EVENT DRIVEN PROGRAMMING

In your introductory programming course, you learned the basic program structures, assignments, loops, tests, etc., associated with C. If you go back and look at those programs you will likely see that your program was always in control of when the input occurred. Your code looked for input at specific places in your program, and waited for it to appear. While your program was looking for a particular kind of input, be it a mouse click or a key press, it wasn’t doing any other work. If, at a particular place in your program, you were looking for a mouse click and a key was pressed, generally the key press was ignored (at least at that instant). While this approach to programming is functional in many situations, it doesn’t work very well when there are multiple possible sources of input whose arrival time you can not predict and where it is unacceptable to ignore any of the inputs. Consider the case of a VCR.

On a typical VCR there are a number of buttons, sensor inputs for tape in and the end of the tape, as well as commands that can be received from the remote control. The program for the embedded computer controlling the VCR must be able to respond to any of those inputs at any time and in any sequence. For example, while running fast forward, you can stop, play, change channels, program the VCR to record or any number of other actions. The program can not move to only looking for the channel select buttons while you are programming, or it would miss the end of tape sensor for the rewinding tape.

Clearly, what we need is a program structure that will allow us to respond to a multitude of possible inputs, any of which may arrive at unpredictable times and in an unpredictable sequence. While this problem is very common in embedded systems, like the VCR, it is also very common in modern desktop computers. Think about using a word processor. While you are typing, the program is accepting your keys and entering the characters into your document. You can also click with the mouse at any place on the screen. If the mouse pointer is at another place in your document, the point of insertion moves. If it is in the menu bar, a menu will drop down. If it is over a formatting button, it may change the format of the next character entered. Your word processor, or most other programs running under Windows, MacOS, X-Windows or any other modern Graphical User Interface (GUI), must also be capable of dealing with the same kind of variable input that the VCR must handle. All of these operating systems, and many embedded applications, have adopted a common program structure to deal with the need to respond to many asynchronous (not time sequenced) inputs. This program structure is generally called event driven programming.

Event Driven Programming

Under the event driven programming model, the program structure is divided into two rough groups, Events and Services. An event represents the occurrence of something interesting. A service is what you do in response to the event. While events most often originate from the outside your program, such as mouse clicks, events can also be generated by other parts of the program. The program executes by constantly checking for possible events and, when an event is detected, executing the associated service.

In order for this approach to work, the events must checked continuously and often. This implies that the services must execute quickly, so that the program can get back to checking for events. In order to meet this requirement, the service can not go into a state where it is waiting for some long or indeterminate time. The most common example of this would be a while loop where the condition for termination was not under program control, i.e. a switch closure. This kind of program structure, an indefinite loop, is referred to as ‘Blocking’ code. In order for the event driven programming model to work, you must only write ‘Non-Blocking’ code.

So how do you deal with the need to wait for the switch to close? You make the switch closure another event. Then, rather than waiting in a loop for the switch to close, you program a service that does the right thing when the switch closes and let all of the event checkers run while you are waiting for the switch to close. In this way, you could also react to another event while you were waiting for the switch to close.
Event Checkers

An event, by its nature, represents a discrete point in time. At one instant, the event has not occurred, at the next it has. In the case of a switch, the opening or closing of the switch represents an event. The switch being open or closed does not represent an event. The event is marked by the transition from open to closed.

The event checkers, then, are small pieces of code that test for the occurrence of an event. In order to test for an event when the input is something like a switch, whose value is continuously available, the event checker must have some sense of the past. An event has only occurred when the current value is different from the past value. In pseudo-code the test would look something like:

```
IF switch is closed now AND switch was open last time
THEN SwitchClosed event has occurred.
```

To complete the event checker, the code must update the value kept as the last state of the switch. The variable used for the last state of the switch must be able to retain its value between successive calls to the event checker. In C this might look like:

```
unsigned char CheckSwitchOpen( void)
{
    static unsigned char LastState = OPEN;
    unsigned char CurrentState;
    unsigned char EventStatus;

    CurrentState = GetSwitchState();
    If((CurrentState != LastState) &&
        (CurrentState == OPEN)) /* the event test */
        EventStatus = TRUE;
    else
        EventStatus = FALSE;

    LastState = CurrentState;  /* update state variable */

    return(EventStatus);
}
```

In examining this function, notice that the LastState variable is local, because it is only needed within this function, but it is declared with the static modifier so that it will retain its value between function calls. This is not required for the other two variables, which are also only needed within the function, but do not need to be retained between function calls.

The use of the CurrentState variable also demonstrates an important concept. By using the CurrentState local variable, we test the switch state in only one place. We will need the state of the switch both to test for an event as well as to update the LastState variable. By testing only once and assigning the result to a local variable, we avoid the possibility that the switch state might change between the time that we test it to check for an event and the time that we update the LastState variable. If this were to happen, we would miss an event since the test would fail, but the LastState variable would be updated with a new value.

