



# Accessing memory-mapped peripherals with Arm DS

Version 1.0

## Non-Confidential

Copyright © 2022 Arm Limited (or its affiliates).  
All rights reserved.

## Issue 01

102635\_0100\_01\_en



# Accessing memory-mapped peripherals with Arm DS

Copyright © 2022 Arm Limited (or its affiliates). All rights reserved.

## Release information

### Document history

Issue	Date	Confidentiality	Change
0100-01	9 December 2022	Non-Confidential	Initial release

## Proprietary Notice

This document is protected by copyright and other related rights and the practice or implementation of the information contained in this document may be protected by one or more patents or pending patent applications. No part of this document may be reproduced in any form by any means without the express prior written permission of Arm. No license, express or implied, by estoppel or otherwise to any intellectual property rights is granted by this document unless specifically stated.

Your access to the information in this document is conditional upon your acceptance that you will not use or permit others to use the information for the purposes of determining whether implementations infringe any third party patents.

THIS DOCUMENT IS PROVIDED "AS IS". ARM PROVIDES NO REPRESENTATIONS AND NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, SATISFACTORY QUALITY, NON-INFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE WITH RESPECT TO THE DOCUMENT. For the avoidance of doubt, Arm makes no representation with respect to, and has undertaken no analysis to identify or understand the scope and content of, patents, copyrights, trade secrets, or other rights.

This document may include technical inaccuracies or typographical errors.

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF ARM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

This document consists solely of commercial items. You shall be responsible for ensuring that any use, duplication or disclosure of this document complies fully with any relevant export laws and regulations to assure that this document or any portion thereof is not exported, directly

or indirectly, in violation of such export laws. Use of the word “partner” in reference to Arm’s customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of the Agreement shall prevail.

The Arm corporate logo and words marked with ® or ™ are registered trademarks or trademarks of Arm Limited (or its affiliates) in the US and/or elsewhere. All rights reserved. Other brands and names mentioned in this document may be the trademarks of their respective owners. Please follow Arm’s trademark usage guidelines at <https://www.arm.com/company/policies/trademarks>.

Copyright © 2022 Arm Limited (or its affiliates). All rights reserved.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349|version 21.0)

## Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Unrestricted Access is an Arm internal classification.

## Product Status

The information in this document is for a product under development.

## Feedback

Arm® welcomes feedback on this product and its documentation. To provide feedback on the product, create a ticket on <https://support.developer.arm.com>

To provide feedback on the document, fill the following survey: <https://developer.arm.com/documentation-feedback-survey>.

## Inclusive language commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used language that can be offensive. Arm strives to lead the industry and create change.

We believe that this document contains no offensive language. To report offensive language in this document, email [terms@arm.com](mailto:terms@arm.com).

# Contents

<b>1. Overview.....</b>	<b>6</b>
<b>2. Arm recommendations.....</b>	<b>12</b>
<b>3. Aligning registers.....</b>	<b>13</b>
<b>4. Mapping variables to specific addresses.....</b>	<b>15</b>
4.1 Test using a struct at a specific address.....	15
4.2 Add the scatter-loading description easily to an Arm DS project.....	16
<b>5. Code efficiency.....</b>	<b>19</b>

# 1. Overview

In this tutorial, learn about mapping a C variable to each register of a memory-mapped peripheral, then using a pointer to that variable to read and write the register.

In most Arm embedded systems, peripherals are at specific addresses in memory. It is often convenient to map a C variable onto each register of a memory-mapped peripheral. Then, use a pointer to that variable to read and write the register. In your code, you must consider not only the size and address of the register, but also its alignment in memory.

## Before you begin

- Install and license [Arm Development Studio \(Arm DS\)](#). For more information, see the [Arm Development Studio documentation](#).



You can test the examples in this tutorial by using the Cortex-A53x1 FVP, Base\_A53x1, with Arm Compiler 6. Both Arm Compiler 6 and the Cortex-A53x1 FVP are included with Arm DS.

- Download this [Arm DS example project](#) (Arm DS example project) that includes the code from this tutorial and a debug launch configuration. Use this project to modify, build, and debug the examples from this tutorial.



