
The Trampoline Handbook

release 2.0

Jean-Luc Béchenec
Sébastien Faucou
Pierre Molinaro
Florent Pavin

CONTENTS

I	The Real-Time Operating System	7
1	Tasks	9
1.1	States a task	9
1.2	The scheduling	10
1.3	Writing the code of a task	12
1.3.1	Task identifier	12
1.4	Tasks services	12
1.4.1	ActivateTask	12
1.4.2	ChainTask	14
1.4.3	TerminateTask	14
1.4.4	Schedule	15
1.4.5	GetTaskID	15
1.4.6	GetTaskState	16
1.5	Inside Task management	16
1.5.1	Static attributes	16
1.5.2	Dynamic attributes	17
1.5.3	Additional task states	17
1.6	The <i>idle</i> task	18
2	Resources	19
2.1	OSEK Priority Ceiling Protocol	19
2.2	The RES_SCHEDULER resource	19
2.3	Standard and Internal Resources	22
2.4	OIL description	22
3	OS Applications	23
3.1	Execution of the OS Applications startup and shutdown hooks	23

4	Timing Protection Implementation	25
4.1	Low Level Functions	25
4.1.1	FRT related functions	25
4.1.2	TPT related functions	26
5	Schedule Table Implementation	27
5.1	The States of a Schedule Table	28
5.2	Processing a Schedule Table	30
6	The communication library	33
6.1	Internals	33
7	System generation and compilation	35
7.1	The generated files	35
7.2	The Configuration Macros	36
7.2.1	Number of objects macros	38
7.2.2	Error Handling Macros	38
7.2.3	Protection Macros	39
7.2.4	Hook call macros	40
7.2.5	Miscellaneous macros	40
7.3	Application configuration	42
7.3.1	Counter related constants declaration	42
7.3.2	Events definition	42
7.3.3	Standard resources definition	42
7.3.4	Tasks definition	43
8	Ports details	47
8.1	PowerPC	47
8.1.1	System services	47
8.1.2	Dispatching the service call	47
8.1.3	Interrupt handler	53
8.1.4	The CallTrustedFunction service	53
8.1.5	The ExitTrustedFunction service	55
8.1.6	Execution of the OS Applications startup and shutdown hooks	57
8.1.7	The MPC5510 Memory Protection Unit	57
II	The Goil system generator	61
9	The Goil templates	63
9.1	The configuration data	64
9.1.1	The <i>PROCESSES</i> , <i>TASKS</i> , <i>BASICTASKS</i> , <i>EXTENDEDTASKS</i> , <i>ISRS1</i> and <i>ISRS2</i> lists	64

9.1.2	The <i>COUNTERS</i> , <i>HARDWARECOUNTERS</i> and <i>SOFTWARECOUNTERS</i> lists	65
9.1.3	The <i>EVENTS</i> list	66
9.1.4	The <i>ALARMS</i> list	66
9.1.5	The <i>REGULARRESOURCES</i> and <i>INTERNALRESOURCES</i> lists	66
9.1.6	The <i>MESSAGES</i> , <i>SENDMESSAGES</i> and <i>RECEIVEMESSAGES</i> lists	67
9.1.7	The <i>SCHEDULETABLES</i> list	68
9.1.8	The <i>OSAPPLICATIONS</i> list	69
9.1.9	The <i>TRUSTEDFUNCTIONS</i> list	70
9.1.10	The <i>READYLIST</i> list	70
9.1.11	The <i>SOURCEFILES</i> , <i>CFLAGS</i> , <i>CPPFLAGS</i> , <i>ASFLAGS</i> , <i>LDFLAGS</i> and <i>TRAMPOLINESOURCEFILES</i> lists	71
9.1.12	The <i>INTERRUPTSOURCES</i> list	71
9.1.13	Scalar data	72
9.2	The Goil template language (or GTL)	74
9.3	GTL types	74
9.3.1	string readers	74
9.3.2	boolean readers	75
9.3.3	integer readers	75
9.3.4	list readers	75
9.4	GTL operators	75
9.4.1	Unary operators	75
9.4.2	Binary operators	76
9.4.3	Constants	76
9.5	GTL instructions	77
9.5.1	The <i>let</i> instruction	77
9.5.2	The <i>if</i> instruction	77
9.5.3	The <i>foreach</i> instruction	77
9.5.4	The <i>for</i> instruction	78
9.5.5	The <i>loop</i> instruction	78
9.5.6	The <i>!</i> instruction	79
9.5.7	The <i>?</i> instruction	79
9.5.8	The <i>template</i> instruction	79
9.5.9	The <i>write</i> instruction	80
9.5.10	The <i>error</i> and <i>warning</i> instructions	80
9.6	Examples	80
9.6.1	Computing the list of process ids	81
9.6.2	Computing an interrupt table	81
9.6.3	Generation of all the files	82

Part I

The Real-Time Operating System

Tasks

A Task is an execution framework for the functions of the application¹. A task is a kind of process. Tasks are executed concurrently and asynchronously, see 1.2. 2 kinds of task exist: basic tasks and extended tasks. A basic task cannot block (i.e. it cannot use a service that may block) while an extended task can. The tasks and their properties are declared in the OIL file, see ???. Their functions are defined in a C file.

1.1 States a task

A task may be in different states. A basic task may be currently executing (in the **RUNNING** state), ready to execute (in the **READY** state) or not active at all (in the **SUSPENDED** state). Figure 1.1 shows the states of a basic task. An extended task has an additional **WAITING** state. Figure 1.2 shows the states of an extended task. See section 1.5.3 for additional informations about the states of a task.

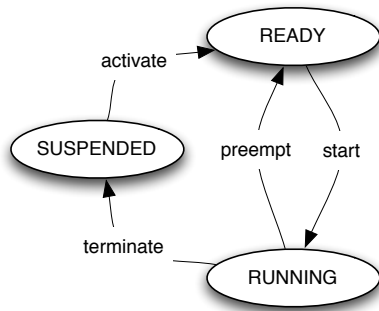


Figure 1.1: *States of a BASIC task.*

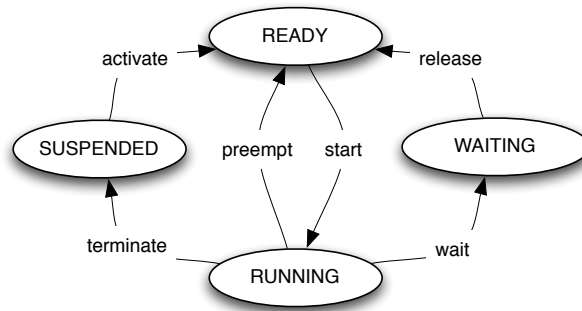


Figure 1.2: *States of an EXTENDED task.*

A task goes from one state to the other according to various conditions as shown in table 1.1.

¹The term *Application* is also used in AUTOSAR to designate a set of object, this manual uses OS Application to name the AUTOSAR applications and Application to name the user level software.

Table 1.1: Transition from state to state of a task.

transition	former state	new state	description
activate	SUSPENDED	READY	the task is set in the READY state on one of the following occurrences: services <code>ActivateTask</code> or <code>ChainTask</code> , activation notification coming from an alarm, a schedule table or a message.
start	READY	RUNNING	the task is set to the running state and begin to execute because it has the highest priority in the system and has been elected by the scheduler.
terminate	RUNNING	SUSPENDED	the task is set to the SUSPENDED state when it calls the <code>TerminateTask</code> service.
preempt	RUNNING	READY	the task is set to the READY state when the scheduler starts a higher priority task.
wait	RUNNING	WAITING	the task may be set to the WAITING state when it calls the service <code>WaitEvent</code> .
release	WAITING	READY	the task is set to the READY state when it gets one of the events it is waiting for.



A system service may do more than one transition at a time. For instance, if a task is activated by calling `ActivateTask` and its priority is higher than the priority of the current running task, the new task will go from **SUSPENDED** to **RUNNING** and the intermediate state **READY** will not be observable.

1.2 The scheduling

Trampoline schedules the tasks dynamically during the execution of the application. A task is scheduled according to its priority and whether it is preemptable or not. The priority of a task is given at design stage, and indicated in the OIL file using the `PRIORITY` attribute, see ??, and may change during execution when the task gets or release a resource. The preemptability of a task may be set too. It is also indicated in the OIL file using the `SCHEDULE` attribute, see ??.

A tasks continues to run until it is preempted because a task having a higher priority is put in the **READY** state, or it blocks because it is waiting for an event. Only extended tasks may block. If more than one task have the same priority, tasks are run one after the other because a task may not preempt an other task having the same priority. So there is no round robin among tasks of the same priority level.

A non-preemptable task runs until it calls `Schedule` and a higher priority task is in the **READY** state or until it blocks. More informations about priority and preemptability may be found in chapter 2.

In the following examples, the horizontal axis is the time. The state of the task is indicated in a rectangle that spans a period of time. When the task is running the rectangle is grayed. An up arrow \uparrow indicates a task activation and a down arrow \downarrow a task termination.

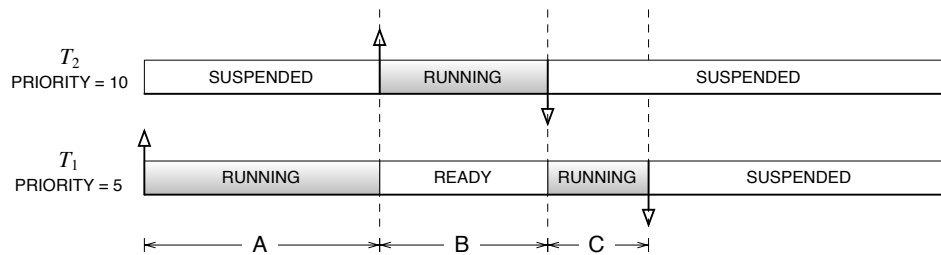


Figure 1.3: Scheduling of preemptable tasks. During A period, T_1 is RUNNING and T_2 is SUSPENDED. Then T_2 is activated. Since $\text{Prio}(T_2) > \text{Prio}(T_1)$, T_1 is preempted and T_2 runs (B period). T_2 terminates and T_1 becomes RUNNING again (C period) until it terminates.

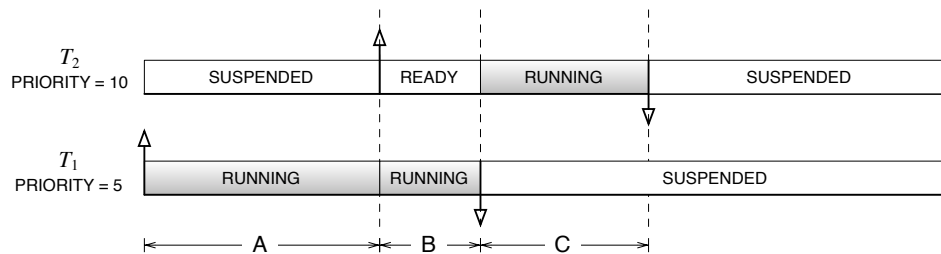


Figure 1.4: Scheduling of non-preemptable tasks. During A period, T_1 is RUNNING and T_2 is SUSPENDED. Then T_2 is activated. Even if $\text{Prio}(T_2) > \text{Prio}(T_1)$, T_1 is non-preemptable and continues to run until it terminates (B period). In the meantime, T_2 is READY. T_1 terminates and T_2 runs (C period) until it terminates.

1.3 Writing the code of a task

Trampoline provides a `TASK` macro to define a task in a C source file. The macro takes one argument which is the identifier of the task:

```
TASK(MyTask)
{
    /* code of the task */
}
```

The code of the task is plain C.

1.3.1 Task identifier

Each task has an identifier of type `TaskType`. This identifier is declared in the OIL file and is used in system calls to refer to a particular task. Before using such an identifier in your program, you have to declare it:

```
DeclareTask(MyTask);
```

This makes the `MyTask` identifier available in the current scope.



When the task has been define above using the macro `TASK`, the identifier of the task is already in the scope and `DeclareTask` is not needed.

1.4 Tasks services

1.4.1 ActivateTask



This service does a rescheduling

Activates a new instance of a task. If activation counter has reached the maximum activation count or the task cannot be activated for timing protection purpose, the service fails. Otherwise if an instance is already active (`RUNNING` or `READY`), the state does not change and the activation is recorded to be done later. If no instance is active, the state of the task is changed to `READY`.

Figures 1.5, 1.6 and 1.8 show 2 examples of task activation.

Prototype of ActivateTask:

```
StatusType ActivateTask(TaskType TaskID);
```

Arguments of ActivateTask:

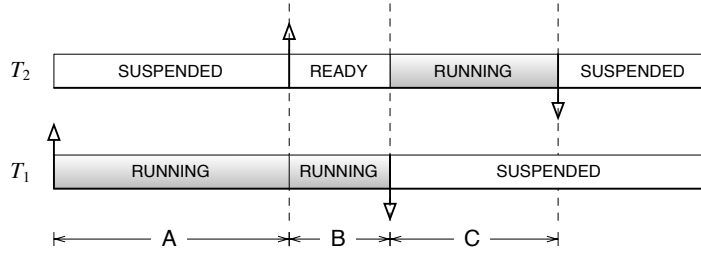
`TaskID` The id of the task to activate.

Status codes returned by ActivateTask:

`E_OK` No error, the task has been successfully activated (extended and standard).

`E_OS_ID` Invalid `TaskID`. No task with such an id exists (extended only).

`E_OS_LIMIT` Too many activations of the task (extended and standard).



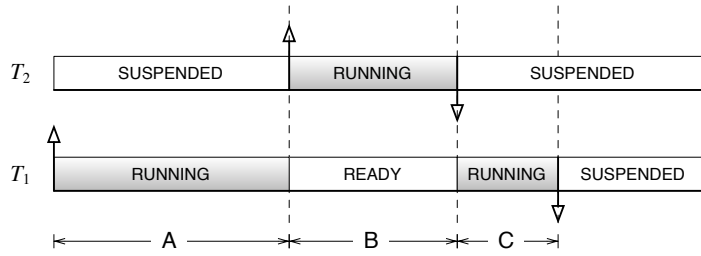
```

TASK(T2) {
    ... /* C period */
    TerminateTask();
}

TASK(T1) {
    ... /* A period */
    ActivateTask(T2);
    ... /* B period */
    TerminateTask();
}

```

Figure 1.5: Activation of a lower priority task. $Prio(T_1) \geq Prio(T_2)$. During A period, T_1 is RUNNING and T_2 is SUSPENDED. Then T_1 calls `ActivateTask(T2)`; . Since T_2 does not have a higher priority, it becomes READY (B period). T_1 terminates and T_2 runs (C period) until it terminates.



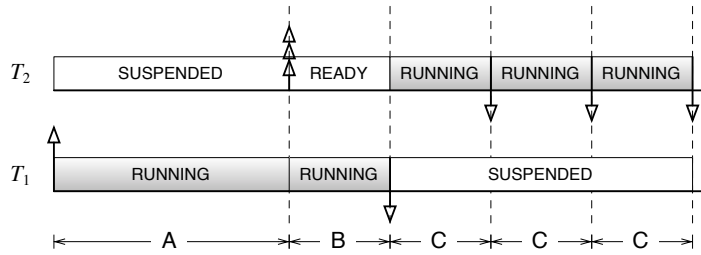
```

TASK(T2) {
    ... /* B period */
    TerminateTask();
}

TASK(T1) {
    ... /* A period */
    ActivateTask(T2);
    ... /* C period */
    TerminateTask();
}

```

Figure 1.6: Activation of a higher priority task. $Prio(T_1) < Prio(T_2)$. During A period, T_1 is RUNNING and T_2 is SUSPENDED. Then T_1 calls `ActivateTask(T2)`; . Since T_2 has a higher priority, it becomes RUNNING (B period). T_2 terminates and T_1 resumes (C period) until it terminates.



```

TASK(T2) {
    ... /* C period */
    TerminateTask();
}

TASK(T1) {
    ... /* A period */
    ActivateTask(T2);
    ActivateTask(T2);
    ActivateTask(T2);
    ... /* B period */
    TerminateTask();
}

```

Figure 1.7: Multiple activations of a lower priority task. $Prio(T_1) \geq Prio(T_2)$. During A period, T_1 is RUNNING and T_2 is SUSPENDED. Then T_1 calls `ActivateTask(T2)`; 3 times. Since T_1 has a higher priority, T_2 does not run immediately and the 3 activations are recorded provided the `ACTIVATION` attribute in the OIL description of the task is a least 3 (B period). When T_1 terminates, the scheduler executes T_2 3 times (C periods).

1.4.2 ChainTask



This service does a rescheduling

This service puts task TaskID in **READY** state, and the calling task in the **SUSPENDED** state. It acts as the **TerminateTask** service for the calling task.

Prototype of ChainTask:

```
StatusType ChainTask(TaskType TaskID);
```

Arguments of ChainTask:

TaskID The id of the task to activate.

Status codes returned by ChainTask:

E_OK No error, the task TaskID has been successfully activated and the calling task has been successfully terminated. Note in this case **ChainTask** does not return so actually E_OK is never returned (extended and standard).

E_OS_ID Invalid TaskID. No task with such an id exists (extended only).

E_OS_LIMIT Too many activations of the task (extended and standard).

E_OS_RESOURCE The calling task still held a resource (extended only).

E_OS_CALLEVEL Called outside of a task (extended only).

