



Accessing memory-mapped peripherals

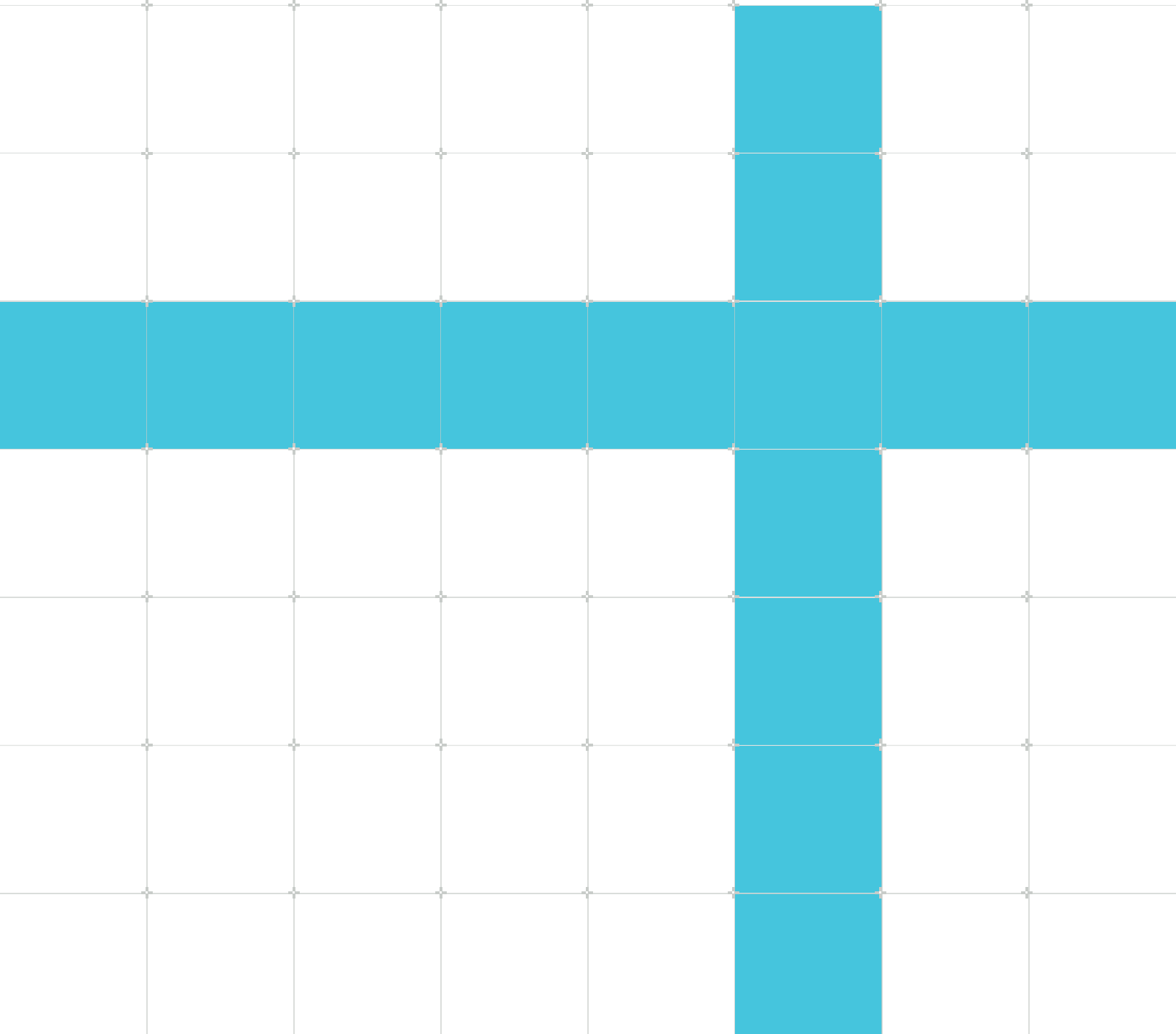
Version 1.0

Non-Confidential

Copyright © 2020 Arm Limited (or its affiliates).
All rights reserved.

Issue 01

102618_0100_01_en



Accessing memory-mapped peripherals

Copyright © 2020 Arm Limited (or its affiliates). All rights reserved.

Release information

Document history

Issue	Date	Confidentiality	Change
0100-01	1 January 2020	Non-Confidential	First 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 © 2020 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 Final, that is for a developed product.

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.

