Operating System with Supertasks, Priority Functions and Priority Objects

Rabih Chrabieh

12th February 2005

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

Contact information
contact@portos.org

Contents

1 Introduction **3**

2 The Limitations of a Traditional Task **3**

3 Supertask **4**

4 Priority Functions Outside Any Supertask **6**

5 Compiler Aided RTOS **6**
5.1 Declaring Supertasks 6
5.2 Declaring Priority Functions (Subtasks) 7
5.3 Timers and Signals 7

6 Supertask Scheduler **8**
6.1 Context Switch 9
6.1.1 State of Registers Expected by F 10
6.2 Calling a Priority Function 11
6.3 Argument Passing Across Different Supertasks 13

7 Priority Space **13**
7.1 Changing Priority Level 14

8 Memory Access Protection **14**

1 Introduction

An RTOS based on priority functions and priority objects is presented in [1] and [2]. Briefly, a priority function is any function that has been assigned a priority level. It can preempt lower priority levels and it gets preempted by higher priority levels. This RTOS does not rely on tasks for managing priority levels, and provides several advantages over traditional RTOSes. Essentially a simpler and cleaner design, reduced memory usage and higher performance. It does have some disadvantages but that can be simply circumvented.

Nevertheless, tasks are sometimes useful. For example, tasks are needed in order to achieve some form of time slicing, or in order to use the MMU features of some DSPs for memory protection. Separate stacks and heaps can be defined for separate tasks, and can be protected from accidental access.

This document introduces the new concept of "supertask": a collection of priority functions with a dedicated stack. A supertask is at the same time lighter, more powerful and more efficient than a traditional task.

2 The Limitations of a Traditional Task

A traditional task is a collection of subtasks that share a common stack and perhaps other resources such as a common heap. The subtasks run in a sequential fashion at the same priority level. Although the priority level can be momentarily changed, this change affects all subtasks. In other words, subtask 1 of task A cannot preempt subtask 2 of the same task A. The priority change affects only other tasks, such as task B.

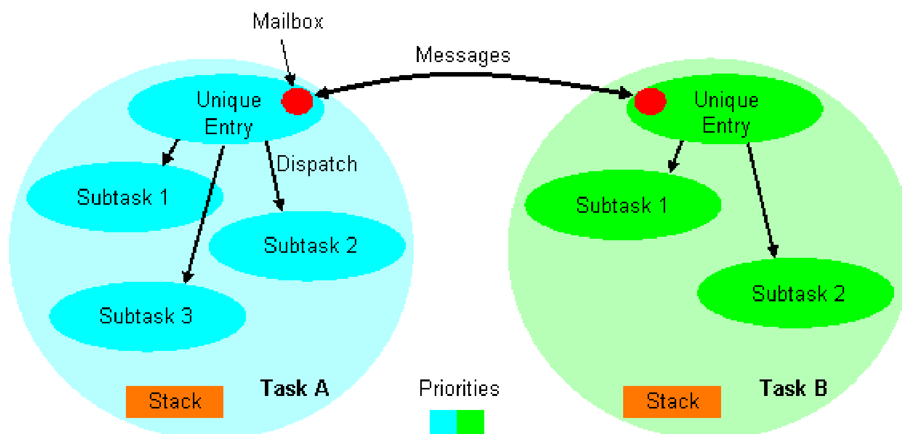

Figure 1: A traditional task consists of several subtasks. It has a unique entry point via a mailbox. Priority is the same for all subtasks (priorities are represented by colors)

The task possesses a unique entry point. This is the point where it suspends when it has nothing left to do and it waits for new messages. Defining a second

entry point does not really work because a task cannot wait for messages at two different points. A mailbox or message queue handles received messages. A dispatcher dispatches each message to the appropriate subtask within the task. Once a subtask is started it cannot be preempted by another subtask belonging to the same task.

