CprE 288 – Introduction to Embedded Systems

Instructors:
Dr. Phillip Jones
Overview

- Announcements
- Bitwise Operations
  - Set, clear, toggle and invert bits
  - Shift bits
  - Test bits
- Memory-Mapped General Purpose I/O (GPIO) Ports
  - Textbook reading:
    - All of 2.6.3: GPIO (~6.5 pages)
    - 4.5.7.2 – 4.5.7.2.2.2, and 4.5.7.2.5.3 (Fig 4.33: DRAModel.c code) (~6.5 pages)
    - 7.1, 7.2, 7.3, Example application 7.4: Figure 7.7. (~13 pages)
- Memory-Mapped Devices I/O
Announcements

- HW 2 due Tuesday 9/11
- HW 3 due Monday 9/17
- **Quiz 4 (15 min):** Thursday 9/20 at beginning of class: in Canvas (Let me know if you need a paper version, before day of quiz)
  - Can use one side of one page of notes (**must be on paper**)
    - Will be collected as part of class participation grade
  - Readings
    - Quiz 3 Datasheet readings
    - NVIC: Textbook 2.4
    - UART: Datasheet 14, Textbook 8.5
  - **UART: In class exercise: Thur 9/20** (Class Participation grade): Directions to be given
• **Quiz 3 (15 min):** Coverage
  - Lecture material (9/6 – 9/11)
  - CPRE 288 Datasheet Trainer
  - Datasheet: Chapter 10 (~15 pages of ~60 pages)
    - See email for ~15 pages you will be responsible for.
  - Textbook readings to help with Datasheet:
    - All of 2.6.3: GPIO (~6.5 pages)
      - Except, light read of 2.6.3.1: “The System Clock
      - Skip sections 2.6.3.3.4 - 5: “Commit and Interrupt Control Regs”
      - Section 2.6.3.3.6 “Pad Control Regs: Skip GPIODR2R – GPIOSLR
        - Skip 2.6.3.3.7: Identification Reg.
    - 4.5.7.2 – 4.5.7.2.2.2, and 4.5.7.2.5.3 (Fig 4.33: DRAModel.c code) (~6.5 pages)
    - 7.1, 7.2, 7.3, Example application 7.4: Figure 7.7. (~13 pages)
BITWISE OPERATIONS
Why Bitwise Operation

Why use bitwise operations in embedded systems programming?

Each single bit may have its own meaning
- Push button array: Bit $n$ is 0 if push button $n$ is pushed
- LED array: Set bit $n$ to 0 to light LED $n$

Data from/to I/O ports may be packed
- Two bits for shaft encoder, six bits for push button packed in PINC
- Keypad input: three bits for row position, three bits for column position

Data in memory may be packed to save space
- Split one byte into two 4-bit integers

http://class.ece.iastate.edu/cpre288
Read the input:

```
GPIO_PORTE_R;
```

Then, how does the code get to know which button is being pushed?

Connected to PORTE, bits 5-0
PINB, bits 1-0 are input from shaft encoder
Bitwise Operations

Common programming tasks:
• Clear/Reset certain bit(s)
• Set certain bit(s)
• Test if certain bit(s) are cleared/reset
• Test if certain bit(s) are set
• Toggle/invert certain bits
• Shift bits around
Bitwise Operators: Clear/Reset Bits

C bitwise AND: \&

ch = ch & 0x3C;

What does it do?

Consider a single bit $x$

- $x \text{ AND } 1 = x$ Preserve
- $x \text{ AND } 0 = 0$ Clear/Reset
Bitwise Operators: Clear/Reset Bits

ch = ch & 0x3C;

<table>
<thead>
<tr>
<th>X7</th>
<th>X6</th>
<th>X5</th>
<th>X4</th>
<th>X3</th>
<th>X2</th>
<th>X1</th>
<th>X0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**AND**

<table>
<thead>
<tr>
<th>X7</th>
<th>X6</th>
<th>X5</th>
<th>X4</th>
<th>X3</th>
<th>X2</th>
<th>X1</th>
<th>X0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Clear bits 7, 6, 1, 0
Preserve bits 5, 4, 3, 2

**Clear bit(s):** Bitwise-AND with a mask of 0(s)
Another example:

char op1 = 1011 1100; We want to clear bit 4 to 0.
char op2 = 1110 1111; We use op2 as a mask
char op3;

op3 = op1 & op2;

1011 1100
AND 1110 1111
1010 1100
Class Exercise

char ch;
short n;

Clear the upper half of ch

Clear every other bits of ch starting from bit-position 0

Clear the lower half of n
C bitwise OR: |

ch = ch | 0xC3;  

What does it do?

