SIMA
Overview

GMV-SKYSOFT
Torre Fernão de Magalhães
Av. D. João II Lote 1.17.02, 7º Andar
1998 - 025 Lisboa Portugal

Property of GMV
© GMV, 2010; all rights reserved.

gmv®
INNOVATING SOLUTIONS

1 PURPOSE

Simulated Integrated Modular Avionics (SIMA) is an execution environment, providing the ARINC 653 Application Programming Interface (API) and robust partitioning to operating systems that do not support these features by themselves. SIMA is designed to run on all POSIX-compliant OSes; it is tested and optimised for the Native POSIX Thread Library (NPTL), available on OSes like GNU/Linux, kernel version 2.6 or higher, and for RTEMS, version 4.6 or higher.

This document focuses on SIMA on Linux. Its purpose is to give a brief overview on the core features of the SIMA tool chain and to describe the command line tools *POS, MOS, makeports* and *simout* and, additionally, the Logbook System.

© GMV, 2010; all rights reserved

2 SIMA OVERVIEW

Simulated Integrated Modular Avionics (SIMA) is an execution environment, providing the ARINC 653 Application Programming Interface (API) and robust partitioning to operating systems that do not support these features by themselves. SIMA is designed to run on all POSIX-compliant OSes; it is tested and optimised for the Native POSIX Thread Library (NPTL), available on OSes like GNU/Linux, kernel version 2.6 or higher, and for RTEMS, version 4.6 or higher.

The ARINC 653 standard specifies a programming interface for a Real-Time Operating System (RTOS), and, in addition, establishes a particular method for partitioning resources over time and memory. Today, this standard has been established as an important foundation for the development of safety-critical systems in the avionics industry.

ARINC 653 defines support for robust partitioning in on-board systems, such that one processing unit, usually called a module, is able to host one or more avionics applications and to execute these applications independently. This can be achieved if the underlying system, often called the Module Operating System (MOS), provides separation of the avionics applications, such that

- Each partitioned function has guaranteed access to the processor. The guarantees shall reflect the frequency as well as the execution time of the specific application;
- A failure in one partitioned function cannot cause a failure in another partitioned function.

In consequence, the partitioning approach allows reducing on-board hardware and, at the same time, eases verification, validation and certification.

The unit of partitioning is called a partition. In a given sense, a partition is equivalent to a program in a single application environment: it comprises data, code and its own context configuration attributes (see Figure 1).

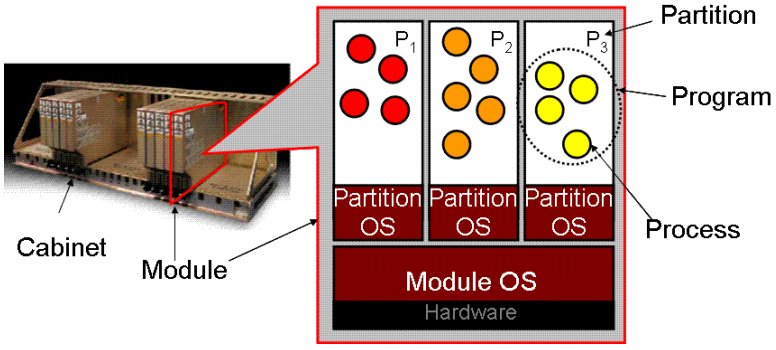

Figure 1: Partitioning

Partitioning separates applications in two dimensions: space and time. Spatial separation means that the memory of a partition is protected. No application can access memory out of the scope of its own partition. Temporal separation means that only one application at a time has access to system resources, including the processor; therefore only one application is executing at one point in time – there is no competition for system resources between partitioned applications.

© GMV, 2010; all rights reserved

ARINC 653 defines a static configuration where each partition is assigned a set of execution windows. The program in the partition associated with the current execution window gains access to the processor. When the execution window terminates, the program is preempted; when the next execution window starts, the program continues execution from the point it was previously preempted.

Processes within the scope of a partition are scheduled by a priority-based preemptive scheduler with first-in-first-out (FIFO) order for processes with the same priority.

