Should embedded microcontroller developers shun object oriented programming methods? Limited system resources, performance, and the lack of a C++ compiler often dissuade developers from pursuing object oriented development methods. There are, however, developers that are interested in OOP concepts in the microcontroller space. For those that are interested, let’s investigate the possibilities.

One of the common objections to implementing OOP firmware in the microcontroller space is that OOP implementations are resource intensive. Typical implementations use the heap as the object store, and this is indeed expensive in regard to the often limited RAM resource of microcontrollers. The typical OPP implementation also uses pointers to functions and historically, 8 bit micro controllers were not C code efficient. However, 16 bit and 32 bit microcontrollers may utilize RAM and FLASH more efficiently.

The lack of a C++ compiler is another reason that developers may be discouraged from OOP development. This may not be a disadvantage at all. Object Oriented Programming using the C language is not a new concept (1). The GCC based MPLAB C30 compiler allows the developer to place object storage in flash. The compiler directive “const” is used to direct the compiler to place objects in FLASH.

The heap object store can be replaced by compile time constructors. This significantly reduces the RAM usage but there are tradeoffs. It does reduce the flexibility and reusability of object models. Objects that are constructed from other objects must often include knowledge about that objects implementation. This may be an acceptable trade off for a developer. It may not be acceptable for a microcontroller vendor whose goal is to distribute a common object library. But, that concern is outside the scope of this article.

Let’s look at both the typical heap object store (HOS) and the const object store (COS). The objects used in this article are purposefully somewhat simple, but objects need not be simple. A complex object, for instance, may be a NMEA sentence parser that derives from a UART object that derives from FIFO or DMA hardware. I had a spare sample of a PIC 30F3013 and I knew a little boy who wanted a Rocket Computer, hence the name of this project. The sample objects defined in the accompanying source code are oButton, oLED, oTimer, and oOC. The output compare hardware is used as a simple pulse width modulation. The sample code is arranged such that oObjects are heap base object store and ocObjects are const object store.

The heap based object store

The typical HOS uses special “constructors” to allocate sufficient heap storage for the object and this constructor initializes member variables and methods which are pointers to functions. These constructors do consume code space but no more than a general peripheral library function. For example, A LED object is constructed with parameters that specify the port and the pin attachment. The LED object expose methods Get (), Set (), and Toggle () which get the current LED state, set the LED to a specific state, or toggle the state.

The constructor specifies the LAT FSR and the bit associated with the pin.

YellowLed = NewLED(&LATB, BIT0);

GreenLed = NewLED(&LATB, BIT4);

The LED object contains both member data and member methods and these object members are initialized when the constructor is called. The LED object defined in oLED.h is created by allocating heap storage for the object structure. The data member variable mMask is initialized to the bit mask for the pin associated with the hardware port connected to the LED. The pointer to the SFR LAT register address is saved in the data member variable mpLat. The convention is “write to a LAT and read from a PORT”. The member functions are declared static to the module and the object constructor initializes pointers to functions for the Set (), Get (), and Toggle () methods. This object constructor is device specific because it must know about peripherals that share pins and, in the case of the dsPIC30F3013 device, the analog A2D pins share IO pins PORTB. The analog functions must be disabled for the pin to act as a general I/O pin. The implementation also saves code space by relying on the order of the general I/O SFR registers which has been very consistent in the 16 bit family of devices. The LED objects data and members are initialized once in the constructor and do not change while the object is in scope. The size of the heap based object is 10 bytes. This is rather expensive and the heap based object store may be an inefficient use of valuable RAM.

struct OLED \* NewLED(volatile unsigned int \*lat, WORD mask)

{

struct OLED \* This;

This = malloc(sizeof(\*This));

if(This != NULL)

{

This->mMask = mask;

This->mpLat = lat;

This->Set = Set;

This->Get = Get;

This->Toggle = Toggle;

if (lat == &LATB) // only port B has analog pins

ADPCFG |= mask; // make digital

\*(lat -2) &= ~mask; // set the TRIS to configure pin as an output

}

return This;

}

The object’s methods are invoked by dereferencing the objects method and passing the object’s instance to the function. This (no pun) is identical to the way C++ passes the object instance to each member function. The “this” pointer is a hidden parameter in C++ whereas in C, the “this” parameter must be explicitly passed. The “this” pointer is nothing mystical, it is just a pointer to the object, and since the object is a pointer, it passes itself.

