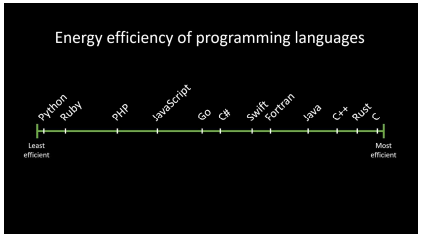
Module 1A
Embedded C Programming

- The slides are prepared using
  - Essential C (Summary of the basic features of the C language)
    - http://cslibrary.stanford.edu/101/EssentialC.pdf
  - Embedded C Tutorial
    - https://community.nxp.com/legacyfs/online/archiveatt/Lecture_3_-_C_Intro.pdf
  - The textbook by Mazidi etal and its publicly available slides
    - http://www.microdigitaled.com/ARM/Freescale_ARM_books.htm
  - C programming tutorial from
    - https://www.eng.auburn.edu/~nelson/courses/elec3040_3050/
  - Valvano's E-book on Embedded C programming
    - http://users.ece.utexas.edu/~valvano/embed/toc1.htm
  - TutorialsPoint
    - https://www.tutorialspoint.com/cprogramming

Why C programming?

- ☺ It is easier and less time consuming to write in C than Assembly.
- ☺ C is easier to modify and update.
- ☺ You can use code available in function libraries.
- ☺ C code is portable to other microcontrollers with little or no modification.
- ☹ Generally generates larger code
- ☹ Programmer has less control and less ability to directly interact with the hardware



☺ [1] Pereira, R. et al. (2017) 'Energy efficiency across programming languages: how do energy, time, and memory relate'. doi:10.1145/3136014.3136031.

☺

Basic C Program Structure

```
#include "STM32L1xx.h"        /* I/O port/register names/addresses for the STM32L1xx microcontrollers */

/* Global variables – accessible by all functions */
int count, bob;               //global (static) variables – placed in RAM

/* Function definitions*/
int function1(char x) {       //parameter x passed to the function, function returns an integer value
 int i,j;                     //local (automatic) variables – allocated to stack or registers
 -- instructions to implement the function
}

/* Main program */
void main(void) {
 unsigned char sw1;           //local (automatic) variable (stack or registers)
 int k;                       //local (automatic) variable (stack or registers)
/* Initialization section */
 -- instructions to initialize  variables, I/O ports, devices, function registers
/* Endless loop */
 while (1) {                  //Can also use:  for(;;) {
  -- instructions to be repeated
 } /* repeat forever */
}
```

Declare local variables

Initialize variables/devices

Body of the program

ANSI C (ISO C89) Integer Data Types and Their Ranges

Data type	Size	Range Min	Range Max
char	1 byte	-128	127
unsigned char	1 byte	0	255
short int	2 bytes	-32,768	32,767
unsigned short int	2 bytes	0	65,535
int	4 bytes	-2,147,483,648	2,147,483,647
unsigned int	4 bytes	0	4,294,967,295
long	4 bytes	-2,147,483,648	2,147,483,647
unsigned long	4 bytes	0 to 4,294,967,295	
long long	8 bytes	-9,223,372,036,854,775,808	9,223,372,036,854,775,807
unsigned long long	8 bytes	0	18,446,744,073,709,551,615

- Instead of defining the exact sizes of the integer types, C defines lower bounds. This makes it easier to implement C compilers on a wide range of hardware.
- Unfortunately, it occasionally leads to bugs where a program runs differently on a 16-bit-int machine than it runs on a 32-bit-int machine.

ISO C99 Integer Data Types and Their Ranges

Data type	Size	Range Min	Range Max
int8_t	1 byte	-128	127
uint8_t	1 byte	0 to	255
int16_t	2 bytes	-32,768	32,767
uint16_t	2 bytes	0	65,535
int32_t	4 bytes	-2,147,483,648	2,147,483,647
uint32_t	4 bytes	0	4,294,967,295
int64_t	8 bytes	-9,223,372,036,854,775,808	9,223,372,036,854,775,807
uint64_t	8 bytes	0	18,446,744,073,709,551,615

Constants/Literals

- Decimal is the default number format
```
int m,n; //16-bit signed numbers
m = 453; n = -25;
```
- Hexadecimal: preface value with 0x or 0X
```
m = 0xF312;
```
- Octal: preface value with zero (0)
```
m = 0453; n = -023;
```
- Don't use leading zeros on "decimal" values. They will be interpreted as octal.
- Character: character in single quotes, or ASCII value following "slash"
```
m = 'a'; //ASCII value 0x61
n = '\n';
//ASCII value 13 "return" character (escape characters) \)
```
- String (array) of characters:
  - unsigned char k[7];

```
strcpy(k,"hello\n");
//k[0]='h', k[1]='e', k[2]='l', k[3]='l', k[4]='o',
//k[5]=13 or '\n' (ASCII new line character),
//k[6]=0 or '\0' (null character - end of string)
```

```
'A' uppercase 'A' character
'\n' newline character
'\t' tab character
'\0' the "null" character --
integer value 0 (different from
the char digit '0')
'\012' the character with value
12 in octal, which is decimal 10
```

Variables

- A variable is an addressable storage location to information to be used by the program
- Each variable must be declared to indicate size and type of information to be stored, plus name to be used to reference the information
```
int x,y,z; //declares 3 variables of type "int"
char a,b; //declares 2 variables of type "char"
```
- Space for variables may be allocated in registers, RAM, or ROM/Flash (for constants)
- Variables can be automatic or static

Automatic Variables

- Declare within a function/procedure
- Variable is visible (has scope) only within that function
- Space for the variable is allocated on the system stack when the procedure is entered
- Deallocated, to be re-used, when the procedure is exited
- If only 1 or 2 variables, the compiler may allocate them to registers within that procedure, instead of allocating memory.
- Values are not retained between procedure calls

Automatic Variable Example

```
void delay () {
int i,j; //automatic variables - visible only within
delay()
for (i=0; i<100; i++) { //outer loop
for (j=0; j<20000; j++) { //inner loop
} //do nothing
}
}
```

Variables must be initialized each time the procedure is entered since values are not retained when the procedure is exited.

Static Variables

- Retained for use throughout the program in RAM

locations that are not reallocated during program execution.

- Declare either within or outside of a function
- If declared outside a function:
  - the variable is global in scope, i.e. known to all functions of the program
  - Use "normal" declarations.
  - Example: int count;
- If declared within a function:
  - insert key word static before the variable definition.
  - The variable is local in scope, i.e. known only within this function.

```
static unsigned char bob;
static int pressure[10];
```

Static Variable Example

```
unsigned char count;  //global variable is static – allocated a fixed RAM location
                      //count can be referenced by any function
void math_op () {
 int i;               //automatic variable – allocated space on stack when function entered
 static int j;        //static variable – allocated a fixed RAM location to maintain the value
 if (count == 0)      //test value of global variable count
     j = 0;           //initialize static variable j first time math_op() entered
 i = count;           //initialize automatic variable i each time math_op() entered
 j = j + i;           //change static variable j – value kept for next function call
}                     //return & deallocate space used by automatic variable i

void main(void) {
 count = 0;           //initialize global variable count
 while (1) {
  math_op();
  count++;            //increment global variable count
 }
}
```

Overflow

- Unlike assembly language programming, high level language programs do not provide indications when overflow occurs and the program just fails silently.
- If you use a `short int` to hold the number of seconds of a day, the second count will overflow from 32,767 to -32,768. Even if your program handles negative second count, the time will jump back to the day before.

Coercion

- If you write a statement with different operand data types for a binary operation, the compiler will convert the smaller data type to the bigger data type. This implicit data type is called **coercion**. For example. 'b' + 15.
- The compiler may or may not give you warning when coercion occurs.
- If the variable is signed and the data size is increased, the new bits are filled with the sign bit (most significant bit) of the original value.
- When you assign a larger data type to a smaller data type variable, the higher order bits will be truncated.

```
char ch;
int i;
i = 321;
ch = i; // truncation of an int value to fit in a char
// ch is now 65
```


Type Conversion (Typecasting)

```
{
int score;
...// suppose score gets set in the range [0,19]
score = (score / 20) * 100; // score/20 truncates to 0
```

- Unfortunately, score will almost always be set to 0 for this code because the integer division in the expression (score/20) will be 0 for every value of score less than 20.
- The fix is to force the quotient to be computed as a floating point number...

```
score = ((double)score / 20) * 100;
// OK -- floating point division from cast

score = (score / 20.0) * 100;
// OK -- floating point division from 20.0
}
```

Contd

- **int Constant Numbers in the source code such as 234 default to type int. They may be followed by an 'L' (upper or lower case) to designate that the constant should be a long such as 42L.**
- **An integer constant can be written with a leading 0x to indicate that it is expressed in hexadecimal; 0x10 is way of expressing the number 16.**
- **The integral types may be mixed together in arithmetic expressions since they are all basically just integers with variation in their width. For example, char and int can be combined in arithmetic expressions such as ('b' + 5).**
  - the compiler "promotes" the smaller type (char) to be the same size as the larger type (int) before combining the values. Promotions are determined at compile time based purely on the types of the values in the expressions. Promotions do not lose information -- they always convert from a type to compatible, larger type to avoid losing information.

Relational and Math Operators in C

- `==` Equal
- `!=` Not Equal
- `>` Greater Than
- `<` Less Than
- `>=` Greater or Equal
- `<=` Less or Equal

- `!` Boolean not (unary)
- `&&` Boolean and
- `||` Boolean or

- `+` Addition
- `-` Subtraction
- `/` Division
- `*` Multiplication
- `%` Remainder (mod)
- `++` increment
- `--` decrement

Pre- and Post-Variations of ++ and --

```
int i = 42;
int j;
j = (i++ + 10);
// i is now 43
// j is now 52 (NOT 53)

j = (++i + 10)
// i is now 44
// j is now 54
```

Example

```
unsigned char value, temp1, temp2, temp3;
value = 0xF5; // 245 base 10
temp1 = (value%10)+0x30;
temp2 = value/10;
temp3 = temp2/10+0x30;
temp2 = temp2%10+0x30;

printf(temp3"\n"); // MSB
printf(temp2"\n");
printf(temp1"\n"); // LSB
```

Bit-wise Operators in C

A	B	AND (A & B)	OR (A | B)	EX-OR (A^B)	Invert ~B
0	0	0	0	0	1
0	1	0	1	1	0
1	0	0	1	1	1
1	1	1	1	0	0

Setting and Clearing (Masking) bits

- Anything ORed with a 1 results in a 1; anything ORed with a 0 results in no change.
- Anything ANDed with a 1 results in no change; anything ANDed with a 0 results in a zero.
- Anything EX-ORed with a 1 results in the complement; anything EX-ORed with a 0 results in no change.

Testing Bit with Bit-wise Operators in C

- When it is necessary to test a given bit to see if it is high or low, the unused bits are masked and then the remaining data is tested.

  Example:

```
while(1)
{
if (var1 & 0x20) /* check bit 5 (6th bit) of var1 */
var2 = 0x55;
/* this statement is executed if bit 5 is a 1 */
else
var2 = 0xAA;
/* this statement is executed if bit 5 is a 0 */
}
```

- C does not have a distinct boolean type; int is used instead.
- The language treats integer 0 as false and all non-zero values as true.

Bit-wise Shift Operation in C

Operation	Symbol	Format of Shift Operation
Shift Right	>>	data >> number of bit-positions to be shifted right
Shift Left	<<	data << number of bit-positions to be shifted left

Compound Operators

Statement	Its equivalent using compound operators
`a = a + 6;`	`a += 6;`
`a = a - 23;`	`a -= 23;`
`y = y * z;`	`y *= z;`
`z = z / 25;`	`z /= 25;`
`w = w | 0x20;`	`w |= 0x20;`
`v = v & mask;`	`v &= mask;`
`m = m ^ togBits;`	`m ^= togBits;`

Bit-wise Operations Using Compound Operators

- The majority of hardware access level code involves setting a bit or bits in a register, clearing a bit or bits in a register, toggling a bit or bits in a register, and monitoring the status bits. For the first three cases, the compound operators are very suitable.

Using Shift Operator to Generate Mask

- One way to ease the generation of the mask is to use the left shift operator. To generate a mask with bit n set to 1, use the expression: 1 << n
- If more bits are to be set in the mask, they can be "or"ed together. To generate a mask with bit n and bit m set to 1, use the expression:

```
(1 << n) | (1 << m)

register |= (1 << 6) | (1 << 1);
```


Setting the Value in a Multi-bit Field

```
register |= 1 << 30; // set bit 30
register &= ~(1 << 29); // clear bit 29
register |= 1 << 28; // set bit 28
register &= ~(7 << 28); // clear bits 28,29,30
register |= 5 << 28; // make three bits 101
register = register & ~(7 << 28) | (5 << 28);
```

Precedence	Operator	Associativity
1	~(Bitwise negation)	Right to left
2	<<(Bitwise LeftShift), >>(Bitwise RightShift)	Left to Right
3	& (Bitwise AND)	Left to Right
4	^(Bitwise XOR)	Left to Right
5	| (Bitwise Or)	Left to Right

Control Structures - If

- Both an if and an if-else are available in C. The <*expression*> can be any valid expression. The parentheses around the expression are required, even if it is just a single variable.

```
if (<expression>) <statement> // simple form with no {}'s or else

if (<expression>) { // simple form with {}'s to group statements
<statement>
<statement>
}

if (<expression>)
{
<statement>
}
else {
<statement>
}
```

```
Example:
if (x < y) {
min = x;
}
else {
min = y;
}
```

Example

```
if (num <= 10 || eli==7)
{
// do something
}
else if (num >= 20)
{
// do something
}
// as many 'else if's as you want
else
{
// default case
}
```

While Loops

- The while loop evaluates the test expression before every loop, so it can execute zero times if the condition is initially false. It requires the parenthesis like the if.

```
while (<expression>) {
<statement>
}
```

- The do-while loop is like a while, but with the test condition at the bottom of the loop.
- Always executed at least once

```
do {
<statement>
} while (<expression>)
```

For Loops

- The for loop in C is the most general looping construct. The loop header contains three parts: an initialization, a continuation condition, and an action.

```
for (<initialization>; <continuation>; <action>) {
<statement>
}
```

- The initialization is executed once before the body of the loop is entered. The loop continues to run as long as the continuation condition remains true (like a while). After every execution of the loop, the action is executed.
- The following example executes 10 times by counting 0..9.

```
for (i = 0; i < 10; i++) {
<statement>
}
```

Structures

- C has the usual facilities for grouping things together to form composite types – arrays and records (which are called "structures").

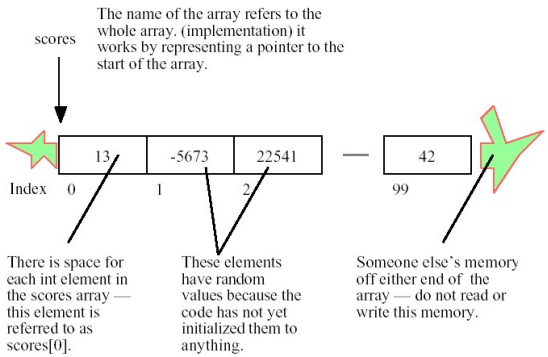
```
struct fraction {
int numerator;
int denominator;
}; // Don't forget the semicolon!
```

- C uses the period (.) to access the fields in a record. You can copy two records of the same type using a single assignment statement, however == does not work on structs.

```
struct fraction f1, f2; // declare two fractions
f1.numerator = 22;
f1.denominator = 7;
f2 = f1; // this copies over the whole struct
```

Arrays

```
int scores[100];  // array defined
scores[0] = 13; // set first element
scores[99] = 42; // set last element
```



More

- Initialization

```
int num[] = {1,2};
```

- Multidimensional Arrays

```
int board [10][10];
board[9][9] = 13;
board[0][0] = 13;
```

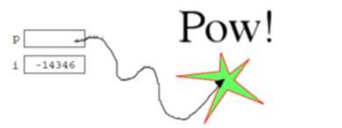
- Array of structures

```
struct fraction numbers[1000];
numbers[0].numerator = 22; // set the 0th
numbers[0].denominator = 7;
```

Pointers

- When using pointers, there are two entities to keep track of. The pointer and the memory it is pointing to, sometimes called the "pointee".
- There are three things which must be done for a pointer/pointee relationship to work...
  (1) The pointer must be declared and allocated
  (2) The pointee must be declared and allocated
  (3) The pointer (1) must be initialized so that it points to the pointee (2).