Calling subtasks from another task can only be done via the unique entry point, i.e., via the mailbox. Such a call involves handling the mailbox and dispatching to the subtask. All this takes up software programming as well as CPU cycles and memory resources.

Switching between tasks entails a context switch that is relatively heavy. All registers have to be saved for one task and they have to be restored for the other task.

3 Supertask

A supertask is also a collection of subtasks. However, each subtask in this case is a priority function. This means the subtasks are essentially independent except that they share a common stack, and perhaps other resources such as a heap.

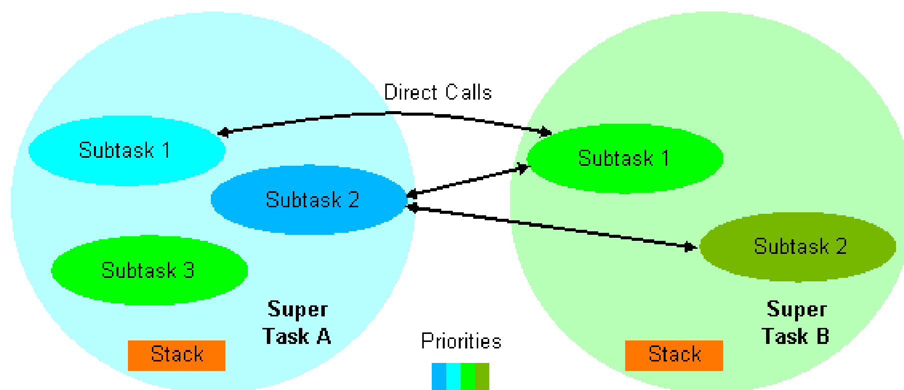

Figure 2: A supertask consists of several subtasks sharing a common stack. “Each subtask is a priority function.” Each subtask has its own entry point. There is no mailbox. Subtasks can have different priority levels (priorities are represented by colors)

The advantages of a supertask over a traditional task are:

- Since a subtask is a priority function it can be called from anywhere, i.e., it has its own entry point into the supertask. There is no notion of mailbox and unique entry point. The supertask can easily wait for two or more different events without the need for special event handling.

- Inter-task messaging is much more efficient and simple: there is no mailbox.

- Subtasks belonging to one supertask can have different priority levels. Higher priority subtasks preempt lower priority subtasks. Within one supertask all the concepts of priority functions and priority objects apply. This results in cleaner and more efficient software. Cf. [1].
- A context switch between supertasks is "much more efficient" than a context switch between traditional tasks. In most cases the context switch is either half or nearly non-existent. This is due to the fact that subtasks are independent and, like regular functions, they return the registers to their original state upon termination. More details later.
- Interrupt response jitter may be higher with traditional tasks because context switches are less efficient and may need to lock interrupts for longer periods of time.
- Since subtasks can have different priority levels, priority inversion issues can be avoided. When some subtask is too urgent, it is given a higher priority level and it is forbidden access to critical data. Or the critical data is managed by special high priority functions. Cf. [1].
- Creating a real time product from non-real-time software or simulations is remarkably faster with supertasks and priority functions than it is with tasks. Most of the real time burden such as creating entry point functions, handling mailboxes, creating messages, dispatching messages, rearranging the source code to suit the mailbox and entry point, etc, is eliminated. The original non-real-time software remains nearly intact and hence more portable.
- Since little extras are required to manage supertasks (no mailbox, no entry point function, etc), it is very simple to turn off the real time aspect of the operating system and to test the code in non-real-time. This allows debugging non-real-time and real-time problems separately.
- Temporary supertasks can be defined during debugging time for memory protection between various pieces of software. Once the code has been debugged, the supertasks can be easily turned off to improve performance and decrease stack usage.

Since it owns a dedicated stack, a supertask can be suspended (e.g., via a semaphore) just like a traditional task. When a supertask is suspended, all its subtasks are automatically suspended. As far as the scheduler is concerned, what happens is that the subtasks (priority functions) are postponed until the supertask resumes. Even if a subtask has a high priority level, it gets postponed if it belongs to a suspended task.

