

Using priority functions and priority objects to enable a compiler to handle real-time applications greatly simplifies the programmer's job.

# How to Augment a Compiler with Real-Time Capability

by Rabih Chrabieh

Adding real-time capability to a compiler or interpreter is hard to achieve with traditional tasks. However, you can do it in a straightforward manner with priority functions and priority objects, assigning high priority levels to time-critical routines so that they preempt low-priority (time-insensitive) routines.

And with an external library providing a scheduler, a memory manager, a timer manager, and a signal manager, writing real-time software, in simulation or in an embedded environment, becomes remarkably simpler.

You can turn simulation code relatively quickly into real-time embedded software. There's no need to pack function arguments inside message structures or to send messages between tasks. The code can be quickly converted as is by adding a few directives to define the priority functions and their priority levels. The programmer focuses only on how to make the design clean and robust by avoiding undesirable interference among

different priority levels.

In order to execute routines at different priority levels, many DSPs and RTOSs—Texas Instruments' TMS320 DSP family and DSP/BIOS kernel, for example—provide three mechanisms: hardware interrupts, software interrupts, and tasks. Other RTOSs denominate tasks as threads, software interrupts as link service routines and such.

Hardware interrupts have the highest priority levels and are used only for very time-sensitive code so as to avoid blocking new hardware interrupts. Typically, they send signals to routines running inside software interrupts and tasks.

Software interrupts and tasks execute routines at various priority levels, which have been assigned and can sometimes be modified. In the case of DSP/BIOS, software interrupts always have a higher priority than tasks. (Note that that's a choice the designers of the kernel made; it doesn't have to be that way.) Routines running inside a software interrupt or a task execute at the

## It's worthwhile to generalize software interrupts to encompass any function and designate them priority functions—indeed, this designation is more appropriate for compilers.

software interrupt's or the task's assigned priority level.

The main difference between a software interrupt and a task is that the software interrupt can't stop and wait for a signal. It has to terminate and return. A task, on the other hand, possesses a dedicated stack that permits it to stop and wait for new events. The task's state is saved on the stack before the task is suspended and restored before it resumes.

A programmer writing code for a TI DSP is often confronted with the question, should I use a software interrupt or a task for this or that job? It's always possible to write the same piece of code using software interrupts or tasks. Often the programmer chooses to use a task, since it has the flexibility to stop and wait. This flexibility is particularly handy for protecting shared data by wrapping the access to the data with a protective semaphore. The semaphore ensures that the task stops and waits in case another task is already engaged in accessing the same shared data.

Despite their flexibility to stop and wait, however, tasks have serious drawbacks. A task owns a dedicated stack, mailboxes, semaphores, and the like, and for that reason typically contains subtasks. As a result, not only are intertask messages necessary, but also intratask messages, which in turn require intratask message dispatchers. All that complicates a real-time program considerably, reducing its modularity and diminishing its upgradability. Furthermore, context switching slows performance.

Therefore it's preferable to base most or all of a program on software interrupts. Furthermore, it's worthwhile to generalize software interrupts to encompass any function and designate them priority functions—indeed, this designation is more appropriate for compilers. Instead of defining software interrupts or tasks and executing functions within their context, you directly assign the priority level to a function and execute it outside any context.

In addition, not only can a compiler automatically handle priority function calls; it can handle packing, unpacking, and saving function arguments, as well.

Packing, unpacking, and saving of function arguments occur when the priority function call is postponed because a higher-priority function is already running.

Briefly, a priority function is any function that has been assigned a priority level. A scheduler ensures that a priority function executes at the appropriate priority level; in other words, it preempts lower-priority functions and waits for higher-priority functions to terminate. The concept of a priority object is adopted from object-oriented programming. In this case, the priority is assigned to the object, and the methods or functions that handle the object inherit its priority level.

The compilers or interpreters that can be augmented with these concepts include those for C, C++, Java, Python, and Matlab (Modula2).

