Embedded System Software

C Language &
ARM Assembler

Topics

- Typical Structures in C
  - Low-level Bit Manipulation
  - Control Structures (loops, case statements, etc.)
- State Machine Structure
- Keil RTX Topics

Typical Structures in C

- Headers and C Function Structure
- Low-level Bit Manipulation
  - Bit-level Logic
  - Arithmetic
- Program Control Structure
  - Loops
  - Case Statements

Functions and Headers

- All C Programs are a Collection of One or More Functions
  - can be nested
- Function Returns a Value (unless it is type void)
- Often Used Functions are Available in Header Files (also contain Constants)
  - **`stdio.h`** – Contains I/O Functions
  - **`math.h`** – contains math functions (eg. **`sin`** and **`M_PI`**)
  - **`string.h`** – functions to manipulate strings of **`char`**

C Keywords

auto	else	long	switch
break	enum	register	typedef
case	extern	return	union
char	float	short	unsigned
const	for	signed	void
continue	goto	sizeof	volatile
default	if	static	while
do	int	struct	_Packed
double			

Variable Types in C

char	1 byte	-128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127
int	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
long	4 bytes	-2,147,483,648 to 2,147,483,647
unsigned long	4 bytes	0 to 4,294,967,295
float	4 byte	1.2E-38 to 3.4E+38 6 decimal places
double	8 byte	2.3E-308 to 1.7E+308 15 decimal places
long double	10 byte	3.4E-4932 to 1.1E+4932 19 decimal places

Constants in C

- Values Beginning with **0x** are Hexadecimal
- Values Beginning with **0** are Octal
- Values beginning with **1** through **9** are Decimal

```
212          /* Integer - Legal */
215u         /* Integer - Legal */
0xFeeL       /* Integer - Legal */
078          /* Integer - Illegal: 8 is not an octal digit */
032UU        /* Integer - Illegal: cannot repeat a suffix */
85          /* Integer - decimal */
0213        /* Integer - octal */
0x4b        /* Integer - hexadecimal */
30          /* Integer - int */
30u         /* Integer - unsigned int */
30l         /* Integer - long */
30ul        /* Integer - unsigned long */
3.14159     /* Floating Point - Legal */
314159E-5L /* Floating Point - Legal */
510E        /* Floating Point - Illegal: incomplete exponent */
210f        /* Floating Point - Illegal: no decimal or exponent */
.e55        /* Floating Point - Illegal: missing integer or
fraction */
```


Character Constants in C

- Values Beginning with **0x** are Hexadecimal
- Values Beginning with **0** are Octal
- Values beginning with **1** through **9** are Decimal

```
\\      \ character
\'      ' character
\"      " character
\?      ? character
\a      Alert or bell
\b      Backspace
\f      Form feed
\n      Newline
\r      Carriage return
\t      Horizontal tab
\v      Vertical tab
\ooo    Octal number of one to three digits
\xhh . . .     Hexadecimal number of one or more digits
"z"     ASCII for z char
"This is a string"     String of ASCII chars
```

Increment/Decrement Operators

- Increments/Decrements and Overwrites
- ARM: Memory read, Arith, Memory write

```
int  x;
int  y;

// Increment operators
x = 1;
y = ++x;    // x is now 2, y is also 2
y = x++;    // x is now 3, y is 2

// Decrement operators
x = 3;
y = x--;    // x is now 2, y is 3
y = --x;    // x is now 1, y is also 1
```


Compound Assignment Operators

- Performs Operation and Assignment in One Statement
- Compound Assignment

```
a += b;   //same as a=a+b; addition
a -= b;   //same as a=a-b; subtraction
a *= b;   //same as a=a*b; multiplication
a /= b;   //same as a=a/b; division
a %= b;   //same as a=a%b; modulus (remainder)
a &= b;   //same as a=a&b; bitwise AND
a |= b;   //same as a=a|b; bitwise OR
a ^= b;   //same as a=a^b; bitwise XOR
a <<= b;  //same as a=a<<b; left shift a by b bits
a >>= b;  //same as a=a>>b; right shift a by b bits
```

- Other Operators

```
sizeof(b);     //returns size of b in bytes
a = b ? c : d; //a=c when b is TRUE else a=d
```

Address Operator

- Address Return Operator

```
&b;           //returns address of b
```

- Example

```
#include <stdio.h>
void main (void)
{
   int  var1;
   char var2[10];
   printf("Address of var1 variable: %x\n", &var1  );
   printf("Address of var2 variable: %x\n", &var2  );
}
```

