RVDS 2.2 Introductory Tutorial

Introduction

Aim

This tutorial provides you with a basic introduction to the tools provided with the RealView Developer Suite version 2.2 (RVDS). This will include the use of command line and GUI tools, to build and debug projects.

The tutorial is split into three practical sessions:

Session 1 –	Command line tools.
Session 2 –	Creating projects and debugging using the CodeWarrior IDE and RealView Debugger (RVD).
Appendix -	An Introduction to AXD

Pre-requisites

This tutorial is intended for use with a Microsoft Windows version of RVDS v2.2. You should be familiar with Microsoft DOS/Windows, and have a basic knowledge of the C programming language.

Note: Explanation of File Extensions:

`.c`	C source file.
`.h`	C header file.
`.o`	object file.
`.s`	assembly language source file.
`.prj`	project file, as used by RealView Debugger (RVD).
`.axf`	ARM Executable file, as produced by **`armlink`**.
`.txt`	ASCII text file.

Additional information

This tutorial is not designed to provide detailed documentation of RVDS. Full documentation is provided with the product.

Further help can be accessed by pressing *F1* when running RVD, from the help menu, or by using the --help switch for a command line tool. The documentation is also available in PDF format. This can be found by going to *Start* → *Programs* → *ARM* → *RealView Developer Suite 2.2* → *PDF Documentation*.

Section 1: Command Line Tools

This section covers the command line tools required to create and examine executable images from the command line. These include:

`armcc`	ARM C compiler.
`tcc`	Thumb C compiler.
`armlink`	Object code linker.
`armasm`	Assembler for ARM/Thumb source code.
`armsd`	ARM command line debugger.
`fromelf`	File format conversion tool.

Help is available from the command line for all of the tools covered in this session by typing the name of the tool followed by **`--help`**.

For more details please refer to the following documentation: *Compiler and Libraries Guide, Linker and Utilities Guide*.

For the exercises in this section, you will receive a warning from the compiler: “-g defaults to –O2 if no optimisation level is specified”. You can ignore this warning, as we will not be debugging the image files in this section.

Consider the following simple C program which calls a subroutine. This file is provided as **`hello.c`** in **`c:\rvds22_tutorial\intro\session1\`**

```
/* hello.c Example code */

#include <stdio.h>
#include <stdlib.h> /*for size_t*/

void subroutine(const char *message)
{
  printf(message);
}


int main(void)
{
  const char *greeting = "Hello from subroutine\n";
  printf("Hello World from main\n");
  subroutine(greeting);
  printf("And Goodbye from main\n\n");
  return 0;
}
```

Exercise 1.1 - Compiling and running the example

Compile this program with the ARM C compiler:

```
armcc -g hello.c
```

The C source code is compiled and an ARM ELF object file, **`hello.o`**, is created. The compiler also automatically invokes the linker to produce an executable with the default executable filename **`__image.axf`**.

The **`-g`** option adds high level debugging information to the object/executable. If **`-g`** is not specified then the program will still produce the same results when executed but it will not be possible to perform high level language debugging operations.

Thus this command will compile the C code, link with the default C library and produce an ARM ELF format executable called **`__image.axf`**.

The generated image will execute on an ARM core.
`armsd` runs the image using the ARMulator (ARM Instruction Set Simulator).

Execute this program using **`armsd`** as follows**:**

```
armsd -exec __image.axf
```

This command informs the debugger to execute the image and then terminate.

`armsd` responds with:

```
Hello World from main
Hello from subroutine
And Goodbye from main

Program terminated normally at PC = 0x000089c8 (_sys_exit + 0x8)
+0008 0x000089c8: 0xef123456  V4.. :    swi      0x123456
Quitting
```

Exercise 1.2 - Compilation options

Different arguments can be passed to the compiler from the command line to customize the output generated. A list of the more common options, together with their effects, can be viewed by entering **`armcc -help`** at the command line. Some of these options are listed below:

`-c`	Generate object code only, does not invoke the linker.
`-o <filename>`	Name the generated output file as 'filename'.
`-S`	Generate an assembly language listing.
`-S --interleave`	Generate assembly interleaved with source code.

When the compiler is asked to generate a non-object output file, for example when using –c or -S, the linker is **not** invoked, and an executable image will not be created. These arguments apply to both the ARM and Thumb C compilers.

RVCT uses a `--` prefix for multi character switches like **`interleave`**. Legacy versions of switches (eg: **`-fs`**) are still supported, but will generate a deprecated option warning.

Use the compiler options with **`armcc`** or **`tcc`** to generate the following output files from hello.c:

`image.axf`	An ARM executable image.
`source.s`	An ARM assembly source.
`inter.s`	A listing of assembly interleaved with source code.
`thumb.axf`	A Thumb executable image.
`thumb.s`	A Thumb assembly source.

Run the Thumb executable image using **`armsd`**, the output generated should be the same as before.

Use a suitable text editor to view the interleaved source file.

To use Notepad from the command line type **`Notepad <filename>`**.

Note the sections of assembly source that correspond to the interleaved C source code.

Exercise 1.3 - armlink

In previous exercises we have seen how the compiler can be used to automatically invoke the linker to produce an executable image. **`armlink`** can be invoked explicitly to create an executable image by linking object files with the required library files. This exercise will use the files, **`main.c`** and **`sub.c`** which can be linked to produce a similar executable to the one seen in the previous exercises.

Use the compiler to produce ARM object code files from each of the two source files.

Remember to use the **`-c`** option to prevent automatic linking

Use **`armlink main.o sub.o -o link.axf`** to create a new ARM executable called **`link.axf`**

`armlink` is capable of linking both ARM and Thumb objects.
If the **`-o`** option is not used an executable with the default filename, **`__image.axf`**, will be created.

Run the executable using **`armsd`** and check that the output is similar to before.

The ability to link files in this way is particularly useful when link order is important, or when different C source modules have different compilation requirements It is also useful when linking with assembler object files.

Exercise 1.4 - fromelf

ARM ELF format objects and ARM ELF executable images that are produced by the compilers, assembler and/or linker can be decoded using the **fromelf** utility, and the output examined. Shown below is an example using the **--text** option with the **/c** switch to produce decoded output, showing disassembled code areas, from the file **hello.o**:

```
fromelf --text=/c hello.o
```

Alternatively re-direct the output to another file to enable viewing with a text editor:

```
fromelf --text=/c hello.o > hello.txt
```

Use the **fromelf** utility to produce and view disassembled code listings from the **main.o** and **sub.o** object files.

A complete list of options available for 'fromelf' can be found from the command line using fromelf **--help**, or by consulting the on-line documentation.

The **--text=/c** option can be replaced with the abbreviated **-c** switch.

Section 1 - Review

We have now seen how the command line tools can be used to compile, link and execute simple projects.

`armcc`	The compiler can be called with many different options. The **`-g`** option is required to enable source level debugging. The compiler can be used to generate executable images, object files and assembly listings.
`tcc`	The Thumb compiler can be used in the same way as **`armcc`**.
`armasm`	The assembler can be used to construct object files directly from assembly source code.
`armlink`	The linker can be used to produce executable images from ARM or Thumb object files.
`fromelf`	The 'fromelf' facility can be used to generate disassembled code listings from ARM or Thumb object or image files.
`armsd`	Can be used to execute applications from the command line.
❓	Help is available from the command line. Alternatively, consult the online documentation for further information.

Section 2: Creating projects and debugging using CodeWarrior and RVD

In this session we will see how the CodeWarrior Integrated Development Environment can be used with the RealView Debugger (RVD) to create and develop projects.

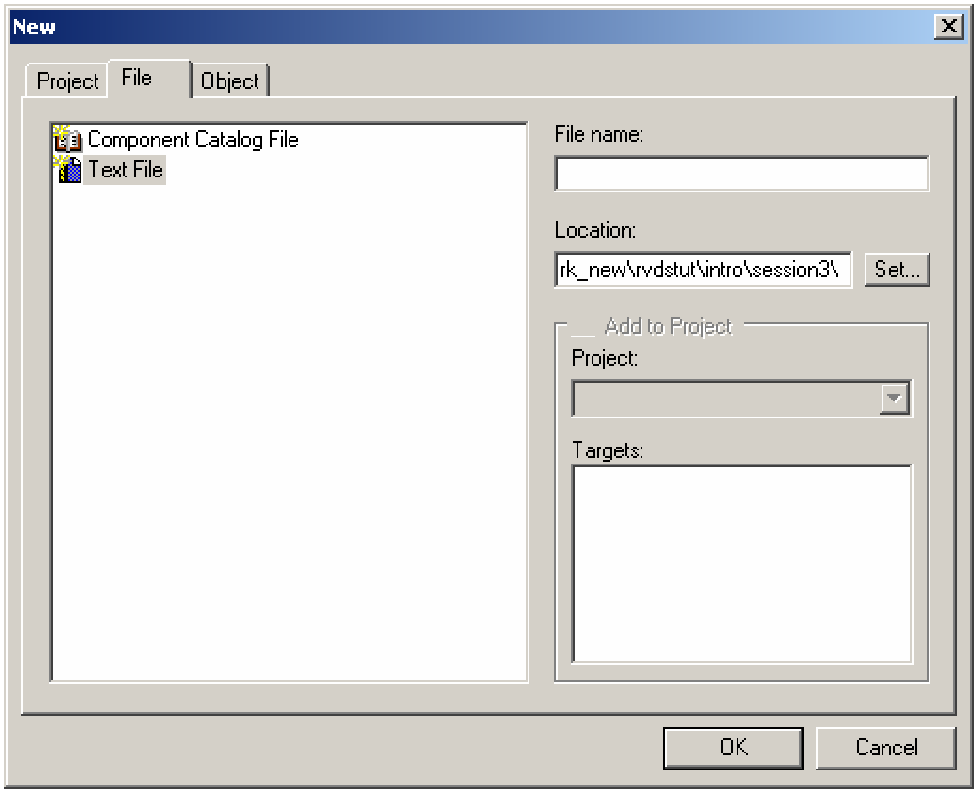
Exercise 2.1 - Creating a header file

In this exercise, we will create a new header file using CodeWarrior.

Start CodeWarrior by clicking on the icon in the Windows Start Menu folder: *ARM→ RealView Developer Suite→ CodeWarrior for RVDS*

Close any open project(s) by clicking the close button in their project windows.

Select *File →New* from the menu. Select the *File* tab as shown below, and click on *Text File* in the selection list. Click *Ok*.



Location: rk_new\rvdstut\intro\session3\

Enter the following C struct definition:

```
/* Struct definition */

struct Date_Format
{
    int day;
    int month;
    int year;
};
```

Select *File→Save As* from the menu.

Navigate the directory structure to:
`c:\rvds22_tutorial\intro\session2\`
and save as filename **`date_format.h`**

You have now created a very simple header file. This will be used later by the example program: **`month.c`**.

