

# Compiler Augmented with Real Time Capabilities

Rabih Chrabieh

9th February 2005

Additional material and software available at  
<http://www.portos.org>

Contact information  
[contact@portos.org](mailto:contact@portos.org)

## **Contents**

<b>1 Introduction</b>	<b>3</b>
<b>2 Declaring Priority Functions</b>	<b>3</b>
<b>3 Calling Priority Functions</b>	<b>4</b>
<b>4 Scheduler</b>	<b>5</b>
<b>5 Memory Manager</b>	<b>6</b>
<b>6 Functions Returning Values and Data Protection</b>	<b>6</b>
<b>7 Dynamic Priority Levels and Priority Objects</b>	<b>7</b>
<b>8 Timer Manager</b>	<b>9</b>
8.1 Case 1: Scheduler Aware Timer Manager . . . . .	10
8.2 Case 2: Scheduler Unaware Timer Manager . . . . .	10
8.3 Calling via a Function Pointer . . . . .	11
<b>9 Generalization</b>	<b>11</b>
<b>10 Conclusion</b>	<b>11</b>

## 1 Introduction

This document describes how to add real time capabilities to a programming language compiler, interpreter or preprocessor. This can be done in a straightforward manner using Priority Functions and Priority Objects. It is harder to achieve with traditional “tasks”.

We describe here how a compiler, interpreter or preprocessor can be augmented to automatically handle priority functions and priority objects. As such, the job of a real time programmer is drastically simplified. Simulation code can be relatively quickly turned into real time embedded software. There is no need to pack function arguments inside message structures. No need to send messages between tasks. The code as is can be quickly converted by adding a few directives to define the priority functions and their priority levels. The programmer only focuses on how to make the design clean and robust by avoiding undesirable interference between different priority levels.

A Real Time Operating System based on Priority Functions and Priority Objects is described in [1]. Briefly, a Priority Function is a function that has been assigned a priority level. A scheduler ensures that such functions are called at the appropriate priority level. A Priority Object is an object that possess a priority level. The methods or functions that handle the object inherit this priority level.

Examples of compilers or interpreters that can be augmented with these concepts are C, C++, Java, Python, Matlab, etc.

## 2 Declaring Priority Functions

A function  $F$  can be declared as 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. The directive can be written, for example, at the start of the function  $F$  or just preceding the declaration of  $F$ . It can be something like

```
_priority_(P)
SET_PRIORITY P
PRIORITY_FUNCTION P
#pragma priority P
#priority P
priority_function(P)
etc.
```

A group of contiguous functions in one file can be declared as priority functions with the same priority level  $P$  by wrapping the group with directives such as

```
SET_PRIORITY P
SET_PRIORITY_END
```

```
#priority P
#priority_end
etc.
```

If a function within the group is not supposed to be a priority function, then this function can have some directive like

```
SET_PRIORITY_NONE
PRIORITY_FUNCTION_NONE
#pragma not_a_priority_function
#priority_none
etc.
```

Note that it would be nice to use directives that do not affect standard compilers. Hence, the same code can be compiled with standard compilers without the real time features. For instance, for the standard C language, a macro or pragma directive is fine. But a `#priority` does not compile properly. The same applies to all directives proposed throughout this document.

### 3 Calling Priority Functions

A programmer calls a priority function the same way he or she would call a normal function. The compiler does the rest of the work. There is no need to save parameters in a message and to pass the message to other functions or tasks. Everything happens seamlessly. From the programmer's point of view, calling  $F$  looks like

```
F(...)
```

where the dots corresponds to the normal arguments of  $F$ .

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

1. Entry point  $F_o$ : this is the old entry point of the function.
2. Entry point  $F_n$ : this is the new entry point of the function.
3. Entry point  $F_s$ : this is the entry point when the function is called from the Scheduler. More on the Scheduler later.

When calling  $F$ , the new entry point  $F_n$  is called and it performs the following steps

1. Compare the priority level  $P$  of  $F$  and the current priority level  $P_{current}$ .
2. If  $P \geq P_{current}$ 
  - (a) Call  $F$  immediately by jumping to old entry point  $F_o$ . When  $F$  is done it returns to the calling code as usual.

3. If  $P < P_{current}$ 
  - (a) Save the arguments of  $F$  in a memory block  $M$  obtained from the Memory Manager. More on the Memory Manager later.
  - (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 does not call the function  $F$ . Rather it stores the call in a database. Thus, the function call is postponed. More on the Scheduler later.
  - (c) Return to calling code without calling  $F$ .

A postponed function call is executed later when its priority level becomes current, i.e., when  $P > P_{current}$ . At this point, the Scheduler calls the function via the entry point  $F_s$ . The Scheduler also supplies a pointer to the memory block  $M$ . The entry point  $F_s$  performs the following steps

1. Restore the function's arguments from the memory block  $M$ .
2. Free the memory block  $M$  by returning it to the Memory Manager.
3. Call the function  $F$  by jumping to entry point  $F_o$  (for faster response, this step can be called before step 2).
4. When  $F$  terminates it returns to the caller, i.e., the Scheduler. The Scheduler can then call other lower priority functions.