In theory, a traditional RTOS can achieve the behavior of supertasks by assigning a dedicated task and stack to every subtask. However, this is infeasible because stack space dramatically increases. With supertasks such subtasks share stack space.

Two software applications running on one processor can be defined as two distinct supertasks. Each software application possesses dedicated stack and heap. Multi-processor architectures can run supertasks on two or more processors. Supertasks can be time-sliced. Their memory can be protected against accidental access.

4 Priority Functions Outside Any Supertask

Some priority functions can run outside any supertask. They use the currently active stack. More efficient code can be written this way. However, such priority functions cannot suspend execution. They act like Hardware or Software Interrupts.

The priority level of such priority functions should be above the priority level of priority functions belonging to tasks. Otherwise the scheduler is infeasible or inefficient. The priority functions running outside any supertask have a dedicated scheduler and dedicated priority space (a segment above the supertask priority space).

5 Compiler Aided RTOS

As described in documents [2] and [3], it is possible to augment a compiler, interpreter or preprocessor to automatically handle priority function calls and inter-task function calls.

In the following we will reuse some of the notation used in the aforementioned documents. The terminology "compiler" will be used to denote either of a compiler, an interpreter or a preprocessor.

Example directives are given for the C programming language. They can be easily extended to any programming language such as C++, Java, Python, Matlab, etc.

5.1 Declaring Supertasks

A supertask is declared like a traditional task via an API. Stack space is reserved, as well as other resources such as a heap. A supertask can be assigned a default priority level that subtasks can inherit if they do not define their own priority level. For example,

```
API_taskCreate(supertask, stack, heap, ...);
```

A handle to the supertask is created. It allows to check or modify the status of the supertask, to check or modify the current priority level, to check all the currently active or pending priority functions within the supertask, and to delete the supertask if needed.

5.2 Declaring Priority Functions (Subtasks)

In order to declare a function F as a priority function (i.e., subtask) belonging to supertask T, the function declaration (or prototype) of F can be preceded by the following directives:

```
_task_(T) _priority_(P)
void F(int a, double *b);
```

The two directives could be combined into one directive `_priority_(T, P)`. This tells the compiler that, by default, function F has priority P and belongs to supertask T. When a programmer calls `F(x, y)` the compiler automatically generates a call for F inside supertask T, at priority level P. T and P can be dynamic variables or expressions evaluated at run-time, e.g., an expression containing arguments of F.

If the parameter P is omitted, the subtask inherits the default priority level of the supertask. If the supertask T parameter is omitted, the subtask is a priority function that runs outside any supertask.

The default settings P, T can be overridden by calling F preceded by the same directive `_priority_` with the desired parameters P1, T1:

```
_task_(T1) _priority_(P1) void F(x, y);
```

Hence F can be easily called inside any supertask and at any priority level. If parameter P1 or T1 is omitted, default values are used.

Note that in order to avoid modifying existing source code it is possible to place the directives and priority function prototypes in separate files that are made available to the compiler as included headers or in some other method. This keeps the source code highly portable.

5.3 Timers and Signals

Similarly to what is mentioned in [2] and [3], a subtask F can be called at specific time t in the future using the directive:

```
_time_(t) void F(x, y);
```

F will be called at time t, inside its default supertask T, at its default priority level P.

In order to change the default priority level of F, the following can be used:

```
_time_(t) _priority_(P1) void F(x, y);
```

Likewise for attaching subtask F to signal S:

```
_signal_(S) void F(x, y);
```

6 Supertask Scheduler

A supertask scheduler is a mixed breed between a traditional task scheduler and a priority function scheduler.

