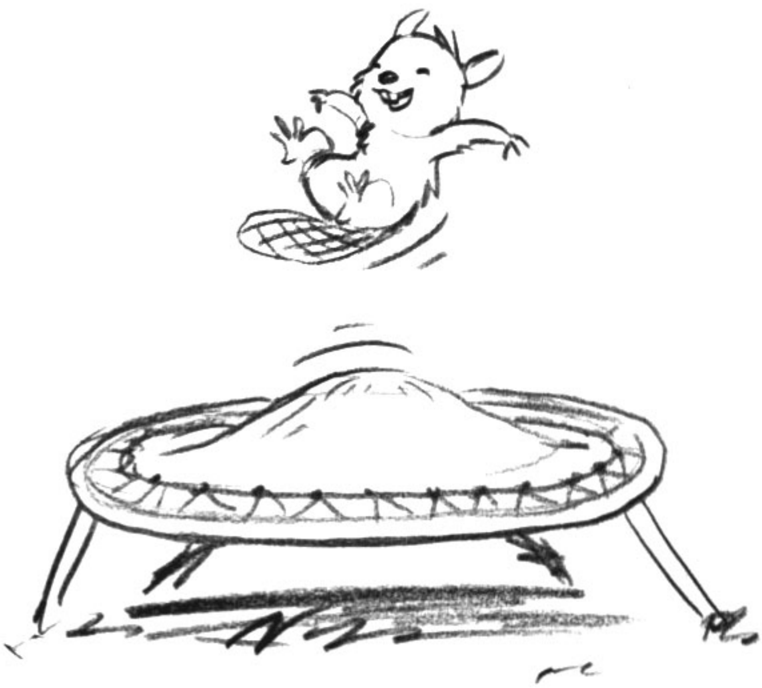
The Trampoline Handbook

release 2.0

Jean-Luc Béchennec
Mikaël Briday
Sébastien Faucou
Pierre Molinaro
Florent Pavin



CONTENTS

Part I

The Real-Time Operating System

CHAPTER

ONE

GETTING STARTED

This chapter shows how to compile and run your first application. We are going to use the Posix port of Trampoline, Trampoline/Posix, that runs over a Linux or Mac OS X operating system. So it is assumed you are using a Linux or Mac OS X computer since Trampoline/Posix does not run over Windows[1]. It is also assumed that you have a basic knowledge of using the command line and the Unix shell.

OSEK/VDX and Autosar os are static operating systems. That means the objects of the application, tasks, events, resources, ..., cannot be created or deleted during the execution of the application. All objects are statically defined and instead of forcing the user to instantiate the OS objects related to the application in C language, a work that can be error prone, a specific language is used, OIL or XML[2]. A compiler, `goil`, is used to translate the description in the equivalent C structures. `goil` performs verifications too.

1.1 Setting up the environment

Before compiling and running the first application, a few tools are required. The first tool needed is a development chain, compiler and linker, for the target platform. In our case, the native development chain, `gcc` under Linux, `clang` under Mac OS X will be used. The two other tools are respectively `goil` and `viper` that we will compile. In the following, all paths are relative to the Trampoline root directory. When setting up path environment variables, **complete the relative path with the installation path of Trampoline**.

[1] An API working like Unix signals is missing on Windows. However it is possible to run Linux in a virtual machine like VirtualBox.

[2] for Autosar

1.1.1 Compiling goil

goil is located in the 'goil' subdirectory. To compile goil, go in the directory corresponding to your operating system, 'goil/makefile-macosx' for Mac OS X or 'goil/makefile-unix' for Linux. Then type `./build.py release`. If everything went well, a goil executable is generated. You can test it by typing `./goil --version`. At the time of writing, the command should output:

```
alflolol:makefile-macosx jlb$ ./goil --version
goil : 3.1.11, build with GALGAS 3.3.11
No warning, no error.
```

You can install goil in '/usr/local/bin' by typing `sudo ./build.py install-release` or you can add to your PATH environment variable the location where goil has been compiled.

In addition you may want to set up the GOIL_TEMPLATES environment variable in your '.profile' or '.bashrc' so that you don't always have to set the --templates= option when calling goil. This variable stores the path to the templates directory used by goil and shall be 'goil/templates'.

1.1.2 Compiling viper

Under Posix, Trampoline requires a runtime support that mimics the minimum behavior of a hardware, mainly timers. viper is a separate application used by Trampoline for this purpose. Go in the 'viper' directory. Type `make` to compile viper. You must also set the environment variable VIPER_PATH to contain the path 'viper'.

1.2 Playing with the one_task application

Go into the 'examples/posix/one_task' directory. In this directory, two files are available: 'one_task.oil' and 'one_task.c'. Start by opening 'one_task.oil'. The content of this file is reproduced below.

```
OIL_VERSION = "2.5";

CPU only_one_task {
  OS config {
    STATUS = EXTENDED;
    BUILD = TRUE {
      APP_SRC = "one_task.c";
      TRAMPOLINE_BASE_PATH = "../../..";
      LDFLAGS="-lrt -lpthread";
      APP_NAME = "one_task_exe";
      LINKER = "gcc";
      SYSTEM = PYTHON;
    };
  };

  APPMODE stdAppmode {};

  TASK my_only_task {
```

```
19      PRIORITY = 1;
20      AUTOSTART = TRUE { APPMODE = stdAppmode; };
21      ACTIVATION = 1;
22      SCHEDULE = FULL;
23    };
24  };
```

`OIL_VERSION = "2.5";` at line 1 specifies which kind of application we are designing. Here it is an OSEK application. For an AUTOSAR application, `OIL_VERSION = "4.0";` would be used.

OIL files consist of two sections, an **IMPLEMENTATION** section that is not used here and a **CPU** section that appears in the line 3. The objects describing the application are located inside the **CPU** section.

The first is the **OS** object at the line 4. This object is used to configure the operating system and, in the case of Trampoline, to specify how to compile it. The first attribute, **STATUS**, indicates the fineness of verification of error conditions by the operating system services. Two values are possible: **STANDARD** and **EXTENDED**. Here, **EXTENDED** is used.

The **BUILD** attribute at line 6 is used to generate a build script. It contains several sub-attributes:

- **APP_SRC** gives the C source code file of your application. If the application is split into several C files, use has many **APP_SRC** as needed.
- **TRAMPOLINE_BASE_PATH** gives the path to the Trampoline root directory.
- **LDFLAGS** is additional flags to pass to the linker. Here we add the *rt* and *pthread* libraries that are needed for multitasking and communication with `viper`.
- **APP_NAME** is the name of the resulting binary file that is directly executable for the Posix target.
- **LINKER** specifies which command is used to invoke the linker.
- **SYSTEM** specifies which build system is used. Here Python build scripts.

The second is the object **APPMODE** at line 16

CHAPTER

TWO

OPERATING SYSTEM EXECUTION

This chapter presents how to start and shutdown the operating system as well as the configuration options and the Application Modes. Application Modes are used to start the operating system in different configurations. Usually, the configuration is read from hardware switches. The current Application Mode is passed to the `StartOS` service and cannot be changed once the operating system is started.

2.1 Configuration Options

2.2 System Services

2.2.1 StartOS

`StartOS` starts the OS in the `AppModeID` Application Mode. First the OS does some initializations, then the Startup Hook, if configured, is called. At last the scheduling is started and the highest priority task runs.

When called from outside a task or an ISR, typically from the `main()`, `StartOS` does not returns. When called from a task or an ISR, a case which is forbidden, `StartOS` returns and the Error Hook (if configured) is called.

If `AppModeID` does not correspond to any Application Mode, no error occurs but none of the **`AUTOSTART`** objects is started.

Prototype of StartOS:

```
void StartOS(AppModeType AppModeID);
```

Arguments of StartOS:

`AppModeID` The Application Mode.

2.2.2 ShutdownOS

`ShutdownOS` shuts down the OS and notify the `Error` error code. If it is configured, the Shutdown Hook is called with `Error` as argument. The behavior may depends on the target platform. On embedded platforms interrupts are disabled and an infinite loop or a `halt` is executed. On POSIX the application exits.

Prototype of ShutdownOS:

```
void ShutdownOS(StatusType Error);
```

Arguments of ShutdownOS:

`Error` The error that occurred.

2.3 Application Modes Declarations

Application Mode are used to specify which **`AUTOSTART`** objects (tasks, alarms or schedule tables) are started when `StartOS` is called. Application Modes are declared in OIL using the **`APPMODE`** object. `goil` accepts the `DEFAULT` boolean attribute. When **`TRUE`**, this attributes specifies the default Application Mode. `DEFAULT` is implicitly **`FALSE`**.

When only one Application Mode is defined, the constant `OSDEFAULTAPPMODE` is set to this Application Mode. When more than one Application Mode are defined, one and only one of the Application Modes `DEFAULT` attribute must be set to **`TRUE`** and the constant `OSDEFAULTAPPMODE` is set to this one.

At most 32 application modes may be declared in the current implementation. We believe it is far enough.

In the following example, 2 Application Modes are declared:

```
APPMODE normal { DEFAULT = TRUE; };

APPMODE diag { };
```

Let's consider 2 tasks and one alarm. The first task, *command*, is **`AUTOSTART`** in any case, the second one, *logging* is not **`AUTOSTART`** and the alarm, *trigger_logging*, is **`AUTOSTART`** in Application Mode `diag` only. The goal is to have a periodic task doing some logging when the OS is started in Application Mode `diag`:

```
TASK command {
  AUTOSTART = TRUE {
    APPMODE = normal;
    APPMODE = diag;
  };
  ...
};
```

```
TASK logging {
  AUTOSTART = FALSE;
  ...
};

ALARM trigger_logging {
  AUTOSTART = TRUE {
    APPMODE = diag;
    ALARMTIME = 10;
    CYCLETIME = 10;
  };
  ACTION = ACTIVATETASK {
    TASK = logging;
  };
  ...
};
```

If `StartOS` is called with argument `normal` or `OSDEFAULTAPPMODE`, the alarm *trigger_logging* is not started by `StartOS` and task *logging* does not run. If `StartOS` is called with argument `diag`, the alarm is started and task *logging* runs. In both cases task *command* is started.

2.4 Application Modes Services

2.4.1 DeclareApplicationMode

On the C side, each declared Application Mode is available as a constant of type `AppModeType`. However, before using one of the constants, you have to put it in the current scope with the DeclareApplicationMode service [1] as follow:

```
DeclareApplicationMode(normal);
DeclareApplicationMode(diag);
```

An exception is the constant `OSDEFAULTAPPMODE` which is in the scope as long as file '`tpl_os.h`' is included.

`DeclareApplicationMode` is a C macro

Prototype of DeclareApplicationMode:

`DeclareApplicationMode(AppModeType AppModeID);`

Arguments of DeclareApplicationMode:

`AppModeID` The Application Mode.

2.4.2 GetActiveApplicationMode

`GetActiveApplicationMode` returns the Application Mode that was used to start the OS.

[1] This macro is not part of [6] but has been added for convenience purpose

```
AppModeType currentAppMode;
currentAppMode = GetActiveApplicationMode();
```

If `GetActiveApplicationMode` is called before the OS is started, `OSNOAPPMODE` is returned.

Prototype of GetActiveApplicationMode:

`AppModeType GetActiveApplicationMode(void);`

2.5 Implementation

At system generation time, an identifier `AppModeID` of type `AppModeType` is attributed to each Application Mode. Identifiers range from 0 to $number\ of\ application\ modes - 1$ and are attributed by `goil` in their order of appearance in the OIL file.

For each `AppModeID`, `goil` computes a mask: `AppModeMask = 1 << AppModeID`. For each task, alarm and schedule table, a table indexed by the object id is computed by `goil`. Each element of these tables is the bitwise or of the `AppModeMask` in which the object is **`AUTOSTART`**. If there is no task, alarm or schedule table defined, the corresponding table is not generated.

`StartOS` iterates over the tasks, alarms and schedule tables Application Mode mask tables. It does a bitwise and with the mask stored in the table and the mask computed from the Application Mode. If the result is not 0 then the corresponding object is **`AUTOSTART`** in this Application Mode and is started.

Using the example of section 2.3 we have

```
CONST(tpl_application_mode, OS_CONST) diag = 0; /* mask = 1 */
CONST(tpl_application_mode, OS_CONST) normal = 1; /* mask = 2 */
```

`AppModeType` is an alias of `tpl_application_mode`.

```
CONST(tpl_appmode_mask, OS_CONST) tpl_task_app_mode[TASK_COUNT] = {
  3 /* task command : normal | diag */ ,
  0 /* task logging :  */
};

CONST(tpl_appmode_mask, OS_CONST) tpl_alarm_app_mode[ALARM_COUNT] = {
  1 /* alarm trigger_logging : diag */
};
```

The `tpl_appmode_mask` type is computed according to the number of Application Modes.

Table 2.1: *Size of `tpl_appmode_mask` type.*

Number of Application Modes	`tpl_appmode_mask` type
$[1, 8]$	u8
$[9, 16]$	u16
$[17, 32]$	u32

CHAPTER

THREE

TASKS

A Task is an execution framework for the functions of the application[1]. A task is a kind of process. Tasks are executed concurrently and asynchronously, see 4.2. 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.

3.1 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 4.1 shows the states of a basic task. An extended task has an additional `WAITING` state. Figure 4.2 shows the states of an extended task. See section 4.5.3 for additional informations about the states of a task.

A task goes from one state to the other according to various conditions as shown in table 4.1.

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.

3.2 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

[1]The 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.

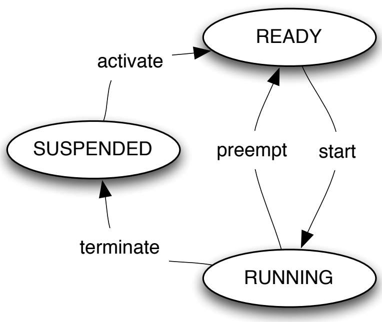


Figure 3.1: *States of a `BASIC` task.*

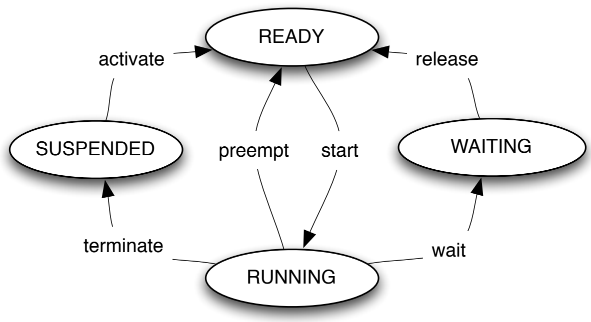


Figure 3.2: *States of an `EXTENDED` task.*

Table 3.1: *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 `ActivateTask` or `ChainTask`, 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 `TerminateTask` or `ChainTask` 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 `WaitEvent`.
release	`WAITING`	`READY`	the task is set to the `READY` state when it gets one of the events it is waiting for.

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 5.

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 ↑ indicates a task activation and a down arrow ↓ a task termination.

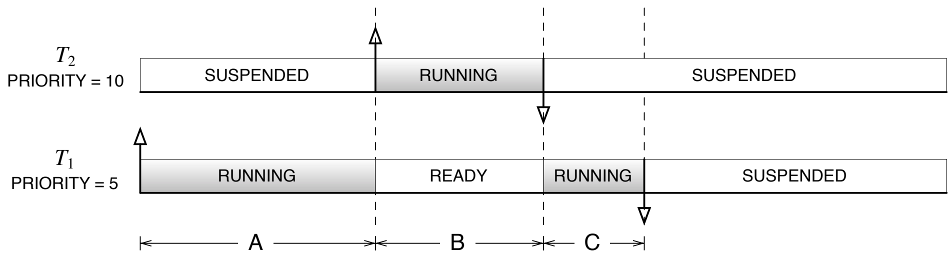


Figure 3.3: ***Scheduling of preemptable tasks.*** *During A period, T_1 is* RUNNING *and T_2 is* SUSPENDED*. Then T_2 is activated. Since $Prio(T_2) > Prio(T_1)$, T_1 is preempted and T_2 runs (B period). T_2 terminates and T_1 becomes* RUNNING *again (C period) until it terminates.*

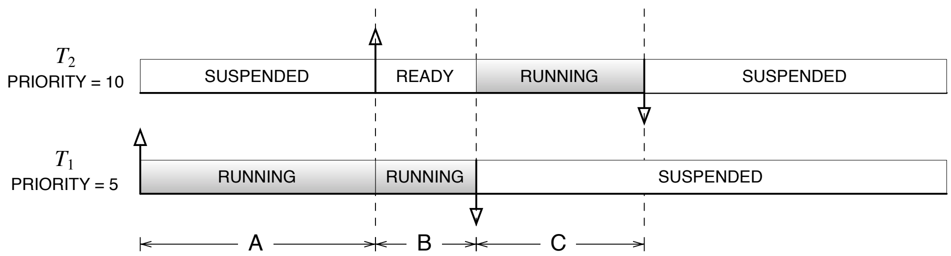


Figure 3.4: ***Scheduling of non-preemptable tasks.*** *During A period, T_1 is* RUNNING *and T_2 is* SUSPENDED*. Then T_2 is activated. Even if $Prio(T_2) > Prio(T_1)$, T_1 is non-preemptable and continues to run until it terminates (B period). In the meantime, T_2 is* READY*. T_1 terminates and T_2 runs (C period) until it terminates.*

3.3 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 */

  TerminateTask();
}
```

The code of the task is plain C.

The task should always end with a call to the `TerminateTask` service. See 4.4.4.

3.4 Tasks services

3.4.1 DeclareTask

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.

DeclareTask is a C macro. 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.

Prototype of DeclareTask:

`DeclareTask(TaskType TaskID);`

Arguments of DeclareTask:

`TaskID` The id of the task to declare.

3.4.2 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 4.5, 4.6 and 4.8 show 2 examples of task activation.

Prototype of ActivateTask:

`StatusType ActivateTask(TaskType TaskID);`

Arguments of ActivateTask:

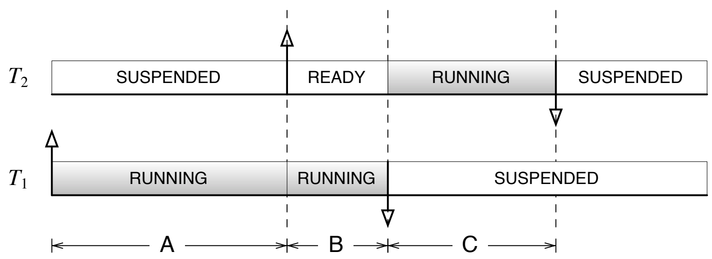
`TaskID` The id of the task to activate.

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).

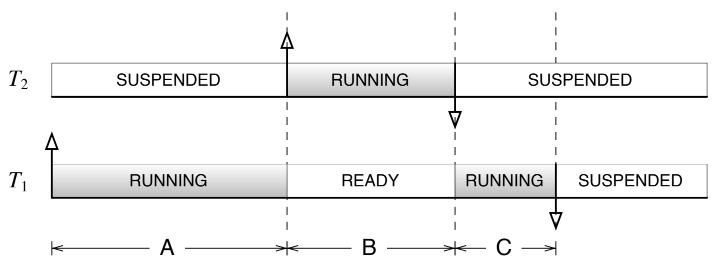
`E_OS_LIMIT` Too many activations of the task (extended and standard).




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

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

Figure 3.5: ***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.*



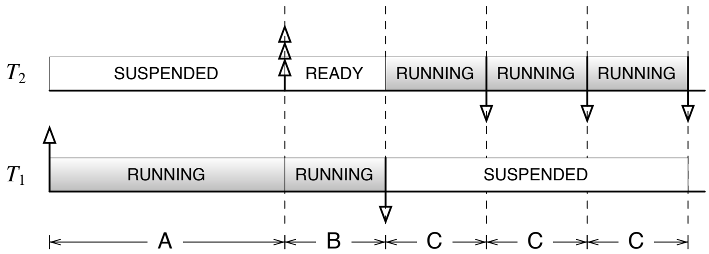

```
TASK(T2) {
  ... /* B period */
  TerminateTask();
}

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

Figure 3.6: ***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.*

3.4.3 ChainTask

This service does a rescheduling




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

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

Figure 3.7: ***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).*

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).

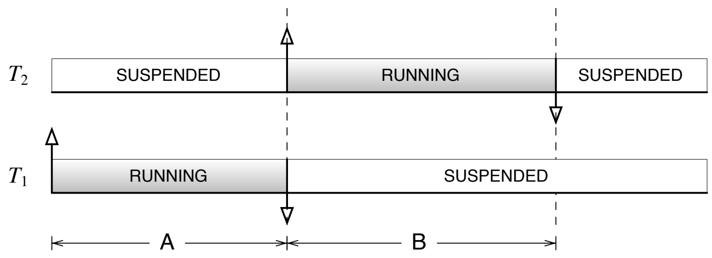
`E_OS_RESOURCE` The calling task still held a resource (extended only).

