ADS 1.2 Introductory Tutorial

Introduction

Aim

This tutorial provides the student with a basic introduction to the tools provided with the ARM Developer Suite version 1.2 (ADS). This will include the use of command line and GUI tools, to build and debug projects.

The tutorial is split into two practical sessions:

Session 1 – Introduction to the ARM command line tools.
Session 2 – Developing projects using CodeWarrior and AXD.

Pre-requisites

The student should be familiar with Microsoft DOS/Windows, and have a basic knowledge of the C programming language. The ARM Developer Suite (version 1.2) should be available.

Note: Explanation of File Extensions:

`.c`	C source file.
`.h`	C header file.
`.o`	object file.
`.s`	ARM or Thumb assembly language source file.
`.mcp`	ARM Project file, as used by the CodeWarrior IDE.
`.axf`	ARM Executable file, as produced by **`armlink`**.
`.txt`	ASCII text file.

Additional information

This tutorial is not designed to provide detailed documentation of ADS, as full on-line documentation is available. To access the on-line documentation:

From the *Start* menu select *Programs* → *ARM Developer Suite v1.2* → *Online Books*.

This opens a new window split into two panels. The left panel displays a 'Table of Contents' for the current level of documentation. The right panel shows the titles available within the collection selected in the left panel. Double click on any book topic to view more information. To search for a specific topic enter a search string in the Find field and press carriage return. This will then display the number of entries that each book contains. Select the book (by double clicking on the book name) and a new window will open with the left hand column now displaying the chapters and listing the number of entries per chapter. Scroll through each chapter to view each specific result found.

Further help can be accessed from the on-line documentation by pressing *F1* when running CodeWarrior or AXD, from the help menu, or by using the -help switch for a command line tool. The documentation is also available in PDF format in the **`PDF`** directory located within the ADS installation directory.

Icon conventions

Various icons are used throughout the tutorial to clarify the purpose of text associated with them. Icons either signify the presence of information on a particular topic, or the requirement for user interaction.

The following icons all indicate that user interaction is required:

	Indicates that command line input is required.
	Indicates other keyboard or mouse input is required.
	Button icon. This indicates that a corresponding button within the current application can be used to perform the operation currently being discussed.
	Application icon. Suggests an application to be used to perform a given operation. This example shows 'Microsoft Notepad'. To use Notepad from the command line type **`Notepad <filename>`**. Alternatively click on the Notepad icon on the 'Start menu' and open the required file using the *File→Open* command.

The following icons show information:

	Indicates a topic is also dealt with elsewhere in the tutorial.
	Suggests that further help is available from other resources.
	Identifies a user friendly hint or tip.
	Highlights important information regarding the current topic.
	Indicates the presence of a 'bug', logic or syntax error in code.

Session 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.

Only a brief reference is made to **armasm** in this tutorial. Further information is available in the ADS Assembler Workbook.

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**.

All the tools covered in this session are documented in the online text found under *ARM Developer Suite → Compiler, Linker and Utilities Guide*.

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

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**:**

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 = 0x00009fb8 (_sys_exit + 0x8)
+0008 0x00009fb8: 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 -fs`	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.

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.

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`** that was produced in Exercise 1.1:

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

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.

Session 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, ask an instructor or consult the online documentation for further information.

Session 2: CodeWarrior and AXD

In this session we will see how the CodeWarrior Integrated Development Environment can be used with AXD to create and develop projects.

CodeWarrior is only available on Windows platforms although AXD also exists on Unix versions of ADS.

Some new icons are introduced here. These correspond to buttons which can be clicked upon within the current application.

Activities covered in this session are documented in the online books found under *ARM Developer Suite →AXD and armsd Debuggers Guide*, in the chapters referring to *AXD*.

Help can be accessed from within AXD by pressing *F1*.

Exercise 2.1 - Creating a header file

In this exercise, we will create a new header file using CodeWarrior's built-in editor.

Start the IDE by clicking on the CodeWarrior IDE icon in the Windows Start Menu.

Select *File →New* from the menu.

Ensure the *File* tab is selected in the *New* dialog-box.

i) Select *Text File*.
ii) Click *OK*.

