# Contents

# 1. Code Development

## 1.1. 實驗目的

This Lab first describes how to create an application using ARM Developer Suite (ADS). Then several examples are used to illustrate how to change between ARM state and Thumb state when writing code for different instruction sets.

## 1.2. 實驗原理

You can mix ARM and Thumb code as you wish, provided that the code conforms to the requirements of the ARM/Thumb Procedure Call Standard. The ARM compilers always create code that conforms to this standard. If you are writing ARM assembly language modules you must ensure that your code conforms. The ARM linker detects when an ARM function is being called from Thumb state, or a Thumb function is being called from ARM state. The ARM linker alters call and return instructions, or inserts small code sections called veneers, to change processor state as necessary.
If you are linking several source files together, all your files must use compatible ATPCS options. If incompatible options are detected, the linker will produce an error message.

In order to branch to Thumb state, the bit 0 in the branch target address is set, this changes the processor state after branching. The bit 5 in the CPSR (t bit) would change to 1 indicating it's in Thumb state.

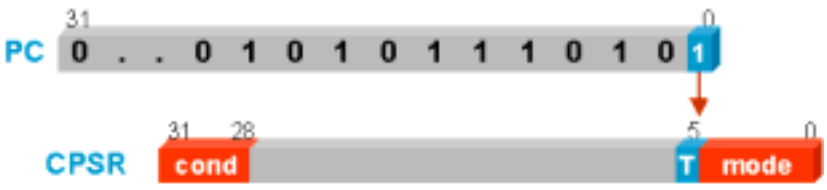


**Figure 1. ARM/Thumb branching.**

## 1.3. 引導實驗

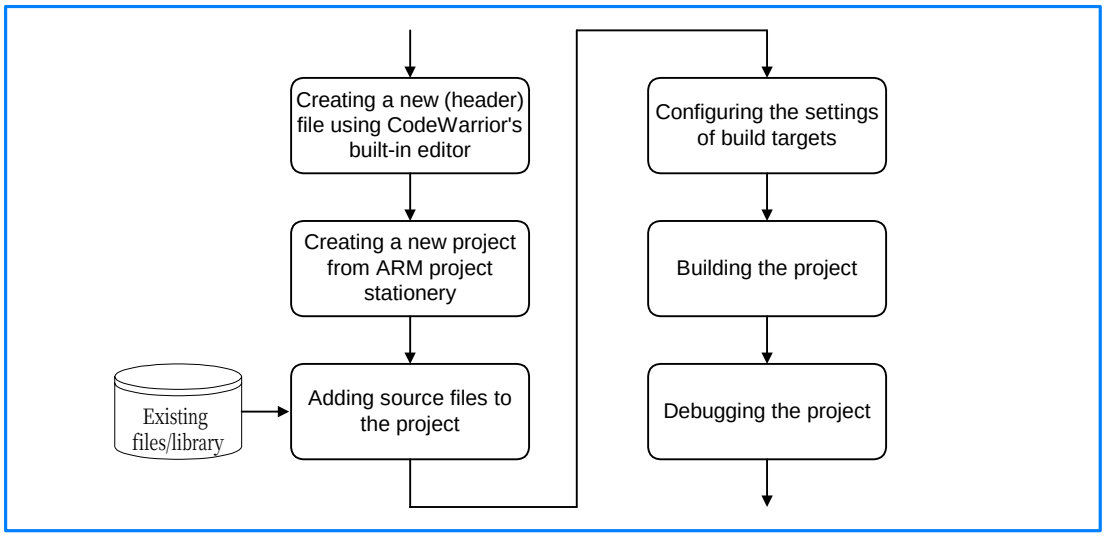


**Figure 2. Flow diagram of this Lab.**

The CodeWarrior IDE provides a graphical user interface to configure the ARM tools to compile, assemble, and link your project code. It enables you to organize source code files, library files, other files, and configuration settings into a ***project***. Each project enables you to create and manage multiple ***build targets***. A build target is the collection of build settings and files that determines the output, which is created when you build your project. Build targets can share files in the same project, while using their own build settings.

CodeWarrior for the ARM Developer Suite provides preconfigured project stationery files for common project types, including:

- ARM Executable Image
- ARM Object Library
- Thumb Executable Image
- Thumb Object Library
- Thumb/ARM Interworking Image.

You can use the project stationery as a template when you create your own projects. The non-interworking ARM project stationery files define three build targets. The interworking (i.e., Thumb/ARM Interworking Image) project stationery defines an additional three build targets to compile Thumb-targeted code. The basic build targets for each of the stationery projects are:

- **Debug**: This build target is configured to build output binaries that are fully debuggable, at the expense of optimization.
- **Release**: This build target is configured to build output binaries that are fully optimized, at the expense of debug information (i.e., no source level debug information, but full optimization).
- **DebugRel**: This build target is configured to build output binaries that provide adequate optimization, and give a good debug view. This is a trade-off between Debug and Release.

### 1.3.1. 實驗步驟: Basic Software Development Flow

## Step 1: Creating a new header file using CodeWarrior's built-in editor

1. Select **Programs → ARM Developer Suite → CodeWarrior for ARM Developer Suite** from the Windows **Start** menu to start the CodeWarrior IDE.
2. Select File → New Text File (or Ctrl+N).
3. Enter the following C text. Make sure that "#defin" instead of "#define" is typed.

```
1  /* This preprocessor results in the C library function
2     clock () begin used for timing measurements.*/
3
4  #defin MSC_CLOCK
```

**Figure 3. Simple C header file.**

1. Select **File → Save As (or Ctrl+S)**.
   4.1 Navigate the directory structure to your working directory, e.g., c:\ARMSoC\Lab_01\.
   4.3 And enter the filename dhry_def.h
2. Click **Save.** Click **Yes** to overwrite (if necessary).

## Step 2: Creating a new project from ARM project stationery

1. Select **File →New… (or Ctrl+Shift+N)**. A New dialog is displayed (Figure 4).
2. Ensure that the **Project** tab in Figure 4 is selected. The available ARM project stationery is listed in the left of the dialog, together with the Empty Project stationery and the Makefile Importer Wizard.

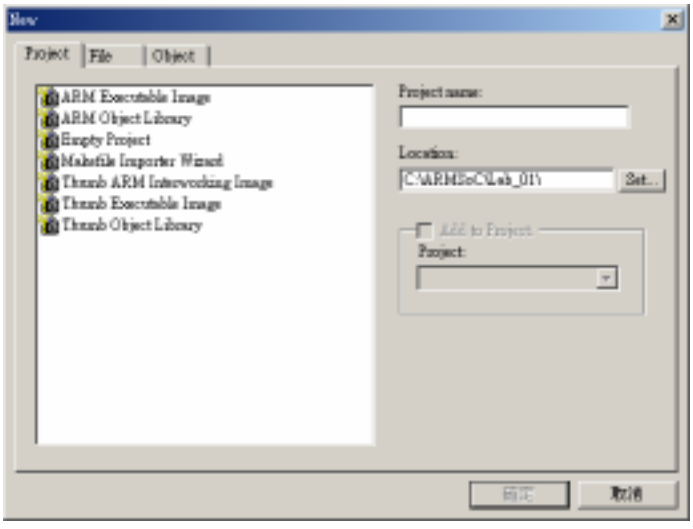


**Figure 4. New dialog.**

3. Select **ARM Executable Image** from the list of project stationery.
4. Set the directory where you want to save the project in the **Location** field or click the **Set…** button (a **Create New Project** dialog is displayed) next to the **Location** field to navigate to the directory c:\ARMSoC\Lab_01\.
   Enter a project name, for example My_Project. A result is shown in Figure 5.

