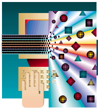


Representing a Microcontroller in C

Author:
Ata. R. Kahn, Director, Product Innovation, Philips Semiconductors

Synopsis:
The ARM core-based microcontrollers, like all 32-bit microcontrollers present a challenge in information assimilation and subsequent application. User Manuals for these microcontrollers run hundreds of pages long and are packed with information usually presented in a dense form. The C development suite for these machines is also hundreds of pages and generally quite terse. And the manuals generally have nothing to do with each other so it's up to the system designer to understand both manuals, and use them together and in a quick and efficient manner.

One way of breaking down the challenge into smaller parts is to focus on different aspects, four different areas to focus on are:
1. C Code constructs and forms for embedded microcontrollers: What elements of the ANSI C language are useful for embedded applications and how they might be used.
2. C Code Efficiency considerations, especially for ARM: The interaction of the architecture with the language, how the ARM architecture and the ARM compiler do common things and how to take advantage of that.
3. Memory maps and representations: controlling how code and variables reside in absolute locations via the linker, the interface between re-locatable code and absolute mapping to physical locations.
4. Microcontroller example: An example of a Philips ARM Microcontroller and a very (very) simple representation of that memory map to the linker.

This article details how to attack the challenges of complex 32-bit microcontroller designs.

C Code Constructs and Forms for Embedded Microcontrollers:

Variables in the embedded world reside in a different environment from their counterparts in PCs. They may be stored in read-only storage and/or at fixed addresses, and their values may change due to hardware events which are asynchronous to program flow. Writing to them may be illegal and cause system exceptions and they may change their values when they are read.

Constructs exist in C which signal these characteristics to the compiler so that it may make the correct assumptions about accessing these variables and generating the appropriate code sequence. This is done by using special keywords, which are used to qualify the type of the variable (storage classes are defined broadly in an embedded context much as they are in a general computing context). Some of these constructs are described as follows:

The Const Type Qualifier (added to the C language by the ANSI C Committee):

A Type qualifier restricts the way an identifier of a certain Type is used. A Type qualifier keyword comes syntactically between the Storage Class and identifier, e.g. the declaration `static const int id` declares id as an identifier of type integer, with storage class Static (initialized once and retains its value upon block reentry and is restricted in scope to the remainder of the source file) and qualified as of type const.

Identifiers thus qualified can be initialized but can never be changed after that. Thus, in the expression `static const int id = 4`, id will always remain 4 and attempts to change its value will result in compilation errors. Variables declared with this qualifier may reside in Read-only memory and the compiler will attempt to detect ways in which (incorrect) assignments to them might be generated. In the example above, if we declare a pointer to the variable id `int *ptr = &id`, the compiler generates an error message complaining about an implicit cast, *ptr = x that would be illegal, although incrementing the pointer itself or assigning it to something else is fine. To access the variable id through a pointer, a qualified pointer should be used:

```
const int *ptr = &id
```

ptr is a pointer to a variable of type integer, which is qualified as const. The value of ptr may be changed but *ptr accesses will cause a compiler error. This would typically be used for accessing variables stored in read-only memory via registers. Accesses of the converse variety are possible. For example, a fixed location in memory (e.g. a DMA base register) may point to anywhere in RAM. If you want the pointer to be constant, but not what it points to, then the declaration would be:

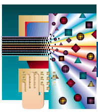
```
int id;
int * const ptr = &id; /* typical case for
microcontroller HW */
```

This initializes ptr, a constant pointer to an integer, to the address of variable id, but the value of ptr itself may not be changed.

There are cases where neither the Pointer or what it points to can be changed, for example, a fixed location in read-only memory where a serial number may be stored. In this case the appropriate declaration is:

```
const int id = 4;
const int *const ptr = &id;
```

ptr is a constant pointer to an integer variable, id, qualified as constant. These values are set once and never changed. Constant pointers are very useful for accessing variables (and constants) that are in specific locations in a memory map. These locations may be addresses



of peripheral registers, flash memory locations, or general read/write memory regions. A constant pointer can be initialized to a fixed memory address as:

```
int *const ptr = (int *)0x40000000;
```

This initializes ptr to be a pointer that is qualified as constant and pointing to an integer. Note that the cast to integer of the hex address value is required to maintain type consistency regardless of the underlying architecture. For the ARM processor, casting between integers and pointers results in no change of representation (32-bits).