### DECLARING PRIORITY FUNCTIONS

You can declare a function, *F*, a priority function with priority *P* by using a compiler directive. The directive tells the compiler that the function should be called in a special way. It can be written, for example, at the start of the function. It can be something like:

```
void F(...)
{
    _priority_(P);
    ...
}
```

Here we opt for the notation `_priority_(P)`, but there are other possibilities, such as:

```
SET_PRIORITY P
PRIORITY_FUNCTION P
#pragma priority P
#pragma priority P
priority_function(P);
```

In the case of C or C++, it is sometimes useful to write the compiler directive preceding or following the

function prototype in the .h header file. This approach allows the compiler to generate more optimal code, since it has more information on the priority level—for example,

```
_priority_ (P) F(...);
```

You call a priority function the same way you would call a normal function; the compiler does the rest of the work. There's no need to save the function's parameters in a message. Everything happens seamlessly. From the programmer's point of view, calling  $F$  looks like  $F(\dots)$ .

A compiler converts a function,  $F$ , into a priority function by creating some wrapping code with new entry points. The entry points can be defined thus:

- Entry point  $F_0$ : the old entry point of the function
- Entry point  $F_n$ : the new entry point of the function
- Entry point  $F_s$ : the entry point when the function is called from the scheduler.

When you call  $F$ , the new entry point,  $F_n$ , is called and performs the following steps:

1. It compares the priority level  $P$  of  $F$  and the current priority level,  $P_{\text{current}}$ .
2. If  $P \geq P_{\text{current}}$ , call  $F$  immediately by jumping to the old entry point,  $F_0$ . When  $F$  is done it returns to the calling code as usual.
3. If  $P \leq P_{\text{current}}$ :
  - (a) Save the arguments of  $F$  in a memory block,  $M$ , obtained from the memory manager.
  - (b) Call the scheduler with a pointer to the entry point  $F_s$ ; a pointer to the memory block,  $M$ ; and the value of the function's priority level,  $P$ . The scheduler doesn't call  $F$ . Rather it stores the call in a database. Thus the function call is postponed.
  - (c) Return to calling code without calling  $F$ .

A postponed function call is executed later, when its priority level becomes current, that is, when  $P > P_{\text{current}}$ . \* At this point, the scheduler calls the func-

tion via entry point  $F_s$ . The scheduler also supplies a pointer to memory block  $M$ . Entry point  $F_s$  performs the following steps:

1. Restores the function's arguments from memory block  $M$ .
2. Frees memory block  $M$  by returning it to the memory manager.
3. Calls the function  $F$  by jumping to entry point  $F_0$  (for faster response, this step can be called before step 2).
4. Returns to the caller—that is, the scheduler—when  $F$  terminates. The scheduler can then call other, lower-priority functions.

Some of the steps mentioned in this section can be in-lined and optimized when the compiler knows (at compile time) the priority level,  $P$ , and the current priority level,  $P_{\text{current}}$ , of the calling function. For example, if  $F$  is being called from function  $F_1$  and if the respective priority levels are known and are  $P \geq P_1$ , the compiler can generate a direct call from  $F_1$  to the old entry point,  $F_0$ . In that way, zero overhead is incurred when calling priority function  $F$ .

Note that priority functions normally execute in a context that has a lower priority than the context of hardware interrupts. Consequently, priority functions called from hardware interrupts are always postponed.

## SCHEDULER

The scheduler, which is independent of the compiler, handles the calls to priority functions that haven't been called immediately, that is, postponed priority functions. The call to such priority functions is stored in an optimized database maintained by the scheduler.

