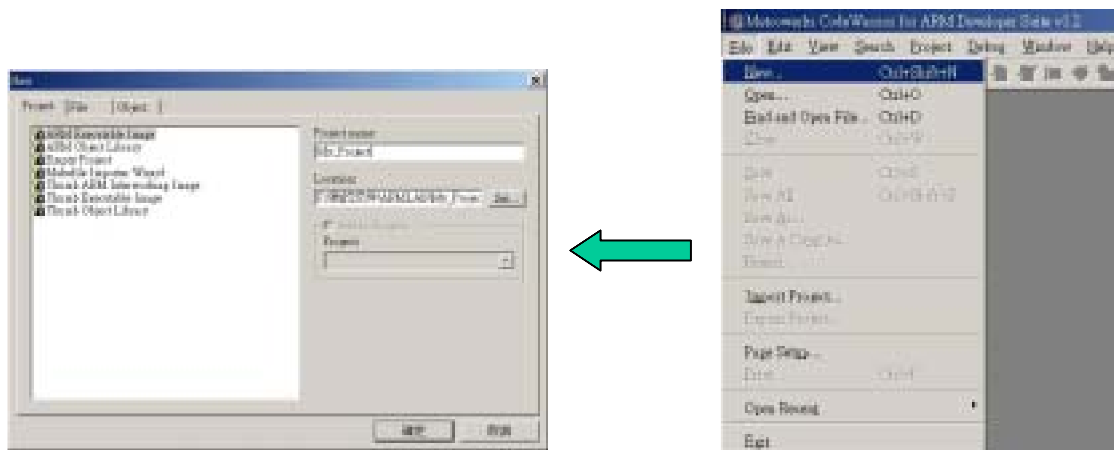


Figure 7 New dialog

3. Adding source files to the project.
   (1) Copy files dhry.h, dhry_1.c, and dhry_2.c to your My_Project directory.
   (2) Select **Project** > **Add Files…**
   (3) Navigate to your My_Project directory and Shift-click on dhry_1.c and dhry_2.c.
   (4) Click **Open**. Then Add all files to targets.

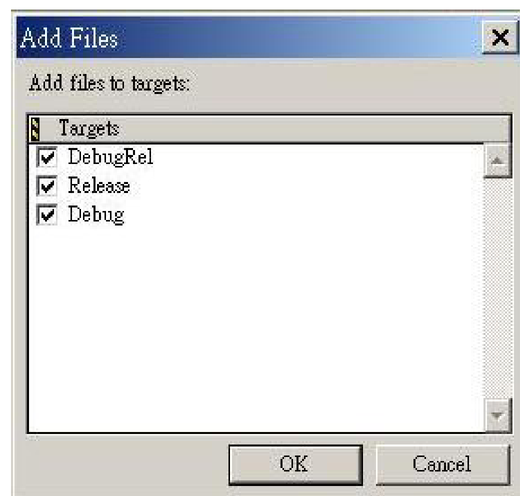


Figure 8 Add files to target dialog box

4. Configuring the project build targets
   (1) Click the DebugRel Settings button. A DebugRel Settings window appears.

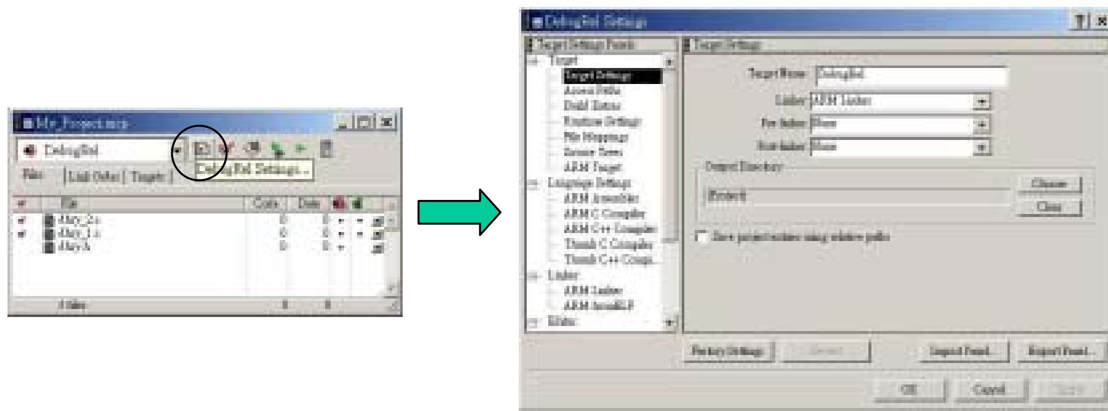


Figure 9 DebugRel Settings

(2) Click the **ARM C Compiler** entry in the Target Settings Panels.
(3) Click **Preprocessor** tab. Type MSC_CLOCK into the text field beneath the List of #DEFINEs and click **Add** to define the MSC_CLOCK. The Equivalent Command Line text box displays the result.
(4) Click **Apply** to apply your changes.

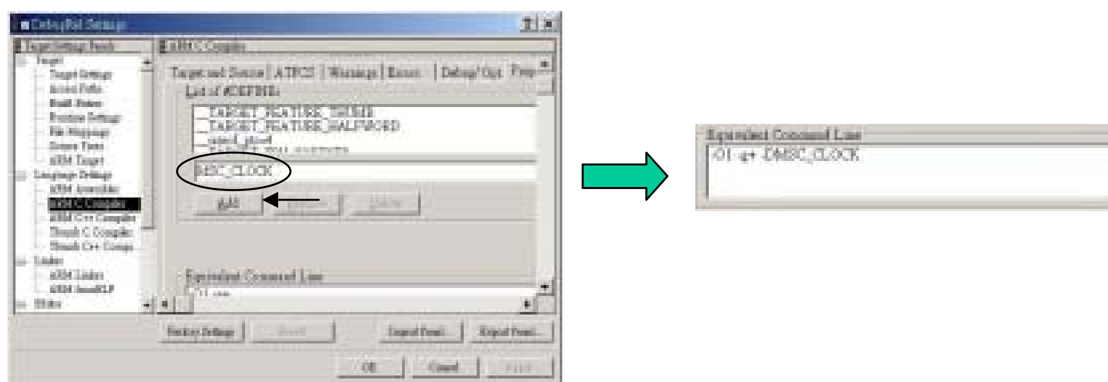


Figure 10 MSC_CLOCK setting

(5) Click on the **Debug/Opt**. Select the **For time**. Click **Apply** to apply your settings.