Enter the following C struct definition:

```
/* Struct definition */

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

Select *File→Save As* from the menu.

Navigate the directory structure to:
`c:\ads_tutorial\intro\session2\` and enter the filename **`datetype.h`**

Click *Save*. Click *Yes* to overwrite (if necessary).

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

Ensure the *Project* tab is selected in the *New* dialog-box.

i) Select *ARM Executable Image*.
ii) Enter the Project name: **`calendar`**

The *Project* tab contains the pre-defined project stationery that is available. Note that both images and object libraries are available as well as empty projects and a makefile wizard. Pre-defined project stationery is useful for quickly creating new projects.

Enter the Project directory: **`c:\ads_tutorial\intro\session2\`**
Use the *Set* button and click *Save* (if required)

Click *OK* to create the project.

The project window, *calendar.mcp* appears. The *Files* tab is highlighted and *DebugRel* is selected as the build target by default. Other build targets can be selected by clicking on the drop-down box.

The three default target variants available refer to the level of debug information contained in the resultant image.

Debug	Contains full debug table information and very limited optimization.
Release	No source level debug information, but full optimization.
DebugRel	A trade-off between the two.

It is the *DebugRel* variant that we shall use for the remainder of this tutorial.

We will now add our source files to the new project.

Highlight the *calendar.mcp* window. If it is not visible, select it from the *Window* menu.

From the menu select *Project* → *Add Files…*

Navigate to **`c:\ads_tutorial\intro\session2\`**

Double click on **`month.c`**. An *Add Files* window appears, click *OK*.

Highlight the editor window, *datetype.h*. If it is not visible, select it from the *Window* menu.

From the menu select *Project* → *Add datetype.h to Project*. Click *OK*.

If the editor window has been closed use *Project* → *Add Files…* again to locate *datetype.h*.

The *Files* tab of the project window now shows that **`datetype.h`** and **`month.c`** have been added to the project. Source files in the project window can be edited by double clicking on their icons. The *Code* and *Data* columns for **`month.c`** both contain **`0`** as no object code or executable image exists for the project as yet.

Exercise 2.3 - Building the project (DebugRel target)

Ensure the *DebugRel* target is selected from the drop-down menu at the top-left of the project window.

From the menu select *Project→Make* (or press *F7*).

An *Errors & Warnings* window appears with the several messages, the first two of which are shown below:

```
Error   : C2285E: expected ')' or ',' - inserted ')' before ';'
month.c line 17

Error   : (Serious) C2363E: member 'year' not found in 'struct
DateType'
month.c line 20
```

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

Double click on the first error message.

The IDE opens the editor and sets the focus on the line of code associated with the error. 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 appears again. The first error message is:

```
Error   : (Serious) C2363E: member 'year' not found in 'struct
DateType'
month.c line 20
```

Double click on the first error message.

The editor window is opened, with focus placed on the problem line. 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 **`DATETYPE`**, 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:

```
-DDATETYPE
```

Tools are configured graphically with CodeWarrior. We must edit the command line arguments for this target's settings:

Close the editor window, *'month.c'*.

Highlight the project window *'calendar.mcp'*.

From the menu select '*Edit→DebugRel Settings*'.

A *DebugRel Settings* window appears.

In the *Target Settings Panels* box, click *ARM C Compiler* (located in the *Language Settings* tree).

Each individual build *Target* allows the project to be built using different tool options.

The *ARM C Compiler* panel appears.

Select the *Preprocessor* tab. A list of *#DEFINEs* appears.
In the field below the definition list enter the following text:

```
DATETYPE
```

Click *Add*, then click *OK*.

(Note that it is also possible to enter the #DEFINE directly into the command line box displayed in the *ARM C Compiler* panel as **`-DDATETYPE`**)

Rebuild the project by pressing *F7* or selecting *Project → Make* from the menu.

The project window *Code* column now indicates that some code has been generated for our source file.

The project has been successfully built. The disassembled code for this file can be examined in CodeWarrior, via the **`fromelf`** decoding utility. CodeWarrior calls the utility and prints the output to the screen.

Highlight the project window *'calendar.mcp'*.

Right-click on **`month.c`** in the project widow and select *Disassemble* from the pop-up menu.

The disassembled code is displayed in a *Disassembly* window.

If a project is already up-to-date then nothing will be done by the IDE when it is requested to build a project. To get an explanatory message displayed when this is happening then select '*Edit→ Preferences*' from the menu. Select '*General→Build Settings*' and set the "*Show message after building up-to-date project*" checkbox. Click *OK* to save and close the IDE Preferences window.

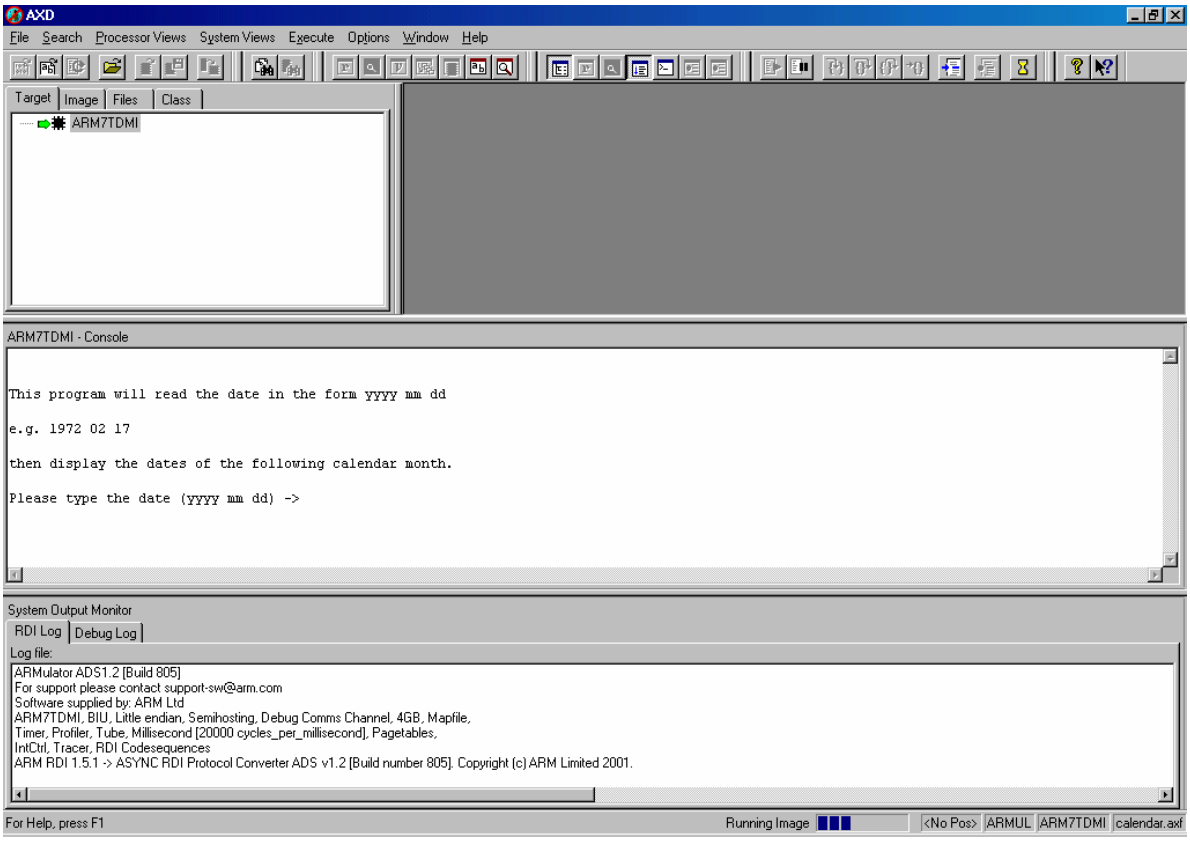
If you wish to do a forced rebuild of all of the source files then select '*Project→Remove Object Code...*'. Select 'All Targets' or 'Current Target' to delete the relevant object files.

Exercise 2.4 - Executing the example

Ensure the *DebugRel* target is selected in the project window.

From the menu select *Project→Run* (or press *Ctrl+F5*).

This will open AXD and run the image that the IDE has built in the instruction set simulator, 'ARMulator'. You will see a window similar to the one shown below:



In the center is the *Console Window* where program input and output takes place. Execution has already begun and the program is now awaiting user input.

Enter today's date in the format described, e.g. **`2000 04 19`**

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

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.

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 2.5 - Debugging the example

Select *Project→Debug* from the IDE menu (or press *F5*).

AXD will load the image ready for debugging, and will have set a breakpoint on **`main`**. The disassembled project code is visible in the *Disassembly* window.

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

Execution will halt on the entry to the **`main`** function in **`month.c`**.

Another window, *C:\ads_tutorial\intro\session2\month.c*, appears. This contains the source code relevant to the currently executing image.

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

You will once again be prompted to enter a date.

This time enter **`2000 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 *Console Window* to view the dates at the end of November. You will find that there is an extra day!

Reload the image into the debugger by selecting *File →Reload Current Image* from the menu.

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

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

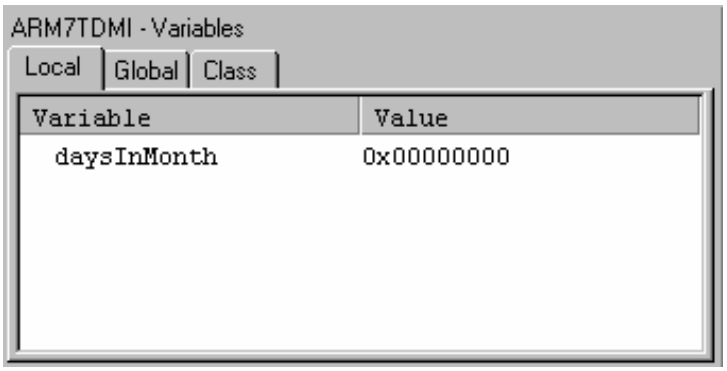
Set a breakpoint on the **`switch`** statement on line **`40`** by double clicking in the gray region to the left of the statement.

The line will receive a red breakpoint marker.

Resume execution and enter the date **`2000 11 30`** again. 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. The window contains our variables: **`daysInMonth`**. Its value has not been determined yet and it is currently set to **`0`**:

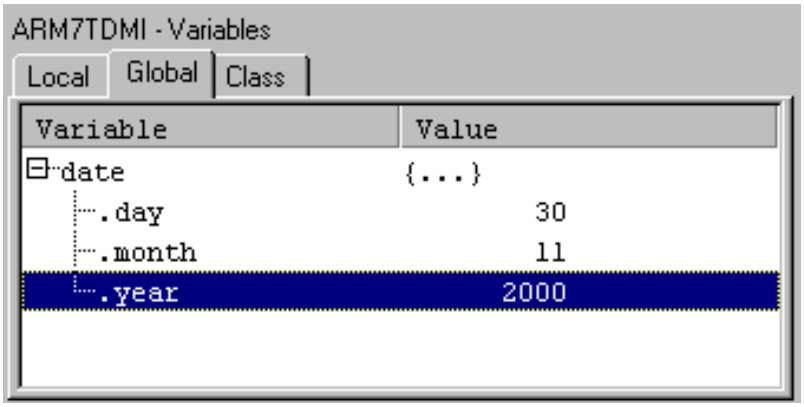


Click on the *Global* tab to display the global variables.

The display is updated. The global variables, including the **`date`** struct are now visible.

