ERIKA Enterprise

Current limitations and possible solutions

Version: 7

February 11, 2016

DRAFT

About Evidence S.r.l.

Evidence is a company operating in the field of software for embedded real-time systems. It started in 2002 as a spin-off company of the Real-Time Systems (ReTiS) Lab of the Scuola Superiore Sant'Anna (Pisa, Italy). Today, Evidence is a dynamic company having collaborations in the field of electronics, telecommunications, automotives, and industrial automation.

People at Evidence are experts in the domain of embedded and real-time systems, with a deep knowledge on the design and specification flow of embedded software, especially for the embedded market.

Besides providing consultancy services, Evidence also provides: BSPs based on Linux for embedded devices, evaluation boards featuring most innovative 8, 16 and 32-bit microcontrollers for the embedded market, development tools for making embedded software development easier, and tools for the schedulability analysis of real-time tasks running on your final product.

For more information see: `http://www.evidence.eu.com`

Contact Info

Evidence Srl,
Via Carducci 56
Località Ghezzano
56010 S.Giuliano Terme
PISA Italy

Tel: +39 050 99 11 224
Fax: +39 050 99 10 812

For more information about Evidence products, please send an e-mail to the following address: `info@evidence.eu.com`. Other information about the Evidence product line can be found at the Evidence web site at: `http://www.evidence.eu.com`.

Contents

List of Figures

List of Tables

About this document

This document assembles a set of notes about the current status of ERIKA Enterprise, and how we could change its features to address the new challenges imposed by new multi- and manycore platforms.

Function of the document

It is not yet formal as a requirements document, but it gives an overview of the known issues, trying to give some ideas for improving the current situation.

Document history

Version	Date	Author	Company	Change Description
1	14 Apr 2015	Paolo Gai, Errico Guidieri	Evidence Srl	First version.
2	29 Jul 2015	Paolo Gai	Evidence Srl	Revised version with data structures.
3	31 Jul 2015	Paolo Gai, Pasquale Bonocunto	Evidence Srl, Retis Lab	Tickless timers.
4	7 Aug 2015	Paolo Gai, Errico Guidieri, Giuseppe Serano	Evidence Srl	Refined build procedure, and other minor issues
5	21 Sep 2015	Paolo Gai, Nicholas Merriam	Evidence Srl, Gliwa Gmbh	Notes on MISRA, Context change
6	10 Feb 2016	Paolo Gai	Evidence Srl	RT-Druid
7	11 Feb 2016	Paolo Gai	Evidence Srl	Changed Intro

Acronyms

Acronym	Meaning
ERIKA	The ERIKA Enterprise RTOS, `http://erika.tuxfamily.org`
BSP	Board Support Package

1 Introduction

1.1 A growing ecosystem

ERIKA Enterprise[1] started as a Ph.D. project more that 16 years ago on a single core 16-bit MCU. Nowadays, ERIKA supports more than 15 CPU architectures, 20 compilers, and a number of evaluation boards. All this thanks to the great community of companies and University Labs (among them, a special role goes to the team of the ReTiS Lab of the Scuola Superiore S. Anna) which are using the kernel in real products, University courses, and Research Projects.

Despite the growth, the internal structure of the kernel (the kernel API as well as the makefiles) recalls the structure of the very initial release, which was not thought to support all this variety of hardware platforms and manage the complexity embodied in latest raising platform such as multi-many core. We received over the years comments and critics on how to manage various issues of the current structure, but we never went on on a complete restructuring of the kernel, mainly because of the huge work involved.

1.2 The many-core challenge

However, the recent recent porting of ERIKA Enterprise on multi and manycores (such as the Kalray MPPA port done in the context of the P-SOCRATES FP7 Project), as well as the complexity of the AUTOSAR OS requirements, showed limitations in terms of scalability and efficiency of the current kernel architecture, that cannot be overcome with simple patches.

For that reason, it is now the right time to make a more efficient, safer, and even more beautiful kernel, and for that reason Evidence decided to take the lead of a complete rewrite of the ERIKA Enterprise codebase.

We are very excited of the work we are doing, because it will give the baseline for an increased open-source development for next years!

1.3 We want your comments!

We do not want to do everything alone. We value comments from all users involved in the ERIKA Enterprise development, use, and deployment. For this reason, we want to share with you all comments and critics we received, trying to address them in the next months.

[1] `http://erika.tuxfamily.org`

1.4 This document

Instead of directly addressing the current issues, we thought that it would be best if we could first list all the current problems and proposals for enhancements. In this way, interested users and researchers from the community could, if they want, provide their feedback helping improving the kernel structure.

This document lists all main known issues in the ERIKA Enterprise Kernel structure. It is divided in chapters. Each chapter is dedicated to a specific topic, and it is divided into a description of the current structure, its limitations, and possible ways to overcome them.

Please send all feedback back to the ERIKA forum or directly to Paolo Gai (you can use pj “at” evidence “dot” eu “dot” com).

1.5 Acknowledgments

The work described in this document started from the need of the porting of ERIKA on Kalray MPPA, and it has been funded by the P-SOCRATES Project, [2], FP7-ICT-611016.

[2]http://www.p-socrates.eu

2 Makefiles and file structure

2.1 Current structure

The current file structure was inspired by the structure which was present in the Altera Nios II IDE components.

The current structure is as follows:

- The main directories of the project are:

  `pkg` This is the directory containing the ERIKA Enterprise kernel. All the code inside this directory follows the GPL2+Linking Exception license. All the code is structured in *components*.