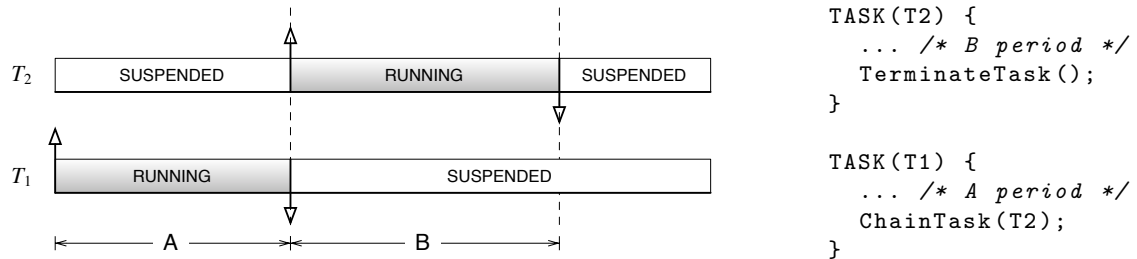


Figure 1.8: Chaining of tasks. During A period, T_1 is **RUNNING** and T_2 is **SUSPENDED**. Then T_1 calls **ChainTask(T2)**; T_1 terminates and T_2 is activated. Then T_2 runs (B periods).

1.4.3 TerminateTask



This service does a rescheduling

This service stops the calling task and puts it in **SUSPENDED** state.

Prototype of TerminateTask:

```
StatusType TerminateTask();
```

Status codes returned by `TerminateTask`:

`E_OK` No error, the calling task has been successfully terminated. Note in this case `TerminateTask` does not return so actually `E_OK` is never returned (extended and standard).

`E_OS_RESOURCE` The calling task still held a resource (extended only).

`E_OS_CALLEVEL` Called outside of a task (extended only).

1.4.4 Schedule

This service does a rescheduling. `Schedule` does not deal directly with tasks but since it is a call to the scheduler, it is presented here.

If called from a preemptable task that does not use an internal resource, `Schedule` has no effect. If called from a preemptable or a task that uses an internal resource, the priority of the task reverts to its base priority and a rescheduling occurs.

`Schedule` allows to implement cooperative multitasking to insure synchronous rescheduling.

Prototype of `Schedule`:

```
StatusType Schedule();
```

Status codes returned by `Schedule`:

`E_OK` No error. (extended and standard).

`E_OS_RESOURCE` The calling task still held a resource (extended only).

`E_OS_CALLEVEL` Called outside of a task (extended only).

1.4.5 GetTaskID

`GetTaskID` writes in the *TaskID* variable passed as reference the identifier of the task currently `RUNNING`. If no task is currently `RUNNING` because `GetTaskID` was called from an ISR or before Trampoline is started, `INVALID_TASK` is got.



The argument is a pointer. Do not pass an uninitialized pointer. Proper use of this service supposes a `TaskType` variable is instantiated, then its address is passed to `GetTaskID` as shown in the example below:

```
TaskType runningTaskID;
GetTaskID(&runningTaskID);
```

Prototype of `GetTaskID`:

```
StatusType GetTaskID(TaskRefType TaskID);
```

Arguments of `GetTaskID`:

`TaskID` Reference to the task..

Status codes returned by `GetTaskID`:

`E_OK` No error. (extended and standard).

`E_OS_PROTECTION_MEMORY` The caller does not have access to the addresses of *TaskID* reference (extended + AUTOSAR scalability class 3 and scalability class 4 only).

1.4.6 GetTaskState

`GetTaskState` writes in the variable passed as reference in *State* the state of the task given in *TaskID*.



The *State* argument is a pointer. Do not pass an uninitialized pointer. Proper use of this service supposes a `TaskState` variable is instantiated, then its address is passed to `GetTaskState` as shown in the example below:

```
TaskStateType T1State;
GetTaskState(T1, &T1State);
```

Prototype of `GetTaskState`:

```
StatusType GetTaskState(TaskType TaskID, TaskStateRefType State);
```

Arguments of `GetTaskState`:

TaskID The id of the task..

State Reference to the state..

Status codes returned by `GetTaskState`:

E_OK No error. (extended and standard).

E_OS_ID Invalid TaskID. No task with such an id exists (extended only).

E_OS_PROTECTION_MEMORY The caller does not have access to the addresses of *State* reference (extended + AUTOSAR scalability class 3 and scalability class 4 only).

1.5 Inside Task management

1.5.1 Static attributes

A task has the following static attributes:

The entry point of the task. A pointer to the code of the task. When the scheduler start a task instance the first time, it uses this pointer to begin the execution.

The internal resource the task uses if any. An internal resource is automatically taken when a task enters the `RUNNING` state and automatically released when the task leaves the `RUNNING` state. See ?? for more informations.

The base priority of the task as specified in the OIL file. This priority is used to reset the current priority when the task is activated.

The maximum activation count of the task as specified in the OIL file.

The kind of task, `BASIC` or `EXTENDED`.

The task id. Used for internal checking.

The id of the OS Application the tasks belong to (only available in AUTOSAR scalability class 3 and scalability class 4).

The timing protection configuration if any (only available in AUTOSAR scalability class 2 and scalability class 4).

1.5.2 Dynamic attributes

A task has also the following dynamic attributes:

The context. This is the chunk of RAM where the current execution context of a task is stored when the task is in the **READY** or **WAITING** state. The execution context is the value of the microprocessor's registers (program counter, stack pointer, other working registers). So the context depends on the target on which Trampoline runs.

The stack(s). This is the chunk of RAM where registers are pushed for function call. This attributes depends on the target architecture. For instance, the C166 micro-controller uses 2 stacks.

The current activation count. When a task is activated while not in **SUSPENDED** state, the activation is recorded and is actually done when the task returns to the **SUSPENDED** state. Many activation may be recorded according to the value given to the **ACTIVATION** task OIL attribute. When a task is activated, the current activation count is compared to the maximum activation count and if \geq , the activation fails.

The list of resources the task currently owns.

The current priority of the task. This priority starts equal to the basic priority and may increase when the task get a resource.

The state of the task as defined in sections 1.1 and 1.5.3.

The trusted counter. If = 0, the task is non-trusted. If > 0 the task is trusted. See chapter ?? for more informations. This counter is available if Trampoline is compiled with memory protection support.

The activation allowed flag. If true, the task may be activated. If false, it cannot be activated. This flag is set by the timing protection facility. It is available if Trampoline is compiled with timing protection support. See chapter ??.

1.5.3 Additional task states

In addition to states presented in section 1.1, 2 extra states are used for internal management:

AUTOSTART This state is used to indicate what task should be started automatically when **StartOS** is called. An **AUTOSTART** task is in this initial state but no task is in this state once the application code is running. **StartOS** iterates through the tasks and activates those that are in the **AUTOSTART** state.

READY_AND_NEW This state is used to flag a task that is ready but has its context uninitialized. This happens when the task has just been activated. The kernel initializes the context of the task the first time it goes to the **RUNNING** state.

Figure 1.9 show a complete task state automaton for both basic and extended tasks with these states added.

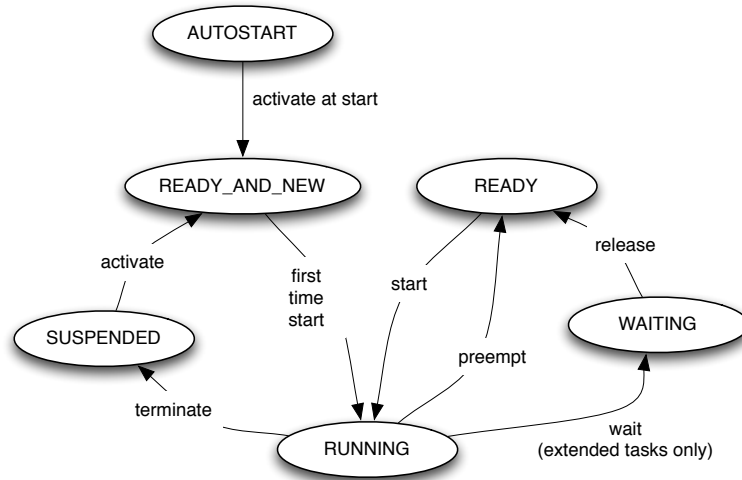


Figure 1.9: States of a task in Trampoline. *AUTOSTART* is the initial state of autostart tasks. *SUSPENDED* is the initial state of both non autostart tasks.

1.6 The *idle* task

The *idle* task is activated by `StartOS`. It is a `BASIC` task with a priority of 0 (i.e. the lowest priority in the system, the lowest priority of tasks defined in the application is 1). So when no other task is currently running, the *idle* task run.

To be able to use specific platform capabilities (to put the micro-controller in stand by mode for example), this task calls repetitively a hardware specific function called `tpl_sleep` (defined in *machines/*). The task is then able to quantify the microprocessor occupation.

GOIL doesn't produce anything about this idle task (unlike application(s) task(s)). The idle task descriptor is defined in '`tpl_os_kernel.c`'.

Resources

A resource is an object used to protect a critical section in a task or in an ISR and to insure mutual exclusion. By using a resource to protect the use of a shared piece of data or a shared hardware device, the programmer avoids race conditions. Figure 2.1 shows an example of race condition.

2.1 OSEK Priority Ceiling Protocol

OSEK uses a modified version of the Priority Ceiling Protocol [5]. A priority is assigned to each resource. This priority is computed to be at least equal to the highest priority of the tasks and ISRs that use the resource. So let T_1, T_2, \dots, T_n a set of tasks sharing the same resource R and P_1, P_2, \dots, P_n their priorities so that $P_i = P(T_i)$. We have $P(R) = \max_{i=1,n}(P_i)$.

When a task gets a resource, its priority is raised to the priority of the resource. That way, the task will run with the priority of the highest priority task and will insure the release of the resource is not delayed by a lower priority task. In addition, since every other tasks that use the same resource have now a priority \leq , they cannot preempt the running task and mutual exclusion is insured. Figure 2.2 show an example of resource use.

The priority of a resource is computed by goil according to the priorities of the tasks and ISRs that use the resource.

2.2 The RES_SCHEDULER resource

Trampoline provides a predefined standard resource called `RES_SCHEDULER`. This resource has a priority \geq to the maximum priority of the tasks but $<$ to the minimum priority of the ISR. When a task gets `RES_SCHEDULER`, it becomes non preemptable. To make `RES_SCHEDULER` available to the application, the `USERESCHEDULER` attribute must be set to `TRUE` within the `OS` object in the OIL file. Unlike resources defined by the application, there is no need to declare `RES_SCHEDULER` is used by a task in the OIL file.

```

int val = 0;
int actCount = 0;

TASK(bgTask)
{
    while (1) {
        val++;
        val--;
    }
}

TASK(periodicTask)
{
    activationCount++;
    if ((actCount % 2) == 1) {
        val++;
    }
    else {
        val--;
    }

    TerminateTask();
}

TASK(displayTask)
{
    printf("val=%d count=%d\n",
           val,
           activationCount);

    TerminateTask();
}

```

```

val=2 count=10
val=3 count=20
val=4 count=30
val=5 count=40
val=2 count=50
val=2 count=60
val=0 count=70
val=-2 count=80
val=-1 count=90
val=-1 count=100
val=-2 count=110
val=0 count=120
val=0 count=130
val=0 count=140
val=0 count=150
val=-2 count=160
val=-1 count=170
val=-2 count=180
val=-4 count=190
val=-4 count=200
val=-6 count=210
val=-4 count=220
val=-5 count=230
val=-6 count=240
val=-7 count=250
val=-6 count=260
val=-3 count=270
val=-3 count=280
val=-5 count=290
val=-5 count=300

```

Figure 2.1: Shared data access. In this example 3 preemptable tasks are used. *bgTask* increments and decrements the global integer variable *shared* in an infinite loop. *periodicTask* runs every 100ms and increments the global integer variable *activateCount*. If *activateCount* is odd, *periodicTask* increments *shared* otherwise it is decremented. A third task, *displayTask* runs every second and displays both variables. On the left, the corresponding program, on the right one of the possible outputs

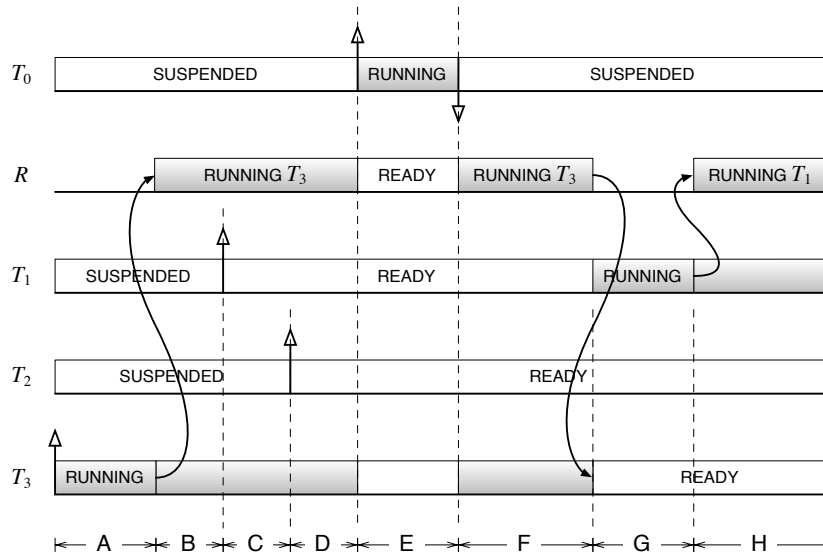


Figure 2.2: Scheduling with a resource used by 3 tasks and a fourth task having a higher priority. $P(T_0) > P(T_1) > P(T_2) > P(T_3)$. R is used by T_1, T_2 and T_3 so $P(T_0) > P(R) \geq P(T_1)$. During A period, T_3 is RUNNING and other tasks are SUSPENDED. Then T_3 gets R and $P(T_3) \leftarrow P(R)$ (B to F periods). T_1 is activated and becomes READY; since $P(T_3) \geq P(T_1)$, T_1 does not run (C to F periods). T_2 is activated and becomes READY; for the same reason it does not run (D to H periods). T_0 is activated and because $P(T_0) > P(R)$ it runs (E period). T_0 terminates and T_3 continues its execution (F period). Then T_3 releases R and $P(T_3)$ reverts to its base priority; so since $P(T_1) > P(T_2) > P(T_3)$, T_1 runs (G period). T_1 gets R and $P(T_1) \leftarrow P(R)$ (H period).

2.3 Standard and Internal Resources

Standard resources are got and released explicitly by tasks and ISRs using the ad-hoc services. Internal resources are got implicitly when the task enters the **RUNNING** state and released implicitly when the task calls **Schedule** or blocks when using **WaitEvent**.

Standard resources are dedicated to the protection of critical sections around the access to a shared data or to a device. Internal resources are used to implement non preemptable tasks within a task group. A task group is a set of task that are non preemptable by each other but remain preemptable by higher priority tasks in the application. A task group priority is the priority of its internal resource.

Trampoline provides a predefined internal **RES_SCHEDULER** resource with the same priority. This internal resource is used to implement non preemptable tasks in the whole application as if all the non preemptable tasks belong to an implicit task group. When a task is non preemptable by setting the **SCHEDULE** attribute to **NON** in its OIL description, the task is assigned the internal **RES_SCHEDULER** resource.

2.4 OIL description

A resource is described using a **RESOURCE** object. **RESOURCEPROPERTY** is the single attribute of this object. A standard resource is defined with the following code:

```
RESOURCE res {
    RESOURCEPROPERTY = STANDARD;
};
```

And an internal resource is defined with the following code:

```
RESOURCE other_res {
    RESOURCEPROPERTY = INTERNAL;
};
```

A third kind of declaration exists for **LINKED** resources. A linked resource may be linked to a linked resource or a standard resource but a link tree of resources must have a standard resource at the root. A linked resource has the same priority as the standard resource it is linked to and is a kind of reference. Linked resources are provided to replace nested access to the same resource (which is prohibited) and are rarely used.

```
RESOURCE l_res {
    RESOURCEPROPERTY = LINKED { LINKEDRESOURCE = res };
};
```

OS Applications

OS Applications are a set of objects managed by Trampoline and sharing common data and access rights.

3.1 Execution of the OS Applications startup and shutdown hooks

These hooks are executed from the kernel but with the access right of a task belonging to the OS Application. The system generation tool should choose one of the tasks of the OS Application to be used as context to execute the OS Application startup and shutdown hooks. Execution of an OS Application startup hook is done by the `tpl_call_startup_hook_and_resume` function. The argument of this function is a function pointer to the hook. Similarly execution of an OS Application shutdown hook is done by the `tpl_call_shutdown_hook_and_resume` function. These functions end by a call to `NextStartupHook` and `NextShutdownHook` services respectively to cycle through the hooks.

Timing Protection Implementation

The Timing Protection Implementation uses 2 timers. The first one is a *Free Running Timer* (FRT) which is used for *Time Frame*. The second one is a classical timer called *Timing Protection Timer* (TPT) which is used for *Execution Time Budget*, *Resource Locking Budget* and *Interrupt Disabling Budget*.

4.1 Low Level Functions

These functions are provided by the *Board Support Package* and are used to manage the timers needed by the Timing Protection.

4.1.1 FRT related functions

`tpl_status tpl_start_frt(void)` starts the FRT. On a microcontroller having a FRT that starts automatically when the system is powered on, this function does nothing but must be present since it is called by Trampoline in initialization stage. An error code is returned: *E_OK* means no error, *E_OS_NOFUNC* means the FRT could not be started.

`tpl_status tpl_read_frt(tpl_tp_tick *out_value)` write the current value of the FRT in *out_value*. An error code is returned: *E_OK* means no error, *E_OS_NOFUNC* means the FRT could not be read.

`tpl_status tpl_elapsed_frt(tpl_tp_tick last_tick, tpl_tp_tick *out_value)` write the number of ticks elapsed since *last_tick* in *out_value*. If the FRT has overflown/underflown between the time *last_tick* was get and the time `tpl_elapsed_frt` is called, `tpl_elapsed_frt` gives a correct value. An error code is returned: *E_OK* means no error, *E_OS_NOFUNC* means the FRT could not be read.

4.1.2 TPT related functions

`tpl_status tpl_init_tpt(???)` initializes the TPT. An error code is returned: *E_OK* means no error, *E_OS_NOFUNC* means the TPT could not be initialized.