Arm strongly recommends that you complete the [Hello World Arm DS Tutorial](#) before working with the examples in this tutorial.

---

## Using the examples in this tutorial

The examples in this tutorial use a little-endian memory system.

- For 32-bit registers, `unsigned int`.
- For 16-bit registers, `unsigned short`.
- For 8-bit registers, `unsigned char`.

The compiler generates the appropriate single load and store instructions, that is `LDR` and `STR` for 32-bit registers, `LDRH` and `STRH` for 16-bit registers, and `LDRB` and `STRB` for 8-bit registers.

You must also ensure that the memory-mapped registers lie on appropriate address boundaries, that is either all word-aligned, or aligned on their natural size boundaries. For example, 16-bit registers must be aligned on halfword addresses.



Arm recommends that all registers, whatever their size, be aligned on word boundaries.

You can also use `#define` to simplify your code:

```
#define PORTBASE 0xC0000000 /* Counter/Timer Base */
#define PortLoad ((volatile unsigned int *) PORTBASE) /* 32 bits */
#define PortValue ((volatile unsigned short *) (PORTBASE + 0x04)) /* 16 bits */
#define PortClear ((volatile unsigned char *) (PORTBASE + 0x08)) /* 8 bits */

void init_regs(void)
{
    unsigned int int_val;
    unsigned short short_val;
    unsigned char char_val;

    *PortLoad = (unsigned int) 0xF00FF00F;
    int_val = *PortLoad;

    *PortValue = (unsigned short) 0x0000;
    short_val = *PortValue;

    *PortClear = (unsigned char) 0x1F;
    char_val = *PortClear;
}
```

This code results in the following (interleaved) code:

```
;;;5      void init_regs(void)
000000    e59f1024 LDR r1,[L1.44]
;;;6      {
;;;7          unsigned int int_val;
;;;8          unsigned short short_val;
;;;9          unsigned char char_val;
;;;10         *PortLoad = (unsigned int) 0xF00FF00F;
000004    e3a00101 MOV r0,#0xC0000000
000008    e5801000 STR r1,[r0,#0]
;;;11         int_val = *PortLoad;
00000c    e5901000 LDR r1,[r0,#0]
;;;12         *PortValue = (unsigned short) 0x0000;
000010    e3a01000 MOV r1,#0
000014    e1c010b4 STRH r1,[r0,#4]
;;;13         short_val = *PortValue;
000018    e1d010b4 LDRH r1,[r0,#4]
;;;14         *PortClear = (unsigned char) 0x1F;
00001c    e3a0101f MOV r1,#0x1f
000020    e5c01008 STRB r1,[r0,#8]
;;;15         char_val = *PortClear;
000024    e5d00008 LDRB r0,[r0,#8]
;;;16     }
000028    e12ffffe BX lr
```

If you debug the previous code with the Cortex-A53 FVP in Arm DS, you can see the generated disassembly code in the Disassembly view.



The instructions used might slightly differ from the previous interleaved code.

**Figure 1-1: Code in Disassembly view**

Console Commands Variables Registers Memory Disassembly +				
<Next Instruction>				100
Address	Opcode	Disassembly		
		<b>init_regs</b>		
EL3:0x00000000800000E8	D10043FF	SUB	sp, sp, #0x10	
EL3:0x00000000800000EC	D2B80008	MOV	x8, #0xc0000000	
EL3:0x00000000800000F0	32049FE9	ORR	w9, wzr, #0xf00ff00f	
EL3:0x00000000800000F4	B9000109	STR	w9, [x8, #0]	
EL3:0x00000000800000F8	B9400108	LDR	w8, [x8, #0]	
EL3:0x00000000800000FC	B9000FE8	STR	w8, [sp, #0xc]	
EL3:0x0000000080000100	D2800088	MOV	x8, #4	
EL3:0x0000000080000104	F2B80008	MOVK	x8, #0xc000, LSL #16	
EL3:0x0000000080000108	7900011F	STRH	wzr, [x8, #0]	
EL3:0x000000008000010C	79400108	LDRH	w8, [x8, #0]	
EL3:0x0000000080000110	790017E8	STRH	w8, [sp, #0xa]	
EL3:0x0000000080000114	D2800108	MOV	x8, #8	
EL3:0x0000000080000118	F2B80008	MOVK	x8, #0xc000, LSL #16	
EL3:0x000000008000011C	528003E9	MOV	w9, #0x1f	
EL3:0x0000000080000120	39000109	STRB	w9, [x8, #0]	
EL3:0x0000000080000124	39400108	LDRB	w8, [x8, #0]	
EL3:0x0000000080000128	390027E8	STRB	w8, [sp, #9]	
EL3:0x000000008000012C	910043FF	ADD	sp, sp, #0x10	
EL3:0x0000000080000130	D65F03C0	RET		
EL3:0x0000000080000134	00000000	UDF	#0	



