TiCOS User Guide

1. What is TiCOS

TiCOS is a time-composable real-time operating system developed within the EU FP7 PROARTIS project [*http://www.proartis-project.eu/, visited on October 2012*] that supports the ARINC653 industrial standard.

TiCOS originated from POK [*http://pok.safety-critical.net/, visited on October 2012*], an open-source implementation of a real-time kernel. TiCOS includes and re-implements some parts of POK with a view to timing composability between OS and application layers.

In order to make the users not only able to install and run TiCOS, but also to understand and take fully benefit from its features, we start providing a very quick overview of the TiCOS architecture; next we move forward providing instruction about TiCOS installation and usage and user application configuration. Finally, we provides some hints about the compilation process in case you already have your own application and want to run it on TiCOS.

2. TiCOS architecture

In this section, we refer to ARINC-specific concepts such as processes, partitions, scheduling slots, etc; readers that are not familiar with these notions can refer to [*APEX Working Group. Draft 3 of Supplement 1 to ARINC Specification 653: Avionics Application Software Standard Interface. 2003*].

The TiCOS architecture is made up of two layers: the kernel layer and the partition layer.

The kernel layer implements a set of standard libraries (e.g., C standard library) and core OS services (e.g., scheduling primitives). In addition, the TiCOS kernel offers support for the implementation of a subset of the ARINC APEX.

The partition layer is made up of three parts: the user code, the ARINC layer, and the middleware layer. The ARINC layer implements the ARINC services via a system call mechanism (often mediated by a middleware-layer redirection) that performs the switch to the supervisor mode and executes the required TiCOS kernel services and primitives.

In the following sections a very basic description of the two layers is provided. Readers interested to implementation details can refer to the document: “TiCOS: a Time-Composable OS”.

2.1. TiCOS Kernel layer

The main services provided by the kernel are the following.

Partitions support. TiCOS provides a time and space isolation among partitions. To this end, every partition (code and data) has its own memory space that do not overlap with memory space of the other partitions.

Scheduling. The TiCOS scheduler first selects the partition according to a cycling scheduling policy, then selects the thread of the current partition according to a fix priority policy.

Lock objects management. In TiCOS lock objects are used by the upper layer to implement middleware events and thus providing synchronization among buffers and blackboards. In fact, every buffer and blackboard is associated to a middleware event that in turn is associated to a lock object. Each lock object memorizes which threads (if any), in a given moment, are locked on it; at the moment no priority policy is used to unlock threads locked on a lock object. In the original POK, lock objects are also used to provide mutual exclusion. However, since in TiCOS we assume a run-to-completion semantics, we do not need to worry about mutual exclusion. The reason is that no process can be preempted while holding a lock so long as every job releases any lock that it has previously acquired.

Ports support. TiCOS kernel provides supports for the implementation of sampling and queuing ARINC 653 ports. In our implementation, operations on ports are always no blocking, so we do not need to worry about ports synchronization. However, as sampling and queuing ports represent an inter-partition communication mechanism, and considering that partitions cannot have shared memory (due to the space isolation requirement), kernel is in charge to copy messages from source ports to destination ports.

2.2. ARINC and Middleware layer

The goal of the ARINC layer is to provide an implementation of the ARINC API services related to the management of:

- processes;
- partitions;
- sampling ports;
- queueing ports;
- blackboards;
- buffers;
- events;
- errors.

From the implementation point of view, between the ARINC layer and the kernel layer, there is a middleware layer that executes TiCOS kernel services and primitives via a system call mechanism. In other words, the ARINC API services in most cases simply wrap the services provided at the middleware layer.

For the ARINC APIs signatures and semantics we referred to the ARINC653 APIs specifications. However, a fully compliant implementation of all these services was out of our scope. For this reason, as detailed in the next subsections, we provided a simplified implementation only. For every service group, we also provide the guideline for configuring an application that makes use of them.

2.2.1. Processes and partitions

An ARINC application is made up of partitions and threads. Partitions are executed periodically according to their own period. Every partition has a main thread that executes only once and that creates the other partitions threads. Threads inside a partition are scheduled according to a fixed-priority policy. TiCOS actually provides a time-composable scheduler that we named O(1) scheduler.

2.2.2. Blackboards

Blackboards represent an intra-partition communication mechanism, that is a mechanism that allows threads of the same partition to exchange messages. Since a blackboard can contain only one message at a time, if a thread displays a message in a no-empty blackboard the new message overwrite the previous

one. Blackboards use events to implement synchronisation among threads that use the same blackboard. To this end, when a blackboard is created an event is associated to it. When a thread tries to read an empty blackboard, this event will be set to “down” and the thread is suspended until the event will became “up”; similarly, when a thread displays a message in a blackboard, the event is set to “up” and all threads waiting for that event are resumed (i.e. they became runnable).

2.2.3. Buffers

Buffers represent another communication mechanism among thread of the same partitions; they show three differences with respect to blackboards: (i) they can contains more than one message; (ii) threads are not allow to send message to a full buffer; (iii) when a thread receives a message from a buffer the message is removed from the buffer itself. As blackboards, also buffers use events to implement synchronisation among threads that send/receive messages to/from the same buffer. In this case, however, the event has a double role, as it is used to perform two different synchronizations. The first one is similar to the one described for blackboards: the event will be set to “down” when a thread tries to receive a message from and empty buffer while will be set to “up” when a message is sent to the buffer. The second synchronization is the following: the event will be set to “down” when a thread tries to send a message to a full buffer, while will be set to “up” when a thread receives a message and removes it from the buffer, making room for a new message.

2.2.4. Sampling and queueing ports

At the middleware layer, sampling and queuing ports functions are wrappers that invoke, via system calls mechanisms, the corresponding kernel level services.

2.2.5. Events

Events are implemented using kernel lock objects, that is to say that every middleware event is mapped on a kernel lock object. It is worth noticing that every event id (at the middleware layer) represents the id of the lock object associated to that event.