The variable type should be recognized from the microcontroller's architecture and the appropriate const qualifiers used. It is harmful to have a write occur to Flash memory, for example; this could cause a data-abort exception and crash the program.

The Volatile Type Qualifier (added to the C language by the ANSI C Committee):
Usage of the *Volatile* qualifier indicates that the variable may be changed independent of the program (typically by hardware events in a microcontroller peripheral). The `volatile int id` statement declares id as an integer whose value can change asynchronously and external to the program's execution flow. The significance of qualifying it as volatile is that the compiler will not optimize references to it relying on its value being unchanged from the last assignment (which it may attempt to do with other variables). I/O peripheral registers must typically be declared volatile when their contents are hardware event dependent.

Memory-mapped Peripheral Registers have fixed locations and Pointers to them are declared as `volatile unsigned int *const ptr =` . Since the pointer ptr is a constant, it is initialized to a fixed address such as `volatile unsigned int *const ptr = (unsigned int *) 0xE0000000` . This means that ptr is a const pointer to a volatile unsigned int and ptr is permanently initialized to 0xE00000000, where the volatile unsigned integer resides. Note that the integer itself may be qualified as both volatile and constant, indicating a register whose value changes due to hardware events and which should not be assigned. The result register of an A/D converter is an example of such an assignment.

Representing a Peripheral Using Structs
The following example uses the General Purpose I/O Registers for one of the 32-bit I/O ports of a Philips ARM core-based microcontroller which work as follows (from the Users Manual):

Reg.Name	Memory-location	Bits	Function
IOPIN	0xE002 8000	31:0	GPIO Pin Value
IOSET	0xE002 8004	31:0	Set Bits by writing 1s
IODIR	0xE002 8008	31:0	Direction Control bits (0 = Input, 1 = Output)
IOCLR	0xE002 800C	31:0	Clear Bits by writing 1s

Note that separate bit-wise Set and Clear functions are required because I/O pins and peripheral register values can change due to hardware events and, therefore, only those bits must be written to that are needed to be changed. This can only be done by having bit-lane write capability. The I/O port allows a 32-bit value to be written if required.

A struct containing the registers above is declared:

```
typedef struct
  { volatile unsigned int IOPIN;
        volatile unsigned int IOSET;
    volatile unsigned int IODIR;
    volatile unsigned int IOCLR;
  } GPIO;
```

Now GPIO is a structure type and can be used to declare ports. For instance, the microcontroller has 4 ports declared as `GPIO Port0, Port1, Port2, Port3` . Pointers to Port0 can be declared as

```
GPIO *const ptr_Port0 = (GPIO *) 0xE0028000;
/* address of Port 0 */
```

This declares `ptr_Port0` as a constant pointer to a variable of type GPIO, which was declared as a Struct containing 4 variables representing the I/O port, the constant pointer is initialized to the base address of the Struct which is the memory-mapped address of the I/O port (note that the cast to pointer to type GPIO is necessary for consistency).

We now have a variable declared as Port0 and a Const pointer to it declared as ptr_Port0 and initialized to point to it.. We can use the Pointer to access the GPIO port0 as `ptr_Port0 -> IOSET = 0xD0D0FEED` . The other variables may be similarly Read and Written to as above. The entire Struct GPIO may also be declared Volatile by writing `Typedef Struct { ... } Volatile GPIO` .

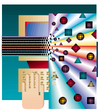
C Coding Considerations For ARM Processors:

There are many basic considerations when coding for an ARM processor. The ARM compiler documentation has great detail on various major and subtle aspects to consider and should be read carefully. Most of the topics discussed in what follows show aspects of working with the ARM architecture and the ARM compiler, more information is provided in the ARM compiler documentation. Note that other compilers, such as GNU, may, and probably do, work differently.