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

3.4.4 TerminateTask

This service does a rescheduling

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




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

TASK(T1) {
  ... /* A period */
  ChainTask(T2);
}
```

Figure 3.8: ***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).*

Prototype of TerminateTask:

`StatusType TerminateTask(void);`

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).

3.4.5 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 not effect. If called from a preemptable or a task that uses an internal resource, the priority of the task revert to its base priority and a rescheduling occurs.

Schedule allows to implement cooperative multitasking to insure synchronous rescheduling.

Prototype of Schedule:

`StatusType Schedule(void);`

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).

3.4.6 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 of 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).

3.4.7 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).

3.5 Inside Task management

3.5.1 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).

3.5.2 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 4.1 and 4.5.3.

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 **??**.

3.5.3 Additional task states

In addition to states presented in section 4.1, 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 4.9 show a complete task state automaton for both basic and extended tasks with these states added.

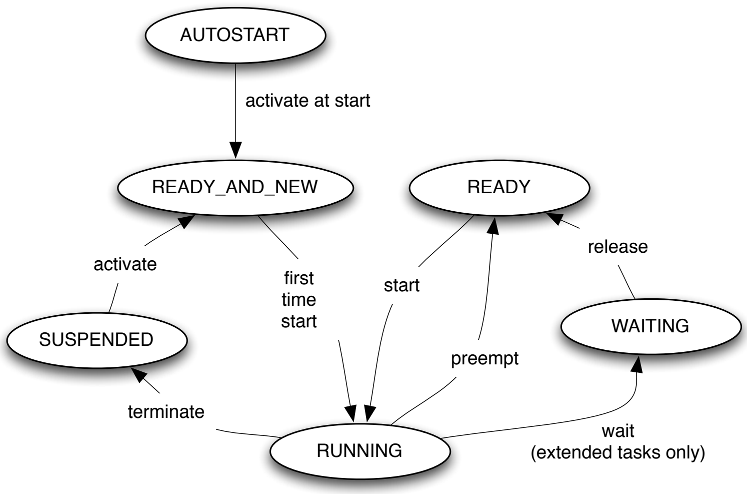


Figure 3.9: *States of a task in Trampoline. `AUTOSTART` is the initial state of autostart tasks. `SUSPENDED` is the initial state of both non autostart tasks.*

3.6 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 tasks 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`'.

CHAPTER

FOUR

ALARMS

ALarms are used to perform an action after an interval of time for a single shot alarm and periodically for a periodic alarm. The action may be the activation of a task, the setting of an event to a task or the execution of an alarm callback function[1].

4.1 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 4.1 shows the states of a basic task. An extended task has an additional `WAITING` state. Figure 4.2 shows the states of an extended task. See section 4.5.3 for additional informations about the states of a task.

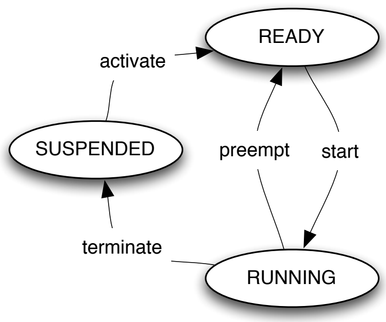


Figure 4.1: *States of a `BASIC` task.*

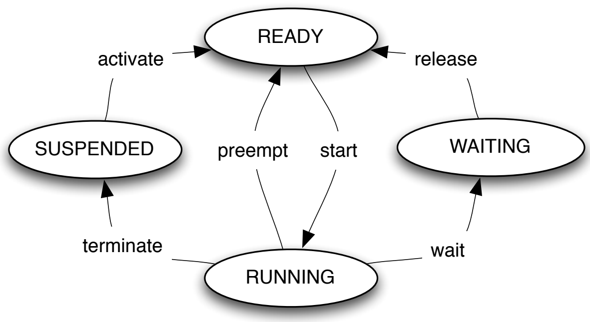


Figure 4.2: *States of an `EXTENDED` task.*

A task goes from one state to the other according to various conditions as shown in table 4.1.

[1]This third action is not available in AUTOSAR

Table 4.1: *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 `ActivateTask` or `ChainTask`, 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 `TerminateTask` or `ChainTask` 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 `WaitEvent`.
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.

4.2 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 5.

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 ↑ indicates a task activation and a down arrow ↓ a task termination.

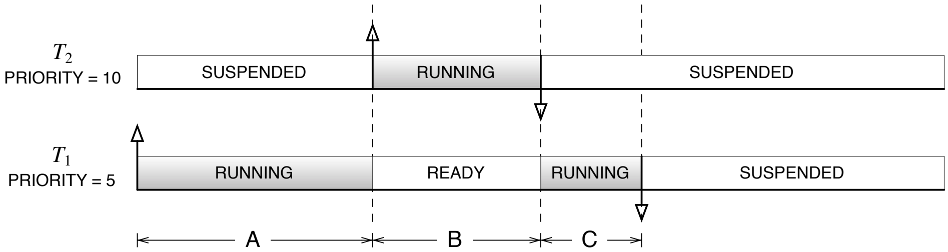


Figure 4.3: ***Scheduling of preemptable tasks.*** *During A period, T_1 is* RUNNING *and T_2 is* SUSPENDED*. Then T_2 is activated. Since $Prio(T_2) > Prio(T_1)$, T_1 is preempted and T_2 runs (B period). T_2 terminates and T_1 becomes* RUNNING *again (C period) until it terminates.*

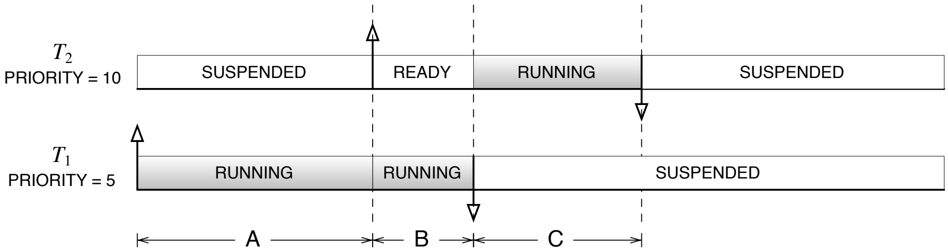


Figure 4.4: ***Scheduling of non-preemptable tasks.*** *During A period, T_1 is* RUNNING *and T_2 is* SUSPENDED*. Then T_2 is activated. Even if $Prio(T_2) > Prio(T_1)$, T_1 is non-preemptable and continues to run until it terminates (B period). In the meantime, T_2 is* READY*. T_1 terminates and T_2 runs (C period) until it terminates.*

4.3 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 */

  TerminateTask();
}
```

The code of the task is plain C.

The task should always end with a call to the `TerminateTask` service. See 4.4.4.

4.4 Tasks services

4.4.1 DeclareTask

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.

DeclareTask is a C macro. 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.

Prototype of DeclareTask:

`DeclareTask(TaskType TaskID);`

Arguments of DeclareTask:

`TaskID` The id of the task to declare.

4.4.2 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 4.5, 4.6 and 4.8 show 2 examples of task activation.

Prototype of ActivateTask:

`StatusType ActivateTask(TaskType TaskID);`

Arguments of ActivateTask:

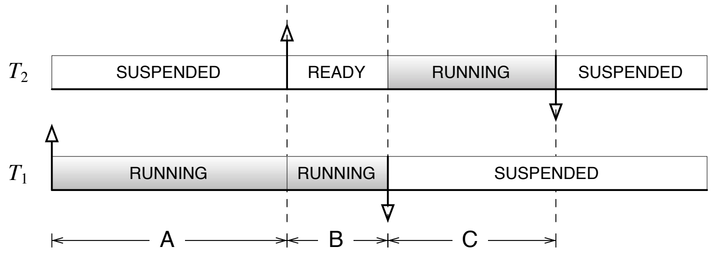
`TaskID` The id of the task to activate.

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).

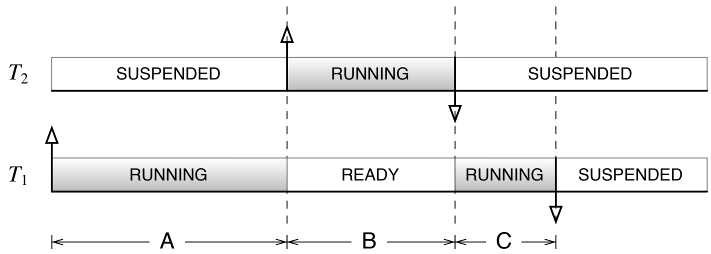
`E_OS_LIMIT` Too many activations of the task (extended and standard).




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

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

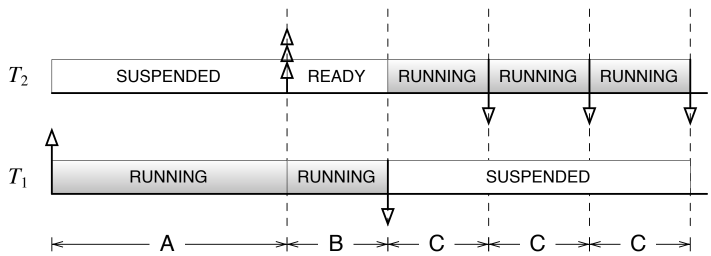
Figure 4.5: ***Activation of a lower priority task.*** *Prio(T_1) $\geq$ Prio(T_2). During A period, T_1 is* RUNNING *and T_2 is* SUSPENDED*. Then T_1 calls* `ActivateTask(T2);`*. Since T_2 does not have a higher priority, it becomes* READY *(B period). T_1 terminates and T_2 runs (C period) until it terminates.*




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

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

Figure 4.6: ***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 4.7: ***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).*

4.4.3 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:

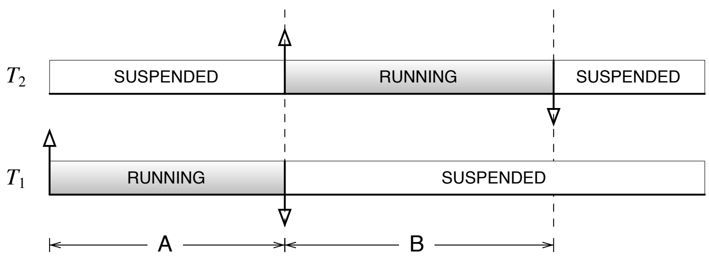
`E_OK` No error, the task TaskID has been successfully activated and the calling task has been successfully terminated. Note in this case `ChainTask` does not return so actually E_OK is never returned (extended and standard).

`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).




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

TASK(T1) {
  ... /* A period */
  ChainTask(T2);
}
```

Figure 4.8: ***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).*

4.4.4 TerminateTask

This service does a rescheduling

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

Prototype of TerminateTask:

```
StatusType TerminateTask(void);
```

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).

4.4.5 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 not effect. If called from a preemptable or a task that uses an internal resource, the priority of the task revert to its base priority and a rescheduling occurs.

Schedule allows to implement cooperative multitasking to insure synchronous rescheduling.

Prototype of Schedule:

`StatusType Schedule(void);`

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).

4.4.6 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 of 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).

4.4.7 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).

4.5 Inside Task management

4.5.1 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).

4.5.2 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 4.1 and 4.5.3.

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 **??**.

4.5.3 Additional task states

In addition to states presented in section 4.1, 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 4.9 show a complete task state automaton for both basic and extended tasks with these states added.

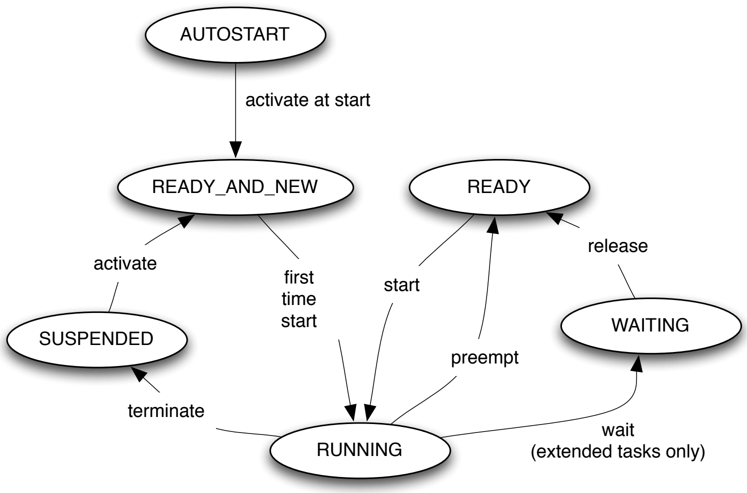


Figure 4.9: *States of a task in Trampoline. `AUTOSTART` is the initial state of autostart tasks. `SUSPENDED` is the initial state of both non autostart tasks.*

4.6 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 tasks 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`'.

CHAPTER

FIVE

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 5.1 shows an example of race condition.

5.1 OSEK Priority Ceiling Protocol

OSEK uses a modified version of the Priority Ceiling Protocol [7]. 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 $T_1, T_2, \ldots, T_n$ a set of tasks sharing the same resource R and $P_1, P_2, \ldots, P_n$ 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 5.2 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.

5.2 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

```
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 5.1: ***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*

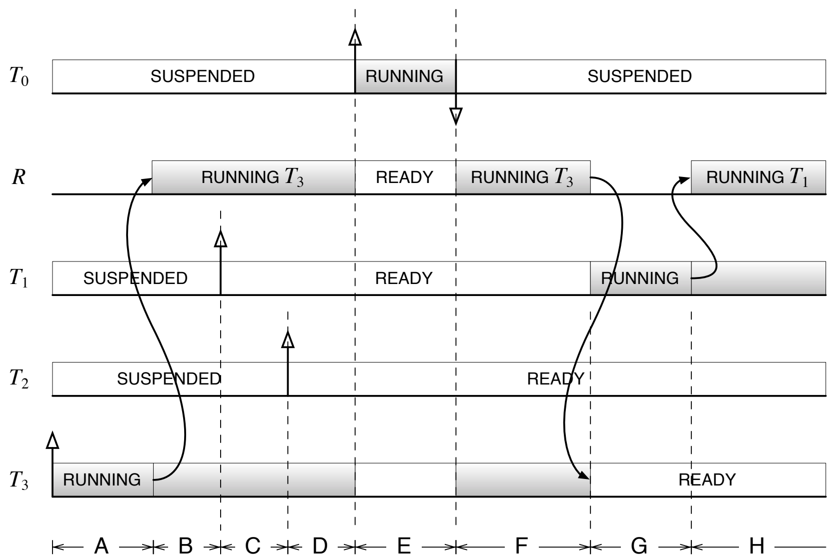


Figure 5.2: ***Scheduling with a resource used by 3 tasks and a fourth task having a higher priority.*** *$P(T_0) > P(T_1) > P(T_2) > P(T_3)$. R is used by T_1, T_2 and T_3 so $P(T_0) > P(R) \geq P(T_1)$. During A period, T_3 is* RUNNING *and other tasks are* SUSPENDED*. Then T_3 gets R and $P(T_3) \leftarrow P(R)$ (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).*

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.

5.3 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`.

At most one internal resource may be used by a task.

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.

5.4 Nested resources accesses

Resources may be accessed in a nested way. That is once a resource is got, another one may be got before releasing the first one and so on. However resources must be released in the reverse order they have been got as if they were pushed on a stack. The following example shows the good usage of resources:

```
TASK(MyTask)
{
  GetResource(rez1);
  ...
  /* critical section protected by rez1 */
  ...
  GetResource(rez2);
  ...
  /* critical section protected by rez2 and rez1 */
  ...
  ReleseResource(rez2);
  ...
  /* more critical section protected by rez1 */
  ...
  ReleaseResource(rez1);
  TerminateTask();
}
```

5.5 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 };
};
```

⚠ Every task and ISR that uses a resource in the C code must declare it in the OIL file. Otherwise `goil` will compute a wrong priority for the resource and the scheduling of tasks and the execution of ISR will not be as expected.

5.6 Resources services

5.6.1 DeclareResource

Each resource has an identifier of type `ResourceType`. This identifier is declared in the OIL file and is used in system calls to refer to a particular resource. DeclareResource declares a resource exists. The result is to make the id of the resource available and allows to use it in services' calls.

DeclareResource is a C macro

Prototype of DeclareResource:

`DeclareResource(ResourceType ResourceID);`

Arguments of DeclareResource:

`ResourceID` The id of the resource.

5.6.2 GetResource

GetResource enters the critical section protected by the resource. For each call to GetResource, a corresponding call to `ReleaseResource` must be made in the control flow of the task or ISR. Nested calls are allowed, see 5.4 for nested resource accesses.

Prototype of GetResource:

`StatusType GetResource(ResourceType ResourceID);`

Arguments of GetResource:

`ResourceID` The id of the resource to get.

Status codes returned by GetResource:

`E_OK` No error (extended and standard).

`E_OS_ID` Invalide resource id. No resource with such an id exists (extended and standard).

`E_OS_ACCESS` The resource is already taken by a task or an ISR or has a priority lower than the base priority of the calling task or ISR. This should not happen if the application is configured correctly except if the same task or ISR try to get the same resource twice (extended only).

5.6.3 ReleaseResource

ReleaseResource leaves the critical section protected by the resource. For each call to ReleaseResource, a corresponding call to `GetResource` must have been made in the control flow of the task or ISR. Nested calls are allowed, see 5.4 for nested resource accesses.

This service does a rescheduling

Prototype of ReleaseResource:

`StatusType ReleaseResource(ResourceType ResourceID);`

Arguments of ReleaseResource:

`ResourceID` the id of the resource.

Status codes returned by ReleaseResource:

`E_OK` No error (extended and standard).

`E_OS_ID` Invalide resource id. No resource with such an id exists (extended and standard).

CHAPTER

SIX

EVENTS

Events are used to synchronize an extended task to a condition external to the task. Each extended task has a private set of events (it owns the event) and an event is explicitly sent to a task. Having the same event attributed to many tasks does not mean the tasks share the event. They share only the value (or mask) associated to the event.

Events may be set by any other task, by an ISR2, by an alarm, by a schedule table or by the arrival of a message. Any task or ISR may read the events of a task but only the extended task owning the event is able to wait for it or to clear it.

If you use AUTOSAR OS Applications, involved objects must belong to the same OS Application or must have an access right to the OS Application of the target task.

A `RUNNING` task that wait for an event is put in the `WAITING` state if the event has not occured or stay in the `RUNNING` state if it has already occured.

A `WAITING` task is put in the `READY` state if one of the events it is waiting for occurs. See chapter 3 for more informations.

Events must be explicitly cleared once read. If a tasks does not clear the previous occurrence of an event, it will be seen as “already occurred” the next time the task will wait for it.

6.1 OIL description

An event is described using a `EVENT` object. `MASK` is the single attribute of this object. `MASK` may be set to a literal value:

```
EVENT ev {
  MASK = 0x1;
};
```

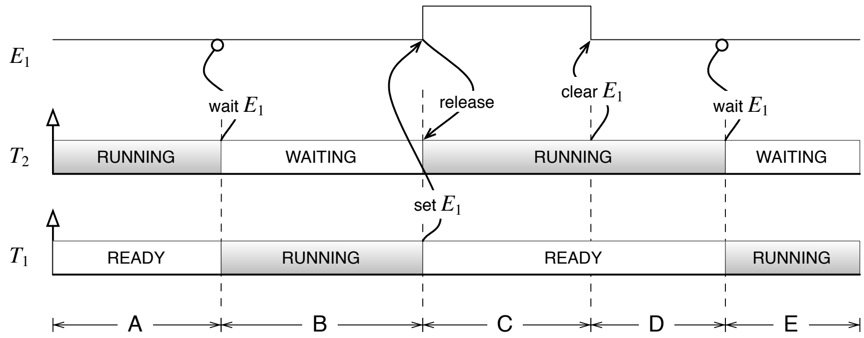


Figure 6.1: ***Scheduling with an event.*** *T_2 is an extended task. During A period, T_2 is* RUNNING *and T_1 is* READY. *Then T_2 wait for E_1 and blocks. T_2 runs (B period) and sets E_1. T_2 is released and since $P(T_2) > P(T_1)$, T_2 runs (C period), clears E_1 and continues to run (D period). Then T_2 wait for E_1 again and blocks, T_1 runs (E period).*

The literal value should have only 1 bit set. Goil emits a warning when this is not the case.

Or `MASK` may be set to `AUTO`. In this case, the system generation tool computes the event mask:

```
EVENT ev {
  MASK = AUTO;
};
```

6.2 Events services

6.2.1 SetEvent

Events of task ***`TaskID`*** are set according to the Mask passed as 2^{nd} argument. This service is non blocking and may be called from a task or an ISR2.

SetEvent may do a rescheduling if the target task is unblocked and goes to the `READY` state.

Prototype of SetEvent:

`StatusType SetEvent(TaskType TaskID, EventMaskType Mask);`

Arguments of SetEvent:

`TaskID` the id of the task.

`Mask` the event mask.

Status codes returned by SetEvent:

`E_OK` No error (extended and standard).

`E_OS_ID` Invalid TaskID (extended only).

`E_OS_ACCESS` TaskID is not an extended task (not able to manage events) (extended only).

`E_OS_STATE` Events cannot be set because the target task is in the `SUSPENDED` state (extended only).

6.2.2 WaitEvent

The calling task waits for event(s) *`Mask`*. If one the events are already set, the task continues its execution. If none of the events are set, the task is put in the `WAITING` state and blocks.

WaitEvent may do a rescheduling if the calling task blocks.

Prototype of WaitEvent:

`StatusType WaitEvent(EventMaskType Mask);`

Arguments of WaitEvent:

`Mask` The event(s) to wait for.

Status codes returned by WaitEvent:

`E_OK` No error (extended and standard).

`E_OS_ACCESS` The calling task is not an extended task (not able to manage events) (extended only).

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

`E_OS_CALLEVEL` The caller is not a task (extended only).

6.2.3 GetEvent

Events of task *`TaskID`* are copied in *`Mask`* argument passed as reference.

GetEvent does not reset the event mask. `ClearEvent` should be used to clear, in the event mask, the events that have been processed.

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

```
EventMaskType myEventMask;
GetEvent(aTask, &myEventMask);
```

Prototype of GetEvent:

`StatusType GetEvent(TaskType TaskID, EventMaskRefType Mask);`

Arguments of GetEvent:

`TaskID` the id of the task.

`Mask` the reference of the event mask where the *`TaskID`* event mask is copied.

Status codes returned by GetEvent:

`E_OK` No error (extended and standard).

`E_OS_ID` Invalid TaskID (extended only).

`E_OS_ACCESS` The task identified by TaskID is not an extended task (not able to manage events) or, in AUTOSAR, the caller cannot access the task (extended only).

`E_OS_STATE` The task identified by TaskID is in `SUSPENDED` state (extended only).

CHAPTER

SEVEN

OS APPLICATIONS

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

7.1 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.

CHAPTER

EIGHT

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*.

8.1 Low Level Functions

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

8.1.1 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.

8.1.2 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.

CHAPTER

NINE

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 9.1 below.

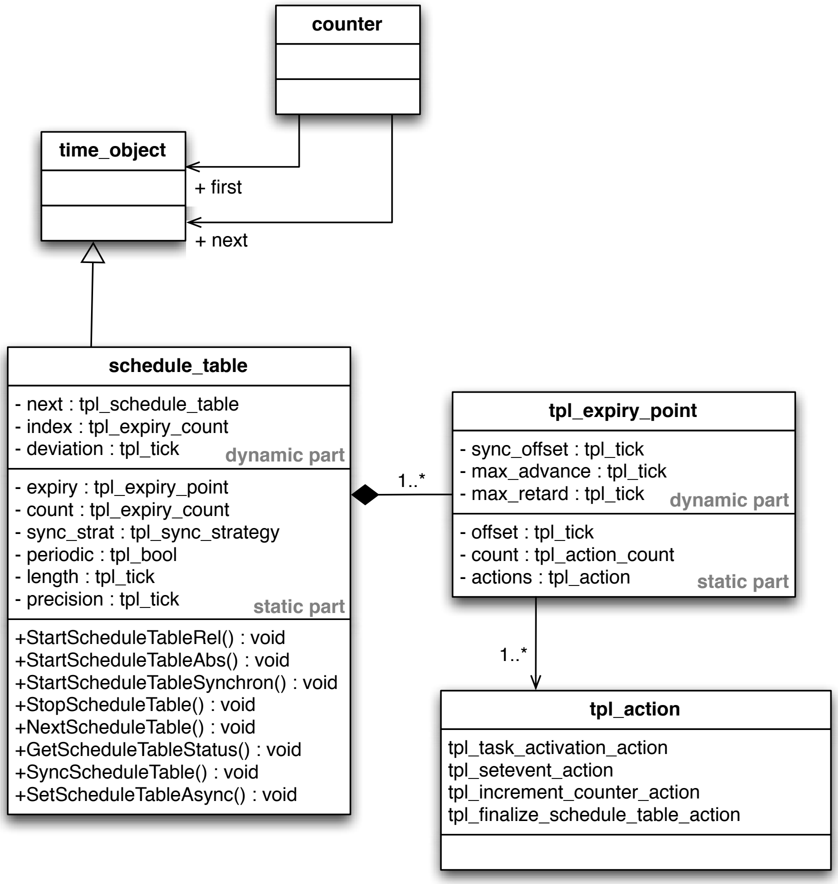


Figure 9.1: *Schedule table class diagram*

9.1 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 date[1]. It can happen when :

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

[1] As 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.

comes between the current date and <tick_val>

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

Figure 9.2 below shows a bootstrap example for the first item.

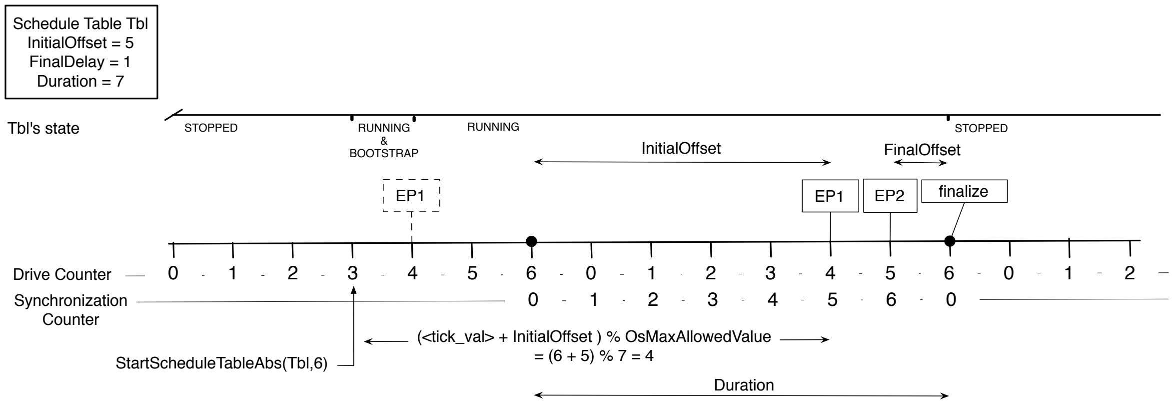


Figure 9.2: *Bootstrap example*

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 9.1: *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 9.3 shows how a schedule table goes from state to state.

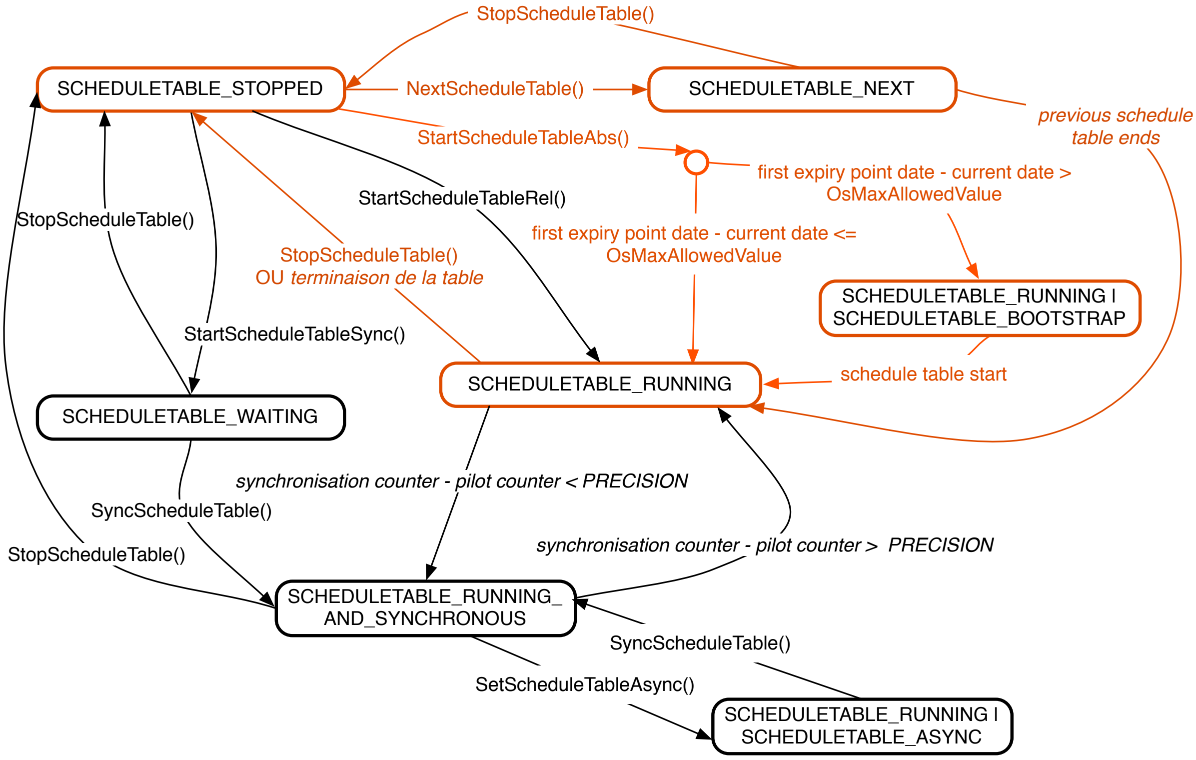


Figure 9.3: *States of a schedule table in Trampoline.*

9.2 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 delared in the OIL file : the "finalize" expiry point (see Figure 9.2). 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 epiry point too.

A schedule table is a time object, like an alarm. `tpl_processing_scheduletable()` 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 9.4.

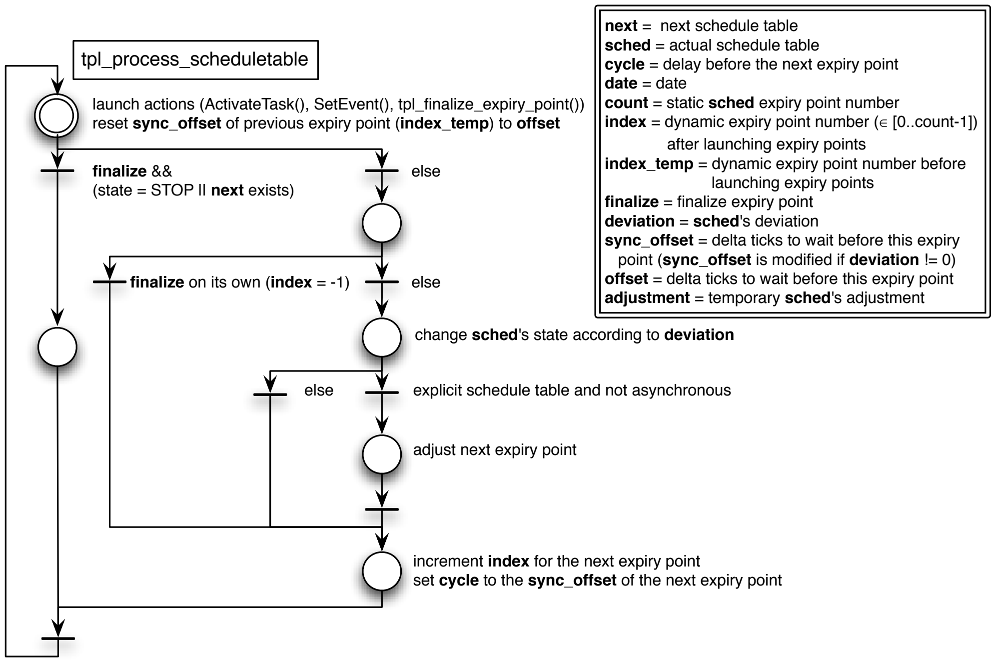
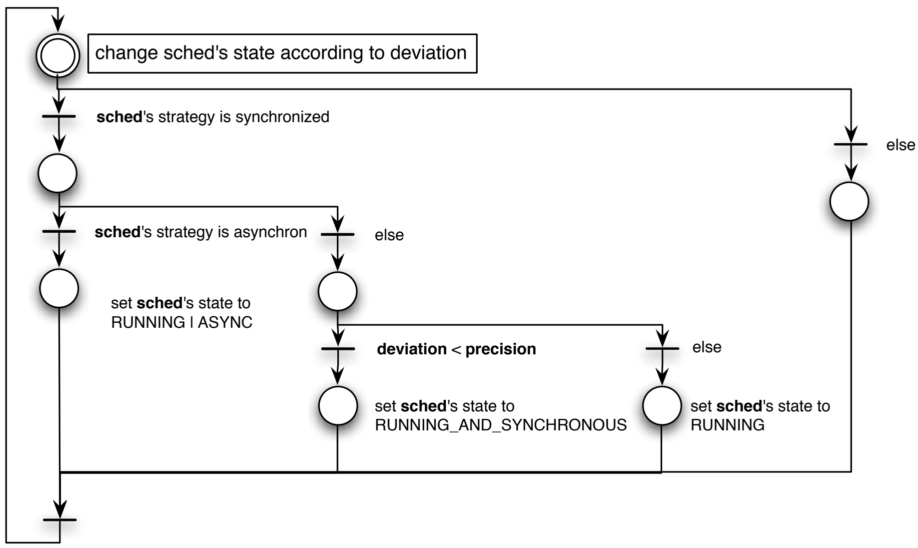
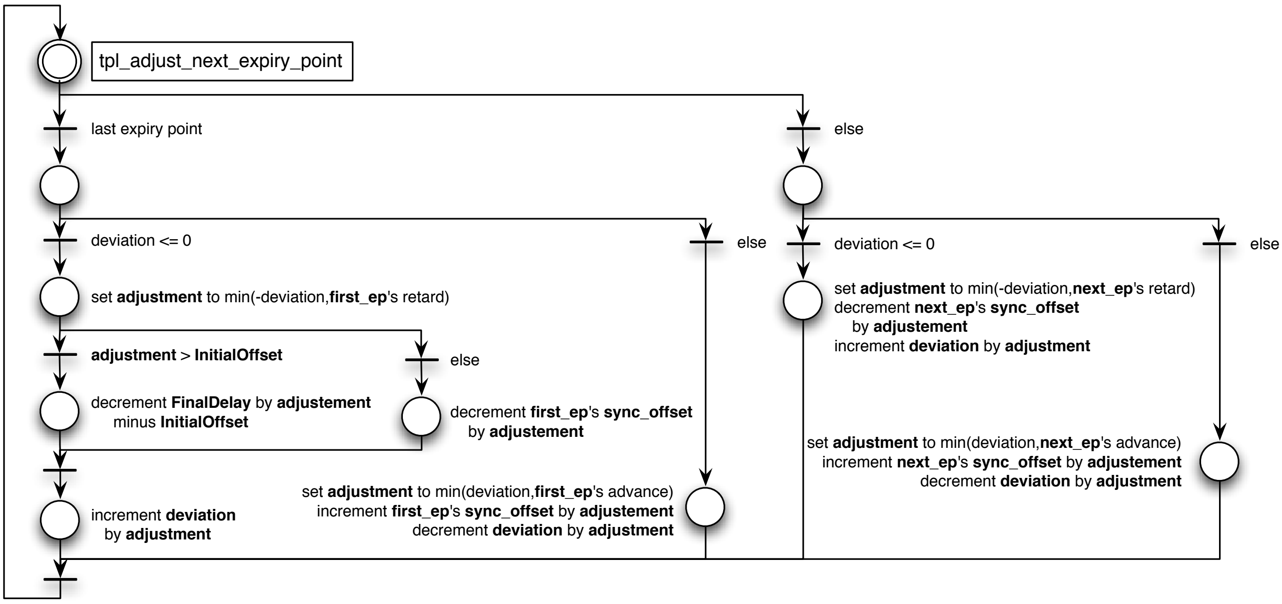


Figure 9.4: *tpl_process_scheduletable's state machine.*





`tpl_finalize_expoiry_point()` state machine is shown in Figure 9.5 below.

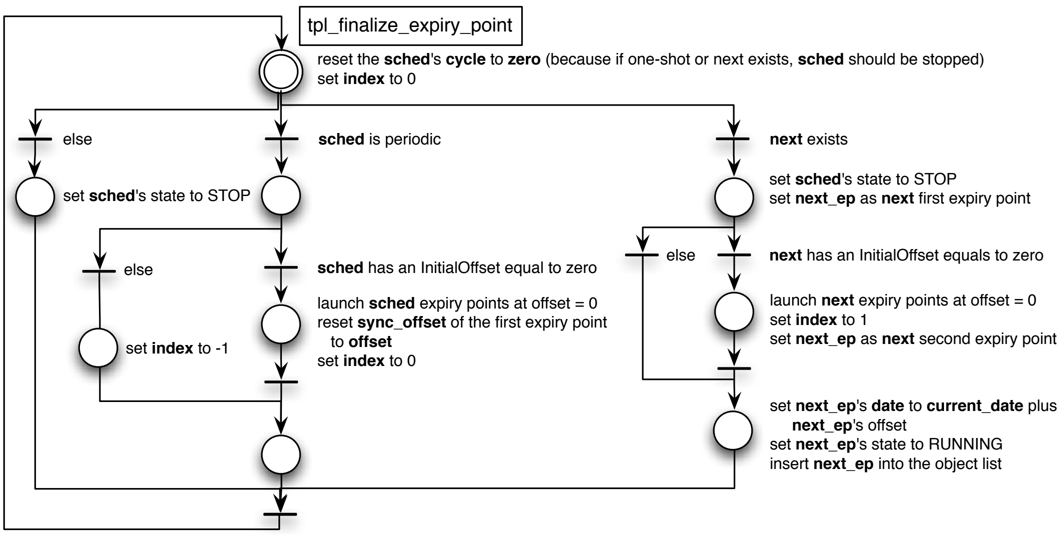


Figure 9.5: *tpl_finalize_expiry_point's state machine.*

CHAPTER

TEN

THE COMMUNICATION LIBRARY

An OSEK/COM compliant library is part of Trampoline. This chapter presents the communication configuration and API. Implementation details as well as examples of extension are provided at the end of the chapter.

10.1 Implementation

10.1.1 Sending Message Objects

In the following paragraphs, acronyms are widely used. Here is the meaning of these acronyms:

MO Message Object

SMO Sending Message Object

RMO Receiving Message Object

Base Sending Message Object

The Base SMO is an abstract *class* that is the common part of all SMOs. Since a SMO may be wired to an IPDU for external communication or a RMO for internal communication, the BSMO type is only a structure with one member: `sender`, a function pointer to a function doing the actual work according to the kind of SMO.

It is easy to extend the communication library by providing a sending function that will manage message sending to a different kind of destination than the standard OSEK/COM one.

The BSMO is declared as follow:

```
struct TPL_BASE_SENDING_MO {
```

```
    tpl_sending_func     sender; /* pointer to the sending function   */
};

typedef struct TPL_BASE_SENDING_MO tpl_base_sending_mo;
```

The sending function has the following prototype:

```
typedef tpl_status (*tpl_sending_func)(
  P2CONST(void, AUTOMATIC, OS_CODE),
  CONSTP2CONST(tpl_com_data, AUTOMATIC, OS_VAR));
```

The first argument is a pointer to the SMO and the second argument is a pointer to the data to be sent.

Internal Sending Message Object

The first concrete *subclass* of `tpl_base_sending_mo` is the `tpl_internal_sending_mo` structure. This structure adds `internal_target`, a pointer to a `tpl_base_receiving_mo` (see 10.1.2) which is the first RMO of a chained list of RMOs:

```
struct TPL_INTERNAL_SENDING_MO {
  /*  common to all sending mo                          */
  tpl_base_sending_mo            base_mo;
  /*  pointer to the internal receiving message object    */
  struct TPL_BASE_RECEIVING_MO    *internal_target;
};
```

10.1.2 Receiving Message Objects

Base Receiving Message Object

The root type is the `tpl_base_receiving_mo` structure. This structure contains two members, `notification` and `next_mo`:

```
struct TPL_BASE_RECEIVING_MO {
   /*! notification structure             */
   tpl_action                   *notification;
   /*! message objects chaining           */
   struct TPL_BASE_RECEIVING_MO    *next_mo;
};
```

`notification` is a pointer to a notification descriptor and is used to perform the notification associated to the receiving message object. `next_mo` is a pointer to another RMO which allows to chain RMOs[1].

Data Receiving Message Object

An abstract *subclass* of `tpl_base_receiving_mo` exists: `tpl_data_receiving_mo`. This *subclass* extends `tpl_base_receiving_mo` and adds the following data related members:

[1]In OSEK/COM a sending message may have more than one RMO

```
struct TPL_DATA_RECEIVING_MO {
   /* common part of the receiving message objects */
   tpl_base_receiving_mo   base_mo;
   /* pointer to the receiving function            */
   tpl_receiving_func      receiver;
   /* pointer to the data copy function            */
   tpl_data_copy_func      copier;
   /* filter descriptor                            */
   tpl_filter_desc         *filter;
};
```

`receiver` is a pointer to a receiving function (ie the function that will copy the data from a source to the destination message object). This function has the following prototype:

```
typedef tpl_status (*tpl_receiving_func)(
   void *,
   tpl_com_data *
);
```

The first argument is a pointer to the RMO and the second one is a pointer to the data to copy in the RMO.

`copier` is a pointer to a function that is used to copy the data from the receiving message object to the application. It is called by the `ReceiveMessage` system service. This function has the following prototype:

