Engscope
An Engineer's Life

Non-Blocking Code

What is non-blocking code and why is it so important? First let's look at the "what."

Code is said to "block" when it takes a long time to return in order for other code to execute. This is particularly true in embedded systems where there is usually only a single thread of execution and plenty of while loops.

Suppose you are programming for a sequence of events. The pseudo code below demonstrates the problem with blocking code.

```
//pseudo code for blocking
#include
void main()     {
        //initialize buttons, adc
        initButtons();
        initAdc();

        while(1)        {
                //wait for button to be pressed
                while(!buttonRisingEdge());
                executeButtonPress();

                //if button is pressed, sample adc
                while(!adcIsBusy());
                sampleAdc();
        }
}
```

Now, I'm sure many of you can see right away the problem with this code. There are two things happening in this small segment. The program is in an infinite loop and it waits for a button press. When a button press has occurred, the ADC is sampled. The while loop of the ADC can block the input capture of the buttons, especially if sampleAdc() takes a long time to return. If a button is pressed while sampleAdc() is executing, then a button press will be missed.

So now that blocking code is explained, let's look at why it is important to write non-blocking code. The best explanation of this subject matter that I have heard was from Dr. Miro Samek during his presentation at ESC Boston 2009. He gave the example of a vending machine. Imagine that we are trying to program this vending machine. In the traditional sequential programming approach you might be tempted to write code that looks something like this:

```
void main()     {
        //initialize components in the vending machine
        initCoinDrop();
        initButtons();
        initSodaDispenser();
        initCoinDispenser();

        while(1) {
                //wait for coin to be dropped
                while(!coinDropped());

                //wait for button to be pressed
                //wait for user to make soda selection
                while(!buttonSelected());

                //dispense selected soda
                sodaDispense();

                //release change
                coinDispense();
        }
}
```

We might be able to model the typical vending machine as a series of events that must occur in a certain sequence. First money must be deposited into the machine. This triggers the vending machine into a state that accepts the user's soda selection. The user then must select his soda of choice. Next the machine dispenses the soda, and the change left over from the sale.

The problem with this approach is that this sequence of events must occur at exactly the same order. What if the user inserts one coin, pushes the button to select his soda, then puts in the rest of his coins. Or the user inserts one coin, and decides he doesn't want to spend his money and pushes the void sale button to get his money back. We cannot use while loops to detect all these possible combination of sequences of events and still expect the vending machine to be responsive and bug free. Somewhere along the line, a while loop will block, and when that happens, the behavior of the machine will be unexpected. This style of code might be okay for an alarm clock, but this is NOT okay for a medical device (imagine a pace maker that "blocks"). So how do we get around this? The ultimate solution of course is to use a simple kernel that allows the sharing of the MCU. Now you start to understand why some people are willing to pay over 100 dollars for an consumer level operation system, and upwards of tens of thousands in licensing costs for embedded OS's (although there are free alternatives such as Linux and FreeRTOS). Sharing a single processor in a system is not the least bit trivial. In addition, a proper system should maintain flexibility and responsiveness all the while executing the required tasks. However, I'm not about to endorse, adopt or create a kernel for a hobby level project.

The simple solution to something like this is to break down your code to "poll" rather than "wait". For example, consider the button routine below:

```
//example button polling
//prototypes, macros
```

```
void BtnProcess();
void BtnInit();

//set to 1 if a button Press is detected
int btnPressEvent;

//set to 1 if a button Release is detected
int btnReleaseEvent;

//stores the current state of the button
int btnState;

void main() {
        //initialize components
        BtnInit();

        while(1)        {
                BtnProcess();

                if (btnPressEvent){
                        //do something
                        doSomething();
                }
        }
}

//function initializes registers
void BtnInit()  {
        int btnPressEvent = 0;
        int btnReleaseEvent = 0;
        int btnState = 0;
}

//function polls the IO pin
void BtnProcess()       {
        //reset events
        btnPressEvent = 0;
        btnReleaseEvent = 0;

        //check the previous state, compare to current
        if (btnState){
                //tests true if a button release has occured
                if(!IOPin){
                        btnReleaseEvent = 1;
                        return;
                }
        }else{
                //tests true if a button press has occured
                if(IOPin){
                        btnPressEvent = 1;
                }
        }
        btnState = IOPin        //store state for next process
}
```

In this example the button press routine is separated into two parts. The btnProcess() function processes the input and detects events. The piece of code inside the perpetual while loop polls for button events. Care still must be taken when writing the doSomething() function. The task should be short and return quickly. The advantage of detect separate events).

The next natural step is to take the same principle one step further. Since we only need to poll the buttons every 5-10 ms or so, we can place the btnProcess() function in an interrupt. This way, our btnProcess() routine will not execute very often, but often enough to still detect changes in button events, freeing up processing power.

This type of interrupt driven code is usually more than enough for most hobby level projects, although I would not use it for any professional projects. There are, however, still vulnerabilities in our code. Suppose our doSomething() code takes 100 ms to complete (ridiculously long function). During this time, it is possible to have a button pressed and released, albeit you'll need ninja like reflexes. If we set the interrupt to 10 ms, our btnProcess() routine would have executed 10 times, during which there would have been a btnPressEvent as well as a btnReleaseEvent. We would only be able to detect the release event because the button press event would have been erased from subsequent btnProcess() calls.

How do you get around this? As you can see, the problems associated with coding and events are quickly mounting. The next logical step is to create an event queue as well as an event processor. Each time an event occurs, it is placed in a FIFO queue. When processing power is freed up, the events are processed. In this manner, all the events are accounted for, even if we are in the middle of a long routine. This is the beginning of a "Run to Completion" kernel (RTC Kernel).

To code this software model is another matter. It is beyond the scope of this article. However if you wish to learn more about event-driven programming, I highly recommend Dr. Samek's book Practical UML State Charts In C/C++, ISBN 9780080569789. In the book, a simple but full-featured kernel is presented, along with the source code (open source, available at Quantum Leap).

Happy coding.

2 Responses to *Non-Blocking Code*

Pat *says:*
September 25, 2012 at 4:57 am

Good stuff, thanks very much

Song *says:*
August 2, 2014 at 2:56 pm

Awesome! Thank you.