  `contrib` This is a repository where third party contributions are stored. The code in this directory may not follow the GPL2+Linking Exception license.

  `testcase` This is a container for the regression tests.

  `examples` This is a container for the examples for the various boards and architectures.

- The `pkg` directory is structured as a first level of directories. The first level directories are:

  `cfg` Makefiles and scripts.

  `cpu` CPU architectures/instruction sets. interrupt disabling/enabling, context change, interrupt handling goes here.

  `mcu` Family of part numbers (e.g., PIC24xxx). Contains basic peripherals (Serial, SPI, I2C, ...), interrupt handling[1]

  `board` Physical boards. Programming LEDs, displays, ... goes her.

  `kernel` All the scheduling infrastructure goes here, including research kernels, OSEK/AUTOSAR implementation

  `test` Assertions used during the testcases.

  Inside these first-level directories, there is one directory per component. The `pkg` directory also has collector include files such as `ee.h`, which are included by all application files.

- Each tail component is mainly composed by following directories:

[1]only for architectures such as ARM7 which does not have an unified interrupt controller along the family.

- `cfg` This directory contains the file `cfg.mk`, which stores the compile instruction for the component. It also stores debug scripts.
- `inc` This directory contains the include files defined by the component.
- `src` This directory contains the `.C/.S` source files.

The configuration file has the following characteristic:

- The number of items defined in the OIL file is exported as a `#define` (example: `EE_MAX_TASK`). These #defines are used inside the code to check the limits of the IDs passed.

The makefile structure is structured as follows:

- Each application has a `makefile`, which is generated from the application OIL file.
- the `makefile` includes a file `rules.mk`, which is a collector that includes architecture-dependent file, typically stored inside the `pkg/cfg` directory.
- Each leaf component has a `cfg.mk` file. All the `cfg.mk` file are included by `rules.mk`.
- The makefile contains a variable `EEOPT`. `EEOPT` lists a set of features. The features are used in two ways:
  - inside the cfg.mk, to select which are the files that needs to be compiled (example, `__PIC30__`);
  - inside the C code, to select specific switches inside the code (example, `__MULTI__`). EEOPT values are translated into `#defines` into the generated file `eecfg.h`.

The compilation process generates a `Debug` directory. The `Debug` Directory contains:

`eecfg.h` The include file which configures the application part of the kernel (how many tasks, task names, ...);

`eecfg.c` The kernel data structures;

`makefile` The makefile;

`obj` A copy of the directory structure of the project, with `.o` (object) and `.d` (makefile dependency) files;

`libee.a` A library containing all the kernel object files (note it does NOT include `eecfg.c`);

ELF file and debug scripts Finally, all the compiled files and debug scripts.

2.2 Limitations of the current structure

Although the file structure presented in Section 2.1 worked well for many years, we found out a number of limitations that suggest a structural change to make the system more usable.

The main requirements come from users who needed to integrate the kernel inside already existing development processes

The following is a short list of them, with a rationale.

1. Too many directories.

   There is often no need to have a `cfg/inc/pkg` directory containing a few files. Also, the path tend to become too long when inside an Eclipse plugin and some compiler stopped working.

2. Include files difficult to be separated from source files.

   To isolate all include files (needed for a binary distribution of the code) you need to run a script.

3. Files contain `#include` directives with path names.

   To store all `.C/.h` files in a single directory you need to change the files to remove the path names.

4. Need for the `-D` directive on the compiler

   The files can be compiled only with the `-D` to add all `EEOPT` switches. This makes compilation more difficult in a generic environment.

5. There is no need to have MCU and board at the main level.

   At the end an MCU/board is useful only within a CPU. With many CPUs/boards supported, the result is somehow confusing.

6. `Debug` directory is misleading for the output of the compilation process.

   In reality `Debug` is more the result of the compilation process. something like `output` or `out` is more suited[2].

7. The ERIKA kernel is stored somewhere else from the application code.

   Local modifications to kernel files for a single application becomes difficult.

8. Listing `APP_SRC` files inside the OIL file on big applications is impractical.

9. Eclipse CDT does not recognize the presence of the include files which are located somewhere else. This results in a number of fake errors provided by the CDT

[2]`out` was the original output directory in the first ERIKA version.

2.3 New structure

To overcome the limitations noted in Section 2.2, we propose the structure defined in the following points:

1. cfg/inc/src directories collapsed.

   Components will be stored in a single directory. include files and source files mixed.

2. no path names in include files.

   That is, instead of:

   ```
   #include ‘‘mcu/msp430/inc/ee_uart.h’’
   ```

   we will have:

   ```
   #include ‘‘ee_uart.h’’
   ```

   which imposes that all filenames will have to be somehow different.

3. Separation between kernel and application configuration.

   - It must be possible to compile the kernel once and link it to different applications (with compatible OIL files)
   - This means also that the kernel must not depend on the number of application objects defined inside the application configuration.
   - `eecfg.[h|c]` will store the application configuration that will contain the definition of constant variables used inside the kernel to check the object IDs

4. MCU/board stored inside the CPU layer

   We will store the MCU/board support under the CPU layer, in a way to cluster the boards together, thus limiting the complexity of the mcu and boards directories only to the mcu/boards which make sense under a given CPU.

5. 2-step build process. See Section 2.3.1.