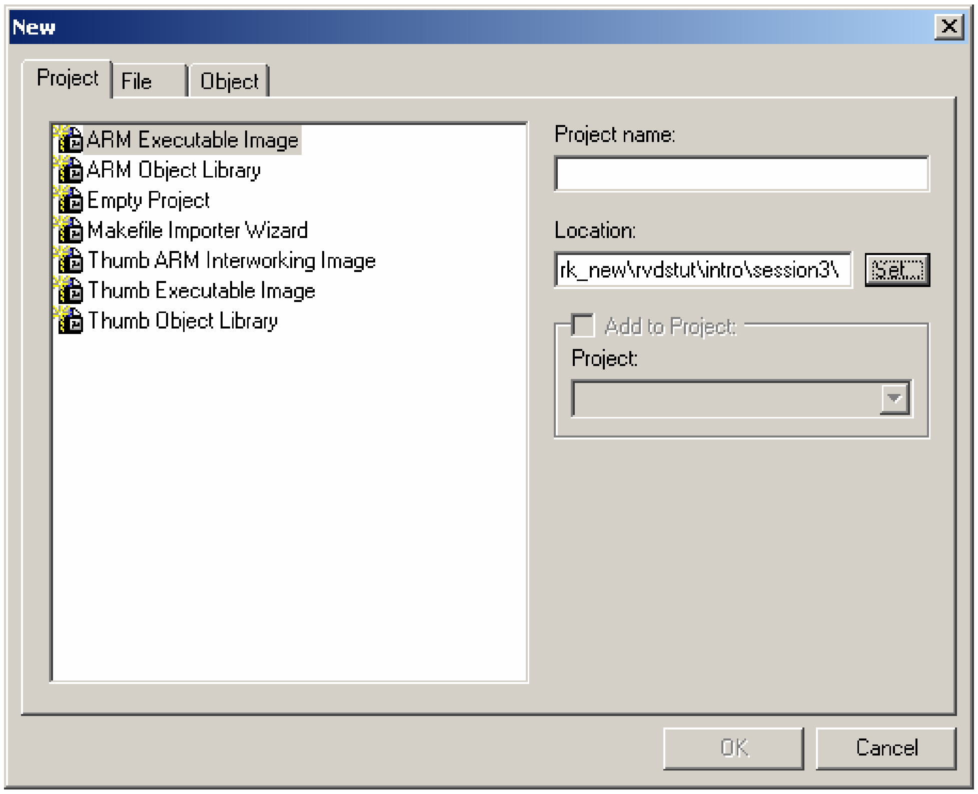
Leave the editor window open for use later in the exercise.

Exercise 2.2 - Creating a new project

We will now create a new project and add our files **`month.c`** and **`datetype.h`** to it

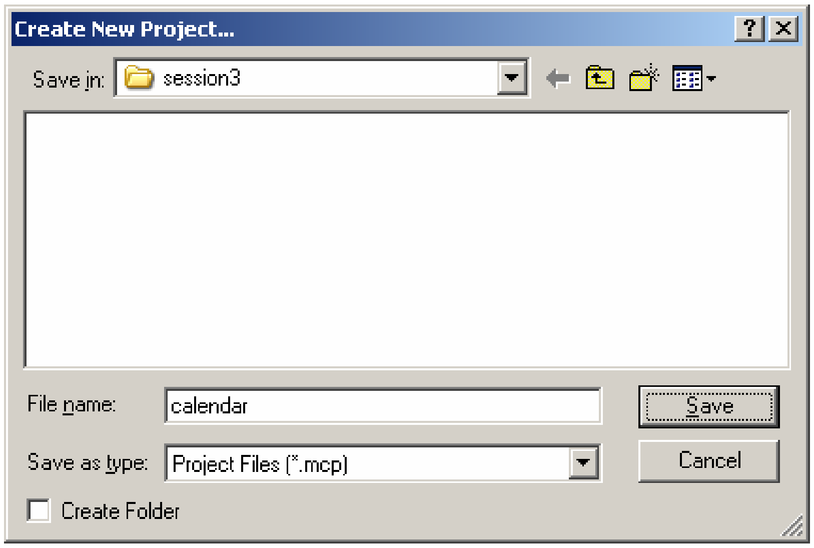
Select *File*→ *New*… from the menu.

The *New* dialog appears again:



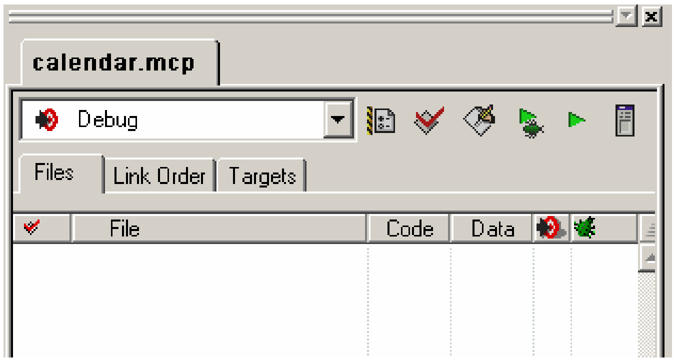
Ensure that the *Project* tab is selected, and that *ARM Executable Image* is highlighted in the stationery selection list.

Use the *Set* button next to the *Location* text box to navigate to the project directory: **`c:\rvds22_tutorial\intro\session2\`**.



Clear the *Create folder* check box and enter **`calendar`** as the name in the *Create New Project* dialog as above. Click *Save*. Click *OK* to close the *New* dialog and create the project.

The project window appears for the calendar project that you have just created:

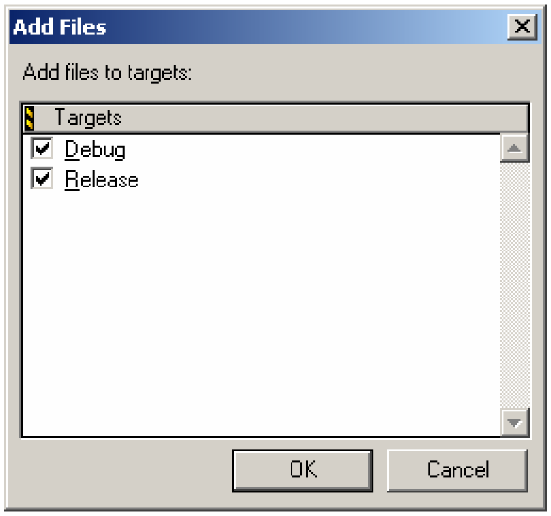


Click in a blank area of the project window to ensure that it has focus, then select *Project → Add Files…* from the main menu.

The *Select files to add…* dialog appears.

Navigate to the directory:
`c:\rvds22_tutorial\intro\session2\`
Double click on the file **`month.c`**

The *Add Files* dialog appears showing the targets to which the files can be added:



Click *OK* to add the file to both of the targets in the project.

We should also add the header file that we created earlier to the project. This is so that any changes to the header file will cause the associated modules to be rebuilt.

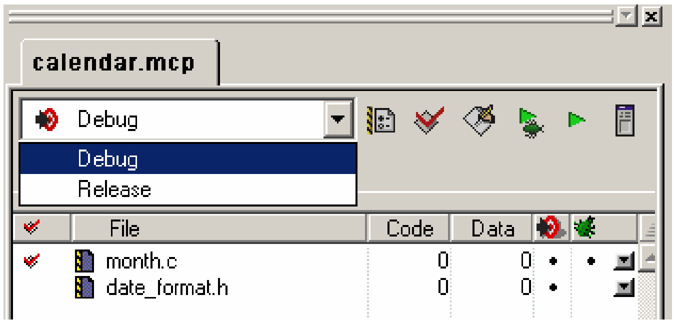
Click the title bar of the *date_format.h* window so that it has the focus. Now select *Project → Add date_format.h to Project…* from the menu.

The *Add Files* dialog reappears.

Click *OK* to add the header file to the project.

Finally, we must check that the correct build target is selected.

Ensure that the *Debug* target is the active target in the project window, as shown below.



	File	Code	Data		
✓	month.c	0	0	•	•
	date_format.h	0	0	•	

The two default targets available refer to the level of debug information contained in the resultant image.

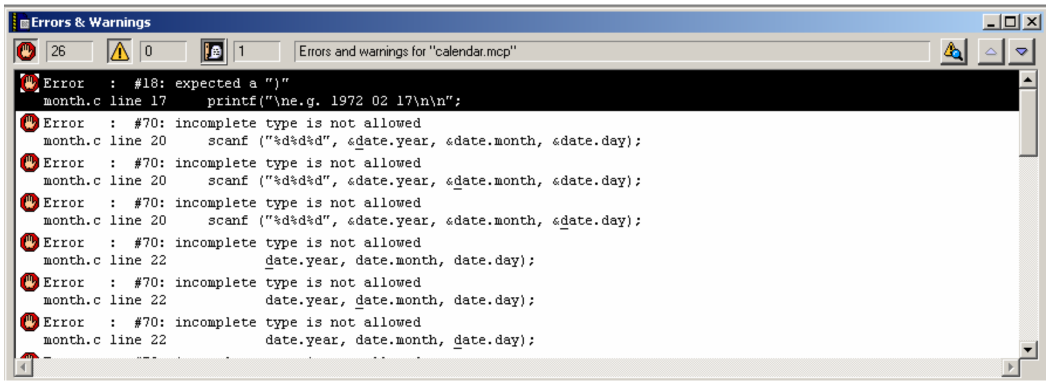
Debug	Contains full debug table information and very limited optimization.
Release	Enables full optimization, resulting in a worse debug view.

It is the *Debug* build target that we shall use for the remainder of this tutorial.

Exercise 2.3 - Building the project (Debug target)

Select *Project→ Make* from the menu, or press *F7*.

The *Errors & Warnings* window appears with the several messages generated as a result of the attempted build:



```
Error   :  #18: expected a ")"
month.c line 17     printf("\ne.g. 1972 02 17\n\n";
Error   :  #70: incomplete type is not allowed
month.c line 20     scanf ("%d%d%d", &date.year, &date.month, &date.day);
Error   :  #70: incomplete type is not allowed
month.c line 20     scanf ("%d%d%d", &date.year, &date.month, &date.day);
Error   :  #70: incomplete type is not allowed
month.c line 20     scanf ("%d%d%d", &date.year, &date.month, &date.day);
Error   :  #70: incomplete type is not allowed
month.c line 22             date.year, date.month, date.day);
Error   :  #70: incomplete type is not allowed
month.c line 22             date.year, date.month, date.day);
Error   :  #70: incomplete type is not allowed
month.c line 22             date.year, date.month, date.day);
```

The code pane of the window opens the relevant source file and an arrow highlights the line of code associated with the first error message. There is something wrong with the code; a close bracket is missing. The line should read:

```
printf("\ne.g. 1972 02 17\n\n");
```

Correct the error by adding the missing bracket and then save the updated source file.

Rebuild the project (*F7*).

The *Errors & Warnings* window again shows the errors associated with the failed build. The first error message is:

```
"month.c", line 20: Error:  #70: incomplete type is not allowed
    scanf ("%d%d%d", &date.year, &date.month, &date.day);
```

Once again, the code pane of the Errors & Warnings window displays the relevant source file and an arrow highlights the line of code associated with the first error message. You will find that there is nothing wrong with the code on this line!