Some of the steps mentioned in this section can be inlined and optimized when the compiler knows (at compile time) the priority level  $P$  and the current priority level  $P_{current}$  of the caller function. For example, if  $F$  is being called from a 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 old entry point  $F_o$ . This way, zero overhead is incurred when calling the priority function  $F$ .

## 4 Scheduler

The Scheduler handles the calls to priority functions that have not been called immediately, i.e., postponed priority functions. The call to such priority functions is stored in a database optimized in one way or another. The database is maintained by the Scheduler. The Scheduler is independent of the compiler.

The compiler needs to call the Scheduler in order to store calls to postponed priority functions. Therefore, the compiler needs the symbol name of the Scheduler's function. This can be either a standard symbol name or a specific name supplied via a compiler directive such as

```
SCHEDULER symbol
#pragma scheduler symbol
#scheduler symbol
etc.
```

In the C language the symbol name can be provided in the compiler's command line arguments, e.g., `-D_SCHEDULER_=symbol`.

The compiler also needs to know how to pass arguments to the Scheduler. This can be either standardized or again specified via a directive.

Note that priority functions normally execute in a context that has lower priority than the context of hardware interrupts. Hence, when calling priority functions from hardware interrupts they are always postponed.

## 5 Memory Manager

A Memory Manager is an efficient dynamic memory handler. This is an equivalent to the well known `malloc` and `free` functions of the C language. However, standard `malloc` and `free` may be too inefficient, hence higher performance versions may be used instead. The Memory Manager is independent of the compiler.

For postponed priority functions, the compiler obtains a memory block  $M$  from the Memory Manager. Therefore, the compiler needs to know the symbol names of the Memory Manager functions. These can be either standard symbol names or specific names supplied via compiler directives such as

```
MEMORY_MANAGER_MALLOC symbol
MEMORY_MANAGER_FREE symbol
#pragma memory_manager_malloc symbol
#pragma memory_manager_free symbol
etc.
```

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. Cf. [1].

## 6 Functions Returning Values and Data Protection

A function that returns a value to the caller, or a function that writes to global data that is also accessed by the caller, or more generally a function that changes a state that affects the caller, cannot be simply converted into a priority function.

If its priority level is higher than the caller's priority level, the function can be easily converted into a priority function. However, if its priority level is lower than the caller's priority level, the function call gets postponed. Hence the state change does not occur as the caller expects. To solve this problem, the function can be sub-divided into two or more sub-functions, some running at low priority, some at high priority.

Also, global data may be accessed by low and high priority code. The global data may be trashed if not protected. Traditionally, data protection is achieved via semaphores or priority ceiling (cf. [1]).

The programmer has to pay special attention to all such problems and may have to redesign some functions. To some extent, a compiler augmented with priority functions can help in detecting the problems. The compiler can detect, for example, that a global data is being accessed from two priority functions with different priority levels. The compiler can issue a warning.

## 7 Dynamic Priority Levels and Priority Objects

Some priority functions inherit the priority level of the objects they are working on. In this case the priority level is assigned to the data object instead. Each method running on the data object inherits the priority level. Moreover, the priority level can be decided in a dynamic way at run time. A variety of ways can be imagined to define the priority level of an object, statically or dynamically. Examples in C are given below. The dynamic priority level can be stored in a variable.

```
/* 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_FUNCTION l1->priority
    ...
}
```

Examples in C++ are given below. The priority level is a special class inherited by other classes. The compiler handles this class in a special way.

```
// C++ priority class
class _priority_ {
    priority;
protected:
    _priority_(int priority) : priority(priority) {}
};

// C++ class definition for a Modem's Layer1 object with a
// priority level. It inherits the priority class.
class Layer1 : public _priority_ {
    ...
}
```

```
Layer1(arg1, arg2, int priority) : _priority_(priority) {...}
};
```

```
// C++ object instantiation with priority P
class Layer1 *L1 = new Layer1(arg1, arg2, P);
```

Another simpler example where the compiler does more work than the programmer

```
class Layer1 {
    _priority_object_;
    ...
};
```

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

```
// Or
class Fifo *fifo = new Fifo(...) _priority_(P);
```

In the case of C++, the compiler adds a field in the object structure that contains the priority level. When an object's method is called, the compiler automatically handles it like it handles priority functions except that the priority level is dynamically obtained from the object's structure.