Consider a single bit \( x \)

\[
\begin{align*}
    x \text{ OR } 1 &= 1 & \text{Set} \\
    x \text{ OR } 0 &= x & \text{Preserve}
\end{align*}
\]
Bitwise Operators: Set Bits

ch = ch | 0xC3;

<table>
<thead>
<tr>
<th>X7</th>
<th>X6</th>
<th>X5</th>
<th>X4</th>
<th>X3</th>
<th>X2</th>
<th>X1</th>
<th>X0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

OR

| 1  | 1  | X5 | X4 | X3 | X2 | 1  | 1  |

Set bits 7, 6, 1, 0
Preserve bits 5, 4, 3, 2

Set bit(s): Bitwise-OR with a mask of 1(s)
Bitwise Operators: Set Bit

Another example:

char op1 = 1000 0101;  We want to set bit 4 to 1.
char op2 = 0001 0000;  We use op2 as a mask
char op3;

op3 = op1 | op2;

\[
\begin{array}{c}
1000 0101 \\
\text{OR} \\
0001 0000 \\
\hline
1001 0101
\end{array}
\]
Bitwise Operators: Toggle Bits

C bitwise XOR: ^

ch = ch ^ 0x3C;

What does it do?

Consider a single bit x

\[ x \text{ XOR } 1 = \overline{x} \quad \text{Toggle} \]
\[ x \text{ XOR } 0 = x \quad \text{Preserve} \]
Bitwise Operators: Toggle Bits

C bitwise XOR: ^

ch = ch ^ 0x3C;

<table>
<thead>
<tr>
<th>X7</th>
<th>X6</th>
<th>X5</th>
<th>X4</th>
<th>X3</th>
<th>X2</th>
<th>X1</th>
<th>X0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

XOR

<table>
<thead>
<tr>
<th>X7</th>
<th>X6</th>
<th>X5</th>
<th>X4</th>
<th>X3</th>
<th>X2</th>
<th>X1</th>
<th>X0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Toggle bits 5, 4, 3, 2
Preserve bits 7, 6, 1, 0

Toggle bit(s): Bitwise-XOR with a mask of 1(s)
Bitwise Operators: Invert Bits

C bitwise invert: \( \sim \)

\[
\begin{align*}
\text{INV} & \quad X_7 & X_6 & X_5 & X_4 & X_3 & X_2 & X_1 & X_0 \\
\overline{X_7} & \quad \overline{X_6} & \quad \overline{X_5} & \quad \overline{X_4} & \quad \overline{X_3} & \quad \overline{X_2} & \quad \overline{X_1} & \quad \overline{X_0}
\end{align*}
\]

Example: \( ch = 0b00001111; \)

\( \sim ch == 0b11110000 \)
char ch;
short n;

Set the lower half of ch

Set every other bits starting from 0 of ch

Set bit 15 and bit 0 of n

Toggle bits 7 and 6 of ch
Bitwise Operators: Shift-Left

unsigned char my_reg = 0b00000001;
unsigned char shift_amount = 5;
unsigned char my_result;

my_result = my_reg << shift_amount;

<<, shifts “my_reg”, “shift_amount” places to the left
0s are shifted in from the right
unsigned char my_reg = 0b10000000;
unsigned char shift_amount = 5;
unsigned char my_result;

my_result = my_reg >> shift_amount;

With unsigned type, >> is shift-to-right logical
0s are shifted in from the left
Bitwise Operators: Shift-Right Arithmetic

`signed` char my_reg = `0b10000000`;
unsigned char shift_amount = 5;
unsigned char my_result;

my_result = my_reg >> shift_amount;

my_reg = `0b01111111`;
my_result = my_reg >> shift_amount;

With signed type, `>>` is **shift-right arithmetic**
Sign bit value are shifted in from the left
Bitwise Operators: Shift and Multiple/Divide

\[ n << k \] is equivalent to \[ n \times 2^k \]

Example:
\[ 5 << 2 \quad = \quad 5 \times 4 \quad = \quad 20 \]
\[ \text{0b0000 0101} << 2 \quad = \quad \text{0b0001 0100} \]

\[ n >> k \] is equivalent to \[ n \div 2^k \]

Example:
\[ 20 >> 2 \quad = \quad 5 \]
\[ \text{0b0001 0100} >> 2 \quad = \quad \text{0b0000 0101} \]
Shift-Right Arithmetic: Why shift in the sign bit?

