How to Use Real-Time Multitasking Kernels In Embedded Systems

by Ralph Moore
October 7, 2001

Contents

Micro Digital Associates, Inc.
2900 Bristol Street #G204
Costa Mesa, CA 92626
1-800-366-2491
(714) 437-7333
support@smxinfo.com

Introduction

Various surveys indicate that 40% of the embedded systems projects which *could* use a commercial kernel or RTOS do not use one. From such projects, we hear reasons such as:

(1) "Our application is too simple."
(2) "We don't have time to learn a kernel."
(3) "We decided to fix the old (in-house) one."
(4) "A commercial kernel is too expensive."

While these answers may seem to make sense, they really don't — not if you understand what a commercial kernel can do for you and how to apply it to your project. As one of our customers has said:

> *Using a kernel makes application software simpler to write. Do a function, then wait on a semaphore — no need for complex testing. It can reduce application software up to 50%.*

Surely a tool which can reduce the amount of software you must create and test by up to 50% is worth looking into!

The purpose of this booklet is to familiarize you with the functions provided by a typical commercial kernel and how to utilize them. Once you have a better understanding of this material, the benefits of using a kernel will become apparent.

What is a Kernel?

Most programmers tend to think of a real-time operating system, RTOS, instead of a *real-time multitasking kernel.* What, then, is the difference between an RTOS and a kernel? The distinction between these has blurred in recent years because of increased modularity (AKA *scalability*) of RTOS's and increased complexity of embedded systems which require more services. However, it is still true that an RTOS has a kernel inside of it, and that an RTOS provides additional services such as file i/o, user interface, TCP/IP, etc. *smx*, for example, is a real-time multitasking kernel, whereas SMX is a modular RTOS.

There still are RTOS's, and OS's masquerading as RTOS's (e.g. Linux and embedded NT), which bootload from a disk into RAM, then load the application in .exe form. These RTOS's assume availability of a keyboard and monitor.

A real-time embedded kernel, on the other hand, does not assume the availability of a keyboard, monitor, or disk drive. Instead, it is targeted at black boxes which do not have these amenities and it is assumed to run from ROM. Also, a kernel is normally

linked with the application code to form a single, executable program. Hence, kernels are good for small to medium size embedded systems.

What Services Does a Kernel Provide?

If you are going to use a kernel, it must provide the *services* you need. Typically, real-time, multitasking kernels provide the following services:

(1) task management
(2) intertask communication
(3) memory management
(4) message management
(5) timing
(6) i/o management
(7) error management

Of course, you can write your own code to perform these functions. But *why* should you? Why not build on other people's work? Also you might wonder "Why do I need tasks?" Read on!

What are Tasks, Semaphores, Etc.?

People new to multitasking kernels usually have difficulty grasping what the kernel objects actually are. In this era of object-oriented programming, it is appropriate to invoke the object paradigm: Tasks, semaphores, etc. are *objects*. Each has information and code associated with it. Usually, the information is stored in a *control block*. For example, a semaphore has the following control block:

```
struct SCB   {      // SEMAPHORE CONTROL BLOCK
        CB_PTR      fl          // forward link
        CB_PTR      bl          // backward link
        byte        cbtype      // control block type
        byte        ctr         // signal counter
        word        thres:8     // signal threshold
        word        tplim:7     // task priority limit
        word        tq:1        // task queue present
        };
```

There is a control block, like this, for every semaphore used by the application.

The code associated with an object is the set of *services* provided by the kernel for that object. For example: *signal(sem)* signals a semaphore and *test(sem)* tests a semaphore. The principle of information hiding applies — normally the operator does not directly access or alter control blocks.

Hence, a multitasking kernel provides an object-oriented environment for embedded applications. This environment consists of objects such as tasks and semaphores and

services such as signal() and test() provided by the kernel. The multitasking paradigm requires the programmer to view his application as a collection of interacting objects rather than as a sequence of operations or as a state machine. This can be a difficult adjustment to make and may be a reason why multitasking kernels are frequently rejected or misused.