Coding considerations are very important when optimizing performance and/or dealing with limited amounts of embedded memory. The cost of overflowing a given memory configuration are usually high (next higher chip or, if one does not exist, then re-designing the entire system may be required). It is important to look at the assembly language emitted by the compiler to gain insight into efficiency and what the compiler is doing.

Debugging Considerations
Another important consideration in embedded applications is debugging efficiency, which sometimes clashes with code efficiency. The compiler allows different debug options which trade-off optimization and code size versus debug information (about a 15% impact is possible); it is the user's responsibility to understand their effects and use the appropriate ones.

Note that code running in debug mode may not run in other modes if there are side effects; it is best to stay as close to debug mode as possible so that the code running in the system will be as close as possible to the code that was verified.



Optimization may result in code that is different from the code used during debugging, and may have unforeseen side effects. The debug mode offers resources (such as Semi-Hosting) that will not be available in the target system; the user must remove or replace such calls with their own code. Also, ARM conditional execution is disabled for all debugging options.

Looping Considerations

Many embedded programs follow the 90/10 rule: A program spends 90% of its time in 10% of the Code (Hennessy & Patterson). The 90% is especially true in the embedded world where programs spend most of their time executing loops (normally with backwards branches). Therefore, looping behavior is important to understand and optimise.

A simple optimization is to write 'for' loops that count down to terminate at zero, rather than counting up. For instance, `for(i = 1; i < n; i++)` , causes the compiler to generate a compare instruction comparing i to n and then a branch-on-less-than instruction terminates the loop. By contrast, the loop written as `for(i=n; i != 0; i--)` allows the compiler to set the zero flag while doing the decrement (via the SUBS instruction) and a subsequent BNE terminates the loop that saves code space and time and frees up a register because the variable n is now an initialization value for the loop pointer instead of having to be maintained in a register and compared to the variable i in every loop iteration.

Register Allocation & Pointer Aliasing

Register allocation is another very important optimization. Variables are kept in ARM registers instead of being accessed from memory with great savings in performance and code size. To ensure variable value consistency though, the compiler must make sure the variable value in memory is not changed by a pointer reference elsewhere in the program (pointer aliasing), therefore a register allocated variable must a) be a local variable or a function parameter; and b) not have its address taken (&var_name) or assigned to another variable.

If a pointer is created pointing to the variable, the compiler is unsure whether the value is not changed from outside the block scope and therefore must locate the variable in memory and not in a register. For this reason, global variables can never be in registers because their values are not private to a given block scope. An example of pointer aliasing follows:

```
int prog1(int var1)
{ somefunction(&var1);
```

The passing of the address of var1 causes subsequent references to var1 to be from memory because the compiler cannot guarantee that a register value will be unchanged. However, by using a local intermediate variable:

```
int prog2(int var2)
{   int local1 = var2;
  somefunction(&local1);
  var2 = local1;
```

Now all subsequent references to var2 can be via a register because its address was not taken and the compiler can locate it in a register and access it with moves instead of using the Stack and Load and Store instructions.

ARM Data Types and Natural Alignment

Between the architecture of a Microcontroller and the variable types in C there can exist some natural alignment e.g. the ability to handle byte types or word types "naturally" (i.e. in one instruction and efficiently).

ARM Code can access data most efficiently if it corresponds to its natural size, the alignment boundaries for C variable types are:

Type	Implementation (Alignment)
char	unsigned byte (1)
short	signed 16-bit halfword (2)
int	signed 32-bit word (4)
long	signed 32-bit (4)
long long	signed 64-bit double word (8)

Since ARM is a 32-bit architecture, it is expected that it would be most efficient in handling its "natural" size type. This proves to be the case.

The architecture is most efficient when using ints because the ARMv4 and subsequent ARMv4T architectures can LOAD signed and unsigned 8- and 16-bit values but the ARM registers are all 32-bits long and the ALU is 32-bits wide. The implications for non-ints such as Char and Shorts are that with Char and Short variables: a) unsigned values are zero-extended and b) signed values are sign extended. Thus, all char & short variables must be cast after every operation to check for rollover boundaries. It is much more efficient to use int for local variables and cast on return value if required. Here, the use of ints actually saves code space and increases performance.

