


1

Special Topics for Embedded Programming

Reference: The C Programming Language by Kernighan & Ritchie




1

2

Overview of Topics

- **Microprocessor architecture**
 - Peripherals
 - Registers
 - Memory mapped I/O
- **C programming for embedded systems**
- **Lab 1: General Purpose I/O**
 - Read data from input pins and write to output pins on the MPC5553
 - GPIO example code



2


3

Freescale MPC55xx Architecture

- 132 MHz 32-bit PowerPC, Temperature range: -40 to 125°C
- 1.5 MB of embedded Flash
- 64 KB on-chip static RAM
- 8 KB of cache
- 210 selectable-priority interrupt sources
- 3 DSPI (serial peripheral interface)
- 2 eSCI (serial communications)
- GPIO
- 2 x 40-ch. ADC
- 24-ch. eMIOS
- 2 CAN
- 32-ch. eTPU
- Direct Memory Access (DMA)

MPC5554 BLOCK DIAGRAM

| GPIO | ITAC | NEXUS |
|----------------|--------------------|---------------------|
| 2 CAN | 3 DSPI | 32-ch. eTPU |
| 2 x 40-ch. ADC | 2 eSCI | 14 0MB SRAM |
| 24-ch. eMIOS | 32-ch. DMA | (FEC) Ethernet |
| 1.5 MB Flash | 8 KB Unified Cache | 64 KB SRAM |
| SPE | PowerPC® 6200B | MMU |
| | | 32-bit External Bus |

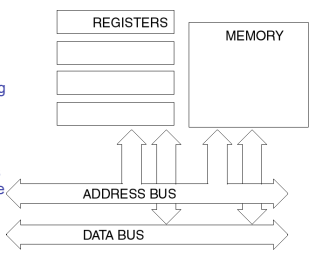


3


4

Microprocessor Architecture

- Microprocessor memory has location ("address") and contents (the data stored at a specific address in memory)
- Data are accessed by specifying a location on the address bus, and reading the contents of the specified address on the data bus
- Registers are memory locations used for calculations, to initialize the processor or check its status, or to access peripheral devices.
 - General purpose registers
 - Special purpose registers



The diagram shows a block labeled 'REGISTERS' with three sub-blocks and a block labeled 'MEMORY'. Below them are two horizontal double-headed arrows: 'ADDRESS BUS' and 'DATA BUS'. Four vertical arrows point from the 'ADDRESS BUS' to the 'REGISTERS' and 'MEMORY' blocks. Four vertical arrows point from the 'DATA BUS' to the 'REGISTERS' and 'MEMORY' blocks.



4

General Purpose Registers

- Hold data values before and after calculations
- C compilers automatically use these registers as resources to load/store data & perform calculations

```

    graph LR
      GPR3[GPR 3:5] --> Plus((+))
      GPR4[GPR 4:2] --> Plus
      Plus --> GPR2[GPR 2:7]
    
```

5

Special Purpose Registers: Memory Mapped I/O

- Access peripherals by writing to and reading from memory
- Each peripheral has a fixed range of memory addresses assigned to it
 - These are "memory-mapped registers," used for interacting with peripherals
- Memory locations can be:
 - Peripheral configuration registers
 - Peripheral status registers
 - inputs from the hardware
 - outputs to the hardware

Table 21-2. Module Memory Map

| Address | Register Name | Register Description | Size (bits) |
|---|---------------|--|-------------|
| Base = 0x0000_0000 (A) 0x0000_4000 (B) | ESGC0_CR1 | eSGC0 control register 1 | 32 |
| Base = 0x0004 | ESGC0_CR2 | eSGC0 control register 2 | 16 |
| Base = 0x0008 | ESGC0_DR | eSGC0 data register | 16 |
| Base = 0x000C | ESGC0_SR | eSGC0 status register | 32 |
| Base = 0x0010 | ESGC0_LCR | LN control register | 32 |
| Base = 0x0014 | ESGC0_LTR | LN transmit register | 32 |
| Base = 0x0018 | ESGC0_LRR | LN receive register | 32 |
| Base = 0x001C | ESGC0_LPR | LN cyclic redundancy check polynomial register | 32 |