`tpl_status tpl_deinit_tpt(void)` deinitializes the TPT. An error code is returned: *E_OK* means no error, *E_OS_NOFUNC* means the TPT could not be deinitialized.

`tpl_status tpl_start_tpt(tpl_tp_tick delay)` starts the TPT with an expiration delay equal to *delay* ticks. At that time, the `tpl_tpt_handler` function is called. An error code is returned: *E_OK* means no error, *E_OS_NOFUNC* means the TPT could not be started because it is not initialized.

`tpl_status tpl_read_tpt(tpl_tp_tick *out_value)` write the current value of the TPT in *out_value*. An error code is returned: *E_OK* means no error, *E_OS_NOFUNC* means the TPT could not be read.

`tpl_status tpl_elapsed_tpt(tpl_tp_tick last_tick, tpl_tp_tick *out_value)` write the number of ticks elapsed since *last_tick* in *out_value*. An error code is returned: *E_OK* means no error, *E_OS_NOFUNC* means the TPT could not be read.

Schedule Table Implementation

Here is the files list :

- ‘`tpl_as_schedtable.c`’ contains the API services.
- ‘`tpl_as_st_kernel.c`’ contains the kernel API services, `tpl_process_schedtable()` and `tpl_adjust_next_expiry_point()`
- ‘`tpl_as_action.c`’ contains `tpl_action_finalize_schedule_table()`
- ‘`tpl_as_definitions.h`’ contains the schedule table’s states (`SCHEDULETABLE_STOPPED`, `SCHEDULETABLE_BOOTSTRAP`, `SCHEDULETABLE_AUTOSTART_ABSOLUTE...`)
- ‘`tpl_os_timeobj_kernel.c`’ contains `tpl_remove_time_obj()` which has been modified for the schedule table object.

The schedule table class diagram is shown in Figure 5.1 below.

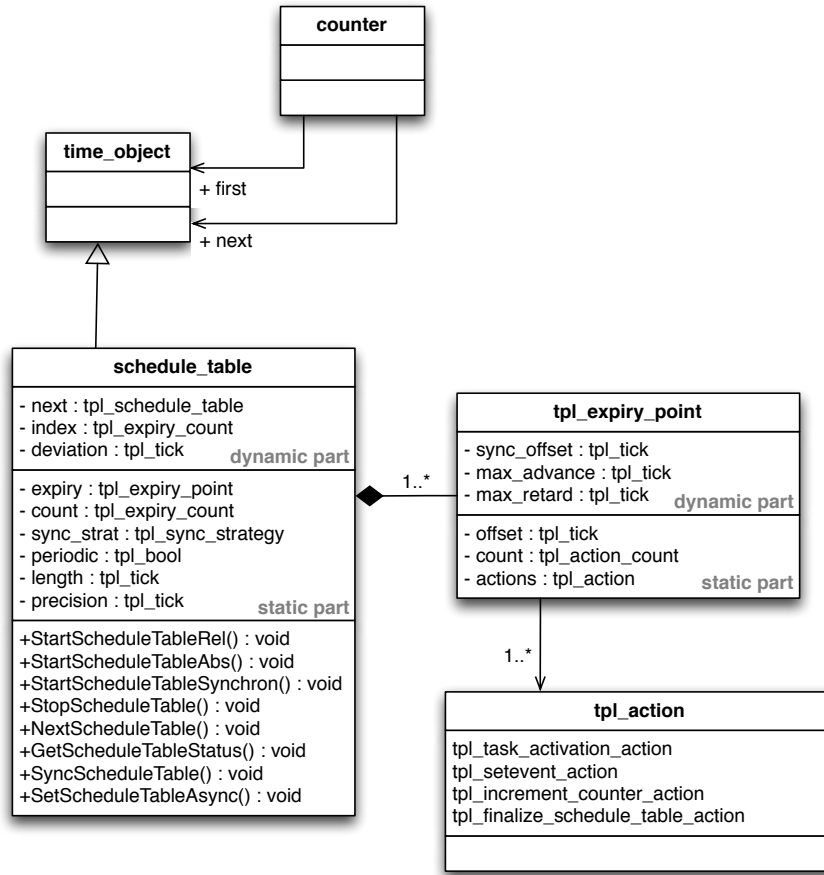


Figure 5.1: Schedule table class diagram

5.1 The States of a Schedule Table

A schedule table always has a defined state. States include those found at page 42 of the AUTOSAR specifications 3.1 and others states used for internal management.

Indeed, **bit 1** is the "autostart" bit. It's used when autostarted schedule tables have been declared in the OIL file. Goil generates schedule tables with SCHEDULETABLE_AUTOSTART_X (X can be RELATIVE, ABSOLUTE or SYNCHRON) state. At startup (in `tpl_init_os()`), the system starts autostarted schedule tables and resets the **bit 1**.

bit 4 is the "bootstrap" bit. It's used when the first expiry point of a schedule table is dated in more than **OsCounterMaxAllowedValue** ticks from the current date¹. It can happen when :

- the schedule table start (<tick_val>) is after the current date and the first expiry point comes between the current date and <tick_val>

¹As the <offset> parameter of `StartScheduleTableRel()` cannot be greater than **OsCounterMaxAllowedValue** minus the **InitialOffset** of the schedule table (OS276), the first expiry point cannot be in more than **OsCounterMaxAllowedValue** ticks from the current date. Thus the "bootstrap" bit can set by `StartScheduleTableAbs()` only.

- `<tick_val>` is before the current date and the first expiry point comes after the current date

Figure 5.2 below shows a bootstrap example for the first item.

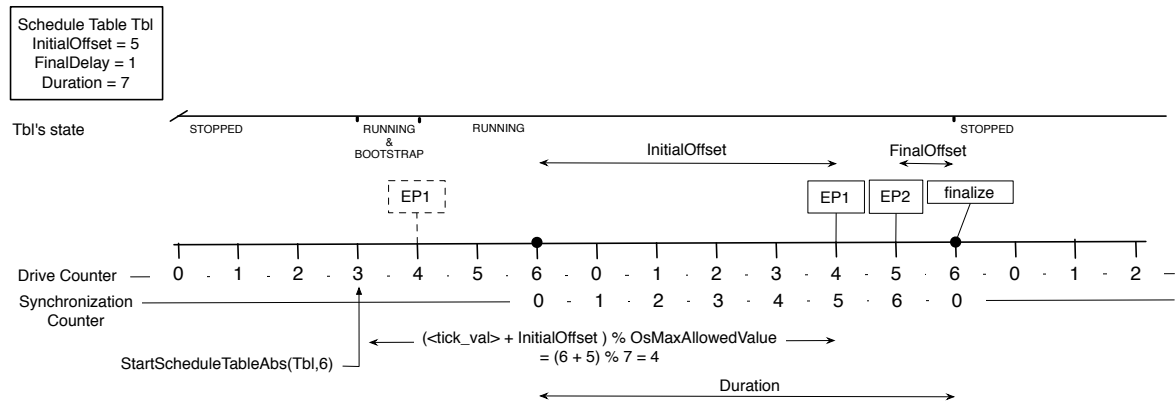


Figure 5.2: Bootstrap example

bit 5 is the "asynchronous" bit. It tells the system that the schedule table is in asynchronous mode.

Thus, the different states of a schedule table are described in Table ?? below.

Table 5.1: States of a schedule table

State code	Binary code	Associated constant
0	000000	SCHEDULETABLE.STOPPED
1	000001	SCHEDULETABLE.RUNNING
5	000101	SCHEDULETABLE.NEXT
9	001001	SCHEDULETABLE.WAITING
13	001101	SCHEDULETABLE.RUNNING_AND_SYNCHRONOUS
6	000110	SCHEDULETABLE.AUTOSTART_ABSOLUTE
10	001010	SCHEDULETABLE.AUTOSTART_RELATIVE
14	001110	SCHEDULETABLE.AUTOSTART_SYNCHRON
16	010000	SCHEDULETABLE.BOOTSTRAP
32	100000	SCHEDULETABLE.ASYNC

Figure 5.3 shows how a schedule table goes from state to state.

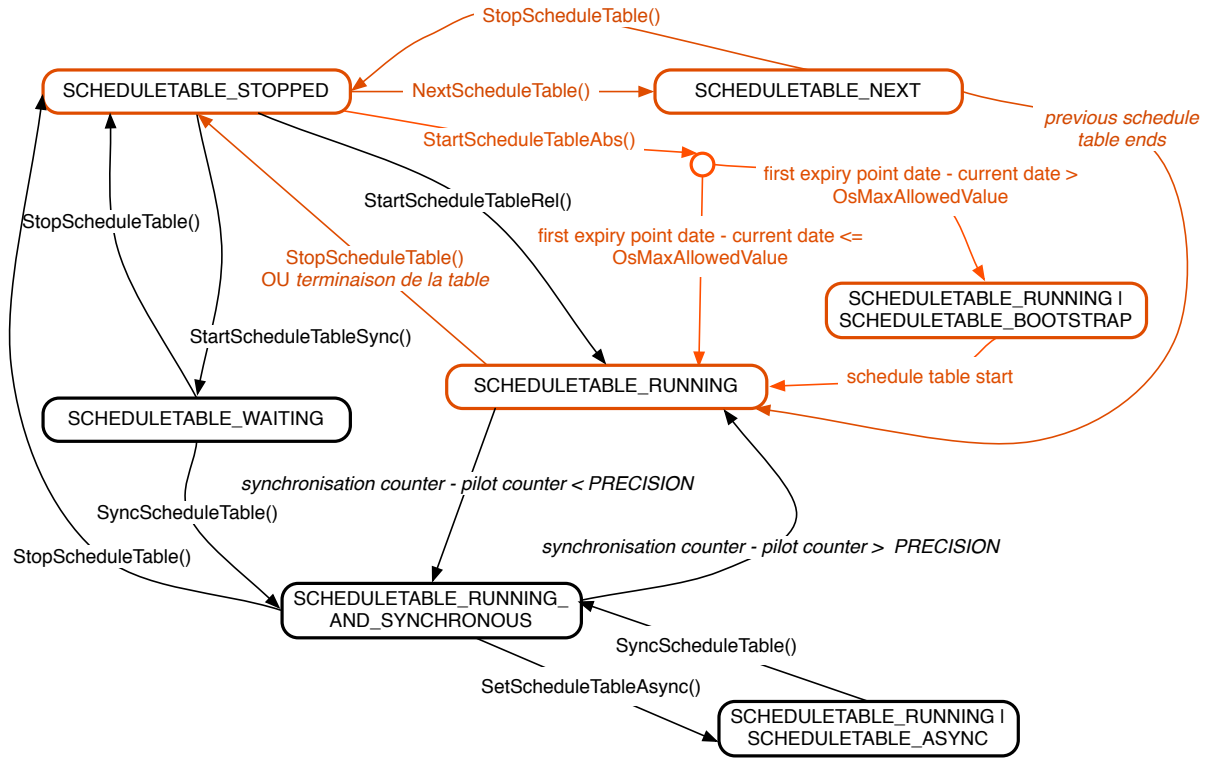


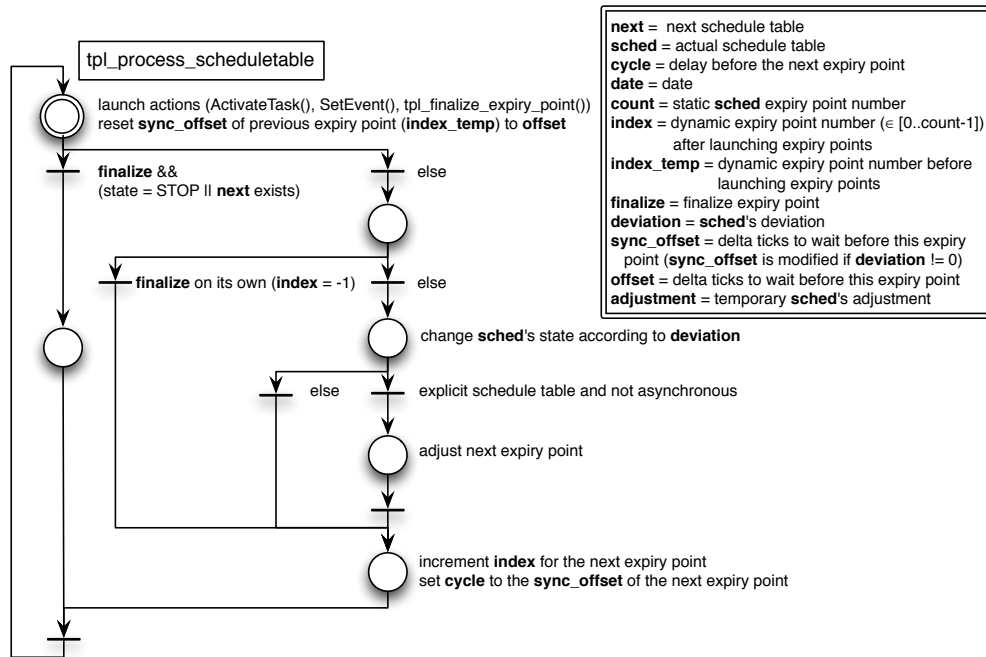
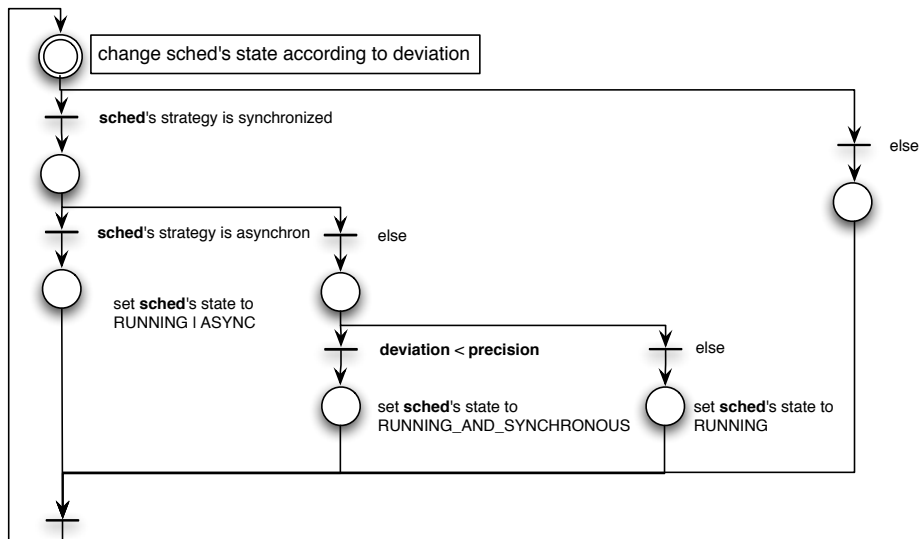
Figure 5.3: States of a schedule table in Trampoline.

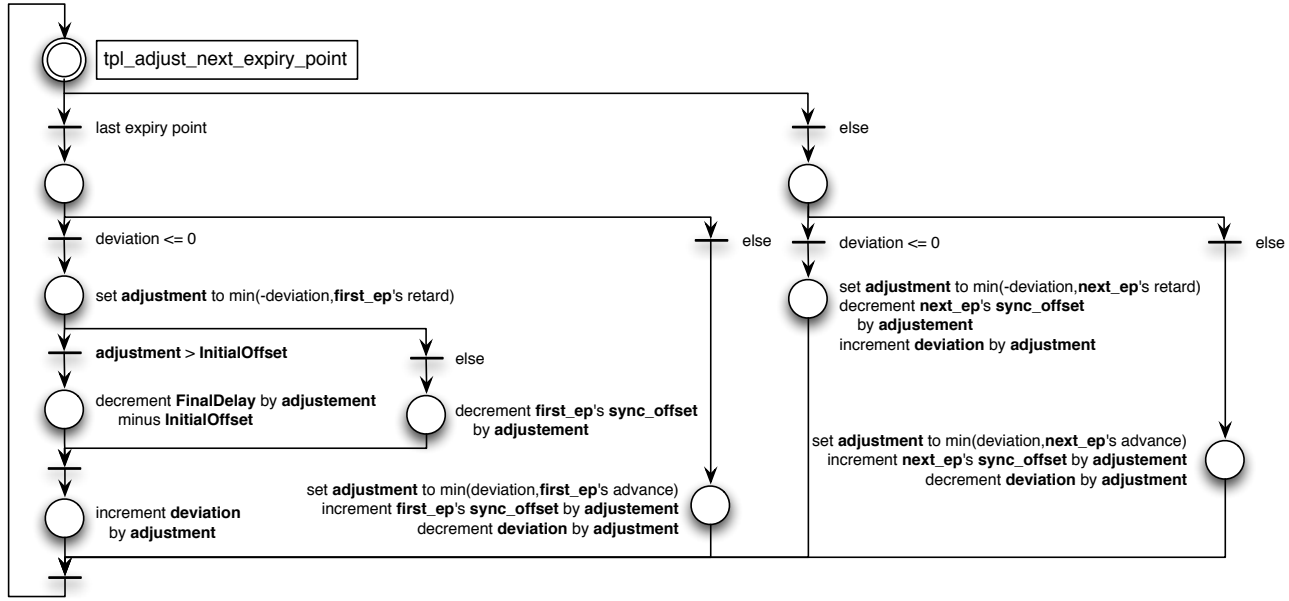
5.2 Processing a Schedule Table

In the same time of producing the schedule tables expiry points, GOIL adds one expiry point more than the number of expiry point declared in the OIL file : the "finalize" expiry point (see Figure 5.2). Indeed, the RUNNING state of a "nexted" schedule table should be set at the finalize expiry point, thus, this expiry point has to be inserted. Moreover, for a periodic schedule table, the "finalize" expiry point helps to launch the first expiry point of the next period.

To process a **synchronized** schedule table, the schedule table's state has to be updated each expiry point and the next expiry point has to be adjusted according to the schedule table's deviation each expiry point too.

