



# Arm Development Studio Tutorial

Revision: 0.0

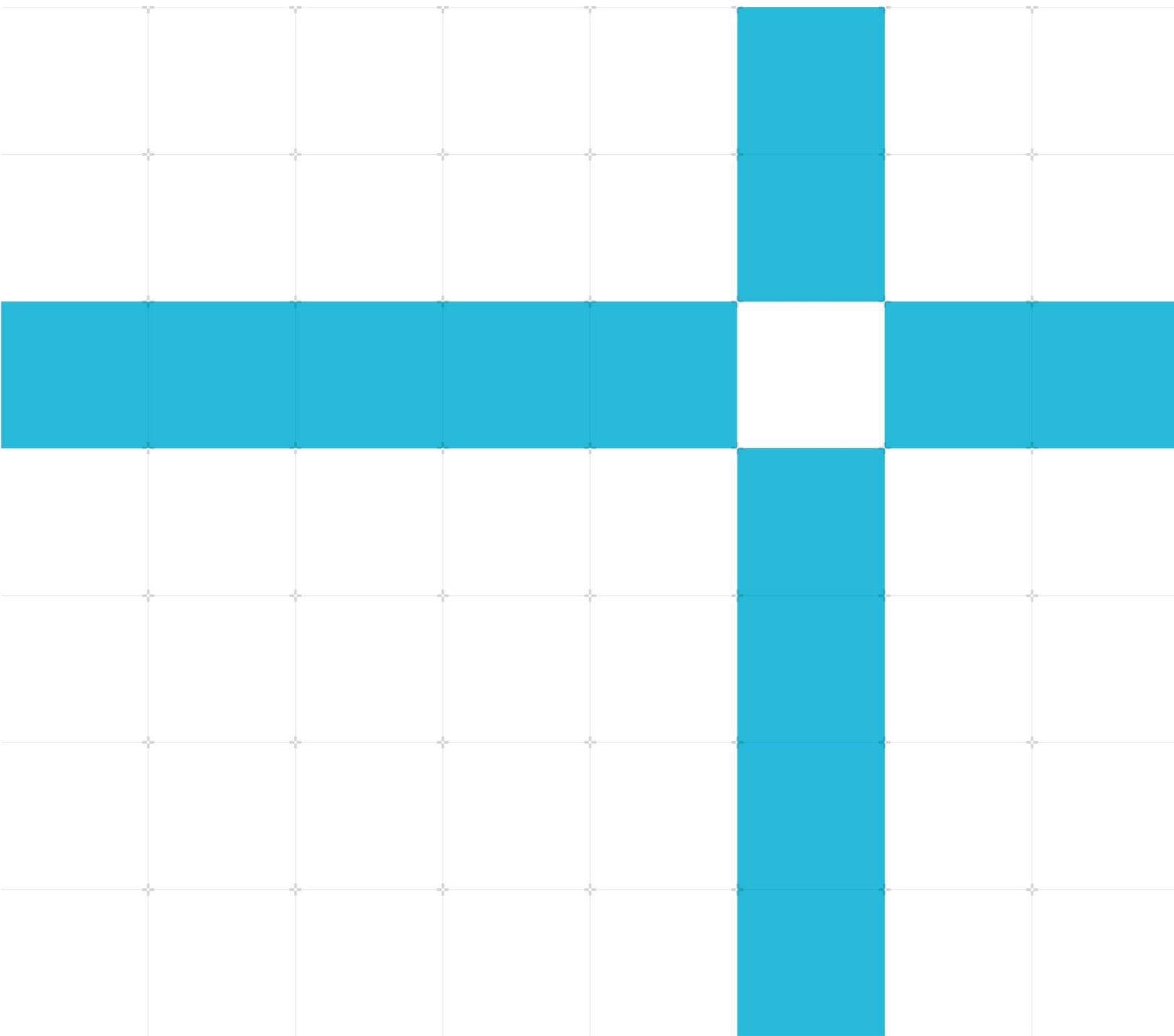
## Accessing memory-mapped peripherals with Arm DS

Non-Confidential

Copyright © 2021 Arm Limited (or its affiliates).  
All rights reserved.