A simple implementation separates the task scheduler from the priority function scheduler. So switching between supertasks involves the usual context switch where all the registers of one supertask are saved and all the registers of the other tasks are restored. Such a scheduler does not take advantage of the property of priority functions that, after execution, return all registers to their original state. It does not either take advantage of the fact that when starting a new priority function in a different supertask there is no state to restore. Therefore, in most situations there is no need to save and/or restore registers.

We will describe here a more complex scheduler that takes full advantage of the properties of priority functions for best performance. However, some of the optimizations mentioned thereafter do not apply if different functions in different supertasks were compiled using different compilers that do not respect the same conventions for passing arguments and preserving registers.

For each supertask, the scheduler maintains a list of pending priority functions. The list is sorted in the priority level order. The scheduler also maintains a list of supertasks. Each supertask is marked with its current status: running, pending (i.e., ready and waiting), suspended (e.g., on semaphore). The list is also sorted in the current priority level order, where the current priority of a supertask is set to the currently highest priority level in that supertask.

When a priority function starts executing, it is removed from the list of pending priority functions. Hence, it is not necessary to store in this list priority functions that will be immediately executed, i.e., without postponing. This improves performance.

On the other hand, if a priority function is temporarily stopped in order to switch to another supertask, a descriptor can be created for this priority function and stored in the list of pending priority functions. The descriptor specifies how to resume the priority function. Alternatively and more efficiently, the stack is used to store the same information.

The priority lists are optimized for priority level sorting and access by using bitmap hash tables.

The lists should preferably preserve calling order of priority functions across supertasks. One solution is to use a common bitmap to all supertasks. But when supertasks are suspended, this solution is slowed down since finding the highest priority function requires a search over non-suspended supertasks. Another solution is to keep one bitmap per supertask and to mark a priority function by its calling order. This is a 32 or 64 bit number, for example, which is incremented for each new call.

The scheduler should provide at least the following services:

1. `API_call`: For calling priority functions. As explained in [1], a priority function F cannot be directly called. It is called via `API_call` that handles priority levels. `API_call` may be invisible to the programmer and auto-

generated by the directives `_priority_`, `_task_` (cf. section 5.2). Also `API_call` may sometimes be inlined by the compiler.

2. `API_suspend`: For suspending a supertask.

3. `API_resume`: For resuming a supertask that was previously suspended.

4. `API_hwi`: If a hardware interrupt calls priority functions, this API follows the hardware interrupt to restore supertask and priority functions context.

For intra-task calls there is no context switch.

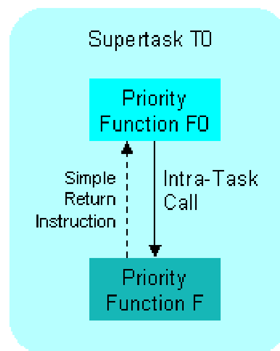

Figure 3: Intra-task Call: F0 calls F. Both priority functions are in supertask T0. No context switching occurs. F is called like a regular function. When F terminates, it returns to F0 via a simple return instruction. Unlike Software Interrupts in traditional RTOSes, priority functions do not need to save all registers but only those few that are in use and are not preserved by function call F.

6.1 Context Switch

When switching from supertask T0 to supertask T, a context switch occurs. In traditional RTOSes such a context switch always involves saving the registers of T0 and restoring the registers of T. With supertasks, some of these operations are often unnecessary.

Denote by F0 the currently running priority function in supertask T0, and by F the priority function in supertask T.

1. **"Full Context Switch": *If F0 has not yet terminated and F has been previously started***, then a full context switch occurs. The registers of T0 are saved and the registers of T are restored. A jump instruction resumes function F. This is what happens in traditional RTOSes. Here it typically happens when supertask T0 is suspended or when its priority level is decreased below that of T.