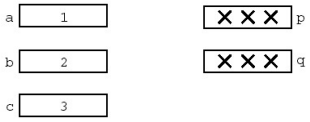
```
{
  int* p;
  *p = 13;    // NO NO NO p does not point to an int yet
              // this just overwrites a random area in memory
}
```

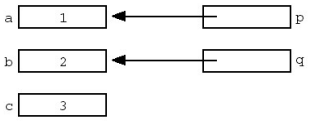


Pointer Example

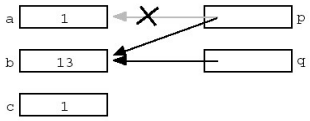
```
p = &a; // set p to refer to a
q = &b; // set q to refer to b
```

```
void PointerTest() {
// allocate three integers
   and two pointers
int a = 1;
int b = 2;
int c = 3;
int* p;
int* q;
```





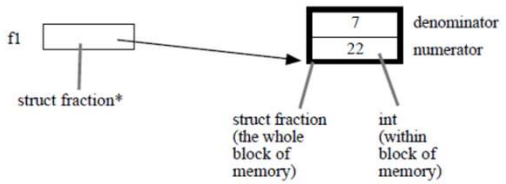

```
c = *p;
p =q;
*p = 13;
```




```
}
```

Pointer to Structures

```
struct fraction* f1;
```



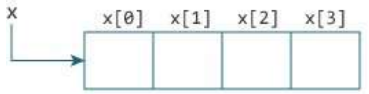
Expression	Type
f1	struct fraction*
*f1	struct fraction
(*f1).numerator	int

- There's an alternate, more readable syntax available for dereferencing a pointer to a struct.
- A "->" at the right of the pointer can access any of the fields in the struct. So the reference to the numerator field could be written f1->numerator.

Pointers

```
void NumPtrExample() {
int num;
int* numPtr; numPtr is a pointer to int
num = 42;
numPtr = &num;
// Compute a reference to "num", and store it in numPtr
// At this point, memory looks like drawing below
}
```

Pointers and Arrays



- There is a difference of 4 bytes between two consecutive elements of array x. It is because the size of int is 4 bytes (on our compiler).
- Notice that, the address of &x[0] and x is the same. It's because the variable name x points to the first element of the array.
- x[0] is equivalent to *x.
- &x[j] is equivalent to x+j and x[j] is equivalent to *(x+j).

Example

```
#include <stdio.h>
int main() {
  int x[5] = {1, 2, 3, 4, 5};
  int* ptr;
  // ptr is assigned the address of the third element
  ptr = &x[2];
  printf("*ptr = %d \n", *ptr);   // 3
  printf("*(ptr+1) = %d \n", *(ptr+1)); // 4
  printf("*(ptr-1) = %d", *(ptr-1));  // 2
  return 0;
}
When you run the program, the output will be:
*ptr = 3
*(ptr+1) = 4
*(ptr-1) = 2
```

Example

```
{
char string[1000]; // string is a local 1000 char array
int len;
strcpy(string, "binky");
len = strlen(string);
/*
 Reverse the chars in the string:
 i starts at the beginning and goes up
 j starts at the end and goes down
 i/j exchange their chars as they go until they meet
*/
int i, j;
char temp;
for (i = 0, j = len - 1; i < j; i++, j--) {
temp = string[i];
string[i] = string[j];
string[j] = temp;
}
// at this point the local string should be "yknib"
}
```

Pointer to a Pointer

```
int main () {
   int  var;
   int  *ptr;
   int  **pptr;
   var = 3000;
   ptr = &var;

   pptr = &ptr;
   printf("Value of var = %d\n", var );
   printf("Value available at *ptr = %d\n", *ptr );
   printf("Value available at **pptr = %d\n", **pptr);

   return 0;
}
When the above code is compiled and executed, it produces
the following result –
Value of var = 3000
Value available at *ptr = 3000
Value available at **pptr = 3000
```

Switch

```
switch (menu) /* select the type of calculation */
      {
      case 1: total = numb1 + numb2;
     calType = '+';             /* assign a       char to
  symbolise calculation type */
         break;
      case 2: total = numb1 - numb2;
        calType = '-';
         break;
      case 3: total = numb1 * numb2;
      calType = '*';
         break;
      case 4: total = numb1 / numb2;
        calType = '/';
         break;
      default: printf("Invalid option selected\n");
      }
```

Ternary Operator

```
<expression1> ? <expression2> : <expression3>
```

- This is an expression, not a statement, so it represents a value.
- **An expression is something that returns a value, whereas a statement does not**.
- The operator works by evaluating expression1. If it is true (non-zero), it evaluates and returns expression2
- Otherwise, it evaluates and returns expression3.
- Example:

```
min = (x < y) ? x : y;
```

C Functions

- Functions partition large programs into a set

of smaller tasks

- Helps manage program complexity
- Smaller tasks are easier to design and debug
- Functions can often be reused instead of starting over
- Can use "libraries" of functions developed by 3rd parties as opposed to designing your own
- A function is "called" by another program to perform a task
  - The function may return a result to the caller
  - One or more arguments may be passed to the function/procedure

Function Definition

Type of value to be returned to the caller*

Parameters passed by the caller

```
int math_func (int k; int n)
{
  int j;              //local variable
  j = n + k - 5;      //function body
  return(j);          //return the result
}
```

* If no return value, specify "void"

Function Arguments

- Calling program can pass information to a function in two ways
- By value: pass a constant or a variable value
  - function can use, but not modify the value
- By reference: pass the address of the variable
  - function can both read and update the variable
- Values/addresses are typically passed to the function by pushing them onto the system stack
- Function retrieves the information from the stack

Example – Pass by Value

```
/* Function to calculate x² */
int square ( int x ) {  //passed value is type int, return an int value
   int y;               //local variable – scope limited to square
   y = x * x;           //use the passed value
   return(y);           //return the result
}

void main {
   int k,n;            //local variables – scope limited to main
   n = 5;
  k = square(n);       //pass value of n, assign n-squared to k
  n = square(5);       // pass value 5, assign 5-squared to n
}
```

Example

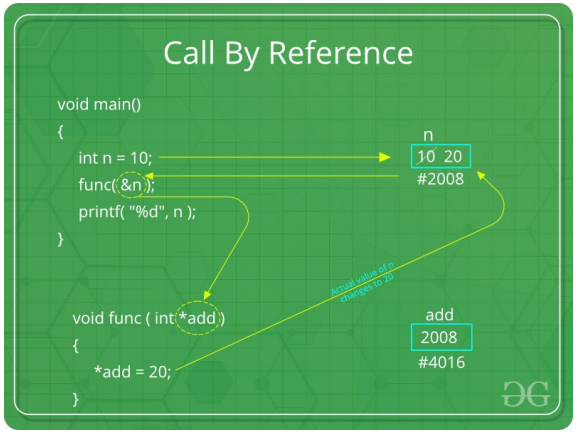
```
void func(int a, int b)
{
    a += b;
    printf("In func, a = %d b = %d\n",
a, b);
}
int main(void)
{
    int x = 5, y = 7;

    // Passing parameters
    func(x, y);
    printf("In main, x = %d y = %d\n",
x, y);
    return 0;
}
Output:
In func, a = 12 b = 7
In main, x = 5 y = 7
```

Example – Pass by Reference

```
/* Function to calculate x² */
void square ( int x, int *y ) {  //value of x, address of y
  *y = x * x;                    //write result to location whose address is y
}

void main {
  int k,n;                       //local variables – scope limited to main
  n = 5;
  square(n, &k);   //calculate n-squared and put result in k
  square(5, &n);   // calculate 5-squared and put result in n
}
```

In the above, *main* tells *square* the location of its local variable, so that *square* can write the result to that variable.

Example



Find Maximum Value in An Array

```
#include <stdio.h>
#include <conio.h>
max(int [],int);
void main()
{
int a[]={10,5,45,12,19};
int n=5,m;
clrscr();
m=max(a,n);
printf("\n MAXIMUM NUMBER
IS %d",m);
getch();
}
```

```
int max(int x[],int k)
{
int t,i;
t=x[0];
for(i=1;i<k;i++)
{
if(x[i]>t)
t=x[i];
}
return(t);
}
```

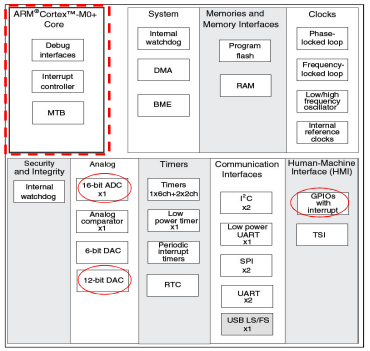
Module 1B - Cortex-M0+ CPU Core

Overview

- Cortex-M0+ Processor Core Registers
- Memory System and Addressing
- Thumb Instruction Set
- References
  - DDI0419C Architecture ARMv6-M Reference Manual

Microcontroller vs. Microprocessor

- Both have a CPU core to execute instructions
- Microcontroller has peripherals for embedded interfacing and control
  - Analog
  - Non-logic level signals
  - Timing
  - Clock generators
  - Communications
    - » point to point
    - » network
  - Reliability and safety



Architectures and Memory Speed

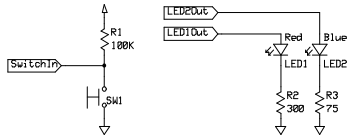
- Load/Store Architecture
  - Developed to simplify CPU design and improve performance
    - » *Memory wall*: CPUs keep getting faster than memory
    - » Memory accesses slow down CPU, limit compiler optimizations
    - » Change instruction set to make most instructions *independent* of memory
  - Data processing instructions can access registers only
    1. Load data into the registers
    2. Process the data
    3. Store results back into memory
  - More effective when more registers are available
- Register/Memory Architecture
  - Data processing instructions can access memory or registers
  - Memory wall is not very high at lower CPU speeds (e.g. under 50 MHz)

Module 2 - General Purpose I/O

Overview

- How do we make a program light up LEDs in response to a switch?
- GPIO
  - Basic Concepts
  - Port Circuitry
  - Control Registers
  - Accessing Hardware Registers in C
  - Clocking and Muxing
- Circuit Interfacing

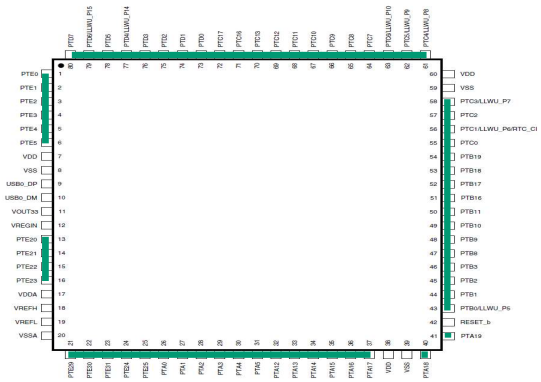
Basic Concepts



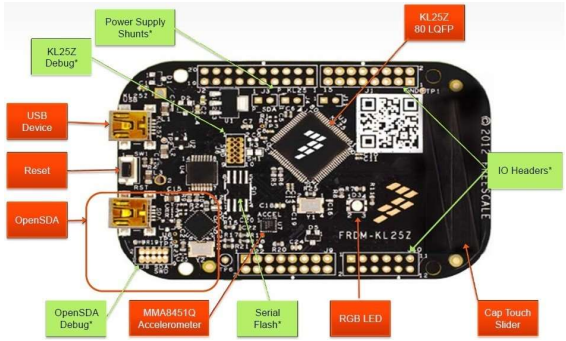
- Goal: light either LED1 or LED2 based on switch SW1 position
- GPIO = General-purpose input and output (digital)
  - Input: program can determine if input signal is a 1 or a 0
  - Output: program can set output to 1 or 0
- Can use this to interface with external devices
  - Input: switch
  - Output: LEDs

KL25Z GPIO Ports

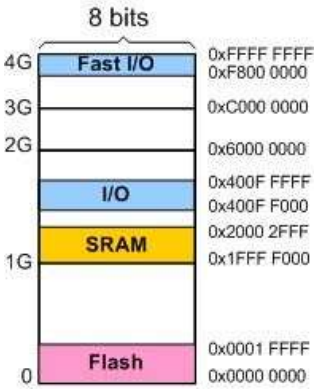
- Port A (PTA) through Port E (PTE)
- Not all port bits are available
- Quantity depends on package pin count

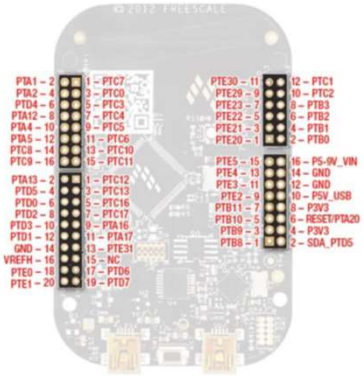


Freedom KL25Z

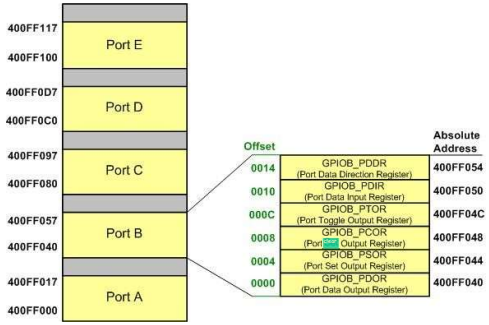


Memory Map

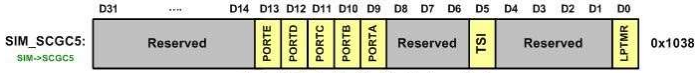




Ports



Clocking Logic



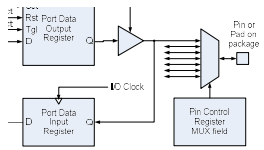
Note: 0: Clock disabled, 1: Clock enabled

Bit	Port
13	PORTE
12	PORTD
11	PORTC
10	PORTB
9	PORTA

- Need to enable clock to GPIO module
- By default, GPIO modules are disabled to save power
- Writing to an unclocked module triggers a hardware fault!
- Control register SIM_SCGC5 gates clocks to GPIO ports
- Enable clock to Port A

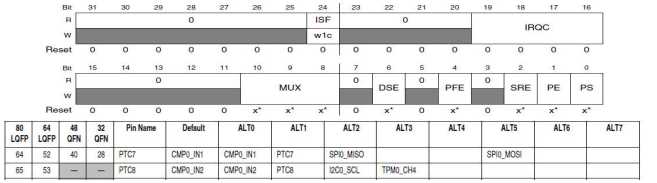
```
SIM->SCGC5 |= (1UL <<  9);
```

Connecting a GPIO Signal to a Pin



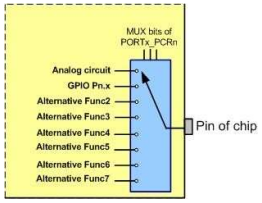
- Multiplexer used to increase configurability - what pin should be connected with internally?
- Each configurable pin has a Pin Control Register

Pin Control Register



80 LQFP	64 LQFP	48 QFN	32 QFN	Pin Name	Default	ALT0	ALT1	ALT2	ALT3	ALT4	ALT5	ALT6	ALT7
64	52	40	28	PTC7	CMP0_IN1	CMP0_IN1	PTC7	SPI0_MISO			SPI0_MOSI		
65	53	—	—	PTC8	CMP0_IN2	CMP0_IN2	PTC8	I2C0_SCL	TPM0_CH4				