Issue 0.0

NA



# Arm Development Studio Tutorial

## Accessing memory-mapped peripherals with Arm DS

Copyright © 2021 Arm Limited (or its affiliates). All rights reserved.

### Release information

### Document history

Issue	Date	Confidentiality	Change
0.0	8 <sup>th</sup> of February 2021	Non-Confidential	First version

## Confidential Proprietary Notice

This document is **CONFIDENTIAL** and any use by you is subject to the terms of the agreement between you and Arm or the terms of the agreement between you and the party authorized by Arm to disclose this document to you.

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: **(i)** for the purposes of determining whether implementations infringe any third party patents; **(ii)** for developing technology or products which avoid any of Arm's intellectual property; or **(iii)** as a reference for modifying existing patents or patent applications or creating any continuation, continuation in part, or extension of existing patents or patent applications; or **(iv)** for generating data for publication or disclosure to third parties, which compares the performance or functionality of the Arm technology described in this document with any other products created by you or a third party, without obtaining Arm's prior written consent.

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, third party 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.

If any of the provisions contained in these terms conflict with any of the provisions of any click through or signed written agreement covering this document with Arm, then the click through or signed written agreement prevails over and supersedes the conflicting provisions of these terms. 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 subsidiaries) 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 <http://www.arm.com/company/policies/trademarks>.

Copyright © 2021 Arm Limited (or its affiliates). All rights reserved.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

LES-PRE-20348

## Confidentiality Status

This document is Confidential. This document may only be used and distributed in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

## Product Status

The information in this document is release quality.

## Web Address

<http://www.arm.com>

# Contents

<b>1 Introduction .....</b>	<b>5</b>
1.1 Accessing memory-mapped peripherals .....	5
1.2 Basic concepts .....	5
<b>2 Arm recommendations .....</b>	<b>10</b>
<b>3 Alignment of registers .....</b>	<b>11</b>
<b>4 Mapping variables to specific addresses .....</b>	<b>13</b>
<b>5 Code efficiency .....</b>	<b>17</b>

# 1 Introduction

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.

## 1.1 Accessing memory-mapped peripherals

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.

This tutorial assumes you have installed and licensed [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 An Arm DS example project (see tutorial webpage) 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.

It is highly recommended to complete the [Hello World Arm DS Tutorial](#) before working with the examples in this tutorial.

**Note:** In this tutorial, the examples use a little-endian memory system.

## 1.2 Basic concepts

- 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.

**Note:** 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:

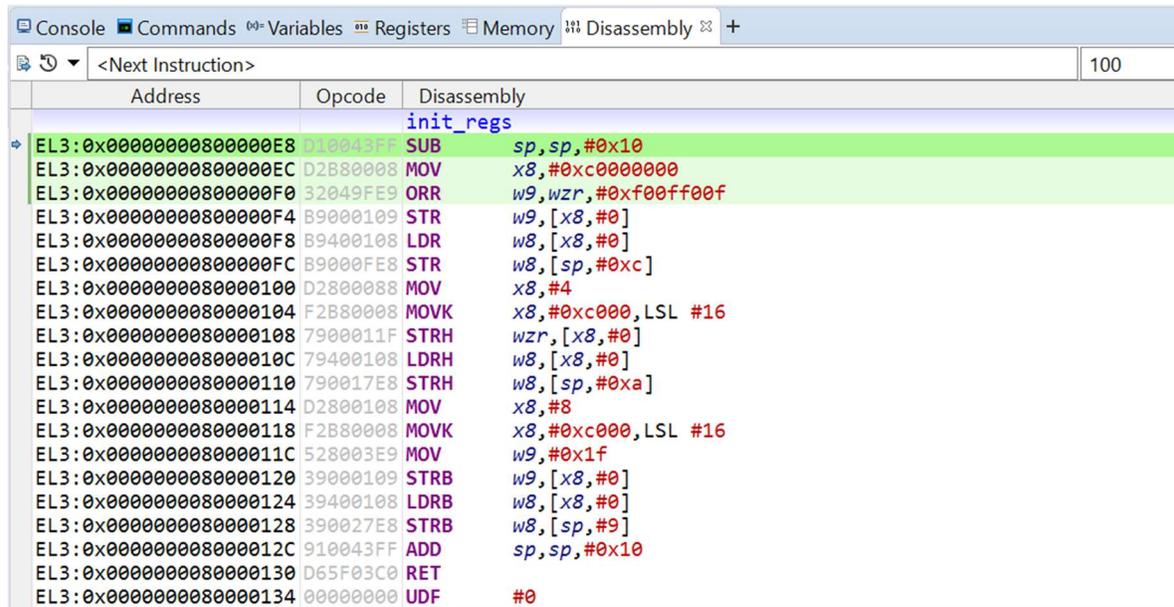
```
;;;5    void init_regs(void)
000000  e59f1024 LDR r1,|L1.44|
;;;6    {
;;;7        unsigned int int_val;
;;;8        unsigned short short_val;
;;;9        unsigned char char_val;
;;;10       *PortLoad = (unsigned int) 0xF00FF00F;
000004  e3a00101 MOV r0,#0xC0000000
000008  e5801000 STR r1,[r0,#0]
;;;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
    
```