2. **"Half Context Switch - Type Save": *If F0 has not yet terminated and F has not been previously started, i.e., F is a new function call***, then a half context switch occurs. The registers of T0 are saved but there are no registers to restore for a fresh start of F. Note that other lower priority functions in supertask T could have been started and are waiting to proceed. Hence the stack of T may not be empty. The function F will use the stack starting from its current state. Upon termination it will return the stack to that state. Also note that if F is being directly called from F0, then it is only necessary to save part of the registers of F0, those that should be preserved by F. If F is being called after hardware interrupt preemption then fewer registers need to be saved, those that the hardware interrupt did not already save.

3. **"Half Context Switch - Type Restore": *If F0 has terminated and F has been previously started, and if the state of the registers is not as expected by F***, then a half context switch occurs. The registers of T are restored. There is no need to save the registers of T0 since F0 has terminated. The state of the registers is not as expected by F means that the function F0 did not return the registers to what F is expecting. Cf. section 6.1.1.

4. **"Tiny Context Switch - Type Resume": *If F0 has terminated and F has been previously started, and if the state of the registers is as expected by F***, then a tiny context switch occurs. No registers are saved or restored. This is the case when F0 returns the state of the registers to what F is expecting. Cf. section 6.1.1.

5. **"Tiny Context Switch - Type New": *If F0 has terminated and F has not been previously started, i.e., F is a new function call***, then a tiny context switch occurs. No registers are saved.

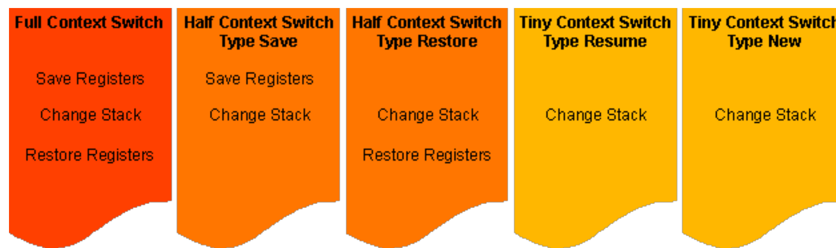

Figure 4: Five types of context switches. Context switching is more efficient for supertasks than it is for traditional tasks.

6.1.1 State of Registers Expected by F

The priority function F in supertask T was previously started. It called a priority function F1 in supertask T1 (directly or after preemption by hardware

interrupt). T1 is different from T. Immediately before calling F1 the state of the registers is denoted by R. When F1 and subsequent priority functions execute, the state of the registers is momentarily modified. When F1 and subsequent priority functions terminate, the state of the registers is returned to R. Hence, there is no need to restore registers for function F even if it lies in a different supertask. If the registers of F were saved via a half context switch before calling F1, then those registers are ignored and deleted from stack.

However, the state of the registers is not always returned to R as expected by F. This occurs if priority function F1 or some subsequent priority function was suspended (or blocked by lowering its priority level) and did not return to F. In that case, it is necessary to restore the registers of F (that were saved during the half context switch).

If a subsequent priority function is suspended, it does not automatically mean that the state of the registers will not be returned to R. It depends on which state the subsequent priority function started from. If it started from state R like F1 did, then suspending this priority function means it will not return to state R.

In order to keep track of the state of the registers R, before calling F1 the function F requests a cookie C that acts like a "time stamp" or "register state stamp" (C is a number that increments for each subsequent request). C is stored on the stack of T and on the stack of T1 before calling F1. When F1 terminates, the state of the registers is back to R. If at this point a subsequent priority function F2 is to be started in supertask T2, before calling F2 the cookie C is removed from the stack of T1 and copied to the stack of T2 (i.e., the cookie is handed over to T2). When all subsequent priority functions (such as F2) terminate or are suspended and it is time to resume function F, the cookie from the terminated or suspended function is cross-checked with the cookie C. If the cookies match, it means that the state of the registers is R as expected by F.

6.2 Calling a Priority Function

Suppose the currently running supertask is T0, the currently running priority function is F0 and the current priority level is P0. A priority function F, with priority level P, that should run in supertask T is called via `API_call`. The scheduler checks first if the priority function F should be called right away or if it should be postponed and stored in the list of pending priority functions.