A schedule table is a time object, like an alarm. `tpl_processing_scheduledtable()` is called by each expiry point (before activating a task, setting an event or finalizing a schedule table via `tpl_finalize_expiry_point()`). The state machine of this function is shown in the Figure 5.4.

Figure 5.4: *tpl_process_scheduledtable*'s state machine.



`tpl_finalize_expiry_point()` state machine is shown in Figure 5.5 below.

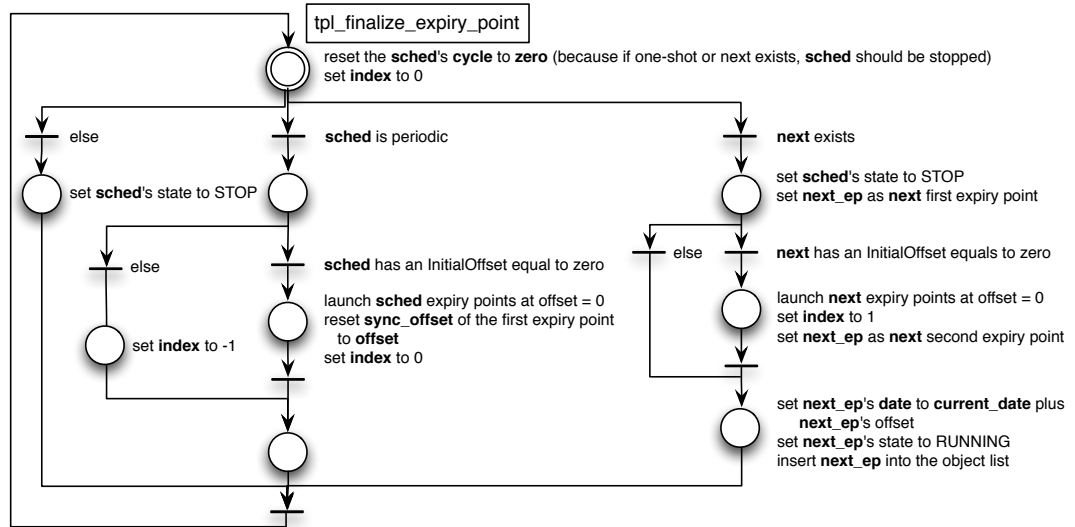


Figure 5.5: `tpl_finalize_expiry_point`'s state machine.

The communication library

6.1 Internals

System generation and compilation

Trampoline is a static operating system. This means all the objects (tasks, ISR, ...) are known at compile time. This way, an application is made of tasks' code and ISRs' code, application data, and statically initialized descriptor for each object the operating system manages. A system generation tool, like goil, generates these descriptors in C files from an application configuration described in OIL or in XML. After that the Trampoline source code, the generated files and the application source code are compiled and linked together to produce an executable file as shown in figure 7.1.

7.1 The generated files

The following files are generated by goil from the OIL file or should be generated if you use a different system configuration tool. More information may be found in part ??.

File name	Usage
<code>tpl_app_define.h</code>	This file contains all the configuration macros (see section 7.2) and is included in all the Trampoline files to trigger conditional compilation. goil generates this file using the ' <code>tpl_app_define_h.goilTemplate</code> ' template file.
<code>tpl_app_config.h</code>	This file contains the declarations of the constants and functions required by the OSEK and Autosar standard (like <code>OSMAXALLOWEDVALUE_x</code> , <code>OSTICKSPERBASE_x</code> or <code>OSMINCYCLE_x</code> constants for counter x). goil generates this file using the ' <code>tpl_app_config_h.goilTemplate</code> ' template file.
<code>tpl_app_config.c</code>	This file contains the definitions of the constants and functions required by the OSEK and Autosar standard and the definitions of object descriptors used by Trampoline (see section ??) goil generates this file using the ' <code>tpl_app_config_c.goilTemplate</code> ' template file.

<code>tpl_app_custom_types.h</code>	Some data types used by Trampoline are not statically defined. They are generated to fit size or performance criterions. For instance, the type used for a <code>TaskType</code> may be a byte if there is less than 256 tasks in the system and a word otherwise. This file defined these data types.
<code>tpl_service_ids.h</code>	This file is generated only if Trampoline is compiled with service calls implemented using a system call. It contains all the identifiers of the services used by the application according to the configuration. goil generates this file using the ' <code>tpl_service_ids_h.goilTemplate</code> ' template file.
<code>tpl_dispatch_table.c</code>	This file is generated only if Trampoline is compiled with service calls implemented using a system call. It contains the dispatch table definition. See section ???. goil generates this file using the ' <code>tpl_dispatch_table_c.goilTemplate</code> ' template file.
<code>tpl_invoque.S</code>	This file is generated only if Trampoline is compiled with service calls implemented using a system call. It contains the API functions for system services. See section ???. The extension (here .S) may change according to the assembler used. goil generates this file using the ' <code>tpl_invoque.goilTemplate</code> ' and ' <code>service_call.goilTemplate</code> ' template files.
<code>MemMap.h</code>	This file is generated only if memory mapping is enabled. It contains macros for compiler abstraction memory mapping of functions and data as defined in the Autosar standard [3]. goil generates this file using the ' <code>MemMap_h.goilTemplate</code> ' template file.
<code>Compiler.h</code>	This file is generated only if memory mapping is enabled. It contains macros for the compiler abstraction of functions and pointer qualifier as defined in the Autosar standard [2]. goil generates this file using the ' <code>Compiler_h.goilTemplate</code> ' template file.
<code>Compiler_Cfg.h</code>	This file is generated only if memory mapping is enabled. It contains macros for the compiler abstraction configuration as defined in the Autosar standard [2]. goil generates this file using the ' <code>Compiler_Cfg_h.goilTemplate</code> ' template file.
<code>script.ld</code>	This file is generated only if memory mapping is enabled. It contains a link script to map the executable in the target memory. goil generates this file using the ' <code>script.goilTemplate</code> ' template file.

The following sections give details about the content of these files.

7.2 The Configuration Macros

Trampoline can be compiled with various options. These options are controlled by setting the appropriate preprocessor configuration macros. These macros are usually set by goilusing the template found in '`tpl_app_define_h.goilTemplate`' file to produce the '`tpl_app_define.h`' file that is included by the files of Trampoline. However, a different generation tool may be used and it should comply to the specification presented in the following tables. When Trampoline is compiled, the coherency and consistency of the configuration macros are checked, by using the preprocessor macros located in the '`tpl_config_check.h`' file, to ensure they correspond to a supported configuration.

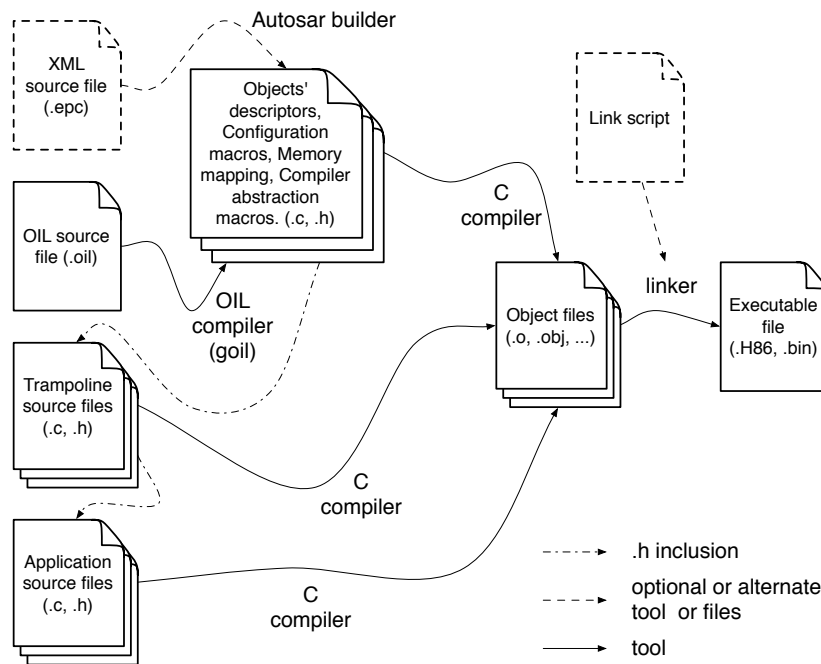


Figure 7.1: Build process of an application with Trampoline. Starting from the left, the .c and .h corresponding to the application description given in OIL (or XML) are generated by goil (or another system generation tool, for instance an Autosar compliant one) and compiled using a C compiler. Trampoline source files are compiled too and include .h from the description for configuration purpose (see section 7.2). Application files are compiled and include .h files from Trampoline. All the object files are then linked together using an optional link script generated by goil or provided with the application.

3 kinds of configuration macros are used: boolean macros, numerical macros, symbol macros and string macros. Boolean macros may take 2 values: **YES** or **NO**. All macros should be defined, Trampoline does not use the **#ifdef** or **\#ifndef** scheme to limit the occurrences of unwanted misconfigurations except to prevent multiple inclusions of the same header file.

7.2.1 Number of objects macros

These macros gives the number of objects of each kind (tasks, ISRs, resources, ...) and other values. They are used in Trampoline to check the validity of the various identifiers and to define tables of the corresponding size.

Macro	Kind	Effect
PRIO_LEVEL_COUNT	Integer	The number of priority levels used in the system.
TASK_COUNT	Integer	The number of tasks (basic and extended) used in the system.
EXTENDED_TASK_COUNT	Integer	The number of extended tasks used in the system.
ISR_COUNT	Integer	The number of ISR category 2 used in the system.
ALARM_COUNT	Integer	The number of alarms used in the system.
RESOURCE_COUNT	Integer	The number of resources used in the system.
SEND_MESSAGE_COUNT	Integer	The number of send messages used in the system.
RECEIVE_MESSAGE_COUNT	Integer	The number of receive messages used in the system.
SCHEDTABLE_COUNT	Integer	The number of schedule tables used in the system. This macros is only used when WITH_AUTOSAR is set to YES .
COUNTER_COUNT	Integer	The number of counters used in the system. This macros is only used when WITH_AUTOSAR is set to YES .
APP_COUNT	Integer	The number of OS applications used in the system. This macros is only used when WITH_AUTOSAR is set to YES .
TRUSTED_FCT_COUNT	Integer	The number of trusted functions used in the system. This macros is only used when WITH_AUTOSAR is set to YES .
RES_SCHEDULER_PRIORITY	Integer	The priority of the RES_SCHEDULER resource. This should be equal to the highest priority among the tasks.

7.2.2 Error Handling Macros

Error handling related macros are used to configure what kind of error Trampoline checks and what extra processing is done when an error is encountered.

Macro	Kind	Effect
WITH_OS_EXTENDED	Bool	When set to YES , Trampoline system services perform error checking on their arguments. WITH_OS_EXTENDED is set to YES with a STATUS = EXTENDED and is set to NO with a STATUS = BASIC in the OIL OS object.
WITH_ERROR_HOOK	Bool	When set to YES , the ErrorHook() function is called if an error occurs. WITH_ERROR_HOOK is set to YES/NO with a ERRORHOOK = TRUE/FALSE in the OIL OS object.

WITH_USEGETSERVICEID	Bool	When set to YES, Trampoline system services store the id of the current service. This id may be retrieved in the <code>ErrorHook()</code> function by using the <code>OSErrorGetServiceId()</code> macro. WITH_USEGETSERVICEID is set to YES/NO with a USEGETSERVICEID = TRUE/FALSE in the OIL OS object.
WITH_USEPARAMETERACCESS	Bool	When set to YES, Trampoline system services store the arguments of the current service. These arguments may be retrieved in the <code>ErrorHook()</code> function by using the ad-hoc access macros (see WITH_USEGETSERVICEID above). WITH_USEPARAMETERACCESS is set to YES/NO with a USEPARAMETERACCESS = TRUE/FALSE in the OIL OS object.
WITH_COM_ERROR_HOOK	Bool	When set to YES, the communication error hook is called when error occurs in the communication sub-system. This macro is only available when WITH_COM is set to YES.
WITH_COM_USEGETSERVICEID	Bool	When set to YES, Trampoline/COM system services store the id of the current service. This id may be retrieved in the <code>COMErrorHook()</code> function by using the <code>COMErrorGetServiceId()</code> macro. WITH_COM_USEGETSERVICEID is set to YES/NO with a COMUSEGETSERVICEID = TRUE/FALSE in the OIL COM object.
WITH_COM_USEPARAMETERACCESS	Bool	When set to YES, Trampoline/COM system services store the arguments of the current service. These arguments may be retrieved in the <code>COMErrorHook()</code> function by using the ad-hoc access macros (see ??). WITH_COM_USEPARAMETERACCESS is set to YES/NO with a COMUSEPARAMETERACCESS = TRUE/FALSE in the OIL COM object.
WITH_COM_EXTENDED	Bool	When set to YES, Trampoline/COM system services perform error checking on their arguments. WITH_COM_EXTENDED is set to YES with a COMSTATUS = EXTENDED and is set to NO with a COMSTATUS = BASIC in the OIL COM object.

7.2.3 Protection Macros

Protection macros deal with protection facilities provided by the AUTOSAR standard.

Macro	Kind	Effect
WITH_MEMORY_PROTECTION	Bool	When set to YES, Trampoline enables the memory protection facility. This is only supported on some ports (MPC5510 and ARM9 at time of writing). Memory protection requires the memory mapping and the use of system call. WITH_MEMORY_PROTECTION is set to YES/NO with the MEMORY_PROTECTION attribute of MEMMAP object (see ??) set to TRUE/FALSE.

WITH_TIMING_PROTECTION	Bool	When set to YES, Trampoline enables the timing protection facility. WITH_TIMING_PROTECTION is set to YES if the AUTOSAR-SC is 2 or 4 (see ??) and a least one of the objects specifies a timing protection related attribute in the OIL file.
WITH_PROTECTION_HOOK	Bool	When set to YES, Trampoline calls the ProtectionHook() with the appropriate argument when a protection fault occurs. WITH_PROTECTION_HOOK is set to YES with a PROTECTIONHOOK = TRUE in the OIL OS object.
WITH_STACK_MONITORING	Bool	When set to YES, Trampoline enables the stack monitoring. Each time a context switch occurs, the stack pointer is checked. If the stack pointer is outside the stack zone of the process, a fault occurs. WITH_STACK_MONITORING is set to YES with a STACKMONITORING = TRUE in the oil OS object.

7.2.4 Hook call macros

Hook call macros control whether a hook is called or not.

Macro	Kind	Effect
WITH_ERROR_HOOK	Bool	see 7.2.2
WITH_PRE_TASK_HOOK	Bool	When set to YES, each time a task is scheduled, the function PreTaskHook() is called. WITH_PRE_TASK_HOOK is set to YES/NO with a PRETASKHOOK = TRUE/FALSE in the OIL OS object.
WITH_POST_TASK_HOOK	Bool	When set to YES, each time a task is descheduled, the function PostTaskHook() is called. WITH_POST_TASK_HOOK is set to YES/NO with a POSTTASKHOOK = TRUE/FALSE in the OIL OS object.
WITH_STARTUP_HOOK	Bool	When set to YES, the function StartupHook() is called within the StartOS service. WITH_STARTUP_HOOK is set to YES/NO with a STARTUPHOOK = TRUE/FALSE in the OIL OS object.
WITH_SHUTDOWN_HOOK	Bool	When set to YES, the function ShutdownHook() is called within the ShutdownOS service. WITH_SHUTDOWN_HOOK is set to YES/NO with a SHUTDOWNHOOK = TRUE/FALSE in the OIL OS object.
WITH_PROTECTION_HOOK	Bool	see 7.2.3

7.2.5 Miscellaneous macros

Here are the other available macros:

Macro	Kind	Effect
WITH_USERESSCHEDULER	Bool	When set to YES, the RES_SCHEDULER resource is used by at least one process. WITH_USERESSCHEDULER is set to YES/NO with a USERESSCHEDULER = TRUE/FALSE in the OIL OS object.
WITH_SYSTEM_CALL	Bool	When set to YES, services are called by the mean of a system call, also known as a software interrupt (see section ??). WITH_SYSTEM_CALL is set to YES/NO according to the target architecture and requires a memory mapping

WITH_MEMMAP	Bool	When set to YES, a memory mapping is used. A 'MemMap.h' files giving the available memory segments is included and should be generated or provided by the user. goil generates such a file. WITH_MEMMAP is set to YES/NO with a MEMMAP = TRUE/FALSE in the OIL OS object.
WITH_COMPILER_SETTINGS	Bool	When set to YES, the compiler dependent macros are used. 'Compiler.h' and 'Compiler_Cfg.h' files are includes and should be generated or provided by the user. goil generates these files if MEMMAP is TRUE and the COMPILER sub-attribute is set.
WITH_AUTOSAR	Bool	When set to YES, Trampoline contains additional system services, code and declarations related to the AUTOSAR standard. For instance, the counter descriptor includes the counter type (hardware or software). WITH_AUTOSAR is set to YES/NO when at least one AUTOSAR object is present in the system configuration (OIL file for instance).
TRAMPOLINE_BASE_PATH	String	The path to Trampoline root directory.
AUTOSAR_SC	Integer	The AUTOSAR scalability class ranging from 0 to 4. 0 means OSEK
WITH_OSAPPLICATION	Bool	When set to YES, OS Application are used.
WITH_TRACE	Bool	When set to YES, the tracing of the operating system is enabled.
TRACE_TASK	Bool	When set to YES, task (de)scheduling events are traced. Only available if WITH_TRACE is set to YES.
TRACE_ISR	Bool	When set to YES, ISR category 2 (de)scheduling events are traced. Only available if WITH_TRACE is set to YES.
TRACE_RES	Bool	When set to YES, resources get and release are traced. Only available if WITH_TRACE is set to YES.
TRACE_ALARM	Bool	When set to YES, alarm activities are traced. Only available if WITH_TRACE is set to YES.
TRACE_U_EVENT	Bool	When set to YES, user events are traced. Only available if WITH_TRACE is set to YES.
TRACE_FORMAT	Symbol	Trace format. A function named <code>tpl_trace_format_<TRACE_FORMAT></code> is expected. Only available if WITH_TRACE is set to YES.
TRACE_FILE	String	File name where the trace is stored. Usable on Posix target only. Only available if WITH_TRACE is set to YES.
WITH_IT_TABLE	Bool	When set to YES, the external interrupts are dispatched using a table of fonction pointers.
WITH_COM	Bool	When set to YES, internal communication is used.
TPL_COMTIMEBASE	Integer	The COMTIMEBASE expressed in nanoseconds.
WITH_COM_STARTCOMEXTENSION	Bool	When set to YES, the communication extension function is called.