The instructions LDR, STR, LDRH, STRH, LDRB, and STRB, the compiler generates as type casting (unsigned int, unsigned short, and unsigned char) which is used to store the wanted values in the peripheral port registers.

It is now possible to debug the code to check the wanted values are correctly written to the wanted memory addresses that correspondent to the peripheral registers. Before executing `init_regs()`, open the Memory view in Arm DS and then search for the port base address of your peripheral, in this case `0xc0000000`. It is observed that the relevant memory addresses only contain uninitialized values:



Figure 1-2: Code in dissassembly view

Console				Commands				Variables				Registers				Memory				Disassembly				+							
<Next Instruction>																		100													
Address																		Opcode				Disassembly									
																						init_regs									
EL3:0x00000000800000E8																		D10043FF				SUB sp,sp,#0x10									
EL3:0x00000000800000EC																		D2B80008				MOV x8,#0xc0000000									
EL3:0x00000000800000F0																		32049FE9				ORR w9,wzr,#0xf00ff00f									
EL3:0x00000000800000F4																		B9000109				STR w9,[x8,#0]									
EL3:0x00000000800000F8																		B9400108				LDR w8,[x8,#0]									
EL3:0x00000000800000FC																		B9000FE8				STR w8,[sp,#0xc]									
EL3:0x0000000080000100																		D2800088				MOV x8,#4									
EL3:0x0000000080000104																		F2B80008				MOVK x8,#0xc000,LSL #16									
EL3:0x0000000080000108																		7900011F				STRH wzr,[x8,#0]									
EL3:0x000000008000010C																		79400108				LDRH w8,[x8,#0]									
EL3:0x0000000080000110																		790017E8				STRH w8,[sp,#0xa]									
EL3:0x0000000080000114																		D2800108				MOV x8,#8									
EL3:0x0000000080000118																		F2B80008				MOVK x8,#0xc000,LSL #16									
EL3:0x000000008000011C																		528003E9				MOV w9,#0x1f									
EL3:0x0000000080000120																		39000109				STRB w9,[x8,#0]									
EL3:0x0000000080000124																		39400108				LDRB w8,[x8,#0]									
EL3:0x0000000080000128																		390027E8				STRB w8,[sp,#9]									
EL3:0x000000008000012C																		910043FF				ADD sp,sp,#0x10									
EL3:0x0000000080000130																		D65F03C0				RET									
EL3:0x0000000080000134																		00000000				UDF #0									

Stepping through the code, when `*PortLoad = (unsigned int) 0xF00FF00F;` is executed, the instruction `STR w9, [x8, #0]` is executed to store the wanted value, `0xF00FF00F`, into the peripheral register `PortLoad`, at the address `0xC0000000`:

Figure 1-3: Stepping through the code

```

void init_regs(void)
{
    unsigned int int_val;
    unsigned short short_val;
    unsigned char char_val;

    *PortLoad = (unsigned int) 0xF00FF00F;
    int_val = *PortLoad;

    *PortValue = (unsigned short) 0x0000;
    short_val = *PortValue;

    *PortClear = (unsigned char) 0x1F;
    char_val = *PortClear;

    return;
}

```

**Figure 1-4: Storing a wanted value**

Console Commands Variables Registers Memory Disassembly Target Console +			
<Next Instruction>			100
Address	Opcode	Disassembly	
EL3: 0x00000000800000E8	D10043FF	SUB sp, sp, #0x10	
EL3: 0x00000000800000EC	D2B80008	MOV x8, #0xc0000000	
EL3: 0x00000000800000F0	32049FE9	ORR w9, wzr, #0xf00ff00f	
EL3: 0x00000000800000F4	B9000105	STR w9, [x8, #0]	