6

C Programming for Embedded Systems

7

Primitive Data Types, Data Declaration

- Integer data types
 - Have both size and sign
 - char (8-bit)
 - short (16-bit)
 - int (32-bit)
 - long (32-bit)
 - signed (positive and negative)
 - unsigned (positive only)
- Floating-point types
 - Only have size
 - Can always be positive or negative
 - float (32-bit)
 - double (64-bit)
- Data declarations should be at the top of a code block

```

{
/* Top of Code Block
*/
signed char A;
char input;
unsigned short var;
int output;
unsigned long var2;

float realNum;
double realNum2;

}
    
```

8

Freescal Defined Types

9

- See `freescal/typedefs.h`

```
- typedef signed char int8_t;
- typedef unsigned char uint8_t;
- typedef volatile signed char vint8_t;
- typedef volatile unsigned char vuint8_t;
- typedef signed short int16_t;
- typedef unsigned short uint16_t;
- ...
```



Functions and Function Prototypes

10

- Function Prototype declares name, parameters and return type prior to the functions actual declaration
- Information for the compiler; does not include the actual function code
- You will be given function prototypes to access peripherals
- Pro forma: `RetType FcnName (ArgType ArgName, ...);`

```
int Sum (int a, int b); /* Function Prototype */
void main()
{
    c = Sum ( 2, 5 ); /* Function call */
}
int Sum (int a, int b) /* Function Definition */
{
    return ( a + b );
}
```



C vs. C++

11

- C++ language features cannot be used
 - No `new`, `delete`, `class`
 - Comment with `/* */`, not `//`
- Variables must be declared at top of code blocks

```
{
    int j;
    double x;

    /* code */
    int q; /* only in C++, not in C */
    /* code */
}
```



11

Useful Features for Embedded Code

12

- Type Qualifiers
 - Volatile
 - Static
- Pointers
- Structures
- Unions
- Bit Operations
- Integer Conversions




12

Volatile Type Qualifier

13

- Variables that tend to be reused repeatedly in different parts of the code are often identified by compilers for optimization.
 - These variables are often stored in an internal register that is read from or written to whenever the variable is accessed in the code.
 - This optimizes performance and can be a useful feature
- **Problem for embedded code:** Some memory values may change without software action!
 - Example: Consider a memory-mapped register representing a DIP-switch input
 - Register is read and saved into a general-purpose register
 - Program will keep reading the same value, even if hardware has changed
- Use `volatile` qualifier:
 - Value is loaded from or stored to memory every time it is referenced
 - Example: `volatile unsigned char *GPIO_pointer;`
 - Freescale type: `vuint8_t *GPIO_pointer;`




13

Static Type Qualifier

14

- Variables local to functions are destroyed when the function returns a value and exits (they have *local scope*)
- In embedded code we often want a variable to retain its value between function calls
 - Consider a function that senses a change in the crankshaft angle: The function needs to know the previous angle in order to compute the difference
- If a local variable within a function is declared static, it is stored in the “heap” (a dedicated pool of memory) instead of the function “stack” (a temporary storage) and will thus retain its value between subsequent function calls
- When used with global variables and functions, static limits the scope of the variable to its file.
- Example: `static int x = 2 ;`




14

Pointers

15

- Every variable has an *address* in memory and a *value*
- A pointer is a variable that stores an address
 - The value of a pointer is the location of another variable
- The size of a pointer variable is the size of an address
 - 4 bytes (32 bits) for the MPC5553
- Two operators used with pointers
 - `&` operator returns the address of a variable
 - `*` is used to “de-reference” a pointer (*i.e.*, to access the *value* at the *address* stored by a pointer)



15

Pointer Example


16

| Address | Value | Variable |
|---------|-------|----------|
| 0x100 | 5 | x |
| 0x104 | 5 | y |
| 0x108 | 0x100 | ptr |

```

int x, y;          /* x is located at address 0x100 */
int *ptr;         /* This is how to declare a pointer */

x = 5;
ptr = &x;         /* The value of ptr is now 0x100 */
y = *ptr;         /* y now has the value 5 */
*ptr = 6;         /* The value at address 0x100 is 6 */
    
```