7.3 Application configuration

The application configuration is generated by goil using the template found in 'tpl_app_config.h.goilTemplate' file and 'tpl_app_config.c.goilTemplate' file to produce the 'tpl_app_define.h' and 'tpl_app_define.c' files.

7.3.1 Counter related constants declaration

The 'tpl_app_config.h' files contains the counters related constants: those of the SystemCounter¹ and those of the counters defined by the user. The SystemCounter constants are located in the generated files because the SystemCounter default attributes may be modified by the user in the OIL or XML file. The constants of a user defined counter are declared as follow:

```
extern CONST(tpl_tick, OS_CONST) OSTICKSPERBASE_<counter name>;
extern CONST(tpl_tick, OS_CONST) OSMAXALLOWEDVALUE_<counter name>;
extern CONST(tpl_tick, OS_CONST) OSMINCYCLE_<counter name>;
```

Where <counter name> is obviously the name given to the counter in the configuration. For the SystemCounter, the following constants are declared:

7.3.2 Events definition

The 'tpl_app_config.c' file should contain the event mask definitions. For each event defined in the configuration, the following lines should appear:

```
#define API_START_SEC_CONST_UNSPECIFIED
#include "tpl_memmap.h"

#define <event name>_mask <mask value>
CONST(EventMaskType, AUTOMATIC) <event name> = <event name>_mask;

#define API_STOP_SEC_CONST_UNSPECIFIED
#include "tpl_memmap.h"
```

Where <event name> is the name given to the event in the configuration and <mask value> is the value set by the user in the configuration or, when set to AUTO, the value computed by the generation tool.

7.3.3 Standard resources definition

Standard resources need the definition of an identifier used to reference the resource in a system service (GetResource() and ReleaseResource()) and an instance of a tpl_resource structure (see ??). This is done with the following definitions:

```
#define API_START_SEC_CONST_UNSPECIFIED
#include "tpl_memmap.h"

#define <resource name>_id <resource id>
CONST(ResourceType, AUTOMATIC) <resource name> = <resource name>_id;
```

¹the default counter of an OSEK operating system

```

#define API_STOP_SEC_CONST_UNSPECIFIED
#include "tpl_memmap.h"

#define OS_START_SEC_VAR_UNSPECIFIED
#include "tpl_memmap.h"

VAR(tpl_resource, OS_VAR) <resource name>_rez_desc = {
    /* ceiling priority of the resource */ <resource priority>,
    /* owner previous priority           */ 0,
    /* owner of the resource             */ INVALID_PROC_ID,
#if WITH_OSAPPLICATION == YES
    /* OS Application id                 */ <resource application id>,
#endif
    /* next resource in the list         */ NULL
};

#define OS_STOP_SEC_VAR_UNSPECIFIED
#include "tpl_memmap.h"

```

Where <resource name> is the name given to the resource in the configuration, <resource priority> is the priority of the resource that is computed by the generation tool and is the maximum priority of the processes that use the resource and <resource application id> is the identifier of the OS Application the resource belongs to. Since this field is protected by *WITH_OSAPPLICATION*, it may be leaved empty when no OS Application is used.

<resource id> ranges from 0 to the number of standard resources minus 1. Once every standard resource descriptor is defined, a table gathering pointers to the resource descriptors and indexed by the resource id has to be defined. This table is used by system services to get the resource descriptor from the resource id. Suppose 3 standard resource, *motor1*, *motor2* and *dac* has been defined and RES_SCHEDULER is used, the table should be as follow:

```

#define OS_START_SEC_CONST_UNSPECIFIED
#include "tpl_memmap.h"
CONSTP2VAR(tpl_resource, AUTOMATIC, OS_APPL_DATA)
tpl_resource_table[RESOURCE_COUNT] = {
    &motor1_rez_desc,
    &motor2_rez_desc,
    &dac_rez_desc,
    &res_sched_rez_desc
};
#define OS_STOP_SEC_CONST_UNSPECIFIED
#include "tpl_memmap.h"

```

&res_sched_rez_desc, the pointer to the resource descriptor of RES_SCHEDULER should always be the last element of the table. If RES_SCHEDULER is not used, simply remove it from the table.

7.3.4 Tasks definition

Each task needs an identifier to reference a task un a system service (*ActivateTask()*, *ChainTask()*, *GetTaskState()*, *SetEvent()* and *GetEvent()*) and the declaration of the task function. The following definitions should appear for each task:

```

#define API_START_SEC_CONST_UNSPECIFIED

```

```
#include "tpl_memmap.h"

#define <task name>_id <task id>
CONST(TaskType, AUTOMATIC) <task name> = <task name>_id;

#define API_STOP_SEC_CONST_UNSPECIFIED
#include "tpl_memmap.h"

#define APP_Task_<task name>_START_SEC_CODE
#include "tpl_memmap.h"

FUNC(void, OS_APPL_CODE) <task name>_function(void);

#define APP_Task_<task name>_STOP_SEC_CODE
#include "tpl_memmap.h"
```

Where <task name> is the name given to the task in the configuration and <task id> is the identifier of the task computed by the system generation tool. Task ids should range from 0 to the number of tasks minus 1. In addition, id allocation must start with extended tasks first and basic task after. In addition an instance of the static task descriptor must be provided:

```
#define OS_START_SEC_CONST_UNSPECIFIED
#include "tpl_memmap.h"
CONST(tpl_proc_static, OS_CONST) <task name>_task_stat_desc = {
    /* context */ <task name>_CONTEXT,
    /* stack */ <task name>_STACK,
    /* entry point (function) */ <task name>_function,
    /* internal ressource */ <internal resource>,
    /* task id */ <task name>_id,
#if WITH_OSAPPLICATION == YES
    /* OS application id */ <application>,
#endif
    /* task base priority */ <task priority>,
    /* max activation count */ <task activation>,
    /* task type */ <task type>
#if WITH_AUTOSAR_TIMING_PROTECTION == YES
    /* pointer to the timing
       protection descriptor */ ,<timing protection>
#endif
};
#define OS_STOP_SEC_CONST_UNSPECIFIED
#include "tpl_memmap.h"
```

Where <task name> is the name given to the task in the configuration. <internal resource> may be one of the following:

- a pointer to the internal resource descriptor (see ??) if an internal resource has been defined in the configuration;
- a pointer to the scheduler internal resource if the task has been defined as non-preemptable in the configuration;
- NULL if none of the above cases apply.

<application> is the id of the OS Application the task belongs to when OS Application are used or, when they are not used, nothing at all. <task priority> is the priority of the task as computed by the system generation tool. <task activation> is the maximum number of task activation allowed as defined in the configuration. <task type> may be **EXTENDED** or **BASIC**. <timing protection> is a pointer to the timing protection descriptor or **NULL** if no timing protection is defined for the task.

Also an instance of the dynamic task descriptor must be provided:

```
#define OS_START_SEC_VAR_UNSPECIFIED
#include "tpl_memmap.h"

VAR(tpl_proc, OS_VAR) <task name>_task_desc = {
    /* resources */ NULL,
#ifdef WITH_MEMORY_PROTECTION == YES
    /* if > 0 the process is trusted */ <trusted count>,
#endif /* WITH_MEMORY_PROTECTION */
    /* activate count */ 0,
    /* task priority */ <task priority>,
    /* task state */ <task state>
#ifdef WITH_AUTOSAR_TIMING_PROTECTION == YES
    /* activation allowed */ ,TRUE
#endif
};

#define OS_STOP_SEC_VAR_UNSPECIFIED
#include "tpl_memmap.h"
```

Where <task name> is the name given to the task in the configuration. <trusted count> is 0 if the task belongs to a non trusted OS Application and 1 if the tasks belongs to a trusted OS Application. <task priority> is the priority of the task as computed by the system generation tool. <task state> is the initial state of the task and must be set to **AUTOSTART** or **SUSPENDED**.

If the task is an **EXTENDED** one, an event mask descriptor is added:

```
VAR(tpl_task_events, OS_VAR) <task name>_task_evts = {
    /* event set */ 0,
    /* event wait */ 0
};
```

Where <task name> is the name given to the task in the configuration.

Ports details

8.1 PowerPC

8.1.1 System services

The PowerPC port uses the `sc` software interrupt to call system services [1]. `sc` stands for System Call. It saves the current *PC* in *SRR0* register and the current *MSR* in *SRR1* register and jump to the System Call handler.

The id of the system service to call is given in the *r0* register and *r0* save and restore are added around. For instance, the following listing gives the `ActivateTask` service code. These function are generated from templates by goil (see 7.1) and are part of the *invoque* layer (see ??):

```
.global ActivateTask
ActivateTask:
    subi    r1,r1,4                /* make room on stack */
    stw     r0,0(r1)              /* save r0 */
    li      r0,OSServiceId_ActivateTask /* load r0 with the id */
    sc                      /* system call */
    lwz     r0,0(r1)              /* restore r0 */
    addi    r1,r1,4                /* restore stack */
    blr                      /* return */

.type ActivateTask,@function
.size ActivateTask,,$-ActivateTask
```

When the System Call begin execution, the process stack has the mapping depicted in figure 8.1.

8.1.2 Dispatching the service call

The System Call handler is usually located in the `0C00H` exception handler but, depending on the CPU kind, it may be located elsewhere. Since the available memory for the interrupt or exception handler may vary, a jump is made to the `tpl_sc_handler`.

`tpl_sc_handler` performs the following tasks:

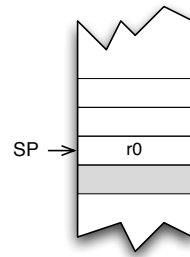


Figure 8.1: Process stack mapping at the beginning of the System Call handler. The grayed zone represents an unknown content depending on from where the service was called.

1. saves additional registers to be able to work
2. disables memory protection
3. switches to kernel stack if needed
4. calls the service
5. performs a context switch if needed and programs the MPU.
6. switches back to the process stack if needed
7. enable memory protection
8. restore registers
9. get back to the process



Currently the PowerPC port does not support tasks that use floating point registers

Saving additional registers

The following registers are saved: *lr*, *cr*, *r11* and *r12*. In fact, it should be not necessary to save *r11* and *r12* because these registers are volatile as defined in the PowerPC EABI [4] but we prefer a conservative approach. Register saving is done by the following code at start of the `tpl_sc_handler` and the mapping of the process stack is depicted at figure 8.2:

```
subi    r1,r1,PS_FOOTPRINT    /* Make room on stack */

stw     r11,PS_R11(r1)        /* Save r11          */
stw     r12,PS_R12(r1)        /* Save r12          */
mflr    r11
stw     r11,PS_LR(r1)         /* Save lr           */
mfcr     r11
stw     r11,PS_CR(r1)         /* Save cr           */
```

Disabling memory protection

This part of the dispatch layer is done in the `tpl_enter_kernel` function and is assembled only if `WITH_MEMORY_PROTECTION` is set to `YES`. After saving the *lr*, the `tpl_kernel_mp` function is called and does the actual job. At last *lr* is restored.

```
#if WITH_MEMORY_PROTECTION == YES
/*
 * Switch to kernel mem protection scheme
```

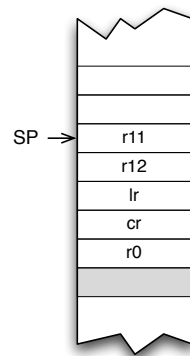



Figure 8.2: Process stack mapping after additional registers have been saved by the beginning of the System Call handler.

```

    */
    subi   r1,r1,4
    mflr   r11
    stw     r11,0(r1)      /* save lr on the current stack */
    bl      tpl_kernel_mp  /* disable memory protection */
    lwz     r11,0(r1)      /* restore lr */
    mtlr    r11
    addi    r1,r1,4
#endif

```

Switching to the kernel stack

Once the dispatch layer has saved the registers it uses and has switched to the kernel memory protection scheme, it switches to the kernel stack. However the kernel stack could be used already because a call to a `PreTaskHook` or a `PostTaskHook` is done on the kernel stack and such a hook may call a service. So the dispatch layer is reentrant. The number of reentrant calls is counted by the `tpl_reentrancy_counter`. In addition the process stack pointer (`r1`), `SRR0` and `SRR1` are saved in the kernel stack. The kernel stack mapping is shown in figure 8.3. For a reentrant call, the same frame is built over the current one. The switch to the kernel stack is done as follows:

```

/*
 * Check the reentrancy counter value and increment it
 * if the value is 0 before the inc, then we switch to
 * the system stack.
 */
lis     r11,TPL_HIG(tpl_reentrancy_counter)
ori     r11,r11,TPL_LOW(tpl_reentrancy_counter)
lwz     r12,0(r11)      /* get the value of the counter */
cmpwi   r12,0
addi    r12,r12,1
stw     r12,0(r11)
bne     no_stack_change

/*
 * Switch to the kernel stack
 *
 * Get the pointer to the bottom of the stack

```

```

    */
    lis    r11,TPL_HIG(tpl_kernel_stack_bottom)
    ori    r11,r11,TPL_LOW(tpl_kernel_stack_bottom)
    stw    r1,KS_SP-KS_FOOTPRINT(r11) /* save the sp of the caller */
    mr     r1,r11                    /* set the kernel stack */

no_stack_change:
    /*
     * make space on the stack to call C functions
     */
    subi   r1,r1,KS_FOOTPRINT

```

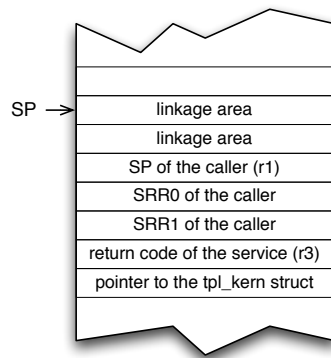


Figure 8.3: Kernel stack mapping after allocation.

Calling the service

Since the registers used to pass parameters to a function, that is *r3* to *r10* as documented in [4], have not been changed until now, calling the function that implements the service respects the register usage conventions.

The first thing to do is to get the function pointer corresponding to the service id. The service id is in *r0* as explained in 8.1.1 and is used as an index to the *tpl_dispatch_table*.

```

    slwi   r0,r0,2                                /* compute the offset */
    /*
     * load the ptr to the dispatch table
     */
    lis    r11,TPL_HIG(tpl_dispatch_table)
    ori    r11,r11,TPL_LOW(tpl_dispatch_table)
    lwzx   r11,r11,r0                             /* get the ptr to the service */
    mtlr   r11                                    /* put it in lr for future use */

```

The second thing to do is to reset the *tpl_need_switch* flag that triggers a context switch. This flag (a byte) is located in the *tpl_kern* kernel struct. This is done as follow:

```

    lis    r11,TPL_HIG(tpl_kern)
    ori    r11,r11,TPL_LOW(tpl_kern)
    stw    r11,KS_KERN_PTR(r1)                    /* save the ptr for future use */
    li     r0,NO_NEED_SWITCH
    stb    r0,20(r11)

```

In the future *tpl_kern* will be reused, so its address is saved in the kernel stack.