Towards the top of the file, the preprocessor directives contain a reference to the macro **`INCLUDE_DATE_FORMAT`**, which has not been defined in any of the source files. Normally a command line parameter would have to be supplied to the C compiler, **`armcc`**, to specify:

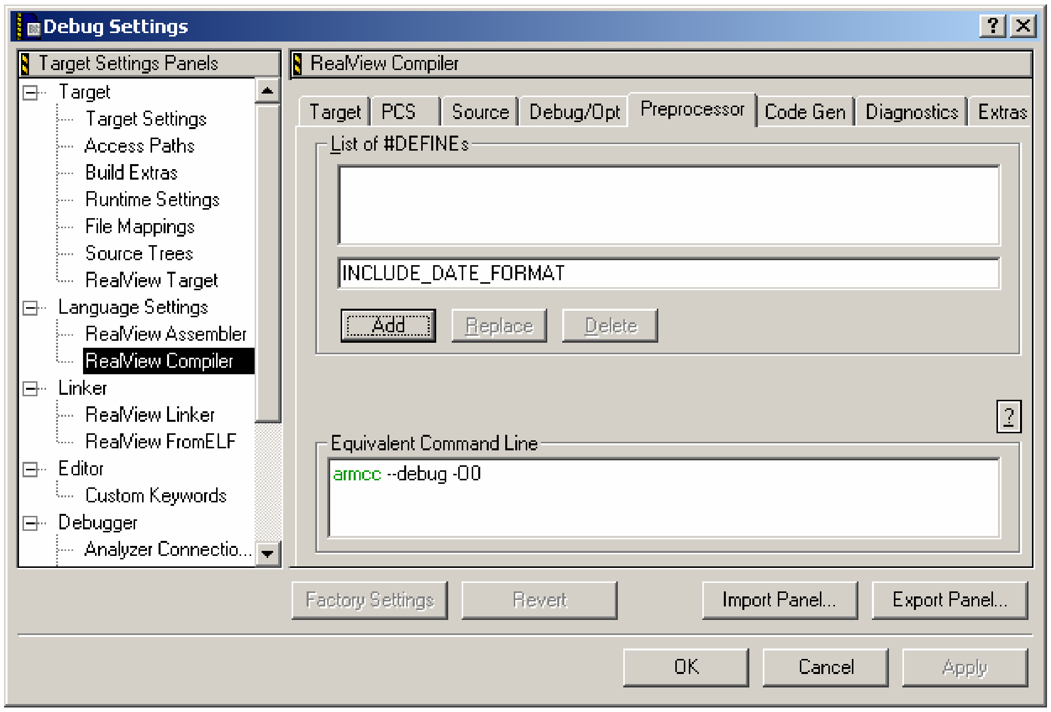
```
-D INCLUDE_DATE_FORMAT
```

We must edit the command line arguments for this project's settings:

Open the *<Target> Settings* window (*Alt + F7*), here we can use the *Target Settings Panels* pane on the left to access the ARM compiler preprocessor settings.

Note that as we are editing the Debug target, the window title and menu option will show *Debug Settings*.

Select the *RealView Compiler* item in the *Language Settings* branch of the tree. Click on the Preprocessor tab.



```
INCLUDE_DATE_FORMAT
```

```
armcc --debug -O0
```

Click in the text box under the empty list of macro definitions. Enter **`INCLUDE_DATE_FORMAT`** and click the *Add* button.

We also need to change the optimisation level to –O1. This is so that variables are stored in registers automatically, while still retaining a reasonable debug view.

Select the *Debug/Opt* tab. In the list of optimisation levels, select level 1 (good debug view, good code). Click *OK* to close the *Target Settings* dialog and save the changes.

Select *File→ Save* from the menu to save the changes to the project.

Finally, select *Project→ Make (F7)* from the menu once more. The project should build successfully.

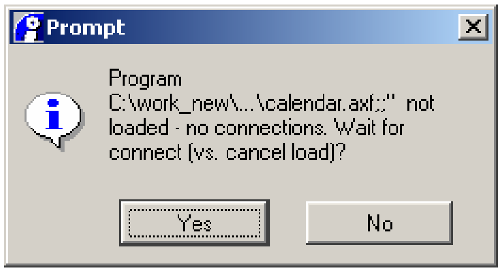
If a project is already up-to-date then nothing will be done by the IDE when it is requested to build a project. If you wish to do a forced rebuild of all the source files then select *Project → Remove Object Code…* to delete the relevant object files.

Exercise 2.4 - Executing the example

Before the image can be executed it must be loaded to an appropriate target using the debugger. This example will use the RealView Instruction Set Simulator (RVISS), as the target to execute the image using the RealView Debugger (RVD).

In CodeWarrior, select *Project* → Debug from the menu. RVD is started.

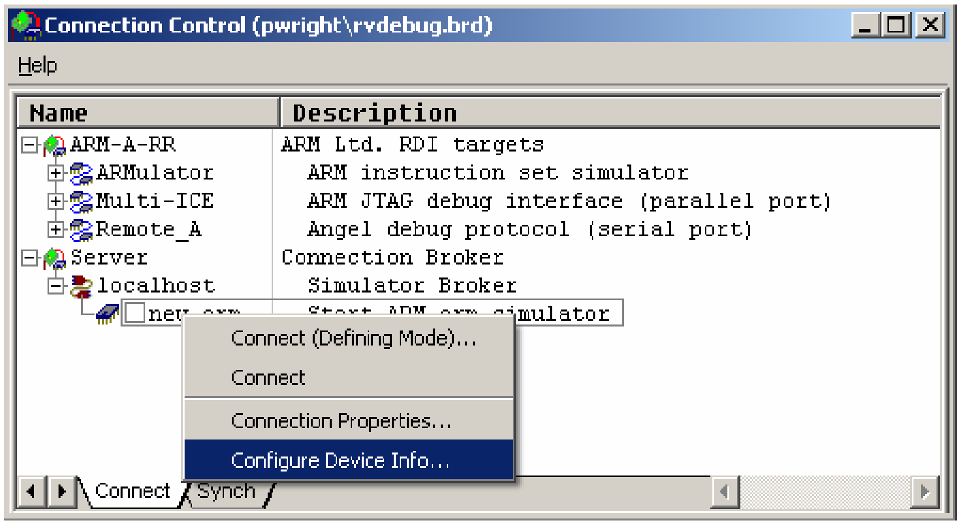
If there is no connection to a target already set up, the dialog box below will appear and RVD will prompt you to wait for a connection to be made. Click *Yes*.



Prompt

Program C:\work_new\...\calendar.axf::" not loaded - no connections. Wait for connect (vs. cancel load)?

Select *Target* → *Connect to Target…* from the menu to launch the *Connection Control* window.

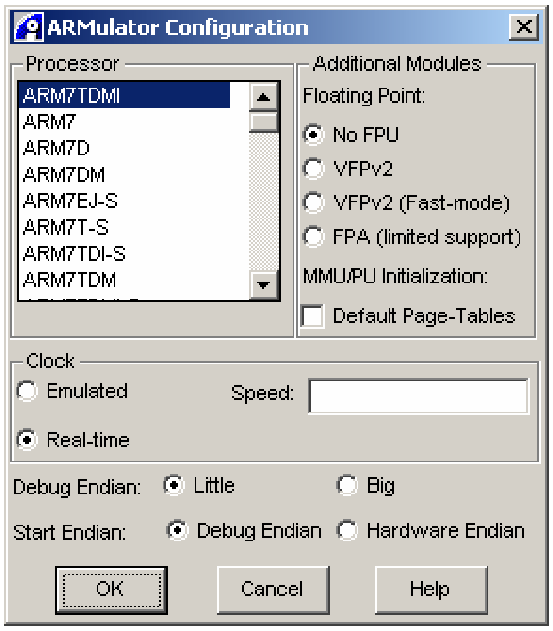


Connection Control (pwright\rvdebug.brd)

```
Name              Description
ARM-A-RR          ARM Ltd. RDI targets
  ARMulator         ARM instruction set simulator
  Multi-ICE         ARM JTAG debug interface (parallel port)
  Remote_A          Angel debug protocol (serial port)
Server            Connection Broker
  localhost         Simulator Broker
    new_arm           Start ARM arm simulator
```

Expand the *Server, Local Host* branches in the *Name* tree, then right click “new_arm” and select *Configure Device Info*.

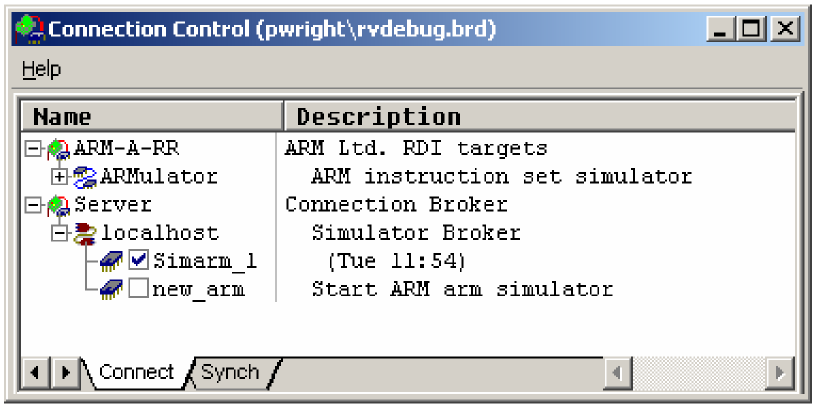
The *ARMulator Configuration* window appears:



Processor: ARM7TDMI
Floating Point: [✓ No FPU] VFPv2 VFPv2 (Fast-mode) FPA (limited support)
MMU/PU Initialization: Default Page-Tables (unchecked)
Clock: Emulated [✓ Real-time]
Speed:
Debug Endian: [✓ Little] Big
Start Endian: [✓ Debug Endian] Hardware Endian

Select ARM7TDMI as shown above and click *OK* to return to the Connection Control window.

Tick the *new_ARM* checkbox in the connection control window to connect.



```
Name              Description
ARM-A-RR          ARM Ltd. RDI targets
  ARMulator         ARM instruction set simulator
Server            Connection Broker
  localhost         Simulator Broker
    ✓ Simarm_1       (Tue 11:54)
      new_arm       Start ARM arm simulator
```

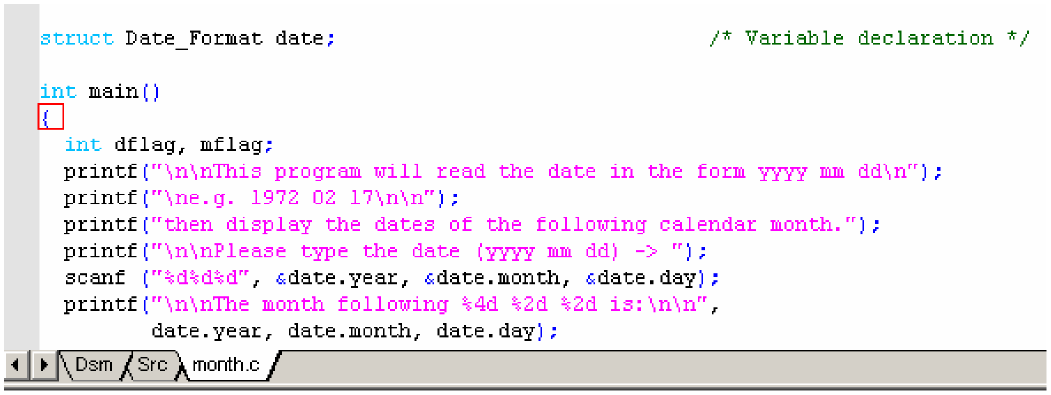
RealView Debugger is now connected to the ARM7TDMI RVISS target.

