RT-Druid reference manual

A tool for the design of embedded real-time systems

version: 1.5.0
December 11, 2012

About Evidence S.r.l.

Evidence is a spin-off company of the ReTiS Lab of the Scuola Superiore S. Anna, Pisa, Italy. We are experts in the domain of embedded and real-time systems with a deep knowledge of the design and specification of embedded SW. We keep providing significant advances in the state of the art of real-time analysis and multiprocessor scheduling. Our methodologies and tools aim at bringing innovative solutions for next-generation embedded systems architectures and designs, such as multiprocessor-on-a-chip, reconfigurable hardware, dynamic scheduling and much more!

Contact Info

Address:
Evidence Srl,
Via Carducci 56
Località Ghezzano
56010 S.Giuliano Terme
Pisa - Italy
Tel: +39 050 991 1122, +39 050 991 1224
Fax: +39 050 991 0812, +39 050 991 0855

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

Contents

1. Introduction

This document provides the user with a basic understanding of the architecture, the features and the operations of RT-Druid and the associated Erika Enterprise Kernel.

RT-Druid is now fully integrated with the open source Eclipse framework, as is distributed as a set of plugins that can be installed on top of various Eclipse versions.

This document covers the description of two main components of RT-Druid, which are the code generator plugins for OSEK/VDX systems and the schedulability analysis plugins.

The Code generator plugins consists of a *core* component, required for all operations, and a number of plugins providing time verification and automatic generation of the implementation of real-time embedded software.

The Schedulability analysis plugins provides schedulability analysis support, allowing the user to model the application architecture using the RT-Druid metamodel. Once modeled, the system can be analyzed using the schedulability tests, providing in this way useful information such as task response times, and sensitivity analysis.

1.1. Software design with RT-Druid

The architecture of RT-Druid is shown in Figure 1.1. Model information (i.e. for both the functional and the architecture-level components) is stored in an internal repository and it is made available by means of an open format based on XML.

The toolset architecture is based on a kernel, or Core module, providing management of internal data structure and basic services for GUI and additional plugin modules.

Plugins exploit kernel services in order to provide support to the design stages in a completely independent way. Here is a list of the plugins currently available:

- RT-Druid Modeler;
- RT-Druid Schedulability Analyzer;
- RT-Druid Code generator from OIL/AUTOSAR XML;

1.2. The open architecture of the RT-Druid tool

RT-Druid allows saving all system information in an open XML format. Information about the system model, configuration information and the result of operations performed by plug-in tools, such as schedulability analysis, tracing or debugging info, can easily be made available to external or third party tools. Similarly, OIL files can be imported from or exported to third party products.

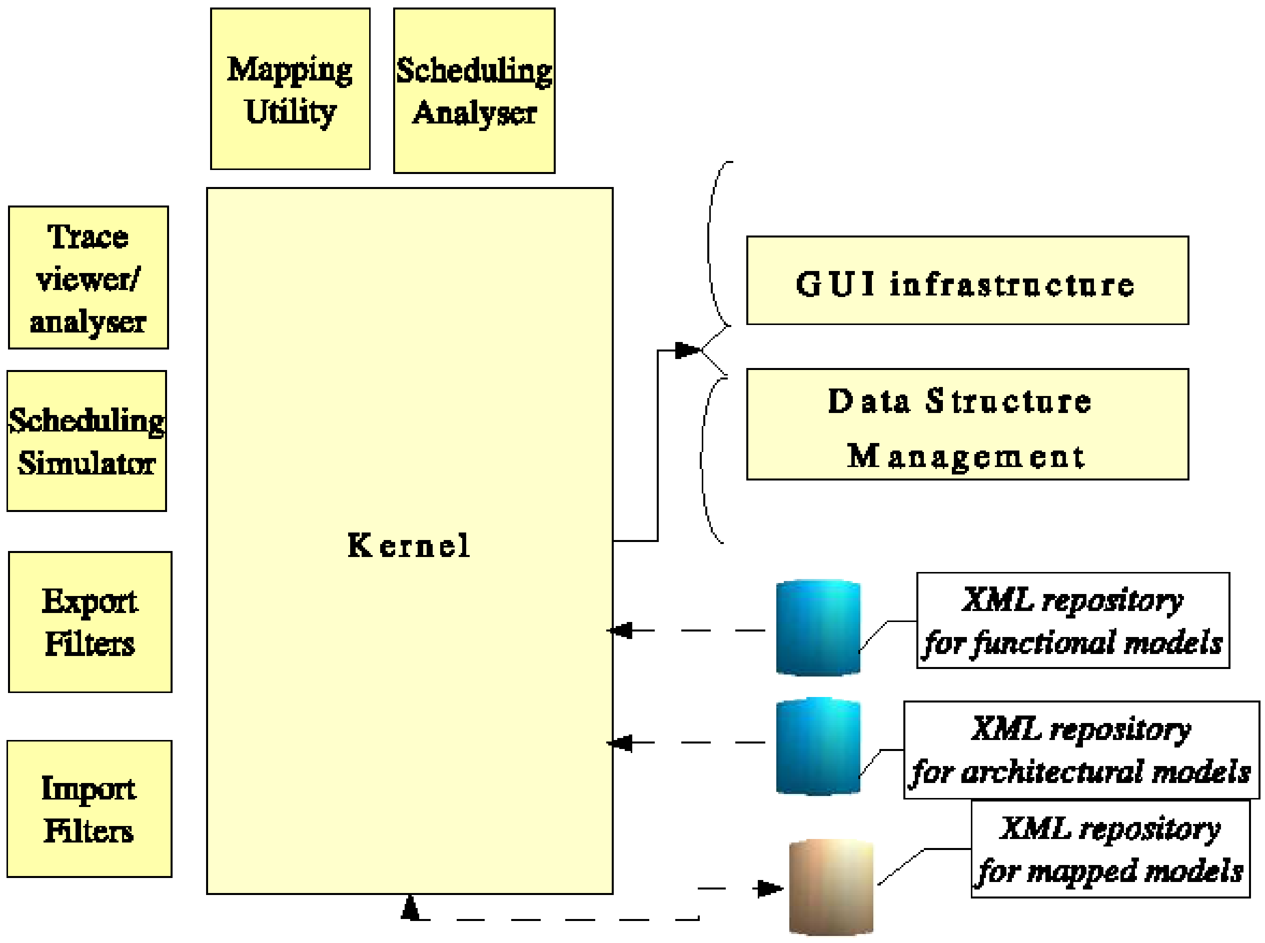

Mapping Utility
Scheduling Analyser
Trace viewer/ analyser
Scheduling Simulator
Export Filters
Import Filters
Kernel
GUI infrastructure
Data Structure Management
XML repository for functional models
XML repository for architectural models
XML repository for mapped models

Figure 1.1.: The plug-in architecture of RT-Druid.

1.3. RT-Druid integration with Eclipse

RT-Druid is entirely written in Java. It is based on well-known development frameworks such as Eclipse, the framework originally propoted by IBM and now released as an open source development environment [3], and on the W3C XML standard. The RT-Druid tool makes use of several Eclipse plug-ins, including EMF [2], GEF [4] and CDT [1][1].

The integration of RT-Druid with the Eclipse framework easily allows any user to perform the operations of editing, compiling, debugging and running the software. The required commands and action sequences are those common to all Eclipse projects, including CDT.

1.4. Content of this document

The document is divided in two parts. The first one is dedicated to the code generator plugins, whereas the second one is dedicated to the schedulability analysis plugins.

The first part about the code generator plugins contains the basic information for operating with the tool and providing the right configuration input for the code generation phase.

In Chapter 2, the code generator plugins are introduced, the code generation process is outlined and the relationships among the products and the Eclipse development environment are explained.

Chapter 3 explains the basic steps that are necessary to start an Rt-Druid project and how to define the basic configuration info that is required by the tool. A fundamental part of the configuration tool is contained in the OIL input file. Syntax and methods for generating the OIL description of the system are the subject of Chapter 4. The Erika Enterprise specific extensions to the OIL language that are necessary to define task placement and other features of multicore systems are described in Section 4.11.

The operations that are required for the code generation phase, together with a detailed description of the input and output data at each step is the subject of the Chapter 5. The kernel configuration and the explanation of the programming model that needs to be used for RT-Druid/Erika Enterprise applications are also described in Chapter 5.

Finally, the first part of the document concludes with a description of the RT-Druid standalone version, which is a command-line version of the code generation plugins, to be used without the Eclipse graphical interface (see Chapter 6).

The second part of the document is related to the Schedulability analysis plugins. The second part starts with Chapter 7, which gives an introduction to the purpose of the Schedulability Analysis plugins. After that, the RT-Druid input file is described in detail (Chapter 8).

Finally, Appendix B contains a description of the ANT scripting support, which enables to run most of the RT-Druid operations batch. Appendix C contains a comprehensive OIL implementation description that can be used as a startup reference.

[1]CDT is the Eclipse component in charge of C/C++ project management.

Part I.

Code Generation Plugins

2. An overview of RT-Druid Code Generator and Erika Enterprise

The RT-Druid Code Generator is an open and extensible environment, based on XML and open standards allowing the generation of configuration code for the Erika Enterprise real-time kernel. The code configuration may start from an OSEK OIL or an AUTOSAR XML definition, to create the configuration code for applications running in a variety of environments.

The code generator plugin is designed aiming at the following general goals:

Modularity: once the kernel module is installed, each design activity in the development flow is in charge of a module that can be separately used as standalone component.

Portability across different execution environments: the tool is designed and implemented in Java for maximum portability to different environments and operating systems (MS Windows, Linux, Macintosh).

Extensibility: The Code Generator tool includes an XSLT transformation engine to easily specify custom modifications to the standard OIL code generator.

Similarly, the operation for creating a new project, as shown in the following Chapter 3 and the editing of the configuration files follow the standard Eclipse pattern. The result of the generation of the configuration file by the RT-Druid wizard and the result of most operations performed by RT-Druid are shown in a dedicated Eclipse console (as happens for most Eclipse plug-ins, for details, please see Section 5).

Integration with Eclipse is not only at GUI interface level, but it also allows performing operations in batch (command line) mode according to the ANT standard [7]. RT-Druid extends the ANT commands (“TASK” in ANT terminology) adding the capability for code generation and the execution of the compilation scripts starting from an OIL file.

2.1. Code generation

The RT-Druid Code Generator is a plugin that is used to automatically generate configuration code at compile time. The steps performed by the Code Generator upon a compilation request are described in this section.

Creation of the build directory and its content Starting from an OIL configuration file, the tool creates a directory that will contain all the generated files. The directory will be the default directory for all the operations of the C/C++ compiler. In the following, we assume that the name selected for this directory in the configuration file is `Debug`.

The first file that is created is the `makefile`, created inside the `Debug` directory itself. The `makefile` is used to compile the application source code. The makefile structure may depend on the final target architecture.

Then, typically the files `eecfg.h` and `eecfg.c` are created, containing the data structures needed to run an Erika Enterprise application. Other files may be present depending on the specific target microcontroller.

Makefile execution After the files are created, RT-Druid automatically runs the `make` command in the `Debug` directory. As a result, the compilation process starts and a set of output files are created.

2.2. Multicore Version

RT-Druid provides special support for the development of multicore applications together with the Erika Enterprise kernel. Currently supported features include:

- Multicore systems with shared memory.
- Support for code placement in a multicore system.

Programming-level implementation is independent from code placement on processors, meaning that the programmer do not need to be aware of the existence of multiple processors. The code generator provides the correct implementation of system primitives based on the placement of the threads and resources as specified in the RT-Druid (OIL) configuration part. Independence from placement options provides:

- Easy testing of different placement configurations.
- Easy extension to a higher degree of parallelism and seamless porting of existing single processor applications

On a multicore system, the main makefile is responsible of triggering the compilation of a separate image for each CPU. In that case, together with the `makefile`, the tool also generates a file `common.mk` (inside the `Debug` directory), containing common makefile settings for the CPUs in the project. Then, one directory is created for each CPU included into the project. The content of that directory is similar to the content of the single core system.

3. Creating an RT-Druid project

Following the standard Eclipse convention, the creation of a new RT-Druid project starts from the wizard for project creation, accessible in several ways, such as, for example, by pressing the “File” button in the menubar and then the “New” and the “Project” buttons in sequence (see Figure 3.1).

After that, the wizard asks for a project template (see Figure 3.2), which is a pre-built application that you can use, and after that for the project name and optionally for the name of the home folder for the project. The use of spacing characters in the project name is **strongly discouraged** and strictly forbidden for the names of all files inside the project folder, since they would create problems with the `make` and `gcc` tools when compiling a project, since (`make` and `gcc` treat spaces as separators inside lists of file names (see Figure 3.3).

Once these steps are completed, the project is created and a OIL configuration file template is automatically generated and inserted into the project.

To edit the OIL File, just double click on it in the Navigation sidebar, and a dedicated OIL Editor will appear.

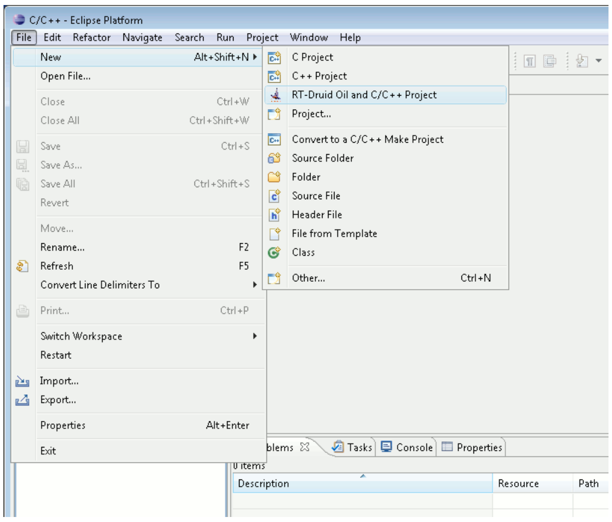
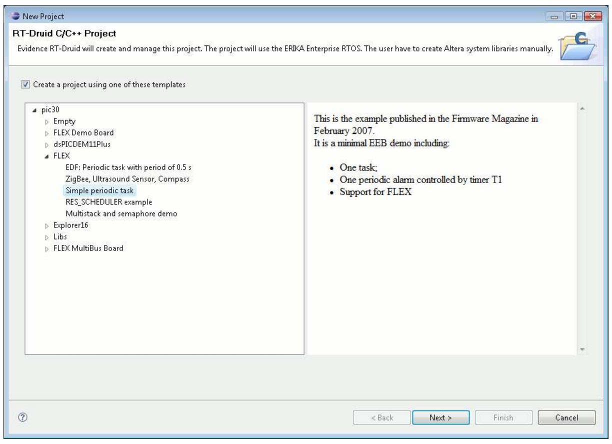


Figure 3.1.: Activating the “New Project” Wizard.



Evidence RT-Druid will create and manage this project. The project will use the ERIKA Enterprise RTOS. The user have to create Altera system libraries manually.

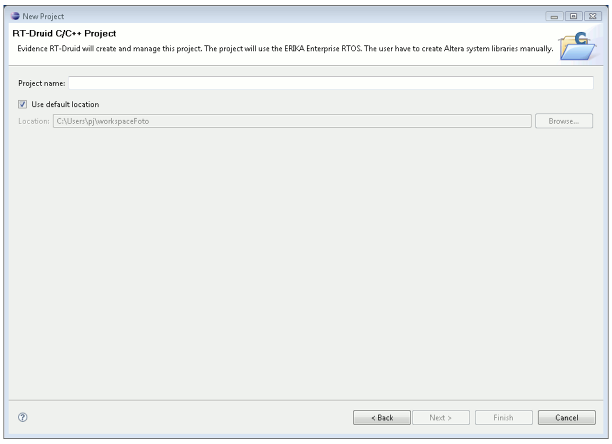
[✓] Create a project using one of these templates

- pic30
  - Empty
  - FLEX Demo Board
  - dsPICDEM11Plus
  - FLEX
    - EDF: Periodic task with period of 0.5 s
    - ZigBee, Ultrasound Sensor, Compass
    - Simple periodic task
    - RES_SCHEDULER example
    - Multistack and semaphore demo
  - Explorer16
  - Libs
  - FLEX MultiBus Board

This is the example published in the Firmware Magazine in February 2007.
It is a minimal EEB demo including:

- One task;
- One periodic alarm controlled by timer T1
- Support for FLEX

Figure 3.2.: Choosing a template application for your new RT-Druid Project.


New Project

RT-Druid C/C++ Project

Evidence RT-Druid will create and manage this project. The project will use the ERIKA Enterprise RTOS. The user have to create Altera system libraries manually.

Project name:

[✓] Use default location

Location: C:\Users\pj\workspaceFoto

Figure 3.3.: Choosing a meaningful name for your new RT-Druid Project.

4. OIL syntax and OIL file generation

OIL (OSEK Implementation Language) is a part of the OSEK/VDX standard, that is used for OS and application configuration. The specification of the OIL file structure and syntax is provided in the OSEK/VDX web site at http://www.osek-vdx.org [5].

In the RT-Druid and in the Erika Enterprise RTOS the configuration of the system is defined inside an OIL file. In this chapter we only provide a quick introduction of the OIL Language (see [5] for a complete description), together with a specification of the specific OIL attributes implemented by RT-Druid.

Standard OIL has no knowledge of multiprocessor systems, nor of distribution of threads and resources. Erika Enterprise provides mechanisms for resource sharing with predictable blocking time in a distributed environment. We defined a set of OIL extensions (see Section 4.11) which explicitly deals with the additional syntax features that are needed for the definition of a multiprocessor system, including placement of threads and resources.

4.1. OIL Basics

In Erika Enterprise all the RTOS objects like tasks, alarms, resources, are static and predefined at application compile time. To specify which objects exist in a particular application, Erika Enterprise uses the OIL Language, which is a simple text description language.

Here is an example of the OIL File for the dsPIC (R) DSC device:

```
CPU mySystem {
        OS myOs {
                EE_OPT = "DEBUG";
                CPU_DATA = PIC30 {
                        APP_SRC = "code1.c";
                        APP_SRC = "code2.c";
                        MULTI_STACK = FALSE;
                        ICD2 = TRUE;
                };

                MCU_DATA = PIC30 {
                        MODEL = 33FFJ256GP710;
                };

                BOARD_DATA = EE_FLEX {
                        USELEDS = TRUE;
                }
                KERNEL_TYPE = FP;
        };

        TASK myTask {
                PRIORITY = 1;
                STACK = SHARED;
                SCHEDULE = FULL;
        };

        TASK myTask {
                PRIORITY = 1;
                STACK = SHARED;
                SCHEDULE = FULL;
        };
};
```

The example contains a single object called CPU, which contains all the specifications and the values used by the system. Inside the CPU, are described the objects which are present in the application: an OS, which specifies the global attributes for the system, and, in the example, two TASKs.

The OIL File is parsed by the RT-Druid code generator and, as a result, part of the RTOS source code is generated and compiled together with the application.

An OIL file consists of two parts: a set of definitions and a set of declarations. Definitions are used to define data types, constants and kernel objects that need to be provided in the declaration part for configuring a specific kernel. In other words, the definition part tells the configurator that there exists different objects like tasks, resources, and so on, describing their attributes and types, like in a C struct declaration. Then, the declaration part is used to specify which objects are really present in a particular application.

In RT-Druid, the definition part of the OIL file is fixed and is contained inside the RT-Druid Eclipse Plugins. The definition part including all the attributes which can be specified by users is included in Appendix C. The user has only to provide the declaration part, specifying for a particular application the objects to be created.

The OIL file basically contains the description of a set of objects. A CPU is a container of these objects. Other objects include the following:

- OS is the Operating System which runs on the CPU. This object contains all the global settings which influence the compilation process and the customization of the Erika Enterprise RTOS.
- APPMODE defines the different application mode. These modes are then used to control the autostart feature for tasks and alarms in the OIL file.
- TASK is an application task handled by the OS.
- RESOURCE is a resource (basically a binary mutex) used for mutual exclusion.

- EVENT is a synchronization flag used by extended tasks.
- COUNTER is a software source for periodic / one shot alarms.
- ALARM is a notification mechanism attached to a counter which can be used to activate a task, set an event, or call a function.

All the attributes in the OIL file can be:

- numbers, i.e. the PRIORITY attribute;
- strings, i.e. the APP_SRC attribute;
- enumerations, i.e. the KERNEL_TYPE attribute.

Attributes can have a default value, as well as an *automatic* value specified with the keyword AUTO. Some of the attributes can be specified more than once in the OIL file, such as the APP_SRC, and the configurator treats them as a *set* of values; i.e., in the case of APP_SRC, the set of application files to be compiled.

Finally, some items can in reality contain a set of sub-attributes, like in a C-language struct definition. For example, CPU_DATA contains a PIC30 object, which is detailed by a set of attributes.

4.2. The CPU Object

The CPU object is only used as a container of all the other objects, and does not have any specific attribute.

4.3. The OS Object

The OS Object is used to define the Erika Enterprise global configuration as well as the compilation parameters.

The attributes which can be specified for the OS object are specified in the following subsections.

4.3.1. Compilation attributes

The OIL file includes a set of fields for controlling the command line parameters which are passed to the compiler tools. The meaning of those elements is the following:

- EE_OPT contains a list of additional compilation flags passed to the Erika Enterprise makefile. In practice, the EE_OPT makefile variable controls which files has to be compiled and with which options. The EE_OPT attributes are translated in #defines in the C code.

- `CFLAGS` contains the list of additional C compiler options.
- `ASFLAGS` contains the list of additional assembly options.
- `LDFLAGS` Contains the list of additional linker parameters.
- `LDDEPS` Contains the list of additional library dependencies which have to be added to the makefile rules.
- `LIBS` Contains the list of additional libraries that needs to be linked.

Example of declaration:

```
CPU mySystem {
  OS myOs {
    EEOPT = "MYFLAG1";
    EEOPT = "MYFLAG2";
    CFLAGS = "-G0";
    CFLAGS = "-O0 -g";
    CFLAGS = "-Wall -Wl,-Map -Wl,project.map";
    ASFLAGS = "-g";
    LIBS = "-lm";
    ...
  };
  ...
}
```

4.3.2. OSEK attributes

The OIL file includes a set of attributes which are part of the OSEK/VDX specification. The meaning of those attributes is the following:

- `STATUS` specifies if the kernel should be compiled with `STANDARD` status or `EXTENDED` status. With the `STANDARD` status, only a subset of the error codes are reported by the kernel primitives to reduce the system footprint. This setting only applies to the OSEK/VDX conformance classes.
- The settings `STARTUPHOOK`, `ERRORHOOK`, `SHUTDOWNHOOK`, `PRETASKHOOK`, `POSTTASKHOOK` specifies which particular hook routine should be included in the kernel.
- `USEGETSERVICEID` specifies if the Service ID debugging functionality of the `ErrorHook()` routine should be included in the kernel.
- `USEPARAMETERACCESS` specifies if the `ErrorHook()` should have access to the parameters passed to the primitives.
- `USERESSCHEDULER` specifies if the kernel includes the `RES_SCHEDULER` resource.

