

# Automating RTOS Inter-Task Function Calls

Rabih Chrabieh

26th December 2004

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

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

## Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Declaring a Task Function</b>	<b>3</b>
<b>3</b>	<b>Calling a Task Function</b>	<b>4</b>
<b>4</b>	<b>Task, Mailbox and Dispatcher</b>	<b>6</b>
<b>5</b>	<b>Memory Manager</b>	<b>6</b>
<b>6</b>	<b>Timer Manager</b>	<b>6</b>
6.1	Calling via a Function Pointer . . . . .	7
<b>7</b>	<b>Future Enhancements</b>	<b>8</b>

## 1 Introduction

This document describes how to automate inter-task function calling and messaging in a real time embedded software. Manually assigning functions to tasks and writing messages to transmit between tasks is a time consuming operation and reduces the modularity of a program. Automation not only saves coding time but also helps in maintaining program modularity by reducing the dependency on the task environment (i.e., RTOS environment).

The automation can be achieved by a programming language compiler, interpreter, or preprocessor. The ideas here apply mainly to the C programming language but can be extended to other programming languages.

A real time embedded program is usually subdivided into several tasks, each running at some priority level. When a task T0 needs to call a function F running inside a task T, the function cannot be directly called. Rather the following steps are needed:

- Task T0 constructs a message M in which it packs the arguments for function F
- Task T0 sends the message M to task T
- Task T receives the message M
- Task T unpacks the arguments from message M
- A dispatcher in task T calls function F with the received arguments. This step often requires maintaining a dispatcher's table of functions, an unfriendly operation.

The above steps can be automated. In order to do so, we need to make a clear distinction between a task and the functions that can run in this task. A programmer calls a function from anywhere without necessarily knowing in which task it will run. The function runs in its default task. Furthermore, a programmer can choose to override the default task of the function and force it to run in a different task. Essentially a task is a virtual layer or group of functions that execute at some priority level. Functions are temporarily or permanently assigned to the virtual layer.

In the following, we will denote by "Task Function", a function F that can run in the current task or in a different task. Such a function cannot be directly called unless we augment the compiler, interpreter or preprocessor with the necessary tools.

## 2 Declaring a Task Function

A function F can be declared as a Task Function by using a directive preceding the function, such as

```
_task_(T)
void F(...)
{
    ...
}
```

The arguments and body of `F` above have been replaced by three dots. `F` normally cannot return a value and hence the “void” return type. This declaration means that `F` will usually run in the default task `T`.

The mailbox `MB` of `T` could also be specified as a second parameter in the directive `_time_(T, MB)`. Or the mailbox can have a name that is derived from `T` by adding a known suffix, for example, `T_mailbox`.

### 3 Calling a Task Function

A programmer calls a Task Function the same way he or she would call a normal function. The function can be called from anywhere in the code, from any task. 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(...);
```

A compiler converts a function `F` into a Task Function that runs in task `T` by creating some wrapping code or new entry points. The entry points can be defined by

1. Entry point `Fo`: this is the old entry point of the function.
2. Entry point `Fn`: this is the new entry point of the function.
3. Entry point `Fd`: this is the entry point when the function is called from the Dispatcher. More on the Dispatcher later.

When calling `F`, the new entry point `Fn` is called and it performs the following steps

1. If task `T`, i.e., `F`’s task, is currently active
  - (a) Call `F` immediately by jumping to old entry point `Fo`. When `F` is done it returns to the calling code as usual.
2. If task `T` is not currently active
  - (a) Save the arguments of `F` and the entry point `Fd` in a memory block `M` (message) obtained from the Memory Manager. More on the Memory Manager later.
  - (b) Send the message `M` to mailbox `MB` of task `T`.

(c) Return to calling code without calling F.

Inside task T, a Dispatcher receives the message M and it executes the following steps

1. Obtain the function's entry point Fd from the memory block M.
2. Call entry point Fd.
3. Free the memory block M by returning it to the Memory Manager (alternatively, this step can be performed by the entry point Fd or elsewhere).

The entry point Fd executes the following steps

1. Unpack the arguments of F.
2. Call the function F by jumping to old entry point Fo.
3. When F terminates it returns to the caller, i.e., the Dispatcher, which can dispatch other messages.

The function F can be executed in a different task than the default task T. This can be done by calling F in the following way