Experience has shown the object model to be a better model for most embedded systems. The other models simply do not deal well with the complexities of multiple, simultaneous events which are typical in modern embedded systems. Flow charts are good for describing sequential processes. State machines are good if there a small number of possible states with well-defined transition rules. But, neither is good for describing complex systems with many interdependent parts. Multitasking, on the other hand, is ideal for such systems — just define a task to handle each part. Then define how the parts interact.

A significant weakness of the sequential process and the state machine approaches is that they are inflexible. A good programmer can initially create a workable solution using these approaches. But requirements invariably change, and the workable design eventually turns into *spaghetti code.* In times past, this was a problem primarily in the later stages of product life. However, because of the current rapid pace of high tech markets, this result is frequently occurring before first delivery can even be made. This creates serious consequences for time to market and success of the product.

Multitasking fosters code that is structured so that it can grow and change easily. Changes are accomplished merely by adding, deleting, or changing tasks, while leaving other tasks unchanged. Since the code is compartmentalized into tasks, propagation of changes through the code is minimized. Hence, multitasking provides a flexibility much needed by modern embedded systems.

Why Tasks?

Breaking a large job into smaller tasks and then performing the tasks one by one is a technique we all use in our daily lives. For example, to build a fence, we first set the posts, then attach the 2x4's, nail on the slats, then paint the fence. Although these operations must be done in order, it is not necessary to complete one operation before starting another. If desirable, we might set a few posts, then start the next task, and so on. This divide and conquer approach is equally applicable to writing embedded systems software. A multitasking kernel takes this one step further by allowing the final embedded system software to actually run as multiple tasks. This has several advantages:

(1) Small tasks are easier to code, debug, and fix than is a monolithic block of software, which, typically, must be completely designed and coded before testing can begin.

(2) A multitasking kernel provides a well defined interface between functions that are implemented as independent tasks, thus minimizing hidden dependencies between them.
(3) The uniformity provided by kernel services and interfaces is especially important if tasks are created by different programmers.
(4) A preemptive multitasking kernel allows tasks handling urgent events to interrupt less urgent tasks. (Such as when the phone rings while you are watching TV.)
(5) New features can easily be added by adding new tasks

Basically, a preemptive, multitasking environment is compatible with the way embedded software is created and is a natural environment for the same software to run in. Let's consider an example: Suppose we need to control an engine using several measured parameters and a complex control algorithm. Also, assume there is an operator interface which displays information and allows operator control. Finally, assume that the system must communicate with a remote host computer. Clearly there are at least three major functions:

(1) engine control
(2) operator interface
(3) host interface

So the system basically looks like this:

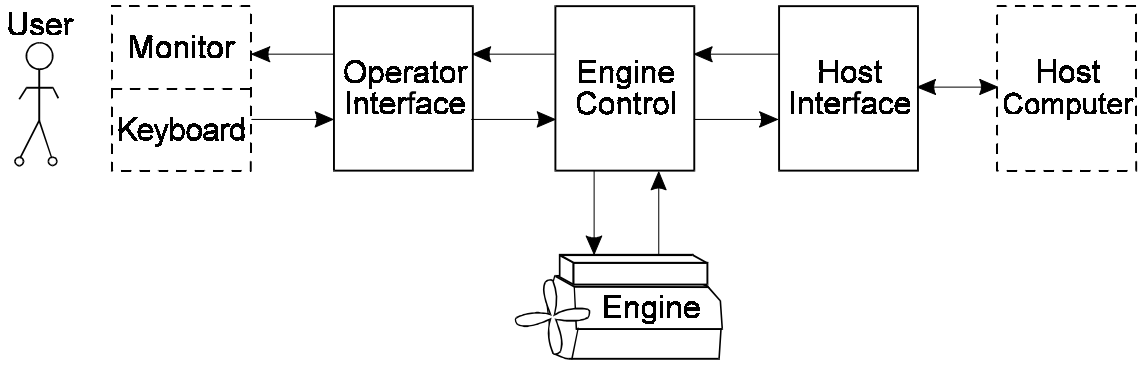


Figure 1 Major Functions for Engine Control System

Each of the above is sufficiently complex, that it is necessary to work on it individually. *Wouldn't it be nice if an environment already existed so that the three functions could be created and operated independently?* This can be done by using a multitasking kernel and by making each function a task.

