# lab
# Adding A Service Call in Trampoline

M. Briday

January 18, 2019

## 1 Adding Semaphore Service Call into Trampoline

In this part we are going to see how to add semaphore services into Trampoline. We want two services : `SemWait` to lock a semaphore and `SemPost` to unlock a semaphore.

### 1.1 Adding a semaphore object

Let's start with the data.

**The first thing** to do is to provide an object type for semaphores in C. We need a `struct` with four members. `token` is the current number of token available. `size` is the number of tasks waiting in `waiting_tasks`. `index` is write index in `waiting_tasks`. `waiting_tasks` is a ring buffer to store the tasks waiting for the semaphore.

```
1 typedef struct {
2   uint32_t      token;
3   uint32_t      size;
4   uint32_t      index;
5   tpl_task_id waiting_tasks[TASK_COUNT];
6 } tpl_semaphore;
```

`TASK_COUNT` is computed by goil and is equal to the number of tasks in the application.

**The second thing** is to provide an object type for semaphores in the OIL description. This is done in the `IMPLEMENTATION` part of the OIL description. Normally the OIL standard does not allow to create new object types. This has been relaxed in goil.

```
1 SEMAPHORE [] {
2   UINT32 TOKEN;
3 };
```

This declares a new object type : `SEMAPHORE` with an `UINT32` attribute, `TOKEN`. The `[]` means multiple instances of semaphore can be used. Now, semaphores can be declared in the OIL file:

```
1 SEMAPHORE sem1 { TOKEN = 3; };
2 SEMAPHORE sem2 { TOKEN = 1; };
```

**The third thing** is to write a template that will generate C source code with a `tpl_semaphore` instance for each `SEMAPHORE` object in the OIL file. To do that, we have to provide a template directory hierarchy as in `goilv2/templates`. This hierarchy is put in the directory where the application source files are.

Since we generate code, the hierarchy is `goilv2/code`. Our code will be embedded in the `tpl_app_config.c` and `tpl_app_define.h`.

The templates are `custom_app_config.goilTemplate` and `custom_app_define.goilTemplate` respectively.

In the first one we have to:

1. generate semaphore object identifiers. A semaphore object identifier has the `SemType` type;
2. generate semaphore objects. A semaphore object has the `tpl_semaphore` type;
3. generate a semaphore table indexed by semaphore object identifiers. Each element of this table is a pointer to the corresponding semaphore object.

### 1.2 Semaphore services

Services are described in configuration OIL files. To add a service, we must provide a description of the service. This can be done in the OIL file (inside CPU section) of the application, or in `config/api.oil` in the template directory:

```
1  APICONFIG semaphore {
2    ID_PREFIX = OS;
3    DIRECTORY = "os";
4    FILE = "tpl_os_semaphore_kernel";
5    HEADER = "tpl_os_semaphore";
6    SYSCALL SemWait {
7      KERNEL = tpl_sem_wait_service;
8    LOCK_KERNEL = TRUE;
9    CALLABLE_BY_ISR1 = FALSE;
10     RETURN_TYPE = StatusType;
11     ARGUMENT sem_id { KIND = CONST; TYPE = SemType; };
12   };
```

```
13    SYSCALL SemPost {
14      KERNEL = tpl_sem_post_service;
15    LOCK_KERNEL = TRUE;
16    CALLABLE_BY_ISR1 = FALSE;
17      RETURN_TYPE = StatusType;
18      ARGUMENT sem_id { KIND = CONST; TYPE = SemType; };
19    };
20  };
```

`APICONFIG` is the root object to define a set of services related to a new object. Here we define an `APICONFIG` for semaphores. goil generates identifiers for services. Identifiers are prefixed by a section name. For instance, operating system services are prefixed by OS and communication services are prefixed by COM. Here we choose to use the OS prefix: `ID_PREFIX = OS;`

The `FILE` attribute allows to list the files where the C kernel function are defined. As many files as needed may be listed. The `HEADER` attribute allows to list the files where the datatypes and constants are declared. As many files as needed may be listed.

The `SYSCALL` attribute is used to define a service. The name, here `SemWait` and `SemPost`, is the service name as seen by the application.