Example of declaration:

```
CPU mySystem {
  OS myOs {
    STATUS = STANDARD;
    STARTUPHOOK = TRUE;
    ERRORHOOK = TRUE;
    SHUTDOWNHOOK = TRUE;
    PRETASKHOOK = FALSE;
    POSTTASKHOOK = FALSE;
    USEGETSERVICEID = FALSE;
    USEPARAMETERACCESS = FALSE;
    USERESSCHEDULER = TRUE;
    ...
  };
  ...
}
```

4.3.3. Multi-core attributes

The attributes `STARTUPSYNC`, and `USEREMOTETASK` are described in the Erika Enterprise Manual for the Altera Nios II target, since they are specific for that architecture.

4.3.4. Nios II target attributes

The attributes `NIOS2_MUTEX_BASE`, `NIOS2_SYS_CONFIG`, `NIOS2_APP_CONFIG`, `IPIC_GLOBAL_NAME`, `IPIC_LOCAL_NAME`, `MP_SHARED_RAM`, `MP_SHARED_ROM`, `NIOS2_DO_MAKE_OBJDUMP`, `SYSTEM_LIBRARY_NAME`, `SYSTEM_LIBRARY_PATH`, `NIOS2_PTF_FILE`, are described in the Erika Enterprise Manual for the Altera Nios II target.

4.3.5. CPU_DATA sections

The `CPU_DATA` section of the `OS` object is used to specify the configuration of a core in a single or in a multiple core device.

In general, the OIL file will contain a `CPU_DATA` section for each core in the system. There is a specific `CPU_DATA` section for each architecture supported by Erika Enterprise.

In particular, the `CPU_DATA` sections currently supported are `NIOSII`, `PIC30` and `AVR_5`, which contain the following attributes:

- `ID` is a symbolic name uniquely identifying the CPU. The name used for the `CPU_ID` attribute must be the same name that is used when allocating objects to a particular CPU.

  CPUs with no name automatically get a default name `default_cpu`. If more than one CPU gets `default_cpu`, an error is raised, because different CPUs cannot have the same name.

`default_cpu` is subsumed also when allocating Tasks (see Section 4.5.10), and counters (see Section 4.8) to a CPU, and when the Master CPU is assigned (see Section 4.11.2).

For single processor systems, it is safe to avoid any declaration of the `CPU_ID` field in the entire OIL file. In this way, all the objects will be mapped to the only CPU in the system, named `default_cpu`.

Example of declaration:

```
CPU mySystem {
  OS myOs {
    CPU_DATA = NIOSII {
      ID = "mycpu";
      ...
    }
    ...
  };
  ...
}
```

- `APP_SRC` declares a list of all files containing code to be executed on the CPU.

  Example of declaration:

```
CPU mySystem {
  OS myOs {
    CPU_DATA = NIOSII {
      APP_SRC = "file1.c";
      APP_SRC = "file2.c";
      ...
    }
    ...
  };
  ...
}
```

- `MULTI_STACK` defines if the system supports multiple stacks for the application tasks (`TRUE`) or not (`FALSE`). The default value is `FALSE`.

  If set to `TRUE`, it is possible to specify if IRQs are executed using a dedicated stack space. The attribute `IRQ_STACK` is used for this purpose.

  Some architectures also allow the specification of a `DUMMY_STACK`, which specifies if the background task is using a shared stack (`SHARED` value) or a dedicated stack segment (`PRIVATE` value). Erika Enterprise schedules the `main()` function as a background task, also called "dummy" task. For example, the Altera Nios II architecture provides support for the above described mechanism, while the dsPIC (R) DSC family does not support it.

- STACK_TOP contains the highest address from which the stack space starts. The address can be provided as a symbol, assuming that the symbol is associated to a value in some other part of the OIL declaration or in some application file. For example, in the Altera Nios II HW version of Erika Enterprise, the typical value for this attribute is __alt_stack_pointer, that is the symbol used inside the Altera Nios II System libraries as the initial stack pointer.

- SYS_SIZE is used to declare the total size of the memory that is allocated to the task stacks.

- SHARED_MIN_SYS_SIZE used to declare the minimum size of the shared stack space. The dimension of the shared stack space is computed as the difference between the available space (SYS_SIZE) and the space required for implementing the private stack spaces. RT-Druid guarantees that the remaining size is higher than or equal to the value defined with the SHARED_MIN_SYS_SIZE directive (an error is raised otherwise). The default value for this attribute is zero.

- The attributes ICD2 and ENABLE_SPLIM are described in the Erika Enterprise Manual for the Microchip PIC24, dsPIC30 (R) DSC and dsPIC33 (R) DSC targets.

- The specific attributes about the AVR_5 architecture are described in the Erika Enterprise Manual for the Atmel AVR5 targets.

Here is an example of a declaration of a Nios II CPU_DATA:

```
CPU mySystem {
  OS myOs {
    CPU_DATA = NIOSII {
      ID = "cpu2";
      MULTI_STACK = TRUE {
        IRQ_STACK = FALSE;
        DUMMY_STACK = SHARED;
      };
      APP_SRC = "cpu2_startup.c";
      STACK_TOP = 0x20004000;
      SHARED_MIN_SYS_SIZE = 1800;
      SYS_SIZE = 0x1000;
      IPIC_LOCAL_NAME = "IPIC_INPUT_CPU0";
    };
    ...
  };
  ...
}
```

The same example can be written in two stages by splitting the declaration of the structure. The only requirement is that the separate declarations do not contain any conflicting assignment to the same field name. The previous example can be rewritten as follows:

```
CPU mySystem {
  OS myOs {
    CPU_DATA = NIOSII {
      ID = "cpu2";
      MULTI_STACK = TRUE {
        IRQ_STACK = FALSE;
        DUMMY_STACK = SHARED;
      };
      APP_SRC = "cpu2_startup.c";
    };

    CPU_DATA = NIOSII {
      ID = "cpu2";
      STACK_TOP = 0x20004000;
      SHARED_SYS_SIZE = 1800;
      SYS_SIZE = 0x1000;
      IPIC_LOCAL_NAME = "IPIC_INPUT_CPU0";
    };

    CPU_DATA = NIOSII {
      /* The ID is not defined, this section refers
         to the "default_cpu" */
      STACK_TOP = "alt_data_end";
    };
    ...
  };
  ...
}
```

4.3.6. MCU_DATA sections

The MCU_DATA section of the OS object is used to specify the configuration of peripherals which are present in a specific microcontroller.

The following microcontrollers are supported:

- Microchip PIC24 microcontrollers and dsPIC DSCs. Please refer to the Erika Enterprise Manual for the Microchip PIC24, dsPIC30 (R) DSC and dsPIC33 (R) DSC targets.

4.3.7. BOARD_DATA sections

The BOARD_DATA section of the OS object is used to specify the configuration of the board where the microcontroller is placed. For example, the board configuration includes the configuration of the external devices like leds, buttons, displays, and other peripherals.

The following boards are supported:

- `NO_BOARD` is a placeholder to specify that no board configuration is required.
- `EE_FLEX` is the Evidence / Embedded Solutions FLEX Board based on the Microchip dsPIC (R) DSC. For details, please refer to the Erika Enterprise Manual for the Microchip PIC24, dsPIC30 (R) DSC and dsPIC33 (R) DSC targets.
- `MICROCHIP_EXPLORER16` is the Microchip Explorer 16 evaluation board. For details, please refer to the Erika Enterprise Manual for the Microchip PIC24, dsPIC30 (R) DSC and dsPIC33 (R) DSC targets.
- `MICROCHIP_DSPICDEM11PLUS` is the Microchip dsPIC Demo Plus 1.1 evaluation board. For details, please refer to the Erika Enterprise Manual for the Microchip PIC24, dsPIC30 (R) DSC and dsPIC33 (R) DSC targets.
- `ATMEGA_STK50X` is the Atmel STK 500 evaluation board for the AVR5 architecture. For details, please refer to the Erika Enterprise Manual for the Atmel AVR5 targets.
- `XBOW_MIB5X0` is the Crossbow MIB 5x0 board used to program wireless sensor network hardware. For details, please refer to the Erika Enterprise Manual for the Atmel AVR5 targets.

4.3.8. Library configuration

Typical microcontroller applications needs to link external libraries to the application code. Erika Enterprise supports both the linking of external binary libraries as well as the development of library code that can be automatically built by the Erika Enterprise build scripts.

Linking an external third-party binary library

A third-party binary library is typically provided as a binary archive with a ".a" extension.

If you need to link this kind of library to your executable, just add the following lines to the OIL file:

```
CPU mySystem {
  OS myOs {
    LDFLAGS = "-Llibrarypath";
    LIBS = "-llibraryname";
  };
};
```

These lines have the effect to add the proper option to tell the linker to load the library you specified.

Building and using libraries which are integrated in the Erika Enterprise build system

In this case, the target is to use the library code which is provided in the Erika Enterprise build tree. Basically, the OIL file can specify a set of libraries which are distributed with or are supported by Erika Enterprise and that have to be linked together with the application. The specification is done by using the `LIB` attribute.

An example of this kind of library are the Scicos library, and other communication libraries which can be found under the `ee/contrib` directory of the Erika Enterprise source tree.

The list of supported libraries depends on the target and can be found in the Erika Enterprise Manual for the specific target.

The `LIB` attribute can be used in one of the following ways:

- This first option helps to build the library files -only-. In particular, `LIB` can be used to specify an OIL file which only compiles the supported libraries (that is, the OIL file is used to configure the libraries but *not* the application). The following example is an OIL file which only compiles the library `mylib.a`:

  ```
  CPU mySystem {
    OS myOs {
      EE_OPT = "__BUILD_LIBS__";
      LIB = ENABLE { NAME = "mylib"; };
      CPU_DATA = PIC30;
    };
  };
  ```

Note: To compile *all* the libraries that are supported by a particular architecture with a single OIL file, the following OIL file configuration can be used:

```
CPU mySystem {
  OS myOs {
    EE_OPT = "__BUILD_ALL_LIBS__";
    CPU_DATA = PIC30;
  };
};
```

- `LIB` can be used for the *on-the-fly* creation of the library during the application compilation process. That is, the build process will create the library as well as the Erika Enterprise library `libee.a`. After that, the application code will be compiled and linked with all the libraries just created. The following example can be used to compile and link the library `mylib.a`.

  ```
  CPU mySystem {
    OS myOs {
  ```

```
    EE_OPT = "__ADD_LIBS__";
    LIB = ENABLE { NAME = "mylib"; };
    ...
  };
  ...
};
```

- In this case, an application will be linked with a library which has been generated using a separate OIL file. The following example shows the OIL file which can be used to link an already existing library which is located in the directory `librarypath`. `librarypath` typically is the `Debug` directory of a project used to build a library, as explained in the first bulled of this list.

```
CPU mySystem {
  OS myOs {
    LDFLAGS = "-Llibrarypath";
    LIB = ENABLE { NAME = "mylib"; };
    ...
  };
  ...
};
```

- Finally, please note that more than one library can be specified in a OIL file in one of the following two ways:

```
CPU mySystem {
  OS myOs {
    LIB = ENABLE { NAME = "mylib1"; };
    LIB = ENABLE { NAME = "mylib2"; };
    LIB = ENABLE { NAME = "mylib3"; };
    ...
  };
  ...
};

CPU mySystem {
  OS myOs {
    LIB = ENABLE {
      NAME = "mylib1";
      NAME = "mylib2";
      NAME = "mylib3";
    };
    ...
  };
  ...
};
```

4.3.9. Kernel Conformance class

An explicit declaration of the kernel conformance class is required in the `KERNEL_TYPE` definition. The definition is shown below:

```
ENUM [
  FP {
    BOOLEAN NESTED_IRQ;
  },
  EDF {
    BOOLEAN NESTED_IRQ;
    STRING TICK_TIME;
    BOOLEAN REL_DEADLINES_IN_RAM = FALSE;
  },
  FRSH {
    ENUM [
      CONTRACT {
        STRING NAME;
        UINT32 BUDGET;
        UINT32 PERIOD;
        STRING CPU_ID;
      }
    ] CONTRACTS[];
    BOOLEAN USE_SYNC_OBJ;
    STRING TICK_TIME;
  },
  BCC1,
  BCC2,
  ECC1,
  ECC2
] KERNEL_TYPE;
```

For the `EDF` and `FRSH` kernels, it is possible to specify the tick length. Given the tick length for the circular timer, which is then used to compute the values to put in the relative deadline task parameter. The tick time can be specified in various unit measures, such as seconds ("s"), milliseconds ("ms"), microseconds ("us"), and nanoseconds ("ns"). Please check the manual for the CPU architecture you are currently using for the proper configuration of the tick parameter. The EDF kernel has an additional option which allows to specify that relative deadlines should be stored in RAM and not in Flash, to allow them to be changed at runtime.

For the `FRSH` kernel[1], a set of contracts have to be specified. For each contract, the user has to specify the contract name, the budget and period (either in clock ticks or using unit measures as for the tick time described above), and the `CPU_ID` (which must only be specified on multicore applications).

For more details on the `FRSH` kernel, please see

Some examples of use within the declaration part are the following:

[1]The FRSH kernel is the result of the IST FP6 FRESCOR Project, `http://www.frescor.org`

1. To configure the BCC1 conformance class:

   ```
   KERNEL_TYPE = BCC1;
   ```

2. To configure the FP conformance class:

   ```
   KERNEL_TYPE = FP;
   ```

   By default, nested IRQs are set to FALSE.

3. To configure the EDF conformance class:

   ```
   KERNEL_TYPE = EDF {
     NESTED_IRQ = TRUE;
     TICK_TIME = "10.5ns";
     REL_DEADLINES_IN_RAM = TRUE;
   };
   ```

   Nested IRQs are set to TRUE.

4. To configure the FRSH conformance class:

   ```
   KERNEL_TYPE = FRSH {
     TICK_TIME = "20ns";
     USE_SYNC_OBJ = TRUE;
     CONTRACTS = CONTRACT {
       CPU_ID = "cpu1";
       NAME = "c1";
       BUDGET = 20000;
       PERIOD = 100000;
     };
     CONTRACTS = CONTRACT {
       CPU_ID = "cpu3";
       NAME = "c7";
       BUDGET = "10ms";
       PERIOD = "50ms";
     };
   };
   ```

   In this case, the FRSH kernel is configured with two contracts on different CPUs. budget and period are specified either with a unit measure or with clock cycles.

4.3.10. ORTI file generation and kernel awareness with Lauterbach Trace32

This section describes the steps to use the Lauterbach Trace32 ORTI support in Erika Enterprise. To generate the ORTI information, the ORTI_SECTIONS attribute has to be specified inside the OS object. The definition of ORTI_SECTIONS is the following:

```
  ENUM [
    NONE,
    ALL,
    OS_SECTION,
    TASK_SECTION,
    RESOURCE_SECTION,
    STACK_SECTION,
    ALARM_SECTION
  ] ORTI_SECTIONS[];
```

Basically, each ORTI section can be selected separately. If `ALL` is specified, then all the ORTI sections are generated.

Notice that the ORTI support currently applies only to the Altera Nios II target for the `BCC1`, `BCC2`, `ECC1`, `ECC2`, `FRSH` conformance classes.

RT-Druid provides the possibility to automatically generate an ORTI[2] file. An ORTI file is basically a text file that specifies which data structures the kernel information are stored in. The file is parsed by an ORTI–enabled debugger, providing useful feedback to the application developer during debug sessions.

Also, RT-Druid automatically produces a set of scripts that can be used to automatically launch a Lauterbach Trace32 debugger [8]. The provided scripts automatically load the FPGA hardware, and start a debug session for each CPU in the system.

To enable all these features, you need to specify a JAM file name[3] inside the OS section of the OIL file, as well as the specification of the ORTI sections that should be generated, as follows:

```
CPU test_application {
  OS EE {
    ...
    NIOS2_JAM_FILE = "JAM_filename.jam";
    ORTI_SECTIONS = ALL;
  }
  ...
}
```

In the above example, `ALL` causes the generation of all the ORTI information, and `JAM_filename.jam` is the path name of the JAM file specified in the `NIOS2_JAM_FILE` attribute. If not specified, `../../fpga.jam` is used.

As a result of the compilation process, a set of files are produced inside the `Debug` directory (see Table 4.1 for a detailed list).

Please refer to Section 5.7 for information on how to use the ORTI Files and launch a Lauterbach Trace32 session.

[2]ORTI is a standard file format specified by the OSEK/VDX consortium.

[3]JAM is one of the file formats containing the FPGA configuration that is accepted by Lauterbach Trace32

File name	Description
`debug.bat`	This batch script loads the FPGA hardware and starts a T32 instance for each CPU. You can double click it on the Nios II IDE to directly launch the debug session.
`debug_nojam.bat`	This batch script starts a T32 instance for each CPU. You can double click it on the Nios II IDE to directly launch the debug session. You can use it if the FPGA has been already programmed with the hardware contents.
`t32.cmm`	Main PRACTICE script, responsible for loading the JAM file and starting all the T32 instances on every CPU.
`testcase_data.cmm`	Internal file used for automatic testcase generation.
`t32/*`	Internal PRACTICE scripts. They are a copy of the files inside `components/evidence_ee/ee/pkg/cpu/nios2/debug/lauterbach/t32`.
`cpuname/config.t32`	Configuration file for T32. Contains the Multicore configuration information.
`cpuname/orti.men`	Trace32 menu automatically generated using the Lauterbach ORTI menu generator.
`cpuname/system.orti`	The ORTI file, for each CPU.
`cpuname/t32.cmm`	The main script file executed by each CPU.

Table 4.1.: Files generated for the Lauterbach Trace32 support. (*cpuname* represents the name of the CPU as specified in the OIL file).

4.4. The Application mode Object

The `APPMODE` object is contained by the `CPU` object and is used to define an application mode.

Example:

```
CPU test_application {
  ...
  APPMODE myAppMode1;
  APPMODE myAppMode2;
  APPMODE myAppMode3;
  ...
}
```

4.5. The Task Object

The `TASK` object is contained inside the `CPU` object and it is used to specify the properties of a task.

4.5.1. Autostart attribute

The `AUTOSTART` attribute specifies if the task should be automatically activated at system startup by the `StartOS()` primitive.

If the task must be activated at startup, the `AUTOSTART` attribute has a value `TRUE`. When `TRUE`, the `APPMODE` sub-attribute lists the application modes for which the task is autostarted.

Example:

```
CPU test_application {
  ...
  TASK myTask1 {
    AUTOSTART = TRUE { APPMODE = myAppMode1; };
    ...
  };
  TASK myTask2 {
    AUTOSTART = FALSE;
    ...
  };
  ...
}
```

4.5.2. Priority attribute

In the FP kernel, the `PRIORITY` attribute specifies the task priority. In the EDF kernel, the value specifies the task preemption level. The value is used by RT-Druid as a relative ordering of priorities and not as an absolute priority value. Higher values correspond to higher priorities.

Example:

```
CPU test_application {
  ...
  TASK myTask1 {
    PRIORITY = 1;
    ...
  };
  ...
}
```

4.5.3. Relative Deadline attribute

The `RELDLINE` attribute specifies the task relative deadline. The value is used by RT-Druid to compute the numerical value of the timing attribute. The value can be expressed as a timing quantity such as seconds ("s"), milliseconds ("ms"), microseconds ("us"), or nanoseconds ("ns"). The value is interpreted as a time, and it is divided by the `TICK_TIME` attribute specified inside the OS attribute `KERNEL_TYPE` to obtain the final tick value which is then programmed inside the microcontroller.

If a number is specified without any time unit (e.g., "1234" and not "1234ms"), then the number is taken "as is" and programmed to the target device.

Please remember that, to be complete, the OIL file should also include a specification of the preemption level of the task by using the `PRIORITY` field.

Example:

The following example specifies the preemption level and the relative deadline of an EDF task.

```
CPU test_application {
  OS myOS {
    ...
    KERNEL = EDF;
  };
  ...
  TASK myTask1 {
    PRIORITY = 3;
    REL_DEADLINE = "10ms";
    ...
  };
  ...
  TASK myTask2 {
    PRIORITY = 4;
    REL_DEADLINE = "1234";
    ...
  };
  ...
}
```

4.5.4. Activation attribute

The `ACTIVATION` attribute specifies the number of pending activations which can be stored by a task. It is only used in the `BCC1`, `BCC2`, `ECC1`, and `ECC2` conformance classes.

Example:

```
CPU test_application {
  ...
  TASK myTask1 {
    ACTIVATION = 3;
    ...
  };
  ...
}
```

4.5.5. Schedule attribute

The `SCHEDULE` attribute specifies if a task is full preemptive (`FULL`) or non preemptive (`NON`). The default is `NON`.

Example:

```
CPU test_application {
  ...
  TASK myTask1 {
    SCHEDULE = FULL;
    ...
  };
  TASK myTask2 {
    SCHEDULE = NON;
    ...
  };
  ...
}
```

4.5.6. Event attribute

The `EVENT` attribute is used to list the Events which belong to a task. It is used in conformance classes `ECC1` and `ECC2`.

Example:

```
CPU test_application {
  ...
  TASK myTask1 {
    EVENT = "TimerEvent";
    EVENT = "ButtonEvent";
    ...
  };
  ...
}
```

4.5.7. Resource attribute

The RESOURCE attribute is used to list the Resources used by a task.

Example:

```
CPU test_application {
  ...
  TASK myTask1 {
    RESOURCE = "Resource1";
    RESOURCE = "Resource2";
    ...
  };
  ...
}
```

4.5.8. Contract attribute

The CONTRACT attribute is used to specify the CONTRACT statically linked to the task. The CONTRACT name must be one of the contracts listed in the FRSH KERNEL_TYPE section.

Example:

```
CPU test_application {
  ...
  TASK myTask1 {
    CONTRACT = "Contract1";
    ...
  };
  ...
}
```

4.5.9. Stack attribute

The STACK attribute is used to specify if the task stack is shared or of the task should have a separate private stack.

Example:

```
CPU test_application {
  ...
  TASK myTask1 {
    STACK = SHARED;
    ...
  };
  TASK myTask2 {
    STACK = PRIVATE {
      SYS_SIZE = 128;
    };
  };
  ...
}
```

4.5.10. Mapping tasks to CPUs using the CPU ID attribute

The `CPU_ID` attribute is used to specify the CPU to which the task is allocated. The placement of the tasks on the CPUs is defined before the compile time and can not be changed during the system execution time. If the CPU identifier is missing, then RT-Druid assumes the default value "default_cpu". An error is generated if the CPU identifier specified for the task does not exist in the system.