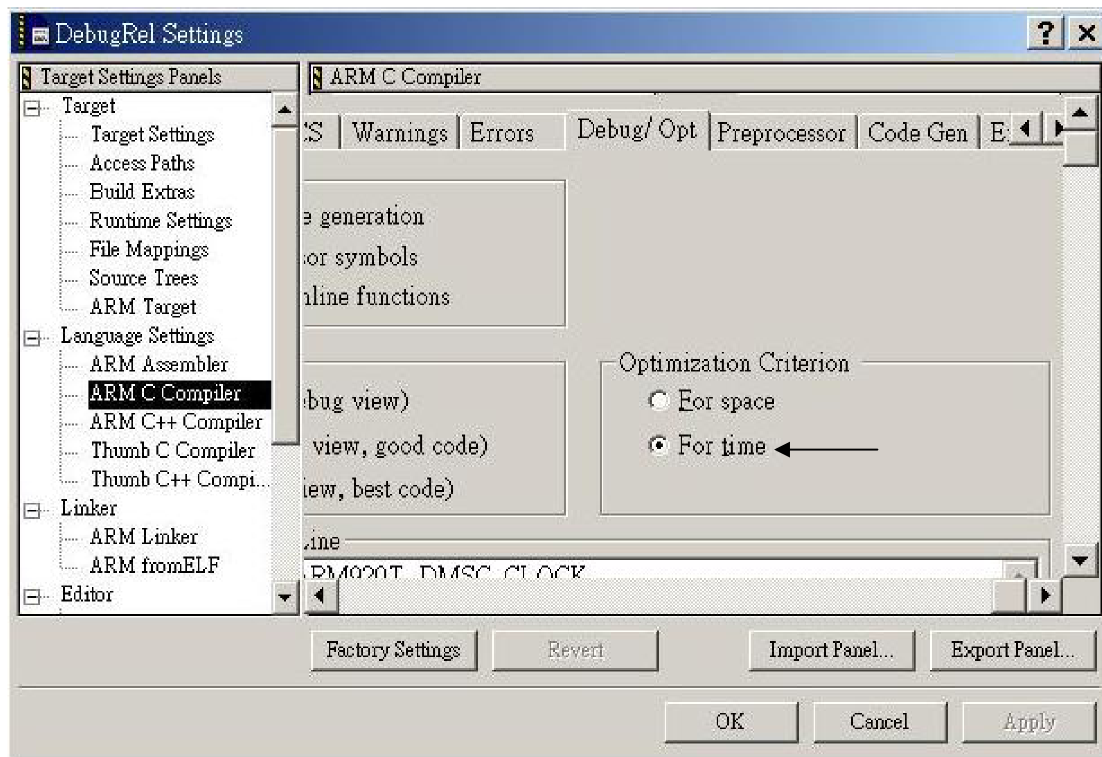


Figure 11 ARM C Compiler Panel

(6) Click the **Current Target** drop-down list and select **Release and debug** form.

(7) Apply the steps you followed above to define MSC_CLOCK the Release and debug form build target.

5. Building the project

(1) Hit the **Make** button to compile and link the project.

(2) A compiling and linking status windows would appear to indicate making progress

(3) After finishing compiling and linking, a result message windows would appear. Check for errors and warnings

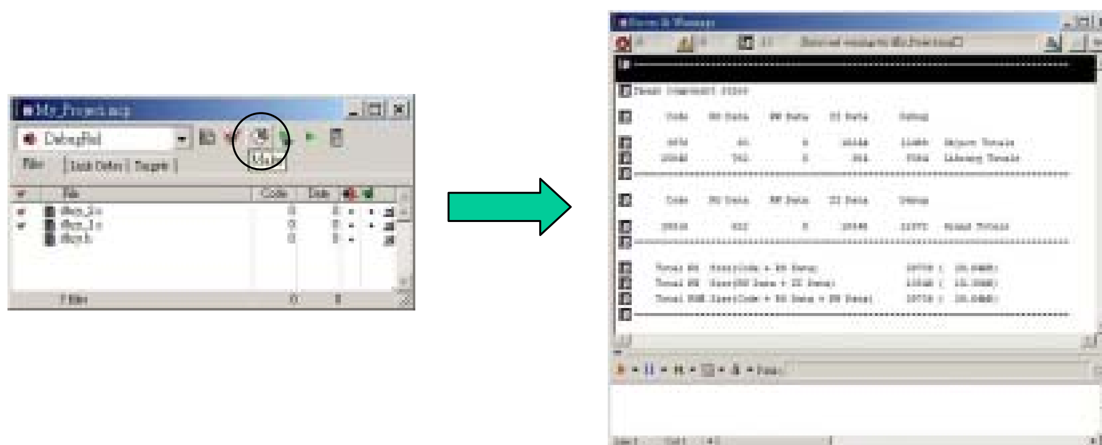


Figure 12 Make the Project

6. Debugging the project
   (1) Hit the **Debug** button to call the AXD debugger to debug and run images built from the CodeWarrior IDE. A blue arrow indicates the current execution point.