Processes in ARINC 653 must not be confused with processes in POSIX: In ARINC 653 processes in the same partition share the same address space. There is no memory separation between processes. However, since partitions are separated, processes in different partitions cannot access each other's memory. Communication between processes in different partitions is achieved by ports and channels. Ports are communication end points either for reading or writing that are identified by a name that is unique in the scope of the partition. Channels connect these ports transparently to application code.

In SIMA, ARINC 653 partitions are mapped to POSIX processes and ARINC 653 processes are mapped to POSIX threads. Each SIMA application is, hence, linked to a single POSIX program, containing user code and data, the APEX code and data and, finally, the platform execution environment, i.e. the NPTL for Linux.

The Module Operating System (MOS), controlling the different POSIX processes, belonging to the same simulated module, is likewise linked to one POSIX process. The following picture illustrates this design:

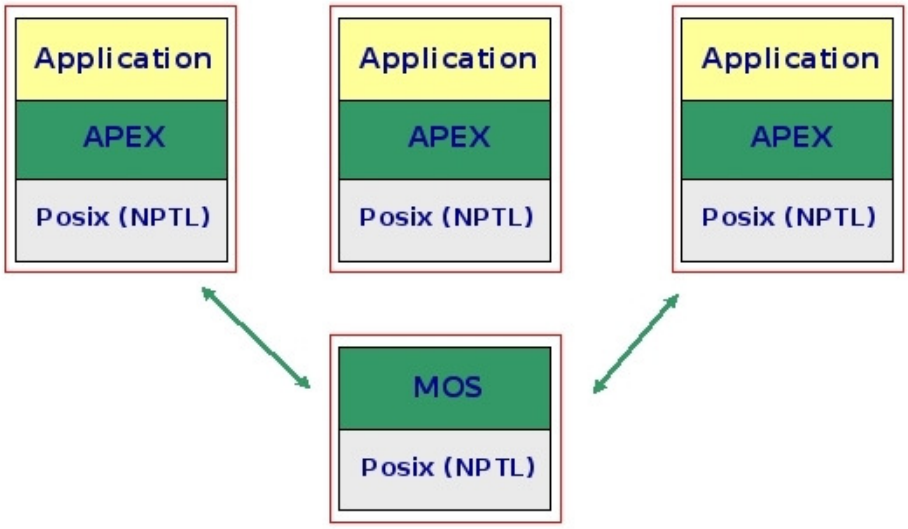

Figure 2: SIMA Architecture

The APEX services are implemented by a static library, called POS. The POS implements the APEX process scheduler on top of the POSIX FIFO scheduler (*sched_fifo*). POSIX features are encapsulated within a core layer; this way main parts of the APEX code do not rely directly on POSIX, but on scheduling policies implemented by the POS itself. The advantage of this approach is enhanced portability - there is even an implementation of the SIMA POS, running on bare hardware - and the fact that scheduler features that introduce subtle differences between different POSIX implementations are handled in the core layer and hidden from the APEX implementation.

The MOS implements the APEX partition scheduler. To be able to suspend and resume partitions, commands are exchanged with the POS layer in the partitions using signals and

© GMV, 2010; all rights reserved

shared memory segments. Obviously, this approach does not answer safety and security threats, caused by random errors in the partitioned code. The POS has to respond correctly to the given commands which may not be true in the case where faulty or malicious application code corrupts the state of the POS. In fact, the POS was designed and developed, following safety critical software guidelines; its purpose is to support embedded applications. The MOS, however, was not; the MOS does only simulate the behaviour of an ARINC 653 compliant OS on top of non-safety aware systems like standard Linux.

POS and MOS are designed to support real-time applications. They use the real-time programming interfaces of the POSIX thread library, like FIFO scheduling and thread priorities. Additionally, all memory used during execution is created during initialisation and locked in RAM, avoiding paging and the latency penalties caused by swapping pages in and out. However, hard real-time guarantees cannot be met without a fully preemptive operating system kernel. Standard Linux is not yet a preemptive kernel. Two alternatives are possible: Deadlines must be restricted to the guarantees possible to be achieved on Linux, or, alternatively, a kernel, patched with the PREEMPT-RT patch by Ingo Mólnar and Thomas Gleixner, can be used to achieve guarantees of much lower granularity.