YellowLed->Toggle(YellowLed);

All objects of type OLED use this same function to toggle the LED. The “This” pointer points to the OLED object and that specific object, in this case, the YellowLed, contains the member variables that allow the member functions to act upon the pin associated with the LAT as initialized by the OLED constructor.

static void Toggle(struct OLED \*This)

{

\*This->mpLat ^= This->mMask;

}

The const object store

The const object store (COS) will use code space as the object store. The object is forced into code space by a compiler keyword “const” and the object constructor is “called” at compile time. The ocLED.h object has the same size as the heap based object but it resides entirely in flash. It uses no RAM resources. Complex objects may not be completely constructed at compile time due to device specifics. In this case, a special object initialize function may be called to complete the object construction. The ocLED object is forced into code space by the use of the modifier keyword “const”. The modifier is often used as “a promise” when passing parameters to functions. The function promises not to modify the parameter or modify what the parameter points to. The const object store also uses another meaning of the “const” modifier which forces the object structure to be allocated from and reside in flash.

The compile time constructor is.

struct OLED const YellowLed = {BIT0, &LATB, Set, Get, Toggle};

The initialization function need not be member function because it is only called once.

void InitNewLED (& YellowLed);

The object member functions do not need dereferencing and can be called directly. The instance of the object, the “this” pointer, must be passed to the member methods, however, the function prototype must take care to use “const” pointers to avoid compiler errors. The COS object is not a pointer, so the address of the object must be passed to make it the parameter a “this” pointer.

YellowLed.Toggle(&YellowLed);

static void Toggle(struct OLED const \*This)

{

\*This->mpLat ^= This->mMask;

}

A slight more complex object.

The oButton object de-bounces key presses. The object method DeBounce () must be called periodically to determine the state of the button pin and apply the de-bounce algorithm. A timer object is used to periodically call this member function, but let’s save the details of the timer object for later.

The oButton object requires a RAM based de-bounce member state variable. The heap based object store oButton.h object contains a member variable that maintains this state. The const object store must implement some other mechanism to access a variable allocated in RAM because it is not practical to have this state variable reside in flash due to the frequency of and performance of program flash re-writes. The ocButton.h object solves this by using a const member pointer to redirect the object to use a RAM based state variable. It is another level of redirection, but the compiler generates very good code.

The Button object constructors are similar to those of the LED objects. The const object store ocButton requires an initialization function to perform initialization that cannot be accomplished at compile time.

Timer objects

The target device has three hardware timers and all three timers are 16 bit. Timers are complex and this article does not attempt to completely abstract all the timer capabilities. The Timer objects expose some significant differences between the two object store models. Now we begin to see that HOS objects are generally more portable, dynamic, and reusable whereas COS objects still save RAM memory but may imply compile time object inter-dependencies.

Timer objects are constructed similar to the other objects we have discussed. Timer objects expose several methods. The Set () method sets various features like the period FSR, enable or disable the interrupt, set the interrupt priority, and others. The Get () member method returns the value of various FSR. The Execute () method starts or stops the timer.

Objects can register to receive a call back from a timer object. A HOS timer object need not know about the object method being called nor the parameter being passed, it is just a pointer to a function and a function parameter. In reality, the call back is a pointer to an object member function. The function takes one parameter which is the “this” pointer to the object. Storage for the callback function and the “this” parameter are allocated from the heap and each callback is to the callback array based list. When the timer ISR is executed, it inspects the list and calls each callback that has registered.

The COS timer object is similar to the HOS object except that the COS timer callback list is constructed at compile time. Since the COS timer callback list is constructed at compile time, this implies that it must have knowledge about the objects in the list in order to construct the callback list. This leads to an inter-dependency of objects and may make the code base difficult to maintain. The advantage to using a COS timer object is significantly less RAM usage.

Even more complex object

The oOC output compare object is interesting because it is an object that is created from another object, a timer object.

TBD add content.

Summary and cautions

The developer can choose to mix and match both HOS and COS methods in the same application.

The C compiler does not always generate atomic register access. A developer must be aware of potential issues when object methods are called from an ISR. Complex objects may want to disable interrupts, specify interrupt priorities, or implement object spinlocks. These sample objects do not protect the developer from known issues.

Heap based object store

Objects may have global or local scope.

Objects can be destroyed and created.

Allows for dynamic configurations

Requires a heap.

Uses significant RAM for large objects.

Const based object store

Objects are global in scope.

All objects must be created at compile time.

Uses little RAM.

Objects may have compile time interdependencies limiting reuse and portability.

References

(1)

Axel-Tobias Schreiner 1993 [www.planetpdf.com/codecuts/pdfs/ooc.pdf](http://www.planetpdf.com/codecuts/pdfs/ooc.pdf)

(2) microchip docs TBD