2.2.6. Semaphores

TiCOS provides dummy stubs for semaphores; in fact, under the run-to-completion semantics, as we have already claimed, we do not need to worry about mutual exclusion. Hence, there is no point in using the ARINC SEMAPHORE primitives as a semaphore lock is always available by construction. We currently provide a void implementation of SEMAPHORE services (simply return a NO_ERROR value) so that legacy applications that make use of the SEMAPHORE API will still be able to run on top of POK.

2.2.7. Error handling

The only error management function implemented at the middleware layer is the raise application error function.

3. Installation and usage

3.1. Supported platform

Currently you can run TiCOS only on the PROARTIS simulator or on QEMU (v. 1.1.0 beBox). PROARTIS sim is a PPC simulator implemented by Barcelona Supercomputing Center (BSC) and distributed in its binary form along with the TiCOS. Since it is a cycle-accurate simulator, it is the recommended choice in case you want to experiment the TiCOS features related to the constant-time functions.

3.1.1. PROARTIS simulator

In this following list we briefly present the main features of PROARTIS sim, paying special attention to the differences from the real ppc750 hardware platform.

- PROARTIS_sim uses the ppc405 module of SoCLib (*http://www.soclib.fr/, visited on October 2012*) appropriately modified in order to resemble a ppc750 core. For this reason it has been added: floating point Unit (FPU), Decrementer register (DEC), single level Hardware page-walker. However, it does noy mean that it implements all the functionality of ppc750.

- All 64-bit reads and writes from and to fp registers are performed in 32-bit chunks. This adds an extra cycle for these accesses.

- Physical memory range: 0x00000000-0x02000000.

- Simulation ends with a write of the program's return value to 0xD0800004. This can be used to simulate either the return of main function or the return value of exit().

- Uncacheable accesses are treated as misses.

- Simple in-order 4-stage pipeline without dependency checking.

- Stalls happen only in case of a cache/TLB miss.

- The core has separate instruction and data caches, which can be accessed in parallel. Caches are virtually indexed-physically tagged, so caches and TLBs are checked in parallel.

- L2 cache can be used, either as unified or not, and can be inclusive or not inclusive.

- Exception vector base address is 0xFFF00000. (MSR.IP=1) - Big endian only (MSR.LE=0)

- IBAT, DBAT and SR registers' functionality is not implemented. Instructions for reading and writing them are supported for being able to use the same initialization code for real target and simulator, but they are not contributing to translation process. Only SR0 is used in the translation.

- WIMG bits of page table entries are not ignored. The simulator works as these bits have the same values for all the address range as follows:

    - Write-through: this value depends on the cache configuration in the param file;
    - Caching-Inhibited: 1 (all address in 0x00000000-0x02000000 are cacheable);
    - Memory coherency: ignored, since simulator models a single core processor;
    - Guarded Memory: 1 because we model an in order processor.

- Hardware address translation is only functionally supported. Accesses to the PTEs are not taken into account in the timing simulation. However, page-faults always induce TLB misses in the timing simulator.

- Sync instruction is not supported, neither it is required, since changes are visible immediately after the execution of each instruction.

- Reading UMMCR0 returns a random value. It implements a hw random number generator exposed to sw.

- Caches and TLBS can be flushed when an instruction address is encountered using the param file option addr_reset_cache_statistics. The value past to it must be in decimal.

- Cache management is implemented through HID0 register. The functionality of the following bits are implemented:

    - ICE Instruction cache enable
    - DCE Data cache enable
    - ILOCK Instruction cache lock
    - DLOCK Data cache lock
    - ICFI Instruction cache invalidate
    - DCFI Data cache invalidate

- Cache locking disables the corresponding TLB, too (e.g. instruction cache lock disables also the instruction TLB). Cache invalidation flushes all caches and TLBs. Caches are not flushed until all their pending requests have been serviced. Write-back caches are written back in a single cycle.

- Caches are disabled by default during boot process (ICE=DCE=0). Moreover they are automatically locked during system calls or exceptions (ILOCK=DLOCK=1), and they return in their previous configuration (hid0=hid0 before interrupt) after rfi instruction execution.

- icbi and dcbi instructions are implemented. However, dcbi address 0 flushes all caches and TLBs.

- TLBs' timing simulation can be enabled even with standalone user space programs running without OS, by enabling force_always_tlb_accesses=yes in the param file.

3.2. Supported standard

TiCOS provides an implementation of most ARINC653 APIs. The APIs implementation provided by TiCOS is only partially compliant with the ARINC653 specification. See section 2.2 for further details.

3.3. Prerequisites

TiCOS requires the following tools:

- Linux OS;
- GCC 4.5.1 cross-compiler for the PowerPC platform; it should be use to compile TiCOS;

- A native gcc compiler (we recommend gcc 4.4.3, other compilers may work but they have not been tested); it should be use to compile the PROARTIS simulator and the code generator program (see later);
- PROARTIS simulator; it should be used to run the TiCOS executable. Currently TiCOS only runs on this PowerPC simulator.

3.4. Installing TiCOS

In order to install TiCOS, the following steps should be followed:

- Retrieve the TiCOS compressed archive and uncompress it.
- Set/modify the environment variables:
  - Add to your path the PowerPC compiler;
  - Add to the `LD_LIBRARY_PATH` the PowerPC compiler directory library;
  - Set the `POK_PATH` variable to the TiCOS installation directory.
  - If you want to run TiCOS on QEMU set the QEMU_HOME variable
- Chose an application example. Currently TiCOS cannot be run in stand-alone mode, so you need a user application that runs on top of TiCOS. You can choose a user application example present in the `examples` directory. Alternatively, you can generate a user application by yourself using the code generator described in the section 4.3.
- Compile TiCOS:
  - Just type `make` from within the *TiCOS_PATH/examples/<user application dir>/generated-code* directory. For more details about the compilation process see section 5.
  - In the directory `TiCOS_PATH/examples/<user application dir>/generated-code/cpu` you will find the TiCOS executable file (*TiCOS.elf*).

3.5. Running TiCOS

Currently TiCOS can be run on PROARTIS Power PC simulator or on QEMU. Please refer to the PROARTIS simulation user guide to run the TiCOS elf file. In order to run TiCOS on QEMU, check if the QEMU_ppc variable in the misc/mk/config.mk file is properly set. Then just type `make run` from the <example-name>/generated-code directory. Currently TiCOS runs on the QEMU v. 1.1.0 (BeBox machine).