Example: (char) 32 >> 2 = 32 / 4 = 8
0b0010 0000 >> 2 = 0b0000 1000

Example: (char) -32 >> 2 = -32 / 4 = -8
0b1110 0000 >> 2 = 0b1111 1000
What’s the effect of the following state?

```c
#define BIT_POS 4
ch = ch | (1 << BIT_POS);
```

What is (1 << 4)?

```
  0000 0001
<< 4
  0001 0000
```

In general case: (1 << n) yields a mask of a 1 at bit n.
The effect of the statement: Set bit 4
Another example:
unsigned char my_mask = 0000 0001;
unsigned char shift_amount = 5;
unsigned char my_result = 1101 0101; Want to force bit 5 to a 1

my_result = my_result | (my_mask << shift_amount);

1101 0101 | 00100000
1101 0101

OR
0010 0000
1111 0101

Shift the 1(s) of the MASK to the appropriate position, then OR with my_result to force corresponding bit positions to 1.
What’s the effect of the following state?

```c
#define BIT_POS 4
ch = ch & ~(1 << BIT_POS);
```

What is \(~(1 << 4)\)?

```
0000 0001
0001 0000
1110 1111
```

In general case: \(~(1 << n)\) yields a mask of a 0 at bit \(n\)

*Note: Compiler does the calculation at compilation time*
unsigned char my_mask = 0000 0111;
unsigned char shift_amount = 5;
unsigned char my_result = 1011 0101; Want to force bit 7-5 to a 0

my_result = my_result & ~(my_mask << shift_amount);

Shift the 0(s) of the MASK to the appropriate position, then AND with my_result to force corresponding bit positions to 0.
Exercise

unsigned char ch;
unsigned short n;

Divide n by 32 in an efficient way

Swap the upper half and lower half of ch
Bitwise Testing

Remember, conditions are evaluated on the basis of zero and non-zero.

The quantity 0x80 is non-zero and therefore TRUE.

if (0x02 | 0x44)
    TRUE or FALSE?
Example

Find out if bit 7 of variable nVal is set to 1

Bit 7 = 0x80 in hex

```c
if ( nVal & 0x80 )
{
    ...
}
```

What happens when we want to test for multiple bits?

if statement looks only for a non-zero value, a non-zero value means at least one bit is set to TRUE
Bitwise Testing: **Any** Bit is Set to 1?

**Example**
See if bit 2 or 3 is set to 1

Bits 2,3 = 0x0C in hex

```c
if (nVal & 0x0C) {
    Some code...
}
```

What happens for several values of nVal?

- nVal = 0x04  bit 2 is set  Result = 0x04  TRUE
- nVal = 0x0A  bits 3,1 are set  Result = 0x08  TRUE
- nVal = 0x0C  bits 2,3 are set  Result = 0x0C  TRUE

http://class.ece.iastate.edu/cpre288
Bitwise Testing: All Bits Are Set to 1?

Why does this present a problem?
- What happens if we want to see if both bits 2 and 3 are set, not just to see if one of the bits is set to true?
- Won’t work without some other type of test

Two solutions
- Test each bit individually
  
  ```c
  if ( (nVal & 0x08) && (nVal & 0x04))
  ```
- Check the result of the bitwise AND
  
  ```c
  if ((nVal & 0x0C) == 0x0C)
  ```

Why do these solutions work?
1. Separate tests – Check for each bit and specify logical condition
2. Equality test – Result will only equal 0x0C if bits 2 and 3 are set
Exercise

char ch;

Test if any of bits 7, 6, 5, 4 is set to 1

Test if all of bits 7, 6, 5, 4 are set to 1

Test if all of bits 7 and 6 are set to 1, and if bits 5 and 4 are cleared to 0
Exercise

Write a program to count the number of binary 1s in integer \( n \)

```c
unsigned int count = 0;
unsigned int n;
```
Bitwise operations: Summary