2.3.1 2-step build process

2.3.1.1 Repository directory structure

mk/pull.mk makefile script which is responsible for copying all files in the final directory

   pull.mk if inside an internal directory there is a pull.mk, then that file will contain the information needed to copy the local files into the workspace

pkg/arch/ARCHNAME one directory per CPU architecture. may have subdirectories to contain what was before named as mcu and board

pkg/kernel/ARCHNAME one directory per scheduler (more or less with the old content)

2.3.1.2 OIL files

The OIL configuration file will be considered as containing two parts:

- Kernel configuration, typically called `os.oil`[3], which includes what is typically stored inside the `OS` OIL object.
- Application configuration, typically called `conf.oil` or `appl.oil`, which includes tasks, alarms, resources, . . .

The application may contain either one OIL file (including both parts) or two OIL files:

Single file In case we have a single OIL file, the two steps are performed in sequence.

Two files In case we have two files, the OS file is processed first to create the Kernel build. Once done, the application build can be done. The application build does not require to remake the Kernel each time.

2.3.1.3 Kernel/BSP Build

The idea is that the Kernel Build will generate a library that allows later linking with one[4] application.

The object of the build is to produce a directory named `erika`[5] (typically at the same level of the OIL file inside the working directory/Eclipse workspace), which contains all information needed to build an application based on the specific kernel OIL file.

The `erika` directory contains a set of subdirectories, each one containing a specific kind of file:

`erika/` The main directory contains license and readme information

`erika/inc` The `inc` directory contains a copy of all include files needed to *use* the ERIKA library[6]. The directory should be possibly without inner hierarchies[7]. When compiling an application, this is the only point where the compiler should point with the `-I` compiler directive. The directory contains also the file `ee_oscfg.h`, which contains the configuration of the kernel.

`erika/src` The `src` directory contains all source files plus internal include files[8] which are needed to compile the ERIKA library. Inner hierarchies are discouraged. The main idea is: everything `.c` inside this directory should be compiled and put inside the ERIKA library. The directory contains also the file `ee_oscfg.c`, which contains the configuration of the kernel.

[3]another name could be `erika.oil`.
[4]...or more (although building the same kernel OIL file with 2 applications is not supported yet).
[5]other possible names could be `os` (but the kernel does not only offer the OS...), or `bsp`
[6]A file only used to compile the kernel can be placed here or (better) in the `src` directory.
[7]Hierarchies could happen in case we need to somehow link third party modules like LWIP. This has to be further analyzed.
[8]These are files needed by the kernel files but which does not export any interface to the final application.

`erika/pull` The `pull` directory contains the `pull.mk`[9] used to pull the `erika/` directory structure from the files in the Eclipse plugins. The `pull.mk` makefile is the only makefile which has a relationship with the ERIKA files location. All other makefiles in the system will work locally.

The directory may also contain a OIL/EMF file[10] which stores the Kernel configuration used to build the package[11].

`erika/out` The `out` directory contains the products of the compilation, that is the dependency files `.d` and the object files `.o`. Other sub-products may be stored here (`.S`, ...). The idea is that all files in this directory can be removed if we want to reduce space in a binary distribution.

`erika/lib` The `lib` directory contains everything binary or source needed to build an application. The directory will contain the compiled libraries (`libee.a` at least, but there could be more in case of external libraries are created in the compilation process), the locator file to be used for the compilation, and eventually some boot code (e.g., `crt0.c`) which may need to be compiled when compiling the application.

`erika/dbg` The `dbg` directory contains all scripts needed for the debugging of the specific platform. Typically these scripts are copied (or parsed) into the application build directory.

`erika/mk` The mk directory contains the makefile needed to compile the library[12], plus an app.mk needed to compile an *application*.

The build will proceed with the following steps:

Step 1 Basic structure

- Input: `os.oil`
- Action: Build Project from Eclipse, equivalent to calling Rt-Druid from command line
- Output: the `erika/pull` directory
  - a copy or export of the OIL file used to generate the structure, into `erika/pull`
  - the `erika/pull/pull.mk`, `erika/pull/ee_oscfg_h.txt`, `erika/pull/ee_oscfg_c.txt` files.
  - additional files (ORTI, ...) generated by RT-Druid

[9]The name of the file is not `makefile` in purpose. The reason is that there will be only one file named `makefile` in the erika directory structure, and will be the makefile used to build the kernel.

[10]We initially thought OIL should be best because human readable; EMF is fine too. . . everything will depend on the tool implementation.

[11]The idea is that the erika directory will be portable without other dependencies with other OIL files.

[12]In this way the library can be remade easily when you do some local modifications, e.g. for quick demos.

  - Those files are the same as the real ones, only with a different extension. In this way, we obtain that all outputs of the Step 1 are included in a single directory.

Step 2 Building the BSP structure

- Input: `erika/pull` directory read only.
- Action: launch the `pull.mk` makefile in `erika/pull`. The target that will be called is named `pull`, which will call in sequence a series of targets.
- Output: `erika/{src|inc|mk|dbg}` directories
  - target `create_src`
    * `erika/src` and `erika/inc` filled with BSP include and C files
    * `erika/pull/ee_oscfg_h.txt` and `erika/cfg/ee_oscfg_c.txt` copied into the `erika/inc` and `erika/src` directories
  - target `create_mk`
    * `erika/mk` directory filled, which will include a `makefile` to compile the library, `app.mk` rules to compile the application, and an OIL file with the configuration.
  - target `create_dbg`
    * `erika/dbg` directory filled
  - At the end of Step 2 the `pull` directory can be deleted if needed. Apart the `pull` directory, all files in under `erika` have local directory names and they do not reference the location where the ERIKA distribution is located.

