EEE3096S

Serious I/O Programming (II)

Embedded Systems II

L18

Dr Simon Winberg

Electrical Engineering
University of Cape Town

Outline of Lecture

- Review of class activity
- Joys of inline assembly
- Jump tables
- Function pointers
- Bitwise operators
- Now you are fully ready for
  - Interrupts in C!!

Outline

1. Peripheral Access
2. Inline Assembly
3. Jump tables
4. Bitwise Operations
5. Shadow Registers (recap)
6. Speed and Code Density
7. Polling and Interrupts
8. Measuring execution time & Watchdog timers

Part 1 — This lecture

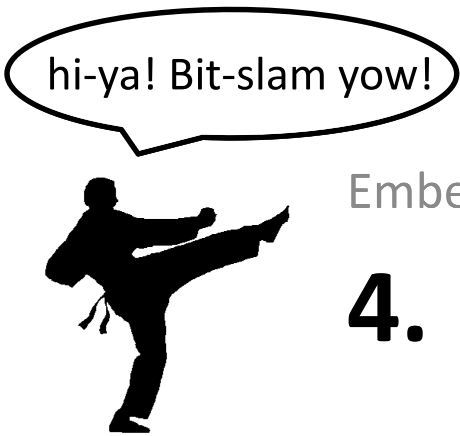
Part 2

You wanted to see the class activity solution…

But first…

You need to be reminded about the FORCE of

BITWISE OPERATORS in C



Embedded Systems Software Techniques

4. BITWISE OPERATORS

Embedded Systems II

Bitwise Operations

- You need to know the C operators **&, |, ~, ^**
- Why are they different from **&&** and **||**?
- These operators are bitwise operations
- **&** Bitwise *and* (AND)
- **|** Bitwise *or (OR)*
- **~** Bitwise unary complement (NOT)
- **^** Bitwise *exclusive or* (XOR)

Bitwise Operations

- Often you need to access a single bit of a value, to test it or set it
- The easiest way to do this is with a **bitmask**

```
#define PB0     (0x1 << 28)  // 29th bit
```

- Use the bitmask with the **bitwise operators**

```
if (AT91_SYS->PIOB_PDSR & PB0)
   pb0_on();
   else
   pb0_off();
```

Bitwise Operations

- Some peripherals have separate bit set and bit clear registers (such as the AT91)
- Some peripherals you need to read a value, change it and write it back.
  The &=, |=, and ^= operators
  are useful for doing this
- Macros can have a **very good** effect on the readability of code which uses bitwise operations

Class Activity & take-home [group] exercise

Reflections on

Consider you are developing a simple digital recording device based on a 32-bit microcontroller.

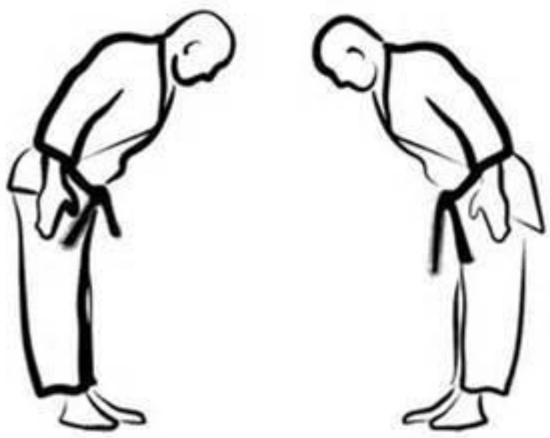
<u>Peripherals need:</u>

- 10-bit ADC (for recording voice)
- USB (for downloading the recorded data)
- 4 x LEDs (“power”, “record”, “full”, “comms”)
- 2 pushbuttons (“record/stop”, “pause/continue”)

<u>Memory:</u>

- 32Kb internal program flash fixed at address 0x0 – 0x7FFF
- 32Kb internal SRAM for data memory fixed at address 0x8000 – 0xFFFF
- 2 Megabytes external RAM on memory bus

Solutions Handout…



onegaishimasu

Start of Review

CLASS ACTIVITY:
SOLUTIONS
Assessing understanding of peripheral registers
For EEE3096S Lecture 17

Please refer to the class activity handout. This document provides suggested solutions.