- Forcing bits to 0: & (AND)
- Forcing bits to 1: | (OR)
- Toggle bits: ^ (XOR)
- Testing for bits set to 1
- Testing for bits cleared to 0
- Generic systematic testing approach:

  if( (val & MASK_ALL1s) == MASK_ALL1 &&
      (~val & MASK_ALL0s) == MASK_ALL0s &&
      (val & MASK_ANY1s) &&
      (~val & MASK_ANY0s) )

  Where: MASK Xxx is a mask with 1s in the positions being tested
Memory Mapped I/O
Memory Mapped I/O

• Package Pins
• Pins with shared (multiplexed) functionality
  – General Purpose I/O
  – Device I/O
• Memory mapped registers
  – Configuration registers
  – Data registers
  – Status registers
TM4C123 I/O Ports

• 64 package pins
TM4C123 I/O Ports

• 64 package pins

Allows Software to access the world outside of the chip
TM4C123 I/O Ports

- 64 package pins
- 6 GPIOs Ports(A – F)
  - Each 8-bits

http://class.ece.iastate.edu/cpre288
TM4C123 I/O Ports

- 64 package pins
- 6 GPIOs Ports (A – F)
  - Each 8-bits

http://class.ece.iastate.edu/cpre288
TM4C123 I/O Ports

- 64 package pins
- 6 GPIOs Ports (A – F)
  - Each 8-bits

http://class.ece.iastate.edu/cpre288
TM4C123 I/O Ports

- 64 package pins
- 6 GPIOs Ports (A – F)
  - Each 8-bits

http://class.ece.iastate.edu/cpre288
TM4C123 I/O Ports

- 64 package pins
- 6 GPIOs Ports (A – F)
  - Each 8-bits

http://class.ece.iastate.edu/cpre288
TM4C123 I/O Ports

- 64 package pins
- 6 GPIOs Ports (A – F)
  - Each 8-bits
TM4C123 I/O Ports

- 64 package pins
- 6 GPIOs Ports(A – F)
  - Each 8-bits

http://class.ece.iastate.edu/cpre288
TM4C123 I/O Ports

- 64 package pins
- 6 GPIOs Ports(A – F)
  – Each 8-bits
- Many GPIOs have alternate functions

http://class.ece.iastate.edu/cpre288
TM4C123 I/O Ports

- 64 package pins
- 6 GPIOs Ports(A – F) – Each 8-bits
- Many GPIOs have alternate functions

[Diagram showing I/O ports connections]

http://class.ece.iastate.edu/cpre288
TM4C123 I/O Ports

- 64 package pins
- 6 GPIOs Ports (A – F)
  - Each 8-bits
- Many GPIOs have alternate functions

http://class.ece.iastate.edu/cpre288
Software writes Configuration Register

Software writes Data Register

Hardware writes Data Register

Out side of chip world
TM4C123 I/O Ports

Software writes Configuration Register

Software writes Data Register

Out side of chip world
CPU (Software) | Data Memory
---|---
address | 
![](http://class.ece.iastate.edu/cpre288)

data |

Software writes Configuration Register

Software writes Data Register

select |

PA1 |

MUX |

U0Tx |

Hardware device logic

Data | Config | Status

Out side of chip world

http://class.ece.iastate.edu/cpre288
Software writes Configuration Register

Software writes Data Register

CPU (Software)

Address Decode

Data Memory

MUX

PA1

U0Tx

Out side of chip world

Hardware device logic

Data

Config

Status

http://class.ece.iastate.edu/cpre288
TM4C123 I/O Ports

CPU (Software) -> Address Decode

Address Decode -> Data Memory

Data Memory -> Hardware device logic

Hardware device logic -> Data, Config, Status

CPU (Software) -> PA1

PA1 -> U0Tx

Out side of chip world

http://class.ece.iastate.edu/cpre288
TM4C123 I/O Ports

CPU (Software) -> Address Decode

Address Decode:
- 0x20000000 - 0x20007FFF
- 0x400043FC
- 0x4000C000

PA1 MUX

U0Tx

Data Memory

Hardware device logic:
- Data
- Config
- Status

Out side of chip world

http://class.ece.iastate.edu/cpre288
TM4C123 I/O Ports

CPU (Software)