```
typedef tpl_status (*tpl_data_copy_func)(
   tpl_com_data *,
   void *
);
```

The first argument is a pointer to the data to copy from the RMO and the second one is a pointer to the RMO.

`filter` is a pointer to a filter descriptor.

CHAPTER

ELEVEN

THE INTER OS-APPLICATION COMMUNICATION LIBRARY

INter OS-application Communication library is an API initially dedicated to communications between tasks from different OS-applications in multicore systems. However, it could also be used for communications between tasks from a same OS-Application. In the fallowing, Inter OS-application Communication will be denoted IOC. This chapter presents the IOC configuration and API. Implementation details as well as examples of utilization are provided.

11.1 IOC declaration in OIL

The IOC configuration is performed using OIL. Parameters such as IOC name, the type of manipulated data, the kind of communication (queued or last is best) and informations about sender/receiver are mandatory. The syntax is presented below using tow example.

Let us consider the case where a task A (as part of OS-application *os-app1*) sends a data to a task B (as part of OS-application *os-app2*). In the first case, we consider a last is best semantic communication where only one data of type u8 is sent. In the second case, we consider a queued semantic communication where a data of type u8 and a data of type *mytype* (defined by user) are sent. It is worth noting that this type have to be defined by user un the file *ioc_types.h* at the root of the project directory.

mytype can be defined like this:

```
struct mytype {
  u8       a;
  u8       b,
}


/* LAST_IS_BEST semantic */
```

```
IOC com_A_to_B_last_is_best {
  DATATYPENAME u8 {
    DATATYPEPROPERTY = DATA;
  };
  SEMANTICS = LAST_IS_BEST {
    INIT_VALUE_SYMBOL = AUTO;
  };
  RECEIVER rcv {
    RCV_OSAPPLICATION = os-app2;
    RECEIVER_PULL_CB = AUTO;
    ACTION = NONE;
  };
  SENDER sender0 {
    SENDER_ID = 0;
    SND_OSAPPLICATION = os-app1;
  };
};

/* QUEUED semantic */

IOC com_A_to_B_queued {
  DATATYPENAME u8 {
    DATATYPEPROPERTY = DATA;
  };
  DATATYPENAME mytype {
    DATATYPEPROPERTY = REFERENCE;
  };
  SEMANTICS = QUEUED {
    BUFFER_LENGTH = 2;
  };
  RECEIVER rcv {
    RCV_OSAPPLICATION = os-app2;
    RECEIVER_PULL_CB = AUTO;
    ACTION = NONE;
  };
  SENDER sender0 {
    SENDER_ID = 0;
    SND_OSAPPLICATION = os-app1;
  };
};
```

The DATATYPENAME parameter defines the name of the data type to be transferred. A file named *ioc_types.h* should be created by user in order to defined new types, if any. The associated property specifies if the data is passed to sending functions by reference or by value. It is worth noting that it is possible to specify many DATATYPENAME as illustrated with the second example. In that case, the applicative sending function should have as many parameters as the number of DATATYPE specify in the OIL file. In case of a last is best semantic, the INIT_VALUE_SUMBOL defines the initial data value. It can be set to AUTO is there are no initial value. Otherwise, the INIT_VALUE_SYMBOL is a string type defined by user and the function *IOC_init()* has to be called at the beginning of application. In case of a queued semantic, only a BUFFER_LENGTH has to be specified. The receiver configuration requires the setting of the target OS-application (RCV_OSAPPLICATION), the king of task notification used when the message has arrived (ACTION = ACTIVATETASK, SETEVENT or NONE) (not functional at the moment) and the callback function to call (not functional at the moment). The sender configuration require the SENDER_ID, as an integer, and the sender OS-application (SND_OSAPPLICATION).

11.2 Implementation

The IOC is divided in two set of source files. First, the APIs (part of the OS) containing kernel functions are generic. They can be found in *ioc/* directory. Second, specific files for the IOC configuration are generated. The IOC API is very closed to internal communication library and will not be detailed here. Let us now detailed what is generated in *tpl_ioc_api_config.c*.

In case of the last is best communication (example 1), the sending operation is performed by the call of *IocWrite_IocName()* function and the receiving operation, by the call of the function *IocRead_IocName()*. These functions have to be called directly by user in applicative functions. The generated part of the API transmit the request to the kernel. Let us now illustrated the generated code for the first example.

```
FUNC(Std_ReturnType, OS_CODE) IocWrite_com_A_to_B_last_is_best(
  VAR(u8, AUTOMATIC) IN0 /* one data is send */
)
{
  /* only one data implies only one element in the message table */
  VAR(tpl_ioc_message, AUTOMATIC) message[1];
  VAR(Std_ReturnType, AUTOMATIC) result;

  /* Fill in the message structure with the data address and its size */
  message[0].data=(tpl_ioc_data *)&IN0;
  message[0].length=sizeof(u8);

  /* Call the kernel function */
  result = IOC_Write(0, message);

  return result;
}

FUNC(Std_ReturnType, OS_CODE) IocRead_com_A_to_B_last_is_best(
  P2VAR(u8, AUTOMATIC, OS_APPL_DATA) IN0
)
{
  VAR(tpl_ioc_message, AUTOMATIC) message[1];
  VAR(Std_ReturnType, AUTOMATIC) result;

  message[0].data=(tpl_ioc_data *)IN0;
  message[0].length=sizeof(u8);

  /* Call the kernel function */
  result = IOC_Read(0, message);
```

```
  return result;
}
```

In the case of a queued communication, the sending and receiving operations are performed by the call of *IocSend_IocName()* and *IocReceive_IocName()* respectively. Generated functions would be of the same form that in last is best case.

Finally, it is possible that several senders send a same data. In that case, many senders can be defined during the OIL configuration. In the applicative functions, user have to call API functions of type *IocWrite_IocName_SenderName() or IocSend_IocName_SenderName()* when sending a message.

CHAPTER

TWELVE

MEMORY MAPPING

THE AUTOSAR consortium has defined a set of macros [4] in order to adapt the memory mapping directives to the different existing compilers. Indeed, memory mapping directives are not part of the C language and it is therefore impossible to write portable code between different compilers without going through this set of macros. In addition, some MCUs have a segmented memory model and require additional pointer directives to specify whether the pointer is to data in the same segment or to data in a different segment. In the first case, it is a *near* pointer (usually stored in a 16 bits word). In the second case it is a *far* pointer (usually stored in a 24 bits word).

It remains that these macros are not particularly intuitive in their use and require some explanations that we will give here.

12.1 Memory mapping directives

Memory mapping consists in assigning to each object of the application (variables, constants, functions) and of the operating system a named memory area where the object will be stored. Memory mapping directives take various forms depending on the compiler. For example, putting a function named `f` in the memory area `.osCode` when using `gcc` is done as follows:

```
void __attribute__ ((section (".osCode"))) f() { ... }
```

while doing the same thing using Freescale's CodeWarrior compiler (previously Metrowerks) requires the following directive:

```
#pragma section code_type ".osCode"
void f() { ... }
```

AUTOSAR defines several macros to encapsulate these directives and these macros work with the declarations of the memory sections.

12.2 The memory sections

For each task and ISR declared in the OIL file, Goil generates several memory sections. These sections are selected via macro definitions with names of the form `APP_Task_<name>_START_SEC_-<section_type>` and `APP_Task_<name>_STOP_SEC_<section_type>` for tasks and `APP_ISR_<name>_-START_SEC_<section_type>` and `APP_ISR_<name>_STOP_SEC_<section_type>` for ISRs. `<name>` is the name of the task or ISR and `<section_type>` is the type of section. The section types are as follows:

CODE is the section used for the process code and for the functions called by the process. If, for example, the task t1 is declared in the OIL file, its code will be written in C as follows:

```
#define APP_Task_t1_START_SEC_CODE
#include "MemMap.h"
TASK(t1)
{
  ...
  TerminateTask();
}
#define APP_Task_t1_STOP_SEC_CODE
#include "MemMap.h"
```

STACK is the section used for the process stack. This section is used in the files generated by `goil`.

VAR_<init_policy>_<alignment> are the sections used for process globals or static variables. `<init_policy>` can take the following values:

NOINIT for uninitialized variables.

POWER_ON_INIT for variables initialized at MCU startup.

`<alignment>` can take the following values:

32BIT for 4 bytes alignment.

16BIT for 2 bytes alignment.

8BIT for 1 byte alignment.

UNSPECIFIED for data sizes that do not fit into any of the other categories.

CONST_<alignment> are the sections used for process globals constants.

For example, if task t1 uses two 8-bit constants, *c1* and *c2*, and one 32-bit variable, *v1*, uninitialized, they will be declared as follows:

```
#define APP_Task_t1_START_SEC_CONST_8BIT
#include "MemMap.h"
CONST(uint8, AUTOMATIC) c1 = 3;
CONST(uint8, AUTOMATIC) c2 = 7;
#define APP_Task_t1_STOP_SEC_CONST_8BIT
#include "MemMap.h"

#define APP_Task_t1_START_SEC_VAR_NOINIT_32BIT
```

```
#include "MemMap.h"
VAR(uint8, AUTOMATIC) v1;
#define APP_Task_t1_STOP_SEC_VAR_NOINIT_32BIT
#include "MemMap.h"
```

Table 12.1: *Sections generated for task t1*

APP_Task_toto_START_SEC_CODE	APP_Task_toto_STOP_SEC_CODE
APP_Task_toto_START_SEC_STACK	APP_Task_toto_STOP_SEC_STACK
APP_Task_toto_START_SEC_VAR_NOINIT_32BIT	APP_Task_toto_STOP_SEC_VAR_NOINIT_32BIT
APP_Task_toto_START_SEC_VAR_NOINIT_16BIT	APP_Task_toto_STOP_SEC_VAR_NOINIT_16BIT
APP_Task_toto_START_SEC_VAR_NOINIT_8BIT	APP_Task_toto_STOP_SEC_VAR_NOINIT_8BIT
APP_Task_toto_START_SEC_VAR_NOINIT_BOOLEAN	APP_Task_toto_STOP_SEC_VAR_NOINIT_BOOLEAN
APP_Task_toto_START_SEC_VAR_NOINIT_UNSPECIFIED	APP_Task_toto_STOP_SEC_VAR_NOINIT_UNSPECIFIED
APP_Task_toto_START_SEC_VAR_POWER_ON_INIT_32BIT	APP_Task_toto_STOP_SEC_VAR_POWER_ON_INIT_32BIT
APP_Task_toto_START_SEC_VAR_POWER_ON_INIT_16BIT	APP_Task_toto_STOP_SEC_VAR_POWER_ON_INIT_16BIT
APP_Task_toto_START_SEC_VAR_POWER_ON_INIT_8BIT	APP_Task_toto_STOP_SEC_VAR_POWER_ON_INIT_8BIT
APP_Task_toto_START_SEC_VAR_POWER_ON_INIT_BOOLEAN	APP_Task_toto_STOP_SEC_VAR_POWER_ON_INIT_BOOLEAN
APP_Task_toto_START_SEC_VAR_POWER_ON_INIT_UNSPECIFIED	APP_Task_toto_STOP_SEC_VAR_POWER_ON_INIT_UNSPECIFIED
APP_Task_toto_START_SEC_VAR_FAST_32BIT	APP_Task_toto_STOP_SEC_VAR_FAST_32BIT
APP_Task_toto_START_SEC_VAR_FAST_16BIT	APP_Task_toto_STOP_SEC_VAR_FAST_16BIT
APP_Task_toto_START_SEC_VAR_FAST_8BIT	APP_Task_toto_STOP_SEC_VAR_FAST_8BIT
APP_Task_toto_START_SEC_VAR_FAST_BOOLEAN	APP_Task_toto_STOP_SEC_VAR_FAST_BOOLEAN
APP_Task_toto_START_SEC_VAR_FAST_UNSPECIFIED	APP_Task_toto_STOP_SEC_VAR_FAST_UNSPECIFIED
APP_Task_toto_START_SEC_VAR_32BIT	APP_Task_toto_STOP_SEC_VAR_32BIT
APP_Task_toto_START_SEC_VAR_16BIT	APP_Task_toto_STOP_SEC_VAR_16BIT
APP_Task_toto_START_SEC_VAR_8BIT	APP_Task_toto_STOP_SEC_VAR_8BIT
APP_Task_toto_START_SEC_VAR_BOOLEAN	APP_Task_toto_STOP_SEC_VAR_BOOLEAN
APP_Task_toto_START_SEC_VAR_UNSPECIFIED	APP_Task_toto_STOP_SEC_VAR_UNSPECIFIED
APP_Task_toto_START_SEC_CONST_32BIT	APP_Task_toto_STOP_SEC_CONST_32BIT
APP_Task_toto_START_SEC_CONST_16BIT	APP_Task_toto_STOP_SEC_CONST_16BIT
APP_Task_toto_START_SEC_CONST_8BIT	APP_Task_toto_STOP_SEC_CONST_8BIT
APP_Task_toto_START_SEC_CONST_BOOLEAN	APP_Task_toto_STOP_SEC_CONST_BOOLEAN
APP_Task_toto_START_SEC_CONST_UNSPECIFIED	APP_Task_toto_STOP_SEC_CONST_UNSPECIFIED
APP_Task_toto_START_SEC_CALIB_32BIT	APP_Task_toto_STOP_SEC_CALIB_32BIT
APP_Task_toto_START_SEC_CALIB_16BIT	APP_Task_toto_STOP_SEC_CALIB_16BIT
APP_Task_toto_START_SEC_CALIB_8BIT	APP_Task_toto_STOP_SEC_CALIB_8BIT
APP_Task_toto_START_SEC_CALIB_BOOLEAN	APP_Task_toto_STOP_SEC_CALIB_BOOLEAN
APP_Task_toto_START_SEC_CALIB_UNSPECIFIED	APP_Task_toto_STOP_SEC_CALIB_UNSPECIFIED
APP_Task_toto_START_SEC_CARTO_32BIT	APP_Task_toto_STOP_SEC_CARTO_32BIT
APP_Task_toto_START_SEC_CARTO_16BIT	APP_Task_toto_STOP_SEC_CARTO_16BIT
APP_Task_toto_START_SEC_CARTO_8BIT	APP_Task_toto_STOP_SEC_CARTO_8BIT
APP_Task_toto_START_SEC_CARTO_BOOLEAN	APP_Task_toto_STOP_SEC_CARTO_BOOLEAN
APP_Task_toto_START_SEC_CARTO_UNSPECIFIED	APP_Task_toto_STOP_SEC_CARTO_UNSPECIFIED
APP_Task_toto_START_SEC_CONFIG_DATA_32BIT	APP_Task_toto_STOP_SEC_CONFIG_DATA_32BIT
APP_Task_toto_START_SEC_CONFIG_DATA_16BIT	APP_Task_toto_STOP_SEC_CONFIG_DATA_16BIT
APP_Task_toto_START_SEC_CONFIG_DATA_8BIT	APP_Task_toto_STOP_SEC_CONFIG_DATA_8BIT
APP_Task_toto_START_SEC_CONFIG_DATA_BOOLEAN	APP_Task_toto_STOP_SEC_CONFIG_DATA_BOOLEAN

CHAPTER

THIRTEEN

TRACING THE EXECUTION

Introduction

The code of Trampoline RTOS embeds a tracing subsystem that can be activated at system configuration time. This toolkit sends a stream of events describing the execution of the application to a target specific backend. The resulting data can then be exploited to compute statistics on some performance figures of the system (such as execution times, jitters, etc.) and/or to feed a visualization tool. Please notice that, in the current implementation, the tracing toolkit has a small, albeit non null, overhead so the system from which traces are computed is not exactly the same than the system without traces.

13.1 Traced events

Events that can be traced during an execution are given below. Each event is described by its name and a set of attributes. These attributes are made available by the kernel to the platform specific backend (see section 13.4.1 below).

PROC_CHANGE_STATE : state of a process (task or ISR) is changed.

- `proc_id`: identifier of the process.
- `target_state`: new state of the proc.

RES_CHANGE_STATE : state of a resource is changed.

- `res_id`: identifier of the resource.
- `target_state`: new state of the resource.

EVENT_SET : a process sets one or more event to another process.

- `proc_id`: owner of the events.
- `ev_id`: list of events that have been set.

EVENT_RESET : a process resets a subset of its events

- `ev_id`: list of events that have been reset.

TIMEOBJ_CHANGE_STATE : state of a timeobj (alarm, schedule table expiry point) is changed.

- `timeobj_id`: identifier of the timeobj.
- `target_state`: new state of the timeobj.

TIMEOBJ_EXPIRE : a timeobj expires.

- `alarm_id`: identifier of the timeobj.

Notice that each event contains the minimal information that is needed to rebuild the whole state of the system. Hence, the running task is never used as an attribute because it can be deduced by analysing the sequence of *PROC_CHANGE_STATE* events.

13.2 OIL declaration

13.2.1 Generic part

Activation of tracing is done at system configuration time through the OIL file. A boolean attribute `TRACE` is defined in the `OS` object. It has several subattributes has shown in the code below:

```
TRACE = TRUE {
    FORMAT = SERIAL;
    PROC = TRUE;
    RESOURCE = TRUE;
    ALARM = TRUE;
    EVENT = TRUE;
};
```

FORMAT specifies the output format of the trace. This is a target dependant attribute. For instance, the posix target allows the `json` output, while the msp430 allows the `serial` and `fram` targets (with sub-attributes). See section 13.2.2.

PROC controls the tracing of event `PROC_CHANGE_STATE`;

RESOURCE controls the tracing of event `RES_CHANGE_STATE`;

ALARM controls the tracing of events `TIMEOBJ_CHANGE_STATE` and `TIMEOBJ_EXPIRE`;

EVENT controls the tracing of events `EVENT_SET` and `EVENT_RESET`;

13.2.2 Target specific part

Posix target

The posix target is the simplest one, as there is no problem to get back the trace. The trace can be saved in `json` mode only. The generated file is always called '`trace.json`'.

MSP430 target

For the MSP430 format, different methods can be implemented to retrieve the trace:

`SERIAL` the serial communication, using the *serial over USB* link on launchpad dev board;

`FRAM` the FRAM storage. In that case, the trace is stored directly on the FRAM and can be read using a gdb dump. Sub-attributs are:

> **`SIZE`** the size of the dump in bytes;
>
> **`ADDRESS`** the base address.
>
> It should not be used if the throughput is too high! (not implemented yet).

`SD` the embedded SD card (not implemented yet).

In all cases, a trace event should take as low room as possible, and is stored in binary format (see section 13.4.2).

13.3 Using the tracing subsystem

The trace are generated by trampoline on the target. A difficulty resides in the ability to retrieve the information on the host system.

The trace system is split in 3 steps, as defined in Figure 13.1:

TraceReader reads *raw events* from the target. A *raw event* deals only with numerical ids from Trampoline. The reader can read events from the serial or from a file at this date. It can store them to a file for later evaluation.

TraceEvaluate associates a name with each ids, thanks to the '`tpl_static_info.json`' file generated by goil. It can make some extra evaluation on the application: there is never an event received to a suspended task, there is never critical section overlap due to bad resource usage, . . .

TraceExport prints the trace (text based or graphical).

For the serial line reader, the events are queued in a dedicated thread, so that there is no event loss, even if evaluation/export are too slow.

On the posix target, trampoline generates directly the `trace.json` file.

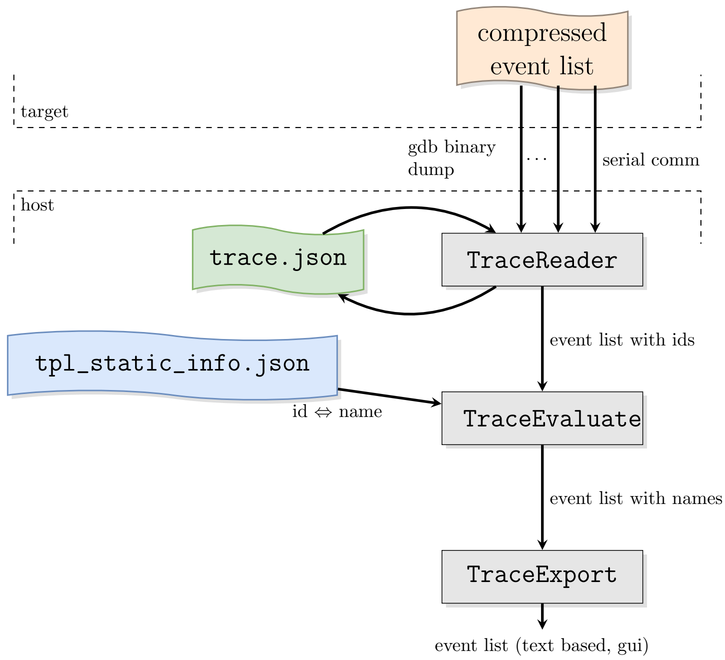


Figure 13.1: *Getting trace list from the target. The '`readTrace.py`' script calls the 3 steps of the trace : `TraceReader`, `TraceEvaluate` and `TraceExport`, for each event.*

Tools are provided in the '`extra/trace-tools/`' directory of Trampoline, and called directly by the generated script '`readTrace.py`'. Run to have information how to use the script:

```
./readTrace.py --help
```

For now, the script can read from a json file (posix or saved trace) or a serial line. It can save a received trace to a file for later use. The export is text based only.

When receiving an endless trace (serial line for instance), you can terminate the reception with `Ctrl+C`
.

13.4 Implementation

The implementation is split in 2 parts:

- the *generic* part is a set of macro in the '`os/`' kernel files to captures trace events. The file '`os/tpl_trace.h`' list all the specific functions that should be implemented.
- the *target specific* part in '`machine/.../tpl_trace.c`' implements the trace back-end.

13.4.1 Implementing target specific backends

The backend consists in a set of 7 functions that should be implemented in the target to store events, and communicate them to an host computer for analysis. Prototypes are in the `os/tpl_-`

`trace.h` are each function is related to a trace event. For instance, the following function will be called by the kernel internal files (*in kernel mode!*) each time a proc (task/isr) state is updated.