QUESTION 1 SOLUTIONS:

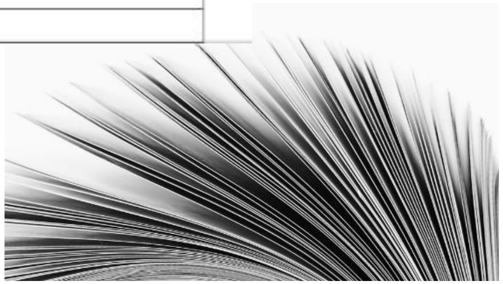
1. Decide the **memory addresses** (i.e. plan a memory map) to place your ADC, USB controller, and block of PIO (or GPIO)

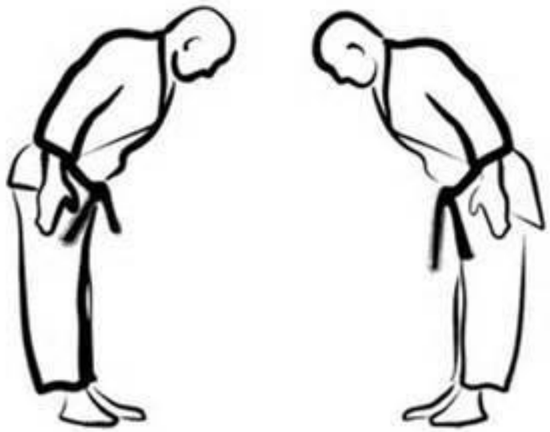
So you should have realized that the second page provides various peripheral register information.

Memory Map

Start	End		Comment
0x00000000	0x00007FFF	32kb Flash memory	This is clearly non-negotiable (given)
0x00008000	0x0000FFFF	32Kb internal SRAM	ditto
0x00010000	0x0FFFFFF	Unused	
0x1000000	0x11FFFFF	2 Megabytes external RAM	$1024^2 = (400_{16})^2 = 100000_{16} * 2 = 200000_{16}$ Starting at 0x1000000 because easier for CS
0x1200000	0xFFFEFFFF	Unused	This may be available for future memory
0xFFFF0000	0xFFFF0000	ADC_Control	This is the starting point of the peripherals
0xFFFF0001	0xFFFF0002	ADC_Samp	
0xFFFF0003	0xFFFF000F	Gap	Unused area
0xFFFF0010	0xFFFF0010	USB_Control	
0xFFFF0011	0xFFFF0011	USB_DataIn	
0xFFFF0012	0xFFFF0012	USB_DataOut	
0xFFFF0013	0xFFFF001F	Gap	

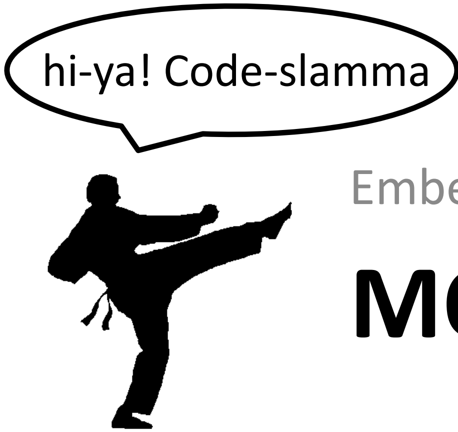
Solutions Handout…





arigatou gozaimashita

End of Review



Embedded Systems Software Techniques

MORE POWER TRICKS

Before we go on to interrupts and then more specifics of ADCs and sampling these are some coding techniques we should cover first.

Embedded Systems Software Techniques

INLINE ASSEMBLY

Inline Assembly

- Assembler code can be included in C code
- Useful for accessing hardware or CPU features that aren't exposed by the C language.
- Inline Assembly is not standardized in ANSI C
- GCC uses the *asm* keyword

  asm("instruction": inputs: outputs);
- Other compilers use other keywords and syntax
- Code with inline asm is not ANSI C!

Inline Assembly Best Practice

- Wrap inline assembly in #ifdef, #endif pair to hide it from other compilers
- Wrap inline assembly in C functions and keep it in a separate file – so the body of your code is still ANSI compliant
- *Some purists say:*
  Don't use inline assembly at all but rather keep assembly code in separate modules
- Inline assembly can prevent the compiler's optimizer from working – making your code run slower.

GCC Inline Assembly

Example – rotate a value right one bit

int rotRight(int val) {

int result;

asm(“mov %0, %1, ror #1”

: “=r” (result)

: “r” (val));

return result;

}

*Don’t worry much about what this code is doing now (we will get into assembly next term) but just know the **asm** keyword in C usually indicates inclusion of assembly that is directly output to the assembler*

Great document can be found here:

http://www.ethernut.de/en/documents/arm-inline-asm.html

Embedded Systems Software Techniques

3. JUMP TABLES

Jump Tables

- Jump tables can be implemented in many different ways in order to make code more optimal
- The “switch” statement in C is often converted into a form of jump table by the compiler
- See Example →

… how is this converted to a jump table …?