- Output of **main** Function

```
Address of var1 variable: bff5a400
Address of var2 variable: bff5a3f6
```


Pointers

- Variable Whose Value is Address of Another Variable
- Declaration of Pointers

```
int *int_var;    /* pointer to variable of type int */
double *double_var; /* pointer to variable of type
                    /* double */
float *var1;    /* pointer to variable of type float */
char *var1; /* pointer to variable of type char */
```

Pointer Example

```
#include <stdio.h>
void main (void)
{
   int  var = 20;   /* actual variable declaration */
   int  *ip;        /* pointer variable declaration */
   ip = &var;  /* store address of var in pointer
               /* variable*/
   printf("Address of var variable: %x\n", &var  );
   /* address stored in pointer variable */
   printf("Address stored in ip variable: %x\n", ip );
   /* access the value using the pointer */
   printf("Value of *ip variable: %d\n", *ip );
}
```

- Output of **`main`** Function

```
Address of var variable: bffd8b3c
Address stored in ip variable: bffd8b3c
Value of *ip variable: 20
```


Bit-level Operators

- Operators Same as Reduction Operators in Verilog
- Don't Confuse with Logical Operators (**`&&`**, **`||`**, ==, and **`!`**)
  - these evaluate to a Boolean (true or false)
  - used in Conditional Expressions

```
/* Each prepocessor directive defines a single bit */
#define MASK       0x5555  // 32-bit Constant

unsigned int value1, value2;  //32-bit for ARM (Keil-MDK)

value1 = MASK | value2;  // sets even bits to 1 (OR)
value1 = MASK & value2;  // sets odd bits to 0 (AND)
value1 = MASK ^ value2;  // inverts even bits (XOR)
value1 = ~value2;        // inverts all bits (NOT)
```

Bit-level Operators

- Usually Stored in **`unsigned int`** or **`char`** Variables
- Different Systems have Differing Lengths of **`int`**
- **`char`** is Always 8 bits

```
/* Each prepocessor directive defines a single bit */
#define KEY_UP        (1 << 0)   // 000001
#define KEY_RIGHT     (1 << 1)   // 000010
#define KEY_DOWN      (1 << 2)   // 000100
#define KEY_LEFT      (1 << 3)   // 001000
#define KEY_BUTTON1   (1 << 4)   // 010000
#define KEY_BUTTON2   (1 << 5)   // 100000

int gameControllerStatus  = 0;
```


Bit-level Operators (cont)

```
/* Sets the gameCtrollerStatus using OR */
void keyPressed(int key) {
     gameControllerStatus |= key;
}

/* Turns the key in gameControllerStatus off
using AND and ~ */
void keyReleased(int key) {
    gameControllerStatus &= ~key;
}

/* Tests whether a bit is set using AND */
int isPressed(int key) {
    return gameControllerStatus & key;
}
```

`if` Statement

- Conditionally Executes BODY Based on CONDITION
- Delimiters {, } When BODY Consists of 2 or More Statements

```
if (a == b)
     {
     a = !b;
     b = MASK & b;
     }
```

```
if (a == b)
     {
     a = !b;
     }
else
     {
      b = MASK & b;
     }
```


`if else if` Statement

```
if (a == 1)
     {
     a++;
     }
else if (a == 2)
     {
      a--;
     }
else
     {
      a=3;
     }
```

switch Statement

```
switch(expression){
    case constant-expression  :
       statement(s);
       break; /* optional */
    case constant-expression  :
       statement(s);
       break; /* optional */

    /* you can have any number of case statements */
    default : /* Optional */
       statement(s);
}
```


switch Example

```
#include <stdio.h>
 void main (void)
{
   /* local variable definition */
   char grade = 'B';
   switch(grade)
   {
   case 'A' :
      printf("Excellent!\n" );
      break;
   case ('B'||'C') :
      printf("Well done\n" );
      break;
   case 'D' :
      printf("You passed\n" );
      break;
   case 'F' :
      printf("Better try again\n" );
      break;
   default :
      printf("Invalid grade\n" );
   }
   printf("Your grade is  %c\n", grade );
}
```

```
Well done
Your grade is B
```

`for` Statement

- Most Common Loop in C
- Tests Condition BEFORE Body Execution
- Loop Executes When Condition is True

```
void func (void)
{
  unsigned int i, int j=0;
  for (i = 0; i < 100; i++)
    {
    j++;
    }
}
```