```
/**
 * trace the execution of a task or ISR
 * This function should be implemented
 * in the machine dependant trace backend.
 */
FUNC(void, OS_CODE) tpl_trace_proc_change_state(
    CONST(tpl_proc_id,AUTOMATIC) proc_id,
    CONST(tpl_proc_state,AUTOMATIC) target_state);
```

A new file is now generated by goil '`tpl_static_info.oil`' that lists the objects defined in the oil file. This file can be combined with the trace to deal with the name of an object instead of its internal id.

13.4.2 Binary format

The binary format is defined in Figure 13.2. It is composed of:

Type There are 6 events types: PROC_CHANGE_STATE, RES_CHANGE_STATE, EVENT_SET, EVENT_RESET, TIMEOBJ_CHANGE_STATE and TIMEOBJ_EXPIRE. One extra type *OVERFLOW* is defined if the communication medium cannot assure a sufficent bandwidth. 3 bits are required.

TimeStamp The time stamp. The overflow should be taken into account by the receiver (events are in chronological order). Most significant byte first.

xxx This is the event dependent data (5+8 bits).

Chksum The checksum (8 bits) is the sum of the 4 previous bytes. Its goal is both to detect errors and frame limits.

Type	xx	TimeStamp	xxx	Chksum
3	5	16	8	8

Figure 13.2: *binary format of a trace event (5 bytes)*

For each type, we define the specific bits:

Due to frame size limits, the number of events is limited to 31, the number of procs (task/isr) to 255, as well as time objects (alarms, schedule tables) and resources.

13.4.3 JSON format

The JSON format embedds a list of events (`trace.json` in Fig.13.1). This trace is generated and handled by the `TraceReader` script (in `extra/trace-tools/TraceReader.py`).

This file is generated from the binary format (see section 13.4.2), it contains raw information (using ids instead of names):

Type	format (5+8 bits)	fields
PROC_CHANGE_STATE	00SSS PPPPPPPP	S (3 bits): proc state
		P (8 bits): proc id
RES_CHANGE_STATE	00SSS RRRRRRRR	S (3 bits): res state
		R (8 bits): res id
EVENT_SET	EEEEE TTTTTTTT	E (5 bits): event id
		T (8 bits): task id
EVENT_RESET	EEEEE 00000000	E (5 bits): event id
TIMEOBJ_EXPIRE	00000 TTTTTTTT	R (8 bits): time obj id
TIMEOBJ_CHANGE_STATE	00SSS TTTTTTTT	S (3 bits): time obj. state
		T (8 bits): time obj id

Table 13.2: *Specific bits for each event type*

`ts` a timestamp (integer value). The overflow that may appear in binary format is not possible in the JSON format;

`type` The event type. Depending on the event, the sub-attributes (always integers) are given:

- **`proc`:** attributes `target_state` and `proc_id`
- **`set_event`:** attributes `event` and `target_task_id`.
- **`reset_event`:** attribute `event`
- **`timeobj`:** attributes `target_state` and `timeobj_id`
- **`timeobj_expire`:** attributes `timeobj_id`

All the attributes are the one manipulated internally by the kernel, and are used to get corresponding application names using the file `tpl_static_info.json` generated by goil.

Here is a (short) example of a JSON trace file:

```
[
  {
    "ts": 28600,
    "type": "timeobj_expire",
    "timeobj_id": 0
  },
  {
    "ts": 28600,
    "type": "proc",
    "target_state": 5,
    "proc_id": 0
  },
  {
    "ts": 28600,
    "type": "proc",
    "target_state": 1,
    "proc_id": 1
```

```
  },
  ...
]
```

The generation of a JSON file from binary events is implemented in function `decodeBinaryEvent` in script `extra/trace-tools/TraceReader.py`.

13.4.4 How to port `trace` to another target

The `tpl_trace.c` is target specific. It should be defined in the `machine` hierarchy. A good starting point is the msp430's version, with a SERIAL implementation. In the `avr/arduino` version, a C++ example is provided.

Then, the goil templates should be updated. This is done in 2 steps:

- Add the new file `tpl_trace.c` (or `.cpp`) to be compiled in the project: in `config/`*`target`*`/config.oil`

  ```
  PLATFORM_FILES targetTrace {
    PATH    = "target";        //path, starting from machine/
    CFILE   = "tpl_trace.c";   // file name (cpp allowed)
  };
  ```

- Add the definition of the trace type to be defined in the oil file: in section IMPLEMENTATION/OS in `config/`*`target`*`/config.oil`:

  ```
  /* trace */
  BOOLEAN [
    TRUE {
      ENUM [
        serial
      ] FORMAT = serial;
    },
    FALSE
  ] TRACE = FALSE;
  ```

In this example, only the serial type is allowed. You can have a look to the msp430 version, where other modes are available, and one have sub-attributes (for fram target).

That's all!

CHAPTER

FOURTEEN

DEBUGGING AN APPLICATION

Debugging an application requires examining the internal structures of Trampoline. The information contained in these structures can be used to find out which task is running, which tasks are ready, which resources are held, the status of alarms, etc. Finding one's way around these data structures can be difficult for a user.

As GDB is the most frequently used debugger, it is possible for GDB to generate a command file to simplify the examination of the internal structures of Trampoline.

14.1 Command generation

The OIL object `OS` has the boolean attribute `GDBCOMMANDS` which, when true, leads to the generation of a file named '`commands.gdb`' in the same directory as the OIL file. An optionnal sub-attribute, `PORT`, is used to specify the TCP/IP port on which the GDB server is listening and to generate the commands allowing GDB to connect to the GDB server, to load the program on the target and to set a breakpoint on the `main`. For example:

```
GDBCOMMANDS = TRUE
{
    PORT = 4242;
};
```

may be used on STMicroelectronics MCU (port 4242 is the default port of ST-LINK debugging system).

14.2 Examining the tasks

For each task declared in the OIL file, 2 commands named `b_<task name>` and `_<task name>` are generated. The first command sets a breakpoint on the task. The second command displays the name of the task, its identifier and its type (basic or extended) as well as:

- its state, SUSPENDED, READY, RUNNING or WAITING;
- its priority in the form <current priority>/<basic priority>;
- its activation count in the form <current activation>/<maximum activation>;
- its internal resource if it exists. For a non-preemptible task, INTERNAL_RES_SCHEDULER is displayed. If the task has no internal resource, NONE is displayed.
- a list of resources that the task holds. The list is displayed between a pair of square brackets from the most recently taken resource to the oldest taken resource. If no resource is held, only the pair of brackets is displayed.
- if the task is extended, the events it is waiting for and those it has received are displayed. If a numerical value is displayed, these are events that are not present in the application and are probably related to a programming error. When no events are displayed, <NONE> is displayed.

Suppose, for example, that the OIL file declares a task named blink as shown below.

```
TASK blink {
    PRIORITY = 1;
    AUTOSTART = FALSE;
    ACTIVATION = 1;
    SCHEDULE = FULL;
    RESOURCE = r1;
    RESOURCE = r2;
};
```

and that the C code of the task blink is the following:

```
1 TASK(blink)
2 {
3     GetResource(r1);
4     GetResource(r2);
5     ledToggle(GREEN);
6     ReleaseResource(r2);
7     ReleaseResource(r1);
8     TerminateTask();
9 }
```

and that a breakpoint has been set at the line 5. The command _blink will be generated and if invoked at the breakpoint it would display the following result:

```
(gdb) _blink
Task blink (id = 0, BASIC):
        state             = RUNNING
        priority          = 2/1
        activate_count    = 1/1
        internal_resource = NONE
        resources         = [ r2 r1 ]
```

⚠ If the command is performed before reaching `main`, i.e. possibly before the copy of the initialized variables has taken place, the variables may not be initialized yet and the state, the current priority or the current number of activations will be wrong. In addition, the pointer to the head of the list of resources held will be wrong, which may lead to an error message from GDB. If the task is in the `SUSPENDED` state, its priority is meaningless.

14.3 Examining the resources

For each resource, a command named `_<resource name>` is generated. This command displays the type (**`STANDARD`** or **`INTERNAL`**) of the resource, its name and identifier as well as for standard resources:

- its ceiling priority;
- the name of the owning process. If the resource is not held, `NONE` is displayed;
- if the resource is held, the previous priority of the process holding it is displayed.

For an internal resource the following information is displayed:

- its ceiling priority;
- if the resource is held;
- if the resource is held, the previous priority of the process holding it is displayed.

Let's continue with the previous example. The execution of the command `_r2` when the execution has reached the line 5 would display:

```
(gdb) _r2
Resource r2 (id = 0, STANDARD):
        ceiling priority    = 2
        owner               = blink
        owner prev priority = 2
```

After reaching the line 7, the execution of the command `_r2` would display:

```
(gdb) _r2
Resource r2 (id = 0, STANDARD):
        ceiling priority    = 2
        owner               = NONE
```

The following example shows the display of internal resource `oups` according to whether it is not held:

```
(gdb) _oups
Resource oups (INTERNAL):
        ceiling priority    = 4
        taken               = 0
```

or held:

```
(gdb) _oups
Resource oups (INTERNAL):
        ceiling priority    = 4
        taken               = 1
        owner prev priority = 3
```

14.4 Examining the alarms

For each alarm, a command `_<alarm name>` is generated. The command displays the alarm identifier as well as the following information:

- the counter to which it is linked with the current date of the counter in brackets;
- its state (`SLEEP`, `ACTIVE` or `AUTOSTART`[1]);
- if the alarm is `ACTIVE`, its expiry date;
- if the alarm is `ACTIVE` and cyclic, its cycle;
- finally its action.

The following output is an example of an alarm display:

```
(gdb) _blink_alarm
Alarm blink_alarm (id = 0):
        counter = SystemCounter(1461)
        state   = ACTIVE
        date    = 1561
        cycle   = 100
        action  = ActivateTask(blink)
```

14.5 Examining the counters

For each counter used in the application, a command `_<counter name>` is generated. The command displays the following information:

- the ticks per base of the counter, i.e. the number of ticks coming from the interrupt source and which are necessary to increase the counter value by 1;
- the maximum allowed value of the counter;
- the minimum cycle of the counter;
- the current number of ticks;
- the current date;
- the list of alarms that are currently scheduled by the counter. The first is the next alarm that will expire. Between the brackets, the date on which the alarm will expire is given.

Here is for example the display of the *`SystemCounter`*:

[1]This state is only possible before starting the OS.

```
(gdb) _SystemCounter
Counter SystemCounter:
        ticks per base    = 1
        max allowed value = 65535
        min cycle         = 1
        current tick      = 0
        current date      = 1224
        alarms            = [ blink_alarm(1324) ]
```

14.6 Examining the *tpl_kern* structure

The *tpl_kern* structure gathers several pieces of information: the identifier of the running process, the identifier of the process chosen by the scheduler to run, two pointers to the static and dynamic structures of the running process, two pointers to the static and dynamic structures of the process chosen by the scheduler to run, a boolean indicating that a rescheduling must be done, a boolean indicating if a context switch must be done and, finally, a boolean indicating that a context save must be done.

The `p_kernel` command displays the contents of the *tpl_kern* structure. An example of display, made during the step-by-step execution of the `blink` task, is shown below:

```
(gdb) p_kernel
tpl_kern:
        running       = blink
        elected       = blink
        need schedule = 1
        need switch   = 0
        need save     = 0
```

If *tpl_kern* is displayed when a process is running, the fields *running* and *elected* are always the same and the fields *need schedule*, *need switch* and *need save* have no meaning. Conversely, if *tpl_kern* is displayed while the kernel code is running, the fields *running* and *elected* may be different when the scheduler has executed and *need schedule*, *need switch* and *need save* reflect the decisions of the scheduler.

14.7 Examining the *tpl_ready_list* structure

The structure *tpl_ready_list* is a binary max heap. Each element has two fields, the dynamic process priority and the process identifier. The dynamic priority is obtained by concatenating the static priority and an order number per priority level. This order number starts at the maximum value and decreases with each activation of a process for the concerned priority level.

The `p_ready_list` command displays the *tpl_ready_list* structure as a tree giving the raw dynamic priority, the dynamic priority as a couple static priority / order number and the name of the process. A sample display is given below:

```
(gdb) p_ready_list
ready_list [6]:
  [45](2,13) read_button
```

```
[29](1,13) t2
  [15](0,15) IDLE
  [28](1,12) t3
[30](1,14) t1
  [27](1,11) t1
```

This corresponds to the tree shown in figure 14.1.

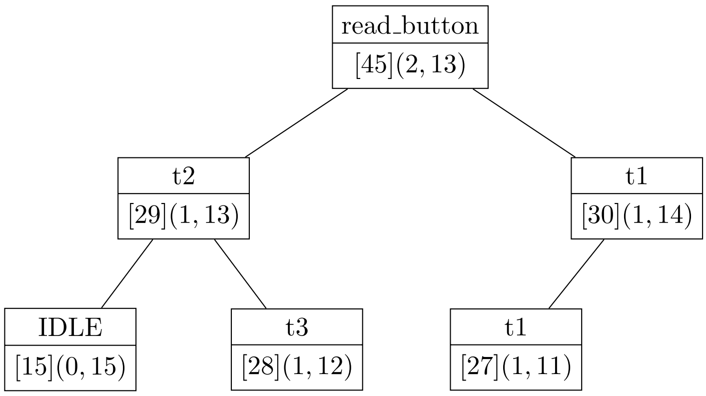


Figure 14.1: *Binary heap tree from the example of 14.7*

Part II

Trampoline RTOS internals

CHAPTER

FIFTEEN

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 15.1.

15.1 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
tpl_app_define.h	This file contains all the configuration macros (see section 15.2) and is included in all the Trampoline files to trigger conditional compilation. `goil` generates this file using the '`tpl_app_define_h.goilTemplate`' template file.
tpl_app_config.h	This file contains the declarations of the constants and functions required by the OSEK and Autosar standard (like OSMAXALLOWEDVALUE_x, OSTICKSPERBASE_x or OSMINCYCLE_x constants for counter x). `goil` generates this file using the '`tpl_app_config_h.goilTemplate`' template file.

tpl_app_config.c	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 `'tpl_app_config_c.goilTemplate'` template file.
tpl_app_custom_types.h	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 TaskType may be a byte if there is less than 256 tasks in the system and a word otherwise. This file defined these data types.
tpl_service_ids.h	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 `'tpl_service_ids_h.goilTemplate'` template file.
tpl_dispatch_table.c	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 `'tpl_dispatch_table_c.goilTemplate'` template file.
tpl_invoque.S	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 `'tpl_invoque.goilTemplate'` and `'service_call.goilTemplate'` template files.
MemMap.h	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 `'MemMap_h.goilTemplate'` template file.
Compiler.h	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 `'Compiler_h.goilTemplate'` template file.
Compiler_Cfg.h	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 `'Compiler_Cfg_h.goilTemplate'` template file.
script.ld	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 `'script.goilTemplate'` template file.

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

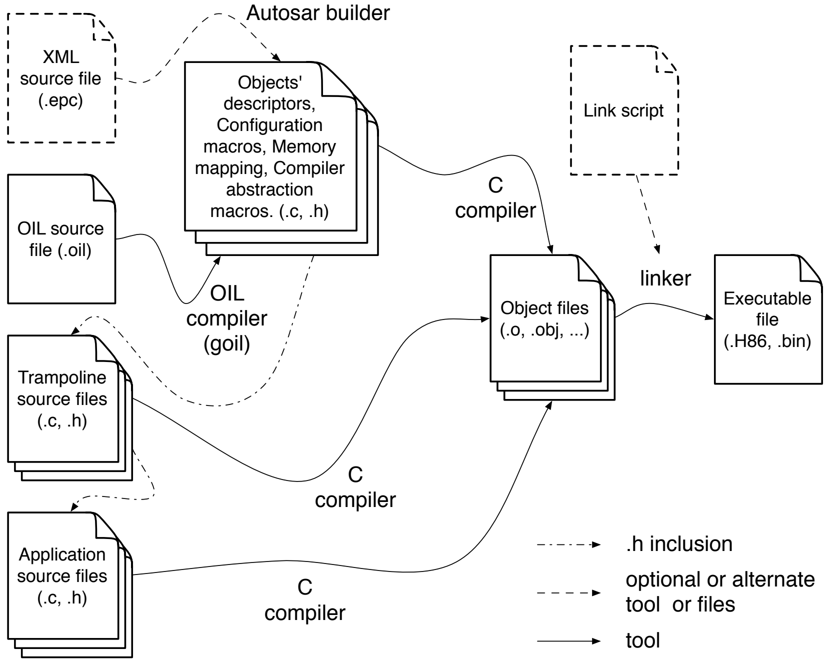


Figure 15.1: ***Build process of an application with Trampoline.*** *Starting from the left, the .c and .h corresponding to the application description given in OIL (or XML) are generated by* `goil` *(or another system generation tool, for instance an Autosar compliant one) and compiled using a C compiler. Trampoline source files are compiled too and include .h from the description for configuration purpose (see section 15.2). 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.*

15.2 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 `goil`using 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.

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.

15.2.1 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.

15.2.2 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 `ErrorHook()` function by using the `OSErrorGetServiceId()` 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 `ErrorHook()` 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 `COMErrorHook()` function by using the `COMErrorGetServiceId()` 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 `COMErrorHook()` 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.

15.2.3 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.

15.2.4 Hook call macros

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

Macro	Kind	Effect
`WITH_ERROR_HOOK`	Bool	see 15.2.2
`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 15.2.3

15.2.5 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 tpl_trace_format_<TRACE_FORMAT> 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.

15.3 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.

15.3.1 Counter related constants declaration

The 'tpl_app_config.h' files contains the counters related constants: those of the SystemCounter[1] 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 confguration. For the SystemCounter, the following constants are declared:

15.3.2 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.

[1] the default counter of an OSEK operating system

15.3.3 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;

#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.

15.3.4 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> mays 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,
#if 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>
#if 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.

CHAPTER

SIXTEEN

KERNEL IMPLEMENTATION

16.1 The *tpl_kern* structure

The *tpl_kern* structure gathers informations about the `RUNNING` process and flags to notify if a context switch and/or a context save are needed. It eases the access to these informations when programming in assembly language. The *tpl_kern* structure is an instance of the `tpl_kern_state` type:

```
typedef struct
{
  P2CONST(tpl_proc_static, TYPEDEF, OS_CONST) s_old;
  P2CONST(tpl_proc_static, TYPEDEF, OS_CONST) s_running;
  P2VAR(tpl_proc, TYPEDEF, OS_VAR)            old;
  P2VAR(tpl_proc, TYPEDEF, OS_VAR)            running;
  VAR(int, TYPEDEF)                           running_id;
  VAR(u8, TYPEDEF)                            need_switch;
} tpl_kern_state;
```

16.2 Ready list implementation

The implementation of the ready list makes it possible to reconcile relative simplicity with good performance regardless of the number of processes[1]. The ready list is implemented by an array indexed by priority. Each element of this array is a FIFO that stores the process identifier. An activated process is stored at the tail of the FIFO. A pre-empted process is stored at the head of the FIFO. Furthermore, in order to quickly find the non-empty FIFO corresponding to the highest priority, a binary heap is used to store the indexes (i.e. priority) of the non-empty FIFOs, the highest index being of course at the root of the heap.

[1] The term process here refers to a task or an SRI2

FIFO sizes are determined during the OIL compilation. Once the priorities assigned to processes and resources are determined, the FIFO size corresponding to a priority is the sum of the activations of each task, the number of resources and the number of ISR2 for this priority. Priority level 0 is only occupied by the task *idle*.

Let's take for example an application composed of 4 tasks and 2 resources, declared in OIL file as shown below.