Example:

```
CPU test_application {
  ...
  TASK myTask1 {
    CPU_ID = "cpu1";
    ...
  };
  ...
}
```

4.5.11. Source files for each CPU

The source files implementing the tasks can be declared inside the OIL file in a dedicated section named `APP_SRC`. This allows identification of the required files when producing the executable code for each CPU. The `makefile` is automatically generated based on this declaration, so that only the files implementing the task executing on the CPU need to be compiled. If the task is moved to another CPU, the makefile is automatically updated.

Example of declaration:

```
CPU mySystem {
  TASK myTask {
    APP_SRC = "file1.c";
    APP_SRC = "file2.c";
    ...
  }
  ...
}
```

4.5.12. Intertask notifications

The attribute `LINKED` is related to intertask notifications on multicore architectures and is described in the Erika Enterprise Manual for the Altera Nios II target.

4.6. The Resource Object

The `RESOURCE` object is contained inside the `CPU` object and it is used to specify the properties of a resource.

The Resource object contains an attribute named `RESOURCEPROPERTY` which can take the following values:

- `STANDARD` is the default used for a normal resource. In that case, a set of source files can be specified using the `APP_SRC` sub-attribute. These files typically contain the resource data definition.
- `LINKED` resources are only links/alias for other resources.
- `INTERNAL` resources are currently not implemented.

Example:

```
CPU mySystem {
  RESOURCE mutex {
    RESOURCEPROPERTY = STANDARD {
      APP_SRC = "shareddata.c";
    };
  };
  ...
};
```

4.7. The Event Object

The `EVENT` object is used to define a bit mask which then can be used by extended tasks. Events with the same name are identical, and have the same mask. Events with the same mask are not identical. If the value `AUTO` is specified for a mask, then RT-Druid automatically computes the mask value.

Example:

```
CPU mySystem {
  EVENT myEvent1 {
    MASK = 0x01;
  };
  EVENT mtEvent2 {
    MASK = AUTO;
  };
  ...
};
```

4.8. The Counter object

The `COUNTER` object is the timing reference that is used by alarms.
The attributes of a counter are the following:

- `CPU_ID` is an indication of the CPU on which the counter is mapped. The default value is `default_cpu`. If the identifier of the CPU does not exist in the system, an error is generated.
- `MINCYCLE` is currently ignored by Erika Enterprise.
- `MAXALLOWEDVALUE` is currently ignored by Erika Enterprise.
- `TICKSPERBASE` is currently ignored by Erika Enterprise.

Example:

```
CPU mySystem {
  COUNTER myTimer {
    MINCYCLE = 32;
    MAXALLOWEDVALUE = 127;
    TICKSPERBASE = 23;
  };
  ...
};
```

4.9. The Alarm Object

The `ALARM` Object is used to implement an asynchronous notification which can activate a task, set an event or call a callback function. Alarms can be autostarted at boot time depending on the application mode.
The attributes of an alarm are the following:

- `COUNTER` specifies the counter to which the alarm is statically linked.
- `ACTION` specifies the kind of action which has to be implemented when the alarm fires. The action is specified using one of the following sub-attributes:
  - `ACTIVATETASK` specifies that a task has to be activated. The name of the task must be specified inside the `TASK` sub-attribute.

  - `SETEVENT` specifies that an event has to be set on a task. The task name and event must be specified inside the `TASK` and `EVENT` sub-attributes.
  - `ALARMCALLBACK` specifies that an alarm callback has to be called. The name of the callback is specified inside the attribute `ALARMCALLBACKNAME`.

- `AUTOSTART` specifies if the alarm has to be autostarted at system startup. If `TRUE`, the alarm properties and the application modes for which the alarm should be autostarted have to be specified in the sub-attributes `ALARMTIME`, `CYCLETIME`, and `APPMODE`.

4.10. Notes on source files

If a system object (CPU, TASK or RESOURCE) is implemented by more than one source file, all the corresponding file names need to appear in one or more corresponding OIL declarations (not necessarily in order, as shown by the following examples for a CPU, a task and a resource):

```
CPU_DATA = NIOSII {
  ID = "cpu0";
  APP_SRC = "cpu0_src0.c";
};
CPU_DATA = NIOSII {
  ID = "cpu0";
  APP_SRC = "cpu0_src1.c";
  APP_SRC = "cpu0_src2.c";
  APP_SRC = "cpu0_src3.c";
  APP_SRC = "cpu0_src4.c";
};

TASK thread1 {
  CPU_ID = "cpu1";
  APP_SRC = "thread1_a.c";
  APP_SRC = "thread1_b.c";
  APP_SRC = "thread1_c.c";
};

RESOURCE mutex {
  RESOURCEPROPERTY = STANDARD {
    APP_SRC = "res_a.c";
    APP_SRC = "res_b.c";
  };
};
```

It is possible, even if we discourage its use, to list several file names in the same declaration, separated by spaces:

```
APP_SRC = "cpu0_src1.c cpu0_src2.c";
```

If a file name appears more than once, all declarations following the first one are ignored.

4.11. OIL extensions for multiprocessing

4.11.1. Partitioning support

When developing a multiprocessor application, the developer faces the job of mapping a multitask application on the CPUs that are available in the system. That mapping procedure involves the partitioning of the application tasks into the CPUs, meeting all application constraints.

As the starting point, each CPU runs a copy of Erika Enterprise, and all the copies of Erika Enterprise on the CPUs have the same configuration. Depending on the application needs, for example, the kernel can be configured to have monostack or multistack support, which is useful when dealing with blocking primitives, and to support debugging features such as hooks, and extended error status report. All these features are set at the same time for all CPUs. For example, the case where a CPU runs with monostack support while other CPUs run with multistack support is not possible.

From the OIL configuration point of view, the developer defines a set of CPUs in the OIL configuration file using the `CPU_DATA` sections inside the `OS` object, and the job of partitioning consists in placing the OIL objects into the existing CPUs.

The OIL Objects that must be explicitly mapped to a processor are Tasks and Counters. As explained in Sections 4.5.10 and 4.8 the OIL extensions implemented by RT-Druid allow the specification of the CPU to which a `TASK` or `COUNTER` is allocated by using the attribute `CPU_ID`. The link between the particular object (Task or Counter) and the CPU is *static* and specified at compile time, and cannot be changed at runtime.

Other OIL Objects are automatically mapped by the system. In particular, Resources will be *local* if the tasks using them are allocated to the same CPU or *global* otherwise. Alarms are linked to a Counter (that in turn is mapped to a CPU). However, please note that Alarm notifications can result in activation of remote tasks, and setting of events on remote tasks.

Finally, other objects are local and are replicated on each CPU. Hooks and Application Modes fall in this category. This means that each CPU will have its own copy of the hook routines, and each CPU will be initialized passing an appropriate Application Model. It is responsibility of the application developer that all CPUs are initialized with the same Application mode.

4.11.2. Master CPU

When designing multiprocessor systems, the user must specify which CPU acts as "Master CPU"[4]. Here is the definition of the `MASTER_CPU` attribute:

[4]For details about the role of the Master CPU please look at the Multiprocessor Sections of the Erika Enterprise manuals

```
STRING MASTER_CPU = "default_cpu";
```

The default value for the `MASTER_CPU` attribute is `default_cpu`.
Example:

```
MASTER_CPU = "cpu0";
```

4.11.3. Specification of the source files.

In order to make easier the change among different application partitionings, each system object should be implemented in a separate file. Then, each file is specified into the OIL file as the implementation of the corresponding object, such as `CPU_DATA`, `TASK`, `RESOURCE`. RT-Druid uses the information to create the `subdir.mk` files that are used by the makefile scripts to compile the source code.

RT-Druid generates a `subdir.mk` file for each CPU, including all the files that refer to the objects allocated to the CPU.

When a source file is specified inside a `CPU_DATA` element, the file is inserted in the `subdir.mk` of that CPU.

When a source file is specified inside a `TASK` element, the file is inserted in the CPU where the task is allocated.

When a source file is specified inside a `RESOURCE` element, the file is inserted in the CPU where all the tasks using it are allocated in case it is a local resource, or on the Master CPU if it is a global resource.

A source file name can be specified more than once inside the OIL file. However, it will be inserted at most once for each CPU.

To better understand this situation, consider the following example:

```
CPU test_application {

  OS EE {
    MASTER_CPU = "cpu0";

    CPU_DATA = NIOSII {
      ID = "cpu0";
      APP_SRC = "cpu0.c";
    };

    CPU_DATA = NIOSII {
      ID = "cpu1";
      APP_SRC = "cpu1.c";
    };
  };

  TASK task0 {
    CPU_ID = "cpu0";
    APP_SRC = "task0.c";
    RESOURCE = globalmutex;
  };

  TASK task1 {
    CPU_ID = "cpu1";
    APP_SRC = "task1.c";
    RESOURCE = globalmutex;
    RESOURCE = localmutex;
  };

  TASK task2 {
    CPU_ID = "cpu1";
    APP_SRC = "task2.c";
    RESOURCE = localmutex;
  };

  RESOURCE globalmutex {
    RESOURCEPROPERTY = STANDARD { APP_SRC = "globalmutex.c"; };
  };

  RESOURCE localmutex {
    RESOURCEPROPERTY = STANDARD { APP_SRC = "localmutex.c"; };
  };
};
```

cpu0's subdir.mk will include the files cpu0.c, task0.c, and globalmutex.c; cpu1's subdir.mk will include the files cpu1.c, task1.c, task2.c, and localmutex.c.

5. Code generation and the Build process

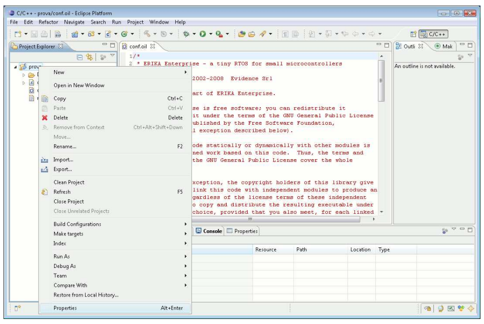
This chapter describes in detail the automatic generation of the configuration code for Erika Enterprise and the build process of an Erika Enterprise application. The process of automatic generation of the configuration code is the method used by RT-Druid to generate Erika Enterprise configuration code starting from an OIL configuration file. The Build Process is the set of operations that are used to compile the application source code together with the configuration source code.

Code Generation and Build Process need to be performed in two separate stages. Each stage can be enabled or disabled by acting on the project properties. To open the Project properties, right click on the project name in the navigation toolbar, and then select "Properties", as shown in Figure 5.1. After that, you can select "Builders" in the left list, and activate only the desired subset of the build process. Please note, that all checkboxes are typically checked, as shown in Figure 5.2.

The first entry in Figure 5.2 is relative to the RT-Druid Plugins shipped with Evidence, whereas the others are part of the CDT [1] plugin.

5.1. Setting the OIL configuration file

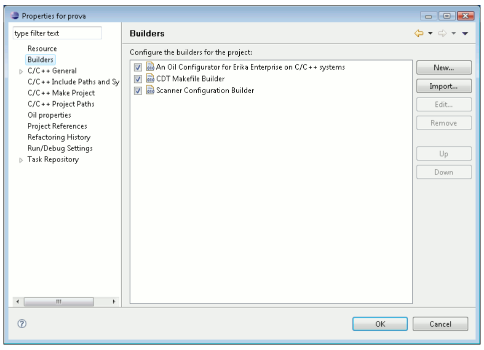
Whenever you need to set or change the OIL file that is used for code generation, you must go to the "Oil properties" tab in the project preference window. Open the Project Preference Window as shown in Figure 5.1, then select the "Oil properties" item in the left list, as shown in Figure 5.3. After, you can type the name of the Oil file in the "File Name" textbox, or you can choose it by pressing the "Browse" button, as shown in Figure 5.4. The file *must* have a `.oil` extension.



```
/*
 * ERIKA Enterprise - a tiny RTOS for small microcontrollers

2002-2008  Evidence Srl

art of ERIKA Enterprise.

se is free software; you can redistribute it
it under the terms of the GNU General Public License
ublished by the Free Software Foundation,
l exception described below).

ode statically or dynamically with other modules is
ned work based on this code.  Thus, the terms and
the GNU General Public License cover the whole

xception, the copyright holders of this library give
link this code with independent modules to produce an
gardless of the license terms of these independent
o copy and distribute the resulting executable under
choice, provided that you also meet, for each linked
```

An outline is not available.

Figure 5.1.: Opening the Project Properties.


Properties for prova

type filter text

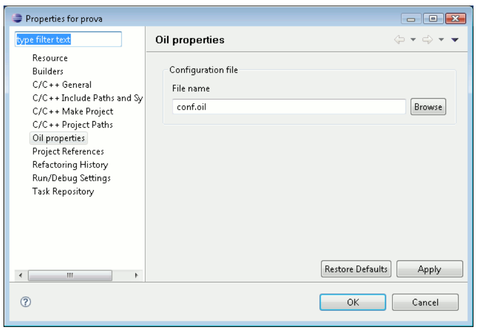
- Resource
- Builders
- C/C++ General
- C/C++ Include Paths and Sy
- C/C++ Make Project
- C/C++ Project Paths
- Oil properties
- Project References
- Refactoring History
- Run/Debug Settings
- Task Repository

Builders

Configure the builders for the project:

- [✓] An Oil Configurator for Erika Enterprise on C/C++ systems
- [✓] CDT Makefile Builder
- [✓] Scanner Configuration Builder

Figure 5.2.: The Builders property page. All checkboxes are selected by default.


Properties for prova

type filter text

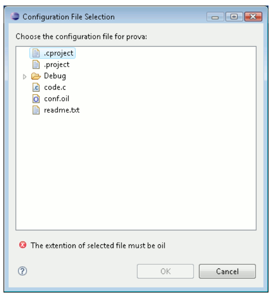
- Resource
- Builders
- C/C++ General
- C/C++ Include Paths and Sy
- C/C++ Make Project
- C/C++ Project Paths
- Oil properties
- Project References
- Refactoring History
- Run/Debug Settings
- Task Repository

Oil properties

Configuration file

File name: conf.oil

Figure 5.3.: Changing the current OIL file for code generation.

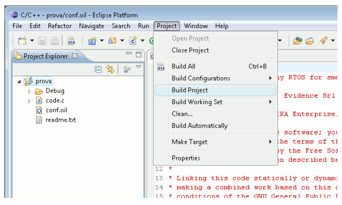

Configuration File Selection

Choose the configuration file for prova:

- .cproject
- .project
- Debug
- code.c
- conf.oil
- readme.txt

The extention of selected file must be oil

Figure 5.4.: Choosing an existing OIL file by browsing the filesystem.



```
ay RTOS for sma
 Evidence Srl
KA Enterprise.
e software; you
the terms of th
by the Free Sof
on described be
*
* Linking this code statically or dynam
* making a combined work based on this c
* conditions of the GNU General Public L
```

Figure 5.5.: The “Build Project” option from the “Project” menu. Please note that the “Build Automatically” option is not selected.

5.2. Starting the Project Build procedure

When the project Build is started, Eclipse executes the selected builders. Figure 5.2 shows the active Builders for a project.

A project Build command can be run explicitly upon user request, or automatically upon saving a file (see the option “Build Automatically” in the “Project” menu). The manual execution of the project Build command can be invoked by right clicking on the project name in the navigation toolbar and then selecting “Build Project”, or directly from the “Project” menu, after having selected the project in the navigation toolbar (see Figure 5.5).

As a first step of the compilation process the RT-Druid builder is launched. In this step the RT-Druid builder generates all the configuration scripts and the source code that is needed for the next steps of the compilation process.

Warning: Be aware that when generating the code, the old build directories are removed and completely overwritten!

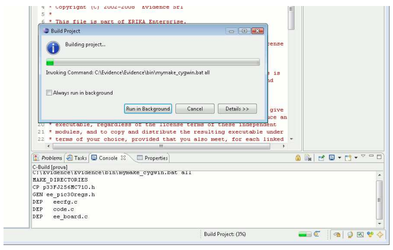
RT-Druid performs a code generation when one of the following conditions apply:

- The OIL configuration file has been modified since the last build.
- The Project properties have been modified, including the case in which the OIL configuration file name has been changed.
- One or more files be generated do not exist anymore.
- One or more files be generated is older than the OIL configuration file.

The modification of the application source files does not imply executing the RT-Druid code generation step.

The RT-Druid code generation is forced every time the OIL configuration file changes or the the project is cleaned as explained in Section 5.5.

After the RT-Druid builder generates the configuration source files and scripts, the CDT builder is executed to compile the application. The CDT Builder executes the make operation on the makefile that has been generated by RT-Druid.



```
Invoking Command: C:\Evidence\Evidence\bin\mymake_cygwin.bat all
```

```
C:\Evidence\Evidence\bin\mymake_cygwin.bat all
MAKE_DIRECTORIES
CP p33FJ256MC710.h
GEN ee_pic30regs.h
DEP   eecfg.c
DEP   code.c
DEP   ee_board.c
```

Figure 5.6.: The Build progress window.

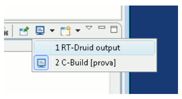


Figure 5.7.: Displaying the RT-Druid Console.

During the entire compilation phase, a progress window is displayed, as shown in Figure 5.6. The compilation can be done in background by clicking on the "Run in Background" button in the progress window.

5.3. RT-Druid Console

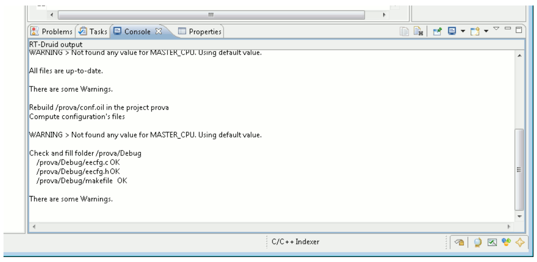
Once the compilation process is completed, or if the compilation is run in background, an RT-Druid console can be used to browse the messages generated during code generation. To do that, the following steps must be performed:

- Select the View "console" that typically appears in the part of the screen after a build command.
- Click on the monitor icon in the console view button list, and then choose "RT-Druid output" (see Figure 5.7).

As a result, a window like the one shown in Figure 5.8 appears. Please note that the text in the window can be selected and copied as normal text.

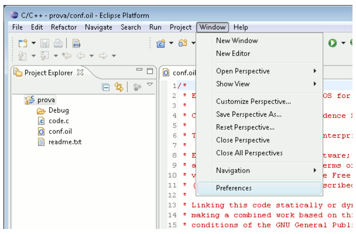
The pop-up menu in Figure 5.8 can be obtained by right clicking on the console. The menu allows to clear the content of the console, to find strings inside it, or to drop the console[1]. Also note that upon a new build the new messages are appended to the existing consoles.

[1]If the console is dropped, a new one will appear at the next build.



```
RT-Druid output
WARNING > Not found any value for MASTER_CPU. Using default value.

All files are up-to-date.

There are some Warnings.

Rebuild /prova/conf.oil in the project prova
Compute configuration's files

WARNING > Not found any value for MASTER_CPU. Using default value.

Check and fill folder /prova/Debug
   /prova/Debug/eecfg.c OK
   /prova/Debug/eecfg.h OK
   /prova/Debug/makefile  OK

There are some Warnings.
```

Figure 5.8.: The RT-Druid console.



```
 * Linking this code statically or dy
 * making a combined work based on th
 * conditions of the GNU General Publ
```

Figure 5.9.: The "Windows/Preferences" menu item.

5.4. Erika Enterprise **Signatures**

The Erika Enterprise kernel may be distributed in binary form. A so called "binary distribution" of Erika Enterprise does not include the kernel C source code, but only with a set of include files and precompiled libraries. The Erika Enterprise code is configured using `#ifdef` directives for efficiency reasons, and each library is the result of the compilation of the Erika Enterprise code with a specific combination of `#define`s.

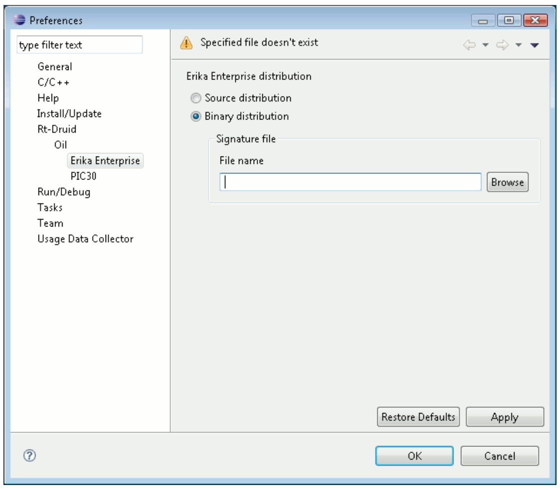
The configurations used when generating the Erika Enterprise libraries are described in the `ee\signature\signature.xml` (for Altera Nios II, the file is inside the `evidence_ee` SOPCBuilder component).

The location of the signature file is contained in the Eclipse preferences, under the "Windows/Preferences" menu (see Figure 5.9). Then, inside the preferences Dialog box, select "RT-Druid/Oil/OS Configurator" (see Figure 5.10).

In this box, the user can select if the code generator must be configured for a source or for a binary distribution of Erika Enterprise. When using a Binary Distribution, the signature file location must be specified. The standard location is "Nios II Devices Drivers/evidence_ee/ee/signature/signature.xml", as shown by Figure 5.10. Figure 5.11 shows the default location of the Erika Enterprise signatures.

If the system is correctly configured the signature file is automatically found by RT-Druid without need of the user intervention.

If RT-Druid is unable to find a library that can be used with the system being generated, an error message is printed on the RT-Druid console, and the Build process is interrupted.



Preferences

type filter text

- General
- C/C++
- Help
- Install/Update
- Rt-Druid
  - Oil
    - Erika Enterprise
    - PIC30
- Run/Debug
- Tasks
- Team
- Usage Data Collector

Specified file doesn't exist

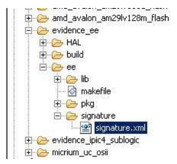
Erika Enterprise distribution

( ) Source distribution
(●) Binary distribution

Signature file

File name

Figure 5.10.: The OS Configurator dialog box.



```
amd_avalon_am29lv128m_flash
evidence_ee
    HAL
    build
    ee
        lib
        makefile
        pkg
        signature
            signature.xml
evidence_ipic4_sublogic
micrium_uc_osii
```

Figure 5.11.: The default location of the signature file.