- MUX field of PCR defines connections



Structure Declarations

- Like other elements of C programming, the structure must be declared before it can be used. The declaration specifies the tagname of the structure and the names and types of the individual members. The following example has three members: one 8-bit integer and two word pointers

```
struct theport{
unsigned char mask;    // defines which bits are active
unsigned long volatile *addr;  // pointer to its address
unsigned long volatile *ddr;}; // pointer to its direction reg
```

- The above declaration does not create any variables or allocate any space. Therefore to use a structure we must define a global or local variable of this type. The tagname (theport) along with the keyword struct can be used to define variables of this new data type:

```
struct theport PortA,PortB,PortE;
```

- The above line defines the three variables and allocates 9 bytes for each of variable. Because the pointers will be 32-bit aligned the compiler will skip three bytes between mask and addr, so each object will occupy 12 bytes. If you knew you needed just three copies of structures of this type, you could have defined them as

```
struct theport{
unsigned char mask;    // defines which bits are active
unsigned long volatile *addr;
unsigned long volatile *ddr;}PortA,PortB,PortE;
```

Continued

- Definitions like the above are hard to extend, so to improve code reuse we can use typedef to actually create a new data type (called port in the example below) that behaves syntactically like char int short etc.

```
struct theport{
   unsigned char mask;    // defines which bits are active
   unsigned long volatile *addr;  // address
   unsigned long volatile *ddr;}; // direction reg
typedef struct theport port_t;
port_t PortA,PortB,PortE;
```

- Once we have used typedef to create port_t, we don't need access to the name theport anymore. Consequently, some programmers use to following short-cut:

```
typedef struct {
   unsigned char mask;    // defines which bits are active
   unsigned long volatile *addr;        // address
   unsigned long volatile *ddr;}port_t; // direction reg
port_t PortA,PortB,PortE;
```

CMSIS C Support for PCR

- MKL25Z4.h defines PORT_Type structure with a PCR field (array of 32 integers)

```
/** PORT - Register Layout Typedef */
typedef struct {
  __IO uint32_t PCR[32]; /** Pin Control Register n, array offset: 0x0, array step: 0x4 */
  __O  uint32_t GPCLR;       /** Global Pin Control Low Register, offset: 0x80 */
  __O  uint32_t GPCHR;       /** Global Pin Control High Register, offset: 0x84 */
       uint8_t RESERVED_0[24];
  __IO uint32_t ISFR;        /** Interrupt Status Flag Register, offset: 0xA0 */
} PORT_Type;
```

CMSIS C Support for PCR

- Header file defines pointers to PORT_Type registers

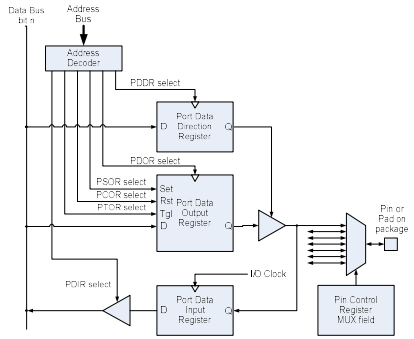
```
/* PORT - Peripheral instance base addresses */
/** Peripheral PORTA base address */
#define PORTA_BASE      (0x40049000u)
/** Peripheral PORTA base pointer */
#define PORTA       ((PORT_Type *)PORTA_BASE)
```

- Also defines macros and constants

```
#define PORT_PCR_MUX_MASK     0x700u
#define PORT_PCR_MUX_SHIFT    8
#define PORT_PCR_MUX(x)(((uint32_t)(((uint32_t)(x))<<PORT_PCR_MUX_SHIFT))&PORT_PCR_MUX_MASK)
```

GPIO Port Bit Circuitry in MCU

- Control
  - Direction
  - MUX
- Data
  - Output (different ways to access it)
  - Input



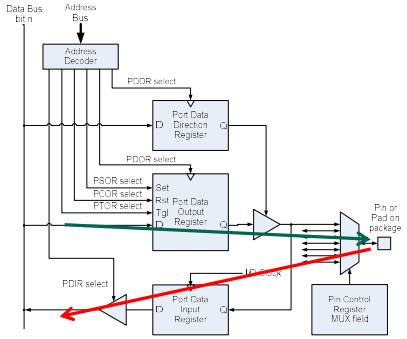
Control Registers

Absolute address (hex)	Register name	Width (in bits)
400F_F000	Port Data Output Register (GPIOA_PDOR)	32
400F_F004	Port Set Output Register (GPIOA_PSOR)	32
400F_F008	Port Clear Output Register (GPIOA_PCOR)	32
400F_F00C	Port Toggle Output Register (GPIOA_PTOR)	32
400F_F010	Port Data Input Register (GPIOA_PDIR)	32
400F_F014	Port Data Direction Register (GPIOA_PDDR)	32

- One set of control registers per port
- Each bit in a control register corresponds to a port bit

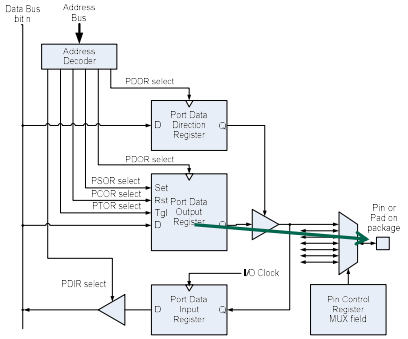
PDDR: Port Data Direction

- Each bit can be configured differently
- Input: 0
- Output: 1
- Reset clears port bit direction to 0



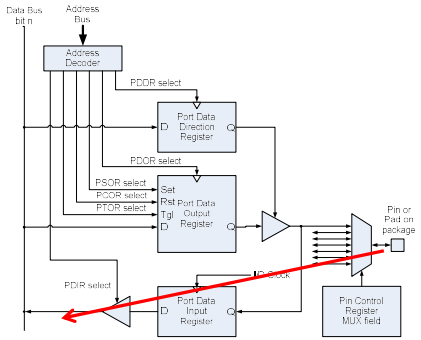
Writing Output Port Data

- Direct: write value to PDOR
- Toggle: write 1 to PTOR
- Clear (to 0): Write 1 to PCOR
- Set (to 1): write 1 to PSOR

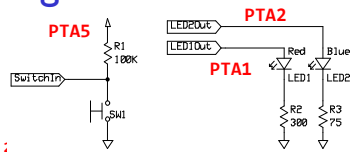


Reading Input Port Data

- Read from PDIR
- Corresponding bit holds value which was read



Pseudocode for Program




```
// Make PTA1 and PTA2 outputs
set bits 1 and 2 of GPIOA_PDDR
// Make PTA5 input
clear bit 5 of GPIOA_PDDR
// Initialize the output data values: LED 1
clear bit 1, set bit 2 of GPIOA_PDOR
// read switch, light LED accordingly
do forever {
        if bit 5 of GPIOA_PDIR is 1 {
                // switch is not pressed, then light LED 2
                set bit 2 of GPIOA_PDOR
                clear bit 1 of GPIO_PDOR
        } else {
                // switch is pressed, so light LED 1
                set bit 1 of GPIOA_PDOR
                clear bit 2 of GPIO_PDOR
        }
}
```

CMSIS - Accessing Hardware Registers in C

- Header file MKL25Z4.h defines C data structure types to represent hardware registers in MCU with CMSIS-Core hardware abstraction layer

```
#define __I volatile const
#define __O volatile
#define __IO volatile
/** GPIO - Register Layout Typedef */
typedef struct {
  __IO uint32_t PDOR;     /**< Port Data Output Register, offset: 0x0 */
  __O  uint32_t PSOR;     /**< Port Set Output Register, offset: 0x4 */
  __O  uint32_t PCOR;     /**< Port Clear Output Register, offset: 0x8 */
  __O  uint32_t PTOR;     /**< Port Toggle Output Register, offset: 0xC */
  __I  uint32_t PDIR;     /**< Port Data Input Register, offset: 0x10 */
  __IO uint32_t PDDR;     /**< Port Data Direction Register, offset: 0x14 */
} GPIO_Type;
```

Accessing Hardware Registers in C (2)

- Header file MKL25Z4.h declares pointers to the registers

```
/* GPIO - Peripheral instance base addresses */
/** Peripheral PTA base address */
#define PTA_BASE    (0x400FF000u)
/** Peripheral PTA base pointer */
#define PTA                 ((GPIO_Type *)PTA_BASE)

PTA->PDOR = …
```

Coding Style and Bit Access

- Easy to make mistakes dealing with literal binary and hexadecimal values
  - "To set bits 13 and 19, use 0000 0000 0000 1000 0010 0000 0000 0000 or 0x00082000"
- Make the literal value from shifted bit positions

```
n = (1UL << 19) | (1UL << 13);
```

- Define names for bit positions

```
#define GREEN_LED_POS (19)
#define YELLOW_LED_POS (13)
n = (1UL << GREEN_LED_POS) | (1UL << YELLOW_LED_POS);
```

- Create macro to do shifting to create mask

```
#define MASK(x) (1UL << (x))
n = MASK(GREEN_LED_POS) | MASK(YELLOW_LED_POS);
```

Using Masks

- Set in n all the bits which are one in mask, leaving others unchanged

```
n |= MASK(foo);
```

- Clear in n all the bits which are zero in mask, leaving others unchanged

```
n &= ~MASK(foo);
```

- Testing a bit value in register

```
if ( n & MASK(foo) == 0) ...
if ( n & MASK(foo) == 1) ...
```

Resulting C Code for Clock Control and Mux

```
// Enable Clock to Port A
SIM->SCGC5 |= SIM_SCGC5_PORTA_MASK;

// Make 3 pins GPIO
PORTA->PCR[LED1_POS] &= ~PORT_PCR_MUX_MASK;
PORTA->PCR[LED1_POS] |= PORT_PCR_MUX(1);
PORTA->PCR[LED2_POS] &= ~PORT_PCR_MUX_MASK;
PORTA->PCR[LED2_POS] |= PORT_PCR_MUX(1);
PORTA->PCR[SW1_POS] &= ~PORT_PCR_MUX_MASK;
PORTA->PCR[SW1_POS] |= PORT_PCR_MUX(1);
```

C Code

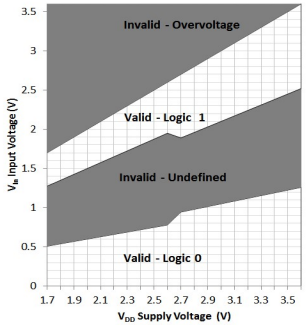
```
#define LED1_POS (1)
#define LED2_POS (2)
#define SW1_POS (5)
#define MASK(x) (1UL << (x))

PTA->PDDR |= MASK(LED1_POS) | MASK (LED2_POS); // set LED bits to outputs
PTA->PDDR &= ~MASK(SW1_POS); // clear Switch bit to input

PTA->PDOR = MASK(LED1_POS);  // turn on LED1, turn off LED2

while (1) {
       if (PTA->PDIR & MASK(SW1_POS)) {
         // switch is not pressed, then light LED 2
         PTA->PDOR = MASK(LED2_POS);
       } else {
         // switch is pressed, so light LED 1
         PTA->PDOR = MASK(LED1_POS);
       }
}
```

Inputs and Outputs, Ones and Zeros, Voltages and Currents

INTERFACING

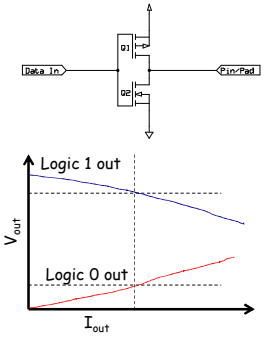
Inputs: What's a One? A Zero?

- Input signal's value is determined by voltage
- Input threshold voltages depend on supply voltage V_{DD}
- Exceeding V_{DD} or GND may damage chip



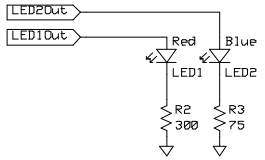
Outputs: What's a One? A Zero?

- Nominal output voltages
  - 1: V_{DD}-0.5 V to V_{DD}
  - 0: 0 to 0.5 V
- Note: Output voltage depends on current drawn by load on pin
  - Need to consider source-to-drain resistance in the transistor
  - Above values only specified when current < 5 mA (18 mA for high-drive pads) and V_{DD} > 2.7 V
- As with many other low-power ARM chips, the KL25Z can only provide (source) or sink a very limited amount of current: up to 5 mA per pin and no more than 100mA across all pins at a time
- If you source or sink more current than 5mA continuously or 20mA instantaneously, you will damage the board.



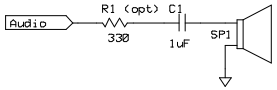
Output Example: Driving LEDs

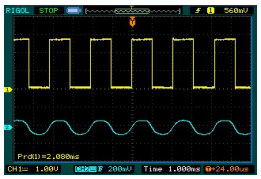
- Need to limit current to a value which is safe for both LED and MCU port driver
- Use current-limiting resistor
  - R = $(V_{DD} - V_{LED})/I_{LED}$
- Set I_{LED} = 4 mA
- V_{LED} depends on type of LED (mainly color)
  - Red: ~1.8V
  - Blue: ~2.7 V
- Solve for R given VDD = ~3.0 V
  - Red: 300 Ω
  - Blue: 75 Ω
- Demonstration code in Basic Light Switching Example



Output Example: Driving a Speaker

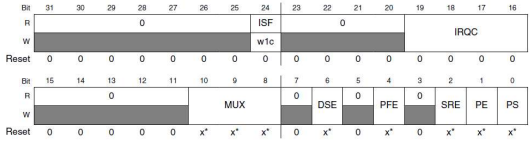
- Create a square wave with a GPIO output
- Use capacitor to block DC value
- Use resistor to reduce volume if needed
- Write to port toggle output register (PTOR) to simplify code





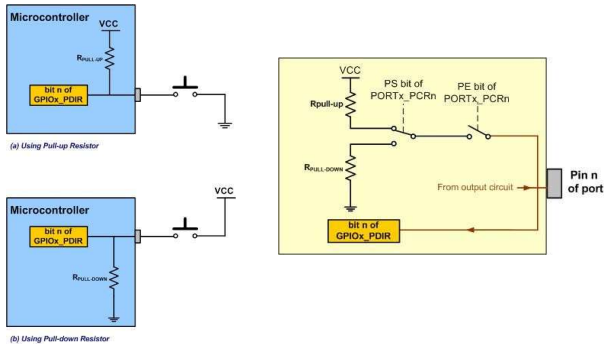
```
void Beep(void) {
   unsigned int period=20000;
   while (1) {
      PTC->PTOR = MASK(SPKR_POS);
      Delay(period/2);
   }
}
```

Additional Configuration in PCR



- Pull-up and pull-down resistors
  - Used to ensure input signal voltage is pulled to correct value when high-impedance
  - PE: Pull Enable. 1 enables the pull resistor
  - PS: Pull Select. 1 pulls up, 0 pulls down.
- High current drive strength
  - DSE: Set to 1 to drive more current (e.g. 18 mA vs. 5 mA @ > 2.7 V, or 6 mA vs. 1.5 mA @ <2.7 V)
  - Available on some pins - MCU dependent

Connecting External Switches



LCD Interfacing

Pin	Symbol	I/O	Description
1	VSS	--	Ground
2	VCC	--	+5V power supply
3	VEE	--	Power supply to control contrast
4	RS	I	RS = 0 to select command register, RS = 1 to select data register
5	R/W	I	R/W = 0 for write, R/W = 1 for read
6	E	I	Enable
7	DB0	I/O	The 8-bit data bus
8	DB1	I/O	The 8-bit data bus
9	DB2	I/O	The 8-bit data bus
10	DB3	I/O	The 8-bit data bus
11	DB4	I/O	The 4/8-bit data bus
12	DB5	I/O	The 4/8-bit data bus
13	DB6	I/O	The 4/8-bit data bus
14	DB7	I/O	The 4/8-bit data bus

Commands

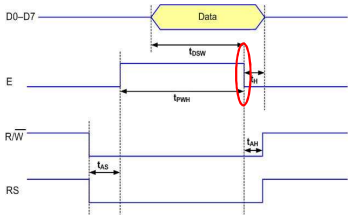
Code (Hex)	Command to LCD Instruction Register
1	Clear display screen
2	Return cursor home
6	Increment cursor (shift cursor to right)
F	Display on, cursor blinking
80	Force cursor to beginning of 1st line
C0	Force cursor to beginning of 2nd line
38	2 lines and 5x7 character (8-bit data, D0 to D7)
28	2 lines and 5x7 character (4-bit data, D4 to D7)

Operation

- **VCC, VSS, and VEE:** While VCC and VSS provide +5V power supply and ground, respectively, VEE is used for controlling the LCD contrast.
- **RS (Register Select):** There are two registers inside the LCD and the RS pin is used for their selection as follows. If RS = 0, the instruction command code register is selected, allowing the user to send a command such as clear display, cursor at home, and so on (or query the busy status bit of the controller). If RS =1, the data register is selected, allowing the user to send data to be displayed on the LCD (or to retrieve data from the LCD controller).
- **R/W (Read/Write):** R/W input allows the user to write information into the LCD controller or read information from it. R/W = 1 when reading and R/W = 0 when writing.
- **E (Enable):** The enable pin is used by the LCD to latch information presented to its data pins. When data is supplied to data pins, a pulse (Low-to-High-to-Low) must be applied to this pin in order for the LCD to latch in the data present at the data pins. This pulse must be a minimum of 230 ns wide, according to Hitachi datasheet.
- **D0–D7:** The 8-bit data pins are used to send information to the LCD or read the contents of the LCD's internal registers.