Note that the tasks are not of equal urgency: The operator can be kept waiting for long periods of time relative to microprocessor speeds, but the engine control task may need to respond quickly to input changes in order to maintain smooth engine

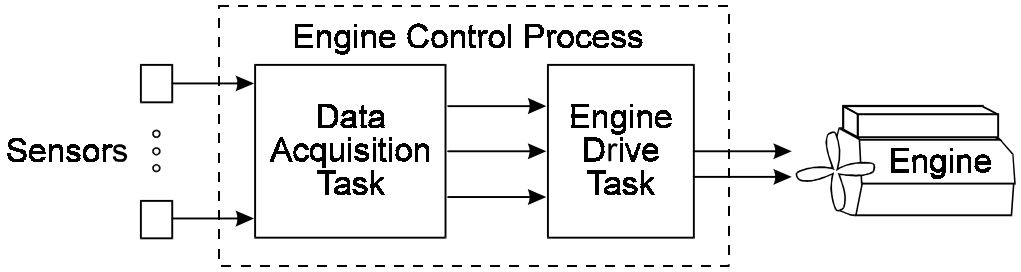
operation. The host probably falls somewhere in between in urgency. With a preemptive multitasking kernel, this can be easily accomplished merely by giving the engine control task a higher priority than the other two tasks. The host task requires an in-between priority to do its job well, and the operator task can operate satisfactorily at low priority.

Many embedded software projects reach this point in the design and still do not pick a commercial multitasking kernel. The reason most often given is: "Our application is too simple to need a kernel." This is often a big mistake. There are many hidden complexities in the above diagram. In the remainder of this booklet, you will learn how a commercial kernel can deal with these complexities more effectively than can ad hoc code.

In this regard, it is important to recognize that a commercial kernel has already been used in a large variety of projects. Hence, many potential problems which may occur in your project have already been anticipated and solved. Ad hoc code, by contrast, deals with problems as they arise. It is created without careful planning, and usually fails to provide general solutions. Also, a commercial kernel contains tested and proven code. *This is of utmost importance when meeting a tight schedule.*

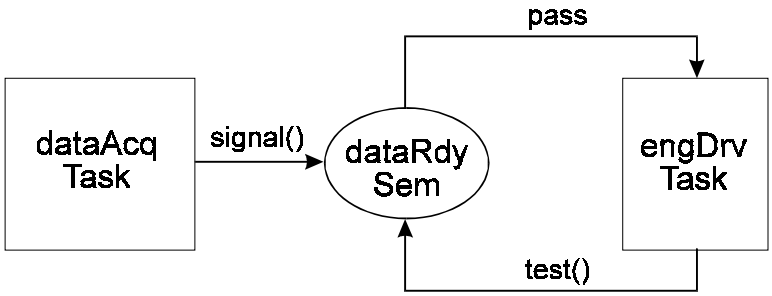
Using Semaphores

Continuing our example, it would be logical to divide the engine control "task" into two smaller tasks. Hence it becomes a "process":



The data acquisition task reads the sensors, converts readings to engineering units, and compensates for non-linearities, temperature changes, etc. The engine drive task performs complex control calculations (e.g. PID) and provides the final engine drive signals.

The above scheme looks workable, but how does the engine drive task know when its data is ready? A simple way to handle this is with a semaphore:



The dataAcq task signals the dataRdy semaphore when data is ready. This causes the engDrv task to run once. Then, engDrv tests dataRdy again for the next signal from dataAcq. If there is no signal, it waits. Hence, engDrv is regulated by dataAcq, as we desire. The code would look like this:

```
void dataAcqMain(void)
{
        // initialize dataAcq task
        while(1)   // infinite loop
        {
                // acquire & convert data
                signalx(dataRdy);
        }
}

void engDrvMain(void)
{
        // initialize engDrv task
        while (test(dataRdy, INF))
        {
                // perform control algorithm and output drive signals
        }
}
```

The above would probably work fine with a simple *binary* (two state) *semaphore*. Suppose, however, that the engine control algorithm is so sensitive (or that the data is so noisy) that it is necessary to smooth the data by running the data acquisition task more often than the engine drive task, and averaging results? This could easily be accomplished with a *counting semaphore* having a *threshold* of the desired number of iterations. A *counting semaphore* is incremented by each signalx(). test() passes only when the count reaches the threshold. Then the count is reset to 0. dataRdy's threshold can be externally changed by another task such as the operator task. This permits tweaking responsiveness vs. smoothness while actually running the engine.