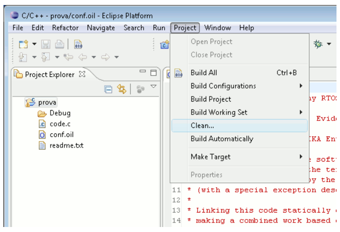
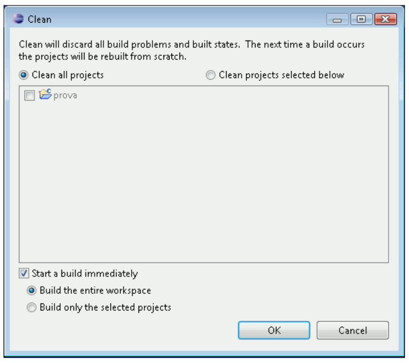


Figure 5.12.: The “Clean...” command on the Project menu.


Clean

Clean will discard all build problems and built states. The next time a build occurs the projects will be rebuilt from scratch.

(●) Clean all projects ( ) Clean projects selected below

[ ] prova

[✓] Start a build immediately

(●) Build the entire workspace

( ) Build only the selected projects

OK Cancel

Figure 5.13.: The Clean dialog box. Please note the bottom left checkbox.

5.5. Project cleanup

RT-Druid provides the feature to clean all the files produced by the code generator. The cleanup process removes, if it exists, the Build Directory.

To clean the project, just select the “Clean...” command inside the “Project” menu. The project need not be be selected if the command is issued after selecting the project itself.

Please note the checkbox on the bottom left that can be used to build the project right after the cleanup has been completed (see Figures 5.13 and 5.13).

Figures 5.14 and 5.15 shows the Navigation toolbar “C/C++ projects”, before and after a project clean. The `Debug` directory have been removed, together with the “Binary” object list. During the cleanup, some errors may be shown in the Problem window. They can be ignored, since they refer to files that have been removed and that will be automatically re-created by RT-Druid at Build time.

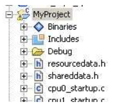
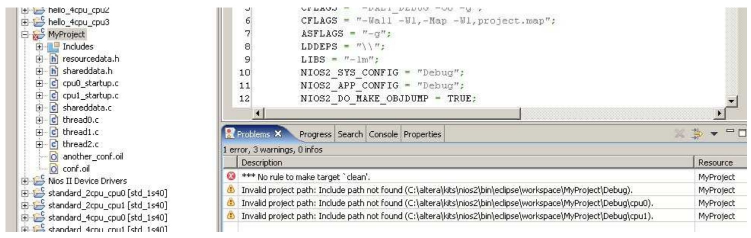


Figure 5.14.: The Navigation toolbar before the Project clean.



```
CFLAGS = "-Wall -Wl,-Map -Wl,project.map";
ASFLAGS = "-g";
LDDEPS = "\\";
LIBS = "-lm";
NIOS2_SYS_CONFIG = "Debug";
NIOS2_APP_CONFIG = "Debug";
NIOS2_DO_MAKE_OBJDUMP = TRUE;
```

1 error, 3 warnings, 0 infos

Description	Resource
*** No rule to make target `clean'.	MyProject
Invalid project path: Include path not found (C:\altera\kits\nios2\bin\eclipse\workspace\MyProject\Debug).	MyProject
Invalid project path: Include path not found (C:\altera\kits\nios2\bin\eclipse\workspace\MyProject\Debug\cpu0).	MyProject
Invalid project path: Include path not found (C:\altera\kits\nios2\bin\eclipse\workspace\MyProject\Debug\cpu1).	MyProject

Figure 5.15.: The Navigation toolbar after the Project clean. The errors in the Problem window can be ignored, because they refer to files that have been removed and that will be automatically created by RT-Druid at Build time.

5.6. Application Debug and Run

This section explicitly refers to the Altera Nios II target.

As a result of the compilation process, one ELF file for each CPU will be placed inside the build directory. In the case of Altera Nios II, these ELF files are equivalent to those built by traditional Altera Build Scripts. To Debug and Run them, the best option is the creation of a "Multiprocessor collection", as explained in the Altera documentation [6].

5.7. Application debug using Lauterbach Trace32

This section explicitly refers to the Altera Nios II target.

This section shows how to run a Lauterbach debug session using the Trace32 scripts and ORTI files automatically generated by RT-Druid (see Section 4.3.10 for more information).

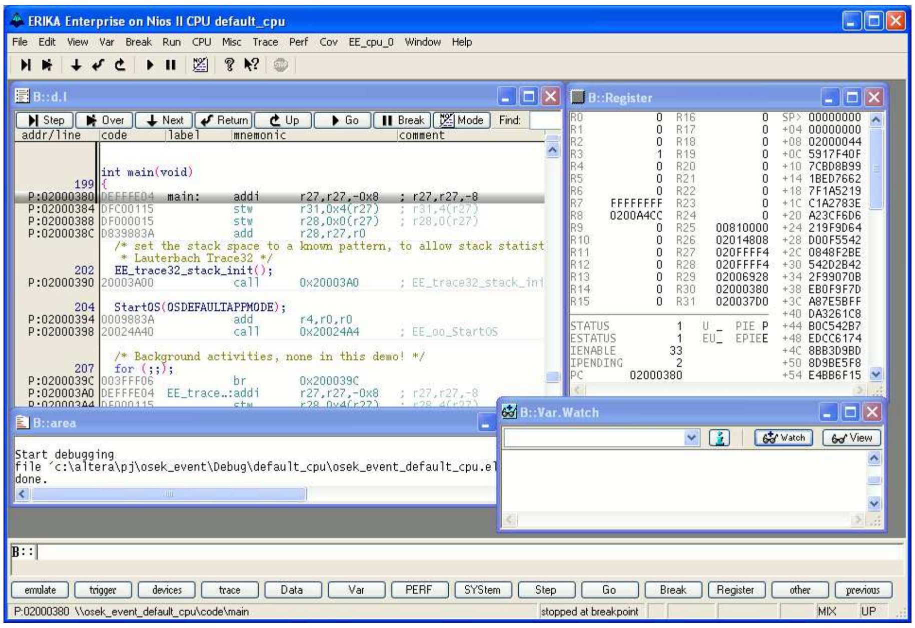
Warning: The scripts generated by RT-Druid suppose Lauterbach Trace32 being installed in `C:\t32`, and the Lauterbach ORTI utility `genmenu.exe` being installed insie `C:\T32\demo\kernel\orti`.

Once the application has been compiled, and the Trace32 scripts have been generated, launch Trace32 by double clicking on the `Debug/debug.bat` file generated during the compilation. The debugger opens up showing a window similar to the one in Figure 5.16.

Please note that each window has a title with the name of the CPU being under debug. The menu list include a submenu named "ee_cpu_0" containing the specification of the ORTI related debug features.

By clicking on each menu item, you can get useful debug informations about Erika Enterprise. In particular:

- Figure 5.17 shows the general information about the kernel global variables, such as the name of the running task, the current priority of the running task, the last


ERIKA Enterprise on Nios II CPU default_cpu
File Edit View Var Break Run CPU Misc Trace Perf Cov EE_cpu_0 Window Help
B::d.l
Step Over Next Return Up Go Break Mode Find:
B::Register
B::area
B::Var.Watch
Watch View
emulate trigger devices trace Data Var PERF SYStem Step Go Break Register other previous

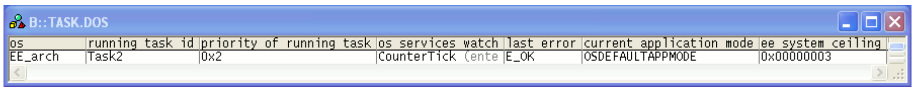
```
addr/line   code      label      mnemonic             comment

            int main(void)
        199 {
P:02000380  DEFFFE04  main:      addi    r27,r27,-0x8    ; r27,r27,-8
P:02000384  DFC00115             stw     r31,0x4(r27)    ; r31,4(r27)
P:02000388  DF000015             stw     r28,0x0(r27)    ; r28,0(r27)
P:0200038C  D839883A             add     r28,r27,r0
              /* set the stack space to a known pattern, to allow stack statist
               * Lauterbach Trace32 */
        202   EE_trace32_stack_init();
P:02000390  20003A00             call    0x20003A0       ; EE_trace32_stack_ini

        204   StartOS(OSDEFAULTAPPMODE);
P:02000394  0009883A             add     r4,r0,r0
P:02000398  20024A40             call    0x20024A4       ; EE_oo_StartOS

              /* Background activities, none in this demo! */
        207   for (;;);
P:0200039C  003FFF06             br      0x200039C
P:020003A0  DEFFFE04  EE_trace..:addi    r27,r27,-0x8    ; r27,r27,-8
```

```
R0          0   R16         0   SP> 00000000
R1          0   R17         0   +04 00000000
R2          0   R18         0   +08 02000044
R3          1   R19         0   +0C 5917F40F
R4          0   R20         0   +10 7CBD8B99
R5          0   R21         0   +14 1BED7662
R6          0   R22         0   +18 7F1A5219
R7   FFFFFFFF   R23         0   +1C C1A2783E
R8   0200A4CC   R24         0   +20 A23CF6D6
R9          0   R25  00810000   +24 219F9D64
R10         0   R26  02014808   +28 D00F5542
R11         0   R27  020FFFF4   +2C 0848F2BE
R12         0   R28  020FFFF4   +30 542D2B42
R13         0   R29  02006928   +34 2F99070B
R14         0   R30  02000380   +38 EB0F9F7D
R15         0   R31  020037D0   +3C A87E5BFF
                                +40 DA3261C8
STATUS         1   U _   PIE P  +44 B0C542B7
ESTATUS        1   EU_   EPIEE  +48 EDCC6174
IENABLE       33                +4C 8BB3D9BD
IPENDING       2                +50 8D9BE5F8
PC      02000380                +54 E4BB6F15
```

```
Start debugging
file 'c:\altera\pj\osek_event\Debug\default_cpu\osek_event_default_cpu.el
done.
```

```
B::
P:02000380 \\osek_event_default_cpu\code\main        stopped at breakpoint        MIX  UP
```

Figure 5.16.: The Lauterbach Trace32 for Altera Nios II.

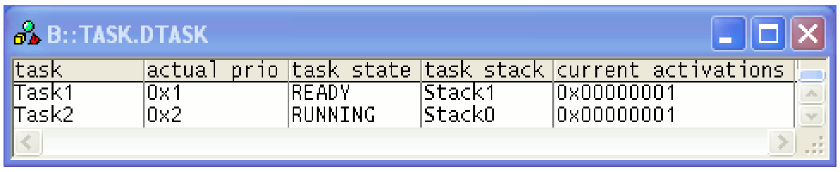


os	running task id	priority of running task	os services watch	last error	current application mode	ee system ceiling
EE_arch	Task2	0x2	CounterTick (ente	E_OK	OSDEFAULTAPPMODE	0x00000003

Figure 5.17.: General information about the Erika Enterprise status.

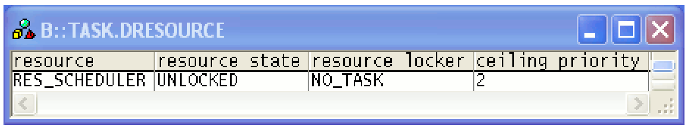
RTOS primitive called, the last error returned by an Erika Enterprise primitive, the current application mode and the current system ceiling.

- Figure 5.18 shows, for each task, the task name, its current priority, which may be different from the nominal priority when the task lock a resource, the task state, the task stack, and the current pending activations.
- Figure 5.19 shows, for each resource, the resource name, the resource status, the task that has locked the resource (if any), and the ceiling priority of the resource.



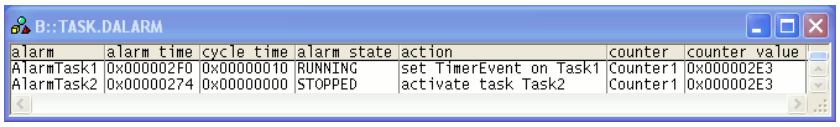
task	actual prio	task state	task stack	current activations
Task1	0x1	READY	Stack1	0x00000001
Task2	0x2	RUNNING	Stack0	0x00000001

Figure 5.18.: Information about the tasks in the system.



resource	resource state	resource locker	ceiling priority
RES_SCHEDULER	UNLOCKED	NO_TASK	2

Figure 5.19.: Information about the resources in the system.

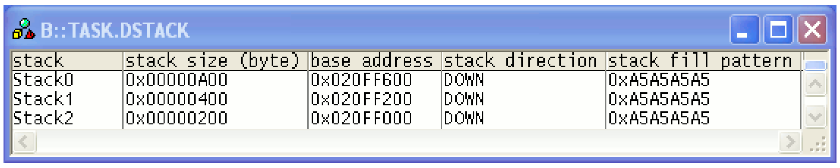


alarm	alarm time	cycle time	alarm state	action	counter	counter value
AlarmTask1	0x000002F0	0x00000010	RUNNING	set TimerEvent on Task1	Counter1	0x000002E3
AlarmTask2	0x00000274	0x00000000	STOPPED	activate task Task2	Counter1	0x000002E3

Figure 5.20.: Information about the alarms in the system.

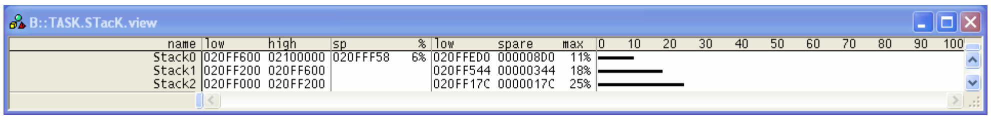
- Figure 5.20 shows, for each alarm in the system, the alarm name, the time to which the alarm will fire, the cycle time of the alarm (`0x0` means the alarm is not cyclic), the alarm state, the action linked to the alarm notification, the counter to which the alarm is attached, and its value.
- Figure 5.21 and Figure 5.22 show information about the stacks that has been configured in the application. In particular, the first figure shows the stack name, size, base address, direction, and fill pattern, while the second figure shows in a graphical way the current stack usage. To obtain the graphical stack usage estimation the application has to call `EE_trace32_stack_init` at system startup. In this example, `Stack0` is the shared stack used by the background task (the `main` function), and by `Task2`. `Stack1` is used by `Task1`, and `Stack2` is the interrupt stack.
- Finally, whereas the previous figures were relative to the ORTI support provided for the `BCC1`, `BCC2`, `ECC1`, `ECC2` kernels, Figure 5.23 and Figure 5.24 show the ORTI support for the `FRSH` kernel. In particular, the first figure shows the ORTI support for contracts, VRES, and budgets which are entities specifically introduced for the FRSH kernel, whereas the second figure shows the context changes as interpreted by Trace32 considering the value changes of the running task pointer of the kernel logged using the Lauterbach tracer module.

The RT-Druid and Erika Enterprise Trace32 support also includes support for the Nios



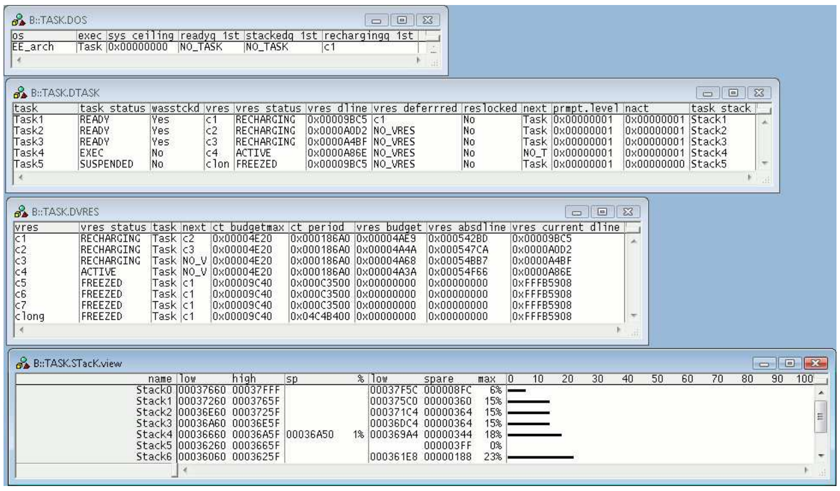
stack	stack size (byte)	base address	stack direction	stack fill pattern
Stack0	0x00000A00	0x020FF600	DOWN	0xA5A5A5A5
Stack1	0x00000400	0x020FF200	DOWN	0xA5A5A5A5
Stack2	0x00000200	0x020FF000	DOWN	0xA5A5A5A5

Figure 5.21.: The application stack list.



```
B::TASK.STacK.view
   name low      high     sp        % low      spare    max 0  10  20  30  40  50  60  70  80  90  100
 Stack0 020FF600 02100000 020FFF58 6% 020FFED0 000008D0 11%
 Stack1 020FF200 020FF600             020FF544 00000344 18%
 Stack2 020FF000 020FF200             020FF17C 0000017C 25%
```

Figure 5.22.: A graphical view of the application stack usage.



```
B::TASK.DOS
os       exec sys ceiling readyq 1st stackedq 1st rechargingq 1st
EE_arch  Task 0x00000000  NO_TASK    NO_TASK      c1
```

```
B::TASK.DTASK
task   task status wasstckd vres vres status vres dline vres deferrred reslocked next prmpt.level nact       task stack
Task1  READY       Yes      c1   RECHARGING  0x00009BC5 c1             No        Task 0x00000001  0x00000001 Stack1
Task2  READY       Yes      c2   RECHARGING  0x0000A0D2 NO_VRES        No        Task 0x00000001  0x00000001 Stack2
Task3  READY       Yes      c3   RECHARGING  0x0000A4BF NO_VRES        No        Task 0x00000001  0x00000001 Stack3
Task4  EXEC        No       c4   ACTIVE      0x0000A86E NO_VRES        No        NO_T 0x00000001  0x00000001 Stack4
Task5  SUSPENDED   No       clon FREEZED     0x00009BC5 NO_VRES        No        Task 0x00000001  0x00000000 Stack5
```

```
B::TASK.DVRES
vres   vres status task next ct budgetmax ct period  vres budget vres absdline vres current dline
c1     RECHARGING  Task c2   0x00004E20   0x000186A0 0x00004AE9  0x000542BD    0x00009BC5
c2     RECHARGING  Task c3   0x00004E20   0x000186A0 0x00004A4A  0x000547CA    0x0000A0D2
c3     RECHARGING  Task NO_V 0x00004E20   0x000186A0 0x00004A68  0x00054BB7    0x0000A4BF
c4     ACTIVE      Task NO_V 0x00004E20   0x000186A0 0x00004A3A  0x00054F66    0x0000A86E
c5     FREEZED     Task c1   0x00009C40   0x000C3500 0x00000000  0x00000000    0xFFFB5908
c6     FREEZED     Task c1   0x00009C40   0x000C3500 0x00000000  0x00000000    0xFFFB5908
c7     FREEZED     Task c1   0x00009C40   0x000C3500 0x00000000  0x00000000    0xFFFB5908
clong  FREEZED     Task c1   0x00009C40   0x04C4B400 0x00000000  0x00000000    0xFFFB5908
```

```
B::TASK.STacK.view
   name low      high     sp        % low      spare    max 0  10  20  30  40  50  60  70  80  90  100
 Stack0 00037660 00037FFF             00037F5C 000008FC  6%
 Stack1 00037260 0003765F             000375C0 00000360 15%
 Stack2 00036E60 0003725F             000371C4 00000364 15%
 Stack3 00036A60 00036E5F             00036DC4 00000364 15%
 Stack4 00036660 00036A5F 00036A50 1% 000369A4 00000344 18%
 Stack5 00036260 0003665F                      000003FF  0%
 Stack6 00036060 0003625F             000361E8 00000188 23%
```

Figure 5.23.: The ORTI windows for the FRSH Kernel.

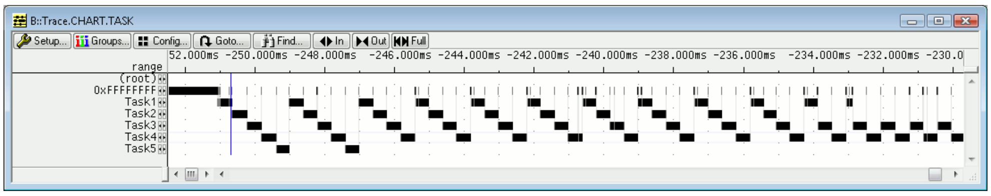


Figure 5.24.: An example of context change graphic produced by Lauterbach from the trace data of an application using the FRSH kernel.

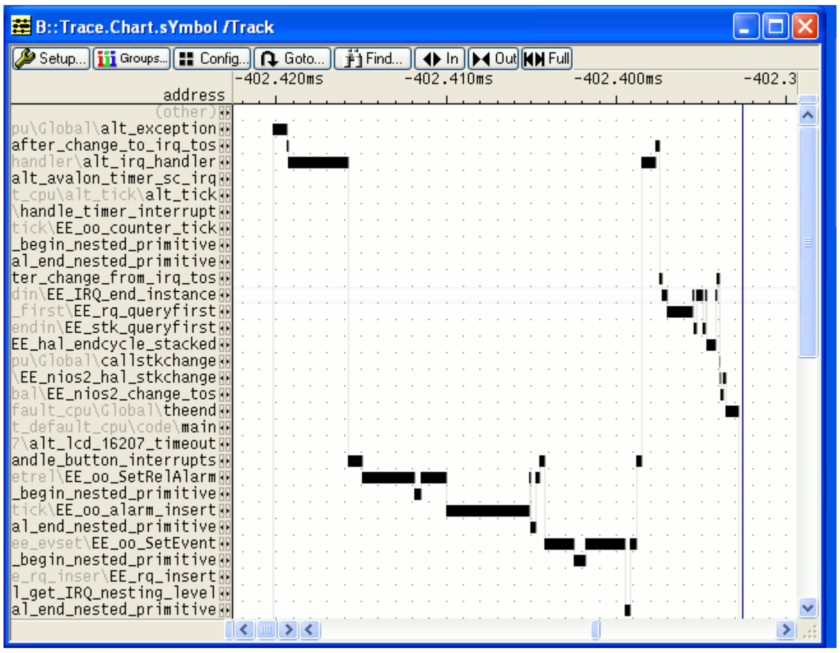


Figure 5.25.: The execution of the Button IRQ as recorded by Lauterbach Trace32.

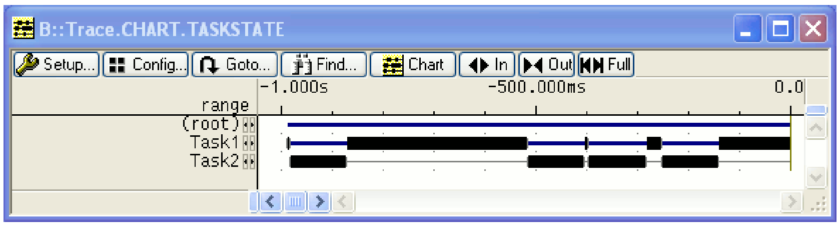


Figure 5.26.: The interpretation of a trace recorded with Lauterbach Trace32 showing the context changes happened in the system.