The priority function is postponed if one of the following is true:

1. Current level is hardware interrupt level. This means supertask T0 was preempted by a hardware interrupt and the hardware interrupt is calling the priority function. Priority functions are always postponed when called from hardware interrupt level.

2. Priority level P is strictly less than current priority level P0.

3. Priority level P is equal to current priority level P0 and supertask T is different from supertask T0. Note that if T and T0 are identical, P and

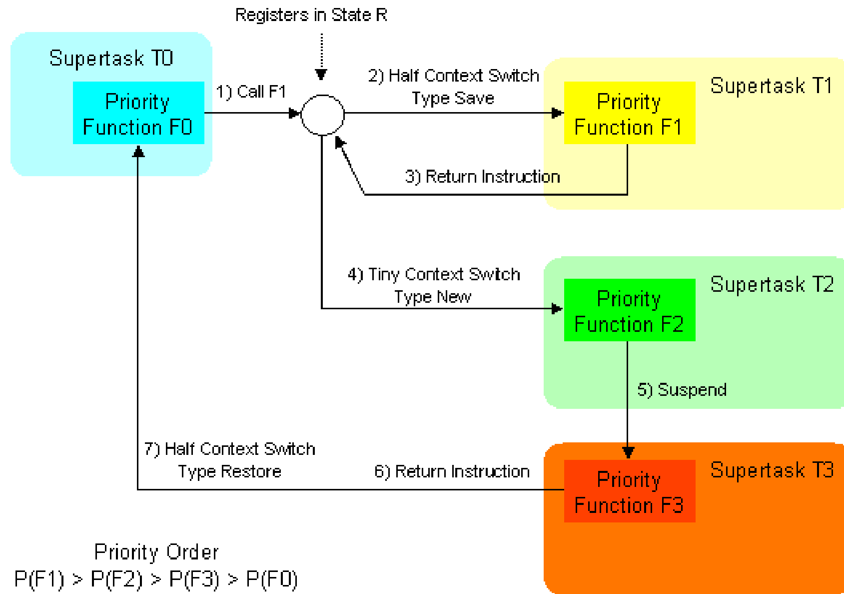

Figure 5: Example of Inter-task calls: F0 calls F1. F1 returns but in the meantime F2 and F3 were scheduled. First F2 is called. It suspends without returning. F3 takes over. When F3 returns, F0 proceeds.

P0 are equal, then the priority function F is called right away. Refer to document [1] for more details on priority functions scheduling.

4. Supertask T is in suspended state.

In all other cases, a priority function is immediately called. This is done as follows:

1. If supertasks T and T0 are identical then a normal function call takes place as explained in document [1]. There is nothing to be done regarding the stack.
2. If supertasks T and T0 are different then a “Half Context Switch - Type Save” occurs. Essentially, only the registers of the running supertask are saved before switching to the new supertask. There are no registers to be restored for the new supertask because the funcion call is starting from zero.

Before calling a new priority function F in supertask T, the scheduler needs to keep track of the currently executing priority function F0 in supertask T0 (T0 different from T) so that F0 can be resumed later. This can be done by leaving some information on the stack such as the instruction pointer address,

the priority level P0 of F0, and the cookie describing the state of the registers R0 of F0. The same cookie C0 is copied onto the stack of T before starting F.

Before F returns to priority function F0, another higher priority function F00 may start in supertask T0. F0 cannot resume before F00 terminates (since they belong to the same supertask). When F00 terminates, the scheduler checks the information stored on the stack, namely priority level P0. This is how the scheduler can keep track of priority functions that are no longer stored in the bitmap hash tables (because they have already started). Alternatively, priority functions can remain stored in the bitmap hash tables after they have started (marked with a flag that they have started).