You can now close the Connection Control window..

If you were prompted to earlier wait for a connection, the image is loaded and the *Code* pane shows the source code for the project image. If not, the *Code* pane now prompts you to load the recently built image to the target. In this case, you need to click on the link to load the image to the target:

Click on the *Load* link to load the image.

The *Code* pane now shows that the image is loaded and the red box indicates the current execution position:

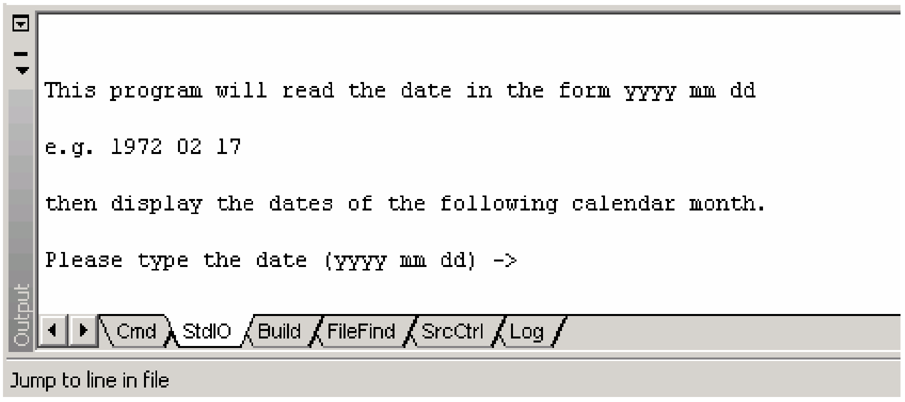


```
struct Date_Format date;                              /* Variable declaration */

int main()
{
  int dflag, mflag;
  printf("\n\nThis program will read the date in the form yyyy mm dd\n");
  printf("\ne.g. 1972 02 17\n\n");
  printf("then display the dates of the following calendar month.");
  printf("\n\nPlease type the date (yyyy mm dd) -> ");
  scanf ("%d%d%d", &date.year, &date.month, &date.day);
  printf("\n\nThe month following %4d %2d %2d is:\n\n",
         date.year, date.month, date.day);
```

Select *Debug* → *Run* from the menu *(F5)*.

Execution begins. The *Output* pane at the bottom of the window shows the *StdIO* tab which performs console I/O operations for the current image. The program is now awaiting user input:



```


This program will read the date in the form yyyy mm dd

e.g. 1972 02 17

then display the dates of the following calendar month.

Please type the date (yyyy mm dd) ->
```

Enter today's date in the format described, e.g. **`2005 01 11`**

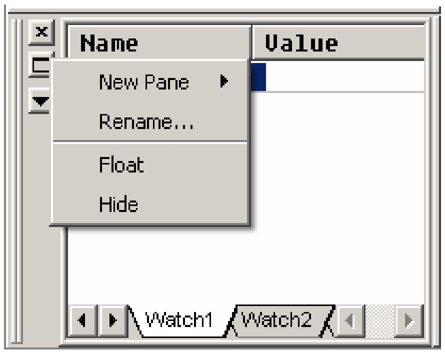
The program will display the dates for the following calendar month and then terminate.

Note that there is no source code available for the system exit routines and RVD displays **`No source for context SYS_S\_sys_exit`**

The disassembled project code can be viewed by selecting the *Dsm* tab in the *Code* pane.

All windows can be resized by clicking and dragging at the edges.

Docked panes can be changed to floating windows by clicking on the *Pane Content* button *(Alt + *)* and selecting *Float* from the menu:



Exercise 2.5 - Debugging the example

Select *Target* → *Reload Image to Target* from the menu.

RVD will load the image ready for debugging. Again the current execution position is shown at **`main`**.

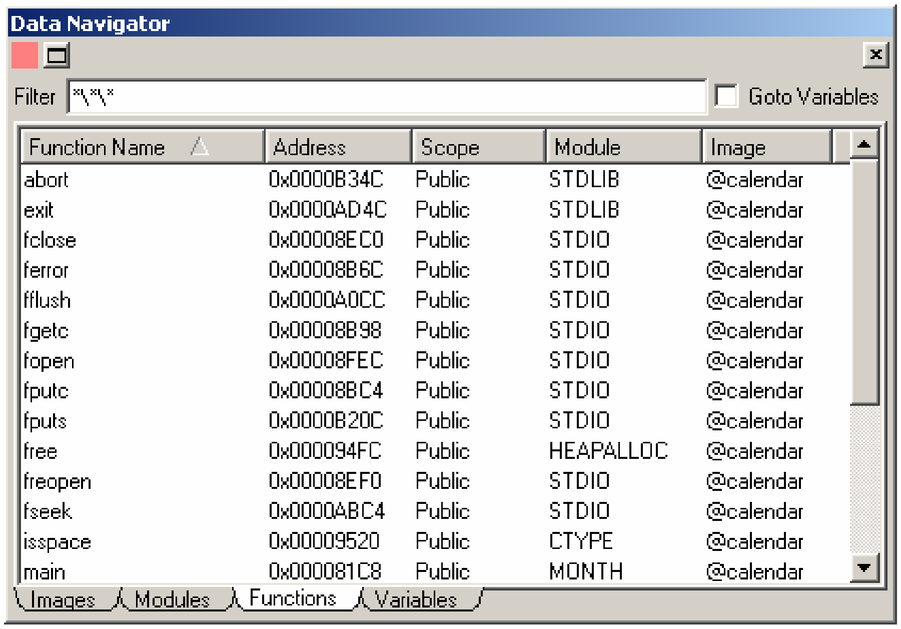
Select *Debug* → *Run* from the menu *(F5)*.

You will once again be prompted to enter a date.
This time enter **`2005 11 30`**. The program will terminate after it has output the set of dates for the following month.

Use the scroll bar at the edge of the *Output Pane* to view the dates at the end of November. You will find that there is an extra day!

Reload the image into the debugger

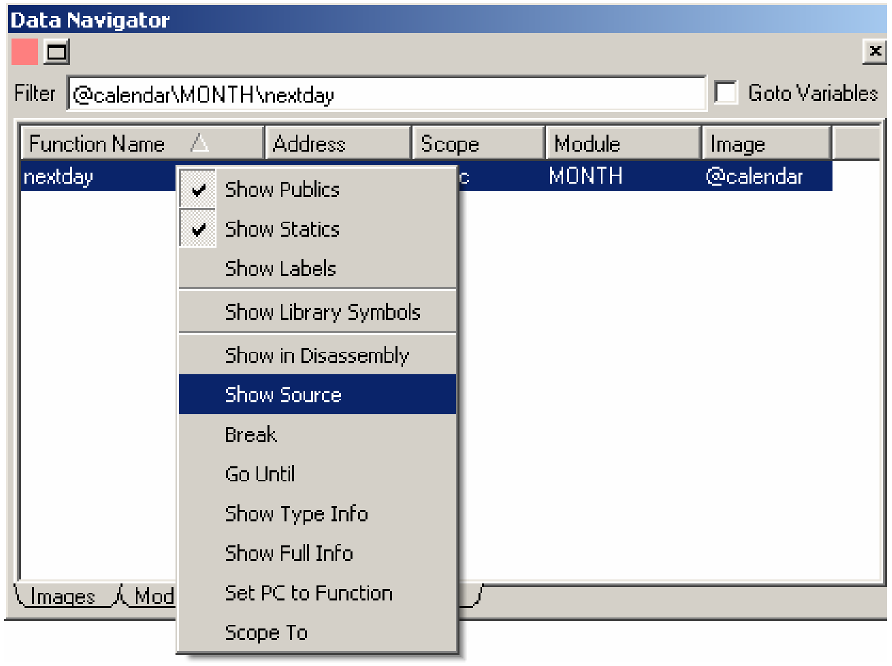
Select *View* → *Data Navigator* from the menu to open a Data Navigator pane. Select the *Functions* tab.



Function Name	Address	Scope	Module	Image
abort	0x0000B34C	Public	STDLIB	@calendar
exit	0x0000AD4C	Public	STDLIB	@calendar
fclose	0x00008EC0	Public	STDIO	@calendar
ferror	0x00008B6C	Public	STDIO	@calendar
fflush	0x0000A0CC	Public	STDIO	@calendar
fgetc	0x00008B98	Public	STDIO	@calendar
fopen	0x00008FEC	Public	STDIO	@calendar
fputc	0x00008BC4	Public	STDIO	@calendar
fputs	0x0000B20C	Public	STDIO	@calendar
free	0x000094FC	Public	HEAPALLOC	@calendar
freopen	0x00008EF0	Public	STDIO	@calendar
fseek	0x0000ABC4	Public	STDIO	@calendar
isspace	0x00009520	Public	CTYPE	@calendar
main	0x000081C8	Public	MONTH	@calendar

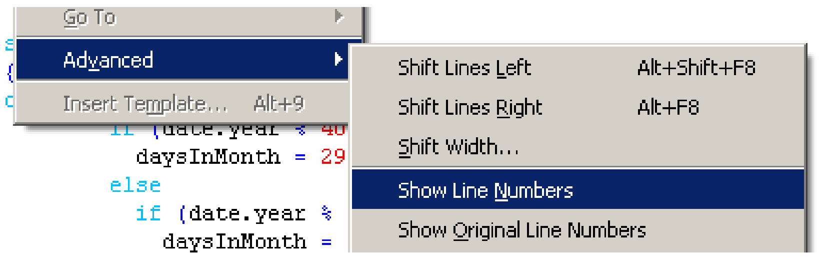
Enter the string **`@calendar\MONTH\nextday`** in the *Filter* textbox and press *Enter*.

The basic filter strings are of the form: **`@image_name\module_name\function_name`**, and you can also use wildcards. For example, to list all of the functions in the month module, you can specify: **`@calendar\MONTH\*`**. Full details of additional filters that can be used can be found in the RVD User Guide.



Highlight the **`nextday`** entry, right-click it and select *Show Source* from the context menu to locate this function in the source file. Then close the *Data Navigator* pane.

Select *Edit → Advanced → Show Line Numbers* from the menu to display line number information in the source code:



```
    if (date.year % 40
        daysInMonth = 29
    else
        if (date.year %
            daysInMonth =
```

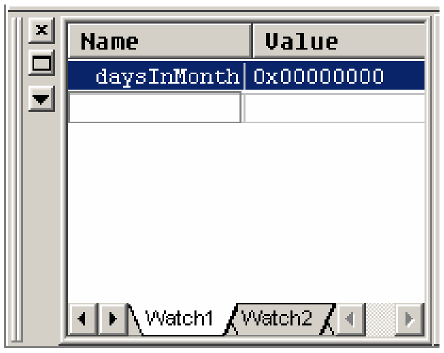
Set a breakpoint on the **`switch`** statement on line **`40`** by double-clicking to the left of the line number.