II tracer module. As an example, Figure 5.25 shows the execution of an interrupt handler as recorded by the tracer module. Figure 5.26 shows an interpretation of the context changes and task status values using the ORTI information.

Acknowledgements

We would like to thank Ing. Maurizio Menegotto from Lauterbach Italy Srl for his support in the integration of RT-Druid and Erika Enterprise with the Lauterbach Trace32 Debugger and Tracer.

6. Standalone version of RT-Druid

6.1. Introduction

This chapter describes the standalone version of RT-Druid. The idea behind of the standalone version is to provide the RT-Druid code generator plugin packed *without* the Eclipse Framework, to have a simple and fast way to generate code and templates from the command line.

The standalone version of RT-Druid is stored inside the `bin` directory inside the RT-Druid installation directory.

6.2. Code generation

To generate code using the standalone version of RT-Druid, please run the following command:

```
rtdruid_launcher.bat --oil filename --output dir
```

Where:

- `filename` is the name of the OIL file.
- `dir` is the directory where the generator should put the generated files. The `--output` option is optional. If not specified, the default directory name used is `Debug`.

6.3. Template code instantiation

This section explains how to obtain the automatic generation of a template example, as it is done from the "New Project" menu item from the Eclipse Framework.

To obtain the list of available templates, please run the following command:

```
template.bat --list
```

as a result, the command displays a list of the available templates, with their IDs.

Then, to instantiate a template, please run the following command:

```
template.bat --template ID --output dir
```

Where:

- `ID` is one of the IDs returned using the `--list` command..

- `dir` is the directory where the generator should put the generated files. The `--output` option is optional. If not specified, the default directory name used is the current directory.

Part II.

Schedulability Analysis Plugins

7. Schedulability Analysis Plugins Introduction

7.1. Generalities

7.1.1. Background

Embedded control systems development is about producing systems or sub-systems according to a set of specifications dictating their required (physics) properties. The output of the design stage is a set of models that describe the system at different levels of abstraction. Properties and constraints are captured by mathematical formalisms.

A typical design flow of embedded software (a v-shape process is represented in Figure 7.1) includes (among others) two stages of fundamental importance. The first stage, at the logical level, is the domain of application developers such as control engineers or other domain experts and produces a model of the software application providing a solution to the functional requirements in terms of one or more block diagrams. Most commercial design flows provide extensive support for this stage. Compliance against functional requirements is typically validated by extensive simulation and the risk of functional errors due to incorrect implementation is reduced by tools for the automated translation of the model (Real-Time Workshop Embedded Coder or TargetLink).

The second stage of the design process is at the architecture level, where software engineers (real-time systems experts) map the functions/components developed in the previous stage to real-time threads, select the scheduler and the resource managers by exploiting the services of a Real-Time Operating System (RTOS) and ultimately check the correctness of the timing requirements upon the target (HW) architecture. This design stage is transparent to software functionality (only provides its implementation), but it is crucial to achieve performance/cost trade-offs with the objective of providing the best possible quality and the best possible accuracy to system functions within the timing constraints and/or the performance target.

7.1.2. Problem description

Unfortunately, even in state-of-the-art processes, the above steps are performed relying solely on the skills or the experience of project developers. As a consequence, projects often suffer from design errors that show up only after deployment with obvious economic losses. If the software manufacturer detects timing errors or unsatisfactory performance during the extensive testing period, a chain of engineering changes is initiated that implies many iterations between the logical and the architectural view, trading implementation complexity for schedulability (or time performance) and tuning many design parameters, including system resources and (when possible) activation rates.

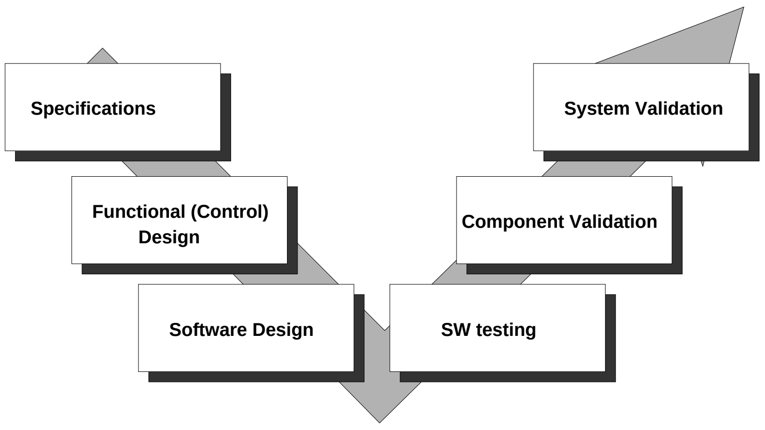


Figure 7.1.: V-shaped design process.

The separation of the design environments for functionality and for time, together with the limited support (feedback) provided by the existing tools make the process quite inefficient. Trial-and-error iterations are often performed without a clear knowledge of what should be modified and how.

7.1.3. Innovating the architecture design

Schedulability analysis theory bears the promise to fill in this gap, since it enables formal validation of the timing behavior of architecture-level solutions at the highest possible level in the flow. To this purpose, timing properties and constraints should be explicitly stated (by appropriate design annotations) and the (hardware and software) resources available for the execution of the application tasks and the resource allocation and scheduling policies must be clearly stated in appropriate design environments.

We take inspiration from recent literature in the field of embedded systems design, where a clear separation of concerns between functional design and implementation choices is advocated and the use of a shared set of abstractions allows for an efficient exchange of information between the two activities. In this way, the functional design remains independent from architectural choices and it does not require an advance commitment to any specific implementation.

7.2. The envisioned methodology: Architecture-level design

7.2.1. Introduction

The distinctive aspect of schedulability analysis is a match between the timing (or, in general, QoS) requirements of logical entities in the logical layer and the corresponding QoS offers of an implementation or architecture layer (Figure 7.2) and consequently

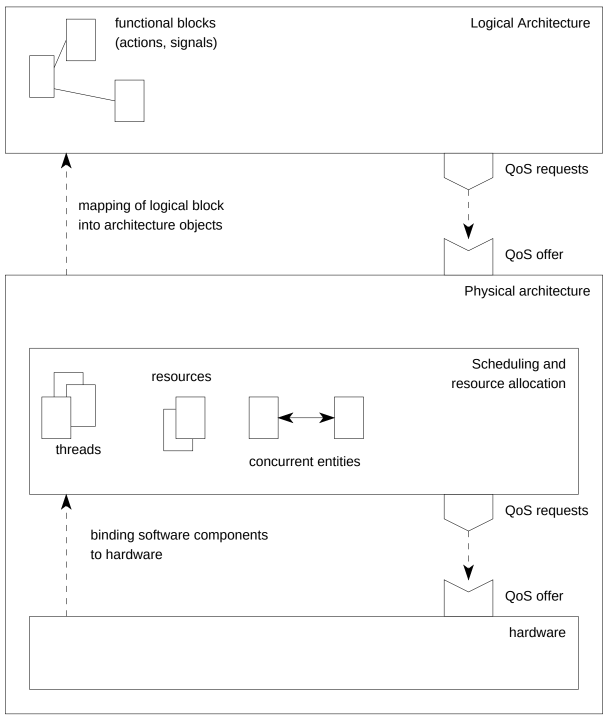


Figure 7.2.: Schedulability analysis requires creating a correspondence between logical and physical architecture.

raises the fundamental problem of mapping logical-level entities into architecture-level entities [9].

The logical architecture[1] (or functional model) consists of a network of functional blocks or components connected by means of communication variables and it is the outcome of the control design phase. This network is typically a straightforward derivation of the control schemes produced, for example, using such CAD tools as Matlab/Simulink or ETAS ASCET-SD. The concurrency description layer contains an implementation description consisting of all the threads that are running in the system and of the policies for managing resources (possibly implemented in the operating system). Finally, the hardware layer describes the hardware and its QoS features. The two bottom layers collectively define the so-called physical architecture layer, describing the mapping of the functional or logical design to a particular real-time execution environment.

The mapping stage consists of assigning each functional block to a software thread and each communication variable to a communication facility of the implementation (e.g. a task's private variable, a protected shared variable, etc.). It also includes an appropriate choice for such high level execution parameters as task activation rates. In order to comply with the timing constraints, it is crucial to perform appropriate choices for such real-time scheduling parameters as, at this stage, task priorities and ceilings

[1]In the sequel, we shall use the adjective s“logical” and “functional” interchangeably.

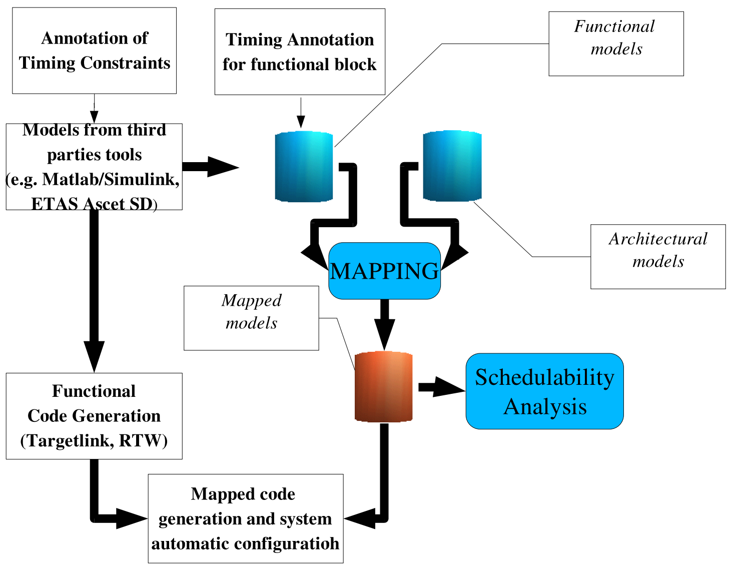

Annotation of Timing Constraints
Timing Annotation for functional block
Functional models
Models from third parties tools (e.g. Matlab/Simulink, ETAS Ascet SD)
Architectural models
MAPPING
Mapped models
Schedulability Analysis
Functional Code Generation (Targetlink, RTW)
Mapped code generation and system automatic configuratioh

Figure 7.3.: Schedulability analysis in the design flow

for semaphores protecting shared resource (see the OSEK/VDX specification). To this regard, a fundamental aid is offered by schedulability analysis. This activity is performed right after mapping (Figure 7.3) to test the correctness of a mapping hypotheses and to automatically synthesize some of the real-time scheduling parameters.

Several mapping-schedulability analysis iterations can be required before coming up with a satisfactory design. In our envisioned methodology such iterations are not *blind-eyed* since schedulability analysis does not simply provide a boolean answer on the schedulability of the system, but it returns performance information (i.e. response times) and sensitivity analysis. This feedback drives mapping and/or architecture design changes but it can also provide a measure of the resource (time) budgets that should be accounted for in a possible functional redesign stage.

7.2.2. Logical level design

At the logical level, most real-time systems feature a control and a dataflow part. The dataflow part is often time-driven, since inputs are sampled at regular intervals and it is suitable for an implementation consisting mainly of periodic activities. In contrast, the control-dominated functionality is highly asynchronous (event-driven) and state-dependent.

The model of the system represents both parts with suitable abstractions. The syntax and the semantics of the modeling language used by control engineers may vary, but in general, all models provide a meta-representation of the design as a structure or network of (functional) components connected via nets to each other's ports (mechanisms to communicate between blocks, such as shared variables) and communicating with each other using message (event) abstractions. Messages may be asynchronous (signals) or synchronous (call messages).

We provide a very simple and general metamodel (and a corresponding syntax based on XML) for representing an abstraction of the functional model where each component and event is annotated with timing attributes and constraints, such as assumptions on the maximum rate of arrival of events and on the worst case computation time of components. This abstraction is suitable for specification of architecture-level mapping and verification of timing properties.

7.2.3. Software architecture design

If the logical architecture primarily focuses on functional requirements, the physical architecture level is where physical concurrency, resource requirements and schedulability constraints are expressed. At this level, the units of computation are the processes or threads (or in general, tasks), executing concurrently in response to environment stimuli or prompted by an internal clock. Tasks cooperate by exchanging data and synchronization or activation signals and compete for the use of shared resources (including the processor, shared buffers, network links, etc.). Eventually, all logical entities have to be mapped onto corresponding software architectural entities, and the latter have to be mapped onto target hardware components in the physical architecture level. This activity entails the selection of an appropriate scheduling policy (for example, offered by a real-time operating system), which must be clearly supported by techniques for analyzing schedulability.

Our conceptual model uses the concepts of execution engine, scheduling policy, resources, and schedulable resources to model the elements of a physical architecture. An execution engine models the processing resource and has a scheduling policy that determines how the tasks will be scheduled.

7.2.4. Mapping

The mapping process consists of establishing an *implemented by* or realization relationship between functional components and architecture-level objects (such as tasks) and in the definition of a binding or deployment relationship between threads and resources and hardware components. The mapping relationship is constrained by the specifications defined at both the logical and the architecture levels, such as precedence and exclusion constraints among functional blocks or timing constraints enforcing the activation rates of software actions.

Mapping and binding decisions have crucial impact on the result of timing analysis. Load balancing and local schedulability clearly depend on binding decisions, when multiprocessor architectures are considered.

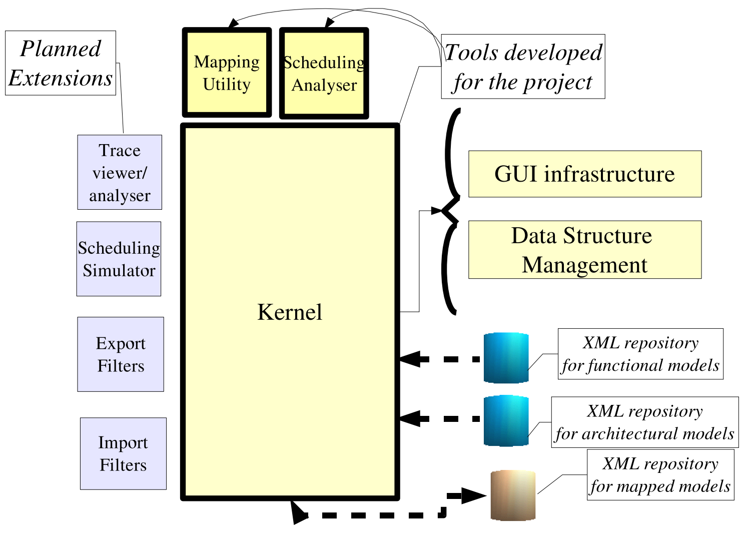

Planned Extensions
Mapping Utility
Scheduling Analyser
Tools developed for the project
Trace viewer/ analyser
Scheduling Simulator
Export Filters
Import Filters
Kernel
GUI infrastructure
Data Structure Management
XML repository for functional models
XML repository for architectural models
XML repository for mapped models

Figure 7.4.: RT-Druid architecture

7.2.5. Schedulability Analysis

Once the application functional model has been mapped on to a set of real-time tasks, it is necessary to verify that all temporal constraints are respected before actually implementing the system.

A real-time task is characterized by a period T (or minimum interarrival time or maximum activation rate), by a relative deadline D (i.e. the time computed from the activation by which each job must complete) and by a worst-case execution time C. The goal of the schedulability analysis is to check if all jobs can execute C units of execution time between their activation time and their deadline.

For example, all OSEK-compliant OS provide the fixed priority scheduling algorithm. For the fixed priority scheduling algorithm, three different schedulability analysis can be performed: the utilization-based test, the response time test and the hyperplane test.

In case of detected timing failures, the Schedulability Analysis returns feedback to the mapping in a form that allows the designer to pinpoint the exact cause that lead to schedulability or timing errors and give a range of actions as well as a measure of the entity of the corrections that would make the design feasible.

7.3. The Schedulability Analysis plugins

The Schedulability Analysis plugins of RT-Druid aids the designer in modeling, analyzing, and simulating the timing behavior of embedded real-time systems. Being an open and extensible environment based on XML and open standards (Java), RT-Druid can be easily integrated with existing systems.

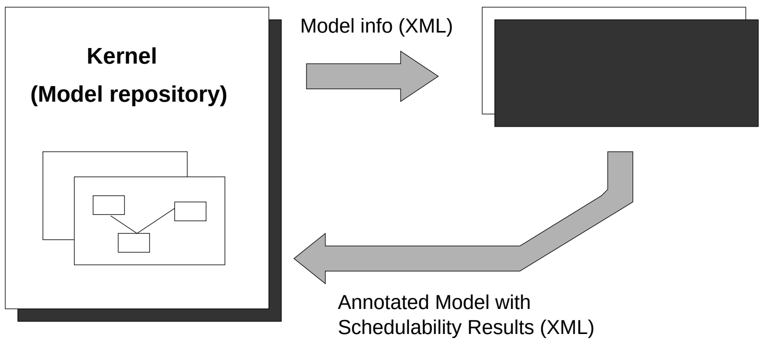


Figure 7.5.: Interactions between the kernel module and the Schedulability analyzer.

The RT-Druid schedulability plugins have a modular architecture shown in Figure 7.4. Model information (i.e. for both the functional and the architecture-level parts) is stored in an internal repository and it is made available by means of an open format based on XML. The tool set architecture is based on a kernel module, providing management of internal data structure and basic services for GUI and additional plugin modules. Plugins exploit kernel services in order to provide support to the design stages in a completely independent way.

The modeling and mapping module provides general Graphics and Text Diagrams to capture textual requirements, flowcharts, component and deployment notation and other general purpose information. Design teams can develop a representation of the embedded real-time system in graphical or textual form and capture the interaction between functional and architecture-level designs without disrupting the existing development process. This is done by using a neutral (XML-based) format and meta-model. Both logical and architecture-level modeling entities can be interactively defined within the context of the tool or imported by translation from external models. XML-based import/export utilities enable the transfer of information between our tools and other tools supporting XML or text-based standards. Modeling elements for logical and architecture-level abstractions mirror the definitions given in the previous sections.

The **schedulability analyzer** plugin provides support for worst case timing analysis. The module interacts with the other components by exchanging design information in the form of back annotations according to the schema of Figure 7.5. The forward path of Figure 7.5 represents the communication of the system model (as defined in the previous sections) to the schedulability analysis tool. The inverse model conversion provides an alternate option for using the results of the schedulability analysis since it can produce its results in the same XML files that are used to encode the design information. In particular, it stores the values for the attributes (priority, priority ceiling) that are synthesized based on the selected scheduling policy and the schedulability results themselves. Back-annotating the results into the same XML file that represents the model of the system helps the designer in keeping all the information related to the current iteration in the design flow aligned and consistent.

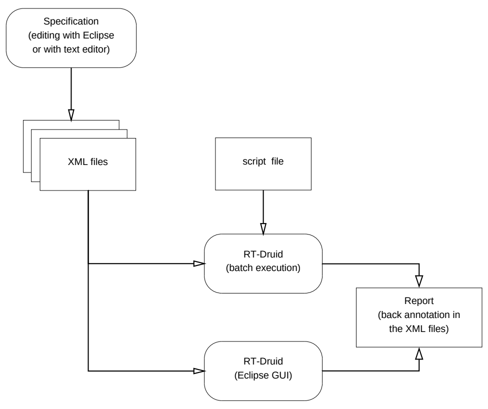


Figure 7.6.: Status of the Schedulability analysis plugin.

7.3.1. Status of the Schedulability analysys plugin

As shown in Figure 7.4, the current version of the tool offers a set of components that enables the user to perform a schedulability analysis of the system under design. A graphical GUI based on the Eclipse Project is available, which allows the user to edit the input files, perform the schedulability analysis and to visualize the related results.

The tool can be used as follows (see Figure 7.6):

- The user is required to specify the system (inclusive of functional model, software architecture and mapping) by means of XML files, whose format is documented in Chapter 8. The input files can be edited with any text editor, or with the GUI. The system specification can be contained in one single XML files, or it can be split among many files. The tool will automatically merge all files in one single specification. The initial specification may also come from an input file expressed in OSEK OIL or AUTOSAR XML (typically in these cases the model must be augmented with execution time informations since these file formats are not meant for schedulability analysis but only for RTOS configuration).

- Once the system has been specified, the user can run the tool in either batch mode or in graphical mode. To perform the operations in batch mode, the user must provide a Ant script file (see the Apache Ant project on the Apache web site). The script file features are described in Chapter B. The tool invocation results into the execution of commands specified in the script file.

- The results of schedulability analysis are then reported in an output file as well as in the RT-Druid Table view (if run from the graphical interface). It is possible to annotate the results in the same input files in a specific section.

8. Format of the Input File

8.1. Introduction

This chapter describes the structure and the syntax of the RTD XML format, which is used to describe the main component of a system. The RTD format has been used since the first versions of RT-Druid as the source of the information for the schedulability analysis plugin.

The DTD of the RTD specification is contained in the file `templates/evidence.DTD` of the RT-Druid documentation.

The structure of this chapter is the following: first, it describes the development process of an embedded real-time system with RT-Druid, and how the different steps of this process are reflected in the sections of the RT-Druid input file. Then, it details the structure of the DTD, describing each of its sections with examples. Finally, it describes other miscellaneous, like common elements and conventions.

8.2. Development process with RT-Druid

The objective of the RT-Druid schedulability plugins is to help the system designer in all phases of the design and development of a complex embedded application. For this reason, the input file is divided into different sections, each one containing information on a single phase of the design and development process. In particular, we envision the following steps:

- Modeling the functional behavior of the application. In this phase, the designer models the behavior of the system and of the application without any regard to the implementation of the system on a specific platform. This phase is typically done with tools like Matlab/Simulink, or similar. The description of the functional model of the system is done in the `FUNCTIONAL` section of the input file (see Section 8.6 for details).

- Modeling the software and hardware architecture of the system. In this phase, the designer models the characteristics of the platform on which the application will be implemented. This involves the definition of the number of computational nodes, the number and types of processors for each node, the RTOS used on each node, the number of real-time tasks, etc. The architectural model is specified in the `ARCHITECTURAL` part of the RT-Druid input file (see Section 8.7 for more details).

- Mapping the functional model to the architectural model. In this phase, the functional elements defined in the `FUNCTIONAL` section are "mapped" on to the appropriate architectural elements. For example, it is important to decide which task will perform a specific operation of the applications, and on which node/RTOS such task is allocated. Of course, many different mapping choices are possible, and the choice of the mapping can influence the performance of the system. In the RT-Druid input file, the mapping is specified in the `MAPPING` section (see Section 8.8 for more details).