In the case of an intra-task call, the scheduler still generates the cookie, and stores on stack the cookie as well as the current priority level. Then before calling the new priority function, the scheduler stores once again the cookie on stack. When the new priority function terminates, it checks if the priority level of the previous priority function is highest. If so, it checks if the cookies match. If so it means there should be no saved registers and it issues a simple return instruction (or a jump to the saved instruction pointer). Otherwise, the scheduler restores the registers before issuing the jump instruction).

6.3 Argument Passing Across Different Supertasks

A priority function F0 running in supertask T0 can call a priority function F running in supertask T (T different from T0) and pass it arguments in the following way:

1. The arguments of F that are stored in registers are automatically passed. No additional action is required by the scheduler.

2. The arguments of F that are stored on stack are copied from stack T0 to stack T (or directly written to stack T if the compiler is able to perform the job).

This is more efficient than creating messages to carry the arguments across supertasks. However, it only works if the function F is the currently highest priority function in supertask T. Otherwise, other more urgent functions need to execute and the stack is busy.

7 Priority Space

A unique space is defined for all priority levels and all supertasks. Two different supertasks can possess subtasks that span the full space of priority levels. Although distinct (orthogonal) priority spaces could be defined for different supertasks, there is no real interest in doing so.

On the other hand, two different software applications could have distinct (orthogonal) priority spaces. Time-slicing with some round robin or other priority scheme lets the two applications share the same processor.

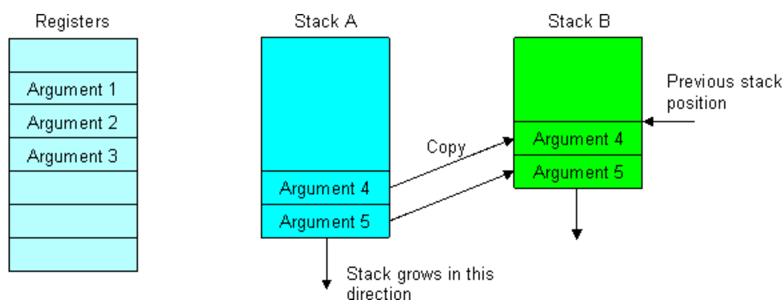

Figure 6: Inter-task call: passing arguments. Arguments in registers are left as is. Arguments in stack are copied to new stack.

7.1 Changing Priority Level

Changing the priority level of a supertask can be confusing since each priority function can have a different priority level. One solution is the following:

1. Raising the priority level of a priority function is simple (cf. [1]). Hence, increasing the priority level of the supertask can mean to increase the priority level of the currently active priority function. Or it can mean to add a positive offset to all priority functions within the supertask.

2. Lowering the priority level of only one priority function within a supertask is impossible (cf. [1]). Hence, the remaining option is to add a negative offset to ALL priority functions within the supertask. Such behavior is compatible with traditional RTOSes when all priority functions within one supertask have equal priority levels.

8 Memory Access Protection

It is sometimes necessary to prevent one supertask from accidentally calling the wrong function address in another supertask (which results in a system crash).

In traditional RTOSes the mailbox being the only entry point, the task cannot directly call a function address in another task. But with supertasks, any function address can be called. A mechanism is needed to prevent bad calls.

One solution is for the compiler to generate a vector table for each supertask. The vector table contains the addresses of priority functions in the supertask that are accessible from other supertasks. The vector table is published and accessible by all supertasks. On the other hand, the code space of each supertask is protected and inaccessible via direct calls.

When calling a priority function in a different supertask, an indirect call is generated via the vector table. The scheduler reroutes the call to the supertask after changing the memory access settings.

The vector table option could be turned on during debug time and turned off in the final product.

References

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

[2] Compiler Augmented with Real Time Capabilities.
http://www.portos.org/doc/compiler.pdf.

[3] Automating RTOS Inter-Task Function Calls.
http://www.portos.org/doc/tasks.pdf.