What other benefits could accrue from dividing the engine control process into dataAcq and engDrv tasks? Suppose, for example, that all engines use the same control algorithm, but that sensors vary from engine to engine. Then it would be

desirable to have a family of dataAcq functions (e.g. dataAcq1Main(), dataAcq2Main(), etc.) and be able to select the one needed. Why would someone want to do this? Suppose that the sensor package is part of the engine and hence is known only when the controller is mated to the engine. At that time, the correct dataAcq task function could be selected and started by the operator. This way, only one version of the controller software need be shipped. The code would look like this:

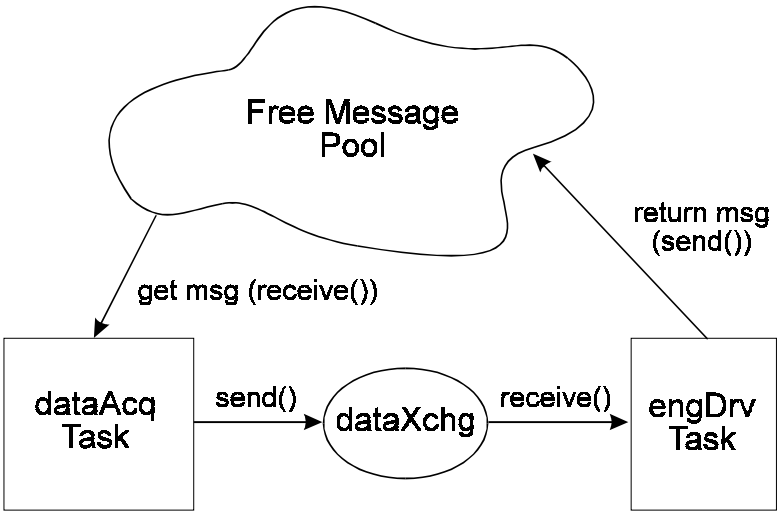
```
switch (sensorType)  // provided by operator
{
        case 1:
                dataAcq = create_task(dataAcq1Main, NORM, 0);
        case 2:
                dataAcq = create_task(dataAcq 2 Main, NORM, 0);
        ...
}
startx(dataAcq)
```

Using Messages

A *message* is a block of data meant to be sent to another task. It is managed with a *message control block.*

Returning to the engine control process, how is data passed from the dataAcq task to the engDrv task? In a multitasking system, it is desirable to isolate tasks from each other as much as possible. Therefore, it is *not good practice to pass data through a global buffer*. Such a buffer would be accessible to both tasks simultaneously. Hence, the data could be overwritten by the dataAcq task before the engDrv task was done using it. This is an example of a *hidden interdependency.*

The preferred approach is to use *messages*. Messages are sent by tasks to *exchanges* and received from exchanges by other tasks. For our example, the process looks like this:



The code would look like this: (This code is additional to the previous code.)

```
void dataAcqMain(void)
{
        MCB_PTR  msgOut;                // message handle
        ENG_DATA  struct *outPtr;       // data template pointer

        // initialize
        while (1)
        {
                // acquire & convert data
                msgOut = receive(msgPool, INF);  // get a free message
                outPtr = msgOut→mp;              // get its pointer
                outPtr→field1 = ...;             // load data into it
                outPtr→field2 = ...;
                ...
                sendx(msgOut, dataXchg)          // send it
        }
}

void engDrvMain(void)
{
        MCB_PTR  msgIn;
        ENG_DATA  struct  *inPtr;

        // initialize
        while (msgIn = receive(dataXchg, INF))   // receive message
        {
                inPtr = msgIn→mp;                // get its pointer
                ... = inPtr→field1 ...;          // process it
                ...
                sendx(msgIn, msgPool);           // return used msg to free pool
        }
}
```

