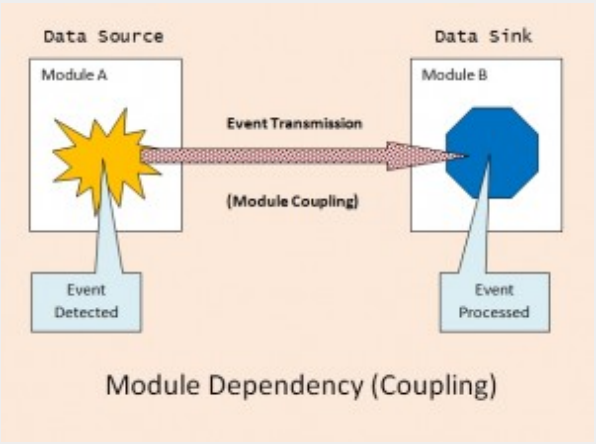
Modules: Pull vs Push

Posted on **August 5, 2012**

A previous posting on modular programming dealt with tight vs loose coupling of modules. In this post a common special case of module interaction is examined. Consider the case where one module (called the source) needs to transmit an event or some data to another module (called the sink). A situation represented in the following illustration:



— Module Dependency

This scenario occurs quite often in embedded systems. Examples include received serial data, completion of an analog data conversion, receipt of a USB command packet, to name just a very few few.

The Source may be an interrupt service routine, a task in a multi-tasking system, or a simple I/O or device library. The Sink can be The Great Loop in simple systems or a task in larger ones. In spite of all this diversity, the approaches taken can be summarized as belonging to one of two basic camps: Pull and Push.

PULL

With the pull strategy, the sink module is responsible for pulling in the data by polling for it. This is simple and popular approach and is typified in the following code snippet:

```
while(1)
{
    if (uartRxRdy())
    {
        nextChar = uartRxRead();
        // Processing of data omitted.
    }

    // The balance of The Great Loop omitted.
}
```

In the above code, the “main loop” code calls the uartRxRdy() function to see if there is any a data to be processed. On the plus side, this code is simple, on the down side, a lot of calls to uartRxRdy() are needed and if the main loop is busy else where, polling can be neglected possibly leading to data loss.

Of course there are more sophisticated versions of this strategy, for example the sink module can employ a task to do the polling, but this does nothing to avoid wasted processor cycles polling for no data. A variant that does have the potential to avoid this waste is to use a blocking call to obtain the data and simply waiting. This approach only yields savings though if the blocking call actually puts the task to sleep when no data is present. If it instead goes into a polling spin loop, the problem is merely hidden rather than solved.

PUSH

With the push strategy, the source module is responsible for pushing out the data by some means. To get the attention of the sink module, there are basically two possible approaches:

1) Modify a variable visible to the sink so that it can be detected. This is such a bad idea; not only is the source meddling with the internals of the sink the sink still has to poll the variable to act on it.

2) Call a function in the sink to process the data or event. This is much better than the above in that it avoids meddling or polling, but there is still a potential problem. When the call to the sink is made, execution is still in the thread of the source. If the source is calling from an interrupt service routine, this places severe restrictions on what may be done in the called routine.

There’s another problem with the function call approach. The source has to know which function to call. This tightens the binding or coupling between the source and the sink. Ideally, the source would require no such knowledge. So, is it possible to call a function without knowing its name? The answer is YES! The mechanism involved is the pointer to a function. Pointers to functions are often avoided or shunned due to the perception that they are complex or difficult to work with. While the syntax in C can be daunting, it need not be so. Consider:

```
//*********************************************************
// Generic function pointer types
//*********************************************************
typedef void (*emPFNV)(void);
typedef void (*emPFNPV)(PV arg);
```

These simple typedefs create definitions for emPFNV a pointer to a function taking no arguments and emPFNPV a pointer to a function taking a pointer to void (called arg) as an argument. Pointers to other types of functions are easily created by cloning and modifying the above templates. So in the source module simply defines a variable to hold the pointer and exports it in its dot H file:

```
extern emPFNV SomeEventDetected;
```

In its initialization, the sink makes the connection with:

```
SomeEventDetected = superEventHandler;
```

and when the event needs to be triggered by the source it simply does:

```
if (SomeEventDetected != NULL)
    SomeEventDetected();
```

This results in the source calling the sink's function (superEventHandler) without placing a code dependency on the source. Note that the source needs to check for the pointer being NULL just in case no sink exists for the event. In this case, an else clause may be added to perform a default action. This is of course entirely optional and if omitted the event or data would simply be ignored in that case.

So far, the scenarios examined have assumed one source and one sink. What if there are many or even an unknown number of sinks? In that case we need to apply management algorithms to a list or array of pointers to functions. This is done by Modules: The Callback Manager and is the topic of a future posting. Conversely, there is the case where there are many or even an unknown number of sources. In this case data needs to be funneled into the sink module. This situation was made famous by the message queue of Windows fame (infamy?) and is the topic of the post Modules: The Message Pump.

As always; remarks, opinions and suggestions are welcomed. Feel free to leave your comments. Some encouragement wouldn't hurt either 😉

Peter Camilleri (aka Squidly Jones)

LIKE THIS:

This entry was posted in **Embedded Development** by **Peter Camilleri**. Bookmark the **permalink [http://teuthida-technologies.com/?p=1054]** .