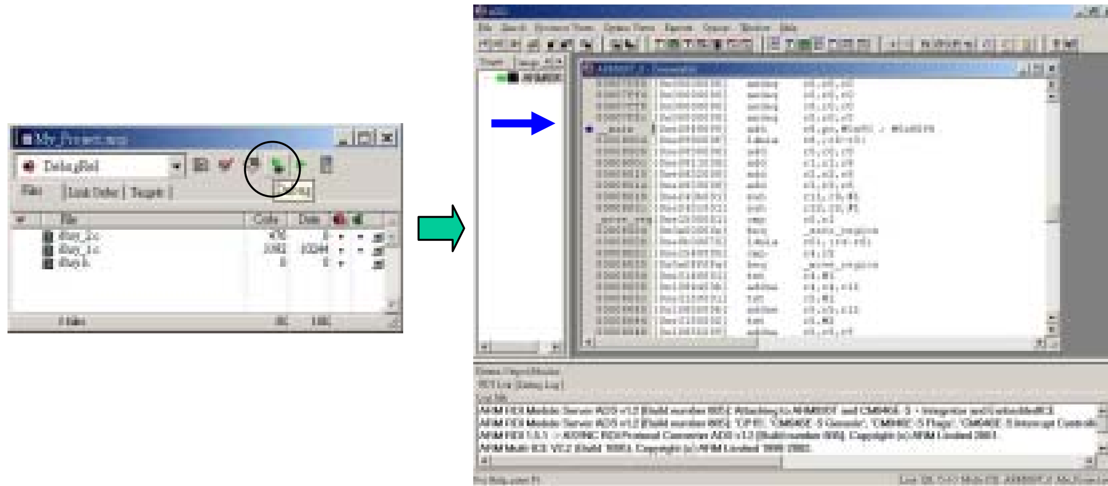


Figure 13 Debug the Project

   (2) Select **Execute** > **Go**. Execution stops at the beginning of the function main(), where a breakpoint is set by default.

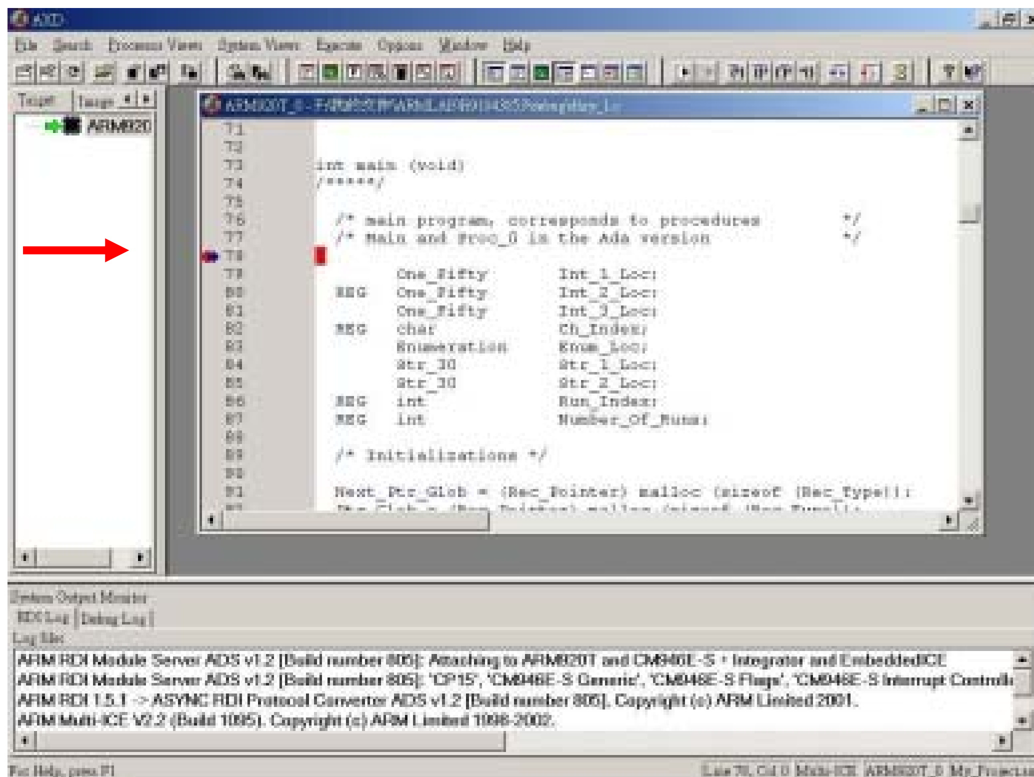


Figure 14 Breakpoint set inside loop

(3) Select **Execute** > **Go**. You are prompted, in the Console processor view, form the number of runs through the benchmark that you want performed. Enter 8000.

Figure 15 Input benchmark

(4) Select **File** > **Reload Current Image** to repeat the execution of the program.

7. Setting a breakpoint
(1) Select **File** > **Reload current Image**.
(2) Select **Execute** > **Go** to reach the first breakpoint, set by default at the beginning of the function main().
(3) Right-click on line 150 to position the cursor there and display the pop-up menu and select **Toggle Breakpoint**. You have set a second breakpoint.

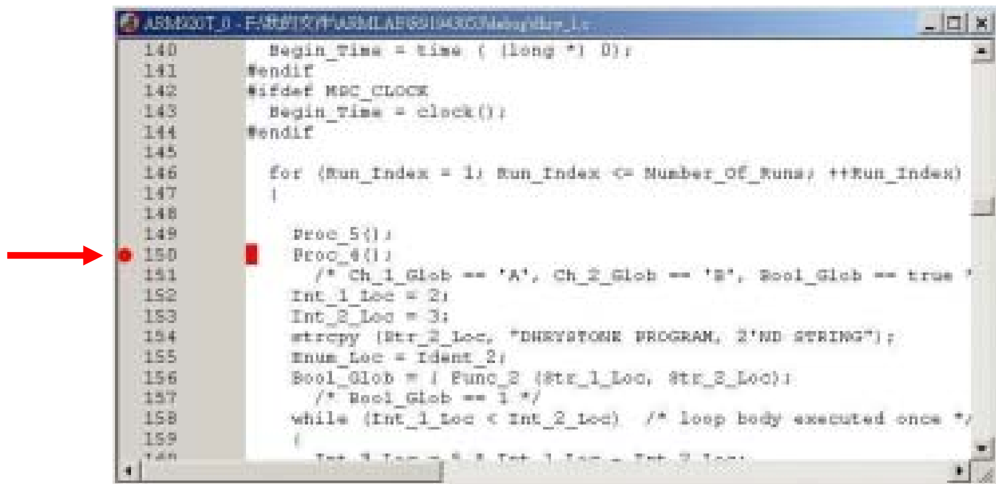


Figure 16 Set Breakpoint

(4) Select **System Views** > **Breakpoints**. The breakpoint pane is displayed.