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

Right click on the **day**, **month** and **year** fields in turn and select *Format* → *Decimal* to change the display format of the variables:



Select *Execute* → *Step* (*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.

Click on the *Local* tab in the *Variables* window again. If necessary change the window format to *Decimal*.

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

Double 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.

Quit AXD by selecting *File*→*Exit*.

Exercise 2.6 – Viewing registers and memory

Select *Project→Debug* from the IDE menu.

AXD will load the image ready for debugging, and will have set a breakpoint on **`main`**. The disassembled project code is visible in the *Disassembly* window.

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

Execution will halt on the entry to the **`main`** function in **`month.c`**.

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

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

You will once again be prompted to enter a date.

This time enter **`2000 12 25`**.

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

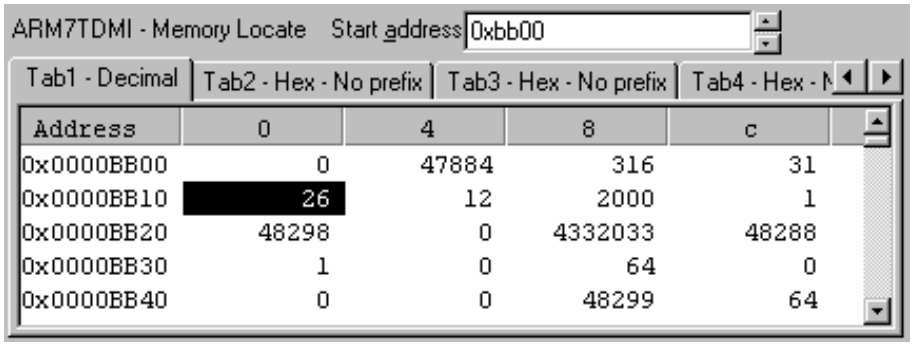
Open the *Low-Level Symbols* window from the *Processor Views* menu. Locate the **`date`** entry in the *Symbol* column.

Right click on the **`date`** entry and select *Locate using Address* from the context menu to view the variable in memory.

Right click on the highlighted values in the *Memory* window and select *Format → Other → Size 32 → Decimal*

Note how the highlighted word and the two successive words in memory correspond to the three fields in the date struct (**`26/12/2000`**). See image below.

The memory window should appear as follows:



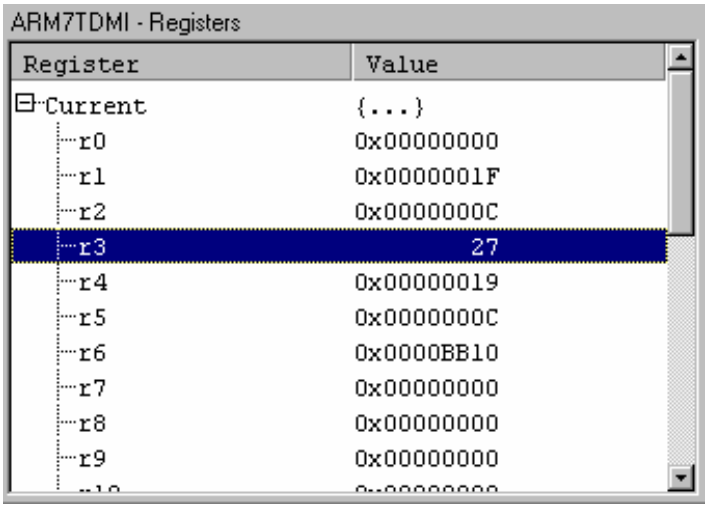
Address	0	4	8	c
0x0000BB00	0	47884	316	31
0x0000BB10	26	12	2000	1
0x0000BB20	48298	0	4332033	48288
0x0000BB30	1	0	64	0
0x0000BB40	0	0	48299	64

Open the *Registers* window from the *Processor Views* menu and click on the cross to the left of **`current`** to expand the view.

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

Right click on the **`r3`** register in the *Register* window. Select *Format → Decimal* from the context menu to change the display format.

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.):



Register	Value
Current	{...}
r0	0x00000000
r1	0x0000001F
r2	0x0000000C
r3	27
r4	0x00000019
r5	0x0000000C
r6	0x0000BB10
r7	0x00000000
r8	0x00000000
r9	0x00000000

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

Double click on the highlighted value **2** in the *Memory* window to edit the value. Change it to **22** and press Enter.

Use the *Go* button (or press *F5*) to pass through the while loop until the program ends.

Note how the value entered in memory affects the value in the register **r3** and the program output.

Quit AXD by selecting *File*→*Exit*.

Exercise 2.7 - 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. This is easily achieved in AXD.

Select *Project→ Debug* from the IDE menu.

Start executing the image by selecting *Execute→ Go* from the AXD menu (*F5*).

The program will run to the first breakpoint at main and the source code for **`month.c`** will come into view.

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

Step through the code until you have passed the date entry point and the next two days have been output. (*F10*).

Quit AXD by selecting *File→Exit*.

Exercise 2.8 – Using the command line

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

Select *Project→ Debug* from the IDE menu to launch AXD.

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 **`40`** of the source file by using the **`break`** command with **`month.c`** as the file context qualifier, then resume program execution:

```
Debug > break month.c|40
Debug > go
```

Enter the date **`2000 11 30`** in the *Console* window when prompted.

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

```
Debug > print daysInMonth dec
```

Note the value of **`daysInMonth`** is zero as it is a *static* variable and has not yet been initialized.

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
```

Check the values of the program variables:

```
Debug > print date.day
Debug > print date.month
Debug > print date.year
```

Remove the breakpoint on line **40** 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
```

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

```
Debug > break month.c|58
Debug > go
```

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

```
Debug > print daysInMonth
```

Correct the value from **31** to **30** using the **let** command:

```
Debug > let daysInMonth 30
```

Use the **go** command to pass through the while loop until the output displays the date **2000 12 3**

```
Debug > go
```

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

```
Debug > memory @date +0xc 32
```

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

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

Use the step command to execute the next instruction:

```
Debug > step
```

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

```
Debug > registers current
```

Note how the values in **r0** and **r1** correspond to the variables that were used to evaluate the **if** statement in the previous instruction.

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

```
Debug > unbreak #2
Debug > go
```

The program terminates normally.

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

Exercise 2.9 – 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 **`month.txt`** found in **`c:\ads_tutorial\intro\session2\`**:

```
go
break month.c|40
go
print daysInMonth dec
format dec
print date.day
print date.month
print date.year
unbreak #2
break month.c|58
go
print daysInMonth
let daysInMonth 30
go
memory @date +0xc 32
step
registers current
unbreak #2
go
```

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

Select *Project→Debug* (*F5*) from the IDE menu to launch AXD.

Ensure the debugger *Command Window* is currently in focus then invoke the script file by using the **`obey`** command.

`Debug:` **`obey c:\ads_tutorial\intro\session2\month.txt`**

Enter the date **`2000 11 30`** in the *Console* window when prompted.

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

Session 2 - Review

- We have now seen how the IDE can be used to create source files and create projects and also how to add source files to projects.
- The IDE allows automatic invoking of the compiler and linker to generate executable images.
- The IDE includes a powerful customizable editor that can be invoked by clicking on a filename from a project, or a compilation warning or error message.

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