Address Decode

0x400043FC

0x4000C000

0x20000000

–

0x20007FFF

Memory Mapped Registers

Data

Config

Status

Hardware device logic

MUX

select

PA1

U0Tx

Out side of chip world

Data

Address Decode

0x20000000 – 0x20007FFF

http://class.ece.iastate.edu/cpre288
TM4C123 I/O Ports

CPU (Software) → Address Decode

Address Decode:
- 0x20000000 - 0x20007FFF
- 0x400043FC
- 0x4000C000

Data Memory

MUX

PA1

U0Tx

Hardware device logic:
- Data
- Config
- Status

Out side of chip world

http://class.ece.iastate.edu/cpre288
TM4C123GH6PM:

- 6 general purpose I/O ports: Port A, B, C, D, E, F
- Processor communicates with them through memory mapped registers.
- A set of data and control registers associated with each port.
Memory Mapped General Purpose I/O (GPIO) Ports

- Processor communicates with arbitrary attachments using ports
- Each GPIO port has several registers, we will focus on 4.

SYSCTL_RCGCGPIO_R – Enables the port’s system clock
GPIO_PORTx_DATA_R – 8bit data register (read/write)
GPIO_PORTx_DIR_R – Data direction register
GPIO_PORTx_DEN_R – Digital enable register
SYSCTL_RCGCGPIO_R (GPIO Run Mode Clock Gating Control)

• A GPIO port’s clock must be enabled to use its registers. (Your program will crash if you attempt to update an GPIO port’s registers without enabling its clock)
  
• bits 5-0 in SYSCTL_RCGCGPIO_R represent ports F-A.

E.g.:

//Enable Port F & B clocks
SYSCTL_RCGCGPIO_R |= 0b100010;
GPIO_PORTx_DIR_R (GPIO Direction Register)
- Writing ‘0’ to a bit programs the corresponding pin of the Port as input
- Writing ‘1’ to a bit programs the corresponding pin of the Port as output

Example:
//All bits of port A prog for input except bit0
GPIO_PORTA_DIR_R = 0b00000001;
**GPIO_PORTX_DATA_R Register:**

For output configured port: If a bit in in the Port’s DATA register is written when the corresponding pin is configured as an output, then the port’s pin will be driven with this value.

**Write to a port using its DATA register.**

E.g.:

```c
SYSCTL_RCGCGPIO_R |= 0b000001; //enable PORTA clock
GPIO_PORTA_DIR_R = 0xFF; //set port A dir to output
GPIO_PORTA_DEN_R = 0xFF; //Enable pins 0-7
GPIO_PORTA_DATA_R = my_char; //set port A to my_char
```

Directly using the definition of `GPIO_PORTA_DATA_R`

```c
*((char *)0x400043FC) = my_char;
```
Memory Mapped General Purpose I/O (GPIO) Ports

// tm4c123gh6pm.h
//*****************************************************************************/
// GPIO registers (PORTA)
//*****************************************************************************/
#define GPIO_PORTA_DATA_BITS_R ((volatile unsigned long *)0x40004000)
#define GPIO_PORTA_DATA_R       (*((volatile unsigned long *)0x400043FC))
#define GPIO_PORTA_DIR_R        (*((volatile unsigned long *)0x40004400))
#define GPIO_PORTA_IS_R         (*((volatile unsigned long *)0x40004404))
#define GPIO_PORTA_IBE_R        (*((volatile unsigned long *)0x40004408))
#define GPIO_PORTA_IEV_R        (*((volatile unsigned long *)0x4000440C))
#define GPIO_PORTA_IM_R         (*((volatile unsigned long *)0x40004410))
#define GPIO_PORTA_RIS_R        (*((volatile unsigned long *)0x40004414))
#define GPIO_PORTA_MIS_R        (*((volatile unsigned long *)0x40004418))
#define GPIO_PORTA_ICR_R        (*((volatile unsigned long *)0x4000441C))
#define GPIO_PORTA_AFSEL_R      (*((volatile unsigned long *)0x40004420))
#define GPIO_PORTA_DR2R_R       (*((volatile unsigned long *)0x40004500))
#define GPIO_PORTA_DR4R_R       (*((volatile unsigned long *)0x40004504))
#define GPIO_PORTA_DR8R_R       (*((volatile unsigned long *)0x40004508))
#define GPIO_PORTA_ODR_R        (*((volatile unsigned long *)0x4000450C))
#define GPIO_PORTA_PUR_R        (*((volatile unsigned long *)0x40004510))
#define GPIO_PORTA_PDR_R        (*((volatile unsigned long *)0x40004514))
#define GPIO_PORTA_SLR_R        (*((volatile unsigned long *)0x40004518))
#define GPIO_PORTA_DEN_R        (*((volatile unsigned long *)0x4000451C))
#define GPIO_PORTA_LOCK_R       (*((volatile unsigned long *)0x40004520))
#define GPIO_PORTA_CR_R         (*((volatile unsigned long *)0x40004524))
#define GPIO_PORTA_AMSEL_R      (*((volatile unsigned long *)0x40004528))
#define GPIO_PORTA_PCTL_R       (*((volatile unsigned long *)0x40004532))
#define GPIO_PORTA_ADCCTL_R     (*((volatile unsigned long *)0x40004534))
#endif

