Embedded Software in C for an ARM Cortex M Microcontroller

Jonathan W. Valvano and Ramesh Yerraballi (1/2015)

Legal Statement

Chapter 9: Structures

What's in Chapter 9?

A structure is a collection of variables that share a single name. In an array, each element has the same format. With structures we specify the types and names of each of the elements or members of the structure. The individual members of a structure are referenced by their subname. Therefore, to access data stored in a structure, we must give both the name of the collection and the name of the element. Structures are one of the most powerful features of the C language. In the same way that functions allow us to extend the C language to include new operations, structures provide a mechanism for extending the data types. With structures we can add new data types derived from an aggregate of existing types.

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;
```

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;
```

Similarly, I have also seen the following approach to creating structures that uses the same structure name as the **typedef** name:

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

Most compilers support all of the above methods of declaring and defining structures.

Accessing Members of a Structure

We need to specify both the structure name (name of the variable) and the member name when accessing information stored in a structure. The following examples show accesses to individual members:

```
PortB.mask = 0xFF;     // the TM4C123 has 8 bits on PORTB
PortB.addr = (unsigned long volatile *)(0x400053FC);
PortB.ddr = (unsigned long volatile *)(0x40005400);
PortE.mask = 0x3F;      // the TM4C123 has 6 bits on PORTE
PortE.addr = (unsigned long volatile *)(0x400243FC);
PortE.ddr = (unsigned long volatile *)(0x40024400);
(*PortE.ddr) = 0;               // specify PortE as inputs
(*PortB.addr) = (*PortE.addr);  // copy from PortE to PortB
```

The syntax can get a little complicated when a member of a structure is another structure as illustrated in the next example:

```
struct theline{
   int x1,y1;    // starting point
   int x2,y2;    // starting point
   unsigned char color;}; // color
typedef struct theline line_t;
struct thepath{
   line_t L1,L2;  // two lines
   char direction;};
typedef struct thepath path_t;
path_t p;        // global
void Setp(void){ line_t myLine; path_t q;
   p.L1.x1 = 5;  // black line from 5,6 to 10,12
   p.L1.y1 = 6;
   p.L1.x2 = 10;
   p.L1.y2 = 12;
   p.L1.color = 255;
   p.L2.x1 = 0;  // white line from 0,1 to 2,3
   p.L2.y1 = 1;
   p.L2.x2 = 2;
   p.L2.y2 = 3;
   p.L2.color = 0;
   p.direction = -1;
   myLine = p.L1;
   q = p;
};
```

Listing 9-1: Examples of accessing structures

The local variable declaration **line myLine;** will allocate 9 bytes on the stack while **path q;** will allocate 19 bytes on the stack. In actuality most C compilers in an attempt to maintain addresses as 32-bit aligned numbers will actually allocate 12 and 28 bytes respectively. In particular, the Cortex M executes faster if accesses occur on word-aligned addresses. For example, a 32-bit data access to an odd address requires two bus cycles, while a 32-bit data access to a word-aligned address requires only one bus cycle. Notice that the expression **p.L1.x1** is of the type **int**, the term **p.L1** has the type **line**, while just **p** has the type **path**. The expression **q=p;** will copy the entire 15 bytes that constitute the structure from **p** to **q**.

Initialization of a Structure

Just like any variable, we can specify the initial value of a structure at the time of its definition.

```
path_t thePath={{0,0,5,6,128},{5,6,-10,6,128},1};
line_t theLine={0,0,5,6,128};
port_t PortE={0x3F,
   (unsigned long volatile *)(0x400243FC),
   (unsigned long volatile *)(0x40024400)};
```

If we leave part of the initialization blank it is filled with zeros.

```
path_t thePath={{0,0,5,6,128},};
line_t thePath={5,6,10,12,};
port_t PortE={1,  (unsigned char volatile *)(0x100A),};
```

To place a structure in ROM, we define it as a global constant. In the following example the structure fsm[3] will be allocated and initialized in ROM-space. The linked-structure finite syatem machine is a good example of a ROM-based structure. For more information about finite state machines see either Section 6.5 of Embedded Systems: Introduction to ARM Cortex M Microcontrollers by Jonathan W. Valvano, or Section 3.5 of Embedded Systems: Real-Time Interfacing to ARM Cortex M Microcontrollers by Jonathan W. Valvano.

```
struct State{
    unsigned char Out;     /* Output to Port B */
    unsigned short Wait;   /* Time (62.5ns cycles) to wait */
    unsigned char AndMask[4];
    unsigned char EquMask[4];
    const struct State *Next[4];};  /* Next states */
typedef const struct State state_t;
typedef state_t * StatePtr;
#define stop &fsm[0]
#define turn &fsm[1]
#define bend &fsm[2]
state_t fsm[3]={
{0x34, 16000,   // stop 1 ms
   {0xFF,   0xF0,   0x27,   0x00},
```

```
   {0x51,   0xA0,   0x07,   0x00},
   {turn,   stop,   turn,   bend}},
{0xB3,40000,   // turn 2.5 ms
   {0x80,   0xF0,   0x00,   0x00},
   {0x00,   0x90,   0x00,   0x00},
   {bend,   stop,   turn,   turn}},
{0x75,32000,   // bend 2 ms
   {0xFF,   0x0F,   0x01,   0x00},
   {0x12,   0x05,   0x00,   0x00},
   {stop,   stop,   turn,   stop}}};
```