- `KERNEL` is the corresponding kernel function.
- `LOCK_KERNEL` is required to remove interrupts so that the kernel can safely update internal structures.
- `CALLABLE_BY_ISR` is a parameter for few system services that can be called by ISR1 (interrupt handling), without any `SVC` access.
- `RETURN_TYPE` is the type of variable returned by the service
- `ARGUMENT` is the name, type and kind of argument. As many arguments as needed may be listed (almost).

The corresponding C source code must be provided in files `tpl_os_semaphore_kernel.h`, `tpl_os_semaphore_kernel.c` and `tpl_os_semaphore.h`.

At last, the template `api.goilTemplate` is modified to add the following template code:

```
1  if exists SEMAPHORE then
2    if [SEMAPHORE length] > 0 then
3      let APIUSED += APIMAP["semaphore"]
4    end if
5  end if
```

### 1.3 Static data structure

The static data structure for the semaphores is generated from the `.oil` input file. The `goil` compiler has to generate the static object `tpl_semaphore` for each semphore, a unique identifer (`semType`), as well as a tabular that refers to all sempahore structures.

To do so, goil is based on templates for the code (in `templates/code`), and we add the files `custom_app_config_c.goilTemplate` and `custom_app_define_h.goilTemplate`.

The goil language is based on templates, and the following code (extracted from ) will generate the structure:

```
foreach sem in SEMAPHORE do
%
tpl_semaphore %! sem::NAME%_sem = {
  %! sem::TOKEN%,
  0,
  0,
  { 0 }
};
%
end foreach
```

The `%` character is used to switch between *commands mode* (foreach, ..) and *text mode* (the code that is written). The `!` command allows to print a varaible to the output.

The output here may be:

```
1  tpl_semaphore sem1_sem = {
2    3,
3    0,
4    0,
5    { 0 }
6  };
7
8  tpl_semaphore sem2_sem = {
9    1,
10   0,
11   0,
12   { 0 }
13 };
```

## 1.4 System service

The goal of the lab is to write the system service related to the 2 system calls `SemWait` to lock a semaphore and `SemPost` to unlock a semaphore. These 2 services should be written directly in the file `trampoline/os/tpl_os_semaphore_kernel.c`.

All the services in OSEK/AUTOSAR return a status value, which is `E_OK` in case of success, or an error identifier if the service is not called correctly (for instance, the `TerminateTask()` called from an interrupt...).

The information that will be useful:

- the semaphore structure from the semaphore id in argument can be retrived using the `tpl_sem_table`
- the `tpl_kern` structure (`tpl_os_kernel.c`) defines the running task (static & dynamic structures, id)
- the `tpl_block()` (`tpl_os_kernel.c`) internal function transfers the `RUNNING` task to the `WAITING` state;
- the `tpl_release(taskId)` (`tpl_os_kernel.c`) internal function transfers task `taskId` from the `WAITING` state to the `READY` state;
- the `tpl_schedule_from_running(coreId)` function calls the scheduler.

### 1.4.1 SemWait service

This service should not be called if the task has an activation max value higher than 1. In that case, the service should return `E_OS_ACCESS` and have no effect.

When the service is called:

- if there are at least one token, the number of token is decremented and the service call ends
- if there are no more tokens, the running task should be blocked, and the task id should be saved (to release the task in the `SemPost` system call).

The task id should be saved using a ring buffer (using a tabular and the `index` structure member so that task a blocked in a FIFO list.

**Question 1** *Implement the `SemWait()` service*

### 1.4.2 SemPost service

This service always returns `E_OK` (no restriction).

When the service is called:

- if there is at least one task that is blocked by the semaphore, then the oldest task is released (FIFO), and a reschedule is done (as we update the ready list).
- if no task is bloked, then the number of token is incremented.

**Question 2** *Implement the* `SemPost()` *service*

### 1.4.3 test application

The provided application is based on a 2producers/2consumers model, using a buffer. The push buttons 4 to 7 can be used to activate a corresponding Task.

**Question 3** *The buffer should be protected with 2 semaphores (overflow/underflow). Update the provided application to test the semaphores.*

**Question 4** *use the application to validate your semaphore implementation. You can use the tft display to print information, or use directly the debugger. Explain your tests scenarios.*