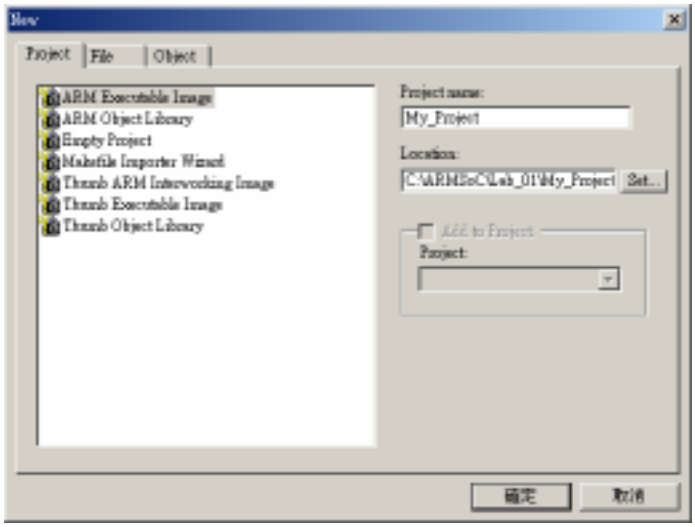


**Figure 5. Setting project name and location path.**

5. Click **OK**. The CodeWarrior IDE creates a new project based on the ARM Executable Image project stationery, and displays a new project window with the **Files** tab is highlighted and DebugRel is selected as the build target by default (Figure 6). Other build targets can be selected by clicking on the drop-down box. It is the DebugRel variant that we shall use for the remainder of this Lab.

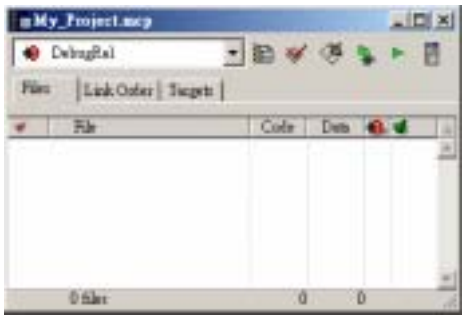


**Figure 6. New Project.**

- Close ADS
- Double click on My_project.mcp, the ADS IDE starts and displays the project window as it did at step Five (Figure 6).

## Step 3: Adding source files to the project

1. Ensure that the **project** (titled as My_Project.mcp in this example) window is the active window.
2. Select **Project →Add Files…(**Figure 7**)**. A **Select files to add…** dialog is displayed. Navigate to the **dhryansi** directory in the **install_directory\ Examples** (e.g., C:\Program Files \ARM \ADSv1_2\ Examples\dhryansi\) directory and Shift-click on **dhry_1.c** and **dhry_2.c** to select them (Figure 8).

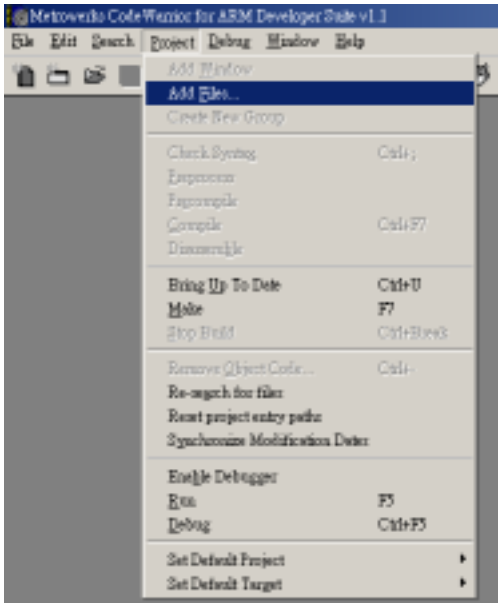


**Figure 7. Add file.**

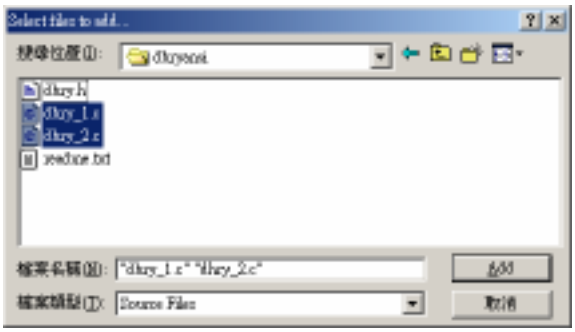


**Figure 8. Select files to add… dialog.**

3. Click **Add**. The CodeWarrior IDE displays an **Add Files** dialog (Figure 9). The dialog contains a checkbox for each build target defined in the current project. In this example, the dialog contains three checkboxes corresponding to the three build targets defined in the ARM Executable Image project stationery.

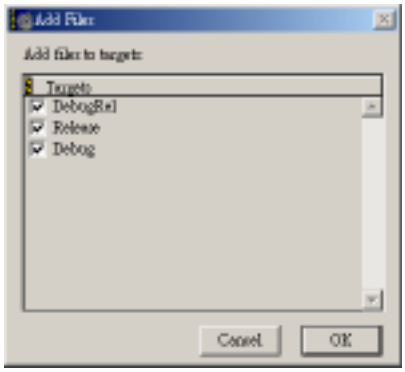


**Figure 9. Add Files.**

4. Leave all the build target checkboxes selected and **click OK**. The CodeWarrior IDE adds the source files to each target in the project and displays a Project Messages window (Figure 10) to inform you that the directory containing the source files has been added to the access paths for each build target.

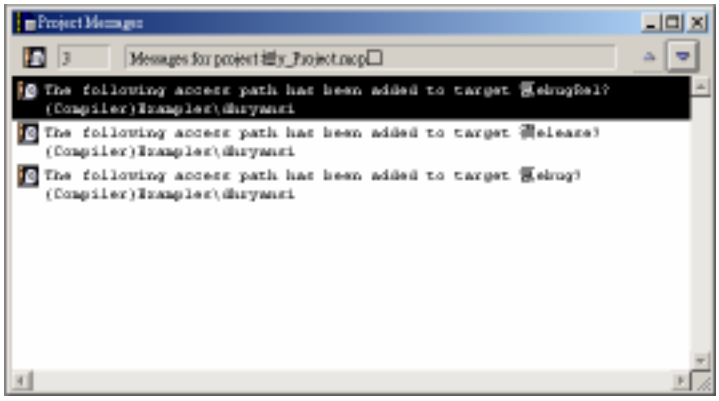


**Figure 10. Project Message window.**

----------Note----------
The access paths for each build target define the directories that will be searched for source and header files. You do not need to explicitly add the header files for the dhryansi project because the CodeWarrior IDE locates them in the newly added access path. As the message "The following access path has been added to target "DebugRel": {Compiler}Examples\dhryansi" shown in the Project Message window. However, you can add header files explicitly if you want, follow the instruction described in step 2 of Section 1.1.3.
------------------------------

Repeat sub-step 2~4 to add the dhry_def.h file you build in Step 1.

5. Ensure that the **Files** tab is selected in the **project window**. The project window displays all the source files in the project. (Figure 11).