http://class.ece.iastate.edu/cpre288
Example: Initialize Push Buttons

/// Initialize PORTE to accept push buttons as input
void init_push_buttons(void) {
    SYSCTL_RCGCGPIO_R |= 0b010000; // Enable port E clock
    GPIO_PORTE_DIR_R &= 0xC0; // Setting PE0-PE5 to input
    GPIO_PORTE_PUR_R |= 0x3F; // Set pins' pullup resistors
    GPIO_PORTE_DEN_R = 0x3F; // Enable pins 0-5
}

Push Button port connection

- Port E, pin 0 to pin 5 (button SW1 to SW6)
- Configure Push Button connection to be inputs
/** Example: Initialize Shaft Encoder **/

```c
void shaft_encoder_init(void) {
    SYSCTL_RCGCGPIO_R |= 0b000010; // Enable Port B clock
    GPIO_PORTB_DIR_R &= 0xFC; // Setting PB0-PB1 to input
    GPIO_PORTB_PUR_R |= 0x03; // Set pins' pull-up resists
    GPIO_PORTB_DEN_R |= 0x03; // Enable PB0, PB1
}
```

**Shaft encoder port connection**
- Port B, pin 1 and 0
- Configure shaft encode connections as Inputs

http://class.ece.iastate.edu/cpre288
Example: Initialize Stepper Motor

```c
/// Initialize PORTB to control the stepper motor
void stepper_init(void) {
    SYSCTL_RCGCGPIO_R |= 0b000010; // Enable Port B clock
    GPIO_PORTB_DIR_R |= 0xF0; // Setting PB4-PB7 to output
    GPIO_PORTB_DEN_R |= 0xF0; // Enabling PB4-PB7
    GPIO_PORTB_DATA_R &= 0x8F; // Init PB7-PB4 = 0b1000
    timer_waitMillis(2);
    GPIO_PORTB_DATA_R &= 0x0F; // Clear PB4-PB7
}
```

Stepper motor port connection

- Port B, pins 4-7
- Motor connected programmed as Output
- Wait for 2 ms for stepper model to settle
Software writes Configuration Register

Software writes Data Register (GPIO)

Hardware writes Data Register (Alternative Function)

Out side of chip world
Figure 2.17   Function block diagram of Analog/Digital GPIO control.

Software writes (GPIO)

Hardware writes (Alternative Function)
• GPIOAFSEL: For each wire select GPIO or Alternative Function
• GPIOPCTL: For each wire choose which Alternative Function

Figure 2.19   The illustration for GPIOAFSEL and GPIOPCTL registers.

Table 2.6
PMC6 Bit Encoding
(1, 2, 3, ... 9, 14)
0001, 0010, 0011,
... 1001, 1110

Table 2.6
PMC0 Bit Encoding
(1, 2, 3, ... 9, 14)
0001, 0010, 0011,
... 1001, 1110
### Table 2.6  GPIO Pins and Alternate Functions