Here's an example of efficiency using the integer type:

```
short somefunction(short i)
{ i = i + 1;
return i; }
```

The code generated will be:

```
ADD a1,a1,#1
MOV a1,a1,LSL #16  /* put sign bit in MSB */
MOV a1,a1,ASR #16 /* sign extend */
MOV pc, lr  /* return */
```

For an int, as in:

```
int somefunction(int i) {i = i +1; return i;}
```

The code generated is:

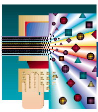
```
ADD a1,a1,#1
MOV pc,lr /* return */
```

Moral: Use ints when possible to avoid cast operations for shorts & chars. Note that there are exceptions to every rule and there may be a trade-off between code size and variable space size.

ARM Thumb® Procedure Call Standard (ATPCS)

The ARM C Compiler uses a standard parameter passing and register use convention called the ATPCS which uses registers as in Figure 1, next page.

Understanding and taking advantage of this convention can lead to more efficient programs. In particular, the registers used by the compiler include: a) Registers r0 to r3 (synonyms a1 to a4) are used to pass parameter values into routines and to return values. The called routine does not have to restore them upon return; b) Registers r4 to r11 can be used to hold Local Variables (Register Allocation); synonyms are v1 to v8. A called routine must restore the state of these registers before returning (save on



Register	Synonym	Special use	Role in ATPCS
r15	-	PC	Program Counter
r14	-	lr	Link register
r13	-	sp	Stack pointer
r12	-	lp	Intra-procedure call register
r11	v8	-	ARM state variable register 8
r10	v7	sl	Stack limit pointer
r9	v6	sb	Static base for position indep.
r8	v5	-	ARM state variable register 5
r7	v4	-	Variable register 4
r6	v3	-	Variable register 3
r5	v2	-	Variable register 2
r4	v1	-	Variable register 1
r3	a4	-	Argument/result/scratch register 4
r2	a3	-	Argument/result/scratch register 3
r1	a2	-	Argument/result/scratch register 2
r0	a1	-	Argument/result/scratch register 1

Figure 1: Procedure Call Standard Register Use.

stack & restore); c) Up to 14 local register variables (r0 through r11 plus r12 and r14) may be used in functions; d) more local variables will cause spilling to memory, spilling to memory is expensive replacing very efficient register accesses with memory accesses and should be avoided by code organization whenever possible.

Implications of ATPCS For Users

Some simple rules when writing functions makes for code that is smaller as well as faster. Note that for optimal power/energy considerations, replacing external memory accesses with internal register accesses is also desirable. Simple guidelines for functions include: a) Up to 4 words can be passed as arguments to functions in registers (in r0..r3). More than 4 arguments will be passed on the stack causing memory access operations to be generated; b) Functions with 4 or fewer arguments save code space and time, keep functions simple!; c) If there are more than 4 arguments, group them into a structure and pass the function a pointer to the structure. Pass pointers to structures in general (very inefficient to pass a structure); d) Keep local variables to less than 10 (for 4 arguments); e) Avoid functions with variable number of parameters (variadic routines) for the same reasons; f) Return values also use ro-r3. More values are returned on the stack.

Structure Alignment and Memory Use

The order that variables are declared in a structure affects the number of bytes of storage it requires and the access means (the kind of instructions generated).

The compiler aligns the start address of the first member of a structure to the largest access length (4 or 8 bytes) and then aligns each following member to the maximum alignment required by its length (e.g. a field with a char type is aligned to the next available byte while an int type is aligned to the next four byte (word) boundary). Padding is inserted between alignment gaps.

Assume a structure that has 2 chars, a short and an int, as members. Then, the structure will be arranged in memory as follows for different declarations:

```
struct {char c;short s; char d; int i;}
bytes: c x s s d x x x i i i i. A total of 12 bytes.
```

Note: x represents bytes added as padding.

If the same struct is declared as:

```
struct {char c; char d; short s; int i}
bytes: c d s s i i i i.
```

Then no padding is generated because the fields are in accordance with the compiler's alignment method. Paying attention to structure arrangement can save bytes.