Example use of switch

```
#include <stdio.h>
int main (int argc, char* args[] ) {
 printf("Enter lines of text...\n");
 while (1) {
  char ch;
  ch = getchar();
  switch (ch) {
   case 'x' : return 0; // exit program
   case 'a' :
   case 'b' : printf("Handled a or b\n");
              break;
   case '\n': // ignore carriage return
              break;
   case 'c' : printf("Handled c\n"); break;
   default  : printf(
               "Cannot process %c\n",ch);
  };
 };
 return 1; // should never get here
}
```

Switch / jump table Implementation

- For limited sequences, e.g., 8-but character , you could cover the whole set of possibilities
- Example (see code on previous slide)

```
void* jtable[256] ;
   // compiler fills with relevant addresses
ch = getchar();
pc = jtable[ch];
// relevant address loaded into pc
```

- Memory cost (32-bit):

  256 * 4 bytes = 958 bytes

  Instructions: 1 word (4 bytes) ?!

jtable

Index	Address
0	Default
1	Default
…	
‘a’	handle_aorb
‘b’	handle_aorb
‘c’	handle_c
…	

Implementation using Lookup table

- Alternatively, a table of two fields could be used:
  - Value : value to be matched
  - Address : address to jump to
- Example (see code previously)

jtable

Index	Value	Address
0	'x'	handle_x
1	'a'	handle_aorb
2	'b'	handle_aorb
3	'c'	handle_c
4	'\0'	Default

Needs a for loop to go through the various options. Possibly divide into sub-tables, e.g.

if (ch>='a' && ch<='c')
**   *pc* = table1[ch]**
else
**   *use_table2***

Each row: value = 1 byte; address = 4 bytes; Memory cost = 5 x 5 = 25 bytes

Obviously performance penalties, such as a loop, sequence of comparisons.

Implementation using hash table

- Could use a hybrid technique, e.g.
  - char ch = getch();
  - int j_index = hash(ch)
  - pc = jtable [j_index] // jump to relevant address
- This technique would probably be difficult to be instrumented as part of an automatic process provided by a compiler
- As a programmer, knowing the possible options, you may be able to think of a suitable hash function (e.g. using #define preprocessor macros which will translate into constants at compile time)

Function Pointers & Implementing a jump table manually

- In C you can use the **typedef** keyword to link any type declaration to a symbol name
- For example:

```
typedef unsigned int word; // declare a new datatype
word w1; // now all variables of type word are unsigned ints
```

- You can do the same with function prototypes:

```
typedef int functype ( int n,  char ch );
functype* jumptable [10]; // create array of functions
```

•Note that you need to instantiate function pointers as pointers. The compiler won't let you declare a variable just of type functype, it must be of type functype*

Function Pointers Example

```
/** Program jump_table2.c */
#include <stdio.h>

typedef int functype ( int n,  char ch );

int duplicate_chars ( int n,  char ch ) {
    int i;
    for (i=0; i<n; i++) printf("%c\n",ch);
    return n;
}

int main ( int argc, char* args[] ) {
    functype* jumptable [10]; // create array of functions
    jumptable[0]= duplicate_chars;// set address for index 0
    jumptable[0](10,'a'); // call the function with parameters
    system("pause");
}
```

Calling one of the functions in the array

Enrichment (homework) Task

- Imagine that the C compiler you have chosen to use in an embedded systems project provides an inefficient implementation of the *switch* statement.
- Use function pointers and functions to implement the following switch statement as a jump table

```
int main ( int argc, char* args[] ) {
  int x;
  printf("Enter value: ");
  fscanf(stdin, "%d", &x);
  switch (x & 0xF) {
   case 1: case 2: case  3:
   case 4: case 5: case 6: case 7:
   case 8: case 9: printf("%d",x); break;
   case 10: case 11: case 12: printf("M"); break;
   case 13: printf("T"); break;
   default: printf("X"); break;
  };
}
```

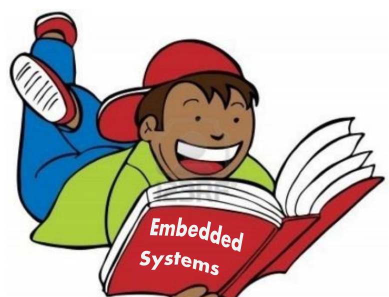
See file: classact_fn.c

classact_fn.c

The Next Episode…

Lecture P19

Interrupts and ADC & sampling details



Reminder: Read sections

3.2.4 Discretization of Values: Analog-to-Digital Converters