Open the Memory view to see that the wanted value is set to the peripheral register:

**Figure 1-5: Wanted value set to peripheral register**

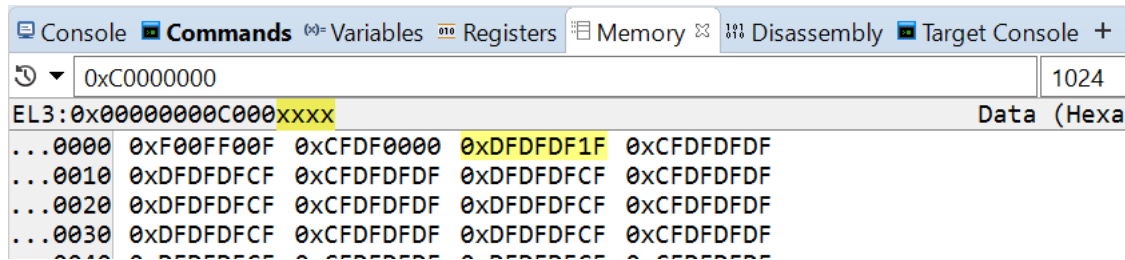
Console Commands Variables Registers Memory Disassembly Target Console +			
0xC0000000			1024
EL3: 0x00000000C000xxxx			Data (Hexade
...0000	0xF00FF00F	0xCFDFDFDF	0xDFDFDFCF 0xCFDFDFDF
...0010	0xDFDFDFCF	0xCFDFDFDF	0xDFDFDFCF 0xCFDFDFDF
...0020	0xDFDFDFCF	0xCFDFDFDF	0xDFDFDFCF 0xCFDFDFDF
...0030	0xDFDFDFCF	0xCFDFDFDF	0xDFDFDFCF 0xCFDFDFDF

Continue stepping until the instruction `STRH wzr, [x8, #0]` is reached. This instruction is executed to store the value `0x0000` in the `PortValue` register, mapped to the address `0xC0000004`. In this case, the value set on the port register is half a word width (16-bits). When viewing memory at a word width, after the halfword write, the bottom 16-bits change and the top 16-bits remain the same. In this case, the bottom 16-bits change to `0x0000` and the top 16-bits remain `0xCFDF` like the following:

**Figure 1-6: Viewing memory at a word width after the halfword write**

Console Commands Variables Registers Memory Disassembly Target Console +			
0xC0000000			1024
EL3: 0x00000000C000xxxx			Data (Hexa
...0000	0xF00FF00F	0xCFDF0000	0xDFDFDFCF 0xCFDFDFDF
...0010	0xDFDFDFCF	0xCFDFDFDF	0xDFDFDFCF 0xCFDFDFDF
...0020	0xDFDFDFCF	0xCFDFDFDF	0xDFDFDFCF 0xCFDFDFDF
...0030	0xDFDFDFCF	0xCFDFDFDF	0xDFDFDFCF 0xCFDFDFDF

Continue stepping to the next relevant instruction of `STRB w9, [x8, #0]`. This instruction is executed to store the value `0x1F` in the `PortClear` register, mapped to the address `0xC0000008`. In this case, the value set on the port register is a byte wide (8-bits). When viewing memory at a word width, after the byte write, the bottom 8-bits change and the top 24 bits (3 bytes) remain the same. In this case, the bottom 8-bits change to `0x1F` and the top 3 bytes remain `0xDFDFDF` like the following:

**Figure 1-7: Viewing memory at a word width after the byte write**

To check the variables are stored to the correct peripheral addresses, in the Variables view, look at the local variables `int_val`, `short_val`, and `char_val`. Look at the values, types, and sizes of the three variables.

Tip: Even if the sizes are different, word-alignment (32-bits) is respected when storing the values in the peripheral registers. Notice that the locations of these variables are not the locations of the peripheral registers. These variables are only local variables to check that the contents of the peripheral registers are the expected values.