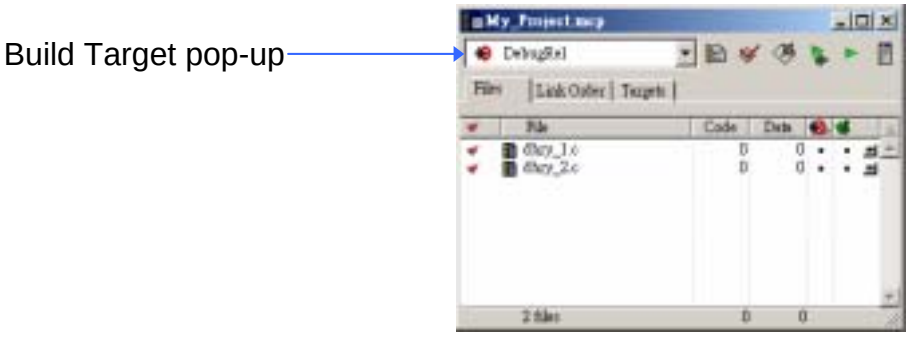


**Figure 11. Source files in Files view.**

6. Select dhry_def.h from the project window and click right button on it. Select Preprocess. After preprocessing this header file, two windows appear. At Error and Warring window, two messages are shown as below:

   Error : (Serious) C2858E: Unknown directive: #defin
   dhry_def.h line 4

   C:\ARMSoC\Lab_01\dhry_def.h: 0 warnings, 0 errors, 1 serious error

The Lower section of the window contains a section of the code that caused the first error message.

**Figure 12. File handling in Project Window.**

----------Note----------

- Source files in the project window can be edited by double clicking on their icons.
- Select the file in the project window and press “delete” will remove the file added in step 2~4 form the project.
- If you close CodeWarrior IDE after the fifth step, the files added to My_project.mcp are automatically saved without your indication. Double click on My_project.mcp, the CodeWarrior IDE starts and displays the project window as it did on the fifth step (Figure 11).

---

7. Double click on the first error message. The editor window is opened, with

focus placed on the problem line (line 4 of dhry_def.h). You may already find out that "#defin" should be replaced with "#define". Instead of doing correctness in this way, we remove (delete) dhry_def.h from the project and supply it as a command line parameter to the C compiler, which will be described in the next section.

## Step 4: Configuring the settings of build targets for your project

Build target settings must be selected separately for each build target in your project. To set build target options for the **dhryansi** example:

1. Ensure that the **DebugRel** build target is currently selected. By default, the DebugRel build target is selected when you create a new project based on the ARM project stationery. The currently selected build target is displayed in the **Build Target** pop-up menu in the project toolbar (Figure 11).

2. Select **Edit→DebugRel Settings… (or Alt + F7)**, as shown in Figure 13. The name of this menu item (Debug, Release, or DebugRel) changes depending on the name of the currently selected **Build Target**. The CodeWarrior IDE displays the **DebugRel Settings Panel** (Figure 15). All the target-specific settings are accessible through configuration panels listed at the left of the panel. An alternative way to do this step is to hit the *Build Target Setting* button in Project Window, as displayed in Figure 14.

**Figure 13. Select DebugRel Settings**

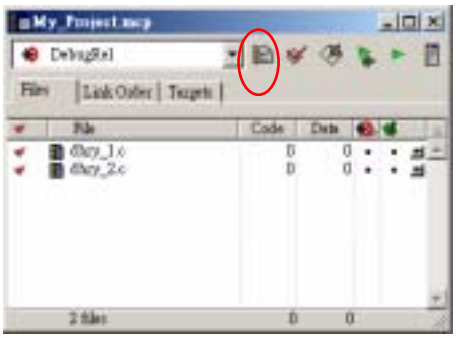

**Figure 14. Select DebugRel Settings from Project Window.**

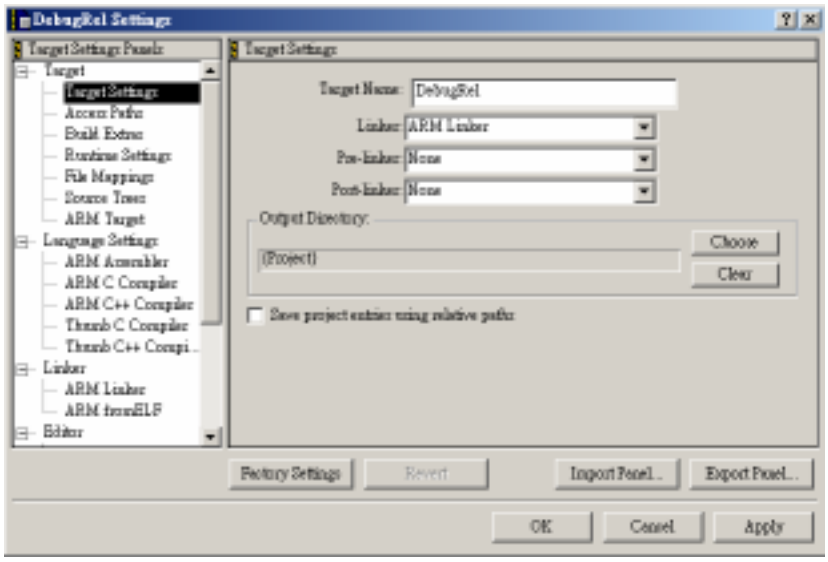


**Figure 15. DebugRel Settings.**

Click the *Access Paths* entry in the Settings Panels list. As displayed in Figure 14, the path {Compiler}Examples\dhryansi added in previous step appears in the User Path. You can add other path by clicking the add button.

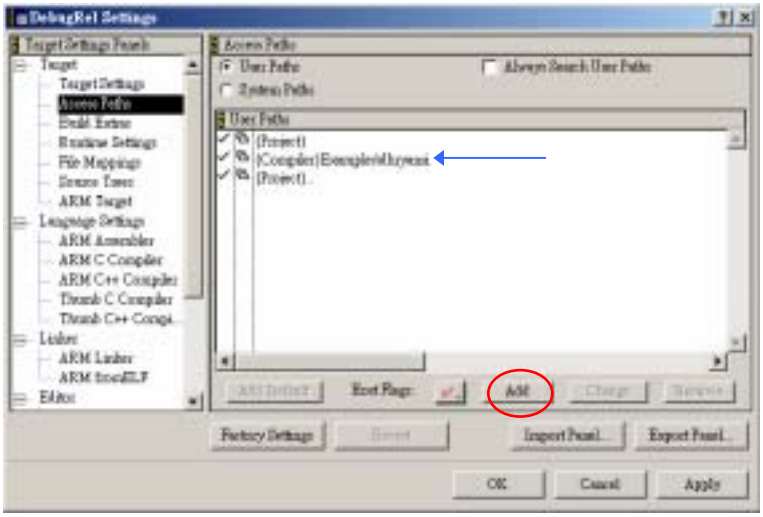


**Figure 16. Access Path configuration**

3. Click the **ARM C Compiler** entry in the Settings Panels list to display the configuration panel for the C compilers. The **Target and Source panel** is displayed. The panel consists of a number of tabbed panes containing groups of configuration options. For this example, the dhryansi source requires a *predefined macro* be set before it will compile.

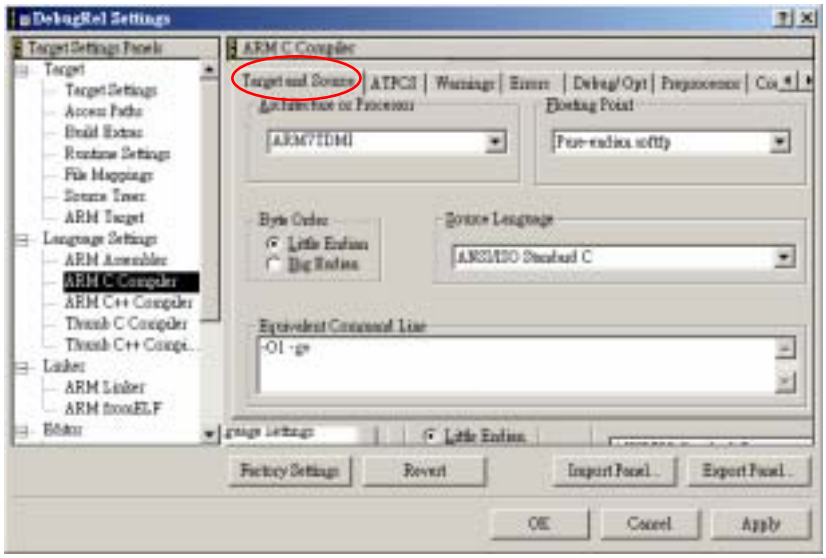


**Figure 17. ARM C compiler panel.**

4. Click the **Preprocessor tab** to display a list of predefined macros (Figure 18). Type **MSC_CLOCK** into the text field beneath the **List of #DEFINES** and **click Add** to define the MSC_CLOCK macro. In Figure 19, the CodeWarrior IDE adds MSC_CLOCK to the List of #DEFINES and the **Equivalent Command Line** text box displays the compiler command-line option required to define MSC_CLOCK.