`for` Example

- **`device_id`** is the “Name” of an Input Device
- **`sensor_read`** Retrieves the Value from **`device_id`**
- Sensor Values are Accumulated and Returned

```
int acc_func (int device_id)
{
  int sensor_read(int device_id);
  unsigned int i, int sum=0;
  for (i = 0; i < 100; i++)
    {
    sum = sensor_read(device_id)+sum;
    }
  return(sum);
  }
```

`while` Statement

- Tests Condition BEFORE Body Execution
- Possible to Never Execute if Condition is False
- Outputs **`n`** Copies of the Character **`ch`**

```
#include <stdio.h>

void repeat_char(int n,char ch)
{
   while (n--)
     {
     putchar(ch);
     }
}
```


`while` Example

- **`func`** Retrieves String from Input Device
- **`process_string`** Processes Retrieved String
- Loop Exits When **`get_string`** Returns **`NULL`** Pointer

```
#include <stdio.h>
char * get_string(void);
void process_string(char *s);

void func (void)
{
   char *string;
   while ((string = get_string()) != NULL)
      {
      process_string(string);
      }
}
```

`do while` Statement

- Used Less Often than **`for`** or **`while`** Loops
- Tests Condition AFTER Body Execution
- Always Executed at Least Once
- **`transfer_1_line`** reads char from I/P, copies to O/P until newline encountered

```
char input_char(void);
void output_char(char);

void transfer_1_line(void)
{
   char c;

   do {
      c = input_char();
      output_char(c);
   } while (c != '\n');
}
```


`do while` Example

- **`prod_do`** Function "prods" Device Until it is Successfully "prodded" (whatever that means)

```
void prod(int device_id);
int prod_status(int device_id);

/* codes returned by prod_status */
#define PROD_FAIL   -1
#define PROD_OK      0

void prod_do_completion(int device_id)
{
   do {
      prod(device_id);
   } while (prod_status(device_id) != PROD_OK);
}
```

State Machine as Control Program

- State Definition
- Case/Switch Implementation
- Transition Table Implementation
- Polling versus Interrupts
- Timing: HW Timers versus Delay Loops

Embedded System Programming

- Many ES Systems are Event-Driven
  - eg. ES Responds to Input Sensor, Processes an Input Data Stream, then Provides Processed Data to Output Device
- Many ES Systems are Real-Time
  - Must Respond to Input Sensor(s) within a Deadline
  - Must Utilize Priority Structure Among Multiple Input Sensors
- Common Structure for Control Software is in Form of a State Machine

Embedded System Programming

- Many ES Systems are Event-Driven
  - eg. ES Responds to Input Sensor, Processes an Input Data Stream, then Provides Processed Data to Output Device
- Many ES Systems are Real-Time
  - Must Respond to Input Sensor(s) within a Deadline
  - Must Utilize Priority Structure Among Multiple Input Sensors
- Common Structure for Control Software is in Form of a State Machine

Comparison: Application vs. ES Software

- Conventional Application
  - Written in High-Level Programming Language
  - Compiled into Machine Code (static schedule)
  - Loaded and Run
- ES Control Program
  - Written in Low-level Language to Optimize Performance, Power, Memory Footprint
  - Responds to Events in Real-time (dynamic schedule)
  - Operates under Deadline Constraints

State Machine Model

- Deterministic Finite Automaton
  - Mathematical Structure
  - Basis of all Conventional Computation (Turing Machine)
- Known as “State Machine”
- Complex ES Control Software may be Implemented as Hierarchical (nested) State Machines
- Two Main Implementation Structures
  - Program in Form of Case Statement with Each Case Corresponding to a State
  - Program in Form of SM with a “Transition Table”

State Machine Model

- When no Operating System or Monitor is Used, Entry Point is a “RESET” Event that Causes Transition into an Initial State
- ES Program Typically has no “Halting State” (unlike an application) and is an Infinite Loop
- Within Each State,
  - another Lower-level SM may be Invoked
  - Timing Constraints Implemented through use of SW Delay Loops or HW Timers that Interrupt Processor
  - Input Events may be “Sensed” through SW Polling Loop or as Processor Interrupts

HW/SW Partitioning

- SM Model Allows Easy Translation into HDL for Custom HW if Needed
- SM Model Allows for Critical Processing within Each State to be implemented as Interaction with HW Assets or as I/O to Dedicated Hardware
- Typically Implemented in C
  - Design Tools Available that Allow SM to be Specified at Higher-Level (UML, Graphical Input) and then Automatically Synthesized into SM in Form of SW Language
- Use of ES Real-time OS (like Keil RTX) Provides OS Services as Callable Functions