4. User application and kernel configuration

In order to run TiCOS on the PROARTIS Power PC simulator you need to produce a single elf file containing both the operating system and the application running on it. The directory containing the application should have the structure shown in Figure 1 and should contain not only the application code but also configuration directives for the operating system.

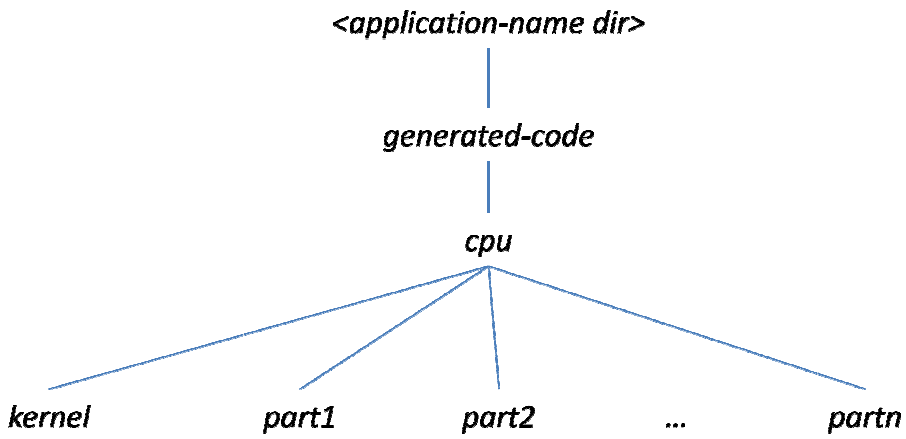

Figure 1: *User application directory structure*

The *cpu* directory contains the *kernel* directory and as many partitions directories as the number of the partitions in your applicatio. The *deployment.** files present both in the kernel and in the partitions directories contain variables and macros definition used by the TiCOS kernel and by the partitions, respectively; every partition directory also has a *main.c* file containing the code of the main thread of the partition and an *activity.c* file containing the code of the other partition threads. In the next section a macros and variables description is presented.

4.1. Macros and variables

In order to compile and run an application, you should define some macros and variables. In TiCOS, macros are used either to include in the compilation a piece of code implementing functionalities required by the application itself (for example if an application uses buffer all functions that implement buffer operation should be included in the compilation process), or to provide configurations directives to the kernel and partition code. In the first case usually their name starts with `POK_NEEDS_`, while in the second they start with `POK_CONFIG_`. A description of the macros you should define when defining your application is provided below. Some macros are related to the time-composable feature of TiCOS, so their meaning might not very clear to you (unless you read the document: "TiCOS: a Time-Composable OS"). However, if you use the code generator to generate the code of your application, it will generate for you all the necessary macros, so it might be the case that you are not required to be aware of the meanings of all macros.

4.1.1. Kernel configuration

The kernel configuration can be specified in the files *deployment.h* and *deployment.c* present in the *generate-code/cpu/kernel* directory that mainly contains macros and variables definition, respectively. While some macros are general (in the following we named it "basic macros"), other macros and variables are related to specific features or ARINC services.

Basic macros to be defined in the *kernel/deployment.h* file:

`POK_NEEDS_CONSOLE 1`
→ required to print output on the screen

`POK_NEEDS_THREADS 1`
→ required to use threads

`POK_NEEDS_PARTITIONS 1`
→ required to use partitions

`POK_NEEDS_SCHED 1`

→ required to use scheduler

`POK_NEEDS_SCHED_O1 1`

→ required to use O(1) scheduler

`POK_NEEDS_SCHED_O1_SPLIT 1`

→ required to use the O1 scheduling with task split

`POK_NEEDS_SCHED_FPPS 1`

→ required to use FPPS scheduler

`POK_NEEDS_ARINC653 1`

→ required to use the ARINC services

`POK_NEEDS_DEBUG_O1 1`

→ required to print specific debug information for the O1 scheduler

`POK_CONFIG_NB_PARTITIONS N`

→ defines the number of partitions (N)

`POK_CONFIG_NB_THREADS N`

→ defines the number of threads (N is the sum of the threads of all partitions plus 2 for the main and the idle thread)

`POK_CONFIG_PARTITIONS_NTHREADS {N,M,...}`

→ defines the number of threads used by each partition (i.e. partition 0 uses N threads, partition 1 uses M threads, etc...)

`POK_CONFIG_PARTITIONS_SIZE {X,Y,...}`

→ defines the size of each partition (i.e. partition 0 has size X, partition 1 has size Y, etc...)

`POK_CONFIG_DEFINE_BASE_VADDR 1`

→ required in case we want to manually specify the base virtual addresses

`POK_CONFIG_PARTITIONS_BASE_VADDR {0x1000,0x4000,...}`

→ provides a user-defined base virtual address (hexadecimal) for each partition. If the above macros are not set, the base virtual address is set to 0x1000 (one page offset) for every partition.

`POK_CONFIG_PARTITIONS_LOADADDR {0x9f000,...}`

→ provides a user-defined physical address (hexadecimal) for each partition

`POK_CONFIG_SCHEDULING_NBSLOTS N`

→ defines the number of time slots (N) of the major frame

`POK_CONFIG_SCHEDULING_SLOTS {N,M,...}`

→ defines the duration of every time slot

`POK_CONFIG_SCHEDULING_MAJOR_FRAME N`

→ defines the duration of the major frame. N is the sum of all scheduling time slot duration

`POK_CONFIG_SCHEDULING_SLOTS_ALLOCATION {N,M,...}`

→ defines the allocation of every slot to a partition (e.g. slot 0 is assigned to partition N, etc...)

```
POK_USER_STACK_SIZE                 8192
```
→ defines the size of the user stack. If not defined, the default value 8192 is used

```
IDLE_STACK_SIZE                     4096
```
→ defines the idle stack size. If not defined, the default value 4096 is used

```
POK_BUS_FREQ_MHZ                    74
POK_FREQ_DIV                        1
POK_FREQ_SHIFT                      0
```
→ macros related to hardware settings; they are used to calculate the frequency of the TiCOS internal clock. If not defined, the same default values are used

```
POK_DISABLE_LOADER_DATA_INIT        1
POK_PREFILL_PTE                     1
```
→ macros defining initialization settings