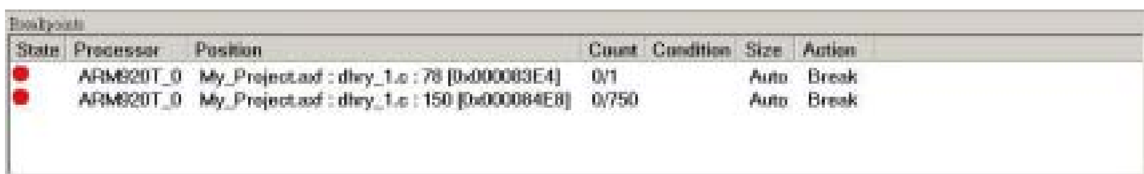


Figure 17 Breakpoints pane

(5) Right-click on the second breakpoint and select **Properties**. Enter 750 in the out of…field in the Condition group.

(6) Click **OK**.

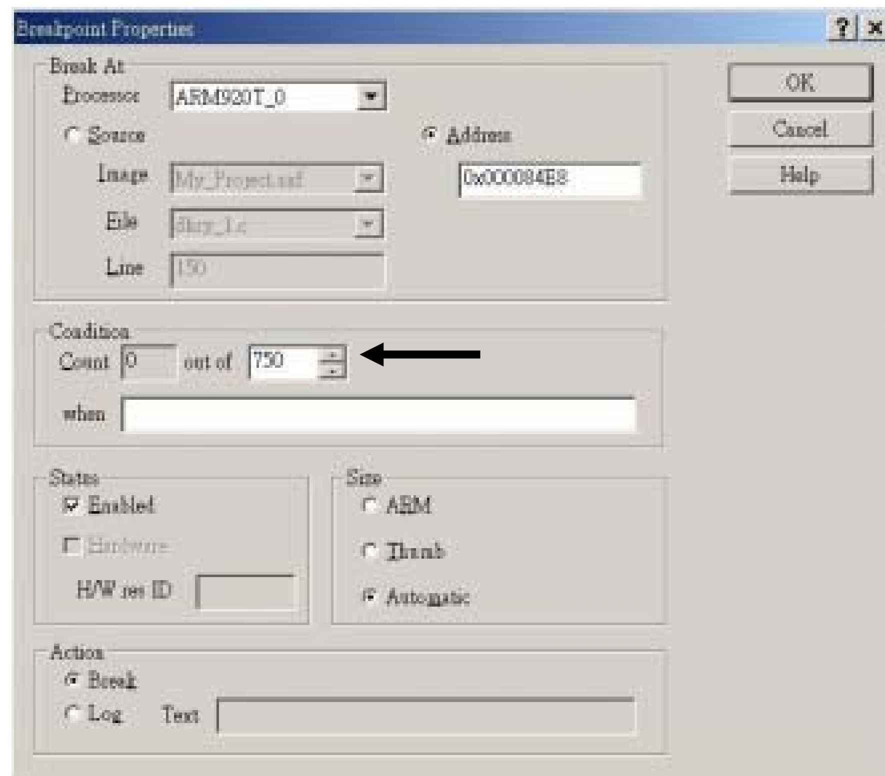


Figure 18 Setting breakpoint details

(7) Press F5 to resume execution, and enter the smaller number of 5000 this time for the number of runs required. Execution stops the 750th time your new breakpoint is reached.

(8) Select **Processor Views** > **Variables**. Click the Local tab and look for the Run_Index variable. Right-click on the variable and select **Format** > **Decimal** and the value is now displayed as 750 (decimal).

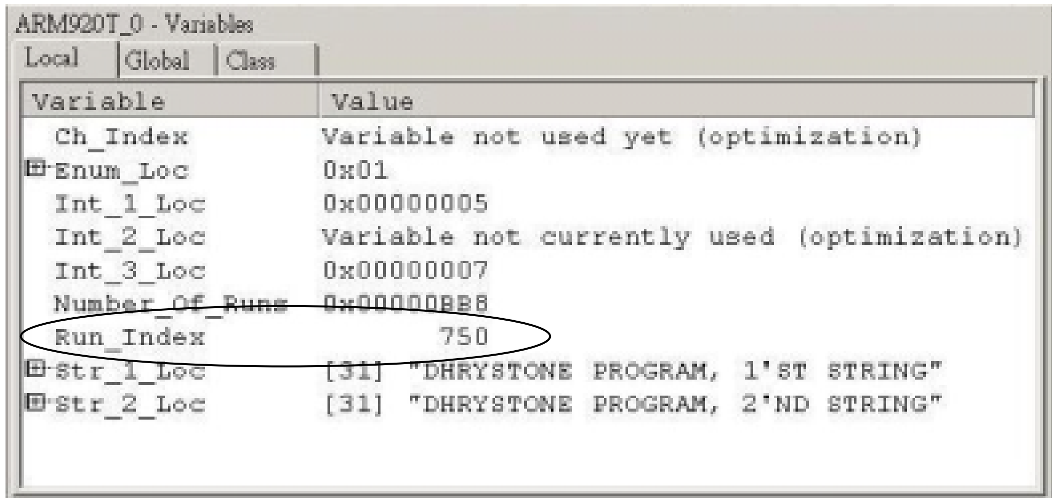


Figure 19 Examining the contents of variable

(9) Press F5 to resume execution, and the value of the Run_Index local variable changes to 1500.

(10) Close down the Breakpoints system view.

