C for embedded systems

- The coding is basically done with device defaults, global declaration and setup routines.
- Main() is the important function
- Other characteristic of embedded C development:
  - In-line assembly language: use preprocessor directives
  - Device knowledge: compiler-specific extensions with the #pragma preprocessor directive, used most commonly in embedded development to describe specific resources of your target hardware, such as available memory, ports, and specialized instruction sets.
    - Defining ports with #pragma directives
      - #pragma portrw PORTA @ 0x0000: PORT A data register available for reading and writing and is located at address 0x0000.
      - #pragma portw DDRA @ 0x0004: physical PORT A data register available for writing only and is located at 0x0004.
      - Endianness is the attribute of a system that indicates whether integers are represented from left to right or right to left.
  - Mechanical knowledge

- Device-independent data direction settings

```
#pragma portw DDR @ 0x05;

#include <port.h>
/* port.h contains numerous definitions such as the following:

   #define IIIIIIII 0b00000000
   #define IIII0000 0b00001111
   #define 00000000 0b11111111

   where 'O'utput sets DDR bits to one ('1')
   and 'I'nput sets DDR bits to zero ('0').
   They can be regenerated for the opposite settings.
*/

/* ... later ... */

DDR = 00000000; /* all bits set for output */
DDR_WAIT();
/* ... perform write to port ... */
DDR = IIIIIIII; /* all bits set for input */
DDR_WAIT();
/* ... perform read of port ... */
```

- Libraries:
  - It has no main() function.
  - Associated header file should declare the variables and functions as a extern within the libra
  - No dynamic linking.
  - Basic code

```
#include <hc705c8.h>
/* #pragma portrw PORTA @ 0x0A; is declared in header
   #pragma portw  DDRA  @ 0x8A; is declared in header */
#include <port.h>
#define ON 1
#define OFF 0
#define PUSHED 1

void wait(registera); /* wait function prototype, not displayed */

void main(void){
  DDRA = IIIIIII0; /* pin 0 to output, pin 1 to input,
                      rest don't matter */
  while (1){
    if (PORTA.1 == PUSHED){
      wait(1);                  /* is it a valid push? */
      if (PORTA.1 == PUSHED){
       PORTA.0 = ON;               /* turn on light */
        wait(10);                     /* short delay */
        PORTA.0 = OFF;                /* turn off light */
      }
    }
  }
} /* end main */
```

- Data Types

Modifiers	Notes
auto	auto Unnecessary for local variables. Compare with static.
const	Allocates memory in ROM.
extern	Flags the reference for later resolution from within a library.
far	Depends upon addressing scheme of target.
near	Depends upon addressing scheme of target.
signed	Generates extra code compared with unsigned.
static	Preserves local variable across function calls.
unsigned	Creates significant savings in generated code.
volatile	(No specific notes; consult the ISO standard for more information)

- A function data type determines the value that a subroutine can return.
  - Example: int returns a signed integer value.
- Parameter data types: indicate the values to be passed in to the function, and the memory to be reserved for storing them.
- Character data type: char, stores character values and is allocated one byte of memory space.
- Integer data type: int size may be 8-bit or 16-bit – uses sign bit value may be positive or negative.
- Bit data types:
  - Bit-single independent bit.
  - Bits-structure of 8-bit value can be assigned to variable directly and the addresses individually.

- Complex data types
  - Pointers:
    - near: occupy a single byte of memory location-it points to the objects in the bottom section of the addressable memory location (256 location).
      - (eg) int near * myNIntptr;
    - far: occupy two byte of memory location- points to object in ROM user defined function and constant ($0000–$FFFF).
      - (eg) const char * myString = "Constant String";

        char far * myIndex = &myString;
  - Array:
    - Need to declare both array type and number of elements.
      - int myarray[8]; /* uninitialized */
      - int my2array[] ={1,2,4,8,16,32,64,128}; /* initialized below */
      - const int myconsts[] ={1,8,2,7,3,6,4,5}; /* no code generated for const array */

- Enumerated types:
  - They are finite sets of named values (value begins with 0 default).
    - enum DIGITS {one='1', two= '2', three='3'};
    - enum ORDINALS {first = 1, second, third, fourth, fifth};
- Structure: meaningful grouping of program data.

```
struct display {
unsigned int hours;
unsigned int minutes;
unsigned int seconds;
char AorP;
};
struct display timetext;
```

- Union:
  - It creates a scratch pad variable that can hold different types of data's.

```
struct lohi_tag{
        short lowByte;
        short hiByte;
};
union tagName {
       int asInt;
       char asChar;
       short asShort:
       long asLong;
       int near * asNPtr;
       int far * asFPtr;
       struct hilo_tag asWord;
} scratchPad;
```

```
struct asByte {
        int TMR1H; /* high byte */
        int TMR1L; /* low byte */
}
union TIMER1_tag {
      long TMR1_word;  /* access as 16 bit register */
        struct asByte halves;
} TMR1;

/* ... */

seed = TMR1.halves.TMR1L;
```

- TMR1 is made up of two 8-bit registers called TMR1H (high byte) and TMR1L (low byte).
- Uses single block of memory.
- Compiler will align the first bits of each element in the lowest address in the memory block.