- Back-annotation of performance measurement. In this phase, the system is run under specific testing conditions, and the performance of each element is measured and back-annotated. For example, in this phase each task is run under different conditions and the execution time is measured and recorded. At the end of the test, the average-case and worst case execution time of the task can be obtained. These information will be useful for the next phase, the schedulability analysis. The back-annotation information is contained in the `ANNOTATION` section of the RT-Druid input file (see Section 8.9 for more details).
- Schedulability analysis. In this phase, a systematic off-line analysis of the system is performed. The analysis will tell the designer if the system is schedulable (i.e. if all tasks will complete in time) under all conditions. Moreover, the analysis gives back additional sensitivity information about the system. For example, if the system is not schedulable, the analysis tells us which task will miss its deadline and how it is possible to modify the system the avoid this. In case the system is schedulable, it tells us how much "free space" is left before a deadline is missed. The results of the analysis are provided by the tool in the `SCHEDULABILITY` section (see Section8.10 for more details).

Not all sections are mandatory. In most cases, it is possible to specify only some section and still be able to analyze the system. For example, under certain conditions, the `FUNCTIONAL` section need not to be specified. This is useful, for example, when analyzing systems that are already implemented.

8.3. RTD and ERTD files

The RT-Druid schedulability plugin stores the input files in two different formats, each one having a different file extension, as shown in Table 8.1.

The two formats express the system semantic in a different form: the .ERTD form is suited for automatic processing by the EMF automatically generated loader, whereas the .RTD form is more easily readable, and hence it is more suited for manual editing. This chapter only describes the .RTD style XML file structure. The .ERTD file structure is an internal XML representationand and it is not described in this manual.

Note: RT-Druid uses the file extension to discriminate which file format should be expected as content. Files in .RTD form are parsed by an XSLT sheet, converted in .ERTD form, and then loaded. When saving the file from the GUI it is possible to save in both formats.

File extension	Description
.RTD	XML file format that has been created for ease human processing.
.ERTD	XML file format automatically generated by EMF. That format is not suited for readability.

Table 8.1.: XML file formats used by the RT-Druid Tool set.

8.4. .RTD XML file structure

The purpose of the file format presented in this section is to describe all the aspects of a complex system. As its root, a system is composed by a single `system` element, that contains a `Name` attribute used to discriminate between different instances of systems. The following is an example of a system definition:

```
<?xml version="1.0" encoding="UTF-8"?>
<SYSTEM Name="mySystem">
[...]
</SYSTEM>
```

Each system is described using sections. Each section is used to describe a particular phase of the design methodology that RT-Druid proposes. Here is a quick list of the various sections that compose a system description:

modes The modes section is used to list different working modes of a system. See section 8.5.

functional The functional section contains all the elements that describe the functional part of the system. The functional part of a system is in general composed by different elements, named `PROC`, `VAR`, `SUBSYSTEM`, and `TRIGGER` (external events). Moreover, the functional part contains the *use* relationships between the different elements in the system, the *partial ordering* between different events (start/end of a method, proc activation, trigger arrival). For each element it is possible to specify some temporal characterization, such as periodicity, deadline, minimum interarrival time, jitter and offset (see sections 7.2.2 and 8.6).

architectural The architectural section contains the specification of different parts of the system that are of interest for the purpose of mapping. This section contains, for example, elements such as `ECU`, `CPU`, `TASK`, `SEMAPHORE`s, `BUS` and related `FRAME`s. This section does not contain the relation between tasks and CPUs, that is instead contained in the mapping section. If a system includes only the architectural but not the functional section, all the references to `VAR`s are substituted with resources that otherwise are ignored (see Sections 7.2.3 and 8.7).

mapping The mapping section is used to describe the assignment of `PROC`s to `TASK`s, to specify which RTOS a task is assigned to, and to specify if a `VAR` element should be implemented as a mutex or through a bus frame. These information depend on the system application modes (see sections 7.2.4 and 8.8).

annotation The annotation section includes the temporal information related to the execution times of each task and of each method related to proc, var and resources. This information depends on the system application modes (see section 8.9).

schedulability The schedulability section contains the result of the schedulability test that is made for each CPU and for each task. This information depends on the system application modes (see sections 7.2.5 and 8.10).

It is not possible to describe more than one system inside a single input file. Not all the sections just described are mandatory in an instance of an input file, however it must be clear that some features may be disabled in case of missing informations. For example, schedulability tests cannot be performed if computation times are missing.

All the identifiers in an input file are *case insensitive*, except for the references that must contain the exact name of an object (that usually has a one-to-one relation with some identifier used in the source code).

Example Here is an example of a typical XML file structure.

```
<?xml version="1.0" encoding="UTF-8"?>
<SYSTEM Name="...">
  <MODES>
    [...]
  </MODES>
  <FUNCTIONAL>
    [...]
  </FUNCTIONAL>
  <ARCHITECTURAL>
    [...]
  </ARCHITECTURAL>
  <MAPPING>
    [...]
  </MAPPING>
  <ANNOTATION>
    [...]
  </ANNOTATION>
  <SCHEDULABILITY>
    [...]
  </SCHEDULABILITY>
</SYSTEM>
```

The following sections present in detail the content of each section in the XML file, detailing all the elements that characterize each part.

8.5. Mode section

Modes are abstractions that can be used to specify a configuration of the system that is used only under particular working conditions. For example, an airplane controller probably has different control algorithms when the airplane is taking off, flying and landing, and these algorithms have requirements expressed by a different set of tasks, resources, and so on. For this reason, the specification of modes in the system is useful for describing the behavior and the composition of the system in all its aspects and in an unified way.

Modes typically influence the mapping phase, because it is in that phase that the designer decides:

- which is the task that contains each PROC;
- which is the RTOS that contains each Task;
- how each variable is handled (private, protected using some kind of mutual exclusion, or bus variables);

As a consequence, the mapping phase influences the computation time and the blocking time of each task, and the schedulability of each single task and CPU in the system.

The Mode section is simply composed by a sequence of `MODE` elements, as described below. RT-Druid defines implicitly a *default Mode* that is always used for all the elements that do not explicitly specify one.

Please note that, although the name is similar, `MODE`s are not equivalent to OSEK/VDX Application Modes. OSEK/VDX Application Modes are used to identify a boot configuration which cannot be changed after the system starts (for example, normal operation, testing, ...). The `MODE` section is instead used to specify runtime aspects of an application (for example, engine stop, engine running at minimum RPM, engine running at 1000 rpm, engine running at 2000 rpm, ...). The idea is that each `MODE` may define a different application setup and a different schedulability analysis (for example, some tasks may have a frequency which is directly linked to the rotation speed of the engine).

8.5.1. `MODE`

`MODE` elements simply describe mode names. A `MODE` element contains only one attribute:

attribute Name is the symbolic name of a mode.

Example In the following example, four modes (`Init`, `Running, Fault` and `default mode`) are defined:

```
[...]
<MODES>
  <MODE Name="Init"/>
  <MODE Name="Running"/>
  <MODE Name="Fault"/>
</MODES>
[...]
```

8.6. Functional section

The functional section contains a set of elements that describe the *functional behavior* of the system. These entities will be mapped onto the architectural elements during the mapping phase. Each entity is represented by an XML element inside the main `FUNCTIONAL` element. The functional section may contain one or more instances of the following elements: `PROC`, `TRIGGER`, `VAR`, `EVENT`, `PARTIALORDER`, `TIMECONST`, `SUBSYSTEM`.

Element names are grouped in namespaces. Two elements inside the same namespace must have different names. RT-Druid defines the following namespaces for the `FUNCTIONAL` element:

- One namespace that contains `TRIGGER` names, `SUBSYSTEM` names, `PROC` and `VAR` external to subsystems.
- One namespace for each `SUBSYSTEM`. This namespace contains its internal `SUBSYSTEM`s, `PROC` and `VAR`;
- One namespace for each `PROC`, `VAR`, `REQUIRED_INTERFACE` and `PROVIDED_INTERFACE`. This namespace contains the names of the `METHOD` specified inside any of the elements listed before.
- One namespace for each `PROC`, `VAR`, `TRIGGER`, and `REQUIRED_INTERFACE`. This namespace contains the names of the `METHODREF` specified inside any of the elements listed before.
- One namespace that contains `EVENT` names [1].

8.6.1. `PROC`

A `PROC` element allows the description of the minimal computational entity, that *provides* one or more methods and *uses* one or more methods exported by other `PROC`s.

A `PROC` element may be internal or external to a subsystem:

- if a `PROC` element is internal to a subsystem, it can use only the `METHOD`s provided by the `PROC`, and the `VAR`iables internal to the same subsystem, or by the `REQUIRED_INTERFACE` of its own subsystem;
- if a `PROC` element is external to a subsystem, it can use the `METHOD`s provided by `PROC`s and `VAR`iables external to the subsystems, or the `PROVIDED_INTERFACE` of the other subsystems.

[1] In all the functional section, even if they are related to different subsystems

A PROC element has only one attribute:

attribute Name this attribute specifies the PROC name inside the current namespace.

A PROC element must contain at least one METHOD[2]. A PROC can contain zero or more METHODREF elements.

METHOD

This element represents a method provided by a PROC. For more information about methods, see Section 8.11.4.

METHODREF

This element specifies the coordinates that identify a method provided by some other element that will be called by the PROC's methods (see Section 8.11.5).

Example The following example shows the definition of two PROC elements named filter1 and filter2. filter1 provides a method method1, that is required by filter2.

```
[...]
<FUNCTIONAL>
  <PROC Name="filter1">
    <METHOD Name="method1"/>
    [...]
  </PROC>
  <PROC Name="filter2">
    <METHODREF
      Name="internal_name_for_method1"
      MethodName="filter1/method1"
    />
    [...]
  </PROC>
  [...]
</FUNCTIONAL>
[...]
```

8.6.2. VAR

The VAR element inside the XML representation is used to model an abstract data type. An abstract data type is typically composed by some data structures, and by a set of methods that are used to access and modify the them.

The VAR element has the following attributes:

attribute Name this attribute specifies the VAR name inside the current namespace.

[2]The next versions of RT-Druid will provide a default method if no method is specified

attribute Type this attribute specifies the type of the VAR element.

A VAR can only contain a set of METHOD elements.

METHOD

This element represents a method provided by the abstract data type implemented inside the VAR. See Section 8.11.4 for details.

Example

The following example describes two VAR elements, the first named image is implemented as an C integer matrix (int[][]), that provides two elements read and write; the second named size is implemented as an C integer (int), that provides two implicit methods read and write.

```
[...]
<FUNCTIONAL>
  <VAR Name="image" Type="int[][]">
    <METHOD Name="read"/>
    <METHOD Name="write"/>
  </VAR>
  <VAR Name="size" Type="int"/>
  [...]
</FUNCTIONAL>
[...]
```

8.6.3. TRIGGER

A TRIGGER element describes an external event that can activate and use one or more methods provided by other elements in the functional specification.

A trigger is characterized by a single attribute:

attribute Name this attribute specifies the TRIGGER name inside the FUNCTIONAL specification.

A TRIGGER can only contain a set of METHODREF elements.

METHODREF

This element represent a method invoked by a TRIGGER. For details see Section 8.11.5.

Example The following example, extracted from the example contained in the directory `examples/001`, describes a trigger named `imageArrival` that requires three methods: `write`, `decodeX` and `derX`, provided respectively by objects `image`, `filter1` and `der1`.

These reference have been renamed inside the trigger as `image.write`, `filter1.decodeX` and `der1.derX`.

```
[...]
<FUNCTIONAL>
  <TRIGGER Name="imageArrival" >
    <METHODREF
      Name="image.write"
      MethodName="image/write"
    />
    <METHODREF
      Name="filter1.decodeX"
      MethodName="filter1/decodeX"
    />
    <METHODREF
      Name="der1.derX"
      MethodName="der1/derX"
    />
    [...]
  </TRIGGER>
  [...]
</FUNCTIONAL>
[...]
```

Please note the difference between, for example, `der1.derX` and `der1/derX`: the former is the name of the `METHODREF` (the "." is part of the name), whereas the latter is the path to the `METHOD` element (as specified in Section 8.11.3, the "/" is used as separator).

8.6.4. EVENT

An `EVENT` element is used to tag the execution of a `METHOD`. `EVENT`s are then used when representing order between events, or timing constraints between different method executions. An `EVENT` can describe the action of activating a method (`activation`), and the fact that a method has started (`begin`) or ended (`end`) its execution on the final architecture.

Subsystems do not have "internal" `EVENT`s. In any case, `EVENT`s can tag internal subsystem methods. An `EVENT` has the following three attributes:

attribute Name this attribute specifies the `EVENT` name inside the functional section. `EVENT` names are unique in the entire functional section.

attribute Type specifies the action that the `EVENT` is tagging:

activation specifies that the `EVENT` tags the activation of a method. Activating a method means that the method is ready to execute, without specifying when the task will be really executed on the architecture.

begin specifies that the `EVENT` tags the beginning of execution of a method. The `begin` of a method always follows its `activation`.

end specifies that the `EVENT` tags the end of the execution of a method. The `end` of the execution of a method always follows its `begin`.

attribute MethodRefName specifies the name of the `METHODREF` that the `EVENT` is tagging.

Example The following example shows how to represent the activation event of a method.

```
[...]
<FUNCTIONAL>
  <EVENT
    Name="my_activation_event"
    Type="activation"
    MethodRefName="imageArrival/der1.derX"
  />
  [...]
</FUNCTIONAL>
[...]
```

8.6.5. `PARTIALORDER`

A `PARTIALORDER` element is a container of `ORDER` elements. An `ORDER` element can be used to describe a precedence relationship between two events. A `PARTIALORDER` is related to a particular system mode.

`PARTIALORDER` has only one attribute:

attribute ModeRef this attribute specifies the Mode to which the `ORDER` elements contained inside the `PARTIALORDER` element are related to. If ModeRef is not specified, the default mode is used.

The `PARTIALORDER` element can contain zero or more `ORDER` elements.

`ORDER`

An `ORDER` element only contains a pair of `EVENTREF` elements. The first `EVENTREF` is the predecessor, the second is the successor. For information about `EVENTREF` elements, see Section 8.11.6.

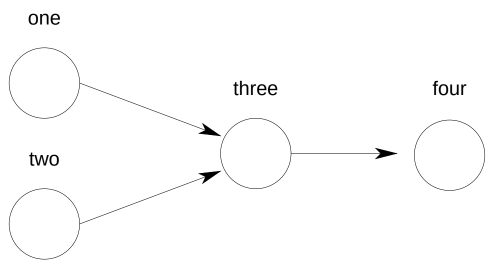


Figure 8.1.: The PARTIALORDER Example.

Example The following PARTIALORDER example shows four events with precedence relationship as shown in Figure 8.1.

```
[...]
<FUNCTIONAL>
  <PARTIALORDER>
    <ORDER>
      <EVENTREF Name="one"/>
      <EVENTREF Name="three"/>
    </ORDER>
    <ORDER>
      <EVENTREF Name="two"/>
      <EVENTREF Name="three"/>
    </ORDER>
    <ORDER>
      <EVENTREF Name="three"/>
      <EVENTREF Name="four"/>
    </ORDER>
  </PARTIALORDER>
  [...]
</FUNCTIONAL>
[...]
```

8.6.6. TIMECONST

A TIMECONST element is a container of one or two EVENTREF followed by a BOUND element. A TIMECONST is used to represent a constraint on events.

When the constraint describes a property of an event (such as an event periodicity), only one EVENTREF is needed.

When the constraint describes a property related to two events (such as an end-to-end deadline), two EVENTREFs are needed.

A TIMECONST has only one attribute:

attribute ModeRef this attribute specifies the Mode to which the TIMECONST elements is related to. If ModeRef is not specified, the default mode is used.

`EVENTREF`

For information about `EVENTREF` elements, see Section 8.11.6.

`BOUND`

The `BOUND` element is used to characterize the type of constraints represented by a `TIMECONST` element.

The available constraints are the following:

deadline specifies a constraint that the distance between two events must not exceed the deadline.

period specifies the period of an event;

mit specifies the Minimum Interarrival Time (MIT) of an event. That is, if an event appeared at time t, then the same event will not appear again before time $t + mit$.

jitter specifies the maximum jitter between two events;

offset specifies the constant time distance between two events;

The `BOUND` object has two attributes:

attribute Type The type of the constraint; Should be either “deadline”, “period”, “mit”, “jitter”, or “offset”.

attribute Value The temporal value related to the constraint.

The current version of RT-Druid only supports deadline, period and offsets.

Example The following example describes how to model the periodicity of an event and a deadline between two events.

```
[...]
<FUNCTIONAL>
  <TIMECONST>
    <EVENTREF Name="my_periodic_event"/>
    <BOUND Type="period" Value="40ms"/>
  </TIMECONST>
  <TIMECONST>
    <EVENTREF Name="start_event"/>
    <EVENTREF Name="end_event"/>
    <BOUND Type="deadline" Value="35ms"/>
  </TIMECONST>
  [..]
</FUNCTIONAL>
[...]
```

8.6.7. SUBSYSTEM

This element is a container that can be used to group a set of SUBSYSTEMs, PROCs and VARs. It has been introduced since RT-Druid 0.2 to permit a modular specification of the functional part of a system. It mimics the concept of subsystem defined in component-based languages.

A SUBSYSTEM consists of three sub-elements:

- an IMPLEMENTATION element that groups internal SUBSYSTEMs, PROCs and VARs; it represents the internal implementation of a module or component.
- a PROVIDED_INTERFACE element; it represents the interface that this subsystem provides to other subsystems or external elements.
- a REQUIRED_INTERFACE element; it represents the interface that this subsystem requires from external subsystems or elements in order to work correctly.

The implementation of a subsystem is hidden to the rest of the system. In this way, it is possible to change the implementation without affecting the implementation of the rest of the system. For this reason, SUBSYSTEM, PROC and VAR elements specified in the IMPLEMENTATION of a SUBSYSTEM cannot be referred directly from external entities. An external entity can only refer to the METHOD elements specified in the PROVIDED_INTERFACE section. On the other hand, a PROC or a VAR inside the IMPLEMENTATION can only refer to the METHOD elements specified in its REQUIRED_INTERFACE.

A subsystem does not contain EVENTs. However, EVENTS can tag internal subsystem methods (see Section 8.6.4).

Note: subsystems can be nested. That is, a subsystem can contain another subsystem.

A subsystem has the following attribute:

attribute Name This attribute is the name of the subsystem.

A subsystem contains three kinds of elements:

- IMPLEMENTATION
- PROVIDED_INTERFACE
- REQUIRED_INTERFACE

IMPLEMENTATION

This element contains one or more SUBSYTEM elements (see Section 8.6.7), one or more PROC elements (see Section 8.6.1) and one or more VAR elements (see Section 8.6.2) internal to a subsystem.

REQUIRED_INTERFACE

This element contains a list of methods that are required by the subsystem because some PROC element inside the subsystem needs that METHOD. Each one of these methods must be mapped on the external PROC, VAR, SUBSYSTEM that provides it, using a METHODREF. Therefore, the REQUIRED_INTERFACE is composed by a sequence of pairs each one composed by a METHOD and its METHODREF. The METHODREF can be missing if the binding of the SUBSYSTEM has not been completed (see the example below).

PROVIDED_INTERFACE

This element contains the list of methods exported by the subsystem. These exported METHODs are mapped to the methods of the objects inside the subsystem using elements of type EXPORTED_METHOD. For that reason, the PROVIDED_INTERFACE is composed by a sequence of pairs each one composed by a METHOD and its EXPORTED_METHOD. The EXPORTED_METHOD can be missing if the binding of the SUBSYSTEM has not been completed (see the example below).

EXPORTED_METHOD

This XML element is used to specify the internal method that is binded to the provided method exported by the subsystem. This element has the following attributes:

attribute ObjRef is the name of the object that contains the method (a PROVIDED_INTERFACE of a SUBSYTEM internal to the current subsystem, or a PROC or a VAR internal to the subsystem);

attribute MethodName is the method name.

Example The following example describes a subsystem named mySub, which exports four methods read, write, start and stop and which requires two methods input and output. The subsystem contains a PROC named subProc and a var named subData. The provided methods are linked to the internal methods. The subsystem requires two external methods, input and output. The first one has been mapped on method getData of subsystem yourSum, whereas the output method has not been mapped yet.

```
[...]
<FUNCTIONAL>
  <SUBSYSTEM Name="mySub">
    <IMPLEMENTATION>
      <PROC Name="subProc">
        <METHOD Name="start"/>
        <METHOD Name="stop"/>
        <METHODREF
          Name="subProc.read"
          MethodName="mySub/input"
        />
        <METHODREF
          Name="subProc.write"
          MethodName="mySub/output"
        />
        <METHODREF
          Name="subProc.data.read"
          MethodName="subData/read"
        />
        <METHODREF
          Name="subProc.data.write"
          MethodName="subData/write"
        />
      </PROC>

      <!-- implicit read and write methods -->
      <VAR Name="subData" Type="int"/>
    </IMPLEMENTATION>

    <PROVIDED_INTERFACE>
       <METHOD Name="read"/>
       <EXPORTED_METHOD
          ObjRef="subData"
          MethodRef="read"
       />
       <METHOD Name="write"/>
       <EXPORTED_METHOD
          ObjRef="subData"
          MethodRef="write"
       />
      <METHOD Name="start"/>
      <EXPORTED_METHOD
          ObjRef="subProc"
          MethodRef="start"
       />
      <METHOD Name="stop"/>
      <EXPORTED_METHOD
          ObjRef="subProc"
          MethodRef="stop"
       />
     </PROVIDED_INTERFACE>
     <REQUIRED_INTERFACE>
       <METHOD Name="input"/>
       <METHODREF
          Name="inputMapping"
          MethodName="yourSub/getData"
       />
       <METHOD Name="output"/>
     </REQUIRED_INTERFACE>
  </SUBSYSTEM>
  [...]
</FUNCTIONAL>
[...]
```

8.7. Architectural section

The architectural section contains a set of elements that describes the *hw/sw architecture* of the system. These entities will be mapped together with methods of the functional section during the mapping phase.

Each architectural entity is represented by an XML element inside the main `ARCHITECTURAL` element. The architectural section may contain the following elements: `ECU`, `TASK`, `RESOURCE`, `BUS`, `FRAME`, `SIGNAL`, `MUTEX`. Element names are grouped in namespaces. Every element inside a namespace must have a different name. The RT-Druid Toolset defines the following namespaces for the `ARCHITECTURAL` section:

- a namespace that contains `ECU` names;
- a namespace that contains `RTOS` names;
- a namespace that contains `TASK` names;
- a namespace that contains `RESOURCE` names;
- a namespace that contains `BUS` names;
- a namespace that contains `FRAME` names;
- a namespace that contains `SIGNAL` names;
- a namespace that contains `MUTEX` names.

8.7.1. `ECU`

The `ECU` (Embedded Control Unit) element represents a computational node composed by one or more `CPU` elements; `CPU` elements belonging to the same ECU may be etherogeneous (e.g., an `ECU` can have an ARM7TDMI and an Hitachi H8). The various `CPU` elements mapped to the same `ECU` can communicate using a shared bus (message passing paradigm) and/or shared variables (shared memory).