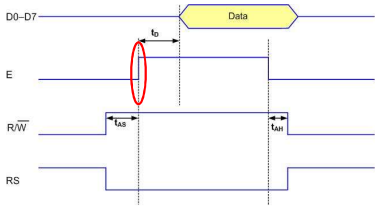
Timing

- Write Timing



t_{PWH} = Enable pulse width = 230 ns (minimum)
t_{DSW} = Data setup time = 80 ns (minimum)
t_H = Data hold time = 10 ns (minimum)
t_{AS} = Setup time prior to E (going high) for both RS and R/W = 40 ns (minimum)
t_{AH} = Hold time after E has come down for both RS and R/W = 10 ns (minimum)

- Read Timing

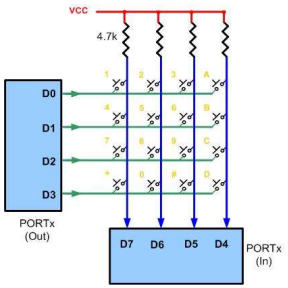


t_D = Data output delay time
t_{AS} = Setup time prior to E (going high) for both RS and R/W = 40 ns (minimum)
t_{AH} = Hold time after E has come down for both RS and R/W = 10 ns (minimum)
Note: Read requires an L-to-H pulse for the E pin.

LCD Operation Contd

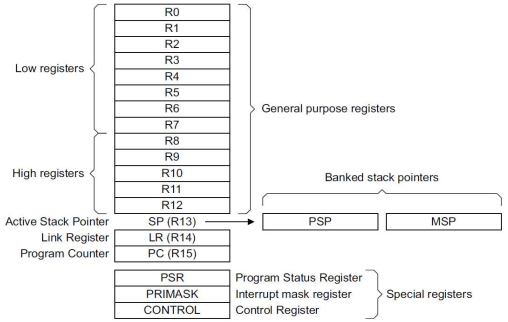
- We can monitor the busy flag and issue data when it is ready.
- To check the busy flag, we must read the command register (R/W = 1, RS = 0).
- The busy flag is the D7 bit of that register.
- Therefore, if R/W = 1, RS = 0.
- When D7 = 1 (busy flag = 1), the LCD is busy taking care of internal operations and will not accept any new information. When D7 = 0, the LCD is ready to receive new information.

Keypad



- To detect the key pressed, the microprocessor drives all rows low.
- Then, it reads the columns.
- If the data read from the columns is D7–D4 = 1111, no key has been pressed and the process continues until a key press is detected.
- However, if one of the column bits has a zero, this means that a key was pressed.
- Starting from the top row, the microprocessor drives one row low at a time; then it reads the columns.
- If the data read is all 1s, no key in that row is pressed and the process is moved to the next row.
- This process continues until a row is identified with a zero in one of the columns.

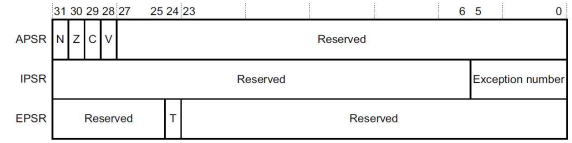
ARM Processor Core Registers



ARM Processor Core Registers (32 bits each)

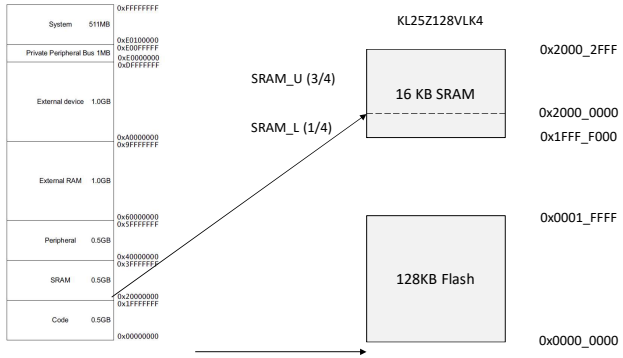
- R0-R12 - General purpose registers for data processing
- SP - Stack pointer (R13)
  - Can refer to one of two SPs
    - » Main Stack Pointer (MSP)
    - » Process Stack Pointer (PSP)
  - Uses MSP initially, and whenever in Handler mode
  - When in Thread mode, can select either MSP or PSP using SPSEL flag in CONTROL register.
- LR - Link Register (R14)
  - Holds return address when called with Branch & Link instruction (B&L)
- PC - program counter (R15)

ARM Processor Core Registers



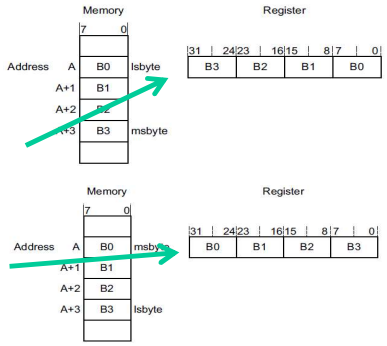
- Program Status Register (PSR) is three views of same register
  - Application PSR (APSR)
    - » Condition code flag bits Negative, Zero, oVerflow, Carry
  - Interrupt PSR (IPSR)
    - » Holds exception number of currently executing ISR
  - Execution PSR (EPSR)
    - » Thumb state

Memory Maps For Cortex M0+ and MCU



Endianness

- For a multi-byte value, in what order are the bytes stored?
- Little-Endian: Start with least-significant byte
- Big-Endian: Start with most-significant byte
- ARMv6-M is Little Endian
- Data: Depends on implementation, or from reset configuration
  - Kinetis processors are little-endian



ARM, Thumb and Thumb-2 Instructions

- ARM instructions optimized for resource-rich high-performance computing systems
  - Deeply pipelined processor, high clock rate, wide (e.g. 32-bit) memory bus
- Low-end embedded computing systems are different
  - Slower clock rates, shallow pipelines
  - Different cost factors – e.g. code size matters much more, bit and byte operations critical
- Modifications to ARM ISA to fit low-end embedded computing
  - 1995: Thumb instruction set
    - » 16-bit instructions
    - » Reduces memory requirements (and performance slightly)

Instruction Set

- Cortex-M0+ core implements ARMv6-M Thumb instructions
- Only uses Thumb instructions, always in Thumb state
  - Most instructions are 16 bits long, some are 32 bits
  - Most 16-bit instructions can only access low registers (R0-R7), but some can access high registers (R8-R15)
- Conditional execution only supported for 16-bit branch
- 32 bit address space
- Half-word aligned instructions
- See ARMv6-M Architecture Reference Manual for specifics per instruction (Section A.6.7)

Assembly Instructions

- Arithmetic and logic
  - Add, Subtract, Multiply, Divide, Shift, Rotate
- Data movement
  - Load, Store, Move
- Compare and branch
  - Compare, Test, If-then, Branch, compare and branch on zero
- Miscellaneous
  - Breakpoints, wait for events, interrupt enable/disable, data memory barrier, data synchronization barrier

Instruction Format: Labels

```
label   mnemonic operand1, operand2, operand3  ; comments
```

- Place marker, marking the memory address of the current instruction
- Used by branch instructions to implement **if-then** or **goto**
- Must be unique

Instruction Format: Mnemonic

```
label   mnemonic operand1, operand2, operand3  ; comments
```

- The name of the instruction
- Operation to be performed by processor core

Instruction Format: Operands

```
label   mnemonic operand1, operand2, operand3  ; comments
```

- Operands
  - Registers
  - Constants (called *immediate values*)
- Number of operands varies
  - No operands: DSB
  - One operand: BX LR
  - Two operands: CMP R1, R2
  - Three operands: ADD R1, R2, R3
  - Four operands: MLA R1, R2, R3, R4
- Normally
  - operand1 is the destination register, and operand2 and operand3 are source operands.
  - operand2 is usually a register, and the first source operand
  - operand3 may be a register, an immediate number, a register shifted to a constant number of bits, or a register plus an offset (used for memory access).

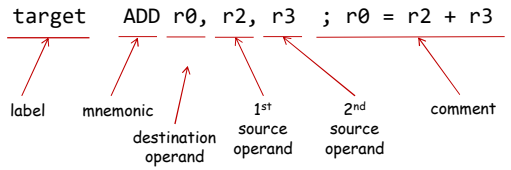
Instruction Format: Comments

```
label   mnemonic operand1, operand2, operand3  ; comments
```

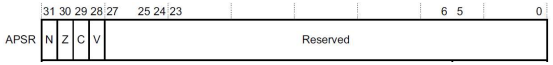
- Everything after the semicolon (;) is a comment
- Explain programmers' intentions or assumptions

ARM Instruction Format

```
label  mnemonic  operand1,  operand2,  operand3        ;
comments
```



Update Condition Codes in APSR?



- "S" suffix indicates the instruction updates APSR
  - ADD vs. ADDS
  - ADC vs. ADCS
  - SUB vs. SUBS
  - MOV vs. MOVS

Instruction Set Summary

Instruction Type	Instructions
Move	MOV
Load/Store	LDR, LDRB, LDRH, LDRSH, LDRSB, LDM, STR, STRB, STRH, STM
Add, Subtract, Multiply	ADD, ADDS, ADCS, ADR, SUB, SUBS, SBCS, RSBS, MULS
Compare	CMP, CMN
Logical	ANDS, EORS, ORRS, BICS, MVNS, TST
Shift and Rotate	LSLS, LSRS, ASRS, RORS
Stack	PUSH, POP
Conditional branch	IT, B, BL, B{cond}, BX, BLX
Extend	SXTH, SXTB, UXTH, UXTB
Reverse	REV, REV16, REVSH
Processor State	SVC, CPSID, CPSIE, SETEND, BKPT
No Operation	NOP
Hint	SEV, WFE, WFI, YIELD

Load/Store Register

- ARM is a load/store architecture, so must process data in registers (not memory)

- LDR: load register with word (32 bits) from memory
  - LDR <Rt>, source address

- STR: store register contents (32 bits) to memory
  - STR <Rt>, destination address

Load-Modify-Store

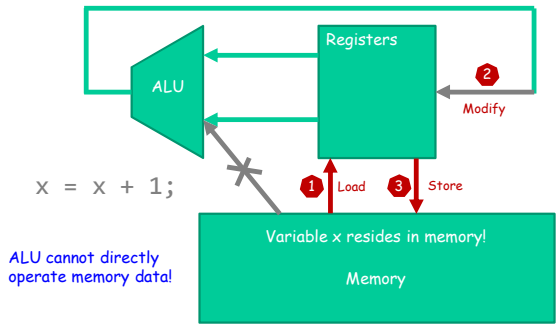
C statement

```
x = x + 1;
```

Assume variable X resides in memory and is a 32-bit integer

```
; Assume the memory address of x is stored in
r1

LDR r0, [r1]     ; load value of x from memory
ADD r0, r0, #1   ; x = x + 1
STR r0, [r1]     ; store x into memory
```

3 Steps: Load, Modify, Store



Example 1: Adding Two Integers

```
int x = 1;
int y = 2;
int z;
```

If values are in registers
- Value of x in r0
- Value of y in r1
- Value of z in r2

C Statement	Assembly Statement
`z = x + y;`	?

Adding Two Integers

```
int x = 1;
int y = 2;
int z;
```

If values are in registers
- Value of x in r0
- Value of y in r1
- Value of z in r2

C Statement	Assembly Statement
`z = x + y;`	`ADD r2, r1, r0`

Destination
Source Operand 2
Source Operand 1

Adding Two Integers

```
int x = 1;
int y = 2;
int z;
```

If addresses are in registers
- Address of x in r0
- Address of y in r1
- Address of z in r2

C Statement

```
z = x + y;
```

```
LDR r3, [r0] ; Read x
LDR r4, [r1] ; Read y
ADD r5, r3, r4
STR r5, [r2] ; Write z
```

Example 2: Set a Bit in C

```
a |= (1 << k)
```

or

```
a = a | (1 << k)
```

Example: `k = 5`

	a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0
a								
1 << k	0	0	1	0	0	0	0	0
a | (1 << k)			1					

The other bits should not be affected.

Set a Bit in Assembly

```
a |= (1 << 5)
```

```
Solution:
    MOVS r4, #1       ; r4 = 1
    LSLS r4, r4, #5   ; r4 = 1<<5
    ORRS r0, r0, r4   ; r0 = r0 | 1<<5
```

Example 3: 64 Bit Addition

```
start
  ; C = A + B
  ; Two 64-bit integers A (r1,r0) and B (r3, r2).
  ; Result C (r5, r4)
  ; A = 00000002FFFFFFFF
  ; B = 0000000400000001
  LDR  r0, =0xFFFFFFFF  ;  A's lower 32 bits
  LDR  r1, =0x00000002  ;  A's upper 32 bits
  LDR  r2, =0x00000001  ;  B's lower 32 bits
  LDR  r3, =0x00000004  ;  B's upper 32 bits

  ;  Add A and B
  ADDS r4, r2, r0  ; C[31..0] = A[31..0] +  B[31..0],  update Carry
  ADC  r5, r3, r1   ; C[64..32] = A[64..32] + B[64..32]  +  Carry

stop  B  stop
```

Module 3 - Cortex-M0+
Exceptions and Interrupts

Overview

- Exception and Interrupt Concepts
  - Entering an Exception Handler
  - Exiting an Exception Handler
- Cortex-M0+ Interrupts
  - Using Port Module and External Interrupts
- Timing Analysis
- Program Design with Interrupts
  - Sharing Data Safely Between ISRs and Other Threads

- Sources
  - Cortex M0+ Device Generic User Guide - DUI0662
  - Cortex M0+ Technical Reference Manual - DUI0484

EXCEPTION AND INTERRUPT CONCEPTS

Example System with Interrupt

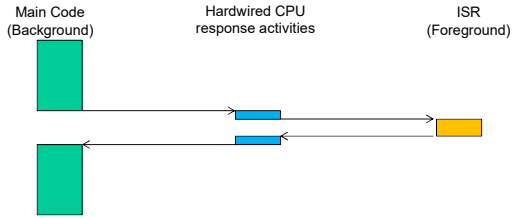
- Goal: Change color of RGB LED when switch is pressed
- Need to add external switch

How to Detect Switch is Pressed?

