POK

An ARINC653-compliant operating system released under the BSD licence

Julien Delange, European Space Agency <julien.delange@esa.int>
Laurent Lec, MakeMeReach <lec.laurent@gmail.com>
Introduction
Problems

Safety-critical systems must grant:

- isolation;
- code analysis;
- automatic system configuration.
Approach

POK was designed to provide:

- a configuration and validation tool (1);
- an analysis framework (2);
- time and space isolation (3).
1 – The POK execution platform
Services of kernel and partitions

- Cipher algorithms
- Device drivers code
- Maths functions
- libc
- POSIX
- ARINC653
- Kernel interface
- Tasking
- Intra-partition communications
- Time isolation
- Space isolation
- Inter-partitions communication
- Fault Handling
- Time Management
- Memory Management
- Hardware Abstraction Layer

Partition Level (libpok)
Kernel services

- Hardware abstraction layer
- Memory management
- Time management
- Fault handling
- Time isolation
- Space isolation
- Inter-partition communication
Partition services

- Kernel interface
- Tasking (threads, locking primitives…)
- Libc, POSIX, ARINC653, maths libraries
- Device drivers
- Cipher algorithms
- Intra-partition communication
- C and Ada support
Time isolation

- Partitions are allocated time slices.
- POK offers both partition (RR) and threads scheduling (FIFO, RMS, EDF, LLF).
- Data is flushed at major time frame and is available for the next cycle.
Space isolation

- Partitions are located in their own memory segment, with the help of the MMU.
- Each segment has its properties (address, size, location) defined at configuration time, which cannot be changed at run time.
- Attention is paid to context-switching: all data is either hidden or flushed.
Device drivers

- Device drivers are running in dedicated partitions.
- They must be granted access to hardware by the kernel.
- Partitions can share a device without sharing data (data isolation).
Some parts of the code must or may not be included (device drivers, tasking primitives, etc.).

The more precise the code:

- the smaller the memory footprint;
- the faster the execution.

This requires « #ifdef beaconing » that is, luckily, completely handled – as we’ll see later.
2 – Partitioned system modeling
System design

- Specifications are written in a real (i.e., not C or plain text file) modeling language: AADL.

- These models can be checked against security policies.

- Most of the code for the system is automatically generated from these models; no written code but what the partitions are exactly doing.
What AADL can do

- Define what the system is made of (processor, virtual processor, process, memory, thread, data, subprogram).
- Change their properties (size, permissions, scheduling options...).
- Detect syntax and semantic errors.
- Enforce specifications requirements.
AADL design example

Graphical representation
3 – Code generation
Code is generated from models.

Automated, just run `make`!

POK takes care of the deployment of the system (compile binaries, install them on a device, can run an emulator, attach a debugger...)
#define POK_GENERATED_CODE 1
#define POK_NEEDS_CONSOLE 1
#define POK_NEEDS_LIBC_STDIO 1
#define POK_CONFIG_NB_THREADS 3
...

From AADL to C
4 – Demonstration
Conclusion
To put it in a nutshell

POK is a developing environment for safety-critical hardware which provides:

- numerous kernel and partition ready-to-use services (communication primitives, libc, ARINC and POSIX APIs, fault handling...);

- static analysis on AADL models to check for potential errors and verify the design of the system;

- automatic code generation and deployment from the AADL models and the already provided POK environment.
How to create your own system?

1. Write the specifications of the system in AADL.
2. Write the C code your partitions should run.
3. $ make run
Contact

Information, sources, documentation:

- http://pok.safety-critical.net/

General questions:

- pok@lists.tuxfamily.org

Technical mailing-list:

- pok-devel@lists.tuxfamily.org