```
RESOURCE r1 { RESOURCEPROPERTY = STANDARD; };
RESOURCE r2 { RESOURCEPROPERTY = STANDARD; };
TASK t1 { PRIORITY = 2; ACTIVATION = 2; RESOURCE = r1; };
TASK t2 { PRIORITY = 3; ACTIVATION = 1; };
TASK t3 { PRIORITY = 1; ACTIVATION = 3; RESOURCE = r2; };
TASK t4 { PRIORITY = 1; ACTIVATION = 2; RESOURCE = r1; RESOURCE = r2; };
```

The missing attributes for all tasks are `SCHEDULE = FULL` and `AUTOSTART = FALSE`.

The OIL compiler calculates the priority of the resource `r1`. As it is likely to be taken by the tasks `t1` and `t4`, its calculated priority is 3 (1 higher than the maximum priorities of `t1` and `t4`). Therefore the priority of `t2` is increased by 1 to allow it to pre-empt `t1` or `t4` when it holds the resource. The same is done for `r2`, which leads to give it the priority of 2. Consequently, the priorities of `t1`, `t2` and `r1` are increased by 1 to make room for `r2`. The occupancy of the priority levels and the corresponding size of the FIFO is therefore as shown in the table 16.1:

Priority level	Content	FIFO size
0	*idle*	1
1	`t3`, `t4`	5
2	`r2`	1
3	`t1`	2
4	`r1`	1
5	`t2`	1

Table 16.1: *Priority levels and FIFO size for the example.*

The data structure for this example is given in figure 16.1.

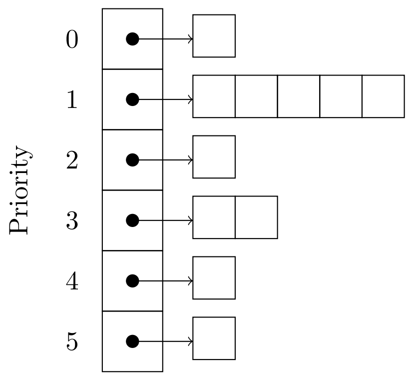


Figure 16.1: *Data structure for the example.*

CHAPTER

SEVENTEEN

PORTING TRAMPOLINE

In this chapter *arch* is used to designate the instruction set of the target like PowerPC®, ARM® or AVR®; *chip* is used to designate the name of an implementation of the architecture like a PowerPC 5516; *board* is used to designate the name of a development board that uses the chip. *compiler* is used to designate the compiler and *linker* is used to designate the linker used to link the project and produced the executable file.

17.1 Adding files to the directory structure

Doing a port of Trampoline on a new target requires

- data structures
- code, some is in C and some is in assembly language of the target
- code templates
- memory mapping templates (depend on the compiler)
- link scripts templates (depend on the linker)

Data structures declarations and code related to the instruction set are located in the `'machines/arch'` directory.

Code templates are located in the `'goil/templates/code/arch'` directory.

Memory mapping templates are located in the `'goil/templates/compiler/compiler/arch'` directory.

Link scripts templates are located in the `'goil/templates/linker/linker/arch'` directory.

For instance, if the goal is to port Trampoline to a Freescale® ColdFire® CPU with the gcc compiler and the gnu ld linker, you have to create a directory `'coldfire'` inside the `'machines'` directory, inside the `'goil/templates/code'` directory and inside the 'goil/templates/code/gnu_ld' directory.

In addition, some code or link scripts may be specific to the *chip* or the *board*. In this case, create sub-directories in the various *arch* directories using the pattern '*arch/chip/board*' to put the corresponding files.

17.2 Using a target with goil

The -t or --target option of goil selects the target by using a *arch/chip/board* path. Goil will look at the code, compiler and linker templates in the corresponding paths. Goil looks for a template at the deeper path first and goes up until it find it or gives an error when it does not find it. This way, a generic *chip* level template may be overridden by a more specific *board* level template for instance.

The link script templates (linker) and the memory mapping templates (compiler) are used only if a project is built using memory mapping. MEMMAP is a boolean attribute of the OS object in the OIL file. COMPILER and LINKER are sub-attributes of MEMMAP when it is TRUE. For instance, a MEMMAP using gcc and gnu ld would described like that:

```
MEMMAP = TRUE {
  COMPILER = gcc;
  LINKER = gnu_ld { SCRIPT = "script.ld"; };
  ...
};
```

Using this description and the target option, goil will look for link script templates in 'goil/templates/gnu_ld/*arch/chip/board*' path and for memory mapping templates in 'goil/templates/gcc/*arch/chip/board*' path.

The SCRIPT sub-attributes gives the name of the generated link script file.

17.3 Target specific code

⚠ The following informations require you use a software interrupt to call the system services.

This code should be located in the 'machines/*arch*' directory or in a sub-directory ('*chip*' or '*board*') if you want to implement a feature that rely on a specific chip or board (for instance to put peripheral devices in sleep mode in the tpl_sleep function). Anyway, you should put the relevant code at the corresponding level. If in the rare instances you may need to use conditional compiling, you may use the C macros TARGET_ARCH, TARGET_CHIP and TARGET_BOARD that contains the *arch*, *chip* and *board* respectively as character strings.

17.3.1 Functions called by Trampoline

The following functions are needed by Trampoline:

```
extern FUNC(void, OS_CODE) tpl_init_context(
  CONST(tpl_proc_id, AUTOMATIC) proc_id);
```

`tpl_init_context` may be written in C. It is called when an activated task runs for the first time. It initializes the context of the task by setting the 'at start' values of registers. Setting at least the values of the stack pointer at the beginning of the stack zone of the task and the return address at the entry point of the task code are required.

```
extern FUNC(void, OS_CODE) tpl_init_machine(void);
```

`tpl_init_machine` is called at the beginning of `StartOS` before calling the `StartupHook` and starting the scheduling. `tpl_init_machine` should do the hardware related initializations that are needed to run the OS (for instance starting the timer of the `SystemCounter`).

```
extern FUNC(void, OS_CODE) tpl_sleep(void);
```

`tpl_sleep` is called from the idle task. It should implement a loop around an instruction that put the CPU in a waiting for interrupt mode. If the *arch* does not have such an instruction, an empty loop may be used.

⚠ `tpl_sleep` should never return.

```
extern FUNC(void, OS_CODE) tpl_shutdown(void);
```

`tpl_shutdown` is called from `ShutdownOS`. It should disable all interrupts and put the CPU in sleep mode. If no sleep mode exists, an empty loop may be used.

⚠ `tpl_shutdown` should never return.

17.3.2 Service call

A service call is done by using a software interrupt[1]. So any function executed by the kernel as a result of API function call is handled by the software interrupt vector.

This code is called `tpl_sc_handler` and performs the following steps:

1. save registers to be able to work
2. disable memory protection
3. switche to kernel stack if needed
4. call the service
5. perform a context switch if needed and programs the MPU.
6. call kernel function `tpl_run_elected`
7. switche back to the process stack if needed
8. enable memory protection
9. restore registers saved at step 1
10. get back to the process

At step 4, the service identifier is used as an index in the function pointer table where all the services are stored. This table is also generated by goil (this allow to add services by your own and

[1] `swi` on ARM, `sc` on PowerPC, `syscall` on Tricore

customize Trampoline) and is called `tpl_dispatch_table`. The function pointer corresponding to the service is read from this table and the service is called.

The identifier of the service is passed to `tpl_sc_handler` in one of the following ways:

- the software interrupt instruction of the target has an argument, the identifier of the service is passed in this argument
- the software interrupt instruction of the target does not have an argument or the argument cannot store big enough value, the identifier of the service is passed in a register or on the stack

The way the PowerPC port manages the system call is explained in details in section 18.2.

17.3.3 Interrupt management

External interrupt handling should follow the same steps as service call when the ISR interacts with the kernel, activate a task or set an event and leads to a rescheduling. Of course, step 4 is a little bit different: instead of using a service id, the interrupt handler uses the interrupt source number. Usually the interrupt source number is got by reading a register of the interrupt controller.

goil provides a dispatch table for interrupts. This table is filled according to the `SOURCE` attribute of counters and ISR category 2. This attribute must be set to a symbolic name that is found in the '`target.cfg`' (located in '`goil/templates/config/`*`arch`*`/`*`chip`*`/`*`board`*' path). Each entry in the '`target.cfg`' file lists the correspondance between the interrupt source number and the symbolic name.

So at step 4, the interrupt handler uses the interrupt source number as an index in the *`tpl_it_table`*, get the corresponding interrupt handling function pointer and calls the function.

If interrupts are fully vectorized, i.e. each interrupt source has its own interrupt vector, goil should generate the code for each vector. See how it is done in '`cortex/armv7em`' port.

17.4 Target specific structures

A file named '`tpl_machine.h`' should exists in the '`machines/`*`arch`*' directory. This file should contain the declarations and definitions of:

- the `tpl_stack_word` type
- the `tpl_stack_size` type
- the `tpl_context` structure
- the `tpl_stack` structure
- the `IDLE_ENTRY` macro that should set to `tpl_sleep`
- the `IDLE_STACK` macro
- the `IDLE_CONTEXT` macro

The `tpl_stack_word` type is used to achieved a correct alignment of the stack

The `tpl_context` context structure contains one of more pointers to structures where all the registers needed for the execution context are stored. More than one pointer may be needed because on some architectures, contexts may be split in 2 or 3 parts to store the integer context, the floating point context and the vector context for instance. This way a task doing only integer computation needs the integer context only. The other pointers are set to NULL and the context switching code does not save or restore contexts for the NULL pointers. A `tpl_context` field is included in the static part of a task descriptor which may be stored in ROM. For instance, on an AVR, the context structure is declared as follow:

```
struct TPL_CONTEXT {
  avr_context *ic;
};
typedef struct TPL_CONTEXT tpl_context;
```

and an `avr_context` is defined as follow:

```
struct AVR_CONTEXT {
  u8 *sp;
  u8 regist[33];  // registers: R0-R15,R17-R31,SREG,R16
};
typedef struct AVR_CONTEXT avr_context;
```

The `tpl_stack` stack structure contains one or more pointers to the stack and one or more stack sizes. Some ABI may use more than one stack (an example is the Infineon C166). A `tpl_stack` field is included in the static part of a task descriptor. The AVR stack structure is as follow:

```
struct TPL_STACK {
  tpl_stack_word  *stack_zone;
  tpl_stack_size   stack_size;
};
typedef struct TPL_STACK tpl_stack;
```

The `IDLE_STACK` macro should expand to a `tpl_stack` initialization. This macro is used to initialize the stack in the idle task descriptor. For instance, the AVR `IDLE_STACK` and the component it uses are defined like this:

```
#define SIZE_OF_IDLE_STACK  50

extern VAR(tpl_stack_word, OS_VAR)
  idle_stack[SIZE_OF_IDLE_STACK/sizeof(tpl_stack_word)];

#define IDLE_STACK { idle_stack, SIZE_OF_IDLE_STACK }
```

The `IDLE_CONTEXT` should expand to a `tpl_context` initialization. This macro is used to initialize the context in the idle task descriptor. For instance, the AVR `IDLE_CONTEXT` and the component it uses are defined like this:

```
extern avr_context idle_task_context;

#define IDLE_CONTEXT {&idle_task_context}
```

17.5 Code templates

See chapter 19 for informations about the goil templates and the goil templates language.

Since service API functions perform a system call, they are to be written in assembly language. Instead of writting each of these functions by hand, they are generated by goil using 3 templates. 2 are generic, the 3rd one, '`service_call.goilTemplate`', is specific.

'`service_call.goilTemplate`' should be located in the '`goil/templates/code/`*`arch`*`/`' directory

For instance the ppc *arch* has the following template:

```
let api_func::FUNC_NAME := exists api_func::ACTUAL default ( api_func::NAME )
%
  .global % !api_func::FUNC_NAME %
% !api_func::FUNC_NAME %:
  /* load the service id in r0 */
  li    r0,% !api_sec::ID_PREFIX %ServiceId_% !api_func::NAME %
  sc     /* system call                */
  blr    /* returns                    */

  .type % !api_func::FUNC_NAME %,@function
  .size % !api_func::FUNC_NAME %,$-% !api_func::FUNC_NAME %
```

REAL and ***API*** are configuration data provided by goil. Both have a value equal to the name of the service (`ActivateTask` for instance). `StartOS` is a special case where ***API*** have the value `StartOS` and ***REAL*** have the value `tpl_start_os`. This is because `StartOS` is the only service that is called before the memory protection is turned on.

For `ActivateTask`, the template execution produces the following code:

```
  .global ActivateTask
ActivateTask:
  /* load the service id in r0 */
  li    r0,OSServiceId_ActivateTask
  sc     /* system call                */
  blr    /* returns                    */

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

17.6 Structures initialization templates

These templates are located in '`goil/templates/code/`*`arch`*'.

The template '`process_specific.goilTemplate`' is used to generate the instantiation of the context and the stack of a process (task or ISR category 2).

The template '`counter_call.goilTemplate`' is used to wrap a counter interrupt source to the Trampoline function that handle counter incrementation.

17.7 The memory mapping and the link script templates

Memory mapping is required with software interrupts because you have to put the interrupt vectors at the good place in memory. Moreover, when you use memory protection, goil generates memory sections for each task and ISR category 2.

The `'MemMap.h'` file that defines the sections is generated from the `'MemMap_h.goilTemplate'`. Files `'Compiler_h.goilTemplate'` and `'Compiler_Cfg_h.goilTemplate'` are used to generate the `'Compiler.h'` and `'Compiler_Cfg.h'` files which define the various AUTOSAR macros that assist to the specification of sections in the source files of Trampoline and of the application. These templates are found at the '`goil/templates/`*compiler*`/`*arch*`/`*chip*`/`*board*' path.

Usually these templates depend on the *compiler* only but, for instance, the Metrowerks® C compiler uses different #pragma according to the *arch*. So memory mapping templates for the Metrowerks C compiler for PowerPC would be located in '`goil/templates/compiler/mwc/powerpc`' and for HCS12 would be located in '`goil/templates/compiler/mwc/hcs12`'

To do that a link script template is used. This template is located in the '`goil/templates/linker/`*linker*`/`*arch*`/`*chip*`/`*board*' path.

The best way is to start with an existing template from a different target for the linker you use and to modify it.

CHAPTER

EIGHTEEN

PORTS DETAILS

18.1 Posix

18.1.1 Overview

18.1.2 Monocore

18.1.3 Multicore

18.2 PowerPC

18.2.1 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 15.1) 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 18.1.

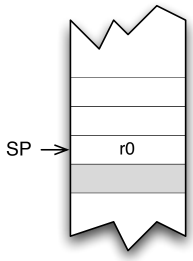


Figure 18.1: *Process stack mapping at the beginning of the System Call handler. The grayed zone represents an unknown content depending on from where the service was called.*

18.2.2 Dispatching the service call

The System Call handler is usually located in the `0C00`$_H$ 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:

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 [5] 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 18.2:

```
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              */
```

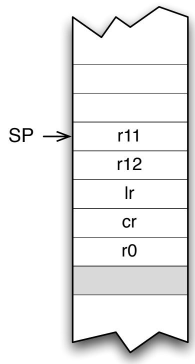


Figure 18.2: *Process stack mapping after additional registers have been saved by the beginning of the System Call handler.*

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
   */
  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 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 18.3. For a reentrant call, the same frame is build over the current one. The switch to the kernel stack is done as follow:

```
  /*
   * Check the reentrency 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
```

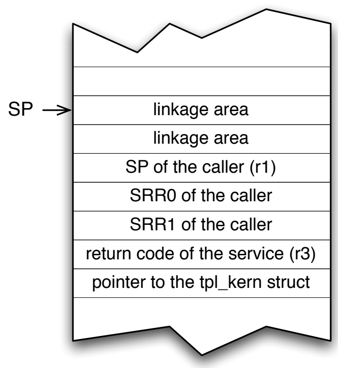


Figure 18.3: *Kernel stack mapping after allocation.*

Calling the service

Since the registers used to pass parameters to a function, that is *r3* to *r10* as documented in [5], 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 18.7.2 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 *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 *need_switch* flag that as 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 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 *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 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 get 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 get 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 an the stack is freed.

```
  lwz    r11,PS_CR(r1)
  mtcr   r11
  lwz    r11,PS_LR(r1)
  mtlr   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 */
```

18.2.3 Interrupt handler

18.2.4 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 explicitely 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 runnning 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 18.4. 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)
```

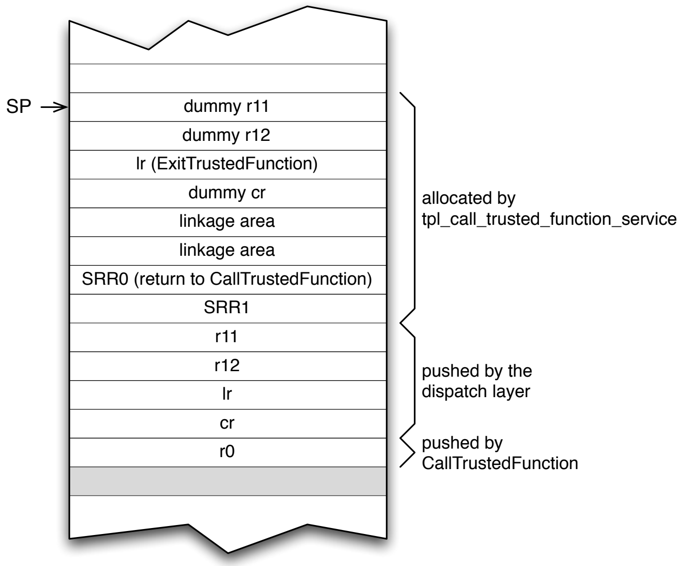


Figure 18.4: *Process stack mapping at the end of `tpl_call_trusted_function_service`. r0, at the bottom of the stack has been pushed by `CallTrustedFunction`. cr to r11 has been pushed by the dispatch layer. SRR0 and SRR1 are saved here by `tpl_call_trusted_function_service` to be able to go back to 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.*

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)
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:

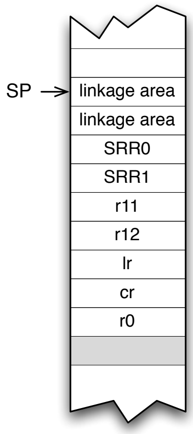


Figure 18.5: *Process stack mapping when the trusted function starts its execution.*

18.2.5 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 18.6.

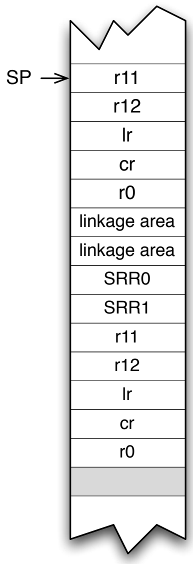


Figure 18.6: *Process stack mapping when the `tpl_exit_trusted_function_service` function starts its execution.*

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 runnning 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
```

18.2.6 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.

18.2.7 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 = rwx, 01 = rx, 10 = rw, 11 = *same as defined by* MxUM. 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 r, the second one w and the third one x.

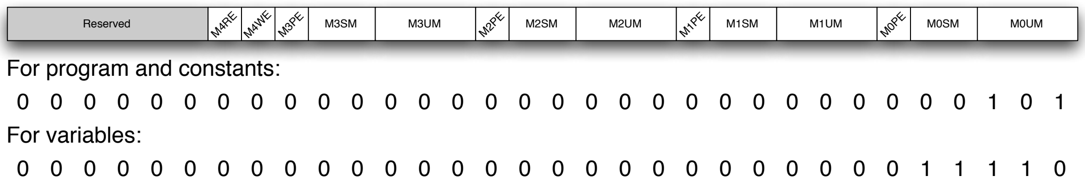
Trampoline uses 4 descriptors:

Descriptor	Usage	MxUM value
MPU_RGD0	Constants and program[1].	rwx = 00 for supervisor mode, rx = 101 for user mode.

[1] This region is set to the whole addressing space. This is not definitive and should be improved because reading

`MPU_RGD1`	Private variables of the process.	$rw = 110$ for supervisor and user mode.
`MPU_RGD2`	Stack of the process.	$rw = 110$ for supervisor and user mode.
`MPU_RGD3`	Variables of the OS Application if OS Applications are used.	$rw = 110$ for supervisor and user mode.

So values of access control bits should be:



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*.

18.3 ARM – Common conventions

18.3.1 File hierarchy

18.3.2 Common definitions

18.3.3 Bootstraping

The bootstrap must be made in specific ARM port and must call the `main` function. If `main` ever returns, the bootstrap code must fall into an infinite loop.

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).

As a reason, many ARM architectures needs early specific and required initializations. This includes steps like memory mapping configuration, DRAM controller configuration, ...

Besides specific initializations, the bootstrap should :

- initialize stack pointer for every ARM exception modes
- keep all external interrupts locked (will be unlocked at the first task context loading)
- call `main` in ”system” mode (0x1F)

18.3.4 Stacks

18.3.5 Interrupt management

Kernel is not interruptible. So hardware interrupt source are disabled entering in kernel (via any case in system call, interrupt request, abort, ...).

But kernel shall be reentrant via system call (because kernel hooks can call some system calls).

Interrupt and category classification

All ARM IRQ are category 2 ISR.

All ARM FIQ are category 1 ISR.

Vector table

Each ARM exception vector points on a so called ”primary” subprogram (like `tpl_primary_syscall_handler`).