**Figure 1-8: Local variables to check that the contents of the peripheral registers**

Console Commands Variables Registers Memory Disassembly Target Console +						
Name	Value	Type	Count	Size	Location	Access
Locals	3 variables					
int_val	4027576335	unsigned int		32	EL3:0x00000000FFFEFFDC	R/W
short_val	0	unsigned short		16	EL3:0x00000000FFFEFFDA	R/W
char_val	'\0'	unsigned char		8	EL3:0x00000000FFFEFFD9	R/W

## 2. Arm recommendations

Arm recommends word alignment of peripheral registers even if they are 16-bit or 8-bit peripherals. In a little-endian system, the peripheral databus can connect directly to the least significant bits of the Arm databus. There is no need to multiplex or duplicate the peripheral databus onto the high bits of the Arm databus. In a big-endian system, the peripheral databus can connect directly to the most significant bits of the Arm databus. There is no need to multiplex or duplicate the peripheral databus onto the low bits of the Arm databus.

The Arm AMBA APB bridge uses the preceding recommendation to simplify the bridge design. The result of this is that only word-aligned addresses must be used for any width transfer. A read results in unused values on any bits which are not connected to the peripheral. So, if a 32-bit word is read from a 16-bit peripheral, the top 16 bits of the register value must be cleared before use.

For example, to access some 16-bit peripheral registers on a 16-bit alignment, you might write:

```
volatile unsigned short ul6_IORegs[20];
```

This code works if your peripheral controller logic can route the peripheral databus to the high part (D31-D16) and low part (D15-D0) of the Arm databus. Which part is used depends on which address you are accessing. To use this code, check if this multiplexing logic exists in your design. The standard Arm APB bridge does not support this multiplexing logic.

### 3. Aligning registers

If you want to map 16-bit registers on 32-bit alignment as recommended, you could use one of these methods:

1. `volatile unsigned short u16_IORegs[40];`

This code only allows accesses to even-numbered registers (each index in the array corresponds to 16-bits). You must double the register number. For example, to access the fourth register you could use index 8:

```
x = u16_IORegs[8];
u16_IORegs[8] = newval;
```

2. `volatile unsigned int u32_IORegs[20];`

The registers are accessed as 32-bit values. But a simple peripheral controller, like an Arm AMBA APB bridge, reads unused values into the top bits from signals that are not connected to the peripheral. In a little-endian system, the used values map to D31-D16. So, when such a peripheral is read, it must be cast to an unsigned short to get the compiler to discard the upper 16-bits.

This type casting effect was the case shown in the previous example in this documentation. This example used a casting to read the peripheral registers and save their contents in the variables `int_val`, `short_val`, and `char_val`.

For example, to access peripheral register 4:

```
x = (unsigned short)u32_IORegs[4];
u32_IORegs[4] = newval;
```

3. Use a `struct`.

Using a `struct` allows descriptive names to be used and can accommodate different peripheral register widths.



The padding is made explicit rather than relying on automatic padding added by the compiler. For example:

```
struct PortRegs {
    unsigned short ctrlreg; /* offset 0 */
    unsigned short dummy1;
    unsigned short datareg; /* offset 4 */
    unsigned short dummy2;
    unsigned int data32reg; /* offset 8 */
} iospace;

x = iospace.ctrlreg;
iospace.ctrlreg = newval;
```



The peripheral locations must not be accessed using:

- 
- a. `__packed structs` where unaligned members are allowed and there is no internal padding
  - OR
  - b. C bitfields, for example, by defining the variable `unsigned int isCorrect: 1`.

If the previous access methods are used, the compiler only uses 1-bit to store data, instead of 32-bits. This 1-bit data store is common when storing the data as a Boolean. However, if you use the previous methods, it is not possible to control the number and type of memory accesses the compiler performs. This can result in code that is non-portable, has undesirable side-effects, and does not work as intended. The recommended way of accessing peripherals is through explicit use of architecturally defined types such as `int`, `short`, and `char` on their natural alignments.

## 4. Mapping variables to specific addresses

Memory mapped registers can be accessed from C in two ways:

- [Force an array or struct variable to a specific address](#)
- [Using a pointer to an array or struct](#)

Both previous methods generate efficient code. Choose the method that you prefer.