The compiler must call the scheduler in order to store calls to postponed priority functions. Therefore

---

\*When a user calls a function, if  $P = P_{\text{current}}$  the function is executed immediately, whereas normally if  $P = P_{\text{current}}$  the task is postponed.; this special treatment for functions permits a layer of functions at the same priority level to call each other sequentially, i.e., without any postponing. When a scheduler calls a postponed function, the function receives no special treatment and is called at the appropriate priority level order, which is denoted by  $P = P_{\text{current}}$ ; in other words, if  $P = P_{\text{current}}$  we proceed with  $P_{\text{current}}$ .

it needs the symbol name of the scheduler's function, which can be either a standard symbol name or a specific name supplied with a compiler directive, such as:

```
scheduler_(symbol_name);
```

In C, you can provide the symbol name in the compiler's command line arguments for example:

```
D_SCHEDULER_=symbol_name.
```

The compiler also needs to know how to pass arguments to the scheduler. The method can be either standardized or, again, specified with a directive.

## MEMORY MANAGER

A memory manager is an efficient dynamic memory handler. It is equivalent to C's well-known `malloc` and `free` functions. However, standard `malloc` and `free` may be too inefficient; higher-performance versions may be used instead. Like the scheduler, the memory manager is independent of the compiler.

For postponed priority functions, the compiler obtains a memory block, `M`, from the memory manager. Accordingly, it needs to know the symbol names of the memory manager functions, which, as with the scheduler, can be either standard symbol names or specific names supplied by compiler directives—for example, in this case:

```
_memory_malloc_(symbol_name_malloc);  
_memory_free_(symbol_name_free);
```

Note that the size of the memory block, `M`, is most often known at compile time. Therefore the compiler can access a highly efficient version of `malloc` that works on a known size at compile time.

A function that returns a value to the caller or, more generally, that changes a state that affects the caller can't simply be converted into a priority function. If its priority level is higher than the caller's priority level, the function can be directly converted into a priority function. However, if its priority level is lower than the caller's, the function call is postponed. Consequently, the state change doesn't occur as the caller expects. To solve

## Listing 1: Example of a Dynamic Priority Level in C

```
/* C example of a dynamic priority level stored inside  
 * a modem's Layer 1 data structure.  
 */  
struct Layer1 {  
    int priority;  
    ...  
} L1;  
  
/* L1 Transmit function. The priority level is dynamic,  
 * stored in the argument struct l1  
 */  
void l1_transmit(struct Layer1 *l1) {  
    _priority_(l1->priority);  
    ...  
}
```

this problem, you can divide the function into two or more subfunctions, some running at low priority, some at high priority.

Also, shared data may be accessed by low- and high-priority code. The shared data could be trashed if unprotected. When programming with priority functions, you can't use semaphores. Two mechanisms, however, are available for protecting shared data:

*Priority ceiling:* The priority level is raised over the critical section. Raising the priority level is a feature provided by the scheduler, and the compiler doesn't need to be aware of it except for special optimizations.

*Dedicated priority functions:* Shared data is accessed exclusively via dedicated priority functions running at a fixed priority level. The fixed priority level ensures data coherency. It implicitly acts like a queue of requests to modify the data. If the results are expected to occur right away, the priority level of the dedicated priority functions should be sufficiently high, like a priority ceiling. Note that this mechanism encourages you to write clean, object-oriented-like code. The mechanism is unfeasible or too costly when programming with tasks.

## DYNAMIC PRIORITY LEVELS

The priority level of a priority function can be dynamic: it can be stored in a variable or a data object. Listing 1 gives an example in C, and Listing 2 an example in C++. In the case of C++, the compiler adds a field in the object's structure that contains the priority level. When an object's method is called, the compiler automatically handles it just as it handles priority functions except that the priority level is dynamically obtained from the object's structure. You can

## Listing 2: Example of a Dynamic Priority Level in C++

```
class Layer1 {
    _priority_;
    ...
};

// C++ object instantiation with variable priority P.
// The methods in the class inherit priority value P.
class Layer1 *L1 = new Layer1(...) _priority_(P);
```

value. Thus it can generate more optimized code. Specifying a range can be done with a directive, such as:

```
// Specify a dynamic
//priority level in the
//range 10 to 20
class Layer1 {
    _priority_ [10, 20];
    ...
};
```

give a C++ class several priority levels. Listing 3 gives an example.