O(1) scheduler macros to be defined in the *kernel/deployment.h* file in order to activate the O(1) scheduler:

```
POK_NEEDS_SCHED_O1                  1
```
→ if defined, the O1 scheduler code is included in the compiled code

```
POK_CONFIG_PRIORITY_LEVELS          N
```
→ defines the number of distinct priority levels for user processes (included one for the main thread and one for idle thread)

```
POK_CONFIG_PARTITIONS_SCHEDULER     {POK_SCHED_O1,POK_SCHED_O1}
```
→ defines the scheduler used in every partition; in the example the O(1) scheduler is required for all partitions (2 in this case)

```
POK_CONFIG_NB_ASYNCH_EVENTS         N
```
→ defines the maximum number of pending asynchronous events in the system at the same time

```
POK_CONFIG_PARTITIONS_NB_ASYNCH_EVENTS {X,Y,...}
```
→ the maximum number of pending asynchronous events per partition at the same time; their summation must be the number of asynch events specified by the `POK_CONFIG_NB_ASYNCH_EVENTS` macro. This will correspond to the size of the partition queue reserved for asynchronous events

O(1) scheduling-with-task-split macros to be defined in the *kernel/deployment.h* file in order to activate the O(1) tasks split:

```
POK_CONFIG_NB_SPORADIC_THREADS      N
```
→ number of sporadic threads

In the next macros, the information related to every thread is put in the position of the array corresponding to the thread numeric identifier. In case of only one partition, the id of the main thread is 0; the ids of the other threads are obtained incrementing by one the value of the identifier of the last thread created. In case of more partitions, the identifier of the main thread of a partition equals to the sum of the threads of all previous partitions.

```
POK_CONFIG_SUCCESSOR_THREADS        {FALSE, TRUE/FALSE, ...}
```
→ for each partition thread, defines if the thread it is a successor or not.

`POK_CONFIG_SUCCESSORS_ID {0, N, ...}`

→ for each partition thread, defines its successor (by means of the thread identifier).

`POK_CONFIG_PREDECESSOR_THREADS {FALSE, TRUE/FALSE, ...}`

→ for each partition thread, defines if the thread it is a predecessor or not.

`POK_CONFIG_PREDECESSORS_EVENTS {0, EVENT_ID, EVENT_ID_1, EVENT_ID_2, ...}`

→ defines the mapping between predecessor and event (only one event per predecessor).

`POK_CONFIG_SPORADIC_TARDINESS {FALSE, FALSE, TRUE, FALSE, FALSE, ... }`

→ for each threads defined in the system (also for main threads, kernel and idle thread) defines if it has tardiness attribute or not (that is if it has to finish its execution by a deadline)

`POK_CONFIG_SPORADIC_DEADLINE {0, N, M, ...}`

→ for each partition threads defines its deadline

Ports macros to be defined in the *kernel/deployment.h* file in order to activate the sampling and queuing ports:

`POK_NEEDS_PORTS_SAMPLING 1`

→ activates the time composable ports sampling implementation

`POK_NEEDS_PORTS_QUEUING 1`

→ activates the time composable ports queuing implementation

`POK_CONFIG_MAX_QUEUING_MESSAGES N`

→ defines the maximum number of messages (N) that can be contained in every queue

`POK_CONFIG_NB_PORTS N`

→ defines the total number (N) of local port in the system

`POK_CONFIG_NB_GLOBAL_PORTS N`

→ defines the total number (N) of global port in the system

`POK_CONFIG_PARTITIONS_PORTS {0,0,1,1, ... }`

→ it is an array that defines the mapping between the ports and the partition. Every position of the array represents a port, every value a partition identifier. The order of the ports is defined in the `pok_ports_names` array (in the kernel/deployment.c file).

`POK_NEEDS_PORTS_SLOT 1`

→ activates the selective preload/post-write at the beginning of each slot. It requires to define in the kernel/deployment.c file which ports must be preloaded/post-written in each slot.

`POK_CONFIG_PARTITIONS_POSTWRITE_TIME {N,M, ... }`

→ specifies the time reserved to output operations for each scheduling slot

Ports setting

Enum declaration to be included in the *kernel/deployment.h* file:

- the identifier of each node;

- the global port identifiers: each identifier is a number that identifies each port in the system;
- the local port identifiers: each identifier is a number that identifies each port on the local node only.

Files to be included in *kernel/deployment.c* file:

- *<user-application-dir>cpu/kernel/deployment.h*
- *<TiCOS-dir>kernel/include/types.h*
- *<TiCOS-dir>kernel/include/middleware/port.h*

Arrays to be declared in the *kernel/deployment.c* file:

- `pok_global_ports_to_local_ports`
  - type: array of `uint8_t`
  - size: `POK_CONFIG_NB_GLOBAL_PORTS`
  - defines the mapping between global and local ports
  - every position of the array represents a global port, every value a local port (the local port identifiers can be used).

- `pok_local_ports_to_global_ports`
  - type: array of `uint8_t`
  - size: `POK_CONFIG_NB_PORTS`
  - defines the mapping between local and global ports
  - every position of the array represents a local port, every value a global port (the global port identifiers can be used).

- `pok_ports_names`
  - type: array of `char*`
  - size: `POK_CONFIG_NB_PORTS`
  - contains the name of each local port
  - the positions of the names in the array determine the port identifiers. (i.e. the port which name is in the position 0 will have identifier 0, etc...).

- `pok_ports_by_partition`
  - type: array of `uint8_t*`
  - size: `POK_CONFIG_NB_PARTITIONS`
  - defines the mapping between partition and local ports
  - every position of the array represents a partition, every element an array of local ports (the local port identifiers can be used)

- `pok_ports_destinations`
  - type: array of `uint8_t*`
  - size: `POK_CONFIG_NB_PORTS`
  - defines the mapping between source ports and destinations
  - every position of the array represents a local port, every element an array of global ports (the global port identifiers can be used)

- `pok_ports_nb_destinations`
  - type: array of `uint8_t`

    - size: `POK_CONFIG_NB_PORTS`
    - defines the number of destination ports of each input port.
    - every position of the array represents a local port, every value the number of destination port for that port.