The _packed Keyword and Bit-Fields

Structures defined with the _packed qualifier have their members set to align to 1 byte (no padding). As an example `Typedef _packed struct { char c; int i; short s; char d;}`, forces the structure to be packed as 'bytes: c i i i i s s d, starting at a 4- or 8-byte boundary. While this saves space, accesses to the variables are still done by unaligned loads and stores and the compiler has to generate shift and merge instructions to form, for instance, the int i into one register. This is very inefficient because it creates more instructions and reduces performance. The use of this feature really needs to be justified.

Bit-fields are a similar example of packing density obtained at the expense of more code and lower performance. A bit-field can be used in structs to create fields that are user-defined in length. Bit-fields can be contiguous as long as they do not overflow the basic container size (e.g. an integer container size is 32 bits). As an example, the declaration `struct bitter { int a:10; int b:20; int c: 30;}` causes the following bit layout in memory:

bits of Word 1:
aaaaa.aaaaa.bbbbb.bbbbb.bbbbb.bbbbb.xx
bits of Word 2:
ccccc.ccccc.ccccc.ccccc.ccccc.ccccc.xx

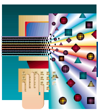
The bits marked x are padding. Bit-fields are allocated in memory because they are addressed via pointers, thus making them inefficient to access and because of their unaligned nature. Logical masking of integer variables is more efficient.

The basic rule is that integer values align naturally with the ARM 32-bit architecture and thought should go into using non-aligned and shorter variables to ensure that any gains are not outweighed by processing and accessing inefficiencies.

The Memory Map

The memory map represents the starting address and the length of various regions in memory corresponding to a) Read-only memory (ROM, Flash) areas: This could include Boot Loader code that could copy code to RAM and turn over execution; b) RAM areas: these areas could be used for execution, heap, and stack purposes; c) peripheral areas: these are memory-mapped peripheral registers that are changed by hardware events and may have special access requirements; d) Interrupt Vector Tables: these are fixed locations in memory that interrupt and exception handlers branch to. These areas are often remapped after booting and for Flash re-programming.

Figure 2. Some points to note about the memory map of the Philips LPC2294 Microcontroller: a) There is a Boot Block located at 8K below 2GB that actually handles system reset events and includes a Real-Time Debug Monitor program. The Boot Block is located at the top of the internal memory area so that it does not change from derivative to derivative; b) Interrupt vectors are remapped after booting to be invoked from the user code



once it is verified that user code exists and is valid (checksum existence). Otherwise the boot loader attempts to download a program into Flash; c) The memory layout is communicated to the linker via a special file called a scatter load file. This allows arbitrarily complex regions to be laid out in memory and to be named via special compiler and assembler directives.

The memory map layout is preserved between derivatives allowing for software re-use. The interrupt service routine locations and interrupt conventions are also preserved as much as possible.

Scatter Loading

Scatter loading literally means "scattering" the executable image into the appropriate regions of memory. It is required to allow an arbitrarily complex memory map to be used, to control the loading order by address of variables and modules, and to relate variable properties to specific memory regions. Memory regions have specific properties that include: a) Read-Only (RO): Code and data that is only to be read, never modified; b) Read-Write (RW); Code and data that can be read and written back; c) Zero-Initialized (ZI): Zero initialized data.