16

More Pointer Examples


- Declare a pointer,


```
volatile int *p;
```
- Assign the address of the I/O memory location to the pointer,


```
p = (volatile int*) 0x30610000;
/* 4-byte long, address of some memory-mapped register */
```
- Output a 32-bit value by setting the value of bytes at 0x30610000,


```
*p = 0x7FFFFFFF; /* Turn on all but the highest bit */
```
- Alternatively,


```
((volatile unsigned short*)
(0x30610000)) = 0x7FFF;
```




Structures

- A struct holds multiple variables
 - Divides range of memory into pieces that can be referenced individually
 - The following structure creates a variable x of type rational with members n and d

```
struct rational { short n;
short d; } x;
x.n = 2;
x.d = 5;
```


- **Recall:** We can treat a peripheral's memory map as a range of memory.
- **Result:** We can use a structure and its member variables to access peripheral registers.



Structures

- Access structure member variables using "." and "->"
 - . is used with structures
 - -> is used with pointers-to-structures
- Example
 - bob is a variable of type student
 - Assign firstnm, lastnm and age
 - Increment age


```
/* Definition */
struct student{
char firstnm[32];
char lastnm[32];
int age;
};
void main()
{
struct student bob;
struct student *pbob;
bob.age = 20;
strcpy(bob.firstnm, "Bob");
strcpy(bob.lastnm, "Smith");
pbob = &bob;
pbob->age++;
/* same as (*pbob).age++ */
};
```



Structure bit-fields

Number of bits per data member in structures is specified with ".n"

```
struct RGB_color{
unsigned char r:2; /* 2-bits */
unsigned char g:2;
unsigned char b:2;
/* padding */
unsigned char:2; /*don't need to explicitly do this. */
};
/* reset g */
struct RGB_color clr; clr.g =
```



Bit-Fields Diagram

21

- But what if we want to access r,g & b all at the same time?

The diagram shows a vertical stack of memory addresses: 0x0000, 0x0100, 0x0200, and an ellipsis. Below these is a horizontal structure with four fields: 'r', 'g', 'b', and 'space'.

Unions

22

- Multiple ways to view memory locations
- Unions contain member variables
 - Member variables **share same memory**
 - All variables are overlaid on each other
- Changing one member results in the other members being changed
- Union is as long as longest member

Two ways
To read the
Same memory
location

Memory Location 0x0716

| | | | |
|--------|-------|-------|---------|
| rgb: 6 | | | padding |
| r : 2 | g : 2 | b : 2 | padding |

Unions

23

The diagram shows a horizontal structure with four fields: 'r', 'g', 'b', and 'space'. Above the 'r', 'g', and 'b' fields is a bracket labeled 'rgb'. Below the fields are memory addresses: 0x00, 0x02, 0x04, and 0x06.

```

union RGB_color{
  struct {
    unsigned char r:2,g:2,b:2;
    unsigned char:2;
  };
  struct {
    unsigned char rgb:6;
    unsigned char:2;
  };
};
union RGB_color clr;
            
```

Pros & Cons of Using Structures

24

- Pros**
 - Readable code
 - Simple way to set or clear individual bits
- Cons**
 - Relies on the compiler implementation
 - Assembly code for a simple bit-write is much longer than if registers were written to directly

25

Constants and Bit Operations

Tools for Manipulating Values




25

26

Integer Constants

- Binary constant: *Obnumber*
 - Example: `0b10110000`
- Octal constant: *Onumber* (prefix is a zero)
 - Example: `0775`
- Hexadecimal constant: *Oxnumber*
 - Example: `0xffff` or `0xFFFF`