© GMV, 2010; all rights reserved

3 HARD REAL-TIME ON LINUX

The main problem, achieving hard real-time behaviour on Linux, is latency, defined as the time between the arrival of an event (like an interrupt) and the execution of its response. As a general purpose OS; Linux is designed to enhance the average response time, whereas real-time systems aim at enhancing the worst case response time as this is the fundamental factor of impact on the system predictability.

Linux high latencies are a consequence of a non-preemptive kernel approach; the kernel contains large protected sections, where the kernel can't be pre-empted by a user task. However, it is the user tasks, implementing the response to an event.

To overcome this situation, Ingo Mólnar, one of the authors of the NPTL, and Thomas Gleixner reworked the kernel code to reduce non-preemptible sequences to a minimum. This code is available as a patch, called PREEMPT-RT patch (see http://rt.wiki.kernel.org).

The latency that can be expected with the PREEMPT-RT patch depends on system configuration. Main drivers for latency are hardware interrupts, causing the kernel to become active and to enter the remaining non-preemptible sequences. Sources of interrupts are, for instance, the network interface, the graphic card, typically when running an X server, and service interrupts coming from the board. This last kind of interrupts is worse with newer hardware and, especially, with all kinds of portable computers. There are scripts available to reduce interrupts on your system, but it is not recommended to apply such scripts if you are not fully aware of what they do in detail. Disabling service interrupts, for instance, may seriously harm your hardware.

On systems with different hardware configurations, the following latencies have been measured, after running benchmarks for 24 - 48 hours. All values in the table are in µs; runlevel 5 means a multi-threaded environment with network enabled and an X server running; runlevel 3 means a multi-threaded environment with network enabled, but without a running X server.

	Runlevel	Best Case	Average	Worst Case
Desktop	5	1	7	54
	3	1	7	17
Laptop	5	1	18	62
	3	1	11	48

Table 1: Latency of preemptible Linux kernel

Industry experience confirms that deadlines down to 100 µs can be guaranteed even on a system that runs graphical user interfaces (GUI). Without GUIs, even shorter deadlines may be possible. However, deadlines of 100µs are sufficient for typical avionics use cases and even great for simulated avionics running on a desktop computer.

© GMV, 2010; all rights reserved

4 THE POS LIBRARY

The POS library provides the ARINC 653 services to hosted applications. In particular, it implements the

- ARINC 653 services of ARINC 653 Part 1 “Required Services”: Partition Management, Process Management, Time Management, Inter-Partition and Intra-Partition Communications, Health Monitor Services;
- A subset of ARINC 653 Part 2 “Extended Services”: Logbook System.

The POS library is statically linked to the hosted application. It provides the library code and data, including interfaces to other simulation components.

SIMA can run in two different modes:

- Simulation of a multi-partition system scheduled by the MOS application;
- Or executing a single partition that may or may not be part of a multi-partition system; this latter mode is called standalone mode.

The simplest way to use the POS library is to build a partition for standalone execution. In standalone execution, the program runs as an ordinary POSIX process, there is not time separation from other APEX applications. Running applications in standalone mode is a good way for verifying the program functional behaviour. It does not require configuration files and allows application debugging.

The user code is linked against the POS library and a set of default objects:

- The main entry point that may be exchange by user code;
- Stubs for applications that do not use ports or logbooks

The following figure illustrates this tool chain:

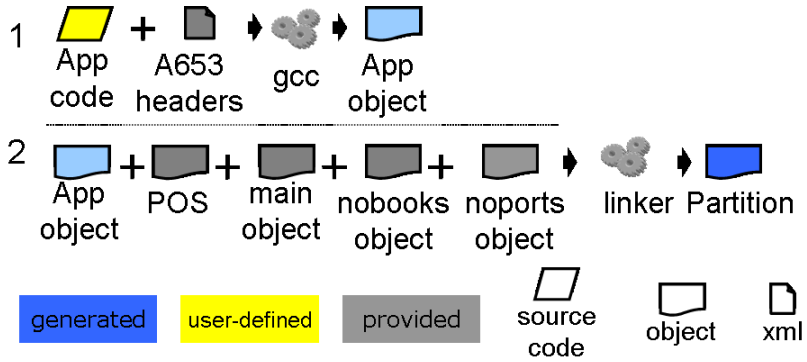

Figure 3: SIMA Basic Tool Chain

© GMV, 2010; all rights reserved

5 MOS SIMULATOR

The purpose of the MOS program is to schedule and to health monitor partitioned applications. The MOS works in three phases:

1. The configuration files are read and the corresponding entities like partitions and health monitor tables are created in memory;
2. The MOS goes into real-time mode and starts the partitioned applications;
3. The MOS enters the scheduling phase; from now on, the program will suspend and resume partitions and wait for health monitoring events.

The MOS reads two configuration files: the main SIMA configuration file, containing simulator-specific information and the ARINC 653 configuration.

The partition scheduling is defined in the ARINC 653 configuration file in three hierarchy levels: the *Module_Schedule* contains one *Partition_Schedule* per partition; each *Partition_Schedule* contains a set of *Window_Schedules*.

A *Window_Schedule* defines the starting point and the duration of one execution window. This way one partition may have n execution windows assigned to it.

The *Window_Schedules* together define the *Module_Schedule* or major execution frame that is repeated during module run-time. If there is a gap between the end of one execution window and the beginning of the next, the MOS automatically fills it up with an execution window without a partition assigned to it.

In the SIMA configuration, *Window_Schedules* are split into smaller units, called *slices*. Each execution window may have up to three slices: the start slice, the main slice and the end slice.

The main slice is always present; it represents the time guaranteed for execution of the application code itself. The start slice and the end slice are reserved for message transportation. Only one process is allowed to execute during start and end slices: the UDP listener. When the next execution window has a start slice defined, the MOS will explicitly set this process from the WAITING to the RUNNING state. When the duration of the start slice expires, the listener is set back to WAITING and the application is resumed. When the duration of the main slice expires, the application is suspended, and, if the execution window has an end slice defined, the listener process will be set to RUNNING. When no slices are defined for a window, the duration of the main slice is equal to the duration of the *Window_Schedule*.

© GMV, 2010; all rights reserved

6 HEALTH MONITORING

Errors occurring during the execution of partitioned applications are reported to the MOS. The MOS looks up the error in the configuration and applies the corresponding action. Errors are handled on one of three possible levels: MODULE, PARTITION or PROCESS. Actions on PARTITION and MODULE level are directly specified in the configuration. Errors on PROCESS level are delegated to a user defined error handler process (EH). When the MOS invokes the EH, the latter is started and the control returned to the POS of the affected partition. Since the EH runs as the highest priority process within the partition, it will preempt any other process and run immediately.

There are four sources of errors:

- Internal errors of the POS
- Deadline misses detected by the POS
- RAISE_APPLICTATION_ERROR issued by the application
- Signals from the Linux kernel

Most errors, like segmentation faults, numeric errors or stack overflows, cannot be detected by the POS or MOS. Instead, the Linux kernel sends a signal to the POSIX process, i.e. the APEX partition or the MOS program; the POS catches those signals and reports the incident to the MOS, the MOS handles them directly. The next figure illustrates this general behaviour:

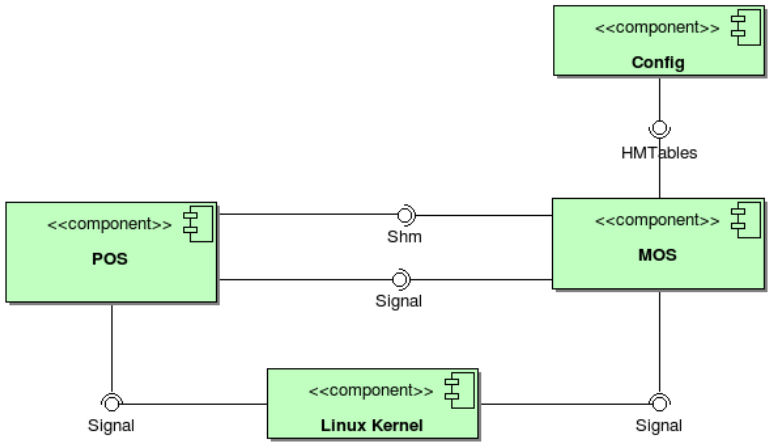

Figure 4: SIMA Error Handling

The handling of the *RAISE_APPLICATION_ERROR* service is an exception to the common error handling. Unlike other errors, this error is not delegated to the MOS, instead, the POS immediately invokes the EH. However, only errors, defined by the application itself, may be raised by this service.

© GMV, 2010; all rights reserved

7 PORTS

ARINC 653 applications use ports to communicate with the outside world. Ports are memory areas within the partition address space where messages are written to or read from by application code. If ports are connected to a channel, the messages in a source port are copied to the memory area of the destination port.

This transport mechanism is invisible to the application. It is also transparent to the application where the other port is located: In a partition on the same module or on another computer.

Channels are defined in the ARINC 653 configuration as a relation between a source port and one or more destination ports. The ARINC 653 standard leaves it open whether messages are sent to one destination or all destinations. SIMA sends a message to all destination ports that are configured. This way, it is possible to emulate a simple unicast environment with 1:1 relation between ports and a multicast environment with 1:n relation.

The ARINC 653 configuration defines the logical relation between ports. The mapping to lower level entities implementing ports is out of the scope of the standard. SIMA maps ARINC 653 ports to UDP ports on Linux. The additional information needed by this mapping is given in the SIMA main configuration file.

The configuration is not read directly by the POS. Instead, a C-stub must be generated from the configuration using the *makeports* tool. The *makeports* tool is called as follows:

```
makeports <sima_config> <partition> > <c-file>
```

```
makeports config/sima.xml System > systemports.c
```

In a makefile the command could be used like this:

```
systemports.c: config/sima.xml config/a653.xml
```

```
makeports config/sima.xml > systemports.c
```

```
systemports.o: systemports.c
```

```
...
```

```
system: ... systemports.o ...
```

```
...
```

The following figure shows the tool chain for the generation of ports:

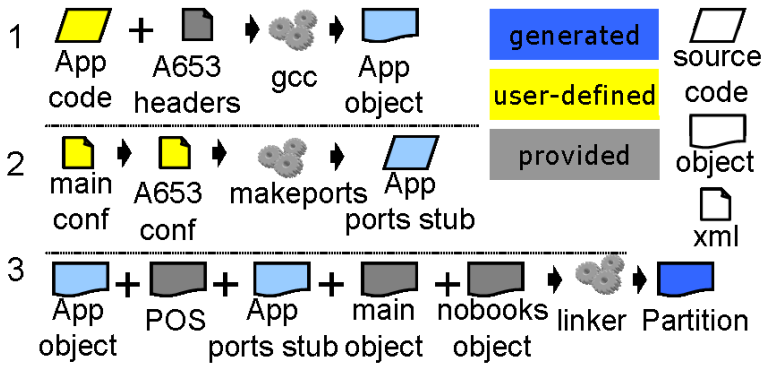

Figure 5: Ports Tool Chain

© GMV, 2010; all rights reserved

The channel between ports is implemented by an internal process, called `_apx_udp_listen`. The process is automatically started when the application is connected to the MOS or with the *--connect* option given in standalone mode.

In standalone mode, this process runs with a priority lower than user process priorities. This implies that messages are only sent and received when no user process is ready. It implies also that the transportation mechanism interferes with the user process activity. These restrictions are acceptable for debugging, but certainly not for the simulation of a complete IMA system. For this purpose, time slices are used in the MOS.

© GMV, 2010; all rights reserved

8 SIMOUT

The SIMOUT program shows the output of the MOS and up to six partitions in a graphical environment based on the *curses* library, available in most Linux distributions. SIMOUT invokes the MOS automatically, using the value in the Startup attribute of the MOS node in the SIMA configuration as path to the MOS startup script.