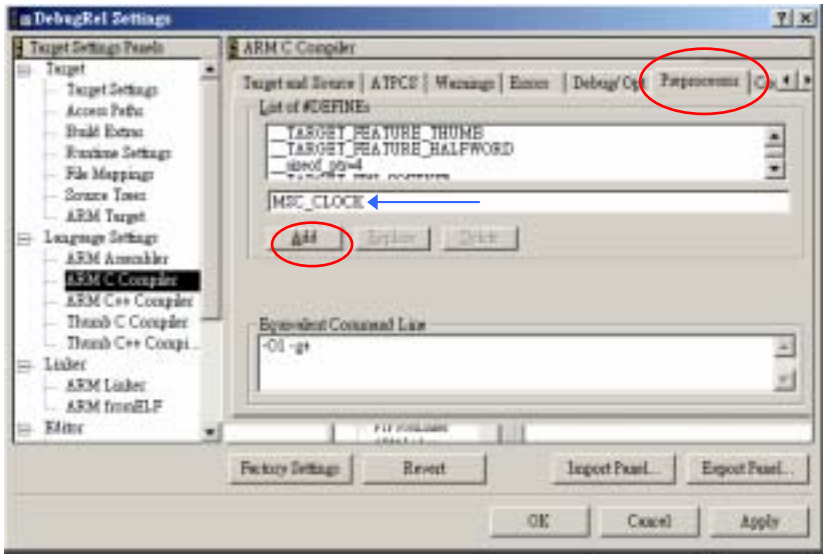


**Figure 18. ARM C compiler preprocessor panel.**

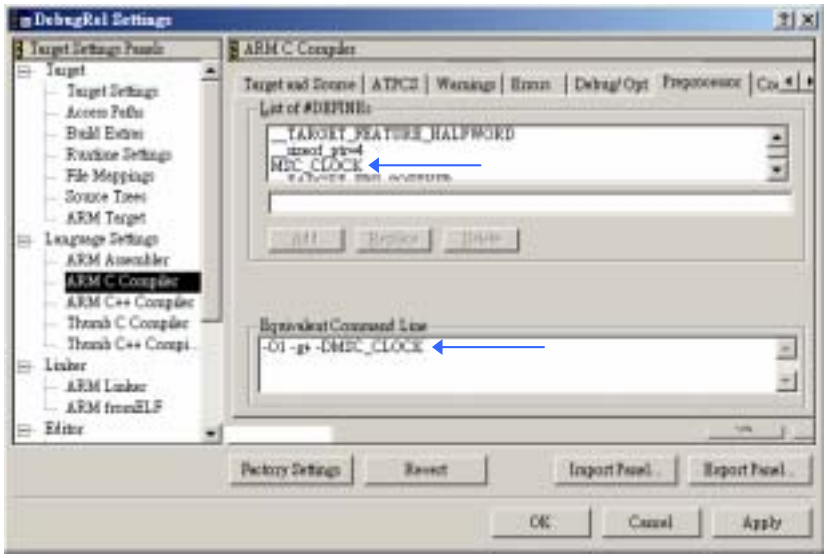


**Figure 19. MSC_CLOCK defined.**

6. **Click OK** to save your changes, and **close the DebugRel Settings panel**.

At this point you have defined the MSC_CLOCK macro for the DebugRel build target only. You must also define the MSC_CLOCK macro for the Release and Debug build targets if you want to use them. To select the **Release** build target:

1. Select ***Release*** from the ***Build Target pop-up menu*** (Figure 11) to change the current build target.
2. Apply the steps you followed above to define MSC_CLOCK for the Release build target.
3. Click on the ***Debug/Opt tab*** to display Debug and Optimization options for the Release build target. Select the ***For time*** Optimization Criterion button. The Equivalent Command Line text box reflects the change, as shown in Figure 20.


**Figure 20. Debug/Opt configuration panel.**

4. **Click Ok** to save your settings.
5. Define MSC_CLOCK in the **Debug** build target in the same way as you have for the DebugRel and Release build targets. Your project is now equivalent to the dhryansi example project (INSTALL_PATH\ADSv1_2\Examples\dhryansi**dhryansi.mcp**) supplied with the ARM Developer Suite.

----------Note----------

- Compiler options
  - -g Tells the compiler to add debug tables
  - -O1 Tells the compiler to select good optimization
  - -c Tells the compiler to compile only (not to link)
- There are configuration panels available for most of the ADS toolchain, including the linker, fromELF, and the assembler. You can use the configuration panels to specify most options available in the tools, including:
  - procedure call options
  - the structure of output images
  - the linker and postlinker to use
  - the ARM debugger to call from the CodeWarrior IDE.

  See the chapter on configuring a build target in the CodeWarrior IDE Guide for a complete description of build target options.

---

## Step 5: Building the project

The Project menu contains a number of commands to compile, or compile and link your project files. These commands apply only to the current build target. To compile and link the example project:

1. Select the build target you want to build (Figure 11). For this example, select the DebugRel build target.

2. Select **Project→Make** (or F7), as shown in Figure 21 (or the **Make** button from the Project Window, as shown in Figure 22). The CodeWarrior IDE builds the project by:
   - compiling newly added, modified, and touched source files to produce ELF object files
   - linking object files and libraries to produce an ELF image file, or a partially linked object
   - performing any postlink operations that you have defined for your build target, such as calling fromELF to convert an ELF image file to another format.

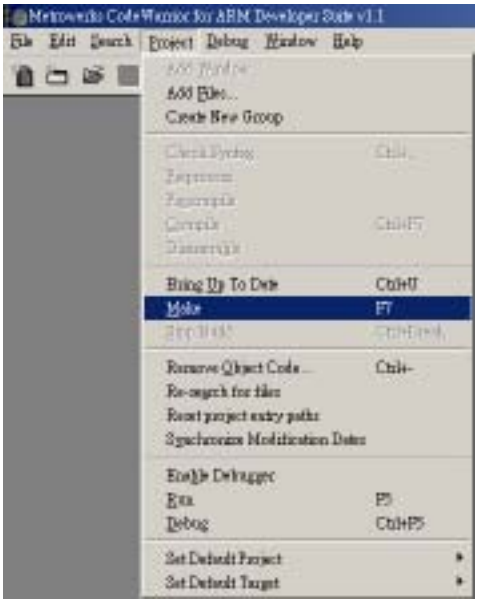


**Figure 21. Make the project.**

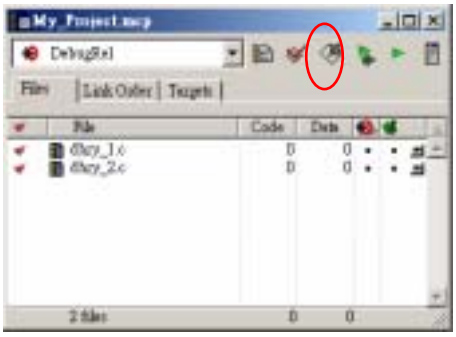


**Figure 22. Make the project from the Project Window.**

The compiler displays build information, errors, and warnings for the build in a messages window (Figure 23). The meaning of the “Image component sizes” will be explained later on.