- Polling - use software to check it
  - Slow - need to explicitly check to see if switch is pressed
  - Wasteful of CPU time - the faster a response we need, the more often we need to check
  - Scales badly - difficult to build system with many activities which can respond quickly. Response time depends on all other processing.
- Interrupt - use special hardware in MCU to detect event, run specific code (*interrupt service routine* - ISR) in response
  - Efficient - code runs only when necessary
  - Fast - hardware mechanism
  - Scales well
    - » ISR response time doesn't depend on most other processing.
    - » Code modules can be developed independently

Interrupt or Exception Processing Sequence

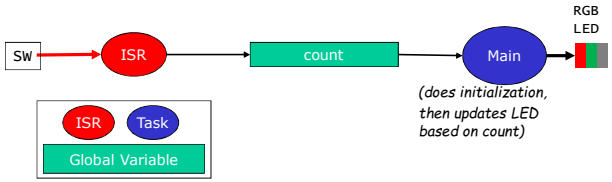
- Other code (background) is running
- Interrupt trigger occurs
- Processor does some hard-wired processing
- Processor executes ISR (foreground), including return-from-interrupt instruction at end
- Processor resumes other code



Interrupts

- Hardware-triggered asynchronous software routine
  - Triggered by hardware signal from peripheral or external device
  - Asynchronous - can happen anywhere in the program (unless interrupt is disabled)
  - Software routine - Interrupt service routine runs in response to interrupt
- Fundamental mechanism of microcontrollers
  - Provides efficient event-based processing rather than polling
  - Provides quick response to events regardless* of program state, complexity, location
  - Allows many multithreaded embedded systems to be responsive without an operating system (specifically task scheduler)

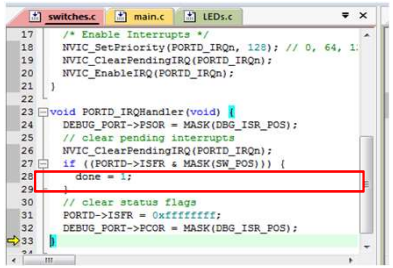
Example Program Requirements & Design



- Req1: When Switch SW is pressed, ISR will increment count variable
- Req2: Main code will light LEDs according to count value in binary sequence (Blue: 4, Green: 2, Red: 1)
- Req3: Main code will toggle its debug line each time it executes
- Req4: ISR will raise its debug line (and lower main's debug line) whenever it is executing

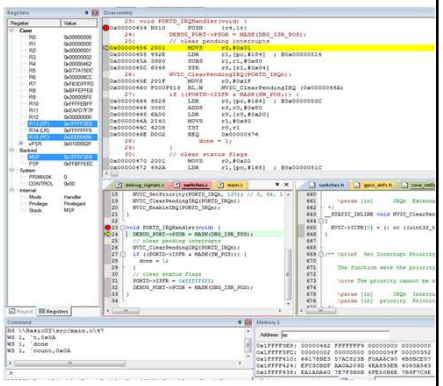
Example Exception Handler

- We will examine processor's response to exception in detail



Use Debugger for Detailed Processor View

- Can see registers, stack, source code, dissassembly (object code)
- Note: Compiler may generate code for function entry (see address 0x0000_0454)
- Place breakpoint on Handler function declaration line in source code (23), not at first line of function code (24)

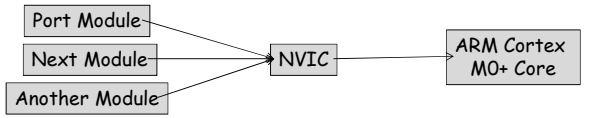


CORTEX-M0+ INTERRUPTS

Microcontroller Interrupts

- Types of interrupts
  - Hardware interrupts
    - ***Asynchronous***: not related to what code the processor is currently executing
    - Examples: interrupt is asserted, character is received on serial port, or ADC converter finishes conversion
  - Exceptions, Faults, software interrupts
    - ***Synchronous***: are the result of specific instructions executing
    - Examples: undefined instructions, overflow occurs for a given instruction
  - We can enable and disable (*mask*) most interrupts as needed *(maskable)*, others are *non-maskable*
- Interrupt service routine (ISR)
  - Subroutine which processor is ***forced to execute*** to respond to a ***specific event***
  - After ISR completes, MCU goes back to previously executing code

Nested Vectored Interrupt Controller



- NVIC manages and prioritizes external interrupts for Cortex-M0+
- Interrupts are types of exceptions
  - Exceptions 16 through 16+N
- Modes
  - Thread Mode: entered on Reset
  - Handler Mode: entered on executing an exception
- Privilege level
- Stack pointers
  - Main Stack Pointer, MSP
  - Process Stack Pointer, PSP
- Exception states: Inactive, Pending, Active, A&P

Some Interrupt Sources (Partial)

Vector Start Address	Vector #	IRQ	Source	Description
0x0000_0004	1		ARM Core	Initial program counter
0x0000_0008	2		ARM Core	Non-maskable interrupt
0x0000_0040-4C	16-19	0-3	Direct Memory Access Controller	Transfer complete or error
0x0000_0058	22	6	Power Management Controller	Low voltage detection
0x0000_0060-64	24-25	8-9	I²C Modules	Status and error
0x0000_0068-6C	26-27	10-11	SPI Modules	Status and error
0x0000_0070-78	28-30	12-14	UART Modules	Status and error
0x0000_00B8	46	30	Port Control Module	Port A Pin Detect
0x0000_00BC	47	31	Port Control Module	Port D Pin Detect

Up to 32 non-core vectors, 16 core vectors

From KL25 Sub-Family Reference Manual, Table 3-6

NVIC Registers and State

Bits	31:30	29:24	23:22	21:16	15:14	13:8	7:6	5:0
IPR0	IRQ3	reserved	IRQ2	reserved	IRQ1	reserved	IRQ0	reserved
IPR1	IRQ7	reserved	IRQ6	reserved	IRQ5	reserved	IRQ4	reserved
IPR2	IRQ11	reserved	IRQ10	reserved	IRQ9	reserved	IRQ8	reserved
IPR3	IRQ15	reserved	IRQ14	reserved	IRQ13	reserved	IRQ12	reserved
IPR4	IRQ19	reserved	IRQ18	reserved	IRQ17	reserved	IRQ16	reserved
IPR5	IRQ23	reserved	IRQ22	reserved	IRQ21	reserved	IRQ20	reserved
IPR6	IRQ27	reserved	IRQ26	reserved	IRQ25	reserved	IRQ24	reserved
IPR7	IRQ31	reserved	IRQ30	reserved	IRQ29	reserved	IRQ28	reserved

- Priority - allows program to prioritize response if both interrupts are requested simultaneously
  - IPR0-7 registers: two bits per interrupt source, four interrupt sources per register
  - Set priority to 0 (highest priority), 64, 128 or 192 (lowest)
  - CMSIS: NVIC_SetPriority(IRQnum, priority)

NVIC Registers and State

- Enable - Allows interrupt to be recognized
  - Accessed through two registers (set bits for interrupts)
    - » Set enable with NVIC_ISER, clear enable with NVIC_ICER
  - CMSIS Interface: NVIC_EnableIRQ(IRQnum), NVIC_DisableIRQ(IRQnum)
- Pending - Interrupt has been requested but is not yet serviced
  - CMSIS: NVIC_SetPendingIRQ(IRQnum), NVIC_ClearPendingIRQ(IRQnum)

Core Exception Mask Register

- Similar to "Global interrupt disable" bit in other MCUs
- PRIMASK - Exception mask register (CPU core)
  - Bit 0: PM Flag
    - » Set to 1 to prevent activation of all exceptions with configurable priority
    - » Clear to 0 to allow activation of all exceptions
  - Access using CPS, MSR and MRS instructions
  - Use to prevent data race conditions with code needing atomicity
- CMSIS-CORE API
  - void __enable_irq() - clears PM flag
  - void __disable_irq() - sets PM flag
  - uint32_t __get_PRIMASK() - returns value of PRIMASK
  - void __set_PRIMASK(uint32_t x) - sets PRIMASK to x

Prioritization

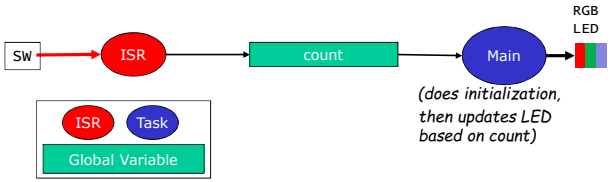
- Exceptions are prioritized to order the response simultaneous requests (smaller number = higher priority)
- Priorities of some exceptions are ***fixed***
  - Reset: -3, highest priority
  - NMI: -2
  - Hard Fault: -1
- Priorities of other (peripheral) exceptions are ***adjustable***
  - Value is stored in the interrupt priority register (IPR0-7)
  - 0x00
  - 0x40
  - 0x80
  - 0xC0

Special Cases of Prioritization

- Simultaneous exception requests?
  - Lowest exception type number is serviced first
- New exception requested while a handler is executing?
  - New priority higher than current priority?
    - » New exception handler **preempts** current exception handler
  - New priority lower than or equal to current priority?
    - » New exception held in **pending state**
    - » Current handler continues and completes execution
    - » Previous priority level restored
    - » New exception handled if priority level allows

EXAMPLE USING PORT MODULE AND EXTERNAL INTERRUPTS

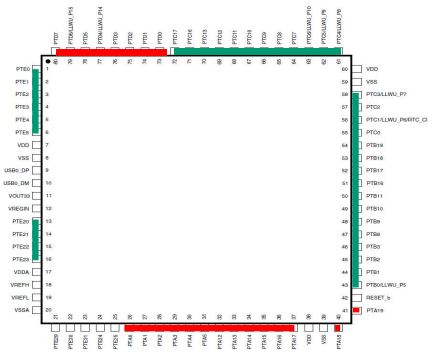
Refresher: Program Requirements & Design



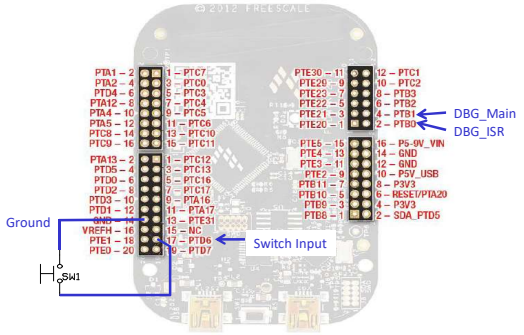
- Req1: When Switch SW is pressed, ISR will increment count variable
- Req2: Main code will light LEDs according to count value in binary sequence (Blue: 4, Green: 2, Red: 1)
- Req3: Main code will toggle its debug line DBG_MAIN each time it executes
- Req4: ISR will raise its debug line DBG_ISR (and lower main's debug line DBG_MAIN) whenever it is executing

KL25Z GPIO Ports with Interrupts

- Port A (PTA) through Port E (PTE)
- Not all port bits are available (package-dependent)
- Ports A and D support interrupts



FREEDOM KL25Z Physical Set-up



Building a Program – Break into Pieces

- First break into threads, then break thread into steps
  - Main thread:
    - » First initialize system
      - » initialize switch: configure the port connected to the switches to be input
      - » initialize LEDs: configure the ports connected to the LEDs to be outputs
      - » initialize interrupts: initialize the interrupt controller
    - Then repeat
      - » Update LEDs based on count
  - Switch Interrupt thread:
    - » Update count
- Determine which variables ISRs will share with main thread
  - This is how ISR will send information to main thread
  - Mark these shared variables as *volatile* (more details ahead)
  - Ensure access to the shared variables is *atomic* (more details ahead)

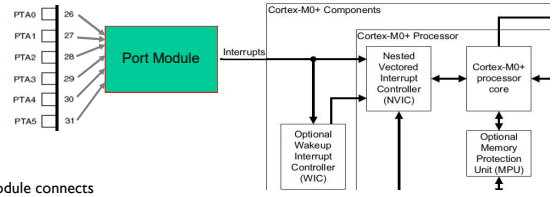
Where Do the Pieces Go?

- main
  - top level of main thread code
- switches
  - #defines for switch connections
  - declaration of count variable
  - Code to initialize switch and interrupt hardware
  - ISR for switch
- LEDs
  - #defines for LED connections
  - Code to initialize and light LEDs
- debug_signals
  - #defines for debug signal locations
  - Code to initialize and control debug lines

Configure MCU to Respond to the Interrupt

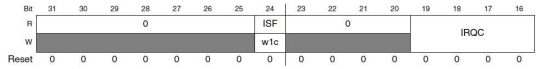
- Set up peripheral module to generate interrupt
  - We'll use Port Module in this example
- Set up NVIC
- Set global interrupt enable
  - Use CMSIS Macro __enable_irq();
  - This flag does not enable all interrupts; instead, it is an easy way to ***disable*** interrupts
    - » Could also be called "don't disable all interrupts"

Port Module



- Port Module connects external pins to NVIC (and other devices)
- Relevant registers
  - PCR - Pin control register (32 per port)
    - » Each register corresponds to an input pin
  - ISFR - Interrupt status flag register (one per port)
    - » Each bit corresponds to an input pin
    - » Bit is set to 1 if an interrupt has been detected

Pin Control Register



- ISF indicates if interrupt has been detected - different way to access same data as ISFR
- IRQC field of PCR defines behavior for external hardware interrupts
- Can also trigger direct memory access (not covered here)

IRQC	Configuration
0000	Interrupt Disabled
....	DMA, reserved
1000	Interrupt when logic zero
1001	Interrupt on rising edge
1010	Interrupt on falling edge
1011	Interrupt on either edge
1100	Interrupt when logic one
...	reserved

CMSIS C Support for PCR

- MKL25Z4.h defines PORT_Type structure with a PCR field (array of 32 integers)

```
/** PORT - Register Layout Typedef */
typedef struct {
  __IO uint32_t PCR[32];/** Pin Control Register n, array offset: 0x0, array
step: 0x4 */
  __O  uint32_t GPCLR;    /** Global Pin Control Low Register, offset: 0x80 */
  __O  uint32_t GPCHR;    /** Global Pin Control High Register, offset: 0x84 */
       uint8_t RESERVED_0[24];
  __IO uint32_t ISFR; /** Interrupt Status Flag Register, offset: 0xA0 */
} PORT_Type;
```

CMSIS C Support for PCR

- Header file defines pointers to PORT_Type registers

```
/* PORT - Peripheral instance base addresses */
/** Peripheral PORTA base address */
#define PORTA_BASE        (0x40049000u)
/** Peripheral PORTA base pointer */
#define PORTA             ((PORT_Type *)PORTA_BASE)
```

- Also defines macros and constants

```
#define PORT_PCR_MUX_MASK 0x700u
#define PORT_PCR_MUX_SHIFT        8
#define PORT_PCR_MUX(x)
(((uint32_t)(((uint32_t)(x))<<PORT_PCR_MUX_SHIFT)) &PORT_PCR_MUX_MASK)
```

Switch Interrupt Initialization

```
#include <MKL25Z4.H>
#include "switches.h"
#include "LEDs.h"
volatile unsigned count=0;
void init_switch(void) {
        /* enable clock for port D */
        SIM->SCGC5 |=  SIM_SCGC5_PORTD_MASK;
        /* Select GPIO and enable pull-up resistors and
                interrupts on falling edges for pin connected to switch */
        PORTD->PCR[SW_POS] |= PORT_PCR_MUX(1) | PORT_PCR_PS_MASK | PORT_PCR_PE_MASK
| PORT_PCR_IRQC(0x0a);
                /* Set port D switch bit to inputs */
        PTD->PDDR &= ~MASK(SW_POS);
        /* Enable Interrupts */
        NVIC_SetPriority(PORTD_IRQn, 128);
        NVIC_ClearPendingIRQ(PORTD_IRQn);
        NVIC_EnableIRQ(PORTD_IRQn);
}
```