Then, to allow reentrancy for a service call in a hook, the *RI* bit of the *MSR* is set to 1. Without that, a *sc* cannot be properly executed.

```
mfmsr r11
ori    r11,r11,RI_BIT_1
mtmsr r11
```

At last, the service is called:

```
blrl
```

Context switch

The *tpl_need_switch* flag that has been possibly modified by the service is now checked to do a context switch if needed.

```
lwz    r11,KS_KERN_PTR(r1) /* get back the tpl_kern address */
lbz    r12,20(r11)         /* get the tpl_need_switch flag */
andi.  r0,r12,NEED_SWITCH /* check if a switch is needed */
beq     no_context_switch
```

A context switch is performed in 3 steps. The first one is the context save of the process that loses the CPU. This step is optional because if the service was a *TerminateTask* or a *ChainTask*, the context needs not to be saved. This information is in the *tpl_need_switch* flag. Before doing the actual context save, the return value of the service must be saved in the proper location of the kernel stack. The *tpl_save_context* function will read it from this location and expects a pointer to the context saving area or the process in *r3*. *s_old*, the address of the context saving area, is in another member of *tpl_kern*. At the end, the *tpl_kern* address is reread because *r11* has been destroyed in *tpl_save_context*.

```
stw    r3,KS_RETURN_CODE(r1) /* save the return value */
andi.  r0,r12,NEED_SAVE      /* r12 contains tpl_need_switch */
beq     no_save
lwz    r3,0(r11)             /* r11 contains the tpl_kern address */
bl      tpl_save_context      /* and s_old is put into r3 */
lwz    r11,KS_KERN_PTR(r1)   /* get back tpl_kern address */
```

The second step consists in loading the configuration of memory protection for the process that gets the CPU by calling the *tpl_set_process_mp* function. This function expects the id of the process in *r3*. Again this id is located in member *proc_id* of *tpl_kern*. This is done only if *WITH_MEMORY_PROTECTION* is YES.

```
#if WITH_MEMORY_PROTECTION == YES
lwz    r3,16(r11) /* get the id of the process which get the cpu */
bl      tpl_set_process_mp /* set the memory protection scheme */
#endif
```

The third step loads the context of the process that gets the CPU. The address of *tpl_kern* is loaded into *r11* because it has been destroyed in *tpl_set_process_mp*, *s_running*, the address of the context saving area of the current process is loaded into *r3* and *tpl_load_context* is called. At last, *r3* is restored.

```

lwz    r11,KS_KERN_PTR(r1)
lwz    r3,4(r11)                /* get s_running */
bl     tpl_load_context
lwz    r3,KS_RETURN_CODE(r1)

```

Switching back to the process stack

At this stage, the *SRR0* and *SRR1* registers saved in the kernel stack are restored. The space reserved in the kernel stack is freed. The reentrancy counter is decremented and the stack switches to the process stack if the reentrancy counter is 0.

```

lwz    r11,KS_SRR0(r1)
mtspr  spr_SRR0,r11
lwz    r11,KS_SRR1(r1)
mtspr  spr_SRR1,r11

addi   r1,r1,KS_FOOTPRINT      /* free back space on the stack */

/*
 * The reentrancy counter is decremented. If it reaches
 * 0, the process stack is restored
 */
lis    r11,TPL_HIG(tpl_reentrancy_counter)
ori    r11,r11,TPL_LOW(tpl_reentrancy_counter)
lwz    r12,0(r11)             /* get the value of the counter */
subi   r12,r12,1
stw    r12,0(r11)
cmpwi  r12,0
bne    no_stack_restore

/* Restore the execution context of the caller
   (or the context of the task/isr which just got the CPU) */
lwz    r1,KS_SP-KS_FOOTPRINT(r1) /* Restore the SP and switch
                                   back to the process stack */

```

Enabling memory protection

Then, if memory protection is used, the user scheme is reenabled. The actual works depends on the kind of MPU and is done in *tpl_user_mp*.

```

#if WITH_MEMORY_PROTECTION == YES
    subi   r1,r1,4
    mflr   r11
    stw    r11,0(r1) /* save lr on the current stack */
    bl     tpl_user_mp /* Enable the memory protection */
    lwz    r11,0(r1) /* restore lr */
    mtlr   r11
    addi   r1,r1,4
#endif

```

Restoring registers

Registers saved at stage 1 on the process stack are restored and the stack is freed.

```
lwz    r11,PS_CR(r1)
mtcr   r11
lwz    r11,PS_LR(r1)
mtlcr  r11
lwz    r12,PS_R12(r1)
lwz    r11,PS_R11(r1)

addi   r1,r1,PS_FOOTPRINT
```

Getting back to the process

At last, the dispatch layer is exited using a `rfi`.

```
rfi                                     /* return from interrupt */
```

8.1.3 Interrupt handler

8.1.4 The CallTrustedFunction service

The `CallTrustedFunction` service is implemented by the `tpl_call_trusted_function_service` function. This function is a special case of service because the kernel stack and the process stack have to be modified. In addition, an `ExitTrustedFunction` service is implemented to restore the process stack when the trusted function exits. Both services have to be written in assembly language since C does not allow to explicitly modify the stack.

`tpl_call_trusted_function_service` performs the following steps:

1. check the trusted function id is within the allowed range
2. increment the trusted counter of the calling process
3. build a frame on the process stack to store the registers pushed by a service call except for `r0` and for `SRR0` and `SRR1`; put the address of `ExitTrustedFunction` in the `lr` location in the process stack; save `SRR0` and `SRR1` in the process stack
4. get the trusted function address and put it in `SRR0`
5. go back to the dispatch layer

Checking the trusted function id

The id of the trusted function is checked to avoid to call a function at an arbitrary address.

```
mov     r11,r3                          /* save r3 in r11 b/c it will be destroyed */
cmpw    r3,TRUSTED_FCT_COUNT            /* check the id of the trusted function */
ori     r3,r0,E_OS_SERVICEID            /* E_OS_SERVICEID return code */
bge     invalid_trusted_fct_id
mov     r3,r11                          /* restore r3 if trusted function id ok */
```

Incrementing the trusted counter

The trusted counter of the process is incremented each time a trusted function is called. When the trusted counter is > 0 , the process is trusted. In such a case, the dispatch layer does not enable memory protection when scheduling the process so it has an unlimited access to the whole addressing space.

```
lwz    r11,KS_KERN_PTR(r1) /* get the ptr to tpl_kern          */
lwz    r11,12(r11)         /* get the ptr to the running process desc */
lwz    r12,4(r11)          /* get trusted_count member              */
addi   r12,r12,1           /* increment it                          */
stw    r12,4(r11)          /* put it back in the process desc       */
```

Building the frame

The frame is used to store the calling context of the trusted function and is shown in figure 8.4. The following code builds this frame:

```
/*
 * First get back the process stack pointer
 */
lwz    r11,KS_SP(r1)
/*
 * Make room to prepare the call of the trusted function
 */
subi   r11,r11,PS_TRUSTED_FOOTPRINT_IN
/*
 * store ExitTrustedFunction as the return address
 */
lis    r12,TPL_HIG(ExitTrustedFunction)
ori    r12,r12,TPL_LOW(ExitTrustedFunction)
stw    r12,PS_LR(r11)
/*
 * Update the stack pointer
 */
stw    r11,KS_SP(r1)
/*
 * second get back SRR0 and SRR1 and save them to the process stack
 */
lwz    r12,KS_SRR0(r1)
stw    r12,PS_SRR0_IN(r11)
lwz    r12,KS_SRR1_IN(r1)
stw    r12,PS_SRR1(r11)
```

Setting the trusted function address

The *SRR0* saved by the dispatch layer after the *CallTrustedFunction* is changed to the address of the trusted function. This way, instead of returning to the caller, the trusted function will be executed.

```
lis    r11,TPL_HIG(tpl_trusted_fct_table)
ori    r11,r11,TPL_LOW(tpl_trusted_fct_table)
```

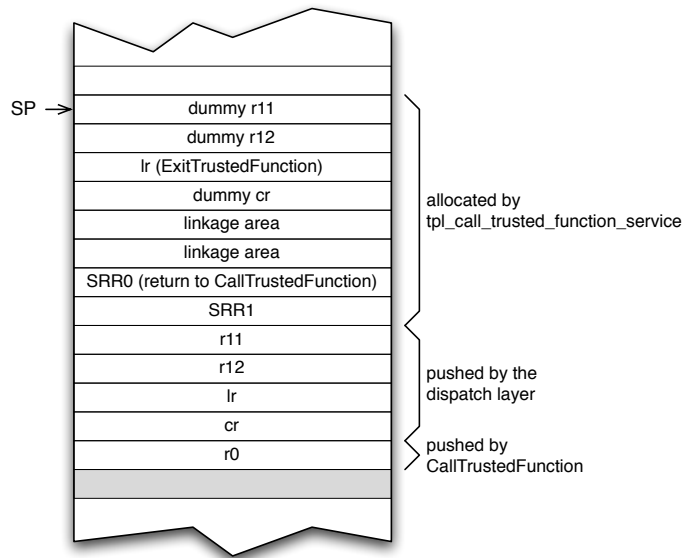


Figure 8.4: Process stack mapping at the end of `tpl_call_trusted_function_service`. `r0`, at the bottom of the stack has been pushed by `CallTrustedFunction`. `cr` to `r11` has been pushed by the dispatch layer. `SRR0` and `SRR1` are saved here by `tpl_call_trusted_function_service` to be able to go back to the calling process. Above, the linkage area allows the trusted function to call functions. Above, a frame that will be used by the dispatch layer to restore an execution context for the trusted function is built.

```
slwi    r0,r3,2
lwzx    r12,r11,r0
stw     r12,KS_SRR0(r1)
```

Going back to the dispatch layer

A simple `blr` goes back to the dispatch layer. The latter cleans up the process stack. Once the trusted function starts execution, the process stack is like that:

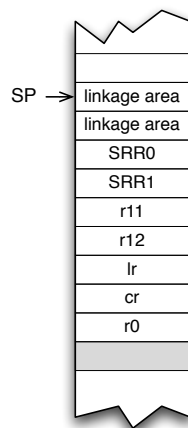


Figure 8.5: Process stack mapping when the trusted function starts its execution.

8.1.5 The ExitTrustedFunction service

When a trusted function finishes, the context of the `CallTrustedFunction` must be restored to return to the caller. `ExitTrustedFunction` does not need to be called explicitly because

its address has been set as the return address of the trusted function by `tpl_call_trusted_function_service`. Calling `ExitTrustedFunction` explicitly may result in an undefined behavior or in the crash of the calling process but see below. The mapping of the process stack at start of `tpl_exit_trusted_function_service` is shown in figure 8.6.

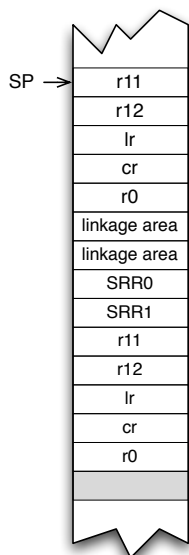


Figure 8.6: *Process stack mapping when the `tpl_exit_trusted_function_service` function starts its execution.*

First, `tpl_exit_trusted_function_service` decrements the trusted counter of the calling process. A particular attention must be given to this point because by building a fake stack frame and calling `Explicitly ExitTrustedFunction` to underflow this counter, a process could get a full access to the memory. So the counter is tested before to avoid to go under 0.

```
lwz    r11,KS_KERN_PTR(r1) /* get the ptr to tpl_kern */
lwz    r11,12(r11)         /* get the ptr to the running process desc */
lwz    r12,4(r11)         /* get trusted_count member */
/*
 * Warning, the trusted counter has to be check (compared to 0) to
 * avoid to decrement it if it is already 0. Without that a process
 * could build an had-hoc stack an call explicitly ExitTrustedFunction
 * to get access to all the memory.
 */
cmpwi  r12,0               /* check it is not already at 0 */
beq    cracker_in_action   /* uh uh */
subi   r12,r12,1           /* decrement it */
stw    r12,4(r11)          /* put it back in the process desc */
```

`tpl_exit_trusted_function_service` has to remove from the process stack the frame that was built by `tpl_call_trusted_function_service`, restore `SRR0` and `SRR1` before returning to the dispatch layer.

`cracker_in_action:`

```
/*
 * get the process stack pointer
 */
```



```

lwz    r11,KS_SP(r1)

/*
 * get back the SRR0 and SRR1
 */
lwz    r12,PS_SRR0_OUT(r11)
stw    r12,KS_SRR0(r1)
lwz    r12,PS_SRR1_OUT(r11)
stw    r12,KS_SRR1(r1)

/*
 * free the process stack and update it in the kernel stack
 */
addi   r11,r11,PS_TRUSTED_FOOTPRINT_OUT
stw    r11,KS_SP(r1)

/*
 * that's all
 */
blr

```

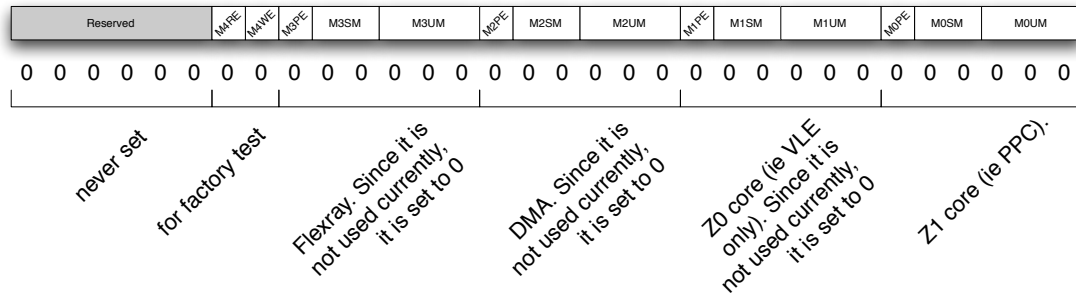
8.1.6 Execution of the OS Applications startup and shutdown hooks

These hooks are executed from the kernel but with the access right of a task belonging to the OS Application. The system generation tool should choose one of the tasks of the OS Application to be used as context to execute the OS Application startup and shutdown hooks. Execution of an OS Application startup hook is done by the `tpl_call_startup_hook_and_resume` function. The argument of this function is a function pointer to the hook. Similarly execution of an OS Application shutdown hook is done by the `tpl_call_shutdown_hook_and_resume` function. These functions end by a call to `NextStartupHook` and `NextShutdownHook` services respectively to cycle through the hooks.

8.1.7 The MPC5510 Memory Protection Unit

The access control rights of the memory region descriptor rules the access of 5 bus masters (labeled from 4 to 0). Unused bus masters are set to the same access right for all the regions. Bus master 4 is used for factory testing only, so the access rights should be set to no access. Bus master 3 is the Flexray controller. Since it is not used in the current version of Trampoline, it is set to no access too. Bus master 2 is the DMA controller and for the same reason it is set to no access. Bus master 1 is the Z0 core. Again it is set to no access.

The access control rights register has the following bit usage:



Bus master 4 is a special case. The 2 bits have the following meaning:

Bit	Meaning
M4RE	If set to 1, bus master 4 may read memory in the region. If 0, no read is allowed
M4WE	If set to 1, bus master 4 may write memory in the region. If 0, no write is allowed

So in our case, these bits are set to 0.

Of course, other bus masters have more sophisticated access right:

Bit	Meaning
MxPE	The PID Enable bit. Set to 0 in our case
MxSM	These 2 bits rules the supervisor mode access control with the following meaning: 00 = <i>rw</i> , 01 = <i>rx</i> , 10 = <i>rw</i> , 11 = <i>same as defined by MxUM</i> . In our case, it is set to 00 for code and constants and to 11 for data.
MxUM	These 2 bits rules the user mode access control. The first bit means <i>r</i> , the second one <i>w</i> and the third one <i>x</i> .

Trampoline uses 4 descriptors:

Descriptor	Usage	MxUM value
MPU_RGD0	Constants and program ¹ .	<i>rw</i> = 00 for supervisor mode, <i>rx</i> = 101 for user mode.
MPU_RGD1	Private variables of the process.	<i>rw</i> = 110 for supervisor and user mode.
MPU_RGD2	Stack of the process.	<i>rw</i> = 110 for supervisor and user mode.
MPU_RGD3	Variables of the OS Application if OS Applications are used.	<i>rw</i> = 110 for supervisor and user mode.

So values of access control bits should be:

Reserved								M4RE	M4WE	M3PE	M3SM	M3UM	M2PE	M2SM	M2UM	M1PE	M1SM	M1UM	M0PE	M0SM	M0UM					
For program and constants:																										
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1		
For variables:																										
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	0

¹This region is set to the whole addressing space. This is not definitive and should be improved because reading a peripheral control register should be protected. So an additional descriptor has to be used to allow the kernel (supervisor mode) a complete access on all the memory space and no access at all for applications (user mode).

So in hexa:

Kind	Value
Program region access	0x00000005
Variable region access	0x0000001E

What happen in case of memory protection violation ?

Two exception handler are used to handle a memory protection violation, one for data access, one for code access.

The Data Storage exception is tied to the IVOR 2 vector (VPR offset = 0x020), see page 8-2 of the *MPC5510 Microcontroller Family Reference Manual*.

The Instruction Storage exception is tied to the IVOR 3 vector (VPR offset = 0x030), see page 8-2 of the *MPC5510 Microcontroller Family Reference Manual*.

However, it appears one of these exceptions is raised when the processor is in user mode. The behavior is different in supervisor mode *to be completed*.

Part II

The Goil system generator

The Goil templates

Goil includes a template interpreter which is used for file generation. Goil generates the structures needed by trampoline to compile the application and may generate other files like a memory mapping file `MemMap.h`, the compiler abstraction files, `Compiler.h` and `Compiler_cfg.h` and a linker script depending on which attributes you set in the OIL file.

A template is a file which is located in the default template directory (set with the environment variable `GOIL_TEMPLATES` or with the `--templates` option on the command line) or in the directory of your project. Goil starts by looking for a template in the directory of your project, then, if the template is not found, in the default templates directory.

Four sets of templates are used:

- code generation templates that are located in the `'code'` subdirectory of the template directory;
- build system templates that are located in the `'build'` subdirectory;
- compiler dependent stuff in the `'compiler'` subdirectory and
- linker script templates in the `'linker'` subdirectory.

Templates are written using a simple language which allow to access the application configuration data and to mix them with text to produce files.

Files are produced by a template program located in the `'root.goilTemplate'` file which is as the root of the template directory. By default the following files are produced:

- `'tpl_app_config.c'` by using the `'tpl_app_config.c.goilTemplate'` file
- `'tpl_app_config.h'` by using the `'tpl_app_config.h.goilTemplate'` file
- `'Makefile'` (if option `-g` or `--generate-makefile` is given) by using the `'Makefile.goilTemplate'` file
- `'script.ld'` (if memory mapping is used and if the default name is not changed) by using the `'script.goilTemplate'` file

- ‘MemMap.h’ (if memory mapping is used) by using the ‘MemMap.h.goilTemplate’ file
- ‘Compiler.h’ (if memory mapping is used) by using the ‘Compiler.h.goilTemplate’ file
- ‘Compiler_Cfg.h’ (if memory mapping is used) by using the ‘Compiler_Cfg.h.goilTemplate’ file

9.1 The configuration data

The configuration data are computed by Goil from the OIL source files, from the options on the command line and from the ‘target.cfg’ file. They are available as a set of predefined boolean, string, integer or list variables. All these variables are in capital letters.



Some configuration data are not listed here because they depend on the target. For instance, the `STACKSIZE` data may be an attribute of each item of a `TASKS` list for ppc target but are missing for the c166 target.

