Placing C variables at specific addresses to access memory-mapped peripherals

**Applies to:** ARM Developer Suite (ADS), DS-5, RealView Development Suite (RVDS)

**Answer**

**Description**

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 read/write the register via a pointer. In your code, you will need to consider not only the size and address of the register, but also its alignment in memory.

**Basic Concepts**

The simplest way to implement memory-mapped variables is to use pointers to fixed addresses. If the memory is changeable by 'external factors' (for example, by some hardware), it must be labelled as `volatile`.

Consider a simple example:

```c
#define PORTBASE 0x40000000
unsigned int volatile * const port = (unsigned int *) PORTBASE;
```

The variable `port` is a constant pointer to a volatile unsigned integer, so we can access the memory-mapped register using:

```c
*port = value; /* write to port */
value = *port; /* read from port */
```

The use of `volatile` ensures that the compiler always carries out the memory accesses, rather than optimizing them out (for example, if the access is in a loop).

This approach can be used to access 8, 16 or 32 bit registers, but be sure to declare the variable with the appropriate type for its size, i.e., `unsigned int` for 32-bit registers, `unsigned short` for 16-bit, and `unsigned char` for 8-bit. The compiler will then generate the correct single load/store instructions, i.e., `LDR/STR, LDRH/STRH, LDB/STRB`.

You should also ensure that the memory-mapped registers lie on appropriate address boundaries, e.g. either all word-aligned, or aligned on their natural size boundaries, i.e., 16-bit registers must be aligned on half-word addresses (but note that ARM recommends that all registers, whatever their size, be aligned on word boundaries - see later).

You can also use `#define` to simplify your code, e.g.:

```c
#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;
}
```

using results in the following (interleaved) code:

```assembly
 000000 e59f1024 LDR r1,[l1.44]  
 000004 e3a00101 MOV r0,#0x40000000  
 000008 e5801000 STR r1,[r0,#0]  
 00000c e5801000 STR r1,[r0,#0]  
```
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:

```c
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).

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];**
   
   ... and only access even numbered registers - you will need to double the register number, for example, to access the fourth register you could use:
   
   ```c
   x = u16_IORegs[8];
   u16_IORegs[8] = newval;
   ```

2. **volatile unsigned int u32_IORegs[20];**
   
   ... where 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 to an **unsigned short** to get the compiler to discard the upper 16 bits.
   
   For example, access reg 4 using:
   
   ```c
   x = (unsigned short)u32_IORegs[4];
   u32_IORegs[4] = newval;
   ```

3. **use a struct**
   
   allows descriptive names to be used (more maintainable and legible)
   
   allows different register widths to be accomodated
   
   Note: padding should be made explicit rather than relying on automatic padding added by the compiler, for example:
   
   ```c
   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;
   ```

Please note that peripheral locations should *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, char** on their natural alignment.
Mapping variables to specific addresses

Memory mapped registers can be accessed from C in two ways: either by forcing a array or struct variable to a specific address, or by using a pointer to an array or struct (see below for details). Both generate efficient code - it is really down to a matter of personal preference.

1. Forcing struct/array to a specific address

The 'variable' should be declared it 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 (code, data, etc) at required locations in the memory map.

a. Create a C source file, for example, iovar.c which contains a declaration of the variable/array /struct, e.g.
   volatile unsigned short u16_IORregs[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:

   area
   IO 0x40000000
   
   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 group of variables (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. For further information, please see the ARM Compiler toolchains Compiler Reference documentation.

2. Using a pointer to struct/array

   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;

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

Code efficiency

The ARM compiler will normally use a 'base register' plus the immediate offset field available in the load/store instruction to compile struct member or specific array element access.
In the ARM instruction set, LDR/STR word/byte instructions have a 4KB range, but LDRH/STRH instructions have a smaller immediate offset of 256 bytes. Equivalent 16-bit Thumb instructions are much more restricted - LDR/STR have a range of 32 words, LDRH/STRH have a range of 32 halfwords and LDRB/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. Further information about the immediate offsets of LDR and STR instructions is available in the ARM Compiler toolchain - Assembler Reference documentation.

There is a choice between one big C struct/array for the whole I/O space and smaller per-peripheral structs. In fact there isn't much difference in efficiency - the big struct might be a benefit if you are using ARM code where a base pointer can have a 4Kbyte range (for word/byte access) and the entire I/O space is <4Kbyte - but arguably it is more elegant to have one struct per peripheral. Smaller per-peripheral structs are more maintainable.

Article last edited on: 2011-09-20 16:26:37

Rate this article
Current article rating: 7/10

Disagree? Move your mouse over the bar and click

Link to this article

Copyright © 2011 ARM Limited. All rights reserved.