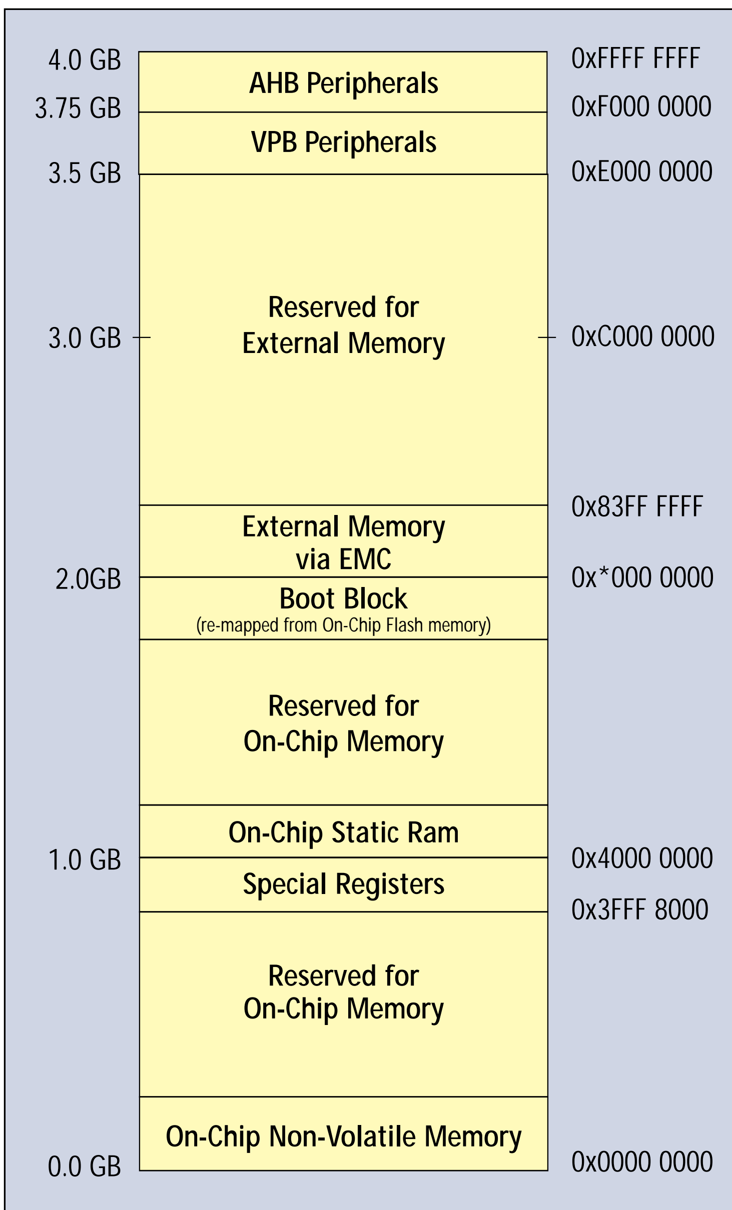


Figure 2: System Memory Map

Many regions with different attributes can exist. Code and data in RAM must be copied from ROM at execution time. Data may need to be initialized to zero for statics etc. Stack and heap addresses must be set and Vectors remapped before execution.

To do all the above, a Scatter load file is created and passed to the Linker:
• The File contains names of Regions, Attributes, Starting Addresses and, optionally, Lengths.

• Within each Region, Areas or Sections of Programs may be ordered so that they come First (or Last).

• Sections can be located at Fixed Addresses so that certain types of information can be found at a given address in all Microcontroller Types (e.g. Manufacturing information).

Given the memory map for the Philips LPC2294 product, a very (very) simple scatter load file would appear as follows:

```
FLASH_IMAGE 0X00000000
{ FLASH_Code_Data 0X00000000 0x10000
  {       vectors.o (Vect, +FIRST) ; Interrupt
& Exception Vectors
    * (+RO) ; All Read-Only Code and Data
  }
  RAM_Code_Data 0X40000000 0x4000
  { * (+RW, +ZI) ; All Read-Write Code and Data
  }
}
```

This file asks for the code image to start at all zero's, specifies that the interrupt and exception vectors be loaded first and that all Read-only Code and Data follow. Read-write (RAM) data is put at 0X40000000 and it is specified that all read-write code and data (zero-initialized) be loaded there.

Other Considerations and Conclusion

Some other general considerations to remember are:
• Avoid division whenever possible; use shifts instead if your ARM core does not have hardware divides.

• ARM instructions are uninterruptible: 'Load and Store Multiple' can greatly increase interrupt latency for slow memory systems; use a compiler switch to limit the number of registers stored or loaded in one instruction.

• Run-Time libraries can be very large; replace commonly used functions with your own functions to avoid linking Run-Time library code.

There are several other considerations to take into account listed in the compiler documentation. Thoroughly study the manual and try examples. Getting the program to compile and studying the assembly language should suggest other areas to consider as well. Finally, a great quote attributed to B. Kernighan (via J. Ganssle's newsletter): "Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it."