Step 3 Building the ERIKA library

- Input: `erika/{src|inc|mk|dbg}` directories read only.
- Action: launch the `makefile` in `erika/mk`.
- Output: `erika/{out|lib}` directories
  - All .c files inside `erika/src` are compiled inside `erika/out`
  - A library with all files needed for linking is generated into `erika/lib` (this may include a locator file, a `crt0.c`
  - If there is need to generate debug scripts at this stage, they should be put into `erika/lib as erika/dbg` should be considered *read-only*!

Table 2.1 shows the dependencies on each step of the build process. Lines contains the items required/generated, columns contains the steps in the process.

Description	App edit	Step 1	Step 2	Step 3	Application Build
`os.oil`	w	r	-	-	-
`erika/pull`	-	w	r	d	d
`erika/inc`	-	-	w	r	r
`erika/src`	-	-	w	r	d
`erika/mk`	-	-	w	r	r
`erika/dbg`	-	-	w	-	r
`erika/out`	-	-	-	w	d
`erika/lib`	-	-	-	w	r

Table 2.1: Build process dependencies. Legend: r read-only in this phase; w written in this phase; d can be deleted in this phase; - does not exists yet, or not used.

2.3.1.4 Application Build

The basic idea here is that we build the application after the steps outlined in Section 2.3.1.3 are done. A directory `out` is created and inside that directory everything happens. The application makefiles should be as more linear as possible (in some case they could be derived directly by the IDE used for the compilation.

The structure of the application build is as follows:

`erika/` This is the directory generated at the previous step in Section 2.3.1.3. The hypothesis is that the directories `erika/{inc|mk|dbg|lib}` are present. The other directories (`erika/{pull|src|out}`) are not required and could be not present in case of a "binary distribution" of ERIKA.

`out/` This is the main target directory generated by RT-Druid.

`out/ee_appcfg.{h|c}` These files are generated by RT-Druid and contain the application configuration derived from the `appl.oil` file

`out/makefile` This is the application makefile. The only dependency of this makefile is with the makefile stub stored in `erika/mk`. Important: no dependencies outside the current workspace.

`out/obj` This contains the object files which are produced during the compilation

The build will proceed with the following steps:

Step 4 Basic structure

- Input: `appl.oil`, `erika/mk` OIL configuration files.
- Action: RT-Druid is launched.
- Output: `out/ee_appcfg.{h|c}`, `out/makefile`.

  - RT-Druid will use `erika/mk` OIL files to get the missing information not present in `appl.oil`. If `erika/mk` is not present, the system checks the existence of an `os.oil` file or the `appl.oil` file (in this order[13]) to generate the `erika/` directory as in Section 2.3.1.3.
  - `out/makefile` contains an include reference to the `erika/mk` makefiles.

Step 5 Application compilation

- Input: `out/makefile`
- Action: the makefile `all` target is launched
- Output:
  - The first thing must be a *consistency check*. The system should check whether the date of the configuration file inside erika/mk is greater than both os.oil and appl.oil. In that case, an error is printed and the makefile stops[14].
  - Then, all `.c` files are compiled, using a `-I` directive including all include files present in the `erika/` directory.
  - Libraries, linker and debug scripts should be derived from those present inside `erika/`
  - Finally, Debug scripts should be copied.

2.3.2 Build Environment

2.3.2.1 Common

- *ERIKA_FILES*
  Erika Enterprise Repository Path (relative or absolute)

- *PRJ_BASE*
  Erika Entrprise Project Base Directory Path (relative or absolute)

- *RTD_VER*
  RT-Druid Version Number (chiedere a nicola la sempantica di tale numero).

- CMN_OPT
  Common Build Options. (*CMN_OPT* attribute).

[13] To be checked: is this the best order?

[14] It is out of the scope for the makefile generated by RT-Druid to run automatically RT-Druid. But they should check that what they are compiling is somehow correct at least with the dates.

2.3.2.2 Kernel/BSP

- *OS_BASE*
  Erika Enterprise Base Directory Path (relative or absolute).

- *OS_OPT*
  Erika Enterprise Build Options. (*OS_OPT* attribute).

2.3.2.3 Application

- *APP_BASE*
  Application Output Base Directory Path (relative or absolute).

- *APP_OPT*
  Application Build Options. (*APP_OPT* attribute).

2.3.2.4 Libraries

- *LIB_OPT*
  External Libraries Build Options. (*LIB_OPT* attribute).

2.3.2.5 Arch - AVR8

- *¡Arch¿_MCU* - *AVR8_MCU*
  MCU model for ¡Arch¿ - AVR8 architecture. (*MODEL* attribute of *MCU_DATA* object.)
  - atmega328p: Atmel ATMega328P.

2.3.3 Build Options

2.3.3.1 Common

2.3.3.2 Kernel/BSP

- OS
  Enable Operating System Build.

- OS_DEBUG
  Enable Debug Symbols Generation for OS.

2.3.3.3 Application

- APP
  Enable Application Build.

- APP_DEBUG
  Enable Debug Symbols Generation for Application.

2.3.3.4 Librarires

- LIB
  Enable Operating System Build.
- LIB_DEBUG
  Enable Debug Symbols Generation for Libraries.

2.3.3.5 Arch - AVR8

- TEST

3 Requirements coming from MISRA

3.1 Current structure

The ERIKA Enterprise kernel was developed since 2000 in an incremental fashion. The code structure has changed significantly since then, but it still has a lot from the original architecture design.