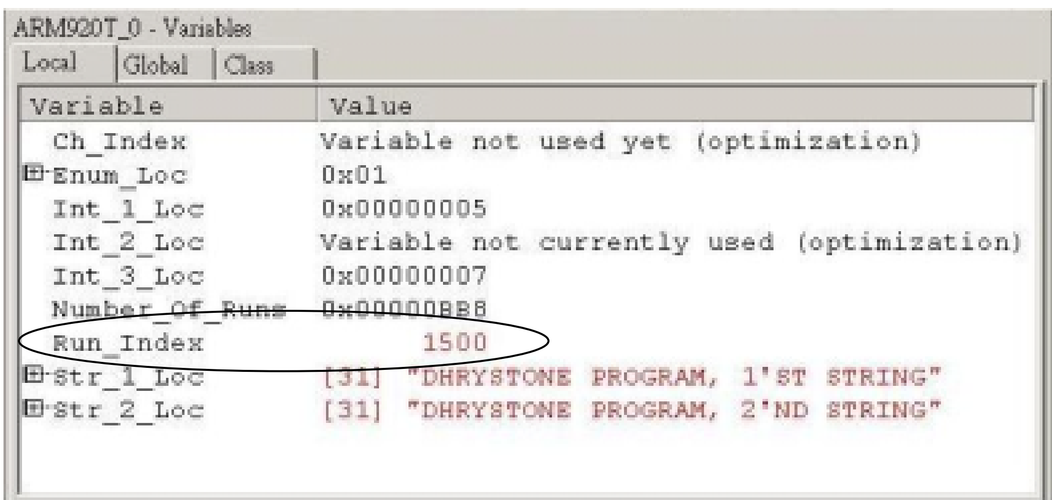


Figure 20 Examining the contents of variable

8. Setting a watchpoint

(1) Select **File** > **Reload current Image**.

(2) Select **Execute** > **Go** to reach the first breakpoint, set by default at the beginning of the function main().

(3) Select **Execute > Go** to continue execution.

(4) Enter 770 when you are prompted for the number of runs to execute. Execution continues until it reached the breakpoint at line 150 for 750th time. This is the breakpoint you defined in *Setting a breakpoint* in step 7.

(5) Select **System Views > Watchpoints**, right-click in the Watchpoints system view, and select **Add** to display the Watchpoint Properties

dialog.

(6) Set Run_Index in the Watch group and set the out of…field in the Condition group to value of 6.

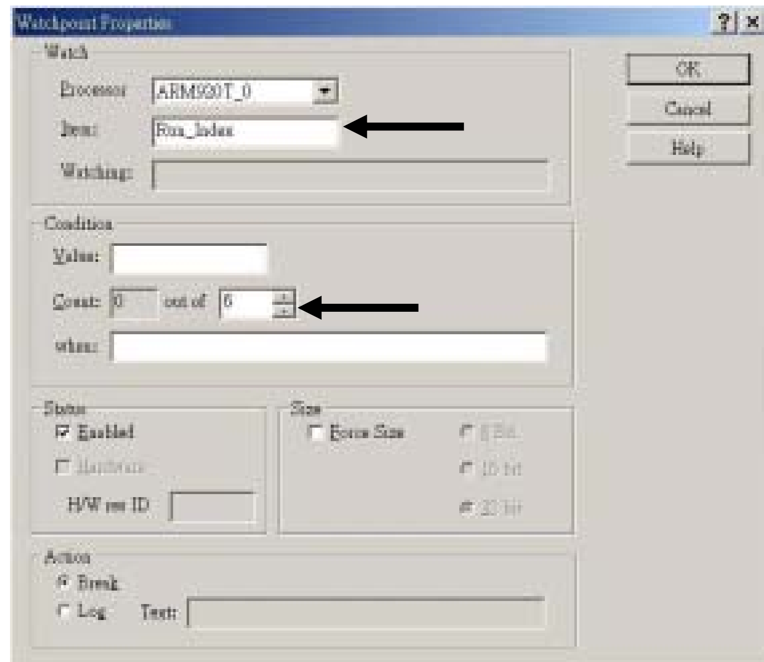


Figure 21 Setting a watchpoint

(7) Select **Processor Views > Variables**. Click the Local tab and look for the Run_Index variable.

(8) Press F5 to resume execution. Soon the value of the Run_Index local variable changes to 756.

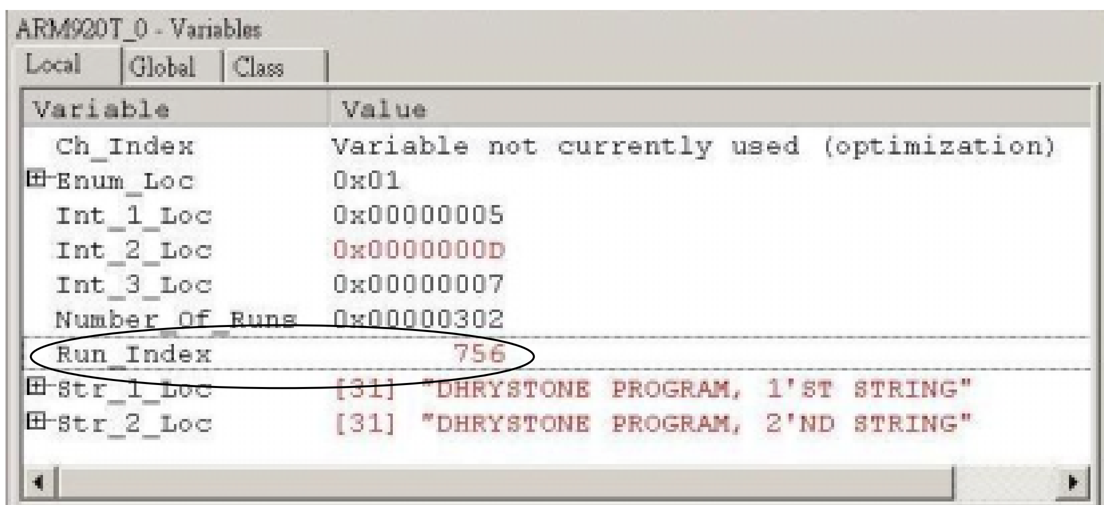


Figure 22 Examining the contents of variable

(9) Delete the Watchpoint and breakpoint you set up for this example, by

right-clicking on its line in the Watchpoints and breakpoints window and select Delete from the pop-up menu, then close down them.