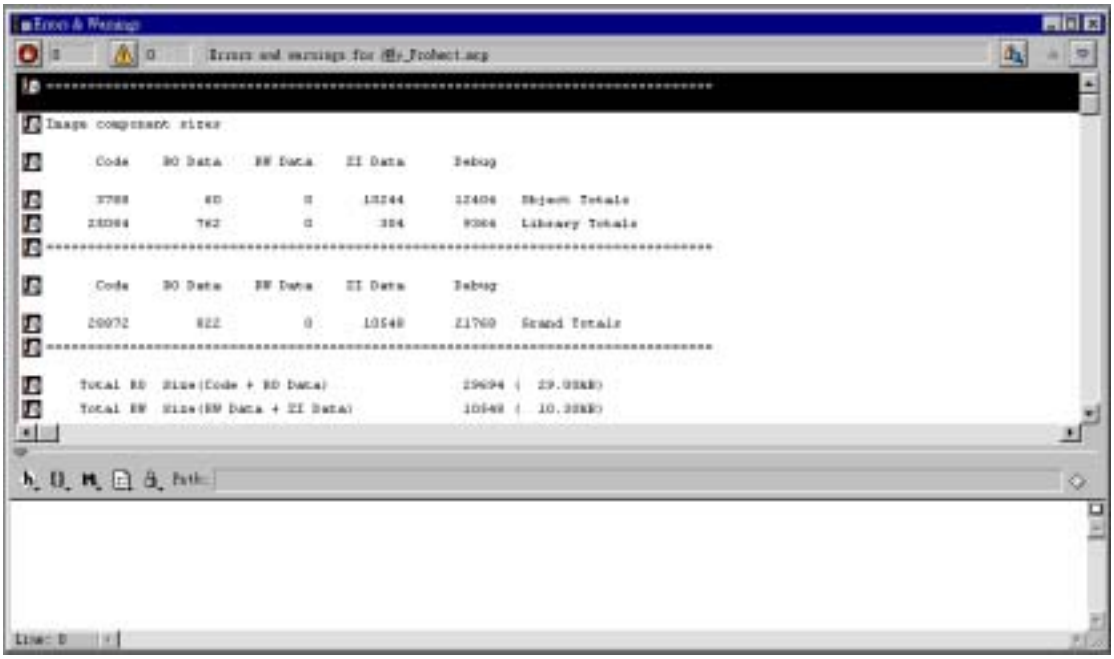

**Figure 23. Errors & Warning message window.**

3. Choice either ***dhry_1.c*** or ***dhry_2.c*** listed in the ***Project Window*** (Figure 21). Then right-click on in the ***Project Window*** and select ***Disassemble*** from the pop-up menu. The disassembled code is displayed in a ***Disassembly window***.

## Step 6: Debugging the project

To execute and debug your example project:

1. Select the build target you want to build (Figure 11). For this example, select the ***DebugRel*** build target.
2. Select ***Project →Debug*** (Ctrl + F5). The CodeWarrior IDE compiles and links any source files that are not up to date, and calls the ***AXD*** debugger to load the image and on standby to execute the image.

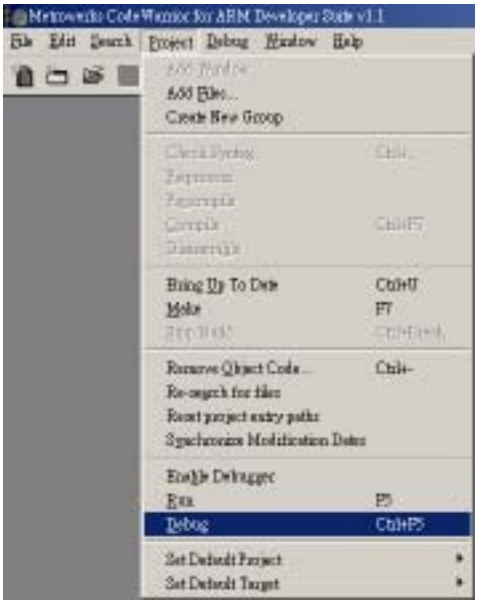


**Figure 24. Select Debug.**

Other ways to start AXD

- Click on ***Run*** button from Project Window, as shown in Figure 25. The CodeWarrior IDE then calls debugger to load and execute the image. The term ARM Runner refers to the ARM debugger that is called to execute, rather than debug, an image file.
- Double click on ***My_project.axf***, AXD starts.
- Select ***Start →Programs →ARM Developer Suite 1.2 →AXD Debugger***
- Using a Windows DOS shell: ***axd -debug filename.axf***

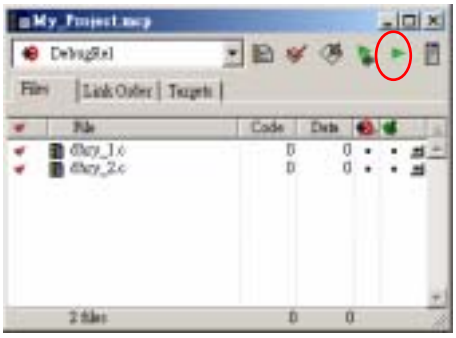


**Figure 25. Debug the project from Project Window.**

3. Select debugging system from ***Options → Configure Target*** (Figure 26). The AXD displays a ***Choose Target Panel*** (Figure 27). Select ***ARMUL*** and then click ***OK***.

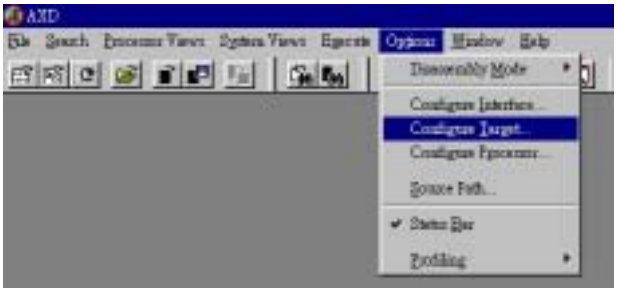


**Figure 26. Configure Target.**

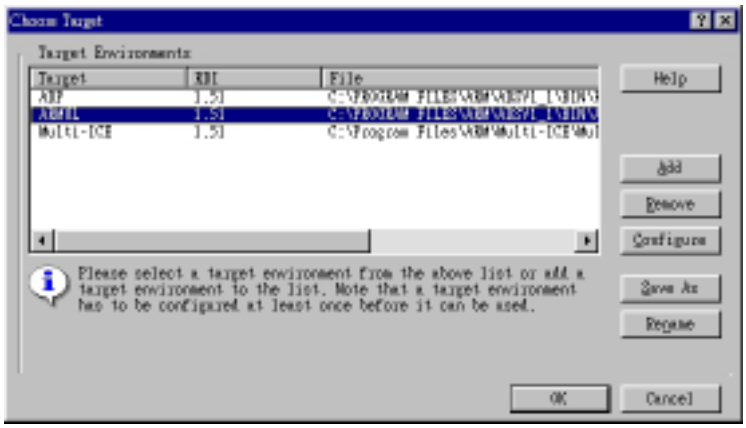


**Figure 27. Choose Target Panel.**

4. Use ***File →Load Images…*** to load a new image. If you start AXD from the CodeWarrior IDE, you can skip this step.


**Figure 28. Loading an image.**

5. Use ***Execute → Step*** (or F10) or the button shown in Figure 29 to step through the application. The disassembled code is displayed and a pointer

indicates the current position (Figure 30). Use ***F10*** to execute the next instruction.