3.2 Limitations of the current structure

1. Names starting with EE* are not allowed.
2. Unnecessary levels of #defines makes the code hard to read
3. Type definition and configuration should be done in a simpler way. Names for types have sometime a non-uniform "structure" (e.g., `EE_TYPEPRIO` and `StatusType`)
4. Code with lines longer than 80 chars, with strange characters, with different CRLF conventions, with different indentation
5. Functions should always have a single exit point. This is linked to the usage of TerminateTask inside Task bodies.

3.3 New structure

1. We should try to avoid using symbol names which are prohibited by MISRA, and in particular the EE* symbols (which unfortunately are everywhere in the current code)
2. We should be careful on naming types, and we should define a rule on how types are named
3. We should define a suggested indentation rule, as well as a rule for Emacs/Notepad++/... that enforces it

4 Context change mechanism

4.1 Current structure

The current structure of the context change mechanism in ERIKA can be shortly described as follows:

1. The kernel HAL internally maintain the concept of a stack. A stack is a region of memory which can be used by the running code to save information such as frames of the function calls, temporary variables, and so on.
2. In the *Mono-stack* version of ERIKA, there is only one stack; in the *Multi-stack* version of ERIKA there can be more than one stack, with optionally an additional stack for interrupt frames.
3. Each stack can host a 1 or more tasks, plus the interrupt frames[1]. Tasks are statically allocated to stacks. A task cannot be present more than once on a stack.
4. The typical ERIKA Enterprise Multi-stack configuration allocates all basic tasks on a "*shared* stack", then it allocates one stack for each extended task, and finally there can be an optional stack for interrupt frames.

Figure 4.1.a shows a typical Mono-stack frame configuration for Kernels FP and EDF:

- We suppose task T1 is running.
- Task T1 activates T2 using a call to `ActivateTask`.
- The `ActivateTask` understands that T2 has to preemt T1. It marks the task witch, and then calls a HAL function, which in turns executes T2 as a normal function. The Caller-saved registers are not saved because they are expected to be dirty at the end of the HAL call. The Callee-saved registers are (typically) NOT saved because they will be saved, if needed, by T2 upwards. In other words, the preemption is implemented without explicitly saving registers. This also means that a value in a register in a task may refer to a value currently needed by a preempted task; it is therefore important that all functions in the kernel use the same register convention.
- T2 is interrupted by an IRQ, which in turns activates task T3 preempting T2. In this case, The IRQ handler saves the Caller-saved registers (*IRQ saved* in Figure 4.1.a), marks that T3 is the running task, and finally executes T3.

[1]If they are not allocated to the separate stack in a Multi-stack configuration.

- The colors in 4.1.a separates what part of the stack is property of a task.

Figure 4.1.b shows a typical Mono-stack frame configuration for the OSEK/VDX Kernels. The main difference is how the system handles the implementation of the `TerminateTask` call. In particular, `TerminateTask` is implemented locally to a task. That is, the task body really called by the kernel is not the *real* task body, but is a *thread stub* which saves the Callee-saved registers before calling the real function. This has the advantage to allow an implementation with and without `TerminateTask` by using the same HAL. Again, colors in 4.1.b shows the "property" of the various stack parts. Note that the *Thread stub* and the *Terminate* registers are property of the executing thread.

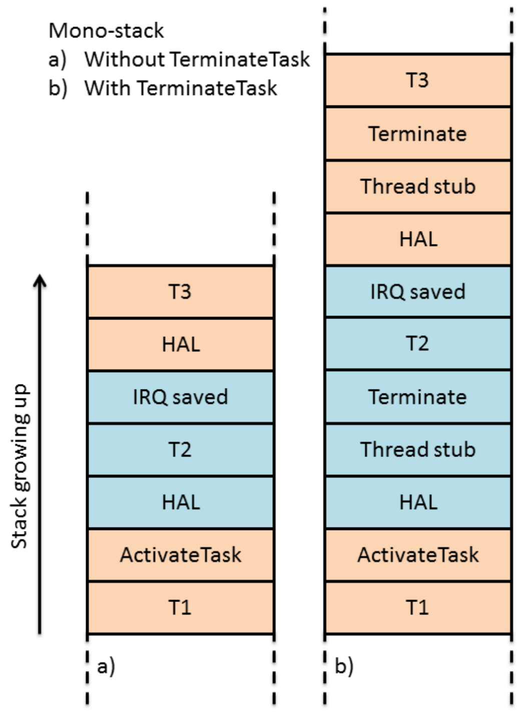

Figure 4.1: ERIKA Enterprise context change structure, monostack.

Figure 4.2 shows the behavior implemented in case of a multi-stack configuration. In particular:

- The main difference is what happens when a task is preempted (synchronously or through an interrupt) by another task.
- In those cases, the *stack* (not the task running into it!) is always left in a known state when the kernel changes the stack. in particular, the system saves the Caller-saved registers (the same saved in the TerminateTask!) on the top of each used stack. Note that at the beginning each stack is initialized having a top of stack which will include some spare space to allocate those Caller-saved registers that will be popped when the specific stack will become the active one. For that reason,

please note the different colors, meaning that the saved registers are property of the *stack* and not of a specific task.

- When an interrupt arrives, after saving the Caller-saved registers, the stack may be changed to the interrupt stack. Subsequent nested interrupts will be allocated on that separate stack. When the last interrupt finishes, the stack is moved back to the preempted task before calling the scheduler.