Note: the current version of the schedulability analysis of the RT-Druid Toolset does not consider shared resources between tasks allocated to different `CPU` elements. Moreover, there is currently no check to prevent such situation. A mutex shared between tasks allocated to different CPUs is treated as a local mutex on both CPUs; no warning message is printed if this condition appears.

The `ECU` element has only one attribute:

attribute Name is the name assigned to the `ECU`.

The `ECU` element contains one or more `CPU`s.

`CPU`

The `CPU` element models a computation unit (that in general can be a generic commercial micro-controller) that is controlled by a Real-Time Operating System (RTOS).

The current version of the RT-Druid Tool set only allows the existence of one `RTOS` for each `CPU`. This constraint will be relaxed in the following versions.

The `CPU` element has the following attributes:

attribute Name is the the name of the `CPU` inside the `ECU`;

attribute Model is the CPU model (e.g., ARM7TDMI, PPC z7, ...). This information is optional.

A `CPU` can contain one or more `RTOS`.

`RTOS`

The `RTOS` element is used to model the operating system that will be installed on the CPU, and it is identified by a name that is used during the mapping phase to specify the relation between task and CPU.

The `RTOS` element has the following attribute:

attribute Name is the the name of the `RTOS` element;

The RTOS element contains two elements named `OSMODEL` and `OILVAR`, that in future releases of the RT-Druid Toolset will specify the peculiarities of the Operating System. In the current version of the RT-Druid Tool set these elements are empty.

Example This example shows a single `ECU` node, that has two ARM7TDMI CPUs each one handled by a different operating system.

```
[...]
<ARCHITECTURAL>
[...]
  <ECU Name="ECU0">
    <CPU Name="CPU0" Model="ARM7TDMI">
      <RTOS Name="CPU0.erikaEnterprise"/> </RTOS>
    </CPU>
    <CPU Name="CPU1" Model="ARM7TDMI">
```

```
      <RTOS Name="CPU1.erikaEnterprise"/> </RTOS>
    </CPU>
  </ECU>
[...]
</ARCHITECTURAL>
[...]
```

8.7.2. TASK

The `TASK` element can be used to represent both an RTOS task or and RTOS ISR.[3]

Each `TASK` element is characterized by a set of parameters (e.g., its priority, period or deadline), that in general depends on the Application Modes.

Moreover, a `TASK` contains some information useful for the OIL standard. These informations are not used in the current version of the RT-Druid Tool set.

The `TASK` element has the following attributes:

attribute Name is the name of the `TASK` element;

attribute Type specifies whether the `TASK` element represents a task or an ISR in the final implementation.

The `TASK` element may contain the following elements:

SCHEDULING

The `SCHEDULING` element is used to specify the scheduling parameters of a task.

The `SCHEDULING` element has the following attributes:

attribute Priority is used to store informations about the Task priority. Lower values are considered to be lower priority in the system; bigger numbers refer to higher priorities [4];

attribute Threshold is used to store informations about Task Threshold (supposing that the Kernel specifies a Preemption Threshold scheduling algorithm).

attribute PreemptionGroupName is used to store the Non-Preemption Group of a Task.

attribute ModeRef is used to specify to which Mode these scheduling parameters are related.

[3]When running the schedulability analysis, the RT-Druid Tool set handles task and ISRs in the same way. When generating RTOS configuration files, tasks and ISRs are considered in a different way.

[4]Please note that this priority assignment extends the OSEK Priority assignment used in OIL, where 0 is considered the lowest priority and bigger number correspond to higher priorities.

`ACTIVATION`

This element is used to specify the activation parameters of a task. For example, this element specifies if the task is activated periodically or not, if it can have pending activations, and moreover it specifies other activation parameters.

The `ACTIVATION` element has the following attributes:

attribute Type is the type of the Task/ISR. Can be "`sporadic`", "`periodic`", "`aperiodic`".

attribute ActNumber is the number of pending activations for a task. This value is used for example in RTOS like those compliant with OSEK BCC2/ECC2 conformance classes, that can admit more than one pending activation for tasks. Typically this value is set to 1.

attribute Class is the class of the task/ISR.

attribute Period is a time value that can be used to specify a typical period of a Task/ISR. In the case of periodic tasks, the value is the period of the task. In the case of sporadic tasks or ISRs, this value represents the Minimum Interarrival Time.

attribute Offset is a time value that specifies the activation offset of the task. It is currently used *only* for the purpose of scheduling analysis, and does not influence the code generated by the RT-Druid Toolset.

attribute Deadline is a time value that represents the relative deadline of a task.

attribute ModeRef is used to specify to which Mode these activation parameters are related.

`RESOURCEREF`

A `RESOURCEREF` element is used to group the methods of the resources used by tasks (depending on the mode of operation). `RESOURCEREF` elements are mainly used for schedulability analysis when a partial system is specified (e.g., a system with incomplete mapping or incomplete functional specification). In this sense, `RESOURCEREF` elements are not the main way a user should follow to specify the relationship between tasks, methods, and mutexes. See Section 8.7.3 for details.

A `RESOURCEREF` element has the following attribute:

attribute ModeRef is used to specify to which Mode these `RESOURCEREF` element is related to.

A `RESOURCEREF` should contain one or more `METHODREF` element (see Section 8.11.5), that specifies the methods used by a task that have to be protected by mutexes.

`OILVAR`

The `OILVAR` element is used to store in an RTD file the corresponding content of an Erika Enterprise OIL File. The idea is the following: the OIL file contains all the information needed to generate the configuration code of the Erika Enterprise kernel. Most of these information are often linked to a specific microcontroller or boards, and it is not worth to model them in detail into an RTD or ERTD file. For this reason, when RT-Druid imports an OIL file inside an RTD file, all the elements which cannot be directly mapped to an RTD element are put inside the OILVAR container.

Example This example describes a periodic TASK, with a 45 ms period, named Writer. The task has a priority equal to 3, and it uses three resources, Res8, Res12 and Res6, to do write operations on the first two resources, and read operations on the latter.

```
[...]
<ARCHITECTURAL>
[...]
  <TASK Name="Writer" Type="task">
     <SCHEDULING Priority="3" />
     <ACTIVATION Type="periodic" Period="45ms" />
     <RESOURCEREF>
        <METHODREF Name="Res8.write"  MethodName="Res8/write" />
        <METHODREF Name="Res12.write" MethodName="Res12/write" />
        <METHODREF Name="Res6.read"   MethodName="Res6/read" />
    </RESOURCEREF>
  </TASK>
[...]
</ARCHITECTURAL>
[...]
```

8.7.3. `RESOURCE`

A `RESOURCE` element is used to specify methods of the resources used by tasks (depending on the mode of operation). `RESOURCE` elements are mainly used for schedulability analysis when a partial system is specified (e.g., a system with incomplete mapping).

Typically a user specifies (using the elements in the `MAPPING` section) the mapping between `TASK` and `PROC` elements, and between `VAR` and `MUTEX` elements (see Figure 8.2). Then, in the `FUNCTIONAL` section the `PROC` elements specifies which `VAR` elements they use. Using these informations the schedulability analyzer can know which are the resources used by `PROC` elements (that information is useful when doing a schedulability analysis).

Suppose now that the user do not have a complete system, because for example the mapping has not been completed yet. In such a situation, the user could not perform schedulability analysis, because there is no way for the system to know which mutexes each task is using.

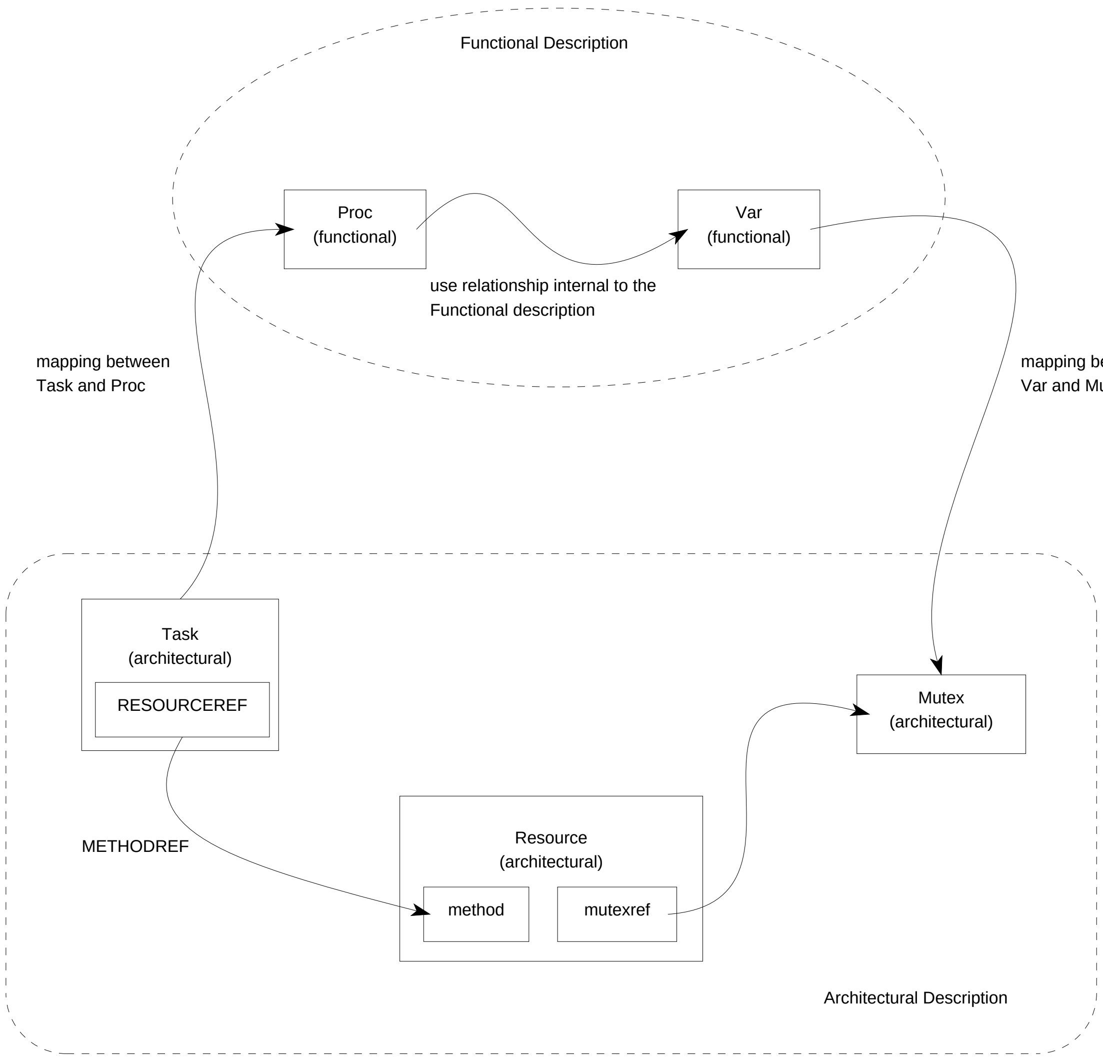


Figure 8.2.: Use of `RESOURCE` elements to enable schedulability analysis without complete mapping specified.

Unfortunately such a situation happens often, for example when the user tries to import an OSEK OIL architectural specification (OIL contains the architectural specification of a system, but not the functional part). `RESOURCEREF` elements is a temporary way the user can use to link resources to tasks directly inside the architectural specification making schedulability analysis possible also with a incomplete mapped system.

If both methods (mapping and `RESOURCE` elements) are specified for an object (e.g. a task) in the a system, the schedulability analyzer will always use the mapping informations regardless of what specified in the `RESOURCE` elements.

The presence of the mapping relationship between `TASK` and `PROC` elements are the discriminant part that is used by the RT-Druid Toolset for choosing the right path to follow. That is, if the mapping is present and a `VAR` has links with any mutex, it is supposed that the `VAR` is a private data structure that does not need to be protected with mutexes. That because the Toolset subsumes that the mapping part can be either not or completely present.

A `RESOURCE` element has the following attribute:

attribute Name is the name of the `RESOURCE` element.

The `RESOURCE` element can contain zero or more `METHOD` elements (same as in Section 8.11.4) and one or more `MUTEXREF` elements (see Section 8.11.7).

A `RESOURCE` always has two *predefined* methods named `read` and `write`.

Example This example shows a resource `Res2` that is protected by mutex `Mutex2`.

```
[...]
<ARCHITECTURAL>
[...]
  <RESOURCE Name="Res2">
     <MUTEXREF MutexName="Mutex2"/>
  </RESOURCE>
[...]
</ARCHITECTURAL>
[...]
```

8.7.4. BUS

The `BUS` element is used to represent the bus of the `ECU`.

The `BUS` element is currently not used.

A `BUS` element has the following attributes:

attribute Name is the name of the bus.

attribute Type is the type of the bus.

attribute Speed is the speed of the bus.

Example No example available for the `BUS` element.

8.7.5. `FRAME`

The `FRAME` element is used to represent a set of messages sent on the network, such as an OSEK COM IPDU.

The `FRAME` element is currently not used.

A `FRAME` has the following attributes:

attribute Name is the name of the frame.

attribute ID is the identifier of the frame.

attribute ActivationType is the activation type of the frame.

attribute ActivationClass is the activation class of the frame.

attribute ActivationRate is the activation rate of the FRAME.

attribute Length is the length of the frame.

Example No example available for the `FRAME` element.

8.7.6. `SIGNAL`

The `SIGNAL` element is used to represent the behavior of OSEK alarms, counters and events.

The `SIGNAL` element is currently not used.

A `SIGNAL` element has the following attributes:

attribute Name is the name of the signal.

attribute Type is the type of the signal.

Example No example available for the `SIGNAL` element.

8.7.7. `MUTEX`

The `MUTEX` element is used to represent a RTOS mutex. A mutex can be thought as a binary semaphore used to ensure mutual exclusion between tasks (and on OSEK OS, also between ISRs). `MUTEX` elements are currently used during the schedulability analysis to compute blocking times of tasks. A `MUTEX` element has the following attributes:

attribute Name is the name of the mutex.

attribute Policy is the policy that must be used for the mutex. The attribute is optional. Policy identifiers have not been specified yet. Currently, the behavior of the Tool set is to ignore the Policy attribute, considering that all the mutexes will use the Immediate Priority Ceiling Protocol.

Example This example shows how to declare a mutex named `Mutex2`.

```
[...]
<ARCHITECTURAL>
[...]
  <MUTEX Name="Mutex2"/>
[...]
</ARCHITECTURAL>
[...]
```

8.8. Mapping section

The mapping section contains a set of elements that describes the *mapping* of the system. The mapping of the system is the result of the mapping phase, which links the functional aspects (that is, `PROC` elements, `VAR` elements, and so on) to the architectural aspects of a system (that is, `TASK`, `MUTEX`es, and so on). Each architectural entity is represented by an XML element inside the main `MAPPING` element. The mapping section may contain the following elements: `PROCMAP`, `TASKMAP`, `VARMAP`.

8.8.1. `PROCMAP`

The `PROCMAP` element is used to represent the mapping relationship between `PROC` and `TASK` elements: that is, the fact that a `PROC` is executed by a particular `TASK`. In general, different `PROC` elements will be mapped to a single Task. The ordering between different `PROC` elements can be also specified passing an integer number as attribute. The `PROCMAP` element has the following attributes:

attribute TaskRef is the name (path) of the `TASK`. If the `TASK` name contains special characters, they must be protected using “\”.

attribute ProcRef is the full name (path) of the `PROC`. If the `PROC` name contains special characters, they must be protected using “\”.

attribute ModeRef is the name of the mode to which this mapping is related. If not specified, the default mode is used.

attribute Order is the mapping order (an integer number). This attribute is currently not used.

Please note that the `PROCMAP` elements do not have “names”.

Example The `Guide` task is composed by three `PROC` contained inside the `Guide` subsystem. These `PROC` elements have the following names: `decode`, `feedBack` and `pwmRef`. This example shows how to specify that the `Guide` task is composed by the sequence of the three `PROC` elements in that particular order.

```
[...]
<MAPPING>
[...]
  <PROCMAP ProcRef="Guide/decode"   TaskRef="Guide" Order="0"/>
  <PROCMAP ProcRef="Guide/feedBack" TaskRef="Guide" Order="1"/>
  <PROCMAP ProcRef="Guide/pwmRef"   TaskRef="Guide" Order="2"/>
[...]
</MAPPING>
[...]
```

8.8.2. TASKMAP

The TASKMAP element is used to represent the mapping relationship between TASK and RTOS elements: that is, the fact that a TASK is executed by a particular RTOS on a given CPU. The TASKMAP element has the following attributes:

attribute TaskRef is the name (path) of the TASK. If the TASK name contains special characters, they must be protected using "\".

attribute RtosRef is the name of the RTOS. The name of the RTOS is NOT a pathname.

attribute ModeRef is the name of the mode to which this mapping is related. If not specified, the default mode is used.

Please note that the TASKMAP elements does not have "names".

Example This example shows how the Guide task is mapped to the RTOS identified by the name CPU0.erika.

```
[...]
<MAPPING>
[...]
  <TASKMAP rtosRef ="CPU0.erika" TaskRef="Guide"/>
[...]
</MAPPING>
[...]
```

8.8.3. VARMAP

The VARMAP element is used to represent the mapping relationship between VAR and MUTEX elements, or between a VAR element and FRAME/BUS elements.

The idea is that a data structure needs some kind of mutual exclusion protection that is provided by a RTOS by means of some sort of binary semaphore (in case of a shared memory paradigm) or by means of some message that needs to be transmitted on a bus (in case of a message passing paradigm).

The TASKMAP element has the following attributes:

attribute VarRef is the name (path) of the `VAR` element. This attribute is mandatory. If the `VAR` name contains special characters, they must be protected using "\".

attribute FrameRef is the name of the `FRAME` element to which the `VAR` is mapped. This attribute is optional. If specified, also the `BUS` element must be specified.

attribute BusRef is the name of the `BUS` element to which the `VAR` is mapped. This attribute is optional. If specified, also the `FRAME` element must be specified.

attribute MutexRef is the name (path) of the `MUTEX` element to which the `VAR` is mapped. This attribute is optional. If the `MUTEX` name contains special characters, they must be protected using "\".

attribute ModeRef is the name of the mode to which this mapping is related. If not specified, the default mode is used.

Example This example shows 2 global variables named `OmegaRef1` and `OmegaRef2`, and two variables local to the `Guide` subsystem, named `img1` and `img2`. These variables are protected by two distinct mutexes: `InputMutex` that protects `img1` and `img2`, and `SpeedMutex` that protects `omegaRef1` and `omegaRef2`.

```
[...]
<MAPPING>
[...]
  <VARMAP VarRef="Guide/img1" MutexRef="InputMutex"/>
  <VARMAP VarRef="Guide/img2" MutexRef="InputMutex"/>

  <VARMAP VarRef="omegaRef1" MutexRef="SpeedMutex"/>
  <VARMAP VarRef="omegaRef2" MutexRef="SpeedMutex"/>
[...]
</MAPPING>
[...]
```

8.9. Annotation section

The annotation section contains a set of elements that describes information about the actual performance of the system being modeled.

The idea is that the system is modeled, mapped, and then tested on the final target. Once a test application have been run, some performance results (such ad example the task execution times) are back-annotated inside the model using the `ANNOTATION` section. Back-annotated informations are then used for other purposes, such as schedulability analysis or scheduling simulation.

The current version of the RT-Druid Tool set needs back-annotated informations for the purpose of scheduling analysis.

The `ANNOTATION` section may contain only the element `EXECTIME`.

8.9.1. `EXECTIME`

The exectime element is used to describe the execution time of an entity in the system. Supported entities for `EXECTIME` are `TASK` elements, `PROC` elements, `METHOD` elements and `RESOURCE` `METHOD` elements.

The `EXECTIME` element has the following attributes:

attribute Ref is the identifier (path) of the element which the `EXECTIME` is referred to. Please note that when specifying the path you must not specify the section name (e.g., ARCHITECTURAL), because it is implicit in the Type attribute.

attribute Type is the type of element to which the `EXECTIME` element is referred to. Available types are:

- **task** The Ref attribute refers to a `TASK` name. The back annotation gives information about an entire `TASK` instance.
- **proc** The Ref attribute refers to a `PROC` name. The back annotation gives information about a `PROC` execution time.
- **method** The Ref attribute refers to a specific `METHOD` of a `VAR` or of a `PROC`.
- **resource_method** The Ref attribute refers to a `METHOD` of an architectural element `RESOURCE`.

attribute ModeRef is the name of the mode to which this mapping is related. If not specified, the default mode is used.

The current version of the schedulability analyzer use the Task exec times as informations for the execution time of tasks. Then, it uses the execution time of methods linked to `VAR` and `RESOURCE` elements as the information from where the blocking times are computed.

The `EXECTIME` element can contain the following elements: `WORST`, `BEST`, `DISTRIBUTION`.

The current version of the RT-Druid Toolset uses only the `WORST` element.

`WORST`

This element can be used to specify the worst case execution time of the element being measured by the `EXECTIME`.

The `WORST` element has only one attribute:

attribute Value is the worst case execution time.

`BEST`

This element can be used to measure the best case execution time of the element being measured by the `EXECTIME`.

The current version of the RT-Druid Tool set does not use this element.

The `BEST` element has only one attribute:

attribute Value is the minimum measured execution time.

DISTRIBUTION

This element can be used to represent the probabilistic distribution of the execution time of the element being measured by the EXECTIME. The current version of the RT-Druid schedulability analysis does not use this element. The DISTRIBUTION element has only one attribute:

attribute Type is used to specify the kind of distribution (e.g., Gaussian, Poisson, ...).

The DISTRIBUTION element can contain the following elements: AVG, VARIANCE, SAMPLE.

AVG The AVG element is used to contain an average execution time. It only has an attribute **Value** that contains that information. Only zero or one element of this kind can appear inside a DISTRIBUTION element.

VARIANCE The VARIANCE element is used to contain the variance of the execution time. It only has an attribute **Value** that contains that information. Only zero or one element of this kind can appear inside a DISTRIBUTION element.

SAMPLE The SAMPLE element is used to represent a discrete probability distribution of the execution times. Each SAMPLE element represents a sample of the probability distribution. For that reason, the element has two attributes, **Value** and **Probability**, that are used to describe the value of the sample and its probability.

Of course, zero or more SAMPLE elements can appear inside a DISTRIBUTION element.