The line will receive a red breakpoint marker.

Resume execution and enter the date **`2005 11 30`** again. The program will stop at the breakpoint.

Right click on the variable name **`daysInMonth`** on line **`38`** and select *Watch* from the context menu.

A new entry appears in the *Watch* pane showing **`daysInMonth`**. Its value has not been determined yet and it is currently set to **`0`**:



Name	Value
daysInMonth	0x00000000

Right click on the word **`date`** in the variable name **`date.month`** on line **`40`** and select *Watch* from the context menu.

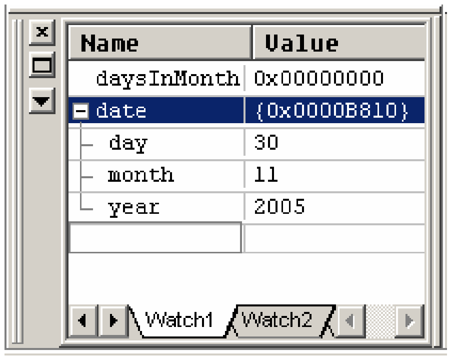
The display in the *Watch* pane is updated. The **`date`** struct is now visible.

Click on the cross to the left of **`date`** to view the struct's fields.

Right click on the **`date`** field and select *Format…* from the context menu to change the display format of the variables.

Select *Signed Decimal* from the *List Selection* dialog and click *OK*.

The *Watch* pane is updated again:



Name	Value
daysInMonth	0x00000000
date	{0x0000B810}
day	30
month	11
year	2005

Select *Debug* → *Execution Control* → *StepOver* (*F10*) to perform the next step in the program.

You will note that the case statements have been skipped and that the **`default`** path will be taken.

As the **`default`** path assumes the month has 31 days. This is not correct for November. There is a fragment of code, **`case 11:`**, missing from line **`51`**. To rectify this permanently we would have to edit the source file. For the purposes of this example we will modify the variable **`daysInMonth`** to produce the desired result.

Double-click on the breakpoint set on line **`40`** to remove it.

Set a new breakpoint on line **`58`** after the block of code containing the **`switch`** statement.

Resume program execution, the debugger will stop at the new breakpoint.

Right click on the **`daysInMonth`** variable in the *Watch* pane and change the format of this variable to *Signed Decimal*.

You will see that the value of **`daysInMonth`** is **`31`**, but we require it to be **`30`**.

Click on the value to edit it and change the value to **`30`**, then press enter.

Remove the breakpoint on line **`58`**.

	Restart the program and finish executing the example.

Note that the output generated by the program is now correct.

Exercise 2.6 – Viewing registers and memory

Reload the image into the debugger

RVD will load the image ready for debugging. Again the current execution position is shown at **`main`**.

Set a breakpoint on the **`printf`** statement on line **`29`** by double clicking in the region to the left of the statement.

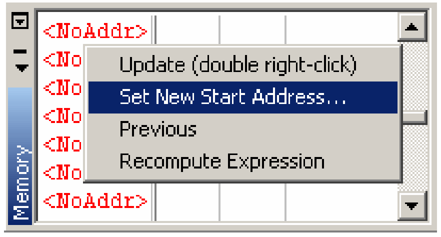
Select *Debug → Run* from the menu *(F5)*.

You will once again be prompted to enter a date.

This time enter **`2005 12 25`**.

The program will stop at the breakpoint on the printf statement.

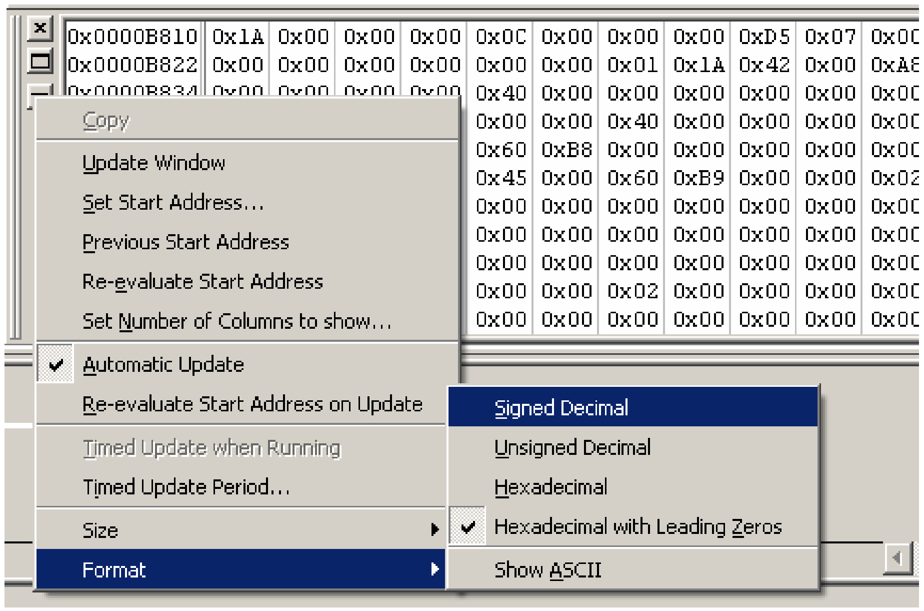
Right click inside the address column of *Memory* pane and select *Set New Start Address…* from the context menu:



Type **`date`** at the *Prompt* dialog and press enter.

The *Memory* pane is updated and now shows memory beginning at the address of the **`date`** struct.

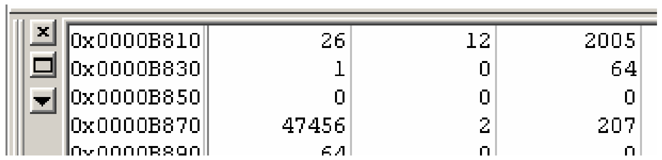
Click on the *Memory Pane Menu* and select *Signed Decimal* as the memory display format:



```
0x0000B810 0x1A 0x00 0x00 0x00 0x0C 0x00 0x00 0x00 0xD5 0x07 0x0C
0x0000B822 0x00 0x00 0x00 0x00 0x00 0x00 0x01 0x1A 0x42 0x00 0xA8
0x0000B834 0x00 0x00 0x00 0x00 0x40 0x00 0x00 0x00 0x00 0x00 0x0C
                               0x00 0x00 0x40 0x00 0x00 0x00 0x0C
                               0x60 0xB8 0x00 0x00 0x00 0x00 0x0C
                               0x45 0x00 0x60 0xB9 0x00 0x00 0x02
                               0x00 0x00 0x00 0x00 0x00 0x00 0x0C
                               0x00 0x00 0x00 0x00 0x00 0x00 0x0C
                               0x00 0x00 0x00 0x00 0x00 0x00 0x0C
                               0x00 0x00 0x02 0x00 0x00 0x00 0x0C
                               0x00 0x00 0x00 0x00 0x00 0x00 0x0C
```

Click on the *Memory Pane Menu* again and select *Words (32 bits)* as the memory display width.

Note how the three successive words in memory correspond to the three fields in the date struct (**`26/12/2005`**):

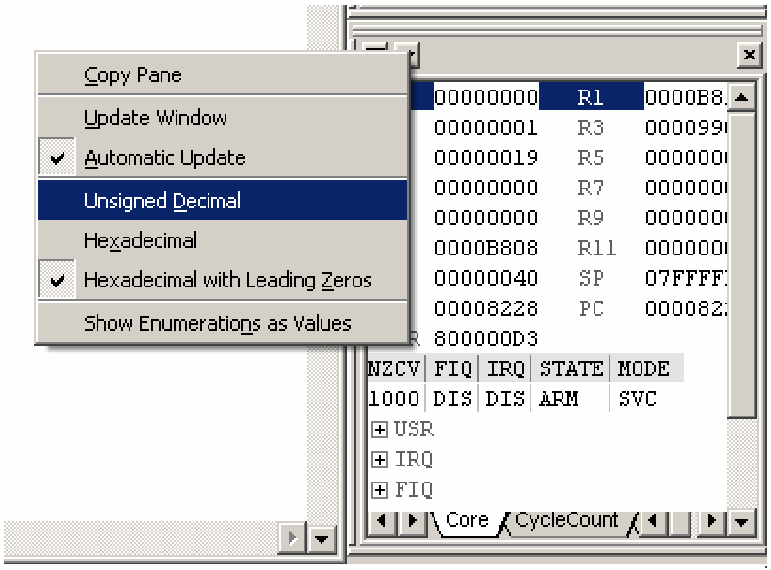


```
0x0000B810      26      12    2005
0x0000B830       1       0      64
0x0000B850       0       0       0
0x0000B870   47456       2     207
```

Restart the program, execution will stop at the breakpoint again.

Open a register pane by selecting *View* → *Registers* from the menu.

Click on the *Register Pane Menu (Alt + -)* and select *Unsigned Decimal* to change the register display format:



```
00000000  R1   0000B8
00000001  R3   000099
00000019  R5   000000
00000000  R7   000000
00000000  R9   000000
0000B808  R11  000000
00000040  SP   07FFFF
00008228  PC   000082
R 800000D3
NZCV FIQ IRQ STATE MODE
1000 DIS DIS ARM   SVC
+USR
+IRQ
+FIQ
```

At this point in the program **`r3`** holds the value stored in the **`day`** field of the **`date`** variable in the *Memory* window (The value of **`day`** is now **`27`** as the **`nextday`** function has been called.):

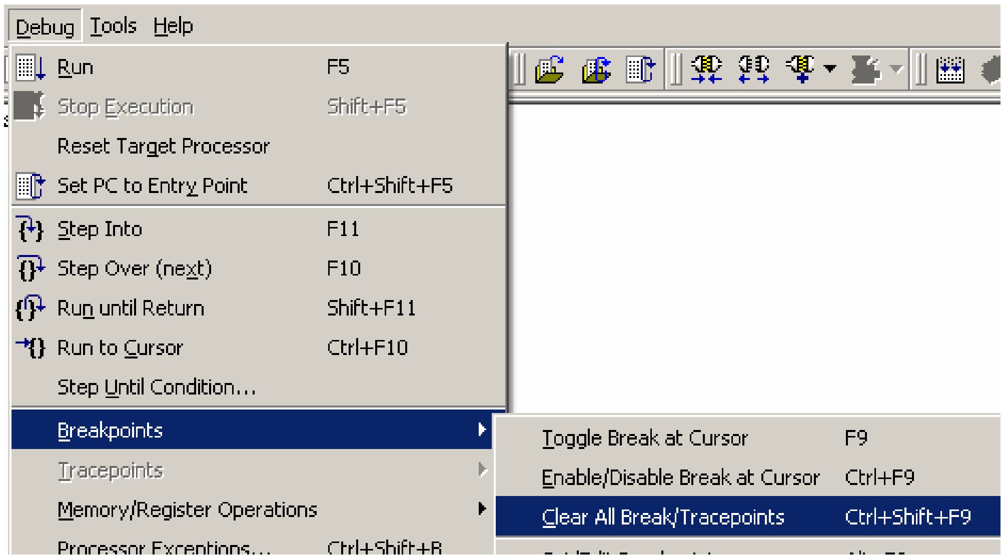
Use the *Go* button to execute the while loop until **`r3`** has the value **`2`**.