To obtain better in-lining and optimizations, you can specify the range of priority values that an object can take. Hence, even if the priority value is unknown at compile time, the compiler can tell that it is above or below some

## TIMER MANAGER AND SIGNAL MANAGER

A timer manager can handle time events independent of the compiler. The timer manager works with priority functions. You can schedule a priority function to run at a specific time in the future, and the timer

>  
powerful  
connections



## Network with the SBC6713e, our highest-performance stand-alone DSP board with on-board 10/100 Ethernet

### Features

- ▶ **300MH , 32-bit, floating-point TMS320C6713 DSP**  
Powerful C/C++ libraries, Windows debugger
- ▶ **10/100 Ethernet via DM642 coprocessor for real-time network I/O**
- ▶ **Two OMNIBUS I/O expansion sites**  
Wide selection of analog input/output Up to 24-bit resolution & 64 MHz sample rate
- ▶ **Capable of 100% stand-alone operation**  
3U Size 100mm x 160mm 4 Mb Flash ROM
- ▶ **fantastic, on-board peripherals**  
RS232, 32-bit digital I/O, watchdog 600K gate Spartan-IIe for user-logic (optional)

### Applications

- ▶ Embedded Servo Control
- ▶ Remote Data Acquisition
- ▶ Industrial Test & Measurement
- ▶ OEM Instrumentation

Download  
Data Sheets  
NOW!

805.520.3300 phone  
[www.innovative-dsp.com](http://www.innovative-dsp.com)

**Innovative  
Integration**  
... real time solutions!

## Listing 3: Example of a C++ Class with Several Priority Levels

```
class Layer1 {
    _priority_(P1, P2); // Unspecified values for now
    ...

    function1() {
        _priority_(P1);
        ...
    }

    function2() {
        _priority_(P2);
        ...
    }
};

Layer1 *l1 = new Layer1(...) _priority_(5, 10);
```

## Listing 4: Simple Program Written for an Augmented Compiler

```
void write_device(...)
{
    _priority_(10);
    ...
}

void read_fifo(Fifo *f, ...)
{
    _priority_(f->priority);
    ...
}

some_function()
{
    Layer2 *l2 = new Layer2(...) _priority_(5);
    _time_(123) write_device(...);
    ...
}
```

manager will call the priority function at the desired time. The scheduler ensures that it's called at the appropriate priority level.

To “manually” call the timer manager to schedule a priority function call at some time, *T*, in the future, you have to handle the packing, unpacking, and saving of the function's arguments. However, the compiler can help considerably by automatically providing this information. From the programmer's point of view, calling a priority function, *F*, at time *T* can look something like:

```
_time_(T) F(...);
```

The compiler executes the following steps to call the timer manager function *F<sub>tm</sub>* to schedule a call for priority function *F* with priority level *P* at time *T*:

1. Stores priority function *F*'s arguments in memory block *M* obtained from the memory manager.
2. Calls the timer manager's function *F<sub>tm</sub>* with a pointer to entry point *F<sub>s</sub>*, a pointer to memory block *M*, value *P*, and time value *T*.

When time *T* is reached, the timer manager calls the scheduler with a pointer to *F<sub>s</sub>*, a pointer to *M*, and the value *P*. The scheduler does the rest. This is exactly what the compiler does when it calls the scheduler for postponed priority functions.

These ideas can be extended to a signal manager. You can schedule priority function, *F*, to be called when signal *S* occurs thus:

```
_signal_(S) F(...);
```

With these changes and with a scheduler, a memory manager, a timer manager, a signal manager, provided by an external library, a program would look something like the code in Listing 4.

## BIBLIOGRAPHY

“Operating System with Priority Functions and Priority Objects,” <http://www.portos.org/doc/whitepaper.pdf>.

*Rabih Chrabieh* (rabih@softwavenet.com) co-founded San Francisco-based Softwave Wireless, LLC. in 2002 and is currently the manager. Previously, he was a manager and technical lead engineer for eight years at ArrayComm in San Jose.