- `pok_ports_kind`
    - type: array of `uint8_t`
    - size: `POK_CONFIG_NB_PORTS`
    - defines the kind (sampling or queuing) attribute of each local port.
    - every position of the array represents a local port, every value the kind of that port

- `pok_ports_nb_ports_by_partition`
    - type: array of `uint8_t`
    - size: `POK_CONFIG_NB_ PARTITIONS`
    - defines the number of ports of each partition.
    - every position of the array represents a partition, every value the number of ports for that partition

- `pok_nb_outputports_to_flush`
    - type: array of `uint8_t`
    - size: `POK_CONFIG_SCHEDULING_NBSLOTS`
    - defines the number of ports to write at the end of each slot
    - every position of the array represents a slot, every value the number of ports to write at the end of that slot

- `pok_outputports_to_flush`
    - type: array of `uint8_t*`
    - size: `POK_CONFIG_SCHEDULING_NBSLOTS`
    - defines which ports to write at the end of each slot.
    - every position of the array represents a slot, every value an array containing the ports to write at the end of that slot

- `pok_nb_inputports_to_preload`
    - type: array of `uint8_t`
    - size: `POK_CONFIG_SCHEDULING_NBSLOTS`
    - defines the number of ports to write at the end of each slot.
    - every position of the array represents a slot, every value the number of ports to write at the end of that slot

- `pok_inputports_to_preload`
    - type: array of `uint8_t*`
    - size: `POK_CONFIG_SCHEDULING_NBSLOTS`
    - defines which ports to preload at the beginning of each slot.
    - every position of the array represents a slot, every value an array containing the ports to preload at the beginning of that slot

Events, buffers and blackboards macros to be defined in the *kernel/deployment.h* file:

```
POK_NEEDS_LOCKOBJECTS          1
```

→ includes the lock object code. Required in case of events, blackboards or buffer.

`POK_CONFIG_NB_LOCKOBJECTS N`

→ number of lock kernel lock objects (should be equals to the sum of events, blackboards and buffers)

`POK_CONFIG_PARTITIONS_NLOCKOBJECTS {H,K,J, ... }`

→ how many lock objects per partition

Error management macros to be defined in the *kernel/deployment.h* file:

`POK_NEEDS_ERROR_HANDLING 1`

→ Includes the error management functions.

4.1.2. Partitions configuration

The partitions configuration can be specified in the files *deployment.h* and *deployment.c* present in the *generate-code/cpu/part<n>* directory that mainly contains macros and variables definition, respectively. While some macros are general (in the following we named it "basic macros"), other macros and variables are related to specific features or ARINC services.

Basic macros to be defined in the *part<n>/deployment.h* file:

`POK_CONFIG_NB_THREADS N`

→ defines the number of the ARINC threads in the partitions (includes the main thread).

`POK_NEEDS_ARINC653 1`

→ required to use the ARINC653 services

`POK_NEEDS_ARINC653_TIME 1`

→ required to use the ARINC653 time-related services

Ports macros to be defined in the *part<n>/deployment.h* file:

`POK_NEEDS_ARINC653_SAMPLING 1`

→ activates the ARINC Sampling port service

`POK_NEEDS_ARINC653_QUEUEING 1`

→ activates the ARINC Queueing port service

`POK_CONFIG_MAX_MESSAGE_SIZE N`

→ defines the maximum size of a message (for sampling and queuing ports)

`POK_CONFIG_PART_NB_INPUT_SAMLING_PORTS N`

→ defines the number of input sampling ports in the partition

`POK_CONFIG_PART_NB_INPUT_QUEUING_PORTS N`

→ defines the number of input queuing ports in the partition

`POK_CONFIG_PART_MAX_QUEUING_MESSAGES N`

→ defines the maximum number of messages that can be contained in every queue of the partition; its value must be equal or less than that of the macro `POK_CONFIG_MAX_QUEUING_MESSAGES` defined in the *kernel/deployment.h* file.

`POK_CONFIG_NB_SYSTEM_PORTS N`

→ defines the total number of ports in the system; should have the same value of the macro `POK_CONFIG_NB_SYSTEM_PORTS` defined in the *kernel/deployment.h* file

Ports settings

Files to be included in *<part_number>/deployment.c* file:

- *<user-application-dir>/cpu/<part_number>/deployment.h*
- *<POK-dir>/libpok/include/arinc653/types.h*

Arrays to be declared in the *<part_number>/deployment.c* file:

- `receiving_addresses`
  - type: array of `MESSAGE_ADDR_TYPE *`
  - size: `POK_CONFIG_NB_SYSTEM_PORTS`
  - defines the mapping between each port and the address of the variable that will be used to read that port (see document "TiCOS: a Time-Composable OS" for further details)
  - every position of the array represents a local port, every value the address of the variable that will be used to read that port
  - not required when the dynamic implementation has been selected, i.e., the `POK_NEEDS_PORTS_(SAMPLING|QUEUING)_DYNAMIC` macro has been defined.

In case of sampling ports (see document "TiCOS: a Time-Composable OS" for further details):

- it is required to allocate as many input message buffers as the number of input sampling ports.
  - type of each input message buffer: array of `unsigned char`
  - size of each input message buffer: `POK_CONFIG_MAX_MESSAGE_SIZE`
- `input_buffers`
  - type: array of `unsigned char *`
  - size: `POK_CONFIG_PART_NB_INPUT_SAMPLING_PORTS`
  - each position of the array points to an input message buffer previously allocated; when a port is created, a mapping between the first buffer not yet assigned and the port is performed

In case of queuing ports:

- Since a queuing port can maintain more than one message at a time, it is required to allocate an array of input message buffers for every queuing port.
  - type: array of `unsigned char *`
  - size: `POK_CONFIG_PART_MAX_QUEUING_MESSAGES`

- The total number of input message buffers to allocate is `POK_CONFIG_PART_NB_INPUT_QUEUING_PORTS` times the value `POK_CONFIG_PART_MAX_QUEUING_MESSAGES`
    - type of each input message buffer: array of `unsigned char`
    - size of each input message buffer: `POK_CONFIG_MAX_MESSAGE_SIZE`

- `input_buffers_queuing`
    - type: array of `unsigned char *`
    - size: `POK_CONFIG_PART_NB_INPUT_QUEUING_PORTS`
    - each position of the array points to an array of input message buffers; when a port is created, a mapping between the first array of input message buffers not yet assigned and the port is performed during the preload phase, the messages present in the input port are copied in array of buffers previously assigned to this port.