Double click on the highlighted **`date.day`** value (**`2`**) in the *Watch* pane to edit it. Change it to **`22`** and press Enter.

Use the *Go* button to pass through the while loop until the program ends.

Note how the value entered in variable watch pane affects the value in the register **`r3`**, the corresponding entry in the memory window and the program output.

Remove all current breakpoints from the image by selecting *Debug → Breakpoints → Clear All Break/Tracepoints* from the menu:

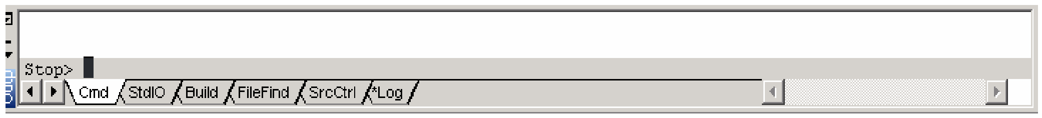

Debug
Tools
Help
Run F5
Stop Execution Shift+F5
Reset Target Processor
Set PC to Entry Point Ctrl+Shift+F5
Step Into F11
Step Over (next) F10
Run until Return Shift+F11
Run to Cursor Ctrl+F10
Step Until Condition...
Breakpoints
Tracepoints
Memory/Register Operations
Processor Exceptions... Ctrl+Shift+B
Toggle Break at Cursor F9
Enable/Disable Break at Cursor Ctrl+F9
Clear All Break/Tracepoints Ctrl+Shift+F9

Exercise 2.7 – Using the command line

Reload the image into the debugger

RVD will load the image ready for debugging. Again the current execution position is shown at **`main`**.

Click on the *Cmd* tab of the *Output* pane to view the command line interface, and click in the grey command line bar to give it the focus:



Re-size any other windows as necessary to ensure the command line interface is in clear view.

Set a breakpoint on line **`40`** of the source file by using the **`break`** command then start program execution using **`go`**:

```
Stop> break #40
Stop> go
```

Enter the date **`2005 11 30`** in the *Console* window when prompted.

Execution will stop at the breakpoint. Now check the values of the program variables:

You will need to click on the *Cmd* tab of the *Output* pane to switch focus.

```
Stop> print daysInMonth
```

Note that as it is a C variable the name is case sensitive. The value of **`daysInMonth`** is zero as it is a *static* variable and has not yet been initialized.

```
Stop> print date
```

Remove the breakpoint on line **`40`** using the **`clear`** command (The **`clear`** command will clear all breakpoints. If you need to clear a specific breakpoint, you can find the reference for the breakpoint you need using the **`break`** command. This will print a list of current breakpoints. For example to clear the first breakpoint listed type: **clear 1**).

```
Stop> clear
```

Set another breakpoint immediately after the **`switch`** statement then resume program execution:

```
Stop> break #58
Stop> go
```

Check the value of the **`daysInMonth`** variable:

```
Stop> print daysInMonth
```

It is possible to use the cursor keys, ↑ and ↓, to recall recent commands.

Correct the value from **`31`** to **`30`** using the **`ce`** command:

```
Stop> ce daysInMonth=30
```

`ce` is an abbreviation of the **`CExpression`** command. This can be used on its own to view the value of an expression, or with a modifier, such as **`=`** in this case, to change the value of an expression.

Use the **`go`** command to pass through the while loop until the output displays the date **`2005 12  3`**

```
Stop> go
```

You will need to toggle between the *StdIO* and *Cmd* tabs of the *Output* pane.

Use the **dump** command to view the **date** variable in memory.

```
Stop> dump /w &date
```

The **/w** argument specifies how to display the area of memory, in this case as words. **&date** specifies the address of the **date** variable.

Note how the successive words in memory correspond to the fields in the **date** struct.

Use the step command to execute the next two instructions:

```
Stop> step
Stop> step
```

Use the **dump** command to view the **date** variable in memory again.

```
Stop> dump /w &date
```

Note how the value of **date.day** has been incremented.

Remove the breakpoint on line **58** and resume program execution

```
Stop> clear
Stop> go
```

The program terminates normally.

Exercise 2.8 – Using include files in RVD

In this exercise we will see how multiple commands can be combined in an *include command file* to control execution within the debugger.

Consider the file **`month.inc`** found in **`c:\rvds22_tutorial\intro\session2`**:

```
break #40
go
print daysInMonth
print date.day
print date.month
print date.year
clear
break #58
go
print daysInMonth
ce daysInMonth=30
go
go
go
dump /w &date
step
step
dump /w &date
clear
go
```

The file consists of a simple selection of commands which will perform the same task that was performed in the previous exercise.

	Reload the image into the debugger
	Invoke the include file by selecting *Tools → Include Commands from File* from the menu.
	Use the *Select File* dialog to locate and open the file **`c:\rvds22_tutorial\intro\session2\month.inc`**
	Enter the date **`2005 11 30`** in the *Console* window when prompted.

When the program has terminated use the *Cmd* tab to view the values of the variables displayed by the script file.

	Check the output is correct then quit the debugger to finish the exercise.

Section 2 - Review

We have seen how the CodeWarrior IDE can be used to:

- Create source files and projects
- Invoke the compiler and linker to generate executable images.
- Automatically open files for editing from a project, or a compilation warning or error message.
- Invoke the compiler and linker to generate executable images.
- Automatically open files for editing from a project, or a compilation warning or error message.

We have seen how the RVD debugger can be used to:

- Control and modify execution of code.
- View and modify locals, globals and memory
- Accept commands via the CLI or from a script file to automate debugging.

A complete range of debugging facilities is available within RVD. Consult the online documentation for complete information.

Appendix: An introduction to AXD

In this session we will use AXD to execute and debug a simple example program which performs an exchange (or bubble) sort on elements of an array.

Exercise A.1 - Building and running the example

The program has only a single source file (**`sort.c`**) and can be built from the command line. Navigate to the directory containing the source file and compile the program using the ARM C compiler.

```
cd c:\rvds22_tutorial\intro\appendix\
armcc -g –O1 -c sort.c
```

The C source is compiled and an ARM ELF object file, sort.o is created. Generate an ARM executable image (**`sort.axf`**) using the ARM linker.

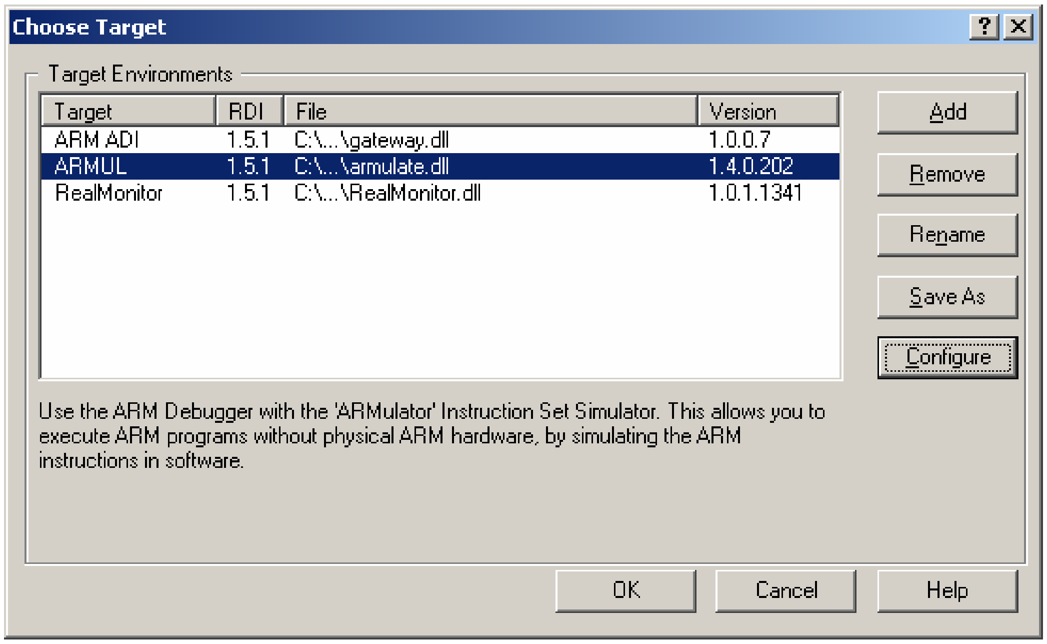
```
armlink sort.o -o sort.axf
```

Start AXD by clicking on the icon in the Windows Start Menu folder: *ARM→ RealView Developer Suite→ AXD Debugger*

You now need to connect AXD to the RealView Instruction Set Simulator (RVISS).

Open the Choose Target window by clicking *Options → Configure Target…* from the menu.

Select the “ARMUL” entry from the list, and click Configure as shown below.

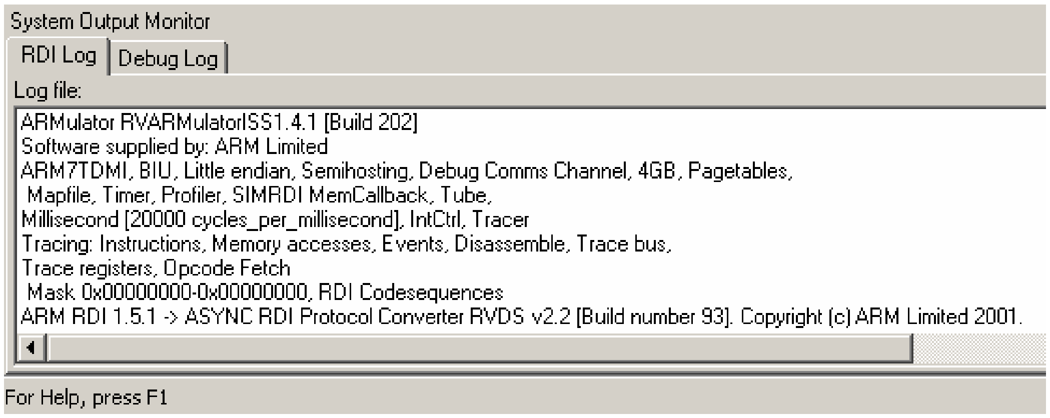


Target	RDI	File	Version
ARM ADI	1.5.1	C:\...\gateway.dll	1.0.0.7
ARMUL	1.5.1	C:\...\armulate.dll	1.4.0.202
RealMonitor	1.5.1	C:\...\RealMonitor.dll	1.0.1.1341

Use the ARM Debugger with the 'ARMulator' Instruction Set Simulator. This allows you to execute ARM programs without physical ARM hardware, by simulating the ARM instructions in software.

In the ARMulator Configuration dialog box, select "ARM7TDMI" as the processor variant, and click *OK*. Click *OK* to close the Choose Target window.