So, basically, dataAcq gets a free message, fills it with data and sends it to dataXchg. Some time later, engDrv gets the message from dataXchg, processes the data, then recycles the (now "empty") message back to the free message pool. Note that each task has *exclusive* access to the data in the message while the message is within the task's domain. This eliminates one possible problem. Another advantage of this scheme is that the tasks are not forced to be in lock step with each other. A message may or may not be waiting at dataXchg when engDrv attempts to receive. If not, engDrv waits. Conversely engDrv may not be waiting at dataXchg when dataAcq sends a message to dataXchg. If not, the message waits. Hence, there is flexibility in the system. This makes it *less vulnerable to breaking under stress.*

In fact, dataAcq can send many messages to dataXchg and they simply will be queued up in the order received. engDrv will process them when it is allowed to run. This is how the previously suggested averaging over many samples (i.e. messages) would be implemented. The code would look like this:

```
int  i;
while (test(dataRdy, INF))
{
        i = 0;
        while (msg = receive(dataXchg, NO_WAIT))
        {
                if (msg)
                {
                        i++
                        // add message data to buffer
                        sendx(msg, msgPool);
                }
                else
                        break;
        }
        // divide buffer by i
        // perform control algorithm and output drive signals
}
```

Observe that the threshold of the dataRdy semaphore (see previous section) controls the number of samples per average. How about that for a nifty implementation? Note also that if there were no need to perform averaging, then the dataRdy semaphore would be superfluous — the dataXchg exchange, alone, would be sufficient to regulate the engDrv task.

Timeouts

What happens when things go wrong? For example, events do not occur? A good kernel protects against errors. In the previous engDrvMain() example, the statement,

```
msgIn = receive(dataXchg, INF);
```

means to wait forever (INF) at dataXchg for input data from dataAcq. This is not good. What if dataAcq croaks? Then the engine is going to be in trouble! Better code would be:

```
if (msgIn = receive(dataXchg, ONE_SEC))
        // process msgIn
else
        // sound alarm
```

If the one second timeout (ONE_SEC) occurs, receive() returns with nothing (0). Hence engDrv can tell there is a problem and do something about it. (We are assuming that data should be acquired more often than once per second.)

Sounding the alarm provides another interesting example of using messages. For this, define a task called alarmMgr in the operator process and define an alarm exchange called X911:

```
TCB_PTR  alarmMgr;      // task handle
XCB_PTR  X911;          // exchange handle

void alarmMgrMain(void)
{
        MCB_PTR  errMsg;
        // initialize alarm system
        while (errMsg = receive(X911, INF))
        {
                // process errMsg
        }
}
```

Then, in engDrvMain():

```
MCB_PTR  errMSG;
errMsg = create_hmsg(NULL, 15);
load(errMsg, “NO SENSOR DATA”);
sendx(errMsg, X911);
```

Note how this example takes advantage of the anonymity of the error processing task — all that needs to be known is to send error messages to X911. This could be advantageous by allowing two versions of alarmMgr: (1) The PC version which outputs error messages to the screen, and (2) the target version which outputs error messages to a serial port or saves them in an error buffer.

Also, note that the error message block is taken from the heap (create_hmsg()). Although slower than receiving it from a message pool, this avoids exhausting the pool if there are many errors at once (a likely occurrence). Also the message block can be exactly the right size for the message (e.g. 15 characters, including NULL termination). (Message pools have uniform-length messages.) Further note that X911 does not care what size the message is. (Of course, alarmMgr must be able to handle variable-size messages.) Also, it is appropriate for alarmMgr to wait forever at X911 because errors are unplanned events.

More to come…

About The Author

Ralph Moore graduated in physics from Caltech in 1962. He did computer research at Honeywell then went to work for Scientific Data Systems in 1965 as a logic designer. He started his own consulting business, Micro Digital Associates, in 1975 doing embedded systems (except no one called them that, then). At that time the 2MHz 8080 cost over $300 in 100 quantity! Over the years the work mix changed steadily from hardware to software. In 1988, Mr. Moore designed *smx*, a real-time multitasking kernel, and took Micro Digital into the software product business. Today he continues as chief architect and president of Micro Digital, Inc.