C Programming for Embedded Systems

Primitive Data Types, Data Declaration

- Integer data types
  - Have both size and sign
  - char (8-bit)
  - short (16-bit)
  - int (32-bit)
  - long (32-bit)
  - signed (positive and negative)
  - unsigned (positive only)
- Floating-point types
  - Only have size
  - Can always be positive or negative
  - float (32-bit)
  - double (64-bit)
  - 12.34: constant of type double
  - 12.34f or 12.34F: constant of type float
- Data declarations should be at the top of a code block

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

float realNum;
double realNum2;
…
}
```

Data Types Defined in `stdint.h`

- Standard C data types
  - `typedef char int8_t;`
  - `typedef short int16_t;`
  - `typedef long int32_t;`

  - `typedef unsigned char uint8_t;`
  - `typedef unsigned short uint16_t;`
  - `typedef unsigned long uint32_t;`

Functions and Function Prototypes

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

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

C vs. C++

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

```
{
int j;
double x;

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

Useful Features for Embedded Code

- Storage Class Speciers & Type Qualifers
  - volatile
  - const
  - static
- Pointers
- Structures
- Bit Operations
- Integer Conversions

Volatile Type Qualifier

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

Const Type Qualifier

- The type qualifier `const` is used to declare that a variable is read-only and may not be changed in software.
  - Example: `uint16_t const max_temp = 1000;`
  - Could also write: `#define max_temp 1000`

- If a variable of type `const` is stored in a memory-mapped register, then its value cannot be changed in software BUT could be changed externally. In this case the variable should be defined as both `volatile` and `const`
  - Example: `volatile const uint32_t var_name;`

- NXP shorthand for `volatile` and `volatile const:`
  - `#define __I volatile const /*'read only' */`
  - `#define __O volatile /*'write only' */`
  - `#define __IO volatile /*'read/write' */`

Static Storage Class

- Two uses: in a file and in a code block (e.g., function)
  1) Declaring a variable `static` in a file limits the scope of that variable to the file in which it is declared. The variable will not conflict with other variables of the same name defined in other files. Similarly, declaring a function static in a file limits the scope of that function to the file.
  2) Declaring a variable `static` in a function allocates it a persistent memory location and it retains its value between successive function calls.
- In embedded code we often want a variable to retain its value between function calls
  – Consider a function that senses a change in the crankshaft angle: The function needs to know the previous angle in order to compute the difference
- Example: `static int x = 2 ;`

Global Variables

- Variables that are visible to routines in *all* files, not just the file in which they are defined
  - such variables are said to have *program scope*

- Use of global variables can make a program difficult to understand and maintain

- To make a variable local, declare it with the `static` keyword:

```
int j;              /*  global  */
static int k;       /*  local    */
main ()
{
    …
     }
```

Pointers

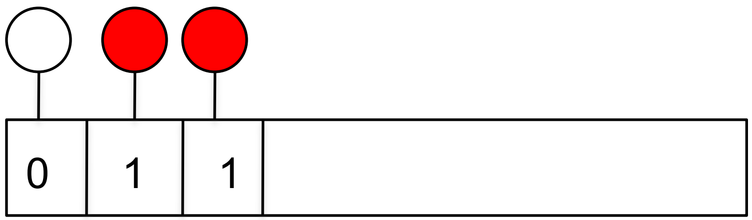
- Every variable has an *address* in memory and a *value*
- A pointer is a variable that stores an address
  - The value of a pointer is the location of another variable
- The size of a pointer variable is the size of an address
  - 4 bytes (32 bits) for the S32K144
- Two operators used with pointers
  - **`&`** operator returns the address of a variable
  - **`*`** is used to "de-reference" a pointer (*i.e.*, to access the *value* at the *address* stored by a pointer

Simple Pointer Example

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

```
int x, y;           /* x is located at address 0x100 */
int *ptr;          /* This is how to declare a pointer. Pointer
                    /* ptr points to an int */
x = 5;
ptr = &x;          /* The value of ptr is now 0x100 */
y = *ptr;          /* y now has the value 5 */
*ptr = 6;          /* The value at address 0x100 is 6 */
```

Another Way to Assign Pointers



- Declare a pointer, `p`, that points to a `uint32_t`

```
volatile uint32_t     *p;
```

- Allocate the memory and assign the address of the I/O memory location to the pointer ("dereference `p`")

```
p = (volatile uint32_t *) 0x30610000;
/* 4-byte long, address of some memory-
mapped register */
```

- Set the contents of memory location 0x30610000,

```
*p = 0x7FFFFFFF;   /* Turn on all but
the leftmost bit */
```

Pointer Arithmetic

- Recall that each memory address refers to an 8-bit byte of memory.
- Suppose that `p` is a pointer to a certain type of object (e.g., int, short, char). Then `p+i` points to the i'th object after the one `p` points to.
- Example

```
int *p, *p1, *p3
p = (int *) 0x1000;
p1 = p++;  /* value of p1 is 0x1004  */
p3 = p+3;  /* value of p3 is 0x100C  */
```

- Pointer indexing

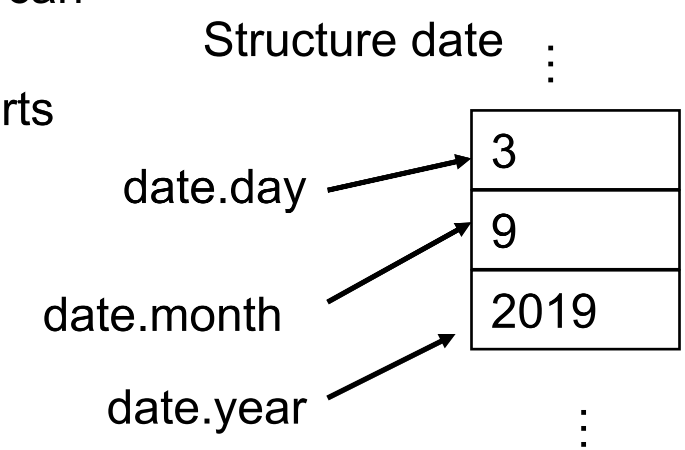
```
p[i] = *(p+i)
```

- Application: memory-mapped registers used to configure HW peripherals. Define pointer to first register and use indexing to access the rest

Structures

- A `struct` holds multiple variables
  - Divides range of memory into pieces that can be referenced individually
  - Example: The date consists of several parts

```
struct date {
 int day;
 int month;
 int year;
};
```



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

Structures

- Access structure member variables using "." and "->"
  - . is used with structures
  - -> is used with pointers-to-structures
- Example
  - `current_time` is a variable of type `time`
  - `pcurrent_time` is a pointer to a structure (pointers to structures have special properties which will be useful in accessing arrays of structures)
  - Assign `hour, minute` and `second`
  - Increment `minute`

```
/* Definition */
struct time {
        int hour, minute,
second;
       };
Void main()
{
  struct time current_time;
  struct time *pcurrent_time;

  current_time.hour =  12;
  current_time.minute = 0;
  current_time.second = 0;

  pcurrent_time = &current_time;
  pcurrent_time -> minute++;
};
  /* same as
(*pcurrent_time).minute++ */
```