9.1.1 The *PROCESSES*, *TASKS*, *BASICTASKS*, *EXTENDEDTASKS*, *ISRS1* and *ISRS2* lists

These variables are lists where informations about the processes¹ used in the application are stores:

List	Content
<i>PROCESSES</i>	the list of processes. The items are sorted in the following order: extended tasks, then basic tasks, then ISRs category 2.
<i>TASKS</i>	the list of tasks, basic and extended. The items are sorted in the following order: extended tasks, then basic tasks.
<i>BASICTASKS</i>	the list of basic tasks.
<i>EXTENDEDTASKS</i>	the list of extended tasks.
<i>ISRS1</i>	the list of ISR category 1.
<i>ISRS2</i>	the list of ISR category 2.

Each item of these lists has the following attributes:

Item	Type	Content
NAME	string	the name of the process.
PROCESSKIND	string	the kind of process: "Task" or "ISR".
EXTENDEDTASK	boolean	true if the process is an extended task, false otherwise.
NONPREEMPTABLE	boolean	true if the process is a non-preemptable task, false otherwise.
PRIORITY	integer	the priority of the process.
ACTIVATION	integer	the number of activation of a task. 1 for and extended task or an ISR.
AUTOSTART	boolean	true if the process is an autostart task, false otherwise.

¹In Trampoline, a process is a task or an ISR category 2.

Item	Type	Content
USEINTERNALRESOURCE	boolean	true if the process is a task that uses an internal resource, false otherwise.
INTERNALRESOURCE	string	the name of the internal resource if the process is a task that uses an internal resource, empty string otherwise.
RESOURCES	list	The resources used by the process. Each item has the following attribute: NAME
TIMINGPROTECTION	struct	The timing protection attributes. This attribute does not exist if no timing protection is defined for the process. See below for the content of this struct.

The *TIMINGPROTECTION* struct has the following sub-attributes:

Item	Type	Content
EXECUTIONBUDGET	integer	The execution budget of a task. This attribute is not defined for an ISR.
EXECUTIONTIME	integer	The execution time of an ISR. This attribute is not defined for a Task.
TIMEFRAME	integer	The time frame.
MAXOSINTERRUPTLOCKTIME	integer	The maximum locking time of OS interrupts.
MAXALLINTERRUPTLOCKTIME	integer	The maximum locking time of all interrupts.
RESOURCESLOCK	list	The maximum locking time of resources.

Each element of the *RESOURCESLOCK* list has the following attributes:

Item	Type	Content
RESOURCENAME	string	The name of the locked resource.
LOCKTIME	integer	The maximum locking time of the resource.

9.1.2 The *COUNTERS*, *HARDWARECOUNTERS* and *SOFTWARECOUNTERS* lists

These list contains all the informations about the counters used in the application, including the *SystemCounter*.

List	Content
<i>COUNTERS</i>	the list of counters, both hardware and software as long as the <i>SystemCounter</i> .
<i>HARDWARECOUNTERS</i>	the list of hardware counters including the <i>SystemCounter</i> .
<i>SOFTWARECOUNTERS</i>	the list of software counters.

Each item of this list has the following attributes:

Item	Type	Content
NAME	string	the name of the counter.
TYPE	string	the type: "HARDWARE_COUNTER" or "SOFTWARE_COUNTER".
MAXALLOWEDVALUE	integer	the maximum allowed value of the counter.
MINCYCLE	integer	the minimum cycle value of the counter.
TICKPERBASE	integer	the number of ticks needed to increment the counter.

Item	Type	Content
SOURCE	string	the interrupt source name of the counter. This is be used to wrap interrupt vector to a counter incrementation function.

9.1.3 The *EVENTS* list

This list contains the informations about the events of the application. Each item has the following attributes:

Item	Type	Content
NAME	string	the name of the event.
MASK	integer	the mask of the event.

9.1.4 The *ALARMS* list

This list contains the informations about the alarms of the application. Each item has the following attributes:

Item	Type	Content
NAME	string	the name of the alarm.
COUNTER	string	the name of the counter that drives the alarm.
ACTION	string	the action to be done when the alarm expire. It can take the following values: "setEvent", "activateTask" and "callback". The last action is not available in AUTOSAR mode.
TASK	string	the name of the task on which the action is performed. This attribute is defined for "setEvent" and "activateTask" actions only.
EVENT	string	the name of the event to set on the target task. This attribute is defined for "setEvent" action only.
AUTOSTART	boolean	true if the alarm is autostart, false otherwise
ALARMTIME	integer	the alarm time of the alarm. This attribute is set if AUTOSTART is true.
CYCLETIME	integer	the cycle time of the alarm. This attribute is set if AUTOSTART is true.
APPMODE	string	the application mode in which the alarm is autostart. This attribute is set if AUTOSTART is true.

9.1.5 The *REGULARRESOURCES* and *INTERNALRESOURCES* lists

These lists contains the informations about the resources of the application.

List	Content
<i>REGULARRESOURCES</i>	the list of STANDARD and LINKED resources.
<i>INTERNALRESOURCES</i>	the list of INTERNAL resources.

Each item has the following attributes:

Item	Type	Content
NAME	string	the name of the resource.
PRIORITY	integer	the priority of the resource.

Item	Type	Content
TASKUSAGE	list	the list of tasks that use the resource. Each item of this list has an attribute NAME which is the name of the task.
ISRUSAGE	list	the list of ISRs that use the resource. Each item of this list has an attribute NAME which is the name of the ISR.

9.1.6 The *MESSAGES*, *SENDMESSAGES* and *RECEIVEMESSAGES* lists

These lists contain the informations about the messages of the application.

List	Content
<i>MESSAGES</i>	the list of messages, both send and receive message.
<i>SENDMESSAGES</i>	the list of send messages.
<i>RECEIVEMESSAGES</i>	the list of receive messages.

Each item has the following attributes

Item	Type	Content
NAME	string	the name of the message.
MESSAGEPROPERTY	string	the type of the message. It can be "RECEIVE_ZERO_INTERNAL", "RECEIVE_UNQUEUED_INTERNAL", "RECEIVE_QUEUED_INTERNAL", "SEND_STATIC_INTERNAL", "SEND_ZERO_INTERNAL" or "RECEIVE_ZERO_SENDERS".
NEXT	string	the name of the next message in a receive message chain. This attribute is defined for receive messages only.
SOURCE	string	the name of the send message which is connected to the receive message. This attribute is defined for receive messages only.
CTYPE	string	the C language type of the message. This attribute is not defined for "RECEIVE_ZERO_INTERNAL" and "SEND_ZERO_INTERNAL" messages.
INITIALVALUE	string	initial value of the receive message. This attribute is defined for "RECEIVE_UNQUEUED_INTERNAL" and "RECEIVE_ZERO_SENDERS" messages only.
QUEUESIZE	integer	queue size of a receive queued message. This attribute is defined for "RECEIVE_QUEUED_INTERNAL" messages only.
TARGET	string	target message of a send message. This is the first message in a receive message chain. This attribute is defined for "SEND_STATIC_INTERNAL" and "SEND_ZERO_INTERNAL" messages only.
FILTER	string	the kind of filter to apply. This attribute may take the following values: "ALWAYS", "NEVER", "MASKEDNEWEQUALSX", "MASKEDNEWDIFFERSX", "NEWISEQUAL", "NEWISDIFFERENT", "MASKEDNEWEQUALSMASKEDOLD", "MASKEDNEWDIFFERSMASKEDOLD", "NEWISWITHIN", "NEWISOUTSIDE", "NEWISGREATER", "NEWISLESSOREQUAL", "NEWISLESS", "NEWISGREATEROREQUAL" or "ONEEVERYN".
MASK	integer	Mask of the filter when needed. This attribute is defined for "MASKEDNEWEQUALSX", "MASKEDNEWDIFFERSX", "MASKEDNEWEQUALSMASKEDOLD" and "MASKEDNEWDIFFERSMASKEDOLD" filters only.

Item	Type	Content
X	integer	Value of the filter when needed. This attribute is defined for "MASKEDNEWEQUALSMASKEDOLD" and "MASKEDNEWDIFFERSX" filters only.
MIN	integer	Minimum value of the filter when needed. This attribute is defined for "NEWISWITHIN" and "NEWISOUTSIDE" filters only.
MAX	integer	Maximum value of the filter when needed. This attribute is defined for "NEWISWITHIN" and "NEWISOUTSIDE".
PERIOD	integer	Period of the filter. This attribute is defined for "ONEEVERYN" filter only.
OFFSET	integer	Offset of the filter. This attribute is defined for "ONEEVERYN" filter only.
ACTION	string	the action (or notification) to be done when the message is delivered. It can take the following values: "setEvent" or "activateTask".
TASK	string	the name of the task on which the notification is performed. This attribute is defined for "setEvent" and "activateTask" actions only.
EVENT	string	the name of the event to set on the target task. This attribute is defined for "setEvent" notification only.

9.1.7 The *SCHEDULETABLES* list

This list contains the informations about the schedule tables of the application.

Item	Type	Content
NAME	string	the name of the schedule table.
COUNTER	string	the name of the counter which drives the schedule table.
PERIODIC	boolean	true if the schedule table is a periodic one, false otherwise.
SYNCSTRATEGY	string	the synchronization strategy of the schedule table. This attribute may take the following values: "SCHEDTABLE_NO_SYNC", "SCHEDTABLE_IMPLICIT_SYNC" or "SCHEDTABLE_EXPLICIT_SYNC".
PRECISION	integer	the precision of the synchronization. This attribute is define when SYNCSTRATEGY is "SCHEDTABLE_EXPLICIT_SYNC".
STATE	string	the state of the schedule table. This attribute may take the following values: "SCHEDULETABLE_STOPPED", "SCHEDULETABLE_AUTOSTART_SYNCHRON", "SCHEDULETABLE_AUTOSTART_RELATIVE" or "SCHEDULETABLE_AUTOSTART_ABSOLUTE".
DATE	integer	the start date of the schedule table. This attribute has an actual value when STATE is "SCHEDULETABLE_AUTOSTART_RELATIVE" or "SCHEDULETABLE_AUTOSTART_ABSOLUTE", otherwise it is set to 0.
LENGTH	integer	The length of the schedule table.
EXPIRYPOINTS	list	The expiry points of the schedule table. See the following table for items attributes.

Each item of the EXPIRYPOINTS list has the following attributes:

Item	Type	Content
ABSOLUTEOFFSET	integer	the absolute offset of the expiry points.

Item	Type	Content
RELATIVEOFFSET	integer	the relative offset of the expiry points from the previous expiry point.
MAXRETARD	integer	maximum retard to keep the schedule table synchronous.
MAXADVANCE	integer	maximum advance to keep the schedule table synchronous.
ACTIONS	list	the actions to perform on the expiry point. See the following table for items attributes.

Each item of the **ACTIONS** list has the following attributes:

Item	Type	Content
ACTION	string	the action to be done when the alarm expire. It can take the following values: "setEvent", "activateTask", "incrementCounter" and "finalizeScheduleTable".
TASK	string	the name of the task on which the action is performed. This attribute is defined for "setEvent" and "activateTask" actions only.
EVENT	string	the name of the event to set on the target task. This attribute is defined for "setEvent" action only.
TARGETCOUNTER	string	the name of the counter to increment. This attribute is defined for "incrementCounter" action only.

9.1.8 The *OSAPPLICATIONS* list

This list contains the informations about the OS Applications of the application.

Item	Type	Content
NAME	string	the name of the OS Application.
RESTART	string	the name of the restart task. This attribute is not defined if there is no restart task for the OS Application.
PROCESSACCESSVECTOR	string	access right for the processes
PROCESSACCESSITEMS	string	access right for the processes as bytes in a table
PROCESSACCESSNUM	integer	number of elements in the previous table
ALARMACCESSVECTOR	string	access right for the alarms
ALARMACCESSITEMS	string	access right for the alarms as bytes in a table
ALARMACCESSNUM	integer	number of elements in the previous table
RESOURCEACCESSVECTOR	string	access right for the resources
RESOURCEACCESSITEMS	string	access right for the resources as bytes in a table
RESOURCEACCESSNUM	integer	number of elements in the previous table
SCHEDULETABLEACCESSVECTOR	string	access right for the schedule tables
SCHEDULETABLEACCESSITEMS	string	access right for the schedule tables as bytes in a table
SCHEDULETABLEACCESSNUM	integer	number of elements in the previous table
COUNTERACCESSVECTOR	string	access right for the software counters
COUNTERACCESSITEMS	string	access right for the software counters as bytes in a table
COUNTERACCESSNUM	integer	number of elements in the previous table
PROCESSES	list	list of the processes that belong to the OS Application. Each item has an attribute NAME which is the name of the process.

Item	Type	Content
HASSTARTUPHOOK	boolean	true if the OS Application has a startup hook.
HASSHUTDOWNHOOK	boolean	true if the OS Application has a shutdown hook.
TASKS	list	list of the tasks that belong to the OS Application. Each item has an attribute NAME which is the name of the task.
ISRS	list	list of the ISRs that belong to the OS Application. Each item has an attribute NAME which is the name of the ISR.
ALARMS	list	list of the alarms that belong to the OS Application. Each item has an attribute NAME which is the name of the alarm.
RESOURCES	list	list of the resources that belong to the OS Application. Each item has an attribute NAME which is the name of the resource.
REGULARRESOURCES	list	list of the standard or linked resources that belong to the OS Application. Each item has an attribute NAME which is the name of the resource.
INTERNALRESOURCES	list	list of the internal resources that belong to the OS Application. Each item has an attribute NAME which is the name of the resource.
SCHEDULETABLES	list	list of the schedule tables that belong to the OS Application. Each item has an attribute NAME which is the name of the schedule table.
COUNTERS	list	list of the counters that belong to the OS Application. Each item has an attribute NAME which is the name of the counter.
MESSAGES	list	list of the messages that belong to the OS Application. Each item has an attribute NAME which is the name of the messages.

9.1.9 The *TRUSTEDFUNCTIONS* list

This list contains the informations about the trusted functions of the application. Each item contains one attribute only.

Item	Type	Content
NAME	string	the name of the trusted function.

9.1.10 The *READYLIST* list

This list contains the informations about the ready list. Items are sorted by priority from 0 to the maximum computed priority. The only attribute of each item is the size of the queue.

Item	Type	Content
SIZE	integer	the size of the queue for the corresponding priority.

9.1.11 The *SOURCEFILES*, *CFLAGS*, *CPPFLAGS*, *ASFLAGS*, *LDFLAGS* and *TRAMPOLINESOURCEFILES* lists

The *SOURCEFILES* list contains the source files as found in attributes `APP_SRC` of the OS object in the OIL file. Each item in the list has one attribute.

Item	Type	Content
FILE	string	the source file name.

The *CFLAGS* list contains the flags for the C compiler as found in attributes `CFLAGS` of the OS object in the OIL file. Each item in the list has one attribute.

Item	Type	Content
CFLAG	string	the C compiler flag.

The *CPPFLAGS* list contains the flags for the C++ compiler as found in attributes `CPPFLAGS` of the OS object in the OIL file. Each item in the list has one attribute.

Item	Type	Content
CPPFLAG	string	the C++ compiler flag.

The *ASFLAGS* list contains the flags for the assembler as found in attributes `ASFLAGS` of the OS object in the OIL file. Each item in the list has one attribute.

Item	Type	Content
ASFLAG	string	the assembler flag.

The *LDFLAGS* list contains the flags for the linker as found in attributes `LDFLAGS` of the OS object in the OIL file. Each item in the list has one attribute.

Item	Type	Content
LDFLAG	string	the linker flag.

The *TRAMPOLINESOURCEFILES* list contains the trampoline source files used by the application. Each item in the list has two attributes.

Item	Type	Content
DIRECTORY	string	the directory of the source file relative to the Trampoline root directory ('os', 'com' or 'autosar').
FILE	string	the source file name.

9.1.12 The *INTERRUPTSOURCES* list

This list is extracted from the '`target.cfg`' file. Each item has the following attributes:

Item	Type	Content
NAME	string	the name of the interrupt source. This is one of the name used in the OIL file as value for the SOURCE attribute.
NUMBER	string	the id of the interrupt source.

9.1.13 Scalar data

The following scalar data are defined:

Data	Type	Content
<i>APPNAME</i>	string	name of executable as given in the APP_NAME attribute in the OS object
<i>ARCH</i>	string	name of the architecture. This is the first item in the target.
<i>ASSEMBLER</i>	string	name of the assembler used. This is the ASSEMBLER attribute in the MEMMAP attribute of the OS object. It is used for assembler dependent templates.
<i>ASSEMBLEREXE</i>	string	name of the assembler executable used. This is the ASSEMBLER attribute in the OS object. It is set to <i>as</i> by default. It is used for build dependent templates.
<i>AUTOSAR</i>	boolean	true if Trampoline is compiled with the Autosar extension.
<i>BOARD</i>	string	name of the board. This is the third item (if any) in the target.
<i>CHIP</i>	string	name of the chip. This is the second item (if any) in the target.
<i>COMPILER</i>	string	name of the compiler used. This is the COMPILER attribute in the MEMMAP attribute of the OS object. It is used for compiler dependent templates.
<i>COMPILEREXE</i>	string	name of the compiler executable used. This is the COMPILER attribute in the OS object. It is set to <i>gcc</i> by default. It is used for build dependent templates. Do not confuse with the COMPILER data.
<i>CPUNAME</i>	string	name given to the OIL CPU object
<i>EXTENDED</i>	boolean	true if Trampoline is compiled in extended error handling mode.
<i>FILENAME</i>	string	the name of the file which will be written as the result of the computation of the current template.
<i>FILEPATH</i>	string	the full absolute path of the file which will be written as the result of the computation of the current template.
<i>ITSOURCESLENGTH</i>	integer	number of interrupt sources as defined in the ' target.cfg ' file.
<i>LINKER</i>	string	name of the linker used. This is the LINKER attribute in the MEMMAP attribute of the OS object. It is used for linker dependent templates.
<i>LINKEREXE</i>	string	name of the linker executable used. This is the LINKER attribute in the OS object. It is set to <i>gcc</i> by default. It is used for build dependent templates. Do not confuse with the LINKER data.
<i>LINKSCRIPT</i>	string	name of the link script file as given in the MEMMAP attribute of the OS object.