Contents

1. Introduction.....	6
2. Basic concepts.....	7
3. Arm recommendations.....	9
4. Alignment of registers.....	10
5. Mapping variables to specific addresses.....	12
6. Code efficiency.....	14

1. Introduction

In most Arm embedded systems, peripherals are located at specific addresses in memory. It is often convenient to map a C variable onto each register of a memory-mapped peripheral, and 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 DS-5 Development Studio. For more information, see [Getting Started with Arm DS-5 Development Studio](#).

2. Basic concepts

The basic concepts are:

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

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 `LDB` and `STRB` for 8-bit registers.

You should 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 half-word addresses. However, note that Arm recommends that all registers, whatever their size, be aligned on word boundaries.

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

```
#define PORTBASE 0x40000000 /* 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 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,#0x40000000
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    e12fff1e BX lr
```


3. 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 and there is no need to multiplex (or duplicate) the peripheral databus onto 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 and there is no need to multiplex (or duplicate) the peripheral databus onto low bits of the Arm databus.

Arm's AMBA APB bridge uses the above technique to simplify the bridge design. The result of this is that only word-aligned addresses should be used (whether byte, halfword or word transfer), and a read will read garbage 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 16-bit alignment, you might write:

```
volatile unsigned short u16_IORegs[20];
```

This is fine providing your peripheral controller has the logic to route the peripheral databus to the high part (D31..D16) of the Arm databus as well as the low part (D15..D0) depending upon which address you are accessing. You should check if this multiplexing logic exists or not in your design (the standard Arm APB bridge does not support this).

4. Alignment of registers

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

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

This only allows access to even numbered registers. You need to double the register number. For example, to access the fourth register you could use:

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

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

The registers are accessed as 32-bit full-width. But a simple peripheral controller such as Arm's AMBA APB bridge will read garbage into the top bits of the Arm register from the signals that are not connected to the peripheral (D31..D16 for a little-endian system). 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.

For example, to access register R4:

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

3. Use a struct.

This allows descriptive names to be used and can accommodate different register widths.



Padding should be 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;
```



Peripheral locations must not be accessed using `__packed` structs (where unaligned members are allowed and there is no internal padding), or using C bitfields. This is because it is not possible to control the number and type of memory access that is being performed by the compiler. The result is code which is non-portable, has undesirable side-effects, and will 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 alignment.

5. 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 methods generate efficient code, the choice of method is a matter of personal preference.

1. Forcing an array or struct variable to a specific address.

The array or struct variable should be declared in a file on its own. When it is compiled, the object code for this file will only contain data. This data can be placed at a specified address using the Arm scatter-loading mechanism. This is the recommended method for placing all AREAs at required locations in the memory map.

- a. Create a C source file, for example, `iovar.c` which contains a declaration of the array or struct variable:

```
volatile unsigned short u16_IORegs[20];
```

or

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

- b. Create a scatter-loading description file (called `scatter.txt`) containing the following:

```
ALL 0x8000
{
    ALL 0x8000
    {
        * (+RO,+RW,+ZI)
    }
}
IO 0x40000000
{
    IO 0x40000000
    {
        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 creates two different load regions in your image: `ALL` and `IO`. The zero-initialised area from `iovar.o` (containing your array) goes into the `IO` area located at `0x40000000`. All code (`RO`) and data areas (`RW` and `ZI`) from other object files go into the `ALL` region which starts at `0x8000`.

If you have more than one set of memory mapped registers, you would need to define each group of variables as a separate execution region (though they could all lie within a single

load region). To do this, each group of variables would need to 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 located in one file, making maintenance 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 only need to re-link the existing objects.

Alternatively, it is possible to use the `#pragma arm section` pragma to place the data into a specific section and then use scatter-loading to place that data at an explicit location.

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 *)0x40000000;
x = iospace->ctrlreg;
iospace->ctrlreg = newval;
```

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

```
volatile struct PortRegs * const iospace = (struct PortRegs *)0x40000000;
```

6. Code efficiency

The Arm compiler will normally use a base register plus the immediate offset field available in the load or store instruction to compile struct member or specific array element access.

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 improvement. Hence, it is important to group related peripheral registers near to each other if possible. The compiler will generally do a good job of minimising the number of instructions required to access the array elements or structure members by using base registers.

There is a choice between one big C struct or array for the whole I/O space and smaller per-peripheral structs. There is little difference in terms of efficiency. A big struct might be a benefit if you are using Arm code where a base pointer can have a 4KB range (for word and byte access) and the 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.