When the MOS and the partitions have been started, an output as the following, showing SIMOUT with five partitions, is presented:

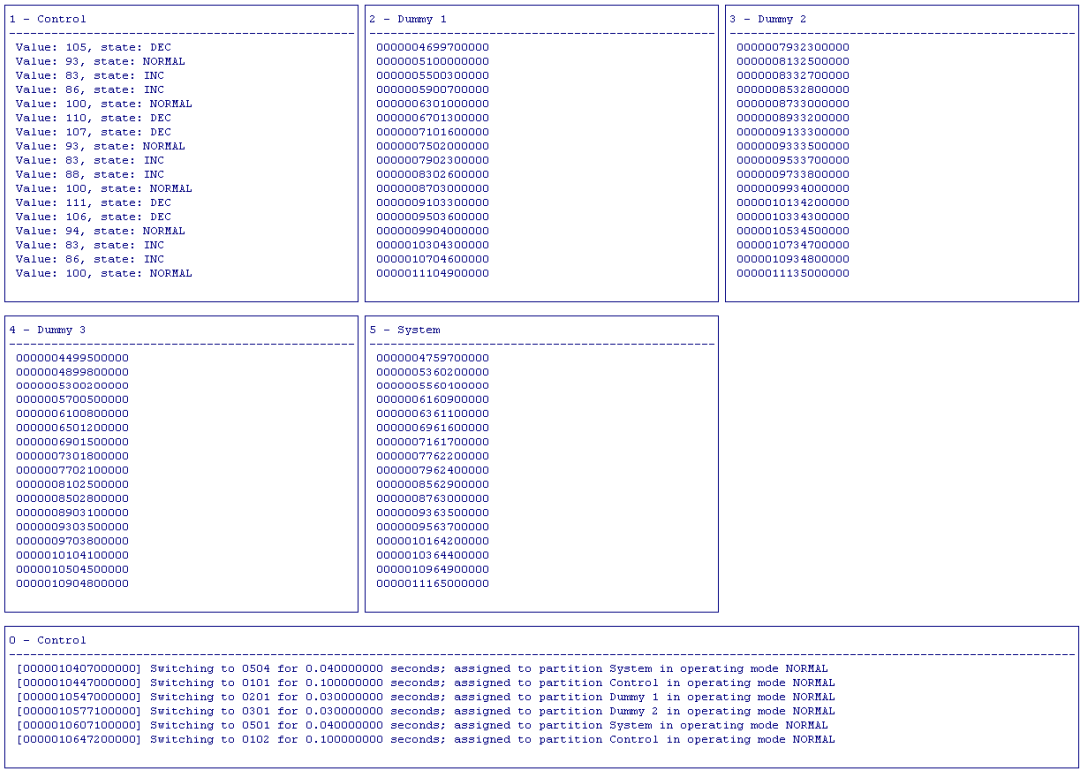

Figure 6: Terminal with SIMOUT

The figure shows a terminal where a simulated module is running. There are five partitions, called “Control”, “Dummy 1” through “Dummy 3”, “System”. Below, the output of the MOS simulator is shown. Since the system is called like the first partition, “Control” is also the title of the MOS window.

© GMV, 2010; all rights reserved

9 SIMA LOGBOOKS

ARINC 653 logbooks services are implemented by SIMA through system partition, shared memory and ordinary files. A system partition is used to engrave messages in the logbook, it reads from the `IN_PROGRESS` buffer and engraves in the non-volatile memory (NVM). Its purpose is to provide the ARINC 653 logbooks two step writing behaviour – the time demanded for engraving messages belongs to the system partition schedule window and not to the application partition that owns the logbook.

A shared memory is used to realize the logbook `IN_PROGRESS` buffer, it is visible to the partition that owns the logbook and to the system partition that engraves the messages. The NVM, implemented by a file is also accessed by both partitions. The application partition writes messages to the shared memory and accesses the NVM for reading operations. While the system partition reads messages from the `IN_PROGRESS` shared memory and writes those messages in the NVM file. Figure 7 illustrates the logbook elements within SIMA implementation.