<table>
<thead>
<tr>
<th>I/O</th>
<th>Pin</th>
<th>Analog Function</th>
<th>Digital Functions (GPIOPCTL PMCx Bit Field Encoding)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>PA0</td>
<td>17</td>
<td>-</td>
<td>U0RX</td>
</tr>
<tr>
<td>PA1</td>
<td>18</td>
<td>-</td>
<td>U0TX</td>
</tr>
<tr>
<td>PA2</td>
<td>19</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>PA3</td>
<td>20</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>PA4</td>
<td>21</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>PA5</td>
<td>22</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>PA6</td>
<td>23</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>PA7</td>
<td>24</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>PB0</td>
<td>45</td>
<td>USB0ID</td>
<td>U1RX</td>
</tr>
<tr>
<td>PB1</td>
<td>46</td>
<td>USB0VBUS</td>
<td>U1TX</td>
</tr>
<tr>
<td>PB2</td>
<td>47</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>PB3</td>
<td>48</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>PB4</td>
<td>58</td>
<td>AIN10</td>
<td>-</td>
</tr>
<tr>
<td>PB5</td>
<td>57</td>
<td>AIN11</td>
<td>-</td>
</tr>
<tr>
<td>PB6</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>PB7</td>
<td>4</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>PC0</td>
<td>52</td>
<td>-</td>
<td>TCK</td>
</tr>
<tr>
<td>PC1</td>
<td>51</td>
<td>-</td>
<td>TMS</td>
</tr>
<tr>
<td>PC2</td>
<td>50</td>
<td>-</td>
<td>TDI</td>
</tr>
<tr>
<td>PC3</td>
<td>49</td>
<td>-</td>
<td>TDO</td>
</tr>
<tr>
<td>PC4</td>
<td>16</td>
<td>C1-</td>
<td>U4RX</td>
</tr>
<tr>
<td>PC5</td>
<td>15</td>
<td>C1+</td>
<td>U4TX</td>
</tr>
</tbody>
</table>
• What values do GPIOAFSEL and GPIOPCTL need to be given to configure Port B to use UART1 TX?
Interrupts
How does a program executing within CPU communicate with the keyboard?

```c
Get_user_ID(char *name);
```
Memory mapped I/O: *Registers within Keyboard appear to be in memory.*

```c
Get_user_ID(char *name);

char * KBDR;
KBDR = (char *) 0xff00;
while ((*(name++) = *KBDR) != newline);
```
Memory Mapped Device I/O

- We need a control mechanism for the keyboard to tell us when fresh data is available in KBDR?
- Keyboard Control Register (KBCR)

```c
char * KBDR;
KBDR = 0xff00;
while ( (*(name++) = *KBDR) != newline );
```

How quickly does the while loop iterate? 10-100 µs?
How quickly do we type? 30-100 chars/minute?
Memory mapped I/O: *Registers within Keyboard appear to be in memory.*

```c
Get_user_ID(char *name);

char * KBDR;
KBDR = 0xff00;
while (*((name++) = *KBDR) != newline);
```

**KBDR- Data Register**

**KBCR- Control Register**
struct KBCR {
    // Big Endian
    unsigned int model : 4;
    unsigned : 1;
    unsigned int KBERROR : 1;
    unsigned int CAPLOCK : 1;
    unsigned int READY : 1;
} KBCR;
• **Maintain multiple views:** byte or bit structure.

```
union KBCR_U {
    struct KBCR KBCR;
    uint8_t KBCR_Aggregate;
} KBCR_U;
```

```
KBCR_U.KBCR_Aggregate = 0xc1;
KBCR_U.KBCR.READY = 1;
```
In one attempt, we may not even get one character?

```c
char *KBDR;
char *KBCR;

KBDR = (char *) 0xff00;
KBCR = (uint8_t) 0xff01;

if (KBCR.READY){
    if((*(name++) = *KBDR) == NEWLINE) break;
}
```
Memory Mapped I/O- Polling

- Without any device (keyboard) specific instructions, we can talk to the device/sensor.
- Even the future devices whose interface we do not know yet can be memory-mapped!

```c
char *KBDR;
char *KBCR;

KBDR = 0xff00;
KBCR = 0xff01;

while (!KBCR.READY);   //polling loop
//guaranteed fresh data
if((*(name++) = *KBDR) == NEWLINE)
    return;
```
if (KBCR.READY){  };

struct KBCR *pKBCR;

pKBCR->CAPLOCK = 0;

struct student student_records[100];
char *KBDR;
char *KBCR;

KBDR = 0xff00;
KBCR = 0xff01;

while (!(KBCR & 0x1));
// guaranteed fresh data
if((*(name++) = *KBDR) == NEWLINE) return;