9. Examining the contents of variables
   Two methods of examining the contents of variables are described:
   (1) Contents of variables (variable processor view):
   This method is simpler and shows the contents of the specified variables.
   (2) Addresses and contents of variables (watch processor view):
   This method shows the addresses of the variables as well as their contents.

10. Contents of variables
   (1) Select **File** > **Reload current Image**.
   (2) Select **Execute** > **Go** to reach the first breakpoint, set by default at the beginning of the function main().
   (3) Select **Execute > Go** to continue execution.
   (4) Select **Processor Views** > **Variables**
   (5) Select **Properties** > **Dec** and click **OK**. The display is now similar to that shown below.

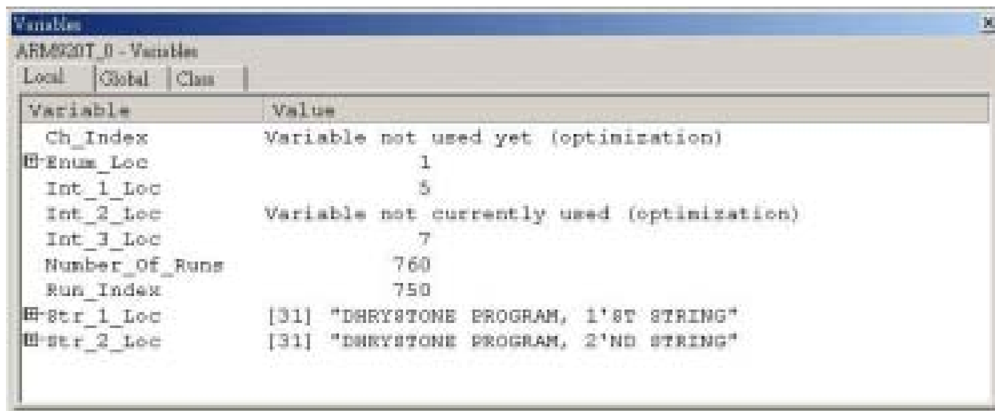


Figure 23 Examining the contents of variable

11. Addresses and contents of variables
   (1) Select **File** > **Reload current Image**.
   (2) Select **Execute** > **Go** to reach the first breakpoint, set by default at the beginning of the function main().
   (3) Select **Execute > Go** to continue execution.
   (4) Enter 760 when you are prompted for the number of runs to execute. Execution continues until it reached the breakpoint at line 150 for 750th time. This is the breakpoint you defined in *Setting a breakpoint*.
   (5) Select **Processor Views** > **Watch**.
   (6) Right-click in the window, and select **Add Watch** from the pop-up menu.
   (7) Enter the first expression in the expression by typing &Enum_Loc.
   (8) Press the Return key or clock on the **Evaluate** button.
   (9) Enter, in the similar way:

&Int_1_Loc
&Int_3_Loc
Run_Index

(10) Ensure that **Proc** and **Tab1** are selected, then Click the **Add to View** button and the **Close** button

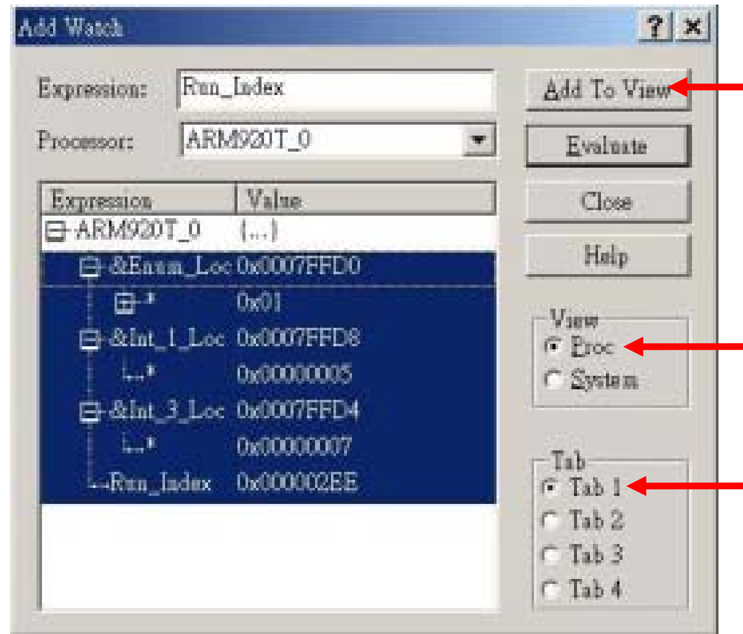


Figure 24 Specifying variable to watch

(11) The variables you have specified are now displayed in the Watch processor view, and if you expand the lines you can see both the addresses and the contents of the variables.

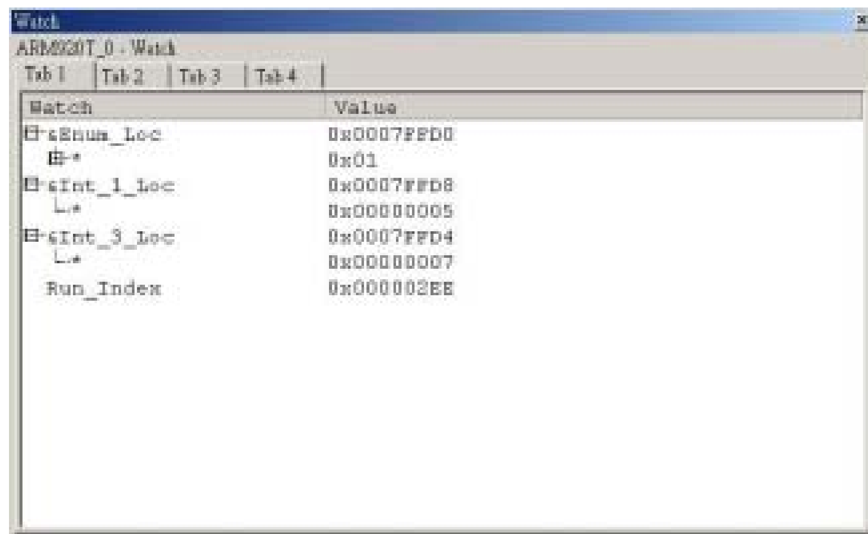


Figure 25Examining contents of registers

(12) Press F10. The program executes a single instruction and stops.
(13) Press F10 a few more times. As you execute the program, one instruction at a time, you can see the values of several of the registers change.
(14) Press F5 to allow the program to complete its execution, the close down the registers processor view.