- typedef
  - It defines a new variable in terms of the existing type compiler cares most about the size of the new type, to determine the amount of RAM or ROM to reserve.

```
typedef int new_int;
new_int result;  /* represents same range of values
                    in a different context. */


typedef struct {
     char * name;
     int start;
     int min_temp;
     int max_temp;
} time_record;

time_record targets[] {
     { "Night",   0, 20, 25},
     { "Day", 5*3600, 20, 25},
     { "Evening", 18*3600, 20, 25},
}
```

- Data type modifiers
  - This allows us to modify the default characteristics of simple data types, they are applied to data only.
- Value Constancy Modifiers:
  - *const*
    - const float PI = 3.1415926; \ROM space is allocate to the pi value and it does not change\
  - *volatile*
    - Variable that is "stored" at the location of a port data register will change as the port value changes.
- Allowable Values Modifiers:
  - *signed and unsigned*
    - signed keyword forces the compiler to use the high bit of an integer variable as a sign bit.
    - Signed bit=1 rest of the value is negative.
    - signed char mySignedChar;

- Size Modifiers:
  - short and long
    - short int myShortInt; \int size is equal to the char size\
    - short myShortInt; \int short int type\
    - long int myLongInt; \int size twice as that of the int type\
- Storage class modifier:
  - This allocates storage for each identifiers. Types: **extern** (function declaration), static, register, and auto.
  - Complier allocates storage for each identifier, process is called linkage. Types: external, internal, and none.
- Combining Statements in a Block:
  - Control structure: while (condition){statements}
  - Decision structure: if(expression) statement else statement

    result = expr ? result_if_true : result_if_false

- Looping structure:

```
void main(void)
{

  while(1)
    {
     PORTB = PORTB << 1;


    }
}
```

- Bit logical operator:
  - Bitwise AND

    int x=5, y=7, z; /* 5 is binary 101 and 7 is binary 111 */

    z = x & y; /* z gets the value 5 (binary 101) */
  - Bitwise OR

    int x=0b00000101,

    y=0b00000111,

    z;

    z = x | y;/* z gets the value 00000111, or 7 */

- Bit shift operator:
  - x >> number; \Right shifting a binary number by n places is the same as an integer division by 2^n\
  - x << number; \Left shifting a binary number is equivalent to multiplying it by 2^n\
  - porta = 0b10000000;

    while (porta.7 != 1){

    porta >> 1;

    }

    while (porta.0 != 1){

    porta << 1;

    }

- To create a library:
    1. Create a C source file named timestmp.c.
    2. Write in the following lines.

```
#ifndef __TIMESTMP_C
#define __TIMESTMP_C
#pragma library;
#include <timestmp.h>
/* Declared above:
bit use_metric = 0:
char buffer[7];*/
void MinutesToTime( int hours, int minutes )
{
;
}
void TimeToMinutes( int near *hours, int near *minutes )
```

```
{
;
}
#pragma endlibrary;
#endif /* __TIMESTMP_C */
```

3. Create a C header file named timestmp.h.
4. Write in the necessary declarations and prototypes.
5. Compile the C file.

```
void MinutesToTime( int hours, int
minutes )
{
char i;
/* Set up string */
buffer[5] = 'h'; buffer[6] = 0; buffer[2] = ':';
/* Deal with 12-hour time */
if(!use_metric) {
buffer[5] = 'a';
if(hours > 11)
{
hours = hours - 12;
buffer[5] = 'p';
}
if(hours == 0)
{
hours = 12;
}
}
```

```
/* Fill in hours */
buffer[0] = '0';
for(i = '2'; hours >= 10; hours -= 10, i--);
buffer[0] = i;
buffer[1] = hours + '0';
/* Fill in minutes */
buffer[3] = '0';
for(i = '5'; minutes >= 10; minutes -= 10, i-
-);
buffer[3] = i;
buffer[4] = minutes + '0';
}
```

Output:

Buffer	5	4	3	2	1	0
Time	P	5	4	:	1	1

- Time=23:45
    - Buffer[5]=‘a’
      hour>11
      hour=hour-12
      hour=23-12=11
      buffer[5]=‘p’
  - Buffer[0]=‘0’
    - for(i = '2'; hours >= 10; hours -= 10, i--);
      - i=‘2’, hour=1,i=‘1’
      - Buffer[0]=i=1
      - Buffer[1]=hour+’0’=1
  - Buffer[3]=‘0’
    - for(i = '5'; minutes >= 10; minutes -= 10, i--);
      - i=5, minutes=35,i=4
      - i=5, minutes=25,i=4
      - i=5, minutes=15,i=4
      - i=5, minutes=5,i=4
    - buffer[3] = i;=4
    - buffer[4] = minutes + '0';=5