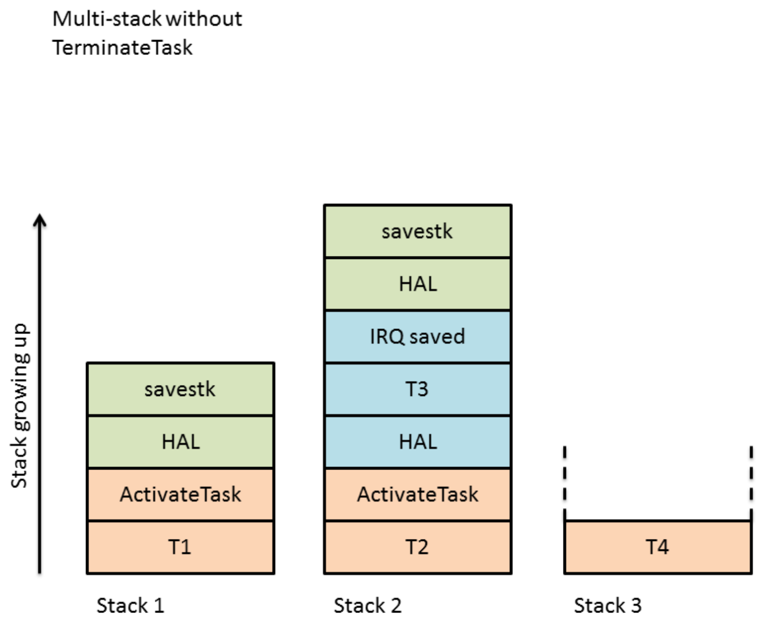

Figure 4.2: ERIKA Enterprise context change structure, multistack.

Figure 4.3 shows the behavior of a multistack configuration with TerminateTask.

4.2 Limitations of the current structure

The current implementation has the following limitations:

- The mono-stack implementation without TerminateTask is highly efficient, but is not the most common one.
- The typical configuration is the one of OSEK/VDX / AUTOSAR OS, with TerminateTask. In that case (see Figure 4.3), Caller-saved registers are typically saved twice (in Figure 4.3, one for *Terminate*, when the task starts, and one for *savestk*, when the task is preempted).
- The TerminateTask always restore the saved registers also when the task is the only one in the stack (in that case, the saved registers could be simply discarded!).

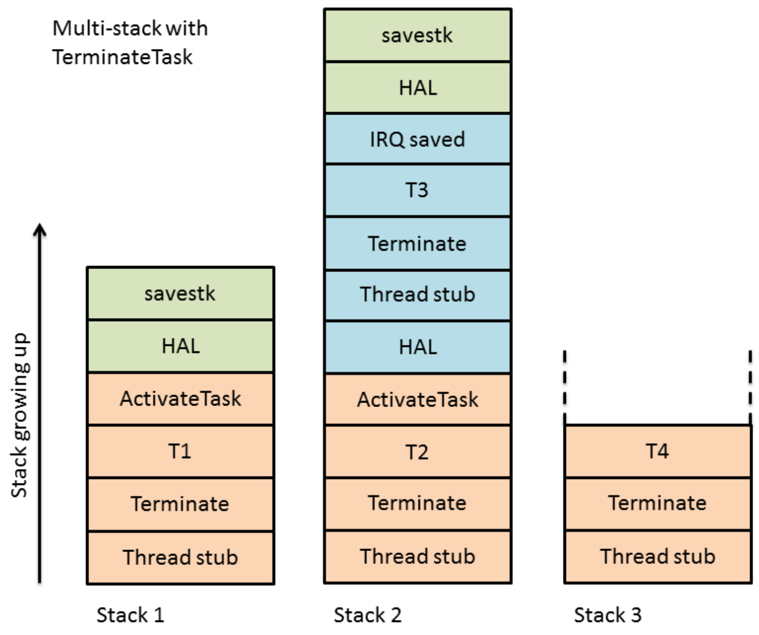

Figure 4.3: ERIKA Enterprise context change structure, multistack with TerminateTask.

4.3 New structure

The main ideas of the new implementation are:

- We try to optimize the case where the system is used as part of OSEK/VDX - AUTOSAR, that is, when used in conjunction with the TerminateTask primitive.
- Mono-stack and Multi-stack versions of the kernel must be supported.
- TerminateTask registers are not saved at the beginning of a task, but are only saved when the task is preempted. In other words, the saved registers are property of the preempted task, and not of the preempting task

Figure 4.4 shows the structure of the context change in the new proposal. In particular:

- Registers are saved only in case of a preemption by a task or IRQ.
- The stack frame which is saved contains a “Previous Top Of Stack (TOS)” pointer to implement the possibility of having more than one task on the same stack.
- Registers are property of the preempted thread, and there are no registers which are property “of the stack” as in Figure 4.3.
- Task termination can be implemented without the need of reloading saved registers as it happens in the old version. The stack frame is simply discarded because the “current TOS” is not modified.