Events macros to be defined in the *part<n>/deployment.h* file:

`POK_NEEDS_EVENTS 1`

→ required to use events

`POK_CONFIG_NB_EVENTS N`

→ defines the number of events needed in the partition

`POK_NEEDS_ARINC653_EVENT 1`

→ activates the ARINC events service

`POK_CONFIG_ARINC653_NB_EVENTS` N^A

→ defines the number of ARINC events needed in the partition

Events related-variables to be declared in the *partA/deployment.c* file:

- An array of `EVENT_ID_TYPE` to represent all the events within the partition.
- An array of strings to hold the event names, e.g.,:
  `char* events_names[POK_CONFIG_ARINC653_NB_EVENTS] = {…};`

Buffers macros to be defined in the *part<n>/deployment.h* file:

`POK_NEEDS_BUFFERS 1`

→required to use buffers

`POK_CONFIG_NB_BUFFERS N`

→defines the number of buffers needed in the partition

`POK_CONFIG_MAX_BUFFERS_MESSAGES N`

→defines the maximum number of messages in each buffer

`POK_NEEDS_ARINC653_BUFFER 1`

→activates the ARINC buffer service

Buffers-related variables to be declared in the *partA/deployment.c* file:

- An array of `BUFFER_ID_TYPE (or uint_8)` to represent all buffers within a partition (although the array representation is not mandatory).
- An array of strings to hold the buffer names, e.g.,:

```
char* pok_buffer_names[POK_CONFIG_NB_BUFFERS] = {...};
```

Blackboards macros to be defined in the *part<n>/deployment.h* file:

`POK_NEEDS_BLACKBOARDS 1`

→ required to use blackboards

`POK_CONFIG_NB_BLACKBOARDS N`

→ defines the number of blackboards needed in the partition

`POK_NEEDS_ARINC653_BLACKBOARD 1`

→activates the ARINC blackboard service

Blackboard-related variables to be declared in the *partA/deployment.c* file**:**

- An array of `BLACKBOARD_ID_TYPE (or uint_8)` to represent all blackboards
- An array of strings to hold the blackboard names, e.g.,:

```
char* pok_blackboards_names[POK_CONFIG_NB_BLACKBOARDS]={ ... };
```

4.2. How to build your own application

The example directory already contains some example applications running on TiCOS, but you can also write your own application. If you want to preserve the time composable nature of ARINC services implemented by TiCOS, however, your applications must fulfill a set of prerequisites. You can find guidelines and requirements to invoke ARINC services in your application and still preserving time-composability in the document: “TiCOS: a Time-Composable OS”.

In case you are planning to build your application from scratch be aware of the fact that it can be a time-consuming and an error-prone process. Alternatively, you can modify an existing application, possibly the one that more resemble the one you want to build. However, if you need to apply several modifications to the original application, also this solution could be error-prone, due to the number of variable and macro definitions required. However, an incorrect configuration may easily lead to a faulty behavior: it is therefore preferable to automatically generate the configuration of your application (and also part of your application code). You just have to describe your application using an XML file (the file should be valid with respect to the XML schema we provided) and then give it as input to our code generator (presented in the next section). You will obtain a *generated-code* directory containing all files needed to compile your application (included the OS configuration directives and the application code skeleton).

4.3. Code generator

TiCOS is distributed along with a code generation utility in the form of a C program that takes as input an XML application description and generates the application code. In every existing user application directory you can find the XML file that has been used to automatically generate the application itself.

For more information please refers to the TiCOS code generator documentation.

5. Compilation process

If you already have your own ARINC application and want to run it on TiCOS, the first action you need to do is to incorporate your application code into TiCOS. To this end, in this section we provide you with some further details about TiCOS compilation process that may make easier the inclusion of your own application into the TiCOS executable.

As already explained, the overall architecture of TiCOS is made up of two parts: the TiCOS kernel and partitions (in turn made up of libTiCOS and user-code). Each part of the system (kernel, partition) is first separately compiled and then integrated into a single executable. The compilation of each part produces a distinct binary `ELF` file. Eventually, all `ELF` files are integrated to produce a single bootable `ELF` binary, which in fact includes different multiple distinct binaries: the code for the kernel and the code of all partitions.

Conventions:

`<makefile-dir> = <TiCOS-dir>/misc/mk/`

`<application-dir> = <TiCOS-dir>/examples/<my-example>/generated-code/`

5.1. TiCOS kernel

Currently, the compilation process of the TiCOS kernel is not independent from the application(s) running on in, and, in fact, in the current directory structure, the compilation of TiCOS kernel can be launched from an application Makefile only.

The application files involved in the TiCOS kernel compilation process are:

a. ***deployment.h*** (in the directory *<application-dir>/cpu/kernel*
   This header file is included in the compilation of all .c files:

   ```
   $(CC) -c $(CFLAGS) $(COPTS) $< -o $@
   (see file <makefile-dir>rules-common.mk)
   ```

   Where:

   ```
   COPTS += -include $(DEPLOYMENT_HEADER)
   ```
   (see file *<makefile-dir>/rules-kernel.mk*)

   ```
   DEPLOYMENT_HEADER=$(shell pwd)/deployment.h
   ```
   (see file *<application-dir>/cpu/kernel/Makefile*)

b. ***deployment.c*** (in the directory *<application-dir>/cpu/kernel*)
   The corresponding object file (*deployment.o*) is included during the linker phase for the creation of the *kernel.lo* file:

   ```
   $(LD) $(LDFLAGS) $(LDOPTS) -r $(LO_DEPS) $(LO_OBJS) -o $(LO_TARGET)
   ```
   (see file *<makefile-dir>rules-common.mk*)

   Where:

```
LO_DEPS = TiCOS.lo
LO_OBJS = deployment.o
```
(see file *<application-dir>/cpu/kernel/Makefile*)

```
LO_TARGET = kernel.lo
```
(see file *<application-dir>/cpu/kernel/Makefile*)

The Figure 2 illustrates the TiCOS kernel compilation process, at the end of which the kernel.lo file is obtained.