```
_task_(T1, MB1) F(...);
```

In this case, F will be executed in task T1 whose mailbox is MB1. Again, if the mailbox's name can be derived from the task's name by adding a known suffix, it is not necessary to supply the mailbox name.

The compiler or preprocessor handles this call by creating another entry point Fm that executes the same steps as Fn except that the task and mailbox are not the default ones but they are supplied in the call as extra arguments. When calling F, usually only entry point Fn is known. Therefore, it may be necessary to locate Fm at a constant distance from Fn so that it can always be derived from Fn. Alternatively, the entry points Fn and Fm are merged and a flag is used to tell if the call is for the default task or not.

Passing extra arguments to Fm can only be done if the processor architecture is known, i.e., which registers can be tampered with and passed to Fm (assuming the prototype of F cannot be altered). Another solution that is independent of the processor architecture is to pass the extra arguments via a global variable. But a global variable can be trashed in case of preemption by a higher priority level. The solution then is to have one global variable per priority level.

Some of the steps mentioned in this section can be inlined and optimized if the compiler knows at compile time from which task F is being called. For example, if F is being called from the same task T, the compiler can generate a direct call to entry point Fo. Zero overhead is incurred. It may help sometimes to tell the compiler or preprocessor that the current call to F is occurring in the same task. This can be done, for example, by having the old entry point Fo accessible to the caller.

## 4 Task, Mailbox and Dispatcher

The programmer needs to create the task T and to start it. When it starts it calls the task's entry point, for example, `T_entry_point`. The programmer must fill out the content of `T_entry_point` as follows (or some variation of that)

```
void T_entry_point(void)
{
    mailbox_create(&T_mailbox);

    while ( 1 ) {
        void *message = mailbox_receive(&T_mailbox);
        stuff_before();
        dispatcher(message);
        stuff_after();
    }
}
```

The task suspends while waiting for messages. Most of the functions running in this task are called via the dispatcher. `stuff_before()` and `stuff_after()` are provided for all sorts of functions that need to execute every time a message is received.

All of the steps above can be automated but there is little gain in doing so.

## 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.

## 6 Timer Manager

A Timer Manager handles time events. Calling a function at a given time in the future via the Timer Manager also involves tedious work of packing and unpacking function arguments in a message. This work can again be automated.

Say F is a Task Function that runs in task T, and that needs to be called at time t. The programmer declares F as a Task Function. Then he calls F as follows

```
_time_(t) F(...);
```

This tells the Timer Manager that F should be called at time t. For this to work, F must be a Task Function, and the task T must be calling the Dispatcher on every received message.

The compiler or preprocessor can automate this work by creating two new entry points for F, say Ft1 and Ft2.

Entry point Ft1 is called when the instruction `_time_(t) F(...)`; is issued. This entry point executes the following steps

1. Store the function F's arguments and the entry point Fd in a memory block M obtained from the Memory Manager (similar to what entry point Fn does).
2. Call the Timer Manager with a pointer to the entry point Ft2, a pointer to the memory block M, and the time value t.

When time t is reached, the Timer Manager calls the entry point Ft2 with a pointer to memory block M. This entry point executes the following steps

1. Send the memory block M (message) to mailbox MB of T

The task T receives the message M and the Dispatcher does the rest of the job.

The same method can be generalized to any application code that needs to call the Task Function F when an event occurs. For example, a Signal Manager can be called in the following way

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

This tells the Signal Manager to attach F to signal S, and hence to call F when S is posted.

## 6.1 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 or preprocessor can determine the associated symbol Ft1. However, if F is a function pointer, the compiler or preprocessor cannot determine Ft1 at compile time. Several solutions can be envisaged:

- *Using a function pointer to Ft1 instead of F:* The programmer could use a function pointer to Ft1 instead of F 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 Ft1.
- *Passing arguments to entry point Fn:* The new entry point Fn 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 entry point Ft1. 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).

## 7 Future Enhancements

The end goal behind this design is to fully automate the tasking environment in an RTOS. Possible enhancements:

- Automatic generation of the task's entry point function.
- Automatic creation and handling of the mailbox.
- Automatic estimation of a task's stack space. Measure the Hardware and Software Interrupts stack space independently. Estimation is done during the debugging phase. The programmer can help by increasing the stack space if needed.
- Automatic task creation. The programmer only supplies the task names and/or priority levels.