AXD will connect to the RealView Instruction Set Simulator (RVISS). The RDI log should show a connection to an ARM7TDMI core as shown.



```
ARMulator RVARMulatorISS1.4.1 [Build 202]
Software supplied by: ARM Limited
ARM7TDMI, BIU, Little endian, Semihosting, Debug Comms Channel, 4GB, Pagetables,
 Mapfile, Timer, Profiler, SIMRDI MemCallback, Tube,
Millisecond [20000 cycles_per_millisecond], IntCtrl, Tracer
Tracing: Instructions, Memory accesses, Events, Disassemble, Trace bus,
Trace registers, Opcode Fetch
 Mask 0x00000000-0x00000000, RDI Codesequences
ARM RDI 1.5.1 -> ASYNC RDI Protocol Converter RVDS v2.2 [Build number 93]. Copyright (c) ARM Limited 2001.
```

Select *File* → *Load Image* from the menu.

Navigate to the directory:
`c:\rvds22_tutorial\intro\appendix\`
Select **`sort.axf`** and click *Open* to load the file into the debugger.

The disassembled code at the programs entry point is visible in the *Disassembly* window.

From the menu select *Execute→ Go* or press *F5*.

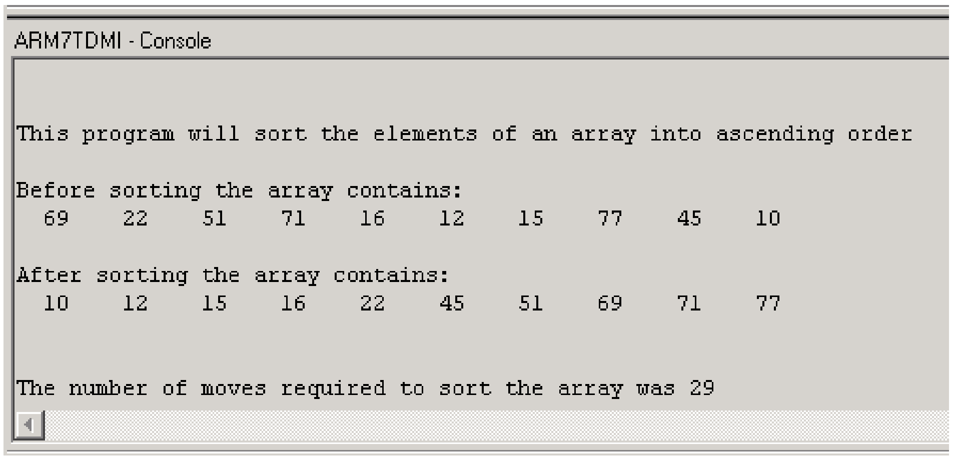
Another window, *C:\rvds22_tutorial\intro\session2\sort.c*, appears. This contains the source code relevant to the currently executing image. Execution has halted on the breakpoint at **`main()`**. AXD places a breakpoint here by default.

To view all current breakpoints, select SystemViews→ *Breakpoints* from the main menu.

All windows can be resized by clicking and dragging at the edges. Alternatively, right click in the title area of a window and select *Float within main window* for greater freedom when resizing windows.

Select *Execute→ Go* (*F5*) from the menu.

The program now completes. The results are printed to the *Console Window* where program input and output takes place.


ARM7TDMI - Console

```
This program will sort the elements of an array into ascending order

Before sorting the array contains:
  69    22    51    71    16    12    15    77    45    10

After sorting the array contains:
  10    12    15    16    22    45    51    69    71    77


The number of moves required to sort the array was 29
```

The disassembled code at the programs exit point is now visible in the *Disassembly Window.*

Quit AXD by selecting *File→ Exit*.

You can have multiple instances of AXD open, but each time it is launched from the IDE a new instance is opened; hence you can end up with a previous instance of AXD still running. It is therefore good practice to close down AXD after each debug session is complete.

Exercise A.2 - Simple Debugging

Re-start AXD, load the image **`sort.axf`** and run to the breakpoint at **`main()`**.

The example uses the local variable **`Moves`** to keep track of how many swap operations need to be performed on adjacent numbers to sort the array.

The function **`bubblesort`** contains two **`for`** loops used to reorder the elements of the array. How many moves occur in one iteration of the outer **`for`** loop?

Find the function body of **`bubblesort`,** by selecting *Low-Level Symbols* from the *Processor Views* menu and double clicking the **`bubblesort`** entry.

The order of symbol display can be chosen by right clicking within the symbol window. Breakpoints can be set on symbols by right clicking on their name and selecting *Toggle Break Point* from the pop-up menu.

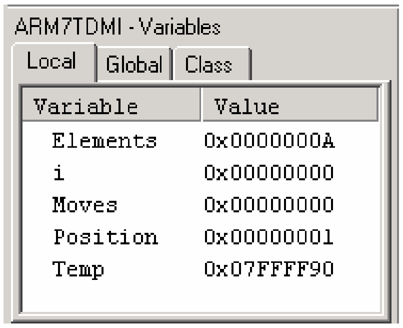
Set a breakpoint on the inner **`for`** loop at line **`34`** by double clicking in the grey region to the left of the statement.

The line will receive a red breakpoint marker.

Resume execution. The program will stop at the second breakpoint.

Display the local variables by selecting *Processor Views→ Variables*, or by pressing *Ctrl+F*.

A *Variables* window appears. Ensure the *Local* tab is selected.



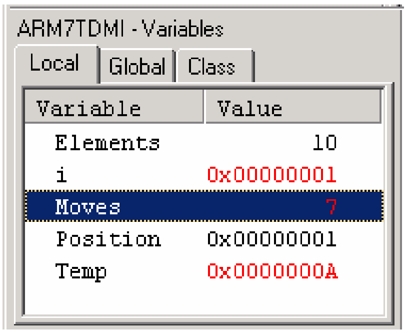
```
Variable      Value
  Elements    0x0000000A
  i           0x00000000
  Moves       0x00000000
  Position    0x00000001
  Temp        0x07FFFF90
```

Note: the values of the outer loop counter **`i`** and the swap operation variable **`Moves`** have been initialised to 0.

Resume execution. The program halts again after it completes the first iteration of the outer **`for`** loop.

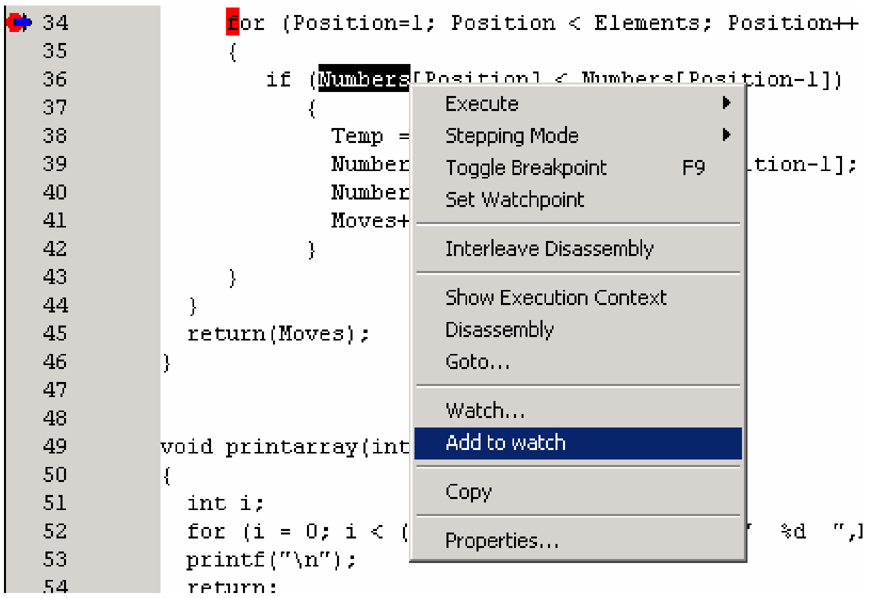
The local variables pane shows the updated value of **`Moves`**.

Right click on the **`Moves`** and **`Elements`** fields in turn and select *Format* → *Decimal* to change the display format of the variables:



Variable	Value
Elements	10
i	0x00000001
Moves	7
Position	0x00000001
Temp	0x0000000A

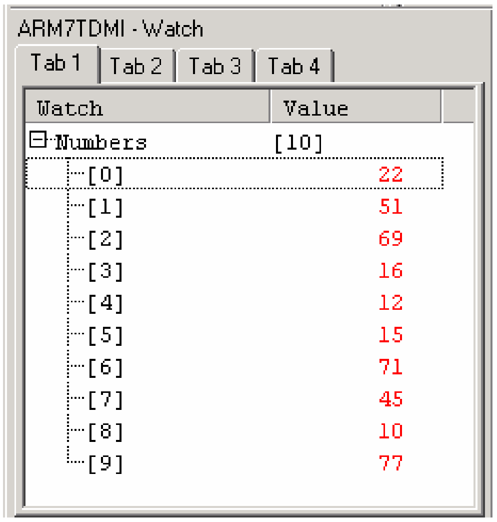
In the source window of AXD, double click on the variable **`Numbers`** at line 36 to select it. Right click and select *Add to watch*.



```
34          for (Position=1; Position < Elements; Position++
35          {
36              if (Numbers[Position] < Numbers[Position-1])
37              {
38                  Temp =
39                  Number                       tion-1];
40                  Number
41                  Moves+
42              }
43          }
44      }
45      return(Moves);
46  }
47
48
49  void printarray(int
50  {
51      int i;
52      for (i = 0; i < (                        %d   ",]
53      printf("\n");
54      return;
```

Expand the array **`Numbers`** in the Watch window and change the format of the elements so they are displayed in decimal

.



```
Watch          Value
⊟Numbers       [10]
   [0]              22
   [1]              51
   [2]              69
   [3]              16
   [4]              12
   [5]              15
   [6]              71
   [7]              45
   [8]              10
   [9]              77
```

Resume execution. The program halts after each iteration of the **`for`** loop.

Note the changes to the local variable **`Moves`** and the contents of the partially sorted contents of the array **`Numbers`** at each iteration.

Continue execution until the program exits.

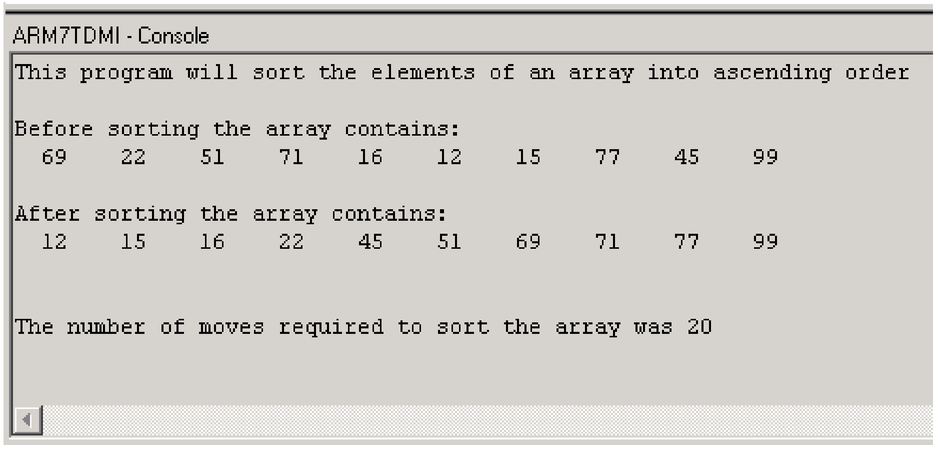
How is the total number of moves affected if the last element of the array is changed to 99?