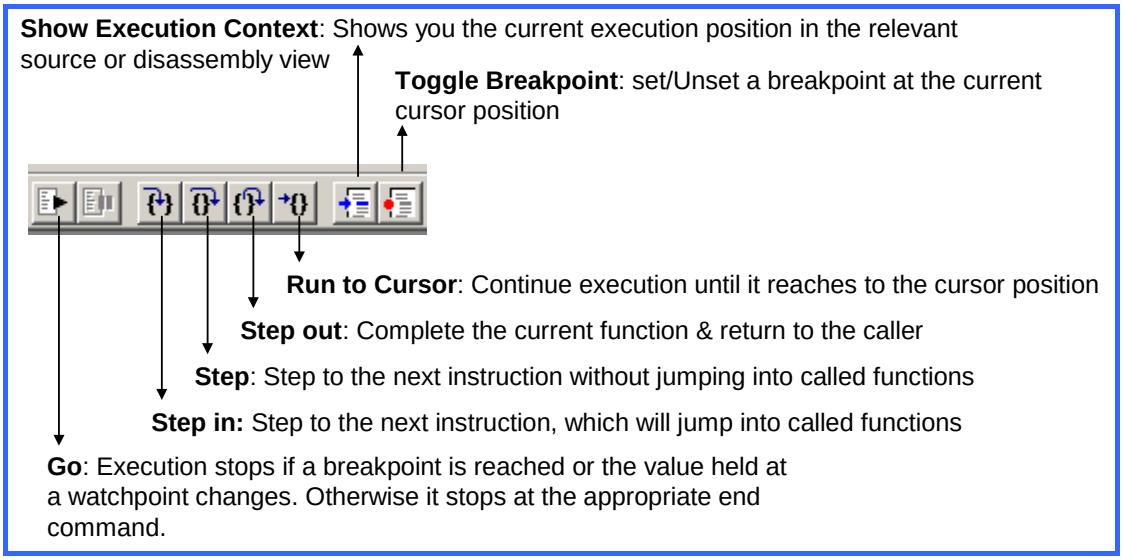


**Figure 29. The Execute menu.**

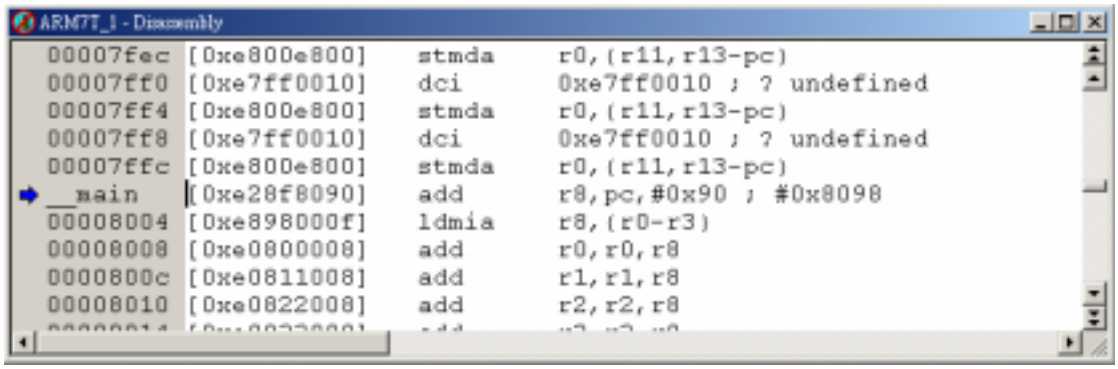


**Figure 30. The disassembled code.**

6. Interleaving source code.
   It is often useful to see interleaved code, i.e., the high level C code, and the low level assembled code together.
   6.1 Select Processor Views ➔ Source, and then choose dhry_1.c.
   6.2 Right click on the c/c++ source code window and select ***Interleave disassembly***.
   6.3 Take a look at the c source code. The source code is now interleaved with assembly code.
   6.4 Press Go. Type 4000 in the ***Console Window*** when be asked. "Program terminated normally." Will be show in ***Console Window*** if the program is successfully executed.
   6.5 Close CodeWarrior and AXD.

----------Note----------

- Before loading an image, you have to make sure the selected target (ARMulator, Multi-ICE, EmbeddedICE or Angel debug monitor) exists, or

you cannot load a image (*.axf).
- The use of AXD will be explained in detail in next Lab.

---

### 1.3.2. 實驗步驟: ARM/Thumb Interworking

This lab contains the following examples and exercises to explain how to change between ARM state and Thumb state when writing code for different instruction sets.

Examples
- ARM/Thumb Interworking Examples
- ARM/Thumb Interworking in C/C++ only.
- ARM/Thumb Interworking in ASM only.
  - No Veneer
  - With Veneer
- ARM/Thumb Interworking between C/C++ and ASM.

Exercise
- Part A: Interworking using C/C++
  - Thumb Main & ARM Sub
- Part B: Interworking using ASM
  - Without Veneer
  - With Veneer
- Part C: Interworking using C/C++ and ASM
  - Modify the given example to practice the interworking using C/C++ and ASM.

## ARM/Thumb Interworking using ASM (no Veneer)

The program ***addreg.s*** shown in Figure 31 does computations among registers. No veneer is needed; interworking instruction change is implemented manually.

The program consists of 4 parts:
1. Main: Start in ARM state. Generate branch address by an ADR instruction to load the branch address and set the least significant bit bit0=1 to arrive at target in Thumb mode. The ADR instruction generates the address by loading r2 with the value ***pc+offset***. See “ADS Compiler, Linker, and Utilities Guide” for more information on the ADR instruction.
2. ThumbProg: Set values for r2 and r3, and then sum them to r2. Executed in Thumb state.
3. ArmProg: Set values for r4, r5. Sum r4, r5 to r4. Executed in ARM state.
4. Stop: Terminate the program. Semihosting SWI is used to report normal application exit. Refer to the “ADS Debug Target Guide” for more information on Semihosting.

```
1         AREA AddReg,CODE,READONLY     ;Name this block of code.
2         ENTRY                   ;Mark first instruction to call.
3   Main
4         ADR   r0,ThumbProg +1   ;Generate branch target address
5                                 ;and set bit 0,hence arrive
6                                 ;at target in Thumb state.
7         BX    r0                ;Branch exchange to ThumbProg.
8         CODE16      ;Subsequent instructions are Thumb code.
9   ThumbProg
10        MOV   r2,#2             ;Load r2 with value 2.
11        MOV   r3,#3             ;Load r3 with value 3.
12        ADD   r2,r2,r3          ;r2 =r2 +r3
13        ADR   r0,ARMProg
14        BX    r0
15        CODE32            ;Subsequent instructions are ARM code.
16  ARMProg
17        MOV   r4,#4
18        MOV   r5,#5
19        ADD   r4,r4,r5
20  Stop
21        MOV   r0,#0x18          ;angel_SWIreason_ReportException
22        LDR   r1,=0x20026       ;ADP_Stopped_ApplicationExit
23        SWI   0x123456          ;ARM semihosting SWI
24        END                     ;Mark end of this file.
```

**Figure 31. addreg.s**

Enter the code by using any text editor and then save the file as ***addreg.s***.

1. Building under DOS command line
   1.1 Type ***armasm –g addreg.s*** to assemble the source file.
   1.2 Type ***armlink addreg.o -o addreg*** to link the file.