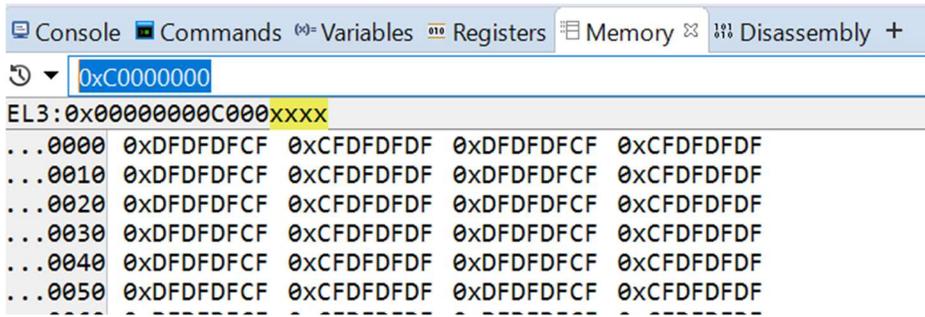
If you debug the previous code with the Cortex-A53 FVP in Arm DS, you can see the generated disassembly code in the **Disassembly** view.

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



Notice that for 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 also contain uninitialized values:



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`:

```

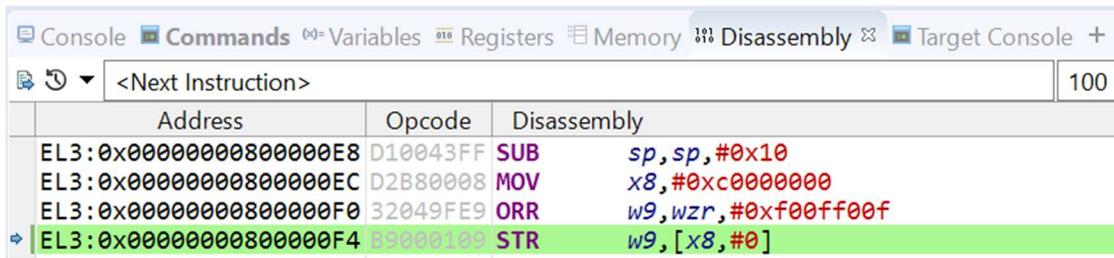
void init_regs(void)
{
    unsigned int int_val;
    unsigned short short_val;
    unsigned char char_val;

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

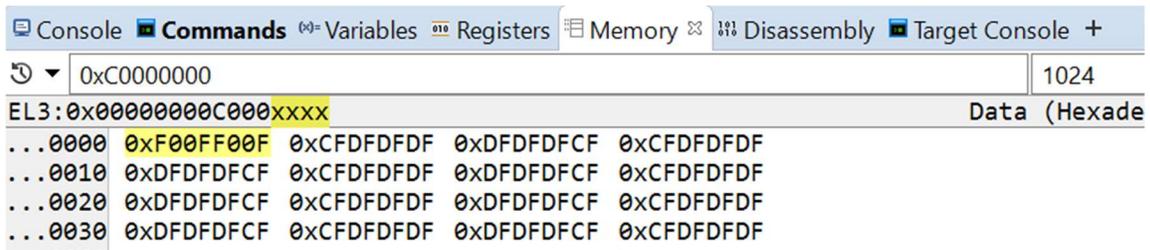
    *PortValue = (unsigned short) 0x0000;
    short_val = *PortValue;

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

    return;
}
    
```