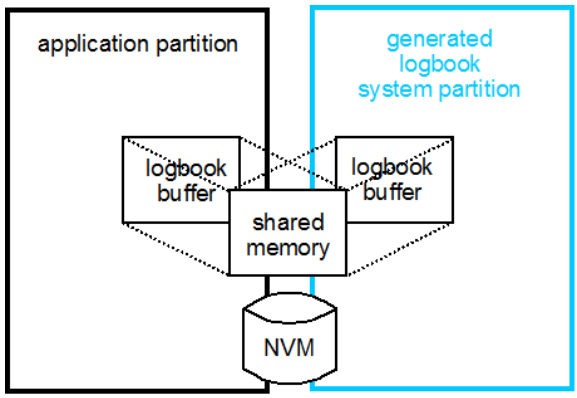

Figure 7 - SIMA Logbook

Within a partition schedule window, messages can be written to the intermediate buffer until this buffer is full. In a second step, at the system partition schedule window, the messages are engraved to the NVM.

The SIMA system integrator must be aware that whenever a logbook is specified within a module, a system partition must also exist in the module. The time spent for accessing the NVM for writing operations is taken from the system partition schedule windows. Enough time should be provided to engrave the messages from the `IN_PROGRESS` buffer.

The position of the system partition schedule windows in the major time frame and the amount of time attributed for its execution determines when the messages are actually engraved; large logbooks (in terms of `IN_PROGRESS` buffer capacity) require longer schedule windows for system partitions or more frequent schedule windows within a major time frame than small logbooks. The messages written to the `IN_PROGRESS` buffer will only change status to `ENGRAVED` after the execution of such system partition. Because ordinary files are used as NVM, the time required for messages to be engraved depends on the time required by the underlying platform to access files.

© GMV, 2010; all rights reserved

Notice that one system partition can be used to engrave messages from all the logbooks in the module (default). The system integrator is responsible for specifying appropriate schedule windows for this partition as it will impact the application partitions behaviour (messages state transition).

Like SIMA ports, SIMA logbooks require information that is not specified in the ARINC configuration. The name for the logbook NVM and a key for the shared memory must be provided within SIMA configuration file. This information (together with the ARINC configuration information) is used by the POS to allocate and initialize the resources (shared memory and files) before the logbook is used.

As illustrated in the listing below, a `DeviceType` node is also specified in the logbook description at SIMA configuration. Currently, only one type is defined; "`file`". The logbook name in the `LogbookName` node must be the same given in the ARINC 653 configuration. The `Logbook` xml node is a child node of `Partition` node within SIMA xml configuration file.

```
<Logbook
        LogbookName="ManagementData"
        NVMName="P2LB1"
        DeviceType="file"
        LogbookKey="60097"/>
</Logbook>
```

SIMA provides a tool for creating: (i) stubs that provide the POS with information from the configuration files (both ARINC 653 and SIMA main configuration), (ii) files required for the logbook NVM and (iii) logbooks system partition. For generating logbook stubs, `makebooks` is used as exemplified below:

```
makebooks <sima_config> <partition> <stub-c-file>
makebooks config/sima.xml "Flight Controls" logbook_stub.c
```

In this example the input for `makebooks` is a SIMA configuration file at (`config/sima.xml`) and the name of logbook owner partition ("`Flight Controls`"). As output, `makebooks` generates the stub for the application partition and the files required by the logbook NVM. The generated stub source code is named according to the third argument provided in the line invoking `makebooks: logbook_stub.c` according to the example.

When used with the parameter `--system`, the `makebooks` tool generates the system partition that engraves the logbook messages:

```
makebooks --system  <sima_config> <partition>
                    <system-partition-c-file>
makebooks --system config/sima.xml "" system_partition.c
```

Notice in the example that the parameter `<partition name>` was given as empty; therefore the generated system partition will engrave messages from all logbooks declared in SIMA and corresponding ARINC 653 configuration. It is possible to generate system partitions to engrave messages from one logbook only by giving this logbook owner partition name as parameter. It can be used for the partition (and the system partition) execution in standalone mode or for distributing the engrave process through different system partitions within the module.

© GMV, 2010; all rights reserved