This basic function outline can be used for most event checkers which test for events that can be represented by two discrete states. Events can also be triggered by variables that can take on a range of values. In that case, the event would be defined as the transition across some threshold value. An example of this might be a light sensor that would generate an event when the light level rose above some threshold.

If you apply this same outline to the light sensor, substituting a test of above/below a threshold for the call...
to GetSwitchState(), you will probably find that you get a string of ON-OFF-ON-OFF events as the light level is raised or lowered. To understand why that might be, take a look at the plot below:

Any variable that represents the state of some real-world (analog) value will have a certain amount of noise associated with it. As the value of the variable approaches the threshold, the noise will alternately take it above, then below the threshold. This will continue until the combination of the signal plus noise stays above the threshold. While the combination of signal plus noise is dancing back and forth across the threshold, a sequence of events is being generated. While this might be desirable in some cases, most of the time we would like to be able to respond to the signal while ignoring the noise.

One approach to this problem would be to go back to the source of the value and attack the noise. This would most likely be done with filtering to reduce the noise level to an undetectable level. This solution requires additional hardware and has the potential to adversely affect the response of the desired signal. There is another approach that allows us to add a small amount of additional software complexity to eliminate the effects of the noise with no additional hardware.

If we modified our test to implement the correct amount of hysteresis at the switch point we could eliminate the multiple events associated with the noise. If the hysteresis band is slightly larger than the amplitude of the noise, we would get a single clean transition as the signal passed through the hysteresis band.
The way to implement hysteresis is to change the threshold from a single fixed value to a variable that can take on one of two different values. The value is initially set at the higher of the two values. As the signal plus noise rises, it will eventually exceed this threshold generating an event and also causing the threshold value to be changed to its lower value. When the falling signal plus noise gets below the new threshold, the off event will be generated and the threshold raised to its higher value. The hysteresis behavior can be encapsulated into a GetLightState function that will implement the hysteresis and create a two valued system like the switch. This will allow us to use the same basic event checking template as we used for the switch.

```c
unsigned char GetLightState( void)
{
    static unsigned char Threshold= HI_THRESHOLD;
    unsigned char LightState;

    if( LightValue >= Threshold) /* above the threshold ? */
    {
        LightState = LIGHT_ON;
        Threshold = LO_THRESHOLD;
    }else     /* must be below */
    {
        LightState = LIGHT_OFF;
        Threshold = HI_THRESHOLD;
    }
    return(LightState);
}
```

Here, again, we use a static local variable to maintain internal information that needs to be preserved between successive calls to the function. In this case it is the threshold value. Whenever the current light value is above the threshold, the light is considered on and the threshold is forced to its lower value. Changing the threshold to the lower value allows the light level to drop somewhat (due to noise) and still be considered on. Only when the light level drops below the new lower threshold will the light be declared off and the threshold raised back to its higher value. Similarly, if the light level then rises slightly (due to noise) it will still be below the new higher value and the state will remain off. It is worthwhile to notice that the LightValue is read only once per call to the function. This prevents any confusion that might be caused by a rapidly changing light value that was sampled twice within a single function call.
Services

Now that we can recognize events, what are services? Services are simply the actions that you want your program to perform when an event has been detected. As an example, think about a mobile robot with bump sensors. The event detector would check for bumping into something, and the service would execute the response. In this case the response would most likely be to stop, or reverse the drive motors and begin to move in another direction.

Services should be very compact functions that initiate the required action and quickly return. This allows the program to get back to checking for other events. The core assumption in event driven programming is that you can check for events quickly enough so that none are missed. This can only happen if both the event checkers and the service routines execute quickly. Neither event checkers or service routines should enter into an indefinite loop.

If you find yourself wanting to code a while loop for something other than stepping through an array, you probably need another event. The way to handle a situation like that is to have a service routine that starts the activity going, and an event checker to detect the end condition. Using the mobile robot as an example, you might want to back up for a second after a bump, in order to move away from the obstacle. You might, improperly, code that as the response to the bump event:

```c
void BumpResponse( void)
{
    unsigned int StartTime = GetTime();
    DriveMotors(REVERSE);
    while((GetTime() – StartTime) < BACKUP_TIME)
        /* wait for backup time to expire */
    DriveMotors(STOP);
}
```

If you did this, you would not be able to respond to any other inputs during the time that you were waiting for the backup time to expire. This would mean that if you hit something while backing up, you would ignore it. The correct approach to this problem would be to write a response routine that started the motors in reverse and also started a timer to run for the backup time. Then, a new event checker could be written to check for when the backup time had expired:

```c
Void BumpResponse( void)
{
    unsigned int StartTime = GetTime();
    DriveMotors(REVERSE);
    StartTimer(BACKUP_TIMER, BACKUP_TIME);
}
unsigned char BackupTimeoutCheck( void)
{
    if(IsTimerExpired(BACKUP_TIMER)
        return(TRUE);
    else
        return(FALSE);
}
```

The response routine for BackupTimeoutCheck() would take care of stopping the motors and initiating any other evasive maneuvers. This approach preserves that ability of the software to perform other checks while it waits for the backup time to expire.