2. Executing using ARM symbolic debugger, ***armsd***, under command line (in DOS window)
   2.1 Type ***armsd addreg*** to load the module into the command-line debugger.
   2.2 Type ***step*** to step through the program one instruction at a time.
   2.3 Type ***reg*** after each instruction execution to display the registers.
   – Watch the processor enter Thumb state. ***CPSR*** changes from "***t***" to "***T***" entering to Thumb state. (t: ARM state; T: Thumb state.)
   2.4 Type ***help*** for help info.
   2.5 Type ***quit*** to quit armsd.

## ARM/Thumb interworking using ASM (using Veneer)

This example explains how you can make use of interworking veneers to interwork between assembly language modules. The example shows how the code sets the values for r0, r1 and r2. Interworking option is then added while linking such that veneer is added by the linker. This example consists of 2 files:

1. Arm.s: Sets the values for r0, r2. Calls for ThumbProg. Executed in ARM state.

2. Thumb.s: Sets the value for r1. Return back to ArmProg. Executed in Thumb state.

```
 1        AREA  Arm,CODE,READONLY ;Name this block of code.
 2        IMPORT      ThumbProg
 3        ENTRY                  ;Mark 1st instruction to call.
 4  ARMProg
 5        MOV r0,#1          ;Set r0 to show in ARM code.
 6        BL ThumbProg       ;Call Thumb subroutine.
 7        MOV r2,#3          ;Set r2 to show returned to ARM.
 8                           ;Terminate execution.
 9        MOV r0,#0x18       ;angel_SWIreason_ReportException
10        LDR r1,=0x20026    ;ADP_Stopped_ApplicationExit
11        SWI 0x123456       ;ARM semihosting SWI
12        END
```

**Figure 32. arm.s**

```
 1        AREA Thumb,CODE,READONLY;Name this block of code.
 2        CODE16             ;Subsequent instructions are Thumb.
 3        EXPORT ThumbProg
 4  ThumbProg
 5        MOV   r1,#2        ;Set r1 to show reached Thumb code.
 6        BX    lr           ;Return to ARM subroutine.
 7        END                ;Mark end of this file.
```

**Figure 33. thumb.s**

1. Building under command line
   1.1 Type ***armasm arm.s***
   1.2 Type ***armasm -16 -apcs  /interwork thumb.s***
   1.3 Type ***armlink arm.o thumb.o -o count***

----------Note----------
- The ARM assembler can assemble both Thumb code and ARM code. By default, it assembles ARM code unless it is invoked with the -16 option.
- The callee must be compiled with interworking option if it is implemented in a different state from the caller.

---

2. Running under command line
   2.1 Type ***armsd count***.
   2.2 Type ***list 0x8000*** to list the linked code at the armsd command prompt to list the ode. Figure 34 shows the output.
   2.3 Observe that ***$Ven$AT$$ThumbProg*** is added to the code. This is the veneer added by the linker.
   2.4 Type ***quit*** to quit armsd.

```
armsd: list 0x8000
ArmProg
        0x00008000: 0xe3a00001      .... : >  mov     r0,#1
        0x00008004: 0xeb000005      .... :    bl      $Ven$AT$$ThumbProg
        0x00008008: 0xe3a02003      . .. :    mov     r2,#3
        0x0000800c: 0xe3a00018      .... :    mov     r0,#0x18
        0x00008010: 0xe59f1000      .... :    ldr     r1,0x00008018 ; = #0x00020026
        0x00008014: 0xef123456      V4..:     swi     0x123456
        0x00008018: 0x00020026      &... :    dcd     0x00020026      &...
ThumbProg
+0000   0x0000801c: 0x2102          .!   :    mov     r1,#2
+0002   0x0000801e: 0x4770          pG :      bx      r14
$Ven$AT$$ThumbProg
+0000   0x00008020: 0xe59fc000      .... :    ldr     r12,0x00008028 ; = #0x0000801d
+0004   0x00008024: 0xe12fff1c      ../. :    bx      r12
+0008   0x00008028: 0x0000801d      .... :    dcd     0x0000801d      ....
+000c   0x0000802c: 0xe800e800      .... :    dcd     0xe800e800      ....
+0010   0x00008030: 0xe7ff0010      .... :    dcd     0xe7ff0010      ....
+0014   0x00008034: 0xe800e800      .... :    dcd     0xe800e800      ....
+0018   0x00008038: 0xe7ff0010      .... :    dcd     0xe7ff0010      ....
```

**Figure 34. Veneer code segment.**

## ARM/Thumb interworking using C/C++

This example consists of 2 parts:
***armmain.c*** for main function using ARM instructions set.

- Print strings
- Call Thumb function
- Compiled using ARM C/C++ compiler.

***thumbsub.c*** for sub function called by main function using Thumb instructions set.

- Print strings
- Return to main function
- Compiled using Thumb C/C++ compiler.

```
 1  #include <stdio.h>
 2  extern void thumb_function(void);
 3
 4  int main(void)
 5  {
 6       printf("Hello from ARM\n");
 7       thumb_function();
 8       printf("And goodbye from ARM\n");
 9       return (0);
10  }
```

**Figure 35. armmain.c**

```
1  #include <stdio.h>
2  void thumb_function(void)
3  {
4      printf("Hello and goodbye from Thumb\n");
5  }
```

**Figure 36. thumbsub.c**

1. Building under MS-DOS command line
   1.1 Type *armcc -c -g -O1 -apcs /interwork armmain.c*
   - *-c* stands for compile.
   - *-g* generate debug information.
   - *-O1* compile with median optimization.

   1.2 Type *tcc -c -g -O1 -apcs /interwork thumbsub.c*
   1.3 Type *armlink armmain.o thumbsub.o -o armtothumb.axf -info veneers -info totals -callgraph -list* ***interworking.log***
   - *-o* specify output image name
   - *-info veneer* print out veneer information (e.g., size) on screen.
   - *-info totals* print out memory size information on screen.
   - *-callgraph* creates static callgraph of functions in an HTML file.
   - *-list XXX.log* redirects information to print in a text file.

   1.4 Observe the result recorded in the ***interworking.log*** file.

2. Building under CodeWarrior IDE
   2.1 Start CodeWarrior IDE.
   2.2 ***File➔New*** to create a new project.
   2.2.1 Select ***Thumb ARM Interworking Image*** under the ***Project*** tab.
   2.2.2 Type the project name, ***C_interworking.***
   2.2.3 Specify the working directory, e.g., ***C:\ARMSoC\Lab_01***
   2.3 Copy ***armmain.c*** & ***thumbsub.c*** from "Install_path/ADSv1_2/Examples/Interworking" to project directory.
   2.4 Select ***Project➔Add Files...*** to add these two files to the project. After adding files to the project, a ***Project Management*** Window would appear.

3. Setting build target
   3.1 Hit ***Build Target Setting*** button  .
   3.2 A ***ThumbDebRel Setting*** window appears. Click ***Language Settings➔ARM C Compiler in Target Setting Panel***, and then click ***ATPCS*** tab Figure 37.
   - Check ***ARM/Thumb Interworking*** in ***ARM/Thumb Procedure Call Standard Options***. A line "***-apcs /interwork***" would be added to ***Equivalent Command line*** automatically.