State Machine Example

```
switch(state)
{
  case STATE_1:
     state = DoState1(transition);
     break;
  case STATE_2:
     state = DoState2(transition);
     break;
}
...
DoState2(int transition)
{
   // Do State Work
   ...
   if(transition == FROM_STATE_2) {
     // New state when doing STATE 2 -> STATE 2
   }
   if(transition == FROM_STATE_1) {
    // New State when moving STATE 1 -> STATE 2
   }
   return new_state;
}
```

Inline Assembler

- Performance Critical Functions Require Manual Generation of Assembler
- Can use Inline or Embedded Assembler
- Use __arm In Place of Normal C Statement

```
__asm("ADD x, x, #1\n"
      "MOV y, x\n");
```

- Can Define Macros

```
#define ADDLSL(x, y, shift) __asm ("ADD " #x ", " #y ", LSL " #shift)
```

http://www.keil.com/support/man/docs/armcc/armcc_cihffbgf.htm

Inline Assembler Rules

- Multiple instructions on the same line must be separated with a semicolon (;).
- If an instruction requires more than one line, line continuation must be specified with the backslash character (\).
- For the multiple line format, C and C++ comments are permitted anywhere in the inline assembly language block. However, comments cannot be embedded in a line that contains multiple instructions.
- The comma (,) is used as a separator in assembly language, so C expressions with the comma operator must be enclosed in parentheses to distinguish them:

```
__asm
{
    ADD x, y, (f(), z)
}
```

Inline Assembler Rules

- Labels must be followed by a colon, :, like C and C++ labels.
- An asm statement must be inside a C++ function. An asm statement can be used anywhere a C++ statement is expected.
- Register names in the inline assembler are treated as C or C++ variables. They do not necessarily relate to the physical register of the same name. If the register is not declared as a C or C++ variable, the compiler generates a warning.
- Registers must not be saved and restored in inline assembler. The compiler does this for you. Also, the inline assembler does not provide direct access to the physical registers. However, indirect access is provided through variables that act as virtual registers.

```
__asm
{
    ADD x, y, (f(), z)
}
```


Inline Assembler Register Usage Rules

- Registers such as **r0-r3**, **sp**, **lr**, and the **NZCV** flags in the **CPSR** must be used with caution.
- If C or C++ expressions are used, these might be used as temporary registers and **NZCV** flags might be corrupted by the compiler when evaluating the expression.
- The **pc**, **lr**, and **sp** registers cannot be explicitly read or modified using inline assembly code because there is no direct access to any physical registers.
- The intrinsic functions can be used to read these registers.

```
__current_pc
__current_sp
__return_address
```

Inline Assembler Register Reads

- If registers other than **CPSR** and **SPSR** are read without being written to, an error message is issued.

```
int f(int x)
{
    __asm
    {
        STMFD sp!, {r0} // save r0 - illegal: read before write
        ADD r0, x, 1
        EOR x, r0, x
        LDMFD sp!, {r0} // restore r0 - not needed.
    }
    return x;
}
```


Inline Assembler Register Reads

- Acceptable Way to Implement

```
int f(int x)
{
    int r0;
    __asm
    {
        ADD r0, x, 1
        EOR x, r0, x
    }
    return x;
}
```

Inline Assembler

- Performance Critical Functions Require Manual Generation of Assembler
- Can use Inline or Embedded Assembler

```
// #pragma ARM  // would do the same as __arm below !
void ChangeIRQ (unsigned int NewState) __arm
{  // use ARM-mode for this function
  __asm  {
     AND    R0,R0,#0          //
     MRS    R0,CPSR           //
     ORR    R0,R0,#0x80;      //

     LDAV   R1,R10,NewState   // load parameter-value 'NewState' into R1
     BIC    R0,R0,R1,LSL #7   //
     MSR    CPSR_c, R0        //
  }
}
```


Delay Loop

- Performance Critical Functions Require Manual Generation of Assembler
- Can use Inline or Embedded Assembler

```
void _nop_(){
__asm("mov r0,r0");
}
//--------------------------
//Function Name: delay
//--------------------------
void delay(void) //delay
{
  int i;
  for(i=0;i<=10;i++)
     {
     _nop_();
     }
}
```

Delay Loop (cont)

- Performance Critical Functions Require Manual Generation of Assembler
- Can use Inline or Embedded Assembler

```
void delay10(void)
{
    int i;
    for(i=0;i<=10;i++)
        {
        delay();
        }
}
```