Listing 9-2: Example of initializing a structure in ROM

Using pointers to access structures

Just like other variables we can use pointers to access information stored in a structure. The syntax is illustrated in the following examples:

```
void Setp(void){ path_t *ppt;
   ppt = &p;         // pointer to an existing global variable
   ppt->L1.x1 = 5;  // black line from 5,6 to 10,12
   ppt->L1.y1 = 6;
   ppt->L1.x2 = 10;
   ppt->L1.y2 = 12;
   ppt->L1.color = 255;
   ppt->L2.x1 = 0;  // white line from 0,1 to 2,3
   ppt->L2.y1 = 1;
   ppt->L2.x2 = 2;
   ppt->L2.y2 = 3;
   ppt->L2.color = 0;
   ppt->direction = -1;
   (*ppt).direction = -1;
};
```

Listing 9-3: Examples of accessing a structure using a pointer

Notice that the syntax **ppt->direction** is equivalent to **(*ppt).direction**. The parentheses in this access are required, because along with () and [], the operators **.** and **->** have the highest precedence and associate from left to right. Therefore ***ppt.direction** would be a syntax error because **ppt.direction** can not be evaluated.

As an another example of pointer access consider the finite state machine controller for the fsm[3] structure shown above. The state machine is illustrated in Figure 9-1, and the program shown in Listing 9-4. There is an example in Chapter 10 that extends this machine to implement function pointers.

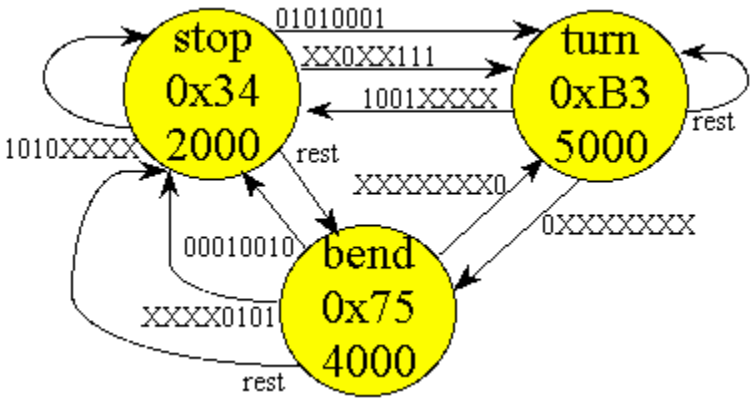

Figure 9-1: State machine

```
void control(void){ StatePtr Pt;
 unsigned char Input; unsigned int i;
  SysTick_Init();
  Port_Init();
  Pt = stop;       // Initial State
  while(1){
    GPIO_PORTA_DATA_R = Pt->Out; // 1) output
    SysTick_Wait(Pt->Wait);      // 2) wait
    Input = GPIO_PORTB_DATA_R;   // 3) input
    for(i=0;i<4;i++)
      if((Input&Pt->AndMask[i])==Pt->EquMask[i]){
        Pt = Pt->Next[i]; // 4) next depends on input
        i=4; }}};
```

Listing 9-4: Finite state machine controller for TM4C123

Passing Structures to Functions

Like any other data type, we can pass structures as parameters to functions. Because most structures occupy a large number of bytes, it makes more sense to pass the structure by reference rather than by value. In the following "call by value" example, the entire 12-byte structure is copied on the stack when the function is called.

```
unsigned long Input(port_t thePort){
   return (*thePort.addr);}
```

When we use "call by reference", a pointer to the structure is passed when the function is called.

```
typedef const struct {
  unsigned char mask;    // defines which bits are active
  unsigned long volatile *addr;      // address
  unsigned long volatile *ddr;}port; // direction reg
```

```
port_t PortE={0x3F,
    (unsigned long volatile *)(0x400243FC),
    (unsigned long volatile *)(0x40024400)};
port_t PortF={0x1F,
   (unsigned long volatile *)(0x400253FC),
   (unsigned long volatile *)(0x40025400)};
int MakeOutput(port_t *ppt){
   (*ppt->ddr) = ppt->mask; // make output
   return 1;}
int MakeInput(port_t *ppt){
   (*ppt->ddr )= 0x00; // make input
   return 1;}
unsigned char Input( port_t *ppt){
 return (*ppt->addr);}
void Output(port_t *ppt, unsigned char data){
 (*ppt->addr) = data;
}
int main(void){ unsigned char MyData;
   MakeInput(&PortE);
   MakeOutput(&PortF);
   Output(&PortF,0);
   MyData=Input(&PortE);
   return 1;}
```

Listing 9-5: Port access organized with a data structure