### Force an array or struct variable to a specific address

The array or struct variable is declared in a file on its own. When it is compiled, the object code for this file only contains data. If using the Arm Compiler, this data can be placed at a specified address using the Arm scatter-loading mechanism. Arm recommends this scatter-loading mechanism as the method for placing all scatter-loading AREAS at required locations in the memory map.

### 4.1 Test using a struct at a specific address

Test the recommended scatter-loading mechanism.

1. In the example project, open `src > iovar.c`.

The `iovar.c` file contains a declaration of the array or `struct` variable:

```
struct{
    volatile unsigned reg1;
    volatile unsigned reg2;
} mem_mapped_reg;
```

2. Open `scatter.txt`.

The `scatter.txt` file contains the following:

```
ALL 0x80000000
{
    ALL 0x80000000
    {
        * (+RO,+RW,+ZI)
    }
}
IO 0xC0000000
{
    IO 0xC0000000
    {
        iovar.o (+ZI)
    }
}
```

The scatter-loading description file must be specified at link time to the linker using the `--scatter scatter.txt` command-line option. This description creates two different load regions in your image: `ALL` and `io`. The zero-initialized area (ZI) from `iovar.o` containing the `struct`,

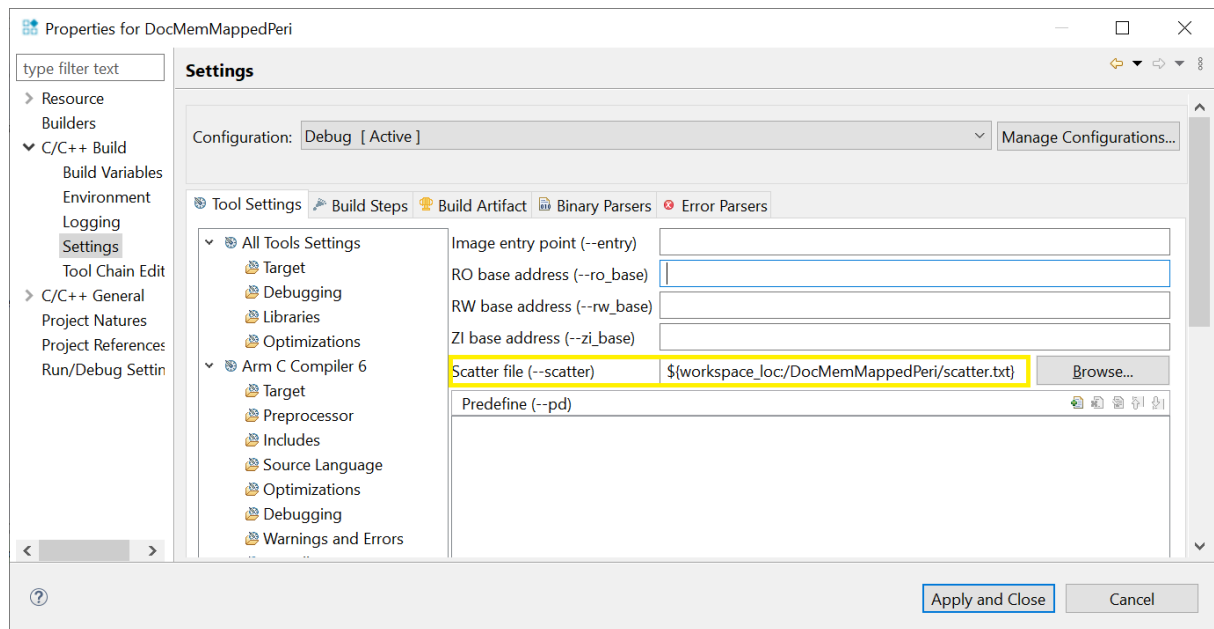
goes into the I/O area at 0xC0000000. All code (RO) and data areas (RW and ZI) from other object files go into the `ALL` region which starts at 0x80000000.

## 4.2 Add the scatter-loading description easily to an Arm DS project

Use this method to add the scatter-loading description to a project.

1. In the Project Explorer view, right-click your project folder, and click Properties.
2. In C/C++ Build > Settings> Tool Settings, click Image Layout under Arm Linker 6.
3. Click Browse to select the scatter file, and click Apply and Close to save the changes.