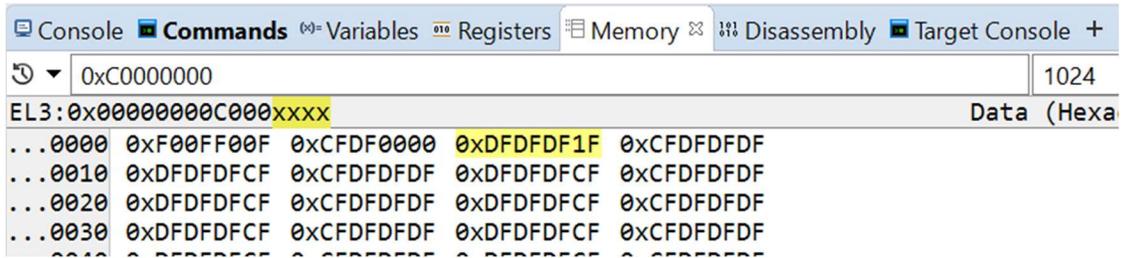
Opening the **Memory** view, see that the wanted value is set to the peripheral register:



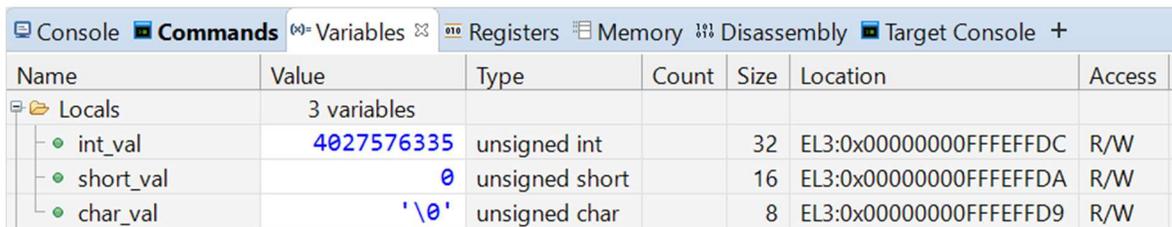
Continue stepping until the instruction `STRH wzr, [x8, #0]` is reached. This instruction is executed to store the value `0x0000` in the `PortValue` register, mapped to the address `0xC0000004`. In this case, the value set on the port register is half a word width (16-bits). When viewing memory at a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



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:



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. Remember that, 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.



## 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 u16_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 Alignment of registers

If you want to map 16-bit registers on 32-bit alignment as recommended, then you could use:

```
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.

**Note:** 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;
```

**Note:** The peripheral locations must not be accessed using:

- `__packed` structs where unaligned members are allowed and there is no internal padding
- OR

- 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:

- Forcing 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.

## 1. Forcing 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. This scatter-loading mechanism is the recommended method for placing all scatter-loading **AREAS** at required locations in the memory map.

To test using a struct at a specific address, do the following:

### A. 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;
```

### B. Open `scatter.txt`.

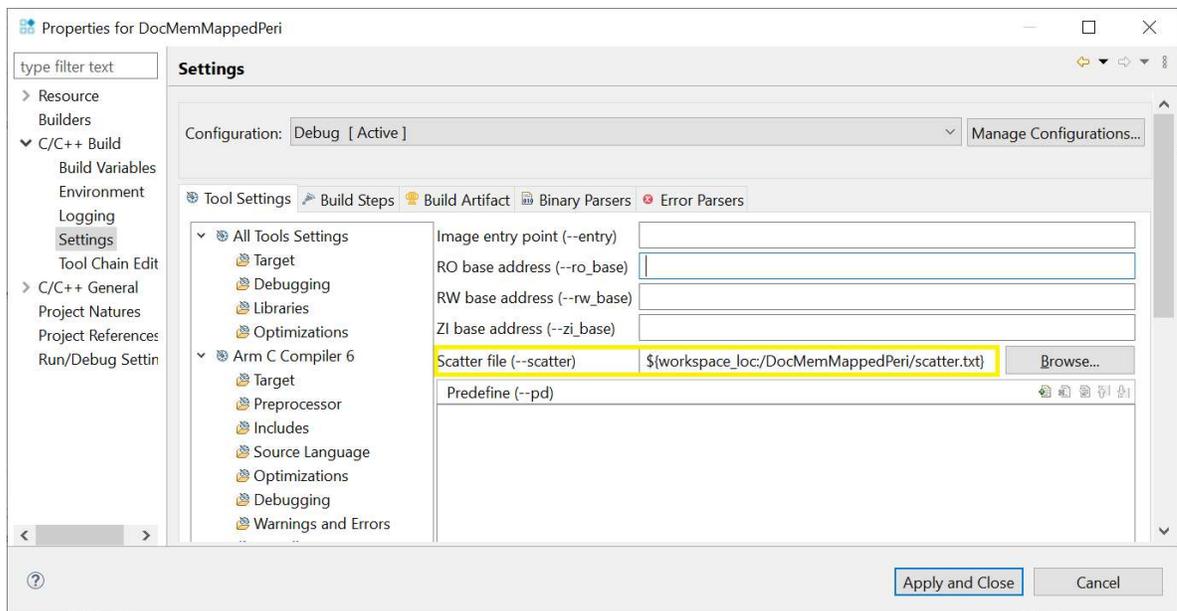
The `scatter.txt` file contains the following:

```
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**, that contains the struct, goes into the I/O area at `0x00000000`. All code (**RO**) and data areas (**RW** and **ZI**) from other object files go into the **ALL** region which starts at `0x80000000`.

You can add the scatter-loading description easily to an Arm DS project. In the **Project Explorer** view, right-click on your project folder and click **Properties**. Then go to **C/C++ Build > Settings > Tool Settings**. Then click **Image Layout** under **Arm Linker 6**. Click **Browse** to select the scatter file and then click **Apply and Close** to save the changes:



**Note:** 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 lie 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:

- ["The scatter-loading mechanism" section of the Arm Compiler Reference Guide](#)
- ["Scatter-loading images with a simple memory map" of the Arm Compiler Reference Guide](#)
- ["Scatter File Syntax" section of 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 [Pragmas documentation in Arm Developer](#).

C. 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 the following line in `main()`:

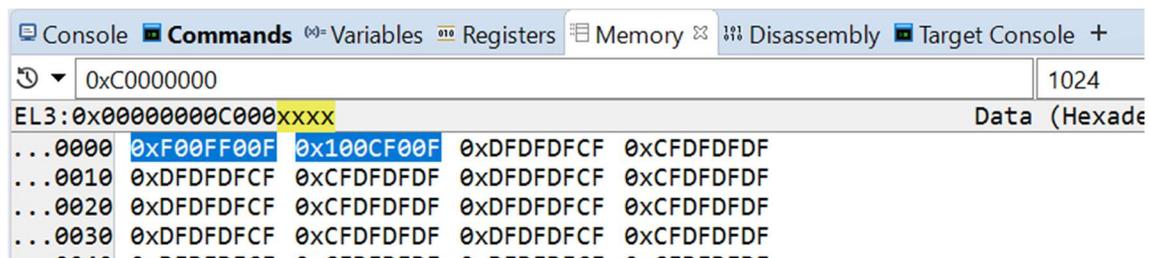
```
//init_regs();
```

D. Clean and build the example project.

E. Launch the Cortex-A53 FVP.

F. 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:



## 2. 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 plus 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 per peripheral. Smaller, per-peripheral `structs` are more maintainable.