Main Function

```
int main (void) {

        init_switch();
        init_RGB_LEDs();
        init_debug_signals();
        __enable_irq();

        while (1) {
                DEBUG_PORT->PTOR = MASK(DBG_MAIN_POS);
                control_RGB_LEDs(count&1, count&2, count&4);
                __wfi(); // sleep now, wait for interrupt
        }
}
```

```
void Control_RGB_LEDs(int r_on, int g_on, int b_on) {
        if (r_on)
                PTB->PCOR = MASK(RED_LED_POS);
        else
                PTB->PSOR = MASK(RED_LED_POS);
        if (g_on)
                PTB->PCOR = MASK(GREEN_LED_POS);
        else
                PTB->PSOR = MASK(GREEN_LED_POS);
        if (b_on)
                PTD->PCOR = MASK(BLUE_LED_POS);
        else
                PTD->PSOR = MASK(BLUE_LED_POS);
}
```

Write Interrupt Service Routine

- No arguments or return values – void is only valid type
- Keep it short and simple
  - Much easier to debug
  - Improves system response time
- Name the ISR according to CMSIS-CORE system exception names
  - PORTD_IRQHandler, RTC_IRQHandler, etc.
  - The linker will load the vector table with this handler rather than the default handler
- Clear pending interrupts
  - Call NVIC_ClearPendingIRQ(IRQnum)
- Read interrupt status flag register to determine source of interrupt

ISR

```
void PORTD_IRQHandler(void) {
        DEBUG_PORT->PSOR = MASK(DBG_ISR_POS);
        // clear pending interrupts
        NVIC_ClearPendingIRQ(PORTD_IRQn);

        if ((PORTD->ISFR & MASK(SW_POS))) {
              count++;
        }
        // clear status flags
        PORTD->ISFR = 0xffffffff;
        DEBUG_PORT->PCOR = MASK(DBG_ISR_POS);
}
```

Volatile Data

- Compilers assume that variables in memory do not change spontaneously, and optimize based on that belief
  - *Don't reload a variable from memory if current function hasn't changed it*
  - Read variable from memory into register (faster access)
  - Write back to memory at end of the procedure, or before a procedure call, or when compiler runs out of free registers
- This optimization can fail
  - Example: reading from input port, polling for key press
    - » while (SW_0) ; will read from SW_0 once and reuse that value
    - » Will generate an infinite loop triggered by SW_0 being true
- Variables for which it fails
  - Memory-mapped peripheral register – register changes on its own
  - Global variables modified by an ISR – ISR changes the variable
  - Global variables in a multithreaded application – another thread or ISR changes the variable

The Volatile Directive

- Need to tell compiler which variables may change outside of its control
  - Use volatile keyword to force compiler to reload these vars from memory for each use

    ```
    volatile unsigned int num_ints;
    ```
  - Pointer to a volatile int

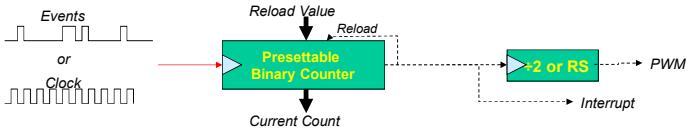
    ```
    volatile int * var; // or
    int volatile * var;
    ```
  - Now each C source read of a variable (e.g. status register) will result in an assembly language LDR instruction
  - Good explanation in Nigel Jones' "Volatile," *Embedded Systems Programming* July 2001

Module 4 - Timers

KL25 Timer Peripherals

- PIT - Periodic Interrupt Timer
  - Can generate periodically generate interrupts or trigger DMA (direct memory access) transfers
- TPM - Timer/PWM Module
  - Connected to I/O pins, has input capture and output compare support
  - Can generate PWM signals
  - Can generate interrupts and DMA requests
- LPTMR - Low-Power Timer
  - Can operate as timer or counter in all power modes (including low-leakage modes)
  - Can wake up system with interrupt
  - Can trigger hardware
- Real-Time Clock
  - Powered by external 32.768 kHz crystal
  - Tracks elapsed time (seconds) in 32-bit register
  - Can set alarm
  - Can generate 1Hz output signal and/or interrupt
  - Can wake up system with interrupt
- SYSTICK
  - Part of CPU core's peripherals
  - Can generate periodic interrupt

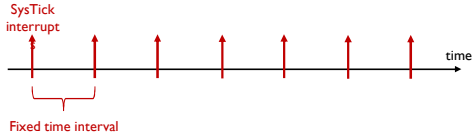
Timer/Counter Peripheral Introduction



- Common peripheral for microcontrollers
- Based on pre-settable binary counter, enhanced with configurability
  - Count value can be read and written by MCU
  - Count **direction** can often be set to up or down
  - Counter's **clock source** can be selected
    - » **Counter mode:** count **pulses** which indicate **events** (e.g. odometer pulses)
    - » **Timer mode**: clock source is periodic, so counter value is proportional to **elapsed time** (e.g. stopwatch)
  - Counter's **overflow/underflow action** can be selected
    - » Generate interrupt
    - » Reload counter with special value and continue counting
    - » Toggle hardware output signal
    - » Stop!

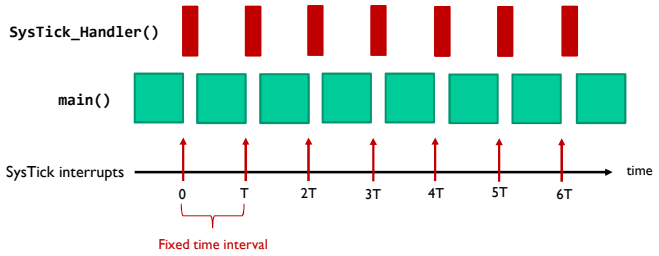
System Timer (SysTick) – *(SysTick Slides are based on the slides of Dr. Yifeng Zhu)*

- Generate SysTick interrupts at a fixed time interval



- Example Usages:
  - Measuring time elapsed, such as time delay function
  - Executing tasks periodically, such as periodic polling, and OS CPU scheduling

System Timer (SysTick)



System Timer (SysTick)

- System timer is a standard hardware component built into ARM Cortex-M.
- This hardware periodically forces the processor to execute the following ISR:

```
void SysTick_Handler(void){
  ...
}
```

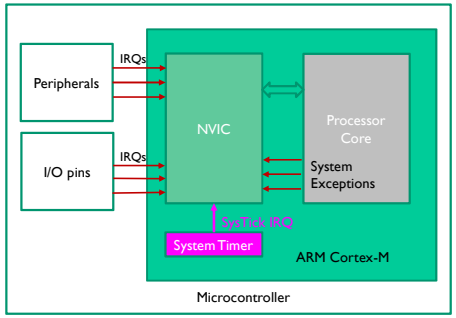
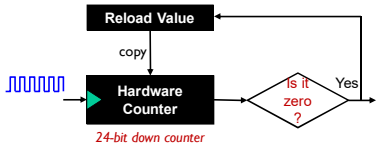


Diagram of System Timer (SysTick)



System Timer

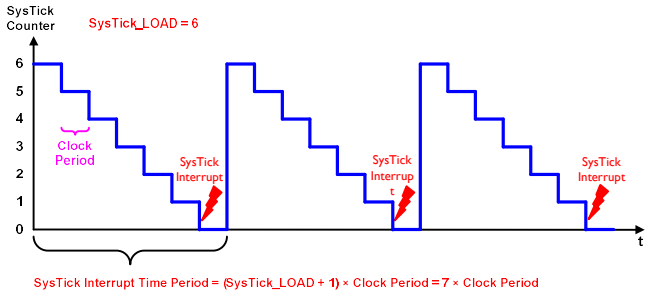


Diagram of System Timer (SysTick)

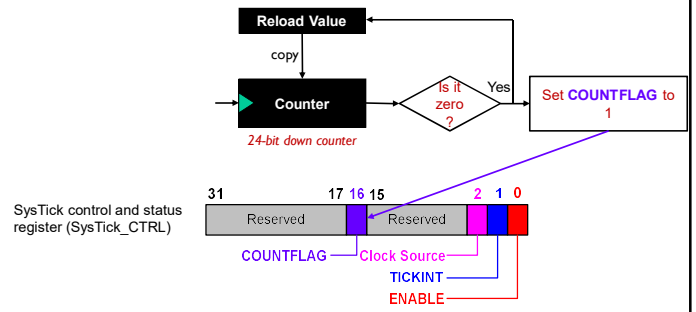
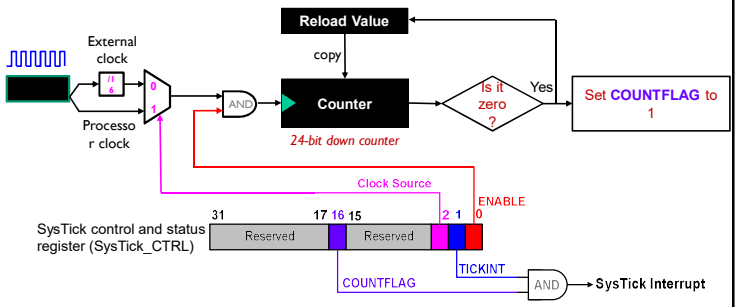
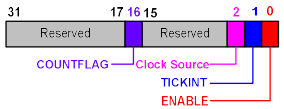


Diagram of System Timer (SysTick)



Registers of System Timer

SysTick control and status register (SysTick_CTRL)



SysTick reload value register (SysTick_LOAD)

SysTick current value register (SysTick_VAL)

Registers of System Timer

SysTick reload value register (SysTick_LOAD)

- 24 bits, maximum value `0x00FF.FFFF` (16,777,215)
- Counter counts down from RELOAD value to 0.
- Writing RELOAD to 0 disables SysTick, independently of TICKINT
- Time interval between two SysTick interrupts

  `Interval = (RELOAD + 1) × Source_Clock_Period`
- If 100 clock periods between two SysTick interrupts

  `RELOAD = 99`

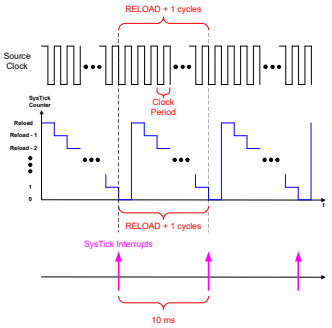
Registers of System Timer

- Reading it returns the current value of the counter
- When it transits from 1 to 0, it generates an interrupt
- Writing to SysTick_VAL clears the counter and COUNTFLAG to zero
  - Cause the counter to reload on the next timer clock
  - But, does not trigger an SysTick interrupt
- It has random value on reset.
  - Always clear it before enabling the timer

Calculating Reload Value

- Suppose clock source = 80MHz
- Goal: SysTick Interval = 10ms
- What is RELOAD value?

$$Reload = \frac{10\ ms}{Clock\ Period} - 1$$
$$= 10ms \times Clock\ Frequency\ - 1$$
$$= 10ms \times 80MHz - 1$$
$$= 10 \times 10^{-3} \times 80 \times 10^{6} - 1$$
$$= 800000 - 1$$
$$= 799999$$



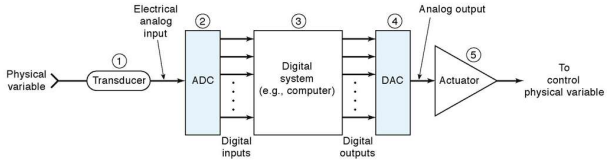
Example Code (Textbook page 189)

```
void Init_SysTick (void) {
    SysTick->CTRL = 0;            // Disable SysTick
    SysTick->LOAD = 0x13FFFF;    // Set reload register to get 1s interrupts clk=20971520
    NVIC_SetPriority(SysTick_IRQn, 3);
    SysTick->VAL = 0;             // Reset the SysTick counter value
    SysTick->CTRL |= SysTick_CTRL_TICKINT_Msk | SysTick_CTRL_ENABLE_Msk;
}

void SysTick_Handler() {
static int n=0;
Control_RGB_LEDs(n&1,n&1,n&1);
n++;
}
```

Module 5 - Analog Interfacing

Interfacing With the Analog World

- Transducer
- ADC
- Computer
- DAC
- Actuator



Why It's Needed

- Embedded systems often need to measure values of physical parameters
- These parameters are usually continuous (*analog*) and not in a digital form which computers (which operate on discrete data values) can process

- Temperature
  - Thermometer (do you have a fever?)
  - Thermostat for building, fridge, freezer
  - Car engine controller
  - Chemical reaction monitor
  - Safety (e.g. microprocessor processor thermal management)
- Light (or infrared or ultraviolet) intensity
  - Digital camera
  - IR remote control receiver
  - Tanning bed
  - UV monitor
- Rotary position
  - Wind gauge
  - Knobs
- Pressure
  - Blood pressure monitor
  - Altimeter
  - Car engine controller
  - Scuba dive computer
  - Tsunami detector
- Acceleration
  - Air bag controller
  - Vehicle stability
  - Video game remote
- Mechanical strain
- Other
  - Touch screen controller
  - EKG, EEG
  - Breathalyzer

CONVERTING BETWEEN ANALOG AND DIGITAL VALUES

The Big Picture – A Depth Gauge

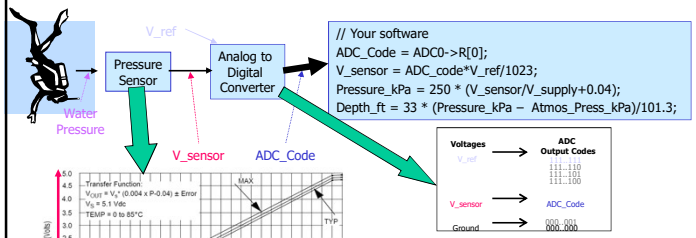
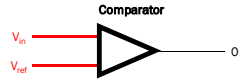


Figure 4. Output vs. Absolute Pressure

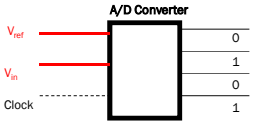
1. Sensor detects *water pressure* and generates a proportional *output voltage* V_sensor
2. ADC generates a proportional digital *integer* (code) based on V_sensor and V_ref
3. Code can convert that integer to a something more useful
   1. first a float representing the *voltage*,
   2. then another float representing *pressure*,
   3. finally another float representing *depth*

Getting From Analog to Digital

- A **Comparator** tells us "Is $V_{in} > V_{ref}$?"
  - Compares an **analog input voltage** with an **analog reference voltage** and determines which is larger, returning a 1-bit number
  - E.g. Indicate if depth > 100 ft
  - Set V_{ref} to voltage pressure sensor returns with 100 ft depth.