**Figure 4-1: Adding a scatter-loading file to a project**



In the example project, the scatter-loading file has already been added to the project.

If you have more than one set of memory mapped registers, you must define each group of variables as a separate execution region. It is possible that all the memory mapped registers could be within a single load region. To define each variable group as a separate execution region, each group of variables must be defined in a separate module.

The benefit of using a scatter-loading description file is that all the (target-specific) absolute addresses chosen for your devices, code, and data are in one file. Because everything is in one



file, maintenance is easy. Furthermore, if you decide to change your memory map, for example, if peripherals are moved, you do not need to rebuild your entire project. You just run the link step on the existing object files.

For more documentation on scatter-loading, check the following links:

- See [the scatter-loading mechanism chapters](#) in the Arm Compiler Reference Guide
- See [the scatter-loading images with a simple memory map chapter](#) in the Arm Compiler Reference Guide
- See [the scatter File Syntax chapters](#) in the Arm Compiler Reference Guide

Alternatively, use the `#pragma arm section` pragma to place the data into a specific section. Then, use scatter-loading to place that data at an explicit location. See [the pragmas chapter](#) on Arm Developer.

4. In the example project `DocMemMappedPeri.c`, uncomment the following code:

```
extern struct{
    volatile unsigned reg1;
    volatile unsigned reg2;
} mem_mapped_reg;
int main(void) {
    ...
    mem_mapped_reg.reg1 = (unsigned int) 0xF00FF00F;
    mem_mapped_reg.reg2 = (unsigned int) 0x100CF00F;
    ...
}
```

And comment this line in `main()`:

```
//init_regs();
```

5. Clean and build the example project.
6. Launch the Cortex-A53 FVP.
7. Step through the code.

The fields `reg1` and `reg2` in the struct variable `mem_mapped_reg` are now mapped to the addresses of the peripheral registers.

When writing to these variables, you are directly writing to the peripheral registers. Enter the address of the peripheral registers, `0xC0000000`, into the Memory view. Observe how the data, `0xF00FF00F` and `0x100CF00F`, is correctly written by using the memory-mapped variables:

### Figure 4-2: Memory view after struct write

## Using a pointer to an array or struct

```

struct PortRegs {
    unsigned short ctrlreg; /* offset 0 */
    unsigned short dummy1;
    unsigned short datareg; /* offset 4 */
    unsigned short dummy2;
    unsigned int data32reg; /* offset 8 */
};

volatile struct PortRegs *iospace = (struct PortRegs *)0xC0000000;
x = iospace->ctrlreg;
iospace->ctrlreg = newval;

```

The pointer can be either local or global. If global, to avoid having the base pointer reloaded after function calls, make `iospace` a constant pointer to the struct by changing its definition to:

```
volatile struct PortRegs * const iospace = (struct PortRegs *)0xC0000000;
```

## 5. Code efficiency

The Arm compiler normally uses a base register and the immediate offset field to compile `struct` members or specific array element accesses. The immediate offset is available in the load or store instructions. In the Arm instruction set, `LDR` and `STR` word and byte instructions have a 4KB range, but `LDRH` and `STRH` instructions have a smaller immediate offset of 256 bytes.

Equivalent 16-bit Thumb instructions are much more restricted. `LDR` and `STR` have a range of 32 words, `LDRH` and `STRH` have a range of 32 halfwords, and `LDRB` and `STRB` have a range of 32 bytes. However, 32-bit Thumb instructions offer a significant range improvement. Because of the range restrictions, it is important to group related peripheral registers near to each other if possible. The compiler is generally good at minimizing the number of instructions required to access array elements or structure members. To perform these accesses, the compiler uses base registers.

You can choose between using one large C `struct` or `array` for the whole I/O space and smaller per-peripheral structs. For the two methods, there is little difference in terms of code efficiency. Using a large `struct` might help if:

- For word and byte accesses, your code works with a base pointer with a 4KB range.
- Entire I/O space is <4KB.

But arguably, it is more elegant to have one `struct` for each peripheral. Smaller, per-peripheral structs are more maintainable.