Data	Type	Content
<i>MAXTASKPRIORITY</i>	integer	the highest computed priority among the tasks.
<i>NATIVEFILEPATH</i>	string	the full absolute path of the file which will be written as the result of the computation of the current template in native OS format.
<i>OILFILENAME</i>	string	name of the root OIL source file
<i>PROJECT</i>	string	name of the project. The name of the project is the <code>-p</code> (or <code>--project</code>) value if it is set or the name of the oil file without the extension.
<i>SCALABILITYCLASS</i>	integer	the Autosar scalability class used by the application. If Autosar is not enabled, <i>SCALABILITYCLASS</i> is set to 0.
<i>TARGET</i>	string	name of the target. This is the <code>-t</code> (or <code>--target</code>) option value of goil.
<i>TEMPLATEPATH</i>	string	path to the template root directory. This is the <code>--templates</code> option value of goil or the value of the <code>GOIL-TEMPLATES</code> environment variable.
<i>TIMESTAMP</i>	string	current date
<i>TRAMPOLINEPATH</i>	string	path to the trampoline root directory. This is the <code>TRAMPOLINE_BASE_PATH</code> attribute of the OS object. It defaults to “.”.
<i>USEBUILDFILE</i>	boolean	true if a build file is used for the project ie option <code>-g</code> or <code>--generate-makefile</code> is given.
<i>USECOM</i>	boolean	true if the application uses OSEK COM.
<i>USECOMPILERSETTINGS</i>	boolean	true if memory mapping is enabled (Goil generates the ‘Compiler.h’ and ‘Compiler_Cfg.h’ files and Trampoline includes them).
<i>USEERRORHOOK</i>	boolean	true if Trampoline uses the Error Hook.
<i>USEGETSERVICEID</i>	boolean	true if Trampoline uses the service ids access macros.
<i>USEINTERRUPTTABLE</i>	boolean	true if the wrapping of interrupt vector to glue functions used to increment a counter or to activate an ISR2 (for instance) should be generated. The actual code generation is up to the port.
<i>USELOGFILE</i>	boolean	true if goil generates a log file, ie option <code>-l</code> or <code>--logfile</code> is given.
<i>USEMEMORYMAPPING</i>	boolean	true if memory mapping is enabled (Goil generates the ‘MemMap.h’ file and Trampoline includes it).
<i>USEMEMORYPROTECTION</i>	boolean	true if Trampoline uses the Memory Protection.
<i>USEOSAPPLICATION</i>	boolean	true if Trampoline uses OS Applications.
<i>USEPARAMETERACCESS</i>	boolean	true if Trampoline uses the parameters access macros.
<i>USEPOSTTASKHOOK</i>	boolean	true if Trampoline uses the Post-Task Hook.
<i>USEPRETASKHOOK</i>	boolean	true if Trampoline uses the Pre-Task Hook.
<i>USEPROTECTIONHOOK</i>	boolean	true if Trampoline uses the Protection Hook.
<i>USERESSCHEDULER</i>	boolean	true if Trampoline uses the <code>RES_SCHEDULER</code> resource.
<i>USESHUTDOWNHOOK</i>	boolean	true if Trampoline uses the Shutdown Hook.
<i>USESTACKMONITORING</i>	boolean	true if Trampoline uses the Stack Monitoring.
<i>USESTARTUPHOOK</i>	boolean	true if Trampoline uses the Startup Hook.
<i>USESISTEMCALL</i>	boolean	true if services are called using a System Call (i.e. a software interrupt).
<i>USETIMINGPROTECTION</i>	boolean	true if Trampoline uses Timing Protection.

Data	Type	Content
<i>USETRACE</i>	boolean	true if tracing is enabled.

9.2 The Goil template language (or GTL)

A template is a text file with file extension `‘.goilTemplate’`. This kind of file mixes literal text with an embedded program. Some instructions (see section 9.5.6) in the embedded program outputs text as a result of the program execution and this text is put in place of the instructions. The resulting file is then stored.

The template interpreter starts in literal text mode. Switching from literal text mode to program mode and back to text mode is done when a `‘%’` is encountered. A literal `‘%’` and a literal `‘\’` may be used by escaping them with a `‘\’`.

9.3 GTL types

GTL supports 5 types: **string**, **integer**, **boolean**, **list** and **struct**. The 4 first types have readers to get informations about a variable. A reader is invoked with the following syntax:

```
[expression reader]
```

A struct is an aggregate of data. The `‘::’` allows to get a member of the struct. For instance one of the member of *TIMINGPROTECTION* is *TIMEFRAME* so to get *TIMEFRAME*, the following syntax is used:

```
TIMINGPROTECTION::TIMEFRAME
```

9.3.1 string readers

The following readers are available for string variables:

Item	Type	Meaning
<code>HTMLRepresentation</code>	string	this reader returns a representation of the string suitable for an HTML encoded representation. <code>‘&’</code> is encoded by <code>&amp;</code> ; <code>‘”’</code> by <code>&quot;</code> ; <code>‘<’</code> by <code>&lt;</code> ; and <code>‘>’</code> by <code>&gt;</code> .
<code>identifierRepresentation</code>	string	this reader returns an unique representation of the string conforming to a C identifier. Any Unicode character that is not a latin letter is transformed into its hexadecimal code point value, enclosed by <code>‘_’</code> characters. This representation is unique: two different strings are transformed into different C identifiers. For example: <code>value3</code> is transformed to <code>value_33_</code> ; <code>+=</code> is transformed to <code>_2B__3D_</code> ; <code>An_Identifier</code> is transformed to <code>An_5F_Identifier</code> .

Item	Type	Meaning
<code>lowercaseString</code>	string	this reader returns lowercased representation of the string.
<code>length</code>	integer	this reader returns the number of characters in the string
<code>stringByCapitalizingFirstCharacter</code>	string	if the string is empty, this reader returns the empty string; otherwise, it returns the string, the first character being replaced with the corresponding upper case character.
<code>uppercaseString</code>	string	this reader returns uppercased representation of the receiver

9.3.2 boolean readers

The following readers are available for boolean variables:

Item	Type	Meaning
<code>trueOrFalse</code>	string	this reader returns "true" or "false" according to the boolean value
<code>yesOrNo</code>	string	this reader returns "yes" or "no" according to the boolean value
<code>unsigned</code>	integer	this reader returns 0 or 1 according to the boolean value

9.3.3 integer readers

The following readers are available for integer variables:

Item	Type	Meaning
<code>string</code>	string	This reader returns the integer value as a character string.
<code>hexString</code>	string	this reader returns an hexadecimal string representation of the integer value.

9.3.4 list readers

The following reader is available for list variables:

Item	Type	Meaning
<code>length</code>	integer	this reader returns the number of objects currently in the list.

9.4 GTL operators

9.4.1 Unary operators

Operator	Operand Type	Result Type	Meaning
<code>+</code>	integer	integer	no operation.
<code>~</code>	integer	integer	bitwise not.
<code>not</code>	boolean	boolean	boolean not.

Operator	Operand Type	Result Type	Meaning
<code>exists</code>	<i>any variable</i>	boolean	true if the variable is defined, false otherwise. But see below



A second form of `exists` is:

```
exists var default (expression)
```

var and *expression* should have the same type. If *var* exists, the returned value is the content of *var*. If it does not exist, *expression* is returned.

9.4.2 Binary operators

Operator	Operands Type	Result Type	Meaning
<code>+</code>	integer	integer	add.
<code>-</code>	integer	integer	subtract.
<code>*</code>	integer	integer	multiply.
<code>/</code>	integer	integer	divide.
<code>&</code>	integer	integer	bitwise and.
<code>&</code>	boolean	boolean	boolean and.
<code> </code>	integer	integer	bitwise or.
<code> </code>	boolean	boolean	boolean or.
<code>^</code>	integer	integer	bitwise xor.
<code>^</code>	boolean	boolean	boolean xor.
<code>.</code>	string	string	string concatenation.
<code><<</code>	integer	integer	shift left.
<code>>></code>	integer	integer	shift right.
<code>!=</code>	<i>any</i>	boolean	comparison (different).
<code>==</code>	<i>any</i>	boolean	comparison (equal).
<code><</code>	integer <i>or</i> boolean	boolean	comparison (lower than).
<code><=</code>	integer <i>or</i> boolean	boolean	comparison (lower or equal).
<code>></code>	integer <i>or</i> boolean	boolean	comparison (greater).
<code>>=</code>	integer <i>or</i> boolean	boolean	comparison (greater or equal).

9.4.3 Constants

Constant	Type	Meaning
<code>emptyList</code>	list	this constant is an empty list
<code>true</code>	boolean	true boolean
<code>false</code>	boolean	false boolean
<code>yes</code>	boolean	true boolean
<code>no</code>	boolean	false boolean

9.5 GTL instructions

9.5.1 The *let* instruction

Data assignment instruction. The general form is:

```
let var := expression
```

A second form allows to add a string to a list (only, this should be extended in the future). The string is added with the *NAME* attribute.

```
let var += expression
```

var is a list and *expression* is a string.

The scope of a variable depends on the location where the variable is assigned the first time. For instance, in the following code:

```
let a := 1
foreach TASKS do
  let b := INDEX
  let a := INDEX
end foreach
!a !b
```

Because *a* is assigned outside the **foreach** loop, it contains the value of the last *INDEX* after the **foreach**. Because *b* is assigned inside the **foreach** loop, it does not exist after the loop anymore and *!b* will trigger an error.

9.5.2 The *if* instruction

Conditional execution. The forms are:

```
if expression then ... end if
if expression then ... else ... end if
if expression then ... elsif expression then ... end if
if expression then ... elsif expression then ... else ... end if
```

The *expression* must be boolean. In the following example, the blue text (within the *%*) is produced only if the *USECOM* boolean variable is true:

```
if USECOM then %
#include "tpl_com.h" %
end if
```

9.5.3 The *foreach* instruction

This instruction iterates on the elements of a list. Each element may have many attributes that are available as variables within the **do** section of the **foreach** loop. The simplest form is the following one

```
foreach expression do ... end foreach
```

In the following example, for each element in the *ALARMS* list, the text between the **do** and the **end foreach** is produced with the *NAME* attribute of the current element of the *ALARMS* list inserted

at the specified location. *INDEX* is not an attribute of the current element. It is generated for each element and ranges from 0 to the number of elements in the list minus 1.

```
foreach ALARMS do
%
/* Alarm % !NAME % identifier */
#define % !NAME %_id % !INDEX %
CONST(AlarmType, AUTOMATIC) % !NAME % = % !NAME %_id;
%
end foreach
```

A more general form of the `foreach` instruction is:

```
foreach expression prefixedby string
  before ...
  do ...
  between ...
  after ...
end foreach
```

`prefixedby` is optional and allows to prefix the attribute names by *string*. If the list is not empty, the `before` section are executed once before the first execution of the `do` section. The `between` section is executed between the execution of the `do` section. If the list is not empty, the `after` section is executed once after the last execution of the `do` section.

In the following example, a table of pointers to alarm descriptors is generated:

```
foreach ALARMS
  before %
  tpl_time_obj *tpl_alarm_table[ALARM_COUNT] = {
%
  do % &% !NAME %_alarm_desc%
  between %,
%
  after %
};
%
end foreach
```

9.5.4 The *for* instruction

The `for` instruction iterates along a literal list of elements.

```
for var in expression, ... , expression do
...
end for
```

At each iteration, *var* gets the value of the current *expression*. As in the `foreach` instruction, *INDEX* is generated and ranges from 0 to the number of elements in the list minus 1.

9.5.5 The *loop* instruction

The `loop` instruction is the classical integer loop. Its simplest form is:

```
loop var from expression to expression do
  ...
end loop
```

Like in the foreach instruction, **before**, **between** and **after** sections may be used:

```
loop var from expression to expression
  before ...
  do ...
  between ...
  after ...
end loop
```

9.5.6 The **!** instruction

! emits an expression. The form is:

```
! expression
```

9.5.7 The **?** instruction

? stores in a variable a number of spaces equal to the current column in the output. The form is:

```
? var
```

9.5.8 The *template* instruction

The **template** instruction includes the output of another template in the output of the current template. Its simplest form is the following one:

```
template template_file_name
```

If the file *template_file_name.goil*Template does not exist, an error occurs. To include the output of a template without generating an error, use the following form:

```
template if exists template_file_name
```

A third form allows to execute instructions when the included template file is not found:

```
template if exists template_file_name or ... end template
```

At last, it is possible to search templates in a hierarchy (code, linker, compiler, build) different from the current one. For instance to include a template located in the linker hierarchy, use one of the following forms:

```
template template_file_name in hierarchy
template if exists template_file_name in hierarchy
template if exists template_file_name in hierarchy or ... end template
```

In all cases, the included template inherits from the current variables table but works on its own local copy.

9.5.9 The *write* instruction

The write instruction defines a block where the template processing output is captured to be written to a file. The general form is:

```
write to expression :
...
end write
```

Where *expression* is a string expression.

In the following example, the result of the ‘script’ template is written to the link script file.

```
if exists LINKER then
  write to PROJECT."/".LINKSCRIPT:
    template script in linker
  end write
end if
```

9.5.10 The *error* and *warning* instructions

It can be useful to generate an error or a warning if a data is not defined or if it looks strange. For instance if a target needs a STACKSIZE for a task or if the STACKSIZE is too large for a 16bit target. **error** and **warning** have 2 forms:

```
error var expression
warning var expression
```

and

```
error here expression
warning here expression
```

expression must be of type string. In the first form, *var* is a configuration data. The file location of this configuration may be a location in the OIL file or in the template file if the variable was assigned in the template. In the second form, **here** means the current location in the template file.

In the following example an error is generated for each task with not STACKSIZE attribute in the OIL file:

```
foreach TASKS do
  if not exists STACKSIZE then
    error NAME "STACKSIZE of Task " . NAME . " is not defined"
  end if
end foreach
```

In this second example, a warning is generated if a template is not found:

```
template if exists interrupt_wrapping or
  warning here "interrupt_wrapping.goilTemplate not found"
end template
```

9.6 Examples

Here are examples of code generation using GTL.

9.6.1 Computing the list of process ids

```
foreach PROCESSES do
  if PROCESSKIND == "Task" then
%
/* Task % !NAME % identifier */
#define % !NAME %_id % !INDEX %
CONST(TaskType, AUTOMATIC) % !NAME % = % !NAME %_id;
%
  else
%
/* ISR % !NAME % identifier */
#define % !NAME %_id % !INDEX
  if AUTOSAR then
    #
    # ISR ids constants are only available for AUTOSAR
    #
%
CONST(ISRType, AUTOMATIC) % !NAME % = % !NAME %_id;
%
  end if
end if
end foreach
```

9.6.2 Computing an interrupt table

```
if USEINTERRUPTTABLE then
  loop ENTRY from 0 to ITSOURCESLENGTH - 1
  before
%
#define OS_START_SEC_CONST_UNSPECIFIED
#include "tpl_memmap.h"
CONST(tpl_it_vector_entry, OS_CONST)
tpl_it_table[% !ITSOURCESLENGTH %] = {
%
  do
    let entryFound := false
    foreach INTERRUPTSOURCES prefixedby interrupt_ do
      if ENTRY == interrupt_NUMBER then
        # check first for counters
        foreach HARDWARECOUNTERS prefixedby counter_ do
          if counter_SOURCE == interrupt_NAME & not entryFound then
            % { tpl_tick_% !interrupt_NAME %, (void *)NULL }%
            let entryFound := true
          end if
        end foreach
      end if
    end foreach
    if not entryFound then
      foreach ISRS2 prefixedby isr2_ do
        if isr2_SOURCE == interrupt_NAME & not entryFound then
          % { tpl_central_interrupt_handler_2, (void*)%
            !([TASKS length] + INDEX) % }%
          let entryFound := true
        end if
      end if
    end if
  end if
```

```

        end foreach
      end if
    end if
  end foreach
  if not entryFound then
    % { tpl_null_it, (void *)NULL }%
  end if
  between %,
%
  after
%
};
#define OS_STOP_SEC_CONST_UNSPECIFIED
#include "tpl_memmap.h"
%
end loop
end if

```

9.6.3 Generation of all the files

This is the default 'root.goilTemplate' file

```

write to PROJECT."/tpl_app_config.c":
  template tpl_app_config_c in code
end write

write to PROJECT."/tpl_app_config.h":
  template tpl_app_config_h in code
end write

write to PROJECT."/tpl_app_define.h":
  template tpl_app_define_h in code
end write

if exists COMPILER then
  write to PROJECT."/MemMap.h":
    template MemMap_h in compiler
  end write
  write to PROJECT."/Compiler.h":
    template Compiler_h in compiler
  end write
  write to PROJECT."/Compiler_Cfg.h":
    template Compiler_Cfg_h in compiler
  end write
end if

if exists LINKER then
  write to PROJECT."/".LINKSCRIPT:
    template script in linker
  end write
end if

```

BIBLIOGRAPHY

- [1] *Programming Environments Manual for 32-Bit Implementations of the PowerPCTM Architecture*, chapter 8, pages 8–157. Freescale semiconductor, rev. 3 edition, September 2005.
- [2] AUTOSAR. Specification of compiler abstraction. Technical report, AUTOSAR GbR, August 2008. http://autosar.org/download/R3.1/AUTOSAR_SWS_CompilerAbstraction.pdf.
- [3] AUTOSAR. Specification of memory mapping. Technical report, AUTOSAR GbR, June 2008. http://autosar.org/download/R3.1/AUTOSAR_SWS_MemoryMapping.pdf.
- [4] Microcontroller Applications IBM Microelectronics. Developing powerpc embedded application binary interface (eabi) compliant programs. Technical report, IBM, Research Triangle Park, NC, September 1998.
- [5] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Comput.*, 39(9):1175–1185, 1990.