To be located at address 0x00000000, this vector table is assigned to a specific section named `.vectbl`. The linker script uses this section name to output it to address 0x00000000.

System call

IRQ handling

FIQ handling

18.4 ARM – ARM926 chip support

18.4.1 Memory protection

To be written...

Some points to explain :

- FCSE mechanism is not used by this port (if someone is interested by this work, she's welcome)
- address translation is not used, all VMA equals physical address
- IDLE task's memory protection configuration is used to provide configuration for trusted applications or kernel

MMU tables generation principle

To be written...

Some points to explain :

- MMU is not disabled in privileged mode, but all useful memory areas are accessible. Thus, we hope we can find bugs easily in privileged code.
- useful memory areas, except processes and applications ones, are configured as accessible (read and write) in privileged mode. These memory areas are called system areas
- some memory areas needs to be accessible by anyone (API, GCC builtin functions, common libraries, ...), they are called common areas (they are read only for unprivileged contexts)
- there is one translation table for each process
- all translation tables have the same system and common areas
- there is one page table set for each process. Page tables are fine page tables. Table entries are tiny page descriptors.
- the number of page table in a set depends on the size of the whole trampoline and application memory footprint. Then this information is given by linker via a symbol which is used by the MMU driver.
- Page tables are accessed via a macro, as they are allocated by linker (and we cannot know the number of page tables)

18.4.2 CPU cache support

18.5 ARM – Armadeus APF27 board

18.5.1 Debugging with Abatron BDI2000 or BDI3000 JTAG probe

A configuration file is provided in '`machines/arm/arm926/armadeus-apf27/bdi-config`'.

To enable JTAG, if your APF27 has a FPGA, you must load the FPGA to wake it up (TO DO : explain how to do this...).

To start a debug session, follow these steps :

1. connects everything together

2. power up everything
3. reset the APF27 (S2 on APF27-Dev)
4. stop u-boot before it loads Linux (if MMU is started, you won't be able to load anything)
5. telnet your BDI
6. type `reset` command in the BDI shell
7. start GDB session (`target remote ...`)

18.5.2 Configuration

All configuration of port is done in '`apf27_config.h`'.

Stacks

Stacks' size (stack of each exception mode) can be adjusted via the following constants. Remember that the size must be aligned to 4.

CPU caches

By default, CPU caches are disabled (for real time determinism).

18.5.3 Memory mapping

This port can be use in one of these three configurations :

1. No memory mapping (and thus no memory protection)
2. Memory mapping without memory protection
3. Memory mapping and memory protection

18.5.4 Memory protection

Memory protection is based on ARM926 shared code (see **??** page **??**)

18.6 ARM – Simtec EB675001 board

18.6.1 Memory map and hardware resources

Talk about configured memory map (use of DRAM, where the bootstrap would be flashed, . . .).

Tell which hardware resources are used by the kernel.

18.6.2 Booting

There is two way to start Trampoline on APF27 :

- from ELF image (in file usually called '`trampoline`')
- from raw binary image (in file usually called '`trampoline.bin`')

Booting from ELF image

Load image with your ELF loader (the file is usually named '`trampoline`'). This can be GDB via a JTAG probe for example. Then, just start execution from `tpl_arm_bootstrap_entry` entry point. Here are commands you can type in GDB :

```
(gdb) load
(gdb) set $pc=tpl_arm_bootstrap_entry
(gdb) break main
(gdb) continue
```

Booting from raw binary image

Load image with your binary loader to 0xA0000000 memory address. Then just start execution at this point (0xA0000000).

With u-boot, you can type these commands :

```
BIOS> tftpboot 0xA0000000 192.168.5.20:trampoline.bin
BIOS> go 0xA0000000
```

18.6.3 Internal kernel drivers

18.6.4 Hardware interrupts handling

18.6.5 Idle task

18.6.6 Exceptions handling

18.6.7 Kernel sleep service

18.7 ARM - Cortex

18.7.1 Overview

The processor

The processor has two modes of execution depending on the kind of execution. The processor modes are:

1. Thread mode: Used to execute application software. The processor enters Thread mode when it comes out of reset. The CONTROL register controls whether software execution is privileged or unprivileged, see CONTROL register on page 24.
2. Handler mode: Used to handle exceptions. The processor returns to Thread mode when it has finished exception processing. Software execution is always privileged.

The privilege levels for software execution are:

1. Unprivileged: Unprivileged software executes at the unprivileged level and:
   (a) Has limited access to the MSR and MRS instructions, and cannot use the CPS instruction
   (b) Cannot access the system timer, NVIC, or system control block
   (c) Might have restricted access to memory or peripherals
   (d) Must use the SVC instruction to make a supervisor call to transfer control to privileged software
2. Privileged: Privileged software executes at the privileged level and can use all the instructions and has access to all resources. Can write to the CONTROL (nPRIV : bit 0) register to change the privilege level for software execution.

Core processor registers are depicted in figure 18.7.

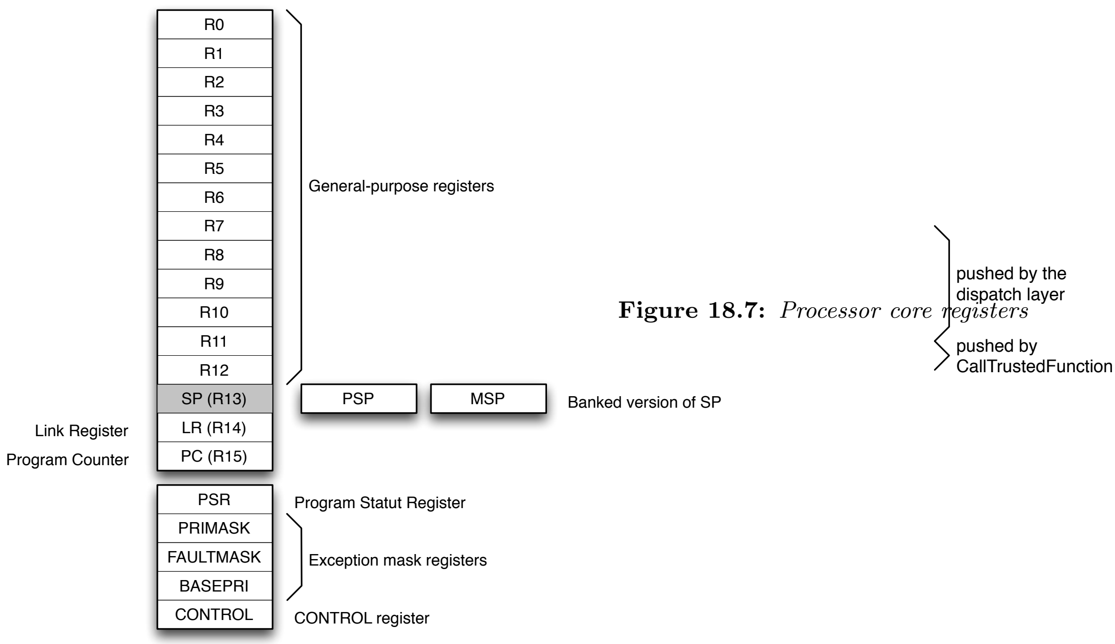


Figure 18.7: *Processor core registers*

The Program Status Register (PSR) combines:

1. Application Program Status Register (APSR)
2. Interrupt Program Status Register (IPSR)

3. Execution Program Status Register (EPSR)

These registers are mutually exclusive bitfields in the 32-bit PSR.

The stack

The processor uses a full descending stack. This means the stack pointer indicates the last stacked item on the stack memory. When the processor pushes a new item onto the stack, it decrements the stack pointer and then writes the item to the new memory location. The processor implements two stacks, the main stack and the process stack, with independent copies of the stack pointer. In Thread mode, the *CONTROL* register controls whether the processor uses the main stack or the process stack. In Handler mode, the processor always uses the main stack. The options for processor operations are:

Processor mode	Used to execute	Privilege level for execution	Stack used
Thread	Applications	Privileged or unprivileged	Main stack or Process stack
Handler	Exception handlers	Always privileged	Main stack

The port of Trampoline will use the two stacks and unprivileged level for software execution. This configuration is made in the function `tpl_init_machine_generic` of file '`tpl_machine_arm_generic.c`':

```
FUNC (void, OS_CODE) tpl_init_machine_specific (void)
{
  tpl_kernel_stack_top = (uint32)&tpl_kernel_stack[KERNEL_STACK_SIZE - 1];
  nested_kernel_entrance_counter = 0;
  __set_MSP(tpl_kernel_stack_top);
  setTimer();
  __set_CONTROL(0x3); /* Switch to use Process Stack, privileged state */
  __ISB(); /* Execute ISB after changing CONTROL register */
}
```

On reset the processor starts in Thread mode and uses the Main stack (*MSP*) for both Handler and Thread modes. We set the Process stack to the current Main stack address and switch to use Process stack for Thread mode. We then set the Main stack to a dedicated area pointed to by ptrMainStack.

The memory

The processor has a fixed memory map that provides up to 4 GB of addressable memory.

The exceptions

The Cortex-M4 processor supports interrupts and system exceptions. The processor and the Nested Vectored Interrupt Controller (NVIC) prioritize and handle all exceptions. The processor uses handler mode to handle all exceptions except for reset. The NVIC registers control interrupt handling and includes the following features:

1. 82 maskable interrupt channels for STM32F407xx (not including the 16 interrupt lines of CortexTM-M4 with FPU)
2. 16 programmable priority levels (4 bits of interrupt priority are used) All interrupts including the core exceptions are managed by the NVIC

Exception number	Priority	Type	Priority
1		Reset	-3 the highest
2	-14	NMI	-2
3	-13	Hard fault	-1
4	-12	Memory management fault	Configurable
5	-11	Bus fault	Configurable
6	-10	Usage fault	Configurable
7-10			
11	-5	SVCall	Configurable
12-13			
14	-2	PendSV	Configurable
15	-1	SysTick	Configurable
16-above	0 and above	Interrupt (IRQ)	Configurable

When an exception arises the processor saves a context state onto a stack pointed to by *sp* (either *MSP* or *PSP* depending on the mode of the processor at the time of the exception) and jumps to the Supervisor Call handler. The context state supports the *ARM Architecture Procedure Calling Standard* (AAPCS). When pushing context to the stack, the hardware saves eight 32-bit words, comprising xPSR, ReturnAddress, LR (R14), R12, R3, R2, R1, and R0. This behaviour is depicted figure 18.8.

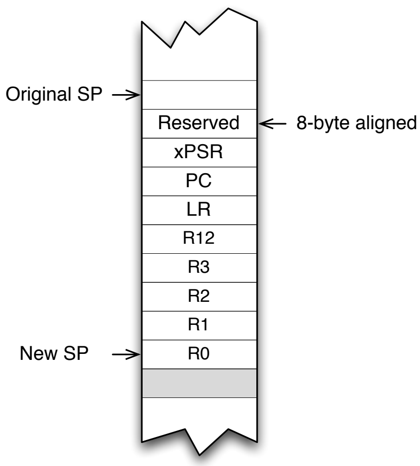


Figure 18.8: *Stacking frame upon exception rising*

18.7.2 System services

The Cortex-M4 port uses the `svc` (Supervisor Call) exception call to call system services [].

The id of the system service to call is given in the *r0* register and *r0* save and restore are added around. A system service may be directly called (without using the Supervisor Call) in case of nested calls. This is due to the architecture of the Cortex-m4. Other microcontrolers (e.g. Power PC) have features that enable nested system calls.

For instance, the following listing gives the `ActivateTask` service code. These functions are generated from templates by goil (see 15.1) and are part of the *invoque* layer (see **??**):

```
        /*
         * Service ActivateTask
         */
        .global ActivateTask
        .type   ActivateTask, %function
ActivateTask:
        /* manage reentrance of kernel */
        ldr r1, =nested_kernel_entrance_counter
        ldr r2, [r1]
        /* If nested_kernel_entrance_counter is greater or equal than 1 */
        cmp r2,#1
        /* then we are in Handler mode and we must call the service with a
         * direct call to the function */
        beq ActivateTask_direct_call
        /* Exception call to the service : use SVC exception */
ActivateTask_exception_call:
        mov r3,#OSServiceId_ActivateTask
        svc #OSServiceId_ActivateTask
        b ActivateTask_exit_call
        /* Procedural call to the service */
ActivateTask_direct_call:
        /* get the appropriate system call address into R3 */
        ldr r1, =tpl_dispatch_table
        mov r3, #OSServiceId_ActivateTask
        ldr r3, [r1, r3, LSL #2]
        push {lr}
        /* call the service  */
        blx r3
        pop {lr}
        /* Function call */
ActivateTask_exit_call:
        bx lr
```

The *ARM Architecture Procedure Calling Standard* (AAPCS) defines the following behaviour for subroutine call:

Register	**Synonym**	**Special**	**Role in the procedure call standard**
r15		PC	-3 The Program Counter
r14		LR	The Link Register
r13		SP	The Stack Pointer
r12		IP	The Intra-Procedure-call Scratch-register
r11	v8		Variable-register 8

r10	v7		Variable-register 7
r9		v6, SB, TR	Platform register. The meaning of this register is defined by the platform standard.
r8	v5	SVCall	Variable-register 5
r7	v4		Variable-register 4
r6	v3	PendSV	Variable-register 3
r5	v2	SysTick	Variable-register 2
r4	v1		Variable-register 1
r3	a4		Argument / Scratch-register 4
r2	a3		Argument / Scratch-register 3
r1	a2		Argument / Scratch-register 2
r0	a1		Argument / Scratch-register 1

Where the role of registers are:

1. **Scratch-register**, **Temporary-register** : A register used to hold an intermediate value during a calculation (usually, such values are not named in the program source and have a limited lifetime).
2. **Variable-register**, **V-register** : A register used to hold the value of a variable, usually one local to a routine, and often named in the source code.

18.7.3 Dispatching the service call

Raising the `svc` exception makes the processor change the stack pointer from the Process Stack Pointer to the Main Stack Pointer and then save a set of registers on top of this Main Stack. The kernel stack is the Main Stack and the process stack is the Process Stack.

The Cortex-M4 locates the Supervisor Call handler in the exception handler 11. 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_primary_syscall_handler`.

`tpl_primary_syscall_handler` performs the following tasks:

1. Prepare the environment
2. Saves additional registers to be able to work
3. Disables memory protection
4. Switches to kernel stack if needed
5. Calls the service
6. Performs a context switch if needed and programs the MPU.
7. Switches back to the process stack if needed
8. Enable memory protection
9. Restore registers
10. Get back to the process

Currently the Cortex-M4 port does not support tasks that use floating point registers

Preparing the environment

When the Supervisor Call begins execution, the kernel stack has the mapping depicted in figure 18.9.

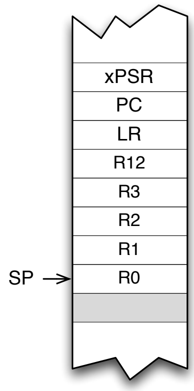


Figure 18.9: *Process stack mapping at the beginning of the Supervisor Call handler. The grayed zone represents an unknown content depending on from where the service was called.*

We will need space on top of current stack (Main stack or Kernel stack) *MSP* in order to save working registers. Some registers could be found in the frame saved by the processor, but we kept a common scheme for handling exception calls.

Saving additional registers

We save following values and registers on top of kernel stack :

- The value of caller *sp* :
- Register *lr* : This register is modified when calling subroutines. We save it to properly restore the caller process stack.
- Return code, the return code of service : The return code of the service is stored into *R0* register when returning from service. We save it into kernel stack as soon as the return of the service so that we can transmit it to the caller of service.
- A pointer to *tpl_kern*
- *r4* : Working register
- *r5* : Working register

Additional values saving is done by the following code at start of the `tpl_sc_handler` and the mapping of the kernel stack is depicted at figure 18.10:

```
sub sp, sp, #KS_FOOTPRINT /* Make space on top of kernel stack. */
mrs r12, psp /* Copy process stack pointer psp into r12 */
str r12, [sp, #KS_PROCESS_SP] /* and save it into kernel stack. */
str r4, [sp, #KS_R4]   /* Save working register r4 on process stack. */
str r5, [sp, #KS_R5] /* Save working register r5 on process stack. */
```

```
ldr r12, =tpl_kern
str r12, [sp, #KS_KERN_PTR] /* Store tpl_kern into kernel stack. */
str lr, [sp, #KS_LR] /* Store lr register into kernel stack. */
```

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 pushing the *lr* on the kernel stack, the `tpl_kernel_mp` function is called and does the actual job. At last *lr* is popped from the kernel stack.

```
#if WITH_MEMORY_PROTECTION == YES
  /*
   * Switch to kernel memory protection scheme
   */
  push {lr}
  bl tpl_kernel_mp
  pop {lr}
#endif
```

Switching to the kernel stack

The Cortex-m4 is configured to use two stacks and the Main Stack is the kernel stack.

The kernel stack may be already used 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 must be reentrant. The number of reentrant calls is counted by the *`tpl_reentrancy_counter`*. For a reentrant call, the same frame is build over the current one.

```
  /*
   * Manage reentrance of kernel
   * Increment nested_kernel_entrance_counter
  */
  ldr r12, =nested_kernel_entrance_counter
  ldr r4, [r12]
  add r4, r4, #1
  str r4, [r12]
```

Calling the service

Since the registers used to pass parameters to a function, that is *r3* to *r10* as documented in [**?**], 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 18.7.2 and is used as an index to the *`tpl_dispatch_table`*.

```
  /*
   * Get the appropriate system call address into R3
   */
  ldr r12, =tpl_dispatch_table
  ldr r3, [r12, r3, LSL #2]
```

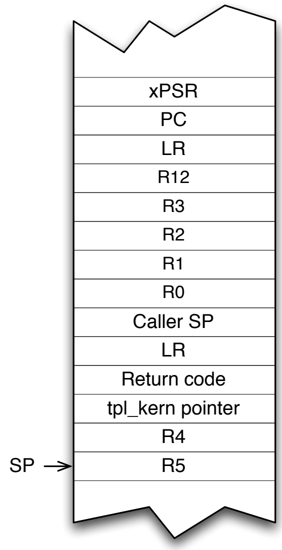


Figure 18.10: *Kernel stack mapping after saving additional registers.*

The second thing to do is to reset the ***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:

```
/*
 * Reset tpl_kern variables
 */
ldr r12, [r5, #KS_KERN_PTR] /* load the tpl_kern base address from kernel stack */
mov r4, #NO_NEED_SWITCH_NOR_SCHEDULE
strb r4, [r12, #TPL_KERN_OFFSET_NEED_SWITCH]
strb r4, [r12, #TPL_KERN_OFFSET_NEED_SCHEDULE]
```

At last, the service is called:

```
blx r3
```

And we then immediately save the return value of the service call into the kernel stack:

```
str r0, [r5, #KS_RETURN_CODE]
```

Context switch

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

```
ldr r12, [r5, #KS_KERN_PTR] /* load the tpl_kern base address */
ldrb r12, [r12, #TPL_KERN_OFFSET_NEED_SWITCH]
cmp r12, #NO_NEED_SWITCH
beq no_context_switch
```

A context switch is performed in 3 steps:

- Save the context of the task that looses the cpu.
- Load the configuration of memory protection for the process that gets the cpu.
- Load the context of the process that get the cpu.

Save the context of the task th at looses the cpu 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 *need_switch* flag. *s_old*, the address of the context saving area, is in another member of *tpl_kern*.

```
ldr r12, [r5, #KS_KERN_PTR] /* load the tpl_kern base address */
ldr r12, [r12, #TPL_KERN_OFFSET_S_RUNNING]
push {lr}
bl tpl_save_context
pop {lr}
```

Load the configuration of memory protection for the process that gets the cpu TODO (The second step consists in loading the configuration of memory protection for the process that get 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
```

)

Load the context of the process that get the cpu The third step calls the function `tpl_run_elected`. This function chooses the process that will get the cpu and write this information into the field *tpl_kern.s_running*.

Then we load the context of the process that got the CPU. The address of *tpl_kern* is loaded into *r12* 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 *r12* and `tpl_load_context` is called.

```
ldr r12, [r5, #KS_KERN_PTR]    /* load the tpl_kern base address */
ldr r12, [r12, #TPL_KERN_OFFSET_S_RUNNING] /* get the address of the context bloc */
push {lr}
bl tpl_load_context
pop {lr}
```

Switching back to the process stack

This is not useful for the Cortex-m4.

Leaving the kernel

Now we leave the kernel by calling the subroutine `tpl_leave_kernel`

```
  push {lr}
  bl tpl_leave_kernel
  pop {lr}
```

In this routine the reentrancy counter is decremented by 1.

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
  /*
  * Switch to user memory protection scheme
  */
  push {lr}
  bl tpl_user_mp
  pop {lr}
#endif
```

Restoring registers

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

```
  ldr r4, [sp, #KS_R4]
  ldr r5, [sp, #KS_R5]
  ldr lr, [sp, #KS_LR]

  ldr r12, [sp, #KS_PROCESS_SP]
  msr psp, r12
  add sp, sp, #KS_FOOTPRINT
```

Getting back to the process

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

```
tpl_sc_handler_exit:
  bx lr
```

===

18.7.4 Interrupt handler

18.7.5 The CallTrustedFunction service

18.7.6 The ExitTrustedFunction service

18.7.7 Execution of the OS Applications startup and shutdown hooks

18.7.8 Memory protection

The base address of a region must be aligned to an integer multiple value of the region size. For example, if the region size is 4KB (0x1000), the starting address must be $Nx0x1000$ where N is an integer.