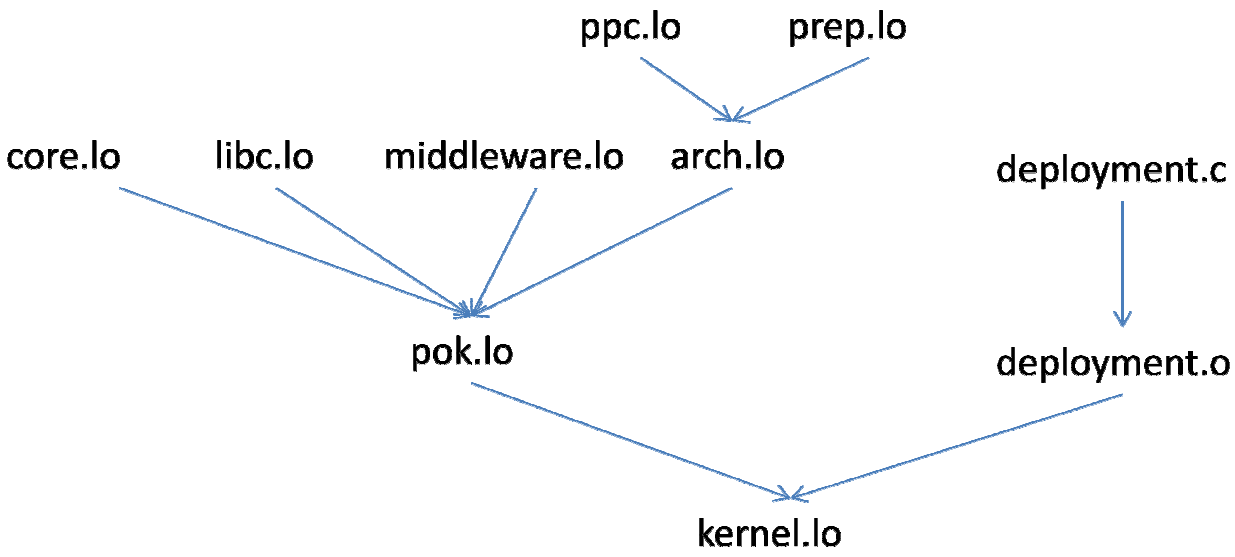

Figure 2*: TiCOS kernel compilation process*

5.2. TiCOS partitions

When compiling an application, an `ELF` file is produced for each partition. This `ELF` file includes the user application code (in the directory *<application-dir>*) and the TiCOS library (in the directory *<TiCOS-dir>/libTiCOS*). In other words, the partition `ELF` file includes both the application-dependant files (such as the files *main.o* and *activity.o*) and the static library *libTiCOS.a* obtained by the compilation and partial linking of the source files in the directory *<TiCOS-dir>/libTiCOS*. It is worth noticing that the libTiCOS files are compiled again each time a different partition has to be compiled (that is to say they are compiled as many time as the number of the partitions). This is because the libTiCOS source files need to be compiled including the application-dependant file *<application-dir>/cpu/<part-name>/deployment.h*.

The steps for compiling the TiCOS partitions (and producing the partition `ELF` files) are:

a. **Compilation of libpok files**:

```
$(CC) -c $(CFLAGS) $(COPTS) $< -o $@
```
(see file *<makefile-dir>rules-common.mk*)

Where:

```
COPTS += -include $(DEPLOYMENT_HEADER)
```

(see file *<makefile-dir>/rules-kernel.mk*)

```
DEPLOYMENT_HEADER=$(shell pwd)/deployment.h
```

(see file *<application-dir>/cpu/<part-name>/Makefile*)

b. **Creation of `.lo` files**:

```
$(LD) $(LDFLAGS) $(LDOPTS) -r $(LO_DEPS) $(LO_OBJS) -o $(LO_TARGET)
```

(see file *<makefile-dir>rules-common.mk*)

The values of `LO_DEPS, LO_OBJS` and `LO_TARGET` vary between different invocations.

c. **Creation of `libTiCOS.a` file**:

```
$(AR) $@ $(LO_DEPS)
```

(see file *<makefile-dir>/rules-common.mk*)

```
Where :

$@ = libTiCOS.a

LO_DEPS = arch/arch.lo core/core.lo drivers/drivers.lo
      middleware/middleware.lo arinc653/arinc653.lo libm/libm.lo
      protocols/protocols.lo libc/libc.lo
```

d. Creation of `<part-name>.elf` file:

The linker script used for creating the partition ELF file is *partition.lds* in the directory *<TiCOS-dir>/misc/ldscripts/ppc/prep*.

```
$(LD) $(LDFLAGS)
      -T $(TiCOS_PATH)/misc/ldscripts/$(ARCH)/$(BSP)/partition.lds
      $+ -o $@ -L$(TiCOS_PATH)/libTiCOS -lTiCOS -Map $@.map
```

Where:

$+ = application object files (e.g. *main.o, activity.o*)

The Figure 3 illustrates the compilation process of a partition.