26

27

Bit Operations

- Bit-shifts
 - Right shift: `num >> shift 0b1001 >> 2 → 0b0010`
 - Left shift: `num << shift 0b0011 << 2 → 0b1100`
- Masking
 - Bit-or: `num | mask 0b0001 | 0b1000 → 0b1001`
 - Bit-and: `num & mask 0b1001 & 0b1000 → 0b1000`
- Complement
 - Not: `~ num ~ 0b0101 → 0b1010`
- Set bits
 - set the 5th bit `x = x | (1 << 4);`
 - set the 5th bit `x |= (1 << 4);`
- Clear bits
 - clear 5th and 6th bit `x = x & ~(0b11 << 4);`
 - clear 5th and 6th bit `x &= ~(0b11 << 4);`




27

28

Type Conversions

- Explicit Casts
 - For specifying the new data type
 - Syntax: `(type-name)expression`
 - Example: `(int)largeVar`
- Integral Promotion
 - Before basic operation (`+` `-` `*` `/`), both operands converted to same type
 - The smaller type is "promoted" (increased in size) to the larger type
 - Value of promoted type is preserved
- Implicit Casts
 - Assigning a value into a different type
 - Widening conversion – preserve value of expression
 - `short x = 10; long y = x;`
 - Narrowing conversions – do NOT preserve value
 - `unsigned long x = 257;`
 - `unsigned char y = x;`



28

Lab 1 GPIO

33

- Three registers for each pin
 - Pad Configuration register (PCR)
 - General Purpose Data Input register (GPDI)
 - General Purpose Data Output register (GPDO)
- Many other SIU registers
- Most pins have alternate function (connected to other peripherals)
- See SIU memory map, Table 6.2

| Address | Register Name | Register Description | Size (bits) |
|---------------|---------------|--------------------------------------|-------------|
| Base + 0x0000 | SIU_RESERVED | Reserved | — |
| Base + 0x0004 | SIU_MCR | MCU ID register | 32 |
| Base + 0x0008 | — | Reserved | — |
| Base + 0x000C | SIU_BIRD | Board ID register | 32 |
| Base + 0x0010 | — | Reserved | — |
| Base + 0x0014 | SIU_PCR0 | Pad configuration registers 0-255 | 16 |
| Base + 0x0018 | SIU_PCR255 | — | — |
| Base + 0x001C | — | Reserved | — |
| Base + 0x0020 | SIU_GPDO0 | GPIO pin data output registers 0-255 | 8 |
| Base + 0x0024 | SIU_GPDO255 | — | — |
| Base + 0x0028 | — | Reserved | — |
| Base + 0x002C | SIU_GPDI0 | GPIO pin data input registers 0-255 | 8 |
| Base + 0x0030 | SIU_GPDI255 | — | — |
| Base + 0x0034 | — | Reserved | — |

Memory Map Table 6.2

GPIO Registers

34

- For each pin:
 - Pad Configuration Register (PCR) (6.3.1.12)
 - Set pin purpose
 - Turn on/off voltage buffers
 - Data Input Register (GPDI) (6.3.1.14)
 - Read voltage on pin
 - On is 1, off is 0
 - Data Output Register (GPDO) (6.3.1.13)
 - Set voltage on pin
 - On is 1, off is 0

Figure 6-129. GPIO Pin Data Input Register 0-213 (SIU_GPDI_n)

Figure 6-128. GPIO Pin Data Output Register 0-213 (SIU_GPDO_n)

SIU Pad Configuration Register

35

- Section 6.3.1.12, Table 6-15. SIU_PCR Field Descriptions
- **PA**: Pin assignment (selects the function of a multiplexed pad)
- **OBE**: Output buffer enable
- **IBE**: Input buffer enable
- **DSC**: Drive strength control
- **ODE**: Open drain output enable
- **HYS**: Input hysteresis enable
- **SRC**: Slew rate control
- **WPE**: Weak pull up/down enable (*Disable* pull up)
- **WPS**: Weak pull up/down select