Building an Event Driven Program

The main body of an event driven program will consist of calls to the initialization routines followed by an endless loop to check for events and execute the service routines. While there are software frameworks available to simplify the event checking and calling of the services, they are simply optimizations of a structure that looks something like:

```c
void main(void)
```
{ 
    DoInitializations();
    while(1)
    {
        if(EventChecker1())
            ServiceRoutine1();
        if(EventChecker2())
            ServiceRoutine2();
        .
        .
    }
}

The bulk of your programming effort will be spent on designing and coding the event checkers and service routines. The emphasis on short event checking and service routines has the added benefit of making the design and implementation of each of these routines simpler and less prone to errors. This implies that, with reasonable care, the major problems to be debugged will be at the design level, not at the code, or implementation, level.

By emphasizing the design phase as the place for debugging, we put ourselves in the enviable position of not needing either hardware or software to get started debugging! Since experience tells us that the debugging phase is the longest and most difficult, getting such an early start will likely get us to a working program sooner than if we had tackled the problem without using event driven programming.

**An Example**

You are to design the software for a simple machine that on a small scale mimics the behavior of a cockroach. At the simplest level this means that the cockroach runs straight ahead while in the light and stops when in the dark. If the cockroach encounters an object, it turns left, reverses direction and runs backwards for 3 seconds. If, at the end of that time, it is not in the dark it should begin running straight ahead again.

Identifying the basic events and services is pretty straightforward:

- Light Goes On
- drive forward
- Light Goes Off
- stop
- Contacts Object
- turn left, reverse for 3 sec.

Notice the wording of the event descriptions. An event represents a transition, not a state, and each of these event descriptions is written to capture the transition. The most common mistake made by programmers when beginning to use this methodology is to write event checking routines that actually detect the state of a sensor or input. This results in a continuous stream of 'events' being detected and the corresponding actions executed. In the example above, 'Light Goes On' is an event that happens when the light transitions from off to on, not a state of the light being on.

State machine based designs can be used within an event driven program and this leads to a very powerful methodology that we'll discuss in the next section. However, you need to be careful about what paradigm is active. Events are commonly used as the triggers to move from one state to another within a state machine. Whether or not you are using state machines, you need to be sure that your event checkers are testing for a transition in the variable.

Another interesting aspect is the 'service' associated with contacting an object. How do you handle the 'reverse for 3 sec' service? As in the earlier example, we shouldn't just sit in the service routine for 3 sec. So we need to introduce a new, internal, 'event': Reverse Timer Expired. Adding this new event and a little more thought yields:
Light Goes On
   drive forward
Light Goes Off
   stop
Contact Object
   turn left, drive in reverse, Set ReverseTimer for 3 sec.
Reverse Timer Expires
   turn straight, Simulate Light Goes Off

Now, when the internal event 'Reverse Timer Expires' occurs, the machine should straighten its wheels and simulate a Light Goes Off event. By simulating a Light Goes Off event, if the light is still on, it will trigger a Light Goes On event and begin driving forward. This structure appears to be entirely governed by the events and services and should behave as the specifications demand. To be sure, we can start testing its behavior. No hardware is required, simply the hypothesis for a sequence of events. The designer can, in his/her head or on a sheet of paper, work through how the machine would behave.

In working through this process, it will become clear that there is a possible conflict between the design and the specifications. Consider this sequence:

   Contact Object
   while Reverse Timer is running, Light Goes Off

In the design described above, as soon as the light goes off, the motors will stop. A closer look at the specifications shows that this is may not be the correct behavior.

The specifications state: “If the cockroach encounters an object, it turns left, reverses direction and runs backwards for 3 seconds”. Running backwards for 3 seconds may, if the machine runs into the dark during that time, conflict with the specification that the machine “stops when in the dark”.

The process of evaluating the proposed software design has uncovered an ambiguity in the specifications. Without the emphasis on design that a methodology like this provides, it is likely that this ambiguity would not have been uncovered until the customer actually saw the behavior and said " that's not what I wanted". One of the real strengths of an event driven programming model is that it allows the designer to begin to evaluate and test the program design very early in the process.

**Summary**

The event driven programming model represents a way of thinking about software that lends itself very nicely to situations where there are many possible inputs and the arrival time of those inputs will be unpredictable. It supports the decomposition of the problem into a set of relatively simple event checking and service routines. The behavior of the resulting program is defined by the events that are tested for and the services carried out in response to the events. It is an approach that supports the testing and evaluation of program designs before implementation (coding).

While the event driven programming model alone is capable of dealing with many problems it may fall short when not tackling many of the more complex problems that you may encounter. However, when combined with the concept of state machines, event driven programming is capable of tackling almost any problem, independent of complexity.

Courtesy of J. Edward Carryer, Stanford University