If some methods in the class should not inherit the priority value supplied at instantiation time then something like this can be used

```
class Layer1 {
    _priority_;
    ...

    // This one inherits the priority at instantiation time
    function1() {
        ...
    }

    // This one does not inherit priority (e.g., fixed priority)
    function2() {
        _priority_(10);
        ...
    }
};
```

A class can be given several priority levels. Example

```
class Layer1 {
```

```
_priority_(int P1, int P2); // Unspecified values for now
...

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

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

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

In order to obtain better inlining and optimizations, it is possible to 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 value. This way it can generate more optimized code. Specifying a range can be done with a directive like

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

## 8 Timer Manager

Independent from the compiler, a Timer Manager can handle time events. The Timer Manager works with priority functions. A priority function can be scheduled to run at a specific time in the future. The Timer Manager calls the priority function at the desired time. The Scheduler ensures that it gets called at the appropriate priority level. Cf. [1] for more details.

In order for the Timer Manager to schedule a priority function call at some time  $T$  in the future, it needs to obtain from the programmer the following items

- A pointer to the priority function (some convenient entry point)
- A pointer to a memory block  $M$  where the arguments have been manually stored
- The time value  $T$

However, the compiler can considerably help in this job by automatically providing this information. From the programmer's point of view, calling  $F$  at time  $T$  can look something like

```

_time_(T) F(...)
F(...) _timer_call_(T);
#timer_call T F(...)
etc

```

What the compiler actually does in order to call some Timer Manager function, say  $F_{TM}$ , to schedule a call for priority function  $F$  with priority level  $P$  at time  $T$  is described below.

### 8.1 Case 1: Scheduler Aware Timer Manager

The compiler executes the following steps

- Store the priority function  $F$ 's arguments in a memory block  $M$  obtained from the Memory Manager.
- Call the Timer Manager's function  $F_{TM}$  with a pointer to the entry point  $F_s$ , a pointer to the memory block  $M$ , the value  $P$ , and the time value  $T$ .

When the time  $T$  is reached, the Timer Manager calls the "Scheduler" with a pointer to entry point  $F_s$ , a pointer to  $M$ , and the value  $T$ . The Scheduler does the rest. This is exactly what the compiler would do when it calls the Scheduler for postponed priority functions.

### 8.2 Case 2: Scheduler Unaware Timer Manager

The compiler executes the following steps

- Store the priority function  $F$ 's arguments in a memory block  $M$  obtained from the Memory Manager. Also store at the end of  $M$  the value of the priority level  $P$ . The compiler must have predefined an additional entry point  $F_p$  that can perform the following
  - Extract from  $M$  the value  $P$ .
  - Call the "Scheduler" with a pointer to  $F_s$ , a pointer to  $M$  and the value  $P$ .
  - Return to calling code.
- Call the Timer Manager function  $F_{TM}$  with a pointer to  $F_p$ , a pointer to memory block  $M$  and the value  $T$ .

When the time  $T$  is reached, the Timer Manager calls  $F_p$  with a pointer to  $M$ . The rest is done by  $F_p$ .

For both cases 1 and 2, the compiler needs to know the symbol name of the Timer Manager's function. This can be done via directives. Also, the Timer Manager's functions themselves may need to be declared as priority functions with very high priority levels for proper behavior.

### 8.3 Calling via a Function Pointer

In the instruction `_time_(T) F(...)`; if  $F$  is a symbol that is known at compile time (or at preprocess time), then the compiler can determine the associated symbol  $F_s$  (and other related symbols). However, if  $F$  is a function pointer, the compiler or preprocessor cannot determine  $F_s$  at compile time. Several solutions can be envisaged at run time:

- *Using a function pointer to  $F_s$  instead of  $F$* : The programmer could use directly a function pointer to  $F_s$  in the function call following a `_time_(T)` directive. This is the simplest solution and the most efficient from an implementation point of view. But it does require a little more effort from the programmer to handle the function pointer to  $F_s$ .
- *Passing extra arguments to entry point  $F_n$* : The new entry point  $F_n$  could receive extra arguments that it decodes in order to tell whether the programmer is issuing a direct call to  $F$  or an indirect call via the directive `_time_(T)`. The extra arguments can be passed in registers or on the stack, not a portable solution. The arguments can be passed via global variables, which may require disabling interrupts for a short period of time (or an equivalent solution).
- *Storing the address of  $F_s$  in a location directly preceding  $F_n$* . Hence, the compiler can obtain the value of  $F_s$  simply by knowing the value  $F_n$ .

## 9 Generalization

The ideas presented in the Timer Manager section can be generalized to any application code that needs to handle priority functions in a direct or indirect manner. For example, a Signal Manager that calls priority functions when a signal occurs. The programmer can schedule a priority function  $F$  to be called when signal  $S$  occurs in the following way

```
F(...) _signal_(S);
#signal S F(...)
etc
```

## 10 Conclusion

We have described in this document how to extend a compiler to handle priority functions or priority objects. And with an external library providing a Scheduler, a Memory Manager, a Timer Manager, a Signal Manager, etc, writing real time software, in simulation or embedded environment, becomes a lot easier. A program would look something like this

```
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(...);
    ...
}
```

## References

- [1] Operating System with Priority Functions and Priority Objects. <http://www.portos.org/doc/whitepaper.pdf>.