Example The following example shows how to annotate the execution time of the read and write methods provided by variable img1 internal to the Guide subsystem. The annotation also shows the amount of time the Guide task has executed on the CPU.

```
[...]
<ANNOTATION>
[...]
  <EXECTIME Ref="Guide/img1/read" Type="METHOD">
     <WORST Value="8us" />
  </EXECTIME>
  <EXECTIME Ref="Guide/img1/write" Type="METHOD">
     <WORST Value="10us" />
  </EXECTIME>

  <EXECTIME Ref="Guide" Type="TASK">
     <WORST Value="0.5ms" />
  </EXECTIME>
[...]
</ANNOTATION>
[...]
```

8.10. Schedulability section

The schedulability section contains a set of elements that store the result of the schedulability analysis. The idea is that the RT-Druid Tool set (once the user has provided all the necessary informations about tasks, methods, resources, ...) is able to compute some schedulability analysis (see Chapter **??**) that returns two kind of informations:

- a textual report;
- a detailed (for each `RTOS`, for each `TASK`) schedulability information.

The schedulability section reports, for each application mode, a scheduling *scenario*, that is, the schedulability information for the system. The system retains the last detailed schedulability informations and all the reports which have been generated. The schedulability section is composed by a sequence of `SCHEDULINGSCENARIO` elements.

8.10.1. `SCHEDULINGSCENARIO`

A scheduling scenario represents the system schedulability information. A scheduling scenario is explicitly referred to a single application mode. Only a scheduling scenario may exist for a given application mode.

The `SCHEDULINGSCENARIO` element has only one attribute:

attribute ModeRef is the application Mode to which the scenario refers to.

The `SCHEDULINGSCENARIO` element can contain one ore more of the following elements: `REPORT`, `CPUSCHED`, `TASKSCHED`.

`REPORT`

The `REPORT` element is an unformatted text container that contains the schedulability report produced by the schedulability tool. More than one report can be present for the same application mode.

`CPUSCHED`

The `CPUSCHED` element gives some overall scheduling informations about an `RTOS` mapped to a `CPU`. The `CPUSCHED` element contains the following attributes:

attribute CpuRef is the `CPU` name which this element refers to. `CpuRef` is not a pathname, because there is only a namespace for `CPU`s.

attribute Utilization is the utilization factor of the taskset allocated to the `CPU`.

attribute SpeedFactor is the speed factor of the taskset allocated to the `CPU`.

attribute Boundary is the theoretical U_{lub}boundary for the taskset.

attribute Schedulable is the answer of the schedulability test. Possible values are "*yes*" and "*no*".

TASKSCHED

The TASKSCHED element gives some overall scheduling informations about a task.

The TASKSCHED element contains the following attributes:

attribute TaskRef is the task (path) name which this element refers to. If the TASKREF name contains special characters, they must be protected using "\".

attribute Utilization is the utilization of the task.

attribute CDelta is the amount of computation time the task should decrease to let the task set being schedulable.

attribute TDelta is the amount of time the period of a task should increase to let the task set being schedulable.

attribute Schedulable is the answer of the schedulability test. Possible values are "*yes*" and "*no*".

attribute ResponseTime is the worst case response time of a task.

Example A detailed explanation of the report and on the results can be obtained running the examples provided with the RT-Druid Toolset. The meaning of each field is described in detail in Chapter **??**.

8.11. Conventions and references

The XML file structure defined in this chapter is written using a set of conventions that specifies how things should be written inside some attribute or element descriptions. Moreover, some elements (e.g., METHOD elements) are common to different element specifications (e.g., a VAR and a PROC can have a METHOD element). Finally, some elements often need to refer other elements (for example, to express that "a PROC is using a METHOD provided by a VAR"). For that reason, specific XML elements have to be used (e.g., a METHODREF), and the meaning of these elements is simply to refer other elements.

This section lists all these reference elements, conventions, and shared elements.

8.11.1. Name convention

The current implementation of the RT-Druid Toolset maps the different entities in the system under analysis (e.g., Tasks, CPUs, execution times, ...) to XML elements. Each XML element will typically refer to a different system entity. Entities have a name specified in the name attribute. Two XML elements of the same kind α and β with the same name attribute value refer to the same system entity. In case two (or more) declarations exist, the tool will try to merge the parameter definitions in the declarations. If the same parameter is assigned different values in different declarations (in case of conflicts), the last one that appears in the input file will prevail, but no error message will be provided.

Future versions of the tool set will provide error messages to inform the user about the overwriting of different system parameters.

8.11.2. Time references

Throughout this document, whenever a timing reference is needed, the user can specify a timing unit chosen from the following list:

h hours

m minutes

s seconds

ms milliseconds

us microseconds

ns nanoseconds

If not specified, milliseconds (**ms**) are used as default timing unit.

Example

Example of time units are:

1ms one millisecond;

1m one minute;

60001ms one minute plus one millisecond.

8.11.3. Paths to elements

The XML structure used by the RT-Druid Tool set describe a tree of XML elements representing the various elements in a given system. Throughout this document, whenever a reference to an existing element is needed, the entire path to that element should be used.

The path should contain the values of the `Name` attribute separated by the *slash character* "/". As an example, a valid path is "`mysubsystem/myproc/write`". If the *slash character* "/" is protected by *backslash character* "\", then in this case it is considered part of a name in the hierarchy. As an example, the path "`mysubsystem\/name/myproc/write`" has three elements: "`mysubsystem/name`", "`myproc`" and "`write`".

The *backslash character* "\" can be protected with another *backslash character* "\". As an example, the path "`mysub\\/name/myproc/write`" has four elements: "`mysub\`", "`name`", "`myproc`" and "`write`".

All others characters are considered part of a path name. A local path refers to a local object, in the form "`objectname/methodname`". A global path refers to an object in any place of the hierarchy, in the form "`name/.../name`".

8.11.4. METHOD

A method in general can be thought as a function that implement some part of the behavior of an object. For example, a `PROC` implementing a control algorithm may have different methods each one implementing a different version of the same control rule. A `VAR` can export different methods that allow to access a shared data structure in different ways.

A `METHOD` element has only one attribute:

attribute Name this attribute specifies the `METHOD` name. Method names are *unique* names for each object (as `PROC` and `VAR`).

8.11.5. METHODREF

A `METHODREF` is used to explicitly model the fact that a PROC α has some relationship with another `PROC` β because α is calling a `METHOD` exported by β. A `METHODREF` requires the following attributes:

attribute Name this attribute specifies a method inside the functional section. Please remind that method names are unique in all the functional section.

attribute MethodName this attribute contains a path to a `METHOD`. Paths inside a `METHODREF` are local pathnames in the form "objectname/methodname" in the case of `PROC` and `REQUIRED_INTERFACE`; are global pathnames in the case of `TRIGGER`, and `RESOURCEREF`.

The `METHODREF` does not give any kind of information about how many times the method pointed by the `METHODREF` is called.

8.11.6. EVENTREF

`EVENTREF` is an element used to reference an `EVENT`. An `EVENTREF` has only one attribute:

attribute Name the Name attribute is the name of the `EVENT` which the `EVENTREF` is referred to.

8.11.7. MUTEXREF

`MUTEXREF` is an element used to reference a `MUTEX`. It is mainly used inside `RESOURCES` (see Section 8.7.3 and Figure 8.2). A `MUTEXREF` has two attributes:

attribute MutexName the `MUTEX` name to which this element refers. MutexName is not a pathname, because all the references to mutexes can be done only in the same namespace of the entity that requires it.

attribute ModeRef is used to specify to which Mode the `MUTEXREF` object is related.

8.12. A complete example

As a complete example, consider the example number 5 (under the directory `examples/005` of the RT-Druid directory tree). The example describes a model of a small toy composed by a little car with two motors directly connected to the wheels. The toy can read a shaded path using a light sensor, following the reference position in the path using a differential control algorithm. The toy example is composed by 2 subsystems:

Guida reads the data from an optical sensor, computing the optimal reference speed of the two motors following a control rule.

ControlloMotore tries to change the power set to the motors to minimize the difference between the optimal values computed by the `Guida` subsystem and the values read by the rotational sensors on the wheels.

All the `PROC` elements of the two subsystems are then mapped on three different `TASK`s:

irq_sensori this `TASK` element is an ISR that reads the light sensors with a period of 1 ms.

Guida this `TASK` element is a task that computes the optimal motor speed values with a period of 1 ms.

ControlloMotore this `TASK` element reads the rotational sensors on the wheels and controls the motor speed trying to reach the value of the optimal speed. The Period is set to 0.05 ms.

Finally, two mutexes, `MutexIngressi` and `MutexVel` are defined to allow data exchange between the various `TASK` elements.

A. History

Version	Comment
1.2.x	Versions for Nios II 5.0.
1.3.0	Minor updates for Nios II 5.1.
1.3.1	Updated Mutex information for Nios II 6.0.
1.3.2	Added ANT scripting chapter.
1.4.1	Moved OIL content to respective architecture manuals (mainly Nios II content).
1.4.2	Completely rewritten the OIL description. Included support for PIC devices. Added standalone version.
1.4.3	Added EDF kernel attributes.
1.4.4	Typos.
1.4.5	Added AVR architecture details.
1.4.6	Added note on the library section.
1.4.7	Better explained library section. Now more than one library can be specified. Updated OIL definition.
1.4.8	Updated OIL definition. Erika Enterprise Basic renamed to Erika Enterprise.
1.4.9	Updated OIL for Nios II 8.0. Updated screenshots.
1.5.0	Updated OIL for EE 1.5.0, added ARM7, Nios II 8.1. FRSH.
1.6.0	Document restyle, with a description of the schedulability analysis plugins.

B. Script File

B.1. Introduction

The RT-Druid Toolset provides an uniform way of running a batch computation, that supports both non-GUI usage (useful for automatic code generation) and an usage that takes advantage of the Eclipse GUI . To do that, the RT-Druid Toolset enhanced the support for the Apache Ant build tool. Apache Ant has been chosen because:

- it allows useful scripting with an expressive power similar to conventional makefiles;
- it is expandible with customized features;
- it supports seamlessy integration with the Eclipse platform, which has been used as base for the graphical environment of the RT-Druid Toolset.

It is out of the scope of this section to describe Ant in detail. For more informations about Apache Ant, please refer to `http://ant.apache.org`; a detailed manual of Ant can be found at `http://ant.apache.org/manual/intro.html`.

B.2. Quick introduction to ANT

Ant is a Java-based build tool produced by the Apache Foundation[1]. Ant build files are somehow similar to makefiles, except that they are written as XML files.

Targets. As in makefiles, Ant can handle "targets". Each buildfile contains one project and at least one (default) target. Targets contains (provoke the execution of) tasks. Each task is run by a Java object that implements a particular Task interface. A target can depend on other targets. You might have a target for compiling, for example, and a target for creating a distributable. In that sense, "target"s are similar to makefile "rules". You can only build a distributable when you have compiled first, so the "distribute" target depends on the "compile" target. Ant resolves these dependencies.

Tasks. A task is a piece of code that can be executed. A task can have multiple attributes (or arguments, if you prefer). The value of an attribute might contain references to a property. These references will be resolved before the task is executed. A task is more or less a method of a Java class inside Ant that run whenever it has to be executed.

[1]Small parts of this section are derived from the manual available at `http://ant.apache.org/manual/intro.html`. Please refer to it for a complete explaination.

We can think as a shell command inside a makefile rule is like the invocation of an Ant Task whose behavior is the same as the command. The RT-Druid Toolset expanded the Ant tasks providing a set of new tasks for doing, for example, schedulability analysis and code generation.

Types. A Type is used whenever the user needs to provide informations to a task, or have to represent list of items, files, and so on. For example, the execution of the `delete` task (for example inside a `clean` target) needs the list of files to be deleted, that is passed using a type.

Properties. A project can have a set of properties. These might be set in the buildfile by the property task, or might be set outside Ant. A property has a name and a value; the name is case-sensitive. Properties may be used in the value of task attributes. This is done by placing the property name between `${` and `}` in the attribute value. For example, if there is a `builddir` property with the value `build`, then this could be used in an attribute like this: `${builddir}/classes`. This is resolved at run-time as build/classes. Properties are very similar to makefile variables.

Case sensitivity. In general, the following rules apply for case sensitivity:

- XML element tags in the script files are case sensitive;
- attribute values surrounded by brackets are case sensitive;
- if attribute values refer to file names, the rules of the host OS applies (e.g. Windows is case insensitive, whereas Linux is case sensitive);
- case sensitive words are written in the manual using a `typewriter` font;

B.3. Launching Ant

Ant can be launched from the command line. For example, to execute the script `build.xml` with a parameter `all` as target (the same as doing `make all` with makefiles), you should execute the following command (this command works on Windows; we used “\” to split the lines which are too long):

```
set CLASSPATH="C:\Programmi\j2sdk1.4.2_05\lib\tools.jar; \
    C:\Programmi\Evidence\eclipse\startup.jar"
java org.eclipse.core.launcher.Main                       \
     -application org.eclipse.ant.core.antRunner all      \
     -buildfilebuild.xml
```

that is, the `antRunner` plugin of Eclipse is started, asking to execute the `build.xml` script file running the `all` target. If the `-buildfile` option is not specified, the default build file is `build.xml`.

B.4. An Ant build script file format example

An Ant build script is currently composed by an XML file that (more or less) sequentially lists the commands run by the Ant. This section does not describe in detail the Ant syntax. For details about the syntax of an Ant build script, please refer to http://ant.apache.org/manual/intro.html.

The following lines show an Ant script that uses some of the RT-DruidToolset features.

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="rtdruid" default="all" basedir=".">

  <target name="all">
    <rtdruid.Oil.Configurator
      inputfile="conf.oil"
      outputdir="Debug"
    />
  </target>
</project>
```

A project named `rtdruid` is declared, with one target named `all`.

Target `all` calls a `rtdruid.Oil.Configurator` task, which performs the code generation for a given OIL file. In particular, the target `all` reads the file `conf.oil` stored in the current directory, and write the outputs inside the `Debug` directory.

B.5. Task "rtdruid.Oil.Configurator"

This task can be used to perform the code generation from an OIL file. The task accepts two parameters, which are `inputfile` (an input OIL file) and `outputdir` (the output directory where the generated files should be stored). If the output directory is created if it does not alreday exist.

The typical usage of this ANT task is to exexute it inside the application working directory, generating the configuration files in a subdirectory. Then, the user can run the makefile which has been generated to compile the application.

An example of an `rtdruid.Oil.Configurator` task is the following:

```
...
    <rtdruid.Oil.Configurator
      inputfile="conf.oil"
      outputdir="Debug"
    />
...
```

C. OIL definition

```
OIL_VERSION = "2.4";

IMPLEMENTATION ee {
  OS {
    STRING EE_OPT[];
    STRING CFLAGS[];
    STRING ASFLAGS[];
    STRING LDFLAGS[];
    STRING LDDEPS[];
    STRING LIBS[];

    ENUM [STANDARD, EXTENDED] STATUS = STANDARD;

    BOOLEAN STARTUPHOOK = FALSE;
    BOOLEAN ERRORHOOK = FALSE;
    BOOLEAN SHUTDOWNHOOK = FALSE;
    BOOLEAN PRETASKHOOK = FALSE;
    BOOLEAN POSTTASKHOOK = FALSE;
    BOOLEAN USEGETSERVICEID = FALSE;
    BOOLEAN USEPARAMETERACCESS = FALSE;
    BOOLEAN USERESSCHEDULER = TRUE;

    BOOLEAN STARTUPSYNC = FALSE;
    ENUM [ALWAYS, IFREQUIRED] USEREMOTETASK = IFREQUIRED;
    STRING MP_SHARED_RAM = "";
    STRING MP_SHARED_ROM = "";

    STRING NIOS2_MUTEX_BASE;
    STRING IPIC_GLOBAL_NAME;
    STRING NIOS2_SYS_CONFIG;
    STRING NIOS2_APP_CONFIG;
    BOOLEAN NIOS2_DO_MAKE_OBJDUMP = FALSE;
    STRING NIOS2_JAM_FILE;
    STRING NIOS2_PTF_FILE;

    STRING MASTER_CPU = "default_cpu";
```

```
 ENUM [
      ARM7 {
     STRING ID = "default_cpu";
     STRING APP_SRC[];

     STRING THUMB_SRC[];

     BOOLEAN [
       TRUE {
         BOOLEAN [
              TRUE {
             UINT32 SYS_SIZE;
           },
           FALSE
         ] IRQ_STACK;
         ENUM [
           SHARED,
           PRIVATE {
             UINT32 SYS_SIZE;
           }
         ] DUMMY_STACK;
       },
       FALSE
     ] MULTI_STACK = FALSE;

     UINT32 STACK_TOP;
     UINT32 STACK_BOTTOM;
     UINT32 SYS_SIZE;
     UINT32 IRQ_SIZE;
     UINT32 SVC_SIZE;
     UINT32 FIQ_SIZE;
     UINT32 ABT_SIZE;
     UINT32 UND_SIZE;
     UINT32 SHARED_MIN_SYS_SIZE; // size of shared stack

   },
NIOSII {
     STRING ID = "default_cpu";
     STRING APP_SRC[];

     BOOLEAN [
       TRUE {
         BOOLEAN [
           TRUE {
            UINT32 SYS_SIZE;
          },
          FALSE
        ] IRQ_STACK;
        ENUM [
          SHARED,
          PRIVATE {
            UINT32 SYS_SIZE;
          }
        ] DUMMY_STACK;
      },
      FALSE
    ] MULTI_STACK = FALSE;

    STRING STACK_TOP;
    UINT32 SYS_SIZE;
    UINT32 SHARED_MIN_SYS_SIZE;

    STRING SYSTEM_LIBRARY_NAME;
    STRING SYSTEM_LIBRARY_PATH;

    STRING IPIC_LOCAL_NAME;


    STRING TIMER_FREERUNNING;
    ENUM [
      SINGLE {
        STRING TIMER_IRQ;
      },
      MULTIPLE {
           STRING TIMER_IRQ_BUDGET;
           STRING TIMER_IRQ_RECHARGE;
           STRING TIMER_IRQ_DLCHECK;
           STRING TIMER_IRQ_SEM;
      }
    ] FRSH_TIMERS;

  },
PIC30 {
    STRING ID = "default_cpu";
    STRING APP_SRC[];

    BOOLEAN [
      TRUE {
        BOOLEAN [
          TRUE {
            UINT32 SYS_SIZE;
          },
          FALSE
        ] IRQ_STACK;
      },
      FALSE
    ] MULTI_STACK = FALSE;

    BOOLEAN ICD2 = FALSE;
    BOOLEAN ENABLE_SPLIM = TRUE;

   },
AVR_5 {
    STRING ID = "default_cpu";
    STRING APP_SRC[];

    BOOLEAN [
      TRUE {
        BOOLEAN [
          TRUE {
            UINT32 SYS_SIZE;
          },
          FALSE
        ] IRQ_STACK;
        ENUM [
          SHARED,
          PRIVATE {
            UINT32 SYS_SIZE;
          }
        ] DUMMY_STACK;
      },
      FALSE
    ] MULTI_STACK = FALSE;

    UINT32 STACK_BOTTOM;

    UINT32 SYS_SIZE;  // available space for all user stacks
    UINT32 SHARED_MIN_SYS_SIZE; // size of shared stack

    ENUM [STOP, DIV1, DIV8, DIV32, DIV64, DIV256, DIV1024] TIMER0 = STOP;
    ENUM [STOP, DIV1, DIV8,     DIV64, DIV256, DIV1024] TIMER1 = STOP;
    ENUM [STOP, DIV1, DIV8,     DIV64, DIV256, DIV1024] TIMER2 = STOP;
    ENUM [STOP, DIV1, DIV8,     DIV64, DIV256, DIV1024] TIMER3 = STOP;



    /* Interrupt Handlers * /
    STRING HANDLER_IRQ0;// external interrupt request 0
    STRING HANDLER_IRQ1;// external interrupt request 1
    STRING HANDLER_IRQ2;// external interrupt request 2
    STRING HANDLER_IRQ3;// external interrupt request 3
    STRING HANDLER_IRQ4;// external interrupt request 4
    STRING HANDLER_IRQ5;// external interrupt request 5
    STRING HANDLER_IRQ6;// external interrupt request 6
    STRING HANDLER_IRQ7;// external interrupt request 7

    STRING HANDLER_T0_MATCH;   // Timer/Counter 0 Compare Match
    STRING HANDLER_T0_OVERFLW; // Timer/Counter 0 Overflow
    STRING HANDLER_T1_EVENT;   // Timer/Counter 1 Capture Event
    STRING HANDLER_T1_MATCH_A; // Timer/Counter 1 Compare Match A
    STRING HANDLER_T1_MATCH_B; // Timer/Counter 1 Compare Match B
    STRING HANDLER_T1_MATCH_C; // Timer/Counter 1 Compare Match C
    STRING HANDLER_T1_OVERFLW; // Timer/Counter 1 Overflow
    STRING HANDLER_T2_MATCH;   // Timer/Counter 2 Compare Match
    STRING HANDLER_T2_OVERFLW; // Timer/Counter 2 Overflow
    STRING HANDLER_T3_EVENT;   // Timer/Counter 3 Capture Event
    STRING HANDLER_T3_MATCH_A; // Timer/Counter 3 Compare Match A
    STRING HANDLER_T3_MATCH_B; // Timer/Counter 3 Compare Match B
    STRING HANDLER_T3_MATCH_C; // Timer/Counter 3 Compare Match C
    STRING HANDLER_T3_OVERFLW; // Timer/Counter 3 Overflow

    STRING HANDLER_SPI; // SPI Serial Transfer Complete

    STRING HANDLER_US0_RX;     // USART0 Rx complete
    STRING HANDLER_US0_EMPTY;  // USART0 Data Register Empty
    STRING HANDLER_US0_TX;     // Usart0 Tx complete

    STRING HANDLER_US1_RX;     // USART1 Rx complete
    STRING HANDLER_US1_EMPTY;  // USART1 Data Register Empty
    STRING HANDLER_US1_TX;     // Usart1 Tx complete

    STRING HANDLER_ADC; // ADC Conversion Complete
    STRING HANDLER_EEPROM;     // EEPROM Ready
    STRING HANDLER_ANALOG_COMP;// Analog Comparator
    STRING HANDLER_2WSI;// Two-wire serial Interface
    STRING HANDLER_SPM_READY;  // Store program Memory Ready
```

```
*/

ENUM [
  HANDLER_IRQ0 {    // external interrupt request 0
    STRING FUNCTION;
    INT32[1,2] TYPE = 1;
  }, HANDLER_IRQ1 { // external interrupt request 1
    STRING FUNCTION;
    INT32[1,2] TYPE = 1;
  }, HANDLER_IRQ2 { // external interrupt request 2
    STRING FUNCTION;
    INT32[1,2] TYPE = 1;
  }, HANDLER_IRQ3 { // external interrupt request 3
    STRING FUNCTION;
    INT32[1,2] TYPE = 1;
  }, HANDLER_IRQ4 { // external interrupt request 4
    STRING FUNCTION;
    INT32[1,2] TYPE = 1;
  }, HANDLER_IRQ5 { // external interrupt request 5
    STRING FUNCTION;
    INT32[1,2] TYPE