12. Examining the contents of registers
(1) Select **File** > **Reload current Image**.
(2) Select **Execute** > **Go** to reach the first breakpoint, set by default at the beginning of the function main().
(3) Select **Processor** > **Registers**.

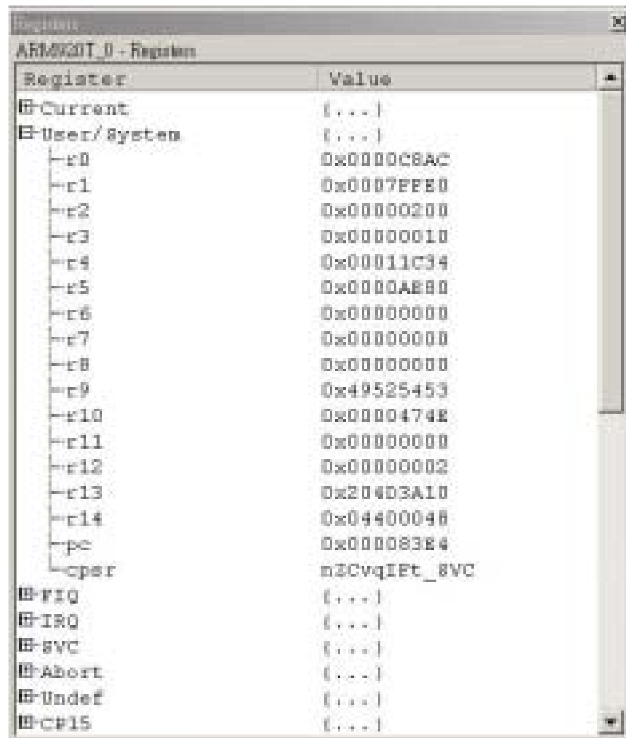


Figure 26 Examining contents of registers

(4) Press F10. The program executes a single instruction and stops.
(5) Press F10 a few more times. As you execute the program, one instruction at a time, you can see the values of several of the registers change.
(6) Press F5 to allow the program to complete its execution, the close down the registers processor view.

13. Examining the contents of memory
(1) Select **File** > **Reload current Image**.
(2) Select **Execute** > **Go** to reach the first breakpoint, set by default at the beginning of the function main().
(3) Select **Execute > Go** to continue execution.
(4) Enter 760 when you are prompted for the number of runs to execute.

Execution continues until it reached the breakpoint at line 150 for 750th time. This is the breakpoint you defined in *Setting a breakpoint*.

(5) Select **Processor Views** > **Memory**.

(6) Set the Start address value to 0x07FFFF00.

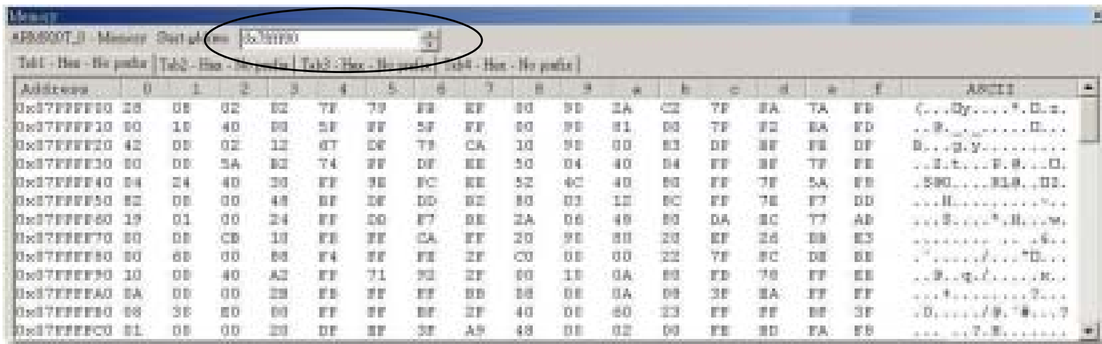


Figure 27 Examining contents of memory

(7) Press F10. The program executes a single instruction and stops.

(8) Press F10 a few more times. As you execute the program, one instruction at a time, you can see the values of several of the memory addresses change.

(9) Press F5 to allow the program to complete its execution, the close down the Memory processor view.

14. Locating and changing values and verifying changes

(1) Select **File** > **Reload current Image**.

(2) Select **Execute** > **Go** to reach the first breakpoint, set by default at the beginning of the function main().

(3) Select **Search** > **Memory**.

(4) Enter 2’ND in the Search for field, set the In range and to addresses 0x0 and 0xFFFF, and select **ASCII** for Search string type.

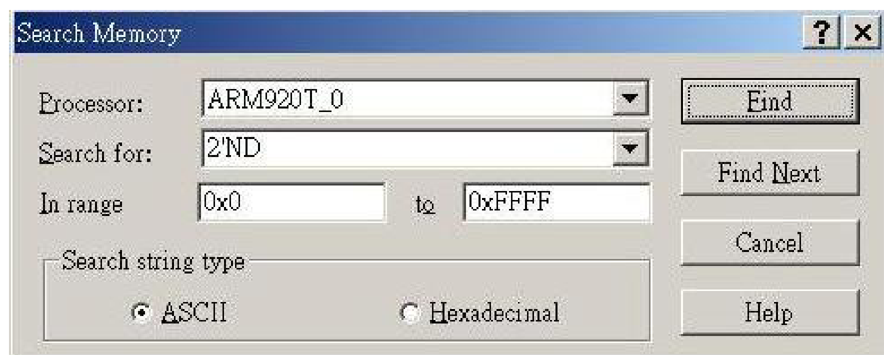


Figure 28 Searching for a string in memory

(5) Click the Cancel button to close the Search Memory dialog.