Private Peripheral Bus (PPB) address ranges (including System Control Space, SCS) and the vector table don't need a memory region. Accesses to PPB (including MPU, NVIC, SysTick, ITM) are always allowed in privileged state, and vector fetches are always permitted by the MPU.

We need to define a handler for HardFault and MemManage (Memory Management) fault. The handler for HardFault is mandatory but not for MemManage unless we configure the bit MEMFAULTENA into register SCB-¿SHCSR. We chose to use MemManage fault.

18.7.9 Monocore

18.7.10 Multicore

18.8 AVR8

18.8.1 System services

The AVR architecture does not support the system call instructions, because there is no supervisor mode. However, the port works as if there were system calls. This allows to switch to a system stack at the beginning of a service call and preserve stack usage of user tasks.

Service calls are generated in the `tpl_invoque.S` file. As other port with system calls, the service id in stored in a table. The `tpl_sc_handler` is called like a function. This call does not respect the ABI in order to preserve registers used for parameters for the internal service code. For instance, the following listing gives the `ActivateTask` service code. These functions are generated from templates by `goil` (see 15.1) and are part of the *invoque* layer (see **??**):

```
  .global ActivateTask
ActivateTask:
  ldi r30,OSServiceId_ActivateTask  /* load the service id in r30 */
  call tpl_sc_handler
  ret
```

18.8.2 Dispatching the service call

`tpl_sc_handler` performs the following tasks:

1. save working registers
2. switch to kernel stack if needed
3. calls the service / counter call / ISR
4. performs a context switch if needed
5. switches back to the process stack if needed
6. restore working registers and get back to the process

Save working registers

Some working registers are saved on the user stack. TODO: use volatile registers...

Switching to the kernel stack

The first objective of the `tpl_sc_handler` is to switch to a kernel stack. However the kernel stack could 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`*.For a reentrant call, the same frame is build over the current one. The switch to the kernel stack is done as follow:

```
//tpl_reentrancy_counter++
lds r30,tpl_reentrancy_counter //load
subi r30, 0xFF //r30 <- R30-(-1)
sts tpl_reentrancy_counter,r30 //store
//tpl_reentrancy_counter == 1?
cpi r30,0x01 // compare with immediat
brne tpl_enter_kernel_end //branch if not equal
//yes => tpl_switch_to_kernel_stack
call tpl_switch_to_kernel_stack ;use r2-r6,r30-r31
```

When the `tpl_switch_to_kernel_stack` returns, SP points to the kernel stack and the stack is empty (figure 18.11

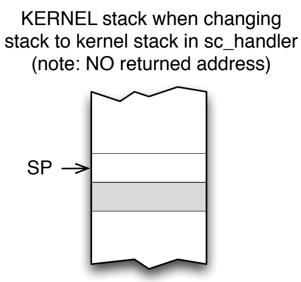


Figure 18.11: *Kernel stack just after the `tpl_switch_to_kernel_stack` function. The grayed zone represents stack bottom.*

Ce qui est fait: Remanier le fichier tpl_os.c pour: * le couper en 2 dans les templates * faire la partie en C directement. Mettre la partie commune de changement de pile directement dans machine/avr.

il faut rajouter en début de fichier tpl_os.c

```
#include <avr/io.h>
#include <avr/interrupt.h>
extern uint8_t tpl_reentrancy_counter;
extern void tpl_switch_to_kernel_stack();
extern void tpl_switch_to_user_stack();
```

18.8.3 Context

The context of the AVR is composed of:

- the stack pointer `sp`
- the program counter `PC`
- the status register `SREG` and
- 32 GPRs `r0` to `r31`. This includes the 3 16-bits indirect registers `X`, `Y` and `Z` which are mapped on these GPRs.

However, the context switch uses the stack to save all theses registers (except `sp` obviously) and the context does not use any structure to save only one variable (SP):

```
typedef u16 avr_context;
typedef avr_context *tpl_context;
```

Moreover, it is not necessary to save all registers. The ABI impose only to save `call-Saved Registers`[2]. Only `R2`-`R17`, `R28` and `R29` should be preserved. We also store the status register `SREG` to allow interrupts at the beginning of tasks (the `I` flag of `SREG` is set in `tpl_init_context`).

18.8.4 Context switch

The context switch implementation uses intensively the stack. The two functions for context switches point to the same code:

```
void tpl_switch_context(tpl_context *old, tpl_context *new);
void tpl_switch_context_from_it(tpl_context *old, tpl_context *new);
```

- save the current context (if `old` is not `NULL`);
- restore the context from `new`

registers are pushed on the stack like in figure 18.12. The `GPR r16` is used during the context switch and is the first on the stack. Then, the status register is saved, and all the remaining registers that should be preserved in the ABI. Note that we use the standard `gcc` frame in interrupts; this frame store all the remaining registers.

The restoration of the new context just gets the new stack pointer from the argument `tpl_-context` and pops all these registers.

[2]see more information on the AVR ABI at https://gcc.gnu.org/wiki/avr-gcc#Register_Layout

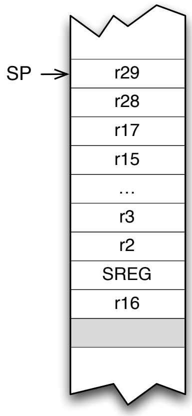


Figure 18.12: *Stack of the old_context at the end of context switch*

18.8.5 Context init

The initialization of the context init (`tpl_init_context`)the stack according to be compliant with the context switch.

⚠ All the AVR8 do not have the same size of the program counter!! Most of them use a 16-bit program counter, while a few ATMega use a 24-bit program counter!! (if there is more than 64ko of program flash).

The gcc compiler defines either the symbol `__AVR_3_BYTE_PC__` or `__AVR_2_BYTE_PC__`, which is used in the `tpl_machine.c` file. So, the AVR kind should be defined:

The stack at the end of context init should be like in figure 18.13:

- The `TerminateTask` or `CallTerminateISR2` is pushed, depending on the type of the process (task or ISR2)
- the PC of the entry point of the process is pushed
- the rest of the context is pushed. All required GPRs are init to `0x0`, and the status register `SREG` to `0x80` to enable interrupts (`I` bit).

18.8.6 Interrupts

Interrupts are handled directly using the standard way by the compiler `GCC`. `GCC` saves all the required registers, which are restored at the end of the interrupt.

It should be noted that `GCC` uses the `ISR` macro to define an interrupt handler, which is in conflict with the `ISR` macro defined in Trampoline.

```
//gcc uses ISR as a keyword to define an interrupt handler.
//Osek uses ISR to define an ISR2 :-/
#ifdef ISR
 #undef ISR
#endif
#include <avr/interrupt.h>
```

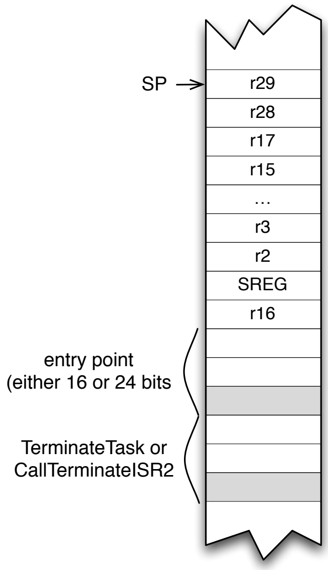


Figure 18.13: *Stack at the end of contact init*

```
ISR(TIMER2_OVF_vect)
{

  tpl_counter_tick(&SystemCounter_counter_desc);

  if (tpl_kern.need_schedule)
  {
    tpl_schedule_from_running();
    LOCAL_SWITCH_CONTEXT()
  }
}
```

18.9 Arduino Port

The Arduino port aims to use the Arduino libraries with trampoline on Arduino[3] AVR cards (first targets are Arduino Uno and Arduino Mega).

Arduino libraries have been set in the directory `machines/avr/arduino` and adapted to Trampoline. They are extracted from the GitHub version (see file machines/avr/arduino/version.txt). Current version is `1.5.8`.

Some adaptations on libraries should be done, and are explained in the next sections. For an easiest merge with next Arduino libraries, the code modification are well identified with comments:

```
// START TRAMPOLINE SECTION
        void trampolineSystemCounter();
```

[3] http://www.arduino.cc/

```
// STOP TRAMPOLINE SECTION
```

Code parts that are removed are also documented:

```
// START REMOVE TRAMPOLINE SECTION
        void trampolineSystemCounter();
// STOP REMOVE TRAMPOLINE SECTION
```

18.9.1 Main adaptation

In the Arduino approach, the `main.cpp` file is hidden, and 2 functions should be user defined:

- `setup()` initialize the system;
- `loop()` is repeated indefinitely;

With Trampoline, the `loop()` function disappears and the `StartOS()` service should be called at the end of the setup().

To be compliant with the Arduino approach, the `main.cpp` is hidden in user projects (but is present in `machines/avr/arduino/main.cpp`. It initializes timers, call the `setup()` user init function and start the OS with `StartOS()`.

Another service is provided to support different application modes[4]. The application mode should be defined *during the `setup()` function*:

```
void SetAppMode(AppModeType appMode);
```

18.9.2 Goil adaptation

A dedicated section for Arduino is provided in `CPU.OS`:

```
CPU test {
  OS config {
    ARDUINO = TRUE {
      BOARD = UNO;
      PORT = "/dev/tty.usbmodem1411";
      AVR_LIBC = "/usr/local/CrossPack-AVR/avr/include/";
      SERIAL = TRUE;
    };
    //...
  }
}
```

Parameters are:

BOARD The Arduino specific board. It should be only `UNO` or `MEGA` at this date;

PORT This is the device associated to the board to flash the AVR, like in the Arduino IDE. On Linux systems it should something like `/dev/ttyUSB0`.

[4] By default, the application mode `OSDEFAULTAPPMODE` is used

AVR_LIBC This is the place where is the `libc` for `avr-gcc`. This is required to get the `avr/io.h` include file. On Debian/Ubuntu, it should be located in `"/usr/lib/avr/include"`

feature Add the required files in the project. Current features[5] are:

- `SERIAL`

18.9.3 System Counter

The Arduino libraries comes with a SysTick associated to `TIMER0` interrupt. `Timer0` has a prescaler factor of 64, and the sysTick period is 1024μs on a 16MHz chip.

The `SystemCounter` counter is automatically defined[6] and connected to that Arduino `SysTick`, with:

- `TICKSPERBASE = 1`
- `MAXALLOWEDVALUE = 65535`
- `MINCYCLE = 1`

This means that the System Counter is hardwired to a 1024 μs period.

The period of this timer cannot be changed has it is used for both Arduino SysTick, and PWM. If you need another resolution, the best way is to use another timer associated to another OSEK counter.

[5] See up-to-date features in Goil specific templates in `<goilTemplatesDir>/config/avr/arduino/config.oil`
[6] see file `<goilTemplatesDir>/config/avr/arduino/config.oil`

Part III

The Goil system generator

CHAPTER

NINETEEN

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

19.1 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.

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

Theses variables are lists where informations about the processes[1] used in the application are stores:

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

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

Item	Type	Content
`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.
`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.

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

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

List	Content
`COUNTERS`	the list of counters, both hardware and software as long as the *`SystemCounter`*.
`HARDWARECOUNTERS`	the list of hardware counters including the *`SystemCounter`*.
`SOFTWARECOUNTERS`	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.
`SOURCE`	string	the interrupt source name of the counter. This is be used to wrap interrupt vector to a counter incrementation function.

19.1.3 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.

19.1.4 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.

19.1.5 The *REGULARRESOURCES* and *INTERNALRESOURCES* lists

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

List	Content
REGULARRESOURCES	the list of `STANDARD` and `LINKED` resources.
INTERNALRESOURCES	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.
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.

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

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

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

Item	Type	Content
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.
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.

19.1.7 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".

Item	Type	Content
`DATE`	integer	the start date of the schedule table. This attribute has an actuel 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.
`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.

19.1.8 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 is 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

Item	Type	Content
`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.
`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.

19.1.9 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.

19.1.10 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.

19.1.11 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.

19.1.12 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.

19.1.13 Scalar data

The following scalar data are defined:

Data	Type	Content
APPNAME	string	name of executable as given in the **APP_NAME** attribute in the OS object
ARCH	string	name of the architecture. This is the first item in the target.
ASSEMBLER	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.
ASSEMBLEREXE	string	name of the assembler executable used. This is the ASSEMBLER attribute in the OS object. It is set to *as* by default. It is used for build dependent templates.
AUTOSAR	boolean	true if Trampoline is compiled with the Autosar extension.
BOARD	string	name of the board. This is the third item (if any) in the target.
CHIP	string	name of the chip. This is the second item (if any) in the target.
COMPILER	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.
COMPILEREXE	string	name of the compiler executable used. This is the **COMPILER** attribute in the OS object. It is set to *gcc* by default. It is used for build dependent templates. Do not confuse with the **COMPILER** data.
CPUNAME	string	name given to the OIL CPU object
EXTENDED	boolean	true if Trampoline is compiled in extended error handling mode.
FILENAME	string	the name of the file which will be written as the result of the computation of the current template.

Data	Type	Content
FILEPATH	string	the full absolute path of the file which will be written as the result of the computation of the current template.
ITSOURCESLENGTH	integer	number of interrupt sources as defined in the `'target.cfg'` file.
LINKER	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.
LINKEREXE	string	name of the linker executable used. This is the **`LINKER`** attribute in the OS object. It is set to *gcc* by default. It is used for build dependent templates. Do not confuse with the **`LINKER`** data.
LINKSCRIPT	string	name of the link script file as given in the **`MEMMAP`** attribute of the OS object.
MAXTASKPRIORITY	integer	the highest computed priority among the tasks.
NATIVEFILEPATH	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.
OILFILENAME	string	name of the root OIL source file
PROJECT	string	name of the project. The name of the project is the `-p` (or `--project`) value if it is set or the name of the oil file without the extension.
SCALABILITYCLASS	integer	the Autosar scalability class used by the application. If Autosar is not enabled, SCALABILITYCLASS is set to 0.
TARGET	string	name of the target. This is the `-t` (or `--target`) option value of goil.
TEMPLATEPATH	string	path to the template root directory. This is the `--templates` option value of goil or the value of the GOIL_TEMPLATES environment variable.
TIMESTAMP	string	current date
TRAMPOLINEPATH	string	path to the trampoline root directory. This is the **`TRAMPOLINE_BASE_PATH`** attribute of the OS object. It defaults to "..".
USEBUILDFILE	boolean	true if a build file is used for the project ie option `-g` or `--generate-makefile` is given.
USECOM	boolean	true if the application uses OSEK COM.
USECOMPILERSETTINGS	boolean	true if memory mapping is enabled (Goil generates the `'Compiler.h'` and `'Compiler_Cfg.h'` files and Trampoline includes them).
USEERRORHOOK	boolean	true if Trampoline uses the Error Hook.
USEGETSERVICEID	boolean	true if Trampoline uses the service ids access macros.
USEINTERRUPTTABLE	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.
USELOGFILE	boolean	true if goil generates a log file, ie option `-l` or `--logfile` is given.
USEMEMORYMAPPING	boolean	true if memory mapping is enabled (Goil generates the `'MemMap.h'` file and Trampoline includes it).

Data	Type	Content
USEMEMORYPROTECTION	boolean	true if Trampoline uses the Memory Protection.
USEOSAPPLICATION	boolean	true if Trampoline uses OS Applications.
USEPARAMETERACCESS	boolean	true if Trampoline uses the parmaters access macros.
USEPOSTTASKHOOK	boolean	true if Trampoline uses the Post-Task Hook.
USEPRETASKHOOK	boolean	true if Trampoline uses the Pre-Task Hook.
USEPROTECTIONHOOK	boolean	true if Trampoline uses the Protection Hook.
USERESSCHEDULER	boolean	true if Trampoline uses the RES_SCHEDULER resource.
USESHUTDOWNHOOK	boolean	true if Trampoline uses the Shutdown Hook.
USESTACKMONITORING	boolean	true if Trampoline uses the Stack Monitoring.
USESTARTUPHOOK	boolean	true if Trampoline uses the Startup Hook.
USESYSTEMCALL	boolean	true if services are called using a System Call (i.e. a software interrupt).
USETIMINGPROTECTION	boolean	true if Trampoline uses Timing Protection.
USETRACE	boolean	true if tracing is enabled.

19.2 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 19.5.6) 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 '\'.

19.3 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 invokes 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
```

19.3.1 string readers

The following readers are available for string variables:

Item	Type	Meaning
`HTMLRepresentation`	string	this reader returns a representation of the string suitable for an HTML encoded representation. ‘`&`’ is encoded by `&` , ‘`"`’ by `"` , ‘`<`’ by `<` and ‘`>`’ by `>` .
`identifierRepresentation`	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 ‘`_`’ characters. This representation is unique: two different strings are transformed into different C identifiers. For example: `value3` is transformed to `value_33_`; `+=` is transformed to `_2B__3D_`; `An_Identifier` is transformed to `An_5F_Identifier`.
`lowercaseString`	string	this reader returns lowercased representation of the string.
`length`	integer	this reader returns the number of characters in the string
`stringByCapitalizingFirstCharacter`	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.
`uppercaseString`	string	this reader returns uppercased representation of the receiver

19.3.2 boolean readers

The following readers are available for boolean variables:

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

19.3.3 integer readers

The following readers are available for integer variables:

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

Item	Type	Meaning
`bitAtIndex`	boolean	this reader takes one *int* argument. It returns true if the bit at the index passed as argument is set and false if it is not set. For instance `let a := 3 let b := [a bitAtIndex: 0]` set `b` to true because bit 0 of `a` is 1
`setBitAtIndex`	integer	this reader takes two arguments. The first one, the value, is a *boolean* and the second one, the index, an *int*. it returns the integer value with the bit at the index passed as second argument set at the value passed as the first argument. For instance `let b := [1 setBitAtIndex: true, 4]` set `b` to 17

19.3.4 list readers

The following reader is available for list variables:

Item	Type	Meaning
`length`	integer	this reader returns the number of objects currently in the list.

19.4 GTL operators

19.4.1 Unary operators

Operator	Operand Type	Result Type	Meaning
+	integer	integer	no operation.
~	integer	integer	bitwise not.
not	boolean	boolean	boolean not.
exists	*any variable*	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.

19.4.2 Binary operators

Operator	Operands Type	Result Type	Meaning
`+`	integer	integer	add.
`-`	integer	integer	substract.
`*`	integer	integer	multiply.
`/`	integer	integer	divide.

Operator	Operands Type	Result Type	Meaning
`&`	integer	integer	bitwise and.
`&`	boolean	boolean	boolean and.
|	integer	integer	bitwise or.
|	boolean	boolean	boolean or.
$\wedge$	integer	integer	bitwise xor.
$\wedge$	boolean	boolean	boolean xor.
`.`	string	string	string concatenation.
<<	integer	integer	shift left.
>>	integer	integer	shift right.
`!=`	*any*	boolean	comparison (different).
`==`	*any*	boolean	comparison (equal).
<	integer *or* boolean	boolean	comparison (lower than).
<=	integer *or* boolean	boolean	comparison (lower or equal).
>	integer *or* boolean	boolean	comparison (greater).
>=	integer *or* boolean	boolean	comparison (greater or equal).

19.4.3 Constants

Constant	Type	Meaning
`emptyList`	list	this constant is an empty list
`true`	boolean	true boolean
`false`	boolean	false boolean
`yes`	boolean	true boolean
`no`	boolean	false boolean

19.5 GTL instructions

19.5.1 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 task in 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 and error.

19.5.2 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
```

19.5.3 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 var in 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
```

19.5.4 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.

19.5.5 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
```

19.5.6 The *!* instruction

! emits an expression. The form is:

```
! expression
```

19.5.7 The *?* instruction

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

```
? var
```

19.5.8 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*.goilTemplate 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.

19.5.9 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
```

19.5.10 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
```

19.6 Examples

Here are examples of code generation using GTL.

19.6.1 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
```

19.6.2 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
          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 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
```

19.6.3 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
```

INDEX

BIBLIOGRAPHY

[1] *Programming Environments Manual for 32-Bit Implementations of the PowerPC^{TM} 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] AUTOSAR. Specification of compiler abstraction. Technical report, AUTOSAR, October 2018.

[5] Microcontroller Applications IBM Microelectronics. Developing powerpc embedded application binary interface (eabi) compliant programs. Technical report, IBM, Research Triangle Park, NC, September 1998.

[6] Consortium OSEK/VDX. *OSEK/VDX Operating System*, 2.2.3 edition, 17th February 2005.

[7] 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.