Reload the image into the debugger and run to the first breakpoint at **`main().`**

In the watch window double click on the contents of the last element of the array and change the value to 99.

Remove the remaining breakpoint (on the **`for`** loop) at line 34, by double clicking in the grey region to the left of the statement.

Run the code to completion and note the new number of **`Moves`** printed in the console window.



```
This program will sort the elements of an array into ascending order

Before sorting the array contains:
  69    22    51    71    16    12    15    77    45    99

After sorting the array contains:
  12    15    16    22    45    51    69    71    77    99


The number of moves required to sort the array was 20
```

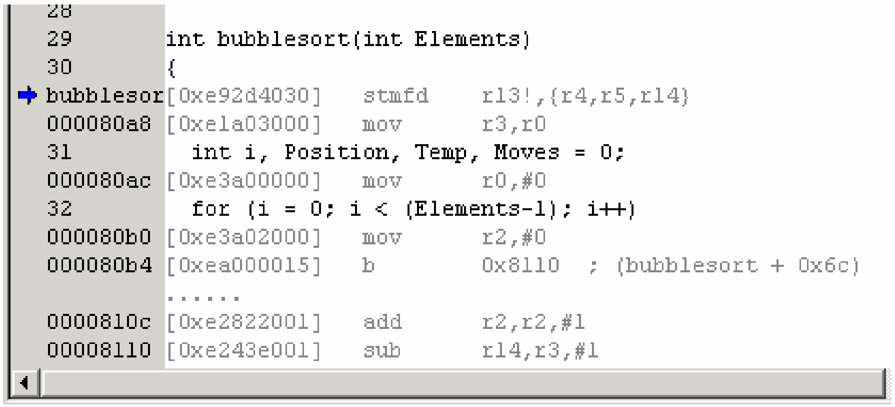
Exercise A.3 – Interleaving Disassemby and Viewing memory

Reload the image into the debugger and run to the first breakpoint at **`main()`.**

Select *Execute→ Step* (*F10*) and step through the program until you reach the call to **`bubblesort`** at line 19

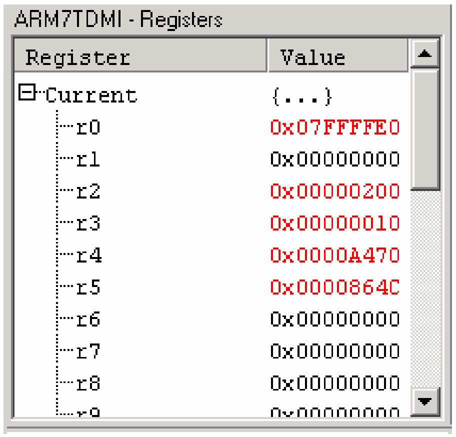
Select *Execute→ Step-In* (*F8*) to step into the function code

Right click on the source code window and select *Interleave disassembly*



```
28
29          int bubblesort(int Elements)
30          {
bubblesor[0xe92d4030]   stmfd    r13!,{r4,r5,r14}
000080a8 [0xe1a03000]   mov      r3,r0
31            int i, Position, Temp, Moves = 0;
000080ac [0xe3a00000]   mov      r0,#0
32            for (i = 0; i < (Elements-1); i++)
000080b0 [0xe3a02000]   mov      r2,#0
000080b4 [0xea000015]   b        0x8110   ; (bubblesort + 0x6c)
         ......
0000810c [0xe2822001]   add      r2,r2,#1
00008110 [0xe243e001]   sub      r14,r3,#1
```

View the current registers contents by selecting *Processor Views→ Registers*.



ARM7TDMI - Registers

Register	Value
Current	{...}
r0	0x07FFFFE0
r1	0x00000000
r2	0x00000200
r3	0x00000010
r4	0x0000A470
r5	0x0000864C
r6	0x00000000
r7	0x00000000
r8	0x00000000
r9	0x00000000

Step the code (*F10*) until the local variable **`Position`** appears in the watch window and has the value 3.

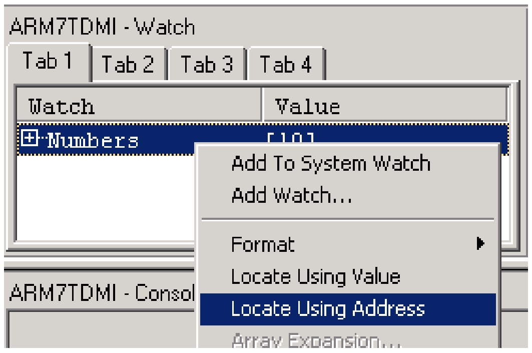
Note how the value of the registers used to keep track of the local loop variables change as you single step.

Return to a source level view by right clicking on the source code window and select *Interleave disassembly*

Complete execution of the function **`bubblesort`** and return to **`main()`** by clicking step-out (Shift F8)

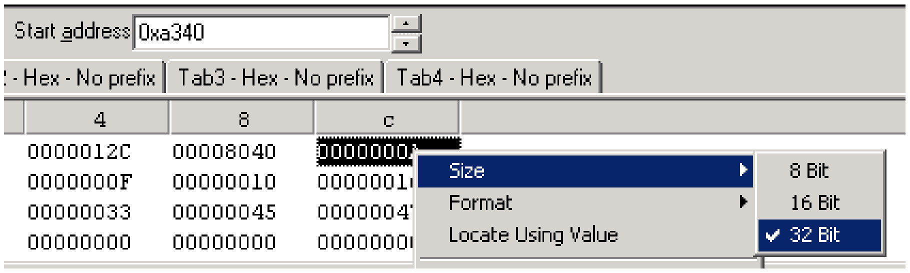
We will edit the contents of the array before the program prints the results.

Go to the *Watch Window*. Right click on the array **`Numbers`** displayed here and select *Locate Using Address*



A memory window opens with the contents of the first element of the array highlighted.

Right click in the *Memory Window* and change the size of the elements displayed to *32bit*.

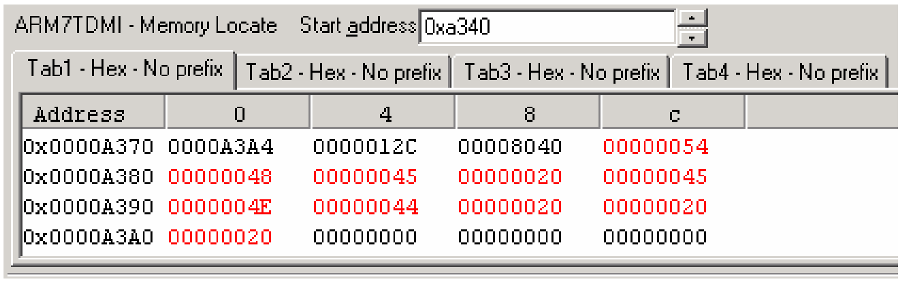


```
    4           8           c
0000012C    00008040    0000000
0000000F    00000010    0000001
00000033    00000045    0000004
00000000    00000000    0000000
```

In the *Watch Window* double click on the first element of the array to change it's value.

Change the values of the elements to have the following hex values: 0x54,0x48,0x45,0x20,0x45,0x4E,0x44,0x20,0x20,0x20

Note how changes to the array in the *Watch Window* are reflected in the *Memory Window* view.



ARM7TDMI - Memory Locate   Start address 0xa340

```
Address     0         4         8         c
0x0000A370  0000A3A4  0000012C  00008040  00000054
0x0000A380  00000048  00000045  00000020  00000045
0x0000A390  0000004E  00000044  00000020  00000020
0x0000A3A0  00000020  00000000  00000000  00000000
```

Change the format of the *Memory Window* to display ASCII, by right clicking.

You will see a message.

Restart the program and finish executing the example.

Quit AXD by selecting *File*→ *Exit*

Exercise A.4 – Using the AXD command line

In this exercise we will see how the tasks performed using the graphical interface can be replicated using the command line.

Re-start AXD, load the image **`sort.axf`**

Select *System Views* → *Command Line Interface* from the menu to open the *Command Line Interface* window.

Re-size any other windows as necessary to ensure the *Console* and *Command Line Interface* windows are in clear view.

Ensure the debugger *Command Line Interface* window is currently in focus then start program execution by using the **`go`** command at the **`Debug >`** prompt.

```
Debug > go
```

Once again execution halts on entry to **`main`** before the first instruction to be executed.

Set another breakpoint on line **`19`** of the source file by using the **`break`** command with **`sort.c`** as the file context qualifier, then resume program execution:

```
Debug > break sort.c|19
Debug > go
```

Execution will stop at the breakpoint. Now check the values of the program variables:

```
Debug > print Elements dec
```

The **`dec`** part of the **`print`** command specifies the format of the output generated.

Change the default output format of the debugger using the **`format`** command:

```
Debug > format dec
```

Use the **memory** command to view the contents of the array **Numbers** in memory.

```
Debug > memory @Numbers +0x28 32
```

The **+0x28** argument specifies how many bytes of memory are to be displayed. **32** specifies the memory display format in bits,

Change the first element of **Numbers** from **69** to **20** using the **let** command:

```
Debug > let Numbers[0] 20
```

Use the step command to step into the function **bubblesort().**

```
Debug > step in
```

Examine the contents of the **current** registers:

```
Debug > registers current
```

Use the step command to step through the first iteration of the loop and note how the local variables in registers change.

```
Debug > step
```

Remove the breakpoint on line **19** using the **unbreak** command (you can find the reference for the breakpoint you need using the **break** command which will print a list of current breakpoints):

```
Debug > unbreak #2
```

Use the **go** command to finish executing the program and print the results to the console.

```
Debug > go
```

Check the output is correct in the *Console* window then quit the debugger to finish the exercise.

Exercise A.5 – Using script files in AXD

In this exercise we will see how multiple commands can be combined in a *script file* to control execution within the debugger.

Consider the file **`sort.txt`** found in **`c:\rvds22_tutorial\intro\appendix\`**:

```
go
break sort.c|19
go
print Elements dec
format dec
memory @Numbers +0x28 32
let Numbers[0] 20
step in
registers current
step
unbreak #1
go
```

The file consists of a simple selection of commands which will perform the same task that was performed in the previous exercise.

Re-start AXD, load the image **`sort.axf`**

Ensure the debugger *Command Window* is currently in focus then invoke the script file by using the **`obey`** command as below:
`obey “c:\rvds22_tutorial\intro\appendix\sort.txt”`

When the program has terminated use the scroll bar on the right hand side of the *Command Line* window to view the values of the variables displayed by the script file.

Check the output in the *Console* window then quit the debugger to finish the exercise.

Appendix - Review

We have seen how the AXD debugger can be used to:

- Control and modify execution of code.
- View code at source and disassembly level.
- View and modify locals, globals and memory
- Accept commands via the CLI or from a script file to automate debugging.

A complete range of debugging facilities is available within AXD. Consult the online documentation for complete information.