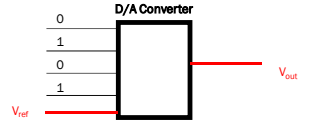


- An **Analog to Digital converter** [AD or ADC] tells us how large V_{in} is as a fraction of V_{ref}.
  - Reads an analog input signal (usually a voltage) and produces a corresponding multi-bit number at the output.
  - E.g. calculate the depth

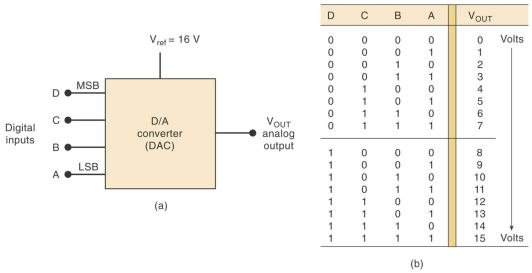


Digital to Analog Conversion

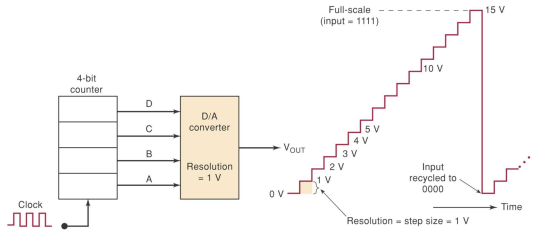
- May need to generate an analog voltage or current as an output signal
  - E.g. audio signal, video signal brightness.
- DAC: "Generate the analog voltage which is this fraction of V_{ref}"
- Digital to Analog Converter equation
  - *n* = input code
  - *N* = number of bits of resolution of converter
  - V_{ref} = reference voltage
  - V_{out} = output voltage. Either
    - $V_{out} = V_{ref} * n/(2^N)$ or
    - $V_{out} = V_{ref} * (n+1)/(2^N)$
    - *The offset +1 term depends on the internal tap configuration of the DAC – check the datasheet to be sure*



4-bit DAC



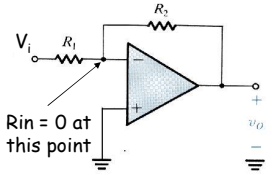
More



Nominal Full-scale value = 16 V
Resolution = Step Size = LSB = 16V / 2^4 = 1 V
Full scale output = Nominal Full-scale value – Step Size = 15 V

Summary of Op-amp Behavior

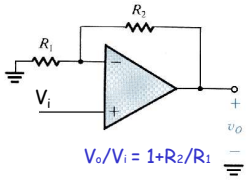
Inverting configuration



$V_o / V_i = - R_2/R_1$

$R_{in} = R_1$

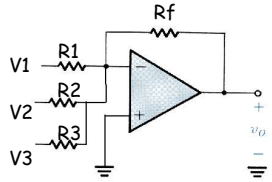
Noninverting configuration



$V_o/V_i = 1+R_2/R_1$

R_{in} = infinity

The Weighted Summer



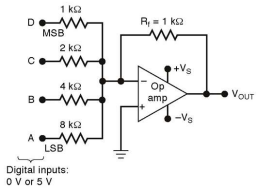
Current in R_1, R_2, and R_3 add to current in R_f

$V_o = -R_f(V_1/R_1 + V_2/R_2 + V_3/R_3)$

This circuit is called a weighted summer

D/A Converters

- A summing operational amplifier with a resolution of 0.625 V



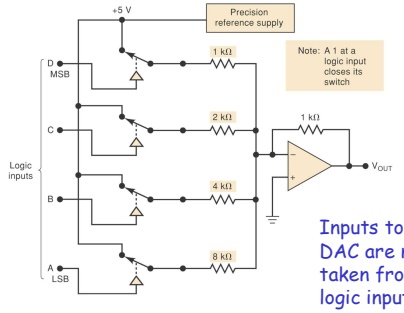
Input code

D	C	B	A	V_{OUT} (volts)	
0	0	0	0	0	
0	0	0	1	−0.625	← LSB
0	0	1	0	−1.250	
0	0	1	1	−1.875	
0	1	0	0	−2.500	
0	1	0	1	−3.125	
0	1	1	0	−3.750	
0	1	1	1	−4.375	
1	0	0	0	−5.000	
1	0	0	1	−5.625	
1	0	1	0	−6.250	
1	0	1	1	−6.875	
1	1	0	0	−7.500	
1	1	0	1	−8.125	
1	1	1	0	−8.750	
1	1	1	1	−9.375	← Full-scale

Resolution = |5V(1K/8K)| = .625V

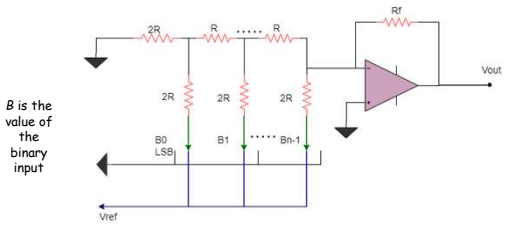
Max out = 5V(1K/8K + 1K/4K + 1K/2K + 1K/1K) = -9.375V

Complete Four-bit DAC Including a ***Precision*** Reference Supply.



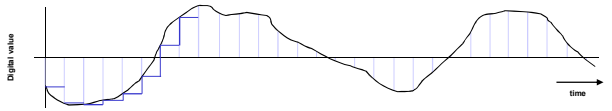
Inputs to the DAC are not taken from the logic inputs due to inaccuracies.

Basic R/2R ladder DAC



https://microcontrollerslab.com/r-2r-ladder-dac-digital-to-analog-converter-working-examples-circuits/

Waveform Sampling and Quantization



- A waveform is **sampled** at a constant rate – every Δt
  - Each such sample represents the instantaneous amplitude at the instant of sampling
  - *"At 37 ms, the input is 1.913419145134514512343 11… V"*
  - Sampling converts a **continuous time** signal to a **discrete time** signal
- The sample can now be **quantized** (converted) into a digital value
  - Quantization represents a **continuous** (analog) value with the closest **discrete** (digital) value
  - *"The sampled input voltage of 1.913419145134514512343 11… V is best represented by the code 0x018, since it is in the range of 1.901 to 1.9980 V which corresponds to code 0x018."*

Forward Transfer Function Equations

What code n will the ADC use to represent voltage V_{in}?

General Equation

n = converted code
V_{in} = sampled input voltage
V_{+ref} = upper voltage reference
V_{-ref} = lower voltage reference
N = number of bits of resolution in ADC

$$n = \left\lfloor \frac{(V_{in} - V_{-ref})\, 2^N}{V_{+ref} - V_{-ref}} + 1/2 \right\rfloor$$

Simplification with V_{-ref} = 0 V

$$n = \left\lfloor \frac{(V_{in})\, 2^N}{V_{+ref}} + 1/2 \right\rfloor$$

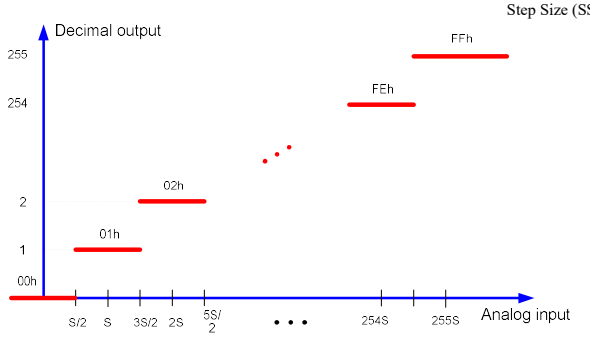
$$n = \left\lfloor \frac{3.30V\ 2^{10}}{5V} + 1/2 \right\rfloor = 676$$

$\lfloor X \rfloor$ = ***I*** *floor function: nearest integer I such that I <= X*
floor(x+0.5) rounds x to the nearest integer

Ideal ADC Example N=8

$V_{-ref} = 0V$

Step Size (SS) = $\frac{V_{+ref}}{2^N}$



Inverse Transfer Function

What range of voltages V_{in_min} to V_{in_max} does code n represent?

General Equation

n = converted code

V_{in_min} = minimum input voltage for code n

V_{in_max} = maximum input voltage for code n

V_{+ref} = upper voltage reference

V_{-ref} = lower voltage reference

N = number of bits of resolution in ADC

$$V_{in\_min} = \frac{n - \frac{1}{2}}{2^N}(V_{+ref} - V_{-ref}) + V_{-ref}$$

$$V_{in\_max} = \frac{n + \frac{1}{2}}{2^N}(V_{+ref} - V_{-ref}) + V_{-r}$$

Simplification with V_{-ref} = 0 V

$$V_{in\_min} = \frac{n - \frac{1}{2}}{2^N}(V_{+re}\ )$$

$$V_{in\_max} = \frac{n + \frac{1}{2}}{2^N}(V_{+ref})$$

ANALOG TO DIGITAL CONVERSION CONCEPTS

What if the Reference Voltage is not known?

- Example - running off an unregulated battery (to save power)
- Measure a known voltage and an unknown voltage

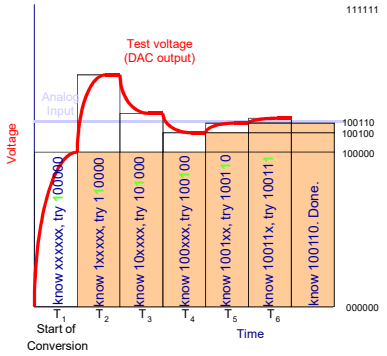
$$V_{unknown} = V_{known}\frac{n_{unknown}}{n_{known}} \qquad V_{ref} = V_{known}\frac{2^N}{n}$$

"My ADC tells me that channel 27 returns a code of 0x6543, so I can calculate that V_{REFSH} = 1.0V * 2^{16}/0x6543 = …

- Many MCUs include an internal fixed voltage source which ADC can measure for this purpose
- Can also solve for Vref

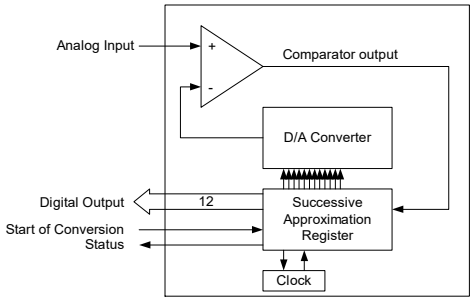
ADC - Successive Approximation Conversion

- Successively approximate input voltage by using a binary search and a DAC
- SA Register holds current approximation of result
- Set all DAC input bits to 0
- Start with DAC's most significant bit
- Repeat
  - Set next input bit for DAC to 1
  - Wait for DAC and comparator to stabilize
  - If the DAC output (test voltage) is **smaller** than the input then set the current bit to 1, else clear the current bit to 0



A/D - Successive Approximation

Converter Schematic



ADC Performance Metrics

- Linearity measures how well the transition voltages lie on a straight line.
- Differential linearity measure the equality of the step size.
- Conversion time: between start of conversion and generation of result
- Conversion *rate* = inverse of conversion *time*

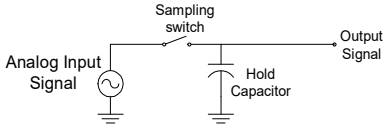
Sampling Problems

- Nyquist criterion
  - $F_{sample} >= 2 * F_{max\ frequency\ component}$
  - Frequency components above ½ F_{sample} are aliased, distort measured signal
- Nyquist and the real world
  - This theorem assumes we have a perfect filter with "brick wall" roll-off
  - Real world filters have more gentle roll-off
  - Inexpensive filters are even worse
  - So we have to choose a sampling frequency high enough that our filter attenuates aliasing components adequately

Inputs

- Differential
  - Use two channels, and compute difference between them
  - Very good noise immunity
  - Some sensors offer differential outputs (e.g. Wheatstone Bridge)
- Multiplexing
  - Typically share a single ADC among multiple inputs
  - Need to select an input, allow time to settle before sampling
- Signal Conditioning
  - Amplify and filter input signal
  - Protect against out-of-range inputs with clamping diodes

Sample and Hold Devices



- Some A/D converters require the input analog signal to be held constant during conversion (e.g. successive approximation devices)
- In other cases, peak capture or sampling at a specific point in time requires a sampling device.
- A "sample and hold" circuit performs this operation
- Many A/D converters include a sample and hold circuit

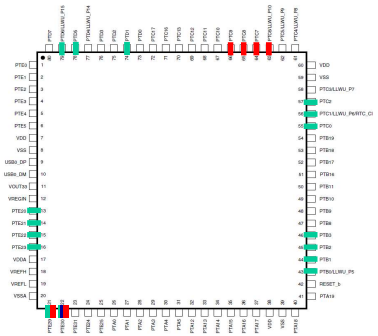
KL25 ANALOG INTERFACING PERIPHERALS

Sources of Information

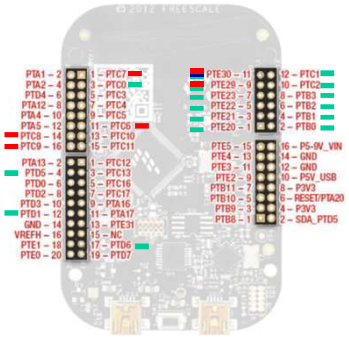
- KL25 Subfamily Reference Manual (Rev. 1, June 2012)
  - Describes architecture of peripherals and their control registers
  - Digital to Analog Converter
    - » Chapter 30 of KL25 Subfamily Reference Manual
  - Analog Comparator
    - » Chapter 29 of KL25 Subfamily Reference Manual
  - Analog to Digital Converter
    - » Chapter 28 of KL25 Subfamily Reference Manual
- KL25 Sub-family Data Sheet (Rev. 3, 9/19/2012)
  - Describes circuit-specific performance parameters: operating voltages, min/max speeds, cycle times, delays, power and energy use

KL25Z Analog Interface Pins

- 80-pin QFP
- Inputs
  - 1 16-bit ADC with 14 input channels
  - 1 comparator with 6 external inputs, one 6-bit DAC
- Output
  - 1 12-bit DAC



Freedom KL25Z Analog I/O



Inputs

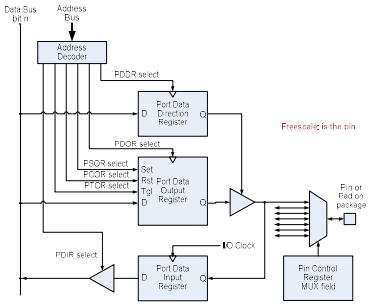
- 14 external ADC channels
- 6 external comparator channels

Output

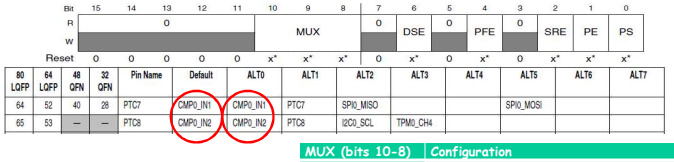
- 1 12-bit DAC

Using a Pin for Analog Input or Output

- Configuration
  - Direction
  - MUX
- Data
  - Output (different ways to access it)
  - Input



Pin Control Register to Select MUX Channel



Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	0								0	DSE	0	PFE	0	SRE	PE	PS
W	MUX															
Reset	0	0	0	0	0	x*	x*	x*	0	x*	0	x*	0	x*	x*	x*

80 LQFP	64 LQFP	48 QFN	32 QFN	Pin Name	Default	ALT0	ALT1	ALT2	ALT3	ALT4	ALT5	ALT6	ALT7
64	52	40	28	PTC7	CMP0_IN1	CMP0_IN1	PTC7	SPI0_MISO			SPI0_MOSI		
65	53	—	—	PTC8	CMP0_IN2	CMP0_IN2	PTC8	I2C0_SCL	TPM0_CH4				

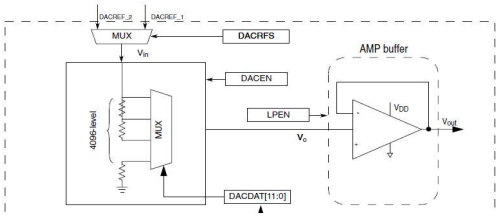
- MUX field of PCR defines connections

MUX (bits 10-8)	Configuration
000	Digital circuits disabled, analog enabled
001	Alternative 1 - GPIO
010	Alternative 2
011	Alternative 3
100	Alternative 4
101	Alternative 5
110	Alternative 6
111	Alternative 7

```
PORTC->PCR[7] &= ~PORT_PCR_MUX_MASK;
PORTC->PCR[7] |= PORT_PCR_MUX(0);
```

DIGITAL TO ANALOG CONVERTER

DAC Overview



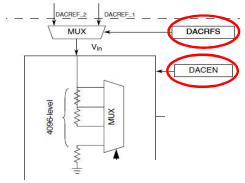
- Load DACDAT with 12-bit data N
- MUX selects a node from resistor divider network to create $V_o = (N+1)*V_{in}/2^{12}$
- V_o is buffered by output amplifier to create V_{out}
  - $V_o = V_{out}$ but V_o is high impedance - can't drive much of a load, so need to buffer it