SIU Pad Configuration Register

36

```

typedef union siu_pcr_u {
    /* Pad Configuration Registers */
    volatile unsigned short REG;
    struct {
        volatile unsigned short :3;
        volatile unsigned short PA:3;
        volatile unsigned short OBE:1;
        volatile unsigned short IBE:1;
        volatile unsigned short DSC:2;
        volatile unsigned short ODE:1;
        volatile unsigned short HYS:1;
        volatile unsigned short SRC:2;
        volatile unsigned short WPE:1;
        volatile unsigned short WPS:1;
    } FIELDS;
} SIU_PCR;
    
```

- Union accesses entire register or individual bit fields
- Provide one union generic enough to suit any SIU pad
- There are over 200 configuration registers!
 - Address each configuration register by declaring your union as a pointer
 - Use pointer indexing (i.e.: `padptr[122]`) to access a specific register

MPC5553 Register Definitions

```

/*****
*/
/* FILE NAME: mpc5553.h
   COPYRIGHT (c) Freescale 2005 */
/* VERSION: 1.5
   All Rights Reserved */
*/
*/
/* DESCRIPTION:
*/
/* This file contains all of the
   register and bit field definitions
   for */
/* MPC5553.
*/
/*****
*/
/*****NOTE! this file is auto-generated
   please do not edit it!<<<<*/

```

- **freescale/mpc553.h** has registers, bit field definitions
- Don't have to write your own structure (except for lab #1)
- Register addresses at the bottom of mpc553.h
- Register definitions use freescale/typedefs.h

37

Read and Write GPIO

```

#include<eecs461.h> /* Typedefs and processor initialization */
#include "my_siu_pcr.h" /* Your SIU PCR configuration goes here */
void main()
{
    int i;
    unsigned char op1, op2;
    int result;
    int temp;
    volatile SIU_PCR *siu_pcr_ptr; /* pointers to registers */
    volatile unsigned char *siu_gpdi_ptr;
    volatile unsigned char *siu_gpdo_ptr;

    siu_pcr_ptr = (SIU_PCR*)(0xc3f90040); /* SIU_BASE + 0x40 for pcr */
    siu_gpdo_ptr = (unsigned char*)(0xc3f90600); /* SIU_BASE + 0x600 for gpdo */
    siu_gpdi_ptr = (unsigned char*)(0xc3f90800); /* SIU_BASE + 0x800 for gpdi */
    /* See Table 6.2 */
}

```

38

Read and Write GPIO (continued)

```

/* configure input dipswitches */
for(i=122; i<130; i++)
{
    siu_pcr_ptr[i].FIELDS.PA = 0; /* GPIO */
    siu_pcr_ptr[i].FIELDS.IBE = 1; /* Input */
    siu_pcr_ptr[i].FIELDS.WPE = 0; /* Weak pull up disabled */
}

/* configure output leds */
for(i=28; i<33; i++)
{
    siu_pcr_ptr[i].FIELDS.PA = 0; /* GPIO */
    siu_pcr_ptr[i].FIELDS.OBE = 1; /* Output */
    siu_pcr_ptr[i].FIELDS.WPE = 0;
}

init_EECS461(1); /* Call this function with lab number = 1 to init processor */

```

39

Read and Write GPIO (continued)

```

while(1)
{
    /* get the number contained on DIP 126-129 and 122-125 */
    /* read the bit
       shift left 1
       read the next bit and continue */
    /* calculate the sum */
    /* display the result on LED 28-32 */
    /* write result LSB
       shift right 1
       write the next bit and continue */
}

```

40


41

Read Serial Port and Write GPIO

- Use keyboard input instead of DIP switches
 - Input 2 digits (0-9), calculate sum and output binary result to LEDs
 - `serial.c`, `serial.h` provided
- ASCII to binary conversion

| ASCII | Binary | Decimal |
|----------|--------|---------|
| 011 0001 | 0001 | 1 |
| 011 0010 | 0010 | 2 |
| ... | ... | ... |
| 011 1001 | 1001 | 9 |

Binary = ASCII & 0xF



42

Lab 1 GPIO

- Read the lab 1 documentation
- Organize your file structure as described in section 2.3
- Pre-lab
 - Read the manual and answer the questions
- In-lab
- Post-lab
 - Summarize the concepts learned in lab 1