(6) The four hexadecimal values highlighted are 32 27 4E 44.

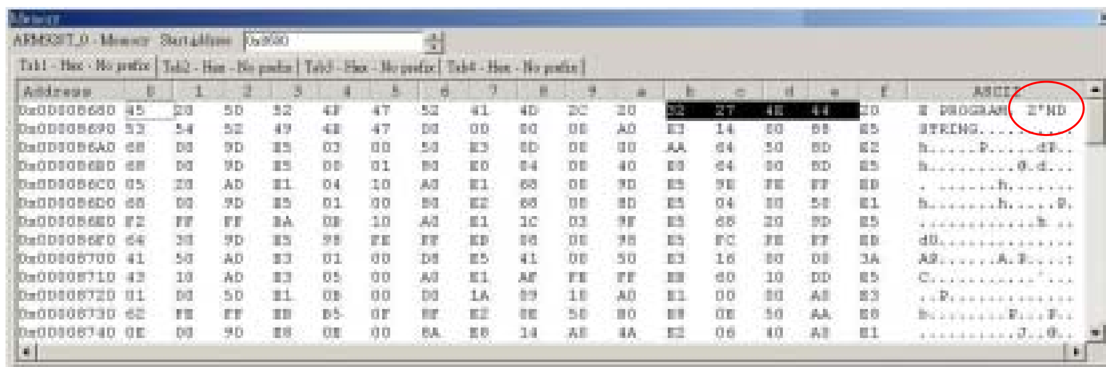


Figure 29 String in memory

(7) Double-click on the value 32 and type 0x4E and press Return
(8) Double-click on the value 27 and type “o and press Return
(9) Double-click on the value 4E and type 46 and press Return
(10) Double-click on the value 44 and type o62 and press Return

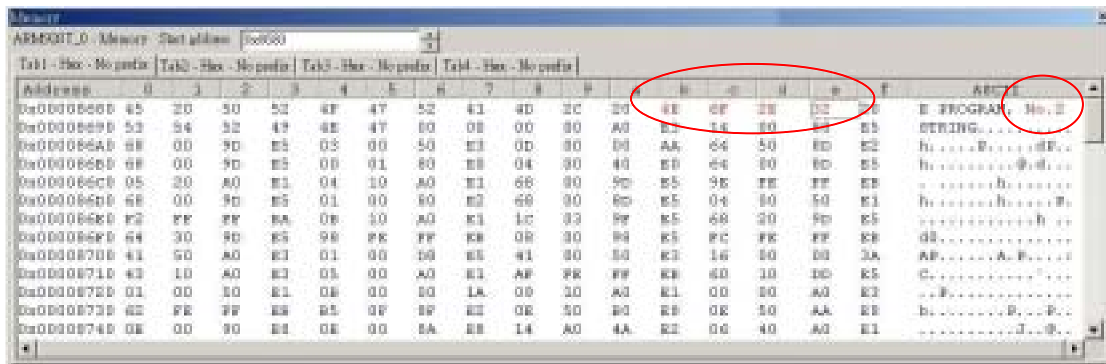


Figure 30 Changing contents of memory

(11) Press F5 to continue execution, and enter a value of 100 when you are prompted in the Console processor view for the number of runs to perform.

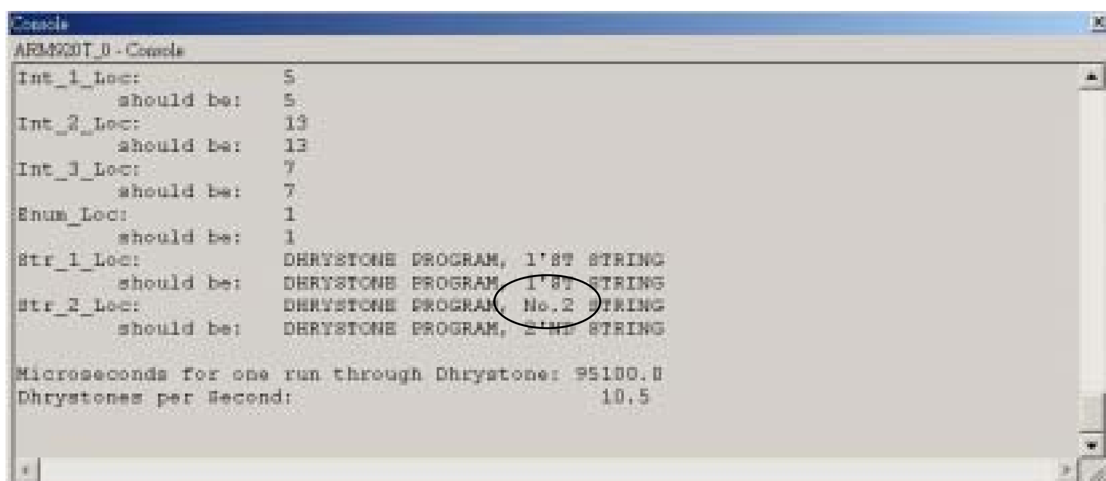


Figure 31 Contents display in the console

----------Note----------
In this example, the change you made was not permanent, because you did not alter the source code or the executable image stored in a disk file. You altered only the temporary copy of the image in the target memory.

9.4. 實驗要求

Write a lotto program that generates N sets of number. The user can specify:
1. Number of the set: N.
2. The numbers must be included in these N sets of number
3. The numbers must not be included in these N sets of number

Note: Numbers cannot be duplicated within the same set of number. In addition, output numbers within the same set in ascending order.

This exercise lets you familiar with the debugging skills and ARM CodeWarrior Editor when you try to build the program correct.

9.5. 問題與討論

What's different between ARMulator and MultiICE that we do the debugging task.

9.6. 參考文件及網頁

- Multi-ICE [DUI_0048F_MICE2_2_UG]
- AXD and armsd Debuggers Guide [DUI_0066D_AXDDG_2_UG]
- Getting Started Guide [DUI_0064D_GSG_UG]