DAC Operating Modes

- Normal
  - DAT0 is converted to voltage immediately
- Buffered
  - Data to output is stored in 16-word buffer
  - Next data item is sent to DAC when a selectable trigger event occurs
    - » Software Trigger - write to DACSWTRG field in DACx_C0
    - » Hardware Trigger - from PIT timer peripheral
  - Normal Mode
    - » Circular buffer
  - One-time Scan Mode
    - » Pointer advances until reaching upper limit of buffer, then stops
  - Status flags in DACx_SR

DAC Control Register 0: DACx_C0

Bit	7	6	5	4	3	2	1	0
Read Write	DACEN	DACRFS	DACTRGSEL	0 DACSWTRG	LPEN	0	DACBTIEN	DACBBIEN
Reset	0	0	0	0	0	0	0	0



- DACEN - DAC Enabled when 1
- DACRFS - DAC reference voltage select
  - 0: DACREF_1. Connected to VREFH
  - 1: DACREF_2. Connected to VDDA
- LPEN - low-power mode
  - 0: High-speed mode. Fast (15 us settling time) but uses more power (up to 900 uA supply current)
  - 1: Low-power mode. Slow (100 us settling time) but more power-efficient (up to 250 uA supply current)
- Additional control registers used for buffered mode

DAC Control Register 1: DACx_C1

Bit	7	6	5	4	3	2	1	0
Read Write	DMAEN	0				DACBFMD	0	DACBFEN
Reset	0	0	0	0	0	0	0	0

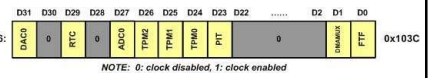
- DACBFEN
  - 0: Disable buffer mode
  - 1: Enable buffer mode
- DACBFMD - Buffer mode select
  - 0: Normal mode (circular buffer)
  - 1: One-time scan mode

DAC Data Registers

- These registers are only eight bits long
- DATA[11:0] stored in two registers
  - DATA0: Low byte [7:0] in DACx_DATnL
  - DATA1: High nibble [11:8] in DACx_DATnH

Example: Waveform Generator

- Supply clock to DAC0 m
  - Bit 31 of SIM SCGC6

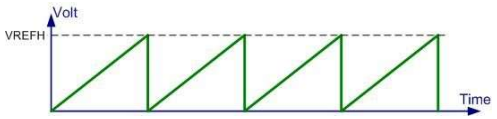


- Set Pin Mux to Analog (0)

- Enable DAC
- Configure DAC
  - Reference voltage
  - Low power mode?
  - Normal mode (not buffered)
- Write to DAC data register

Bit	Field	Descriptions
7	DACEN	DAC Enable (0: DAC is disabled, 1: DAC is enabled)
6	DACRFS	DAC Reference Select (0: DACREF_1, 1: DACREF_2)
5	DACTRGSEL	DAC Trigger Select (0: hardware trigger, 1: software trigger)
4	DACSWTRG	DAC Software Trigger
3	LPEN	DAC Low Power Control (0: High-Power mode, 1: Low-Power mode)
1	DACBTIEN	DAC Buffer read pointer Top flag Interrupt Enable
0	DACBBIEN	DAC Buffer read pointer Bottom flag Interrupt Enable

Program 7.4 from Mazidi etal - Saw Tooth Generation




```
void DAC0_init(void);
void delayMs(int n);
int main (void) {
int i;
DAC0_init(); /* Configure DAC0 */
while (1) {
for (i = 0; i < 0x1000; i += 0x0010) {
/* write value of i to DAC0 */
DAC0->DAT[0].DATL = i & 0xff; /* write low byte */
DAC0->DAT[0].DATH = (i >> 8) & 0x0f;/* write high byte */
delayMs(1); /* delay 1ms */
}
}
}
```

Code

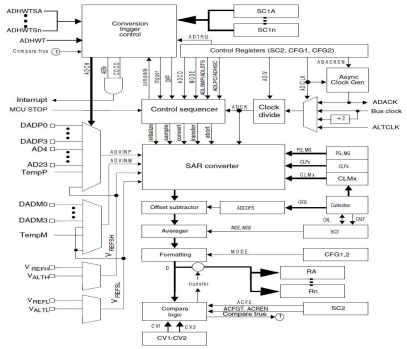
```
void DAC0_init(void)
{
PORTE->PCR[30] &= ~(0x700); /* alt function 0 */
SIM->SCGC6 |= 0x80000000; /* clock to DAC module */
DAC0->C1 = 0; /* disable the use of buffer */
DAC0->C0 = 0x80 | 0x20; /* enable DAC and use software trigger */
}
/* Delay n milliseconds
The CPU core clock is set to MCGFLLCLK at 41.94 MHz in SystemInit(). */
void delayMs(int n) {
int i;
int j;
for(i = 0 ; i < n; i++)
for (j = 0; j < 7000; j++) {}
}
```

ANALOG TO DIGITAL CONVERTER

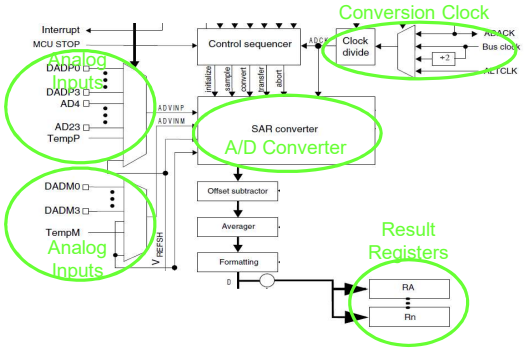
ADC Overview

- Uses successive approximation for conversion
- Supports multiple resolutions: 16, 13, 12, 11, 10, 9, and 8 bits
- Supports single-ended and differential conversions
- Signed or unsigned results available
- Up to 24 analog inputs supported (single-ended), 4 pairs of differential inputs
- Automatic compare and interrupt for level and range comparisons
- Hardware data averaging
- Temperature sensor

ADC System Overview



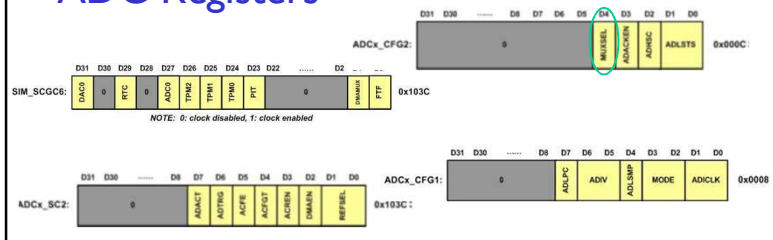
ADC System Fundamentals



Using the ADC

- ADC initialization
  - Configure clock
  - Select voltage reference
  - Select trigger source
  - Select input channel
  - Select other parameters
- Trigger conversion
- Read results

ADC Registers

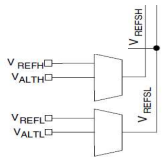


Bit	Field	Descriptions
7	ADACT	Conversion active: Indicates that the ADC is converting data (0: Conversion not in progress, 1: Conversion in progress)
6	ADTRG	ADC conversion trigger select (0: software trigger, 1: hardware trigger)
5	ACFE	Compare Function Enable (0: compare function disabled, 1: enabled)
4	ACFGT	Compare Function Greater Than Enable
3	ACREN	Compare range Enable
2	DMAEN	DMA Enable
1-0	REFSEL	Voltage Reference Select

Bit	Field	Descriptions
7	ADLPC	Low-Power Configuration
6-5	ADIV	Clock Divide Select: The clock is divided by 2^{ADIV} as shown in Figure 7-7.
4	ADLSMP	Sample time configuration (0: Short sample time, 1: Long sample time)
3-2	MODE	Conversion mode selection
1-0	ADICLK	Input Clock Select

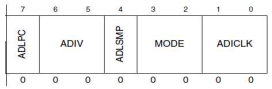
Voltage Reference Selection

- Two voltage reference pairs ava
  - V_{REFH}, V_{REFL}
  - V_{ALTH}, V_{ALTL}
- Select with SC2 register's REFSE
  - 00: V_{REFH}, V_{REFL}
  - 01: V_{ALTH}, V_{ALTL}
  - 10, 11: Reserved
- KL25Z
  - V_{ALTH} connected to V_{DDA}



Conversion Options Selection

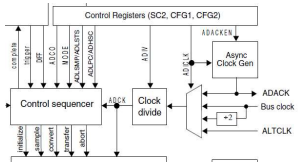
- Low power
  - Set ADLPC (in ADCx_CFG1) to 1
  - Slower max clock speed
- Long sample time select
  - Set ADLSMP (in ADCx_CFG1) to 1
  - Can select longer sample time with ADLSTS bits (in ADCx_CFG2) to add 20, 16, 10 or 6 ADCK cycles)
- Conversion mode
  - MODE (in ADCx_CFG1)
  - Sets result precision (8 through 16 bits)
- Continuous vs. single conversion
  - Set ADCO (in ADCx_SC3) to 1 for continuous conversions



	DIFF	
MODE	0	1
0	Single ended 8-bit	Differential 9-bit 2's complement
1	Single ended 12-bit	Differential 13-bit 2's complement
2	Single ended 10-bit	Differential 11-bit 2's complement
3	Single ended 16-bit	Differential 16-bit 2's complement

Clock Configuration

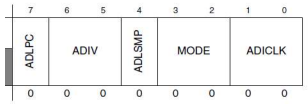
- Select clock source with ADICLK
  - Bus Clock (default)
  - ADACK: Local clock, allows ADC operation while rest of CPU is in stop mode
  - ALTCLK: alternate clock (MCU-specific)

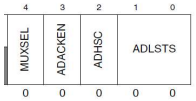


- Divide down selected clock by factor of ADIV, creating ADCK
- Resulting ADCK must be within valid range to ensure accuracy (See KL25 Subfamily datasheet)
  - 1 to 18 MHz (<= 13-bit mode)
  - 2 to 12 MHz (16-bit mode)

Clock Configuration Registers

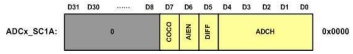
- ADCx_CFG1
  - ADIV: divide clock by 2^{ADIV}
    - 00: 1
    - 01: 2
    - 10: 4
    - 11: 8
  - ADICLK: Input clock select
    - 00: Bus clock
    - 01: Bus clock/2
    - 10: ALTCLK
    - 11: ADACK





- ADCx_CFG2
  - ADACKEN: Enable asynchronous clock

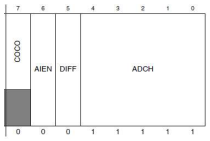
Registers

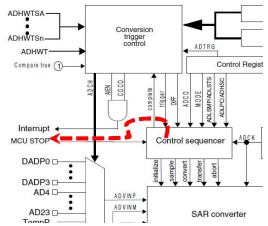


Bit	Field	Descriptions
7	COCO	Conversion Complete Flag: (0: Conversion is not completed, 1: Conversion is completed) The COCO is cleared when the ADCx_SC1n register is written or the ADCx_Rn register is read.
6	AIEN	Interrupt Enable: The ADC interrupt is enabled by setting the bit to HIGH. If the interrupt enable is set, an interrupt is triggered when the COCO flag is set.
5	DIFF	Differential mode (0: Single-ended mode, 1: Differential mode)
4-0	ADCH	ADC input channel: The field selects the input channel as shown in Figure 7-7. When DIFF = 0 (single-ended mode), values 0 to 23 choose between the 24 input channels (ADC_SE0 to ADC_SE23). When DIFF = 1 (Differential mode), values 0 to 3 select between the 4 differential channels. See the reference manual for more information. When ADCH = 11111, the module is disabled.

Pin Name	Description	Pin
ADC_SE0	ADC input 0	PTE20
ADC_SE3	ADC input 3	PTE22
ADC_SE4	ADC input 4	PTE21, PTE29
ADC_SE5	ADC input 5	PTD1
ADC_SE6	ADC input 6	PTD5
ADC_SE7	ADC input 7	PTD6, PTE23
ADC_SE8	ADC input 8	PTB0
ADC_SE9	ADC input 9	PTB1
ADC_SE11	ADC input 11	PTC2
ADC_SE12	ADC input 12	PTB2
ADC_SE13	ADC input 13	PTB3
ADC_SE14	ADC input 14	PTC0
ADC_SE15	ADC input 15	PTC1
ADC_SE23	ADC input 23, DAC0 output	PTE30
ADC_SE26	Temperature sensor	
ADC_SE27	Bandgap reference	
ADC_SE29	V_{REFH}	
ADC_SE30	V_{REFL}	
ADC_SE31	Module disabled	

Conversion Completion

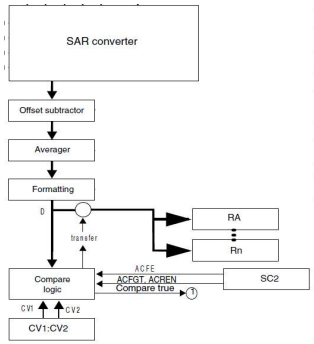




- Signaled by COCO bit in SC1n
- Can generate conversion complete interrupt if AIEN in SC1 is set
  - CMSIS-defined ISR name for ADC Interrupt is ADC0_IRQHandler

Result Registers

- Optional output processing before storage in result registers
  - Offset subtraction from calibration
  - Averaging: 1, 4, 8, 16 or 32 samples
  - Formatting: Right justification, sign- or zero-extension to 16 bits
  - Output comparison
- Two result registers RA and Rn
  - Conversion result goes into register corresponding to SC1 register used to start conversion (SC1A, SC1n)



Procedure – Polling Method

- Enable the clock to I/O pin used by the ADC channel. Table shows the I/O pins used by various ADC channels.
- Set the PORTX_PCRn MUX bit for ADC input pin to 0 to use the pin for analog input channel. This is actually the power-on default.
- Enable the clock to ADC0 modules using SIM_SCGC6 register.
- Choose the software trigger using the ADC0_SC2 register.
- Choose clock rate and resolution using ADC0_CFG1 register.
- Select the ADC input channel using the ADC0_SC1A register. Make sure interrupt is not enabled and single-ended option is used when you select the channel with this register.
- Keep monitoring the end-of-conversion COCO flag in ADC0_SC1A register.
- When the COCO flag goes HIGH, read the ADC result from the ADC0_RA and save it.
- Repeat steps 6 through 8 for the next conversion.

Example – Listing 6.5

```
#define ADC_POS (20)
void Init_ADC(void) {

          SIM->SCGC6 |= SIM_SCGC6_ADC0_MASK;
          SIM->SCGC5 |= SIM_SCGC5_PORTE_MASK;

          // Select analog for pin
          PORTE->PCR[ADC_POS] &= ~PORT_PCR_MUX_MASK;
          PORTE->PCR[ADC_POS] |= PORT_PCR_MUX(0);

          // Low power configuration, long sample time, 16
bit single-ended conversion, bus clock input
          ADC0->CFG1 = ADC_CFG1_ADLPC_MASK |
ADC_CFG1_ADLSMP_MASK | ADC_CFG1_MODE(3) | ADC_CFG1_ADICLK(1);
          // Software trigger, compare function disabled,
DMA disabled, voltage references VREFH and VREFL
          ADC0->SC2 = ADC_SC2_REFSEL(0);
}
```

Listing 6.6

```
float Measure_Temperature(void){
          float n, temp;

          ADC0->SC1[0] = 0x00; // start conversion on channel 0

          // Wait for conversion to finish
          while (!(ADC0->SC1[0] & ADC_SC1_COCO_MASK))
                    ;
          // Read result, convert to floating-point
          n = (float) ADC0->R[0];

          // Calculate temperature (Celsius) using polynomial equation
          // Assumes ADC is in 16-bit mode, has VRef = 3.3 V

          temp = -36.9861 + n*(0.0155762 + n*(-1.43216E-06 + n*(7.18641E-11
                    + n*(-1.84630E-15 + n*(2.32656E-20 + n*(-1.13090E-25))))));
          return temp;
}
```