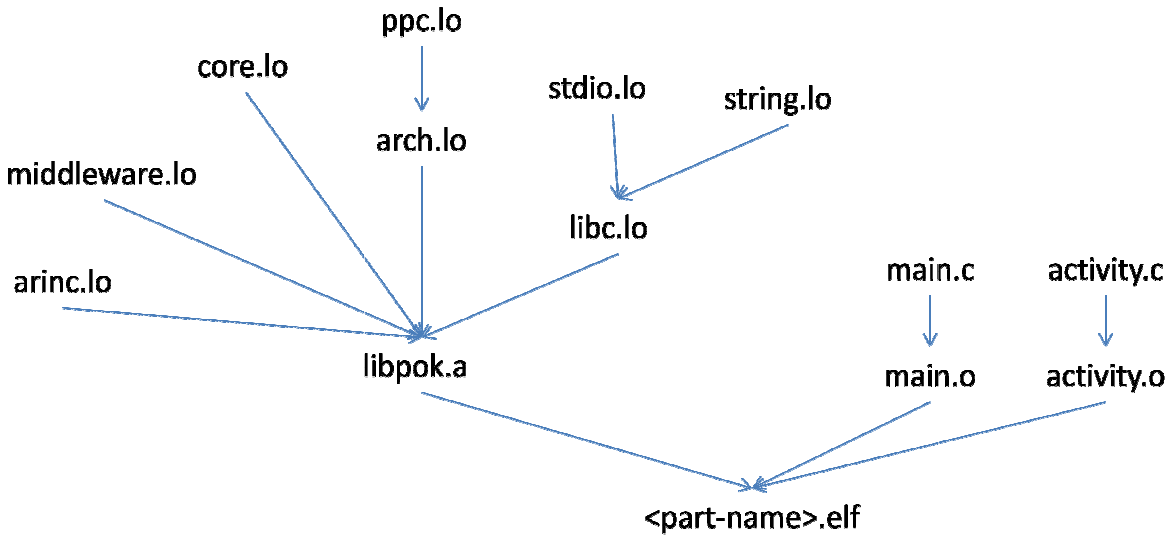

Figure 3: *TiCOS partition compilation process*

5.3. Combining TiCOS kernel and partitions

In order to obtain the *TiCOS.elf* executable on the simulator, the following steps are performed:

a. **The `ELF` files** (one for each partition) previously obtained **are**:
   - padded to get aligned file size

     ```
     for v in $(PARTITIONS); do \
     dd if=/dev/zero of=$$v oflag=append conv=notrunc bs=1
     count=`echo "4 - (\`ls -l $$v | awk '{print $$5}'\` % 4)" | bc`;\
     done
     ```

   - appended to a single binary archive named `partitions.bin`:

     ```
     cat $(PARTITIONS) > partitions.bin
     ```

     (see the target `assemble-partitions` of the file *<makefile-dir>/rules-main.mk*)

b. **The *partitions.bin* archive is** then **added to the object file** (*sizes.o*) obtained from an automatically generated definition of the partition sizes (file *sizes.c*).

```
$(RM) -f sizes.c
$(TOUCH) sizes.c
$(ECHO) "#include <types.h>" >> sizes.c
$(ECHO) "uint32_t part_sizes[] = {" >> sizes.c
N=1 ; for v in $(PARTITIONS); do \
if test $$N -eq 0; then $(ECHO) "," >> sizes.c ; fi ; N=0 ;\
      ls -l $$v|awk '{print $$5}' >> sizes.c ; \
done
$(ECHO) "};" >> sizes.c
```

```
$(CC) $(CONFIG_CFLAGS) -I $(TiCOS_PATH)/kernel/include -c sizes.c -o
sizes.o
$(OBJCOPY) --add-section .archive2=partitions.bin sizes.o
```

(see target `$(TARGET)` of the file *<makefile-dir>/rules-main.mk*).

c. **The files *sizes.o* and *kernel.lo*** (obtained from the TiCOS kernel compilation process) **are then linked together** (using the linker script *kernel.lds*)

```
$(LD) $(LDFLAGS)
-T $(TiCOS_PATH)/misc/ldscripts/$(ARCH)/$(BSP)/kernel.lds
-o $@ $(KERNEL) $(OBJS) sizes.o -Map $@.map
```

Where:

`KERNEL=kernel/kernel.lo` (see file `<application-dir>/cpu/Makefile`)

The following figure illustrates the overall creation process of the `TiCOS.elf` executable file.

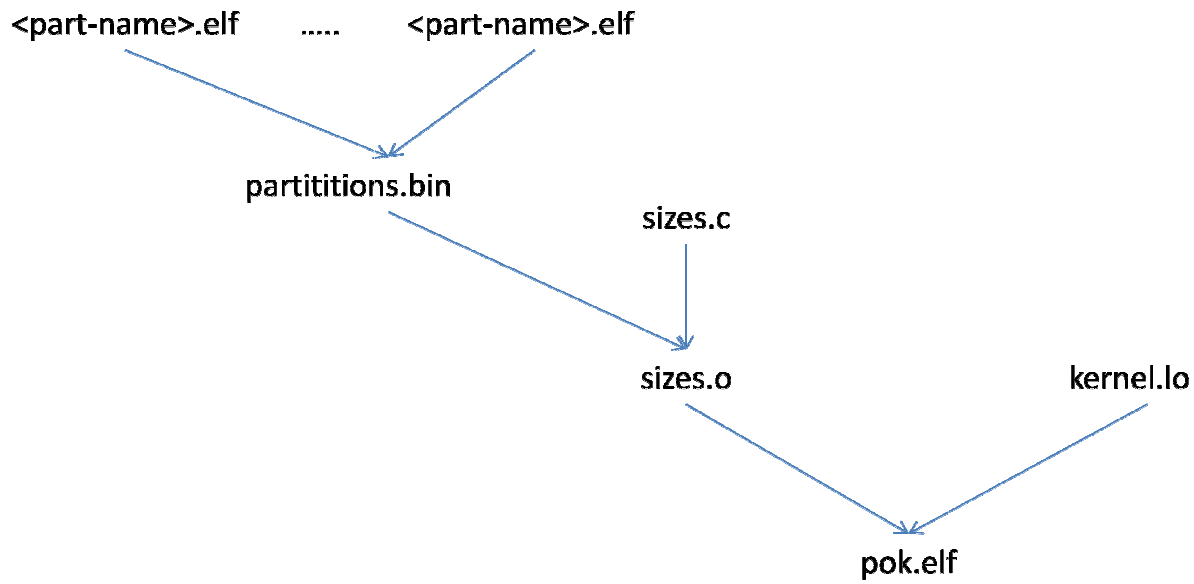

Figure 4: *TiCOS compilation process*