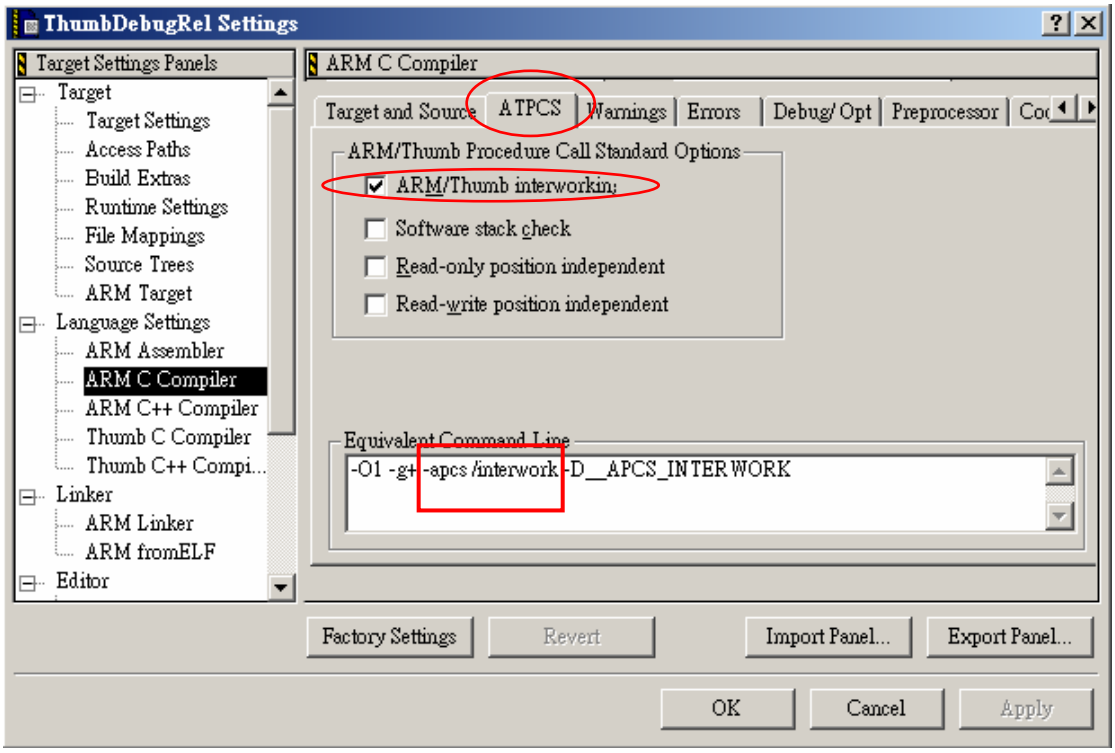


**Figure 37. ATPCS setting.**

4. Hit the ***Make*** button to compile and link the project.
   4.1 A compiling and linking status window would appear to indicate making progress.
   4.2 After finishing compiling and linking, a result messages window would appear. Check for errors and warnings.

5. Using "***armlink -info veneers armmain.o thumbsub.o***" to see Veneers information (An example is shown in Figure 38. You may get different results).

```
Adding veneers to the image

    Adding AT veneer (12 bytes) for call to '__rt_lib_init' from kernel.o(.text).
    Adding AT veneer (12 bytes) for call to '__rt_lib_shutdown' from kernel.o(.text).
    Adding AT veneer (12 bytes) for call to '_sys_exit' from kernel.o(.text).
    Adding AT veneer (12 bytes) for call to '__raise' from rt_raise.o(.text).
    Adding AT veneer (12 bytes) for call to '_no_fp_display' from printf2.o(x$fpl$printf2).

5 Veneer(s) (total 60 bytes) added to the image.
```

**Figure 38. Veneers information.**

6. Run the program and trace how the program running in both source and disassembly view.

## ARM/Thumb interworking between C/C++ & ASM using Veneer

As we know, C and C++ code compiled to run in one state can call assembly language code designed to run in the other state, and vice versa. To do this, write the caller routine as any non-interworking routine and, if calling from assembly language, use a BL instruction to make the call. The following example is that the thumb C caller calls the ARM ASM callee with a parameter ***i***. The ARM ASM callee then returns that parameter with a constant four added.

```
1  #include <stdio.h>
2  extern int arm_function(int);
3  int main(void)
4  {
5        int i =1;
6        printf("i =%d \n",i);
7        printf("And now i =%d \n", arm_function(i));
8        return (0);
9  }
```

**Figure 39. thumb.c**

```
1        AREA Arm,CODE,READONLY  ;Name this block of code.
2        EXPORT arm_function
3  arm_function
4        ADD   r0, r0, #4        ;Add 4 to first parameter.
5        BX    LR                ;Return
6        END
```

**Figure 40. armsub.s**

1. Building both programs under command line
   - 1.1 Type ***tcc -c -apcs /interwork thumb.c***
   - 1.2 Type ***armasm -apcs /interwork armsub.s***
   - 1.3 Type ***armlink armsub.o thumb.o -o add***

2. Running under command line
   - 2.1 Type ***armsd add*** to load the code
   - 2.2 Type ***go***.
   - 2.3 Type ***list main*** to list the linked code for main function.
   - 2.4 Type ***list arm_function*** to list the linked code.
   - 2.5 Observe that ***$Ven$AT$$ThumbProg*** is added to the code. This is the veneer added by the linker.

## 1.4. 實驗要求

### 1.4.1. Exercise 1

Write a program in C/C++. The main function is implemented in Thumb instructions set. The called function is implemented in ARM state.

- Specifications:
  - Thumbmain: Prints “Hello from thumb main!” & “Goodbye from Thumb main!!”. Calls ARM function. Implemented in Thumb instructions set.
  - Armsub: Prints “Hello from ARM sub.” Return back to main. Implemented in ARM instruction set.
  - Show the veneers in the linked code and its info.
  - Observe how the t-bit in *CPSR* changes.

### 1.4.2. Exercise 2

Write a program in ASM that swaps the value of [r1, r2], [r3, r4].

- No linker added veneers should be attached.
  - Swap function is implemented in ARM instructions.
  - Main Program is implemented in Thumb instructions.
  - Manually change the instruction set using, no linker added veneer.
  - Observe the linked code and the registers.
- Using veneer:
  - Do the above exercise using linker added veneer.
  - Show the veneers added.

✓ Hints & Notes:
  - ARM is in ARM state at the beginning. A change to Thumb state is needed.
  - *ARMASM* doesn’t include ARM-to-Thumb header automatically as ARMCC does. You must manually change the state to thumb at initial.
  - Veneers are added when there’s a ARM/THUMB or THUMB/ARM procedure call.

### 1.4.3. Exercise 3

Modify the last example (interworking between C/C++ and ASM using veneer) such that the main is implemented in ASM and the function is implemented in C.

- Specifications:

- Main: Implement in ASM using Thumb instructions. Call the subroutine with a parameter.
- Sub: Implement in C/C++ using ARM instructions. Add 4 to the parameter passed from main and return.
- Show the linked code.
- Observe the register.
- No need to print the results in the console window.

✓ Hints & Notes:
- C functions called by ASM code must have a return value.
- 1st parameter and function return value use *R0* to pass value.
- 2nd to 4th parameters use *R1* to *R3* to pass values.
- 5th and other more parameters should use stack to pass values.
- Standard I/O in C function does not work when it is being called by ASM codes, which means you cannot use *printf()* in C functions called by ASM main.

## 1.5. 問題與討論

1. Discuss the advantages and disadvantages of ARM and Thumb instruction sets.

## 1.6. 參考文件及網頁


- Code Development flow: ARM Developer Suite Version 1.2 Getting Started Guide".
- CodeWarrior IDE Guide
- Overview of ARM architecture: ADS Assembler Guide
- ARM instruction reference: ADS Assembler Guide, QRC_Armside
- Thumb instruction reference: ADS Assembler Guide, QRC_Thumbside
- Interworking with ARM & Thumb: ADS Developer Guide
- ARM-Thumb Procedure Call Standard: ADS Developer Guide, ATPCS spec
- AXD, armsd: ADS Debugger Guide
- Mixing C, C++, ASM: ADS Developer Guide
- http://twins.ee.nctu.edu.tw/courses/ip_core_02/index.html
- http://twins.ee.nctu.edu.tw/courses/ip_core_01/index.html