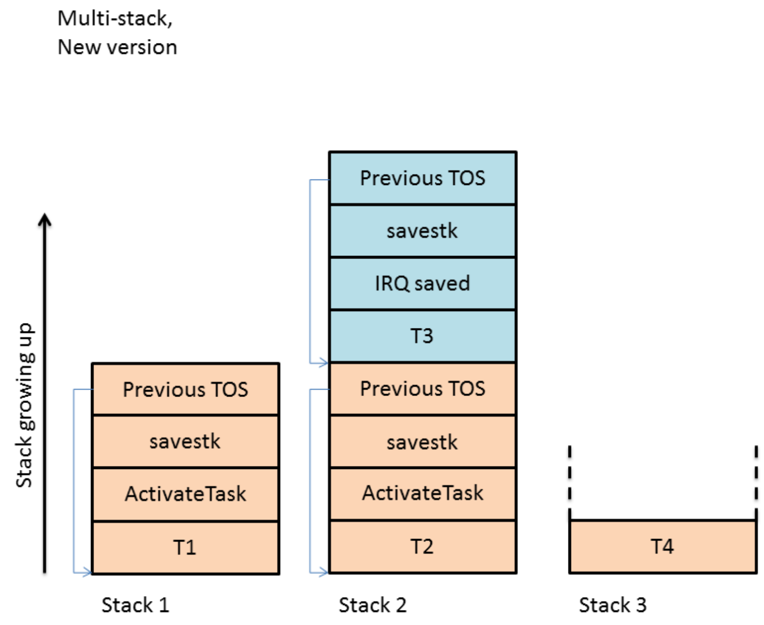

Figure 4.4: ERIKA Enterprise context change structure, new version.

It is possible to implement this context change function with a platform independent HAL version that includes calls to specific functions that will implement the wanted behavior.

We also need to understand if we can do some optimizations with regards to TerminateTask. In particular, the fact that TerminateTask does not return makes it non-MISRA compliant when used in the middle of a task. For that reason, we could have some kind of optimization linked to the fact that we can make the hyphothesis that TerminateTask will be alsoways the *last* thing called at the end of a task. See also Section 3.2 about MISRA compliancy.

5 Multicore support

5.1 Current structure

The Multicore support in ERIKA was initially developed in 2001 and had the following set of features:

- Partitioned scheduling. One copy of the RTOS for each core.
- Compilation flow that is done in steps:
  1. The first CPU, named “master”, is compiled with all the global data structures included;
  2. From the first CPU ELF, a set of symbols are extracted and inserted in the locator files of the other CPUs;
  3. The other CPUs are compiled afterwards with the visibility of the global data structures;

  This was needed to address the fact that early multicore CPUs (such as ST Janus) did not have a homogeneous address space.
- MSRP algorithms, G-T queuing spin-locks implementation, global resources defined by the mapping on the OIL file.

Then, since the support of multicore PPC, we implemented the following features:

- Allocation of shared data using compiler sections.

Afterwards, with the implementation of the AUTOSAR compliance, we implemented the following features:

- MSRP deprecated, because AUTOSAR does not have queuing spin locks. Synchronous RPC implemented instead of asynchronous notifications.

Finally, a Single ELF version of a multicore was done using compiler-specific features of the HighTec compiler. It is not a real Single-ELF version, but more a “big” ELF containing a collage of the other ELFs.

Additional note: given an “object”, the original code of ERIKA had one array per object member, without the usage of structures. The rationale behind this was to save as much RAM as possible avoiding compiler struct packing with padding.

5.2 Limitations of the current structure

- Global addressing disabled to allow sharing when cache coherence is not available (see Altera Nios II implementation);
- Task identifiers implemented as local defines. Their numbers are local on the CPUs, and this may be a problem when sharing a task number inside a global data structure;
- Partitioning of the code based on partitioning .c files. Gets complicated for big projects.
- Does not scale for a large number of CPUs
- Data structures with arrays only limits the efficiency of the code (the compiler has always to load the base address), and its readability

5.3 New structure

5.3.1 Multicore-related things

The new structure tries to overcome the previous implementation with the following features:

- The target architectures are multicore CPUS which have an homogeneous memory address space. Cache coherence mechanisms are not a requirement. Architectures following this approach are for example:
  - Single cores (as a special case of multicores!).
  - PPC like Leopard, Cobra55, K2, ...
  - Tricore AURIX.
- Single-ELF compilation. All code must be compiled for all cores in a single executable. All symbols are visible from all CPUs.
- Since there is only one ELF, Task IDs are global and uniform across cores.
- There is one copy of the kernel code across all the cores.
- All this must be obtained by maintaining application API compatibility with the previous version of ERIKA. Data structures, and internal API may change. We will try to maintain the OIL structure as similar as possible compared to the new version.

5.3.2 Data structures

About the data structures used in the kernel:

- we should give the possibility to share data among cores
- data structures should contain as much data as possible in flash
- pointers should be used with care. Rule of thumb:
  - flash to flash allowed
  - flash to RAM allowed
  - RAM to RAM are not suggested. The suggested way is to use integers to index arrays
- structs should be used instead of arrays to store common data related to the same object.
- we should be careful on sizes of indexes. Probably 2 bytes are enough for all usages. Maybe we should create a type exactly for that.
- Figure 5.1 shows a sketch of the idea behind the data structures. Data structures are named as Descriptor Blocks (DB, in Flash) or Control Blocks (CB, in RAM). Everything can be reached by two global pointers (one for the kernel, one for the CPU layer (HAL)
- This is a short description of the data structures involved:

  KDB Kernel Descriptor Block. Contains the main pointer to all the data structures of the kernel. It stays in Flash, and it should be shared among all CPUs

  CDB/CCB CPU Descriptor/Control Block. It stays in the Kernel. It contains all the information related to the specific CPU (ready queue, ...).

  - In the HAL, there is a pointer to the CPU-specific Kernel data structures. There is an architecture dependent global pointer that the kernel primitive implicitly calls to get the right CDB of the kernel.

  TDB/TCB Task Descriptor/Control block. Stores every info in the Kernel needed about a task

  HDB/HCB HAL Descriptor/Control block. Stores every info about a task abstraction in the HAL

  SDB/SCB Stack Descriptor/Control block. Stores info about a stack. it is separated from HDB/HCB because we want to support stack sharing
- The data structures as they are designed allow the possibility to place CPU dependent content (e.g., all info in RAM about tasks) inside CPU private memories
- Kernel data and HAL data are still separated

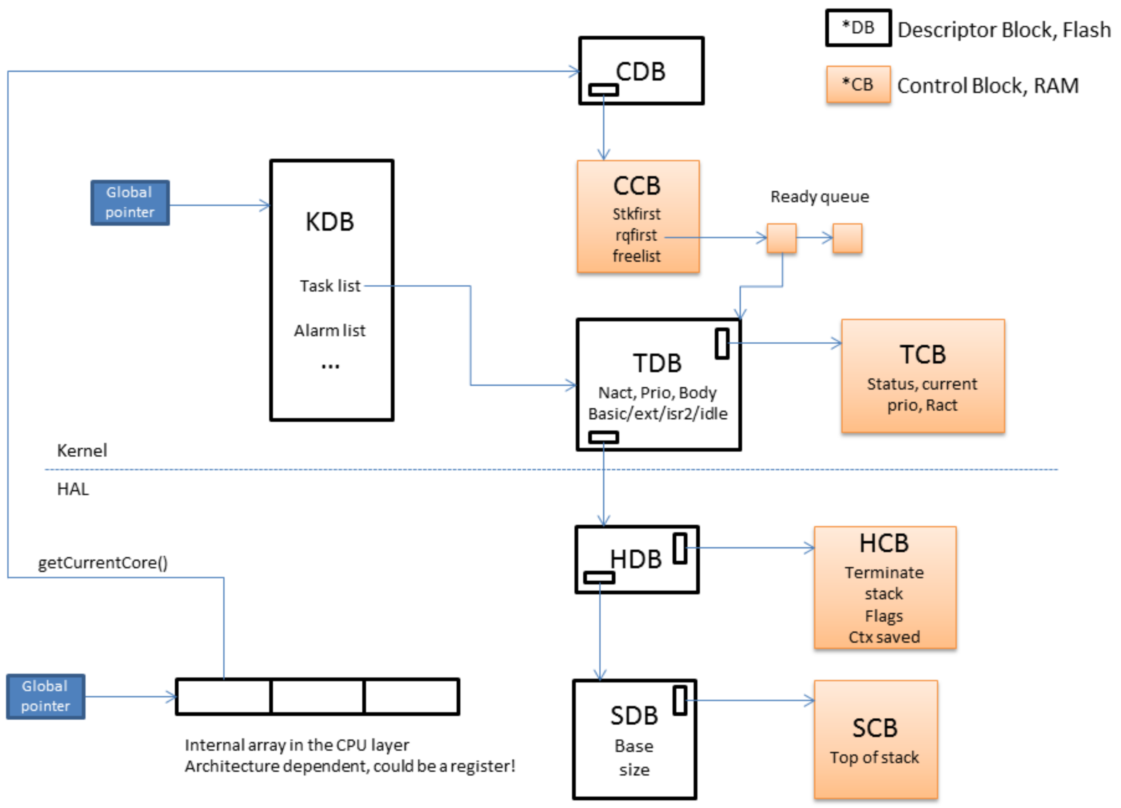

Figure 5.1: Sketch of the data structures.

5.3.3 IRQ and tasks

AUTOSAR specifies tight requirements for the protection levels at which IRQ handlers should run.

The current handling is prone to duplication of code because the same issues about protection applies to both IRQs and Tasks.

Thus, we have here the opportunity to handle IRQ “as tasks”. The idea is to create a proper interrupt handler that in a “fast way” is able to schedule a “special task”, which is put on the stack and executed immediately.

This will add some overhead to the system, but will also simplify the memory protection handling.

6 RT-Druid

6.1 Current structure

One of the limitations that is currently seen in RT-Druid is the fact that it is difficult to modify RT-Druid for adding a modification to the OIL files.

Moreover, the tool is divided in a set of plugins, with code snippets repeated various times in each architecture

6.2 Limitations of the current structure

Adding a modification to the OIL requires either a XSLT transformation, or Java programming, which makes modifications difficult to implement.

6.3 New structure

The idea would be to give the possibility to configure different kernels using templates. We are considering Acceleo as a template language

One of the requirements is to keep Eclipse and EMF as the reference platform, as it is widely used in the industrial/automotive contexts. The idea is then to have two EMF representations of the OIL file, one for the XText editor, one for the Acceleo code generator.

Moreover, we will split the code generator in two parts:

- the engine, which will be an Eclipse plugin, responsible for the generation;
- the configuration files, which will stay inside the ERIKA repository.

7 Other topics

7.1 Kernels more “dynamic”

With “new kernel semantics” we mean situations in which the kernel should be more “dynamic”.

Examples are simil-POSIX APIs, FreeRTOS APIs, and in general APIs where we can dynamically create tasks.

Although not mandatory for AUTOSAR OS implementations, we should take a look at the possibility of supporting those dynamic APIs.

7.2 Tickless

Support for tickless timers in the alarms is welcome. To be understood how to handle it together with `IncrementCounter` AUTOSAR primitive.

7.3 Visualization of the schedule

Various people requested the possibility to have simple tools which could help visualizing the schedule of the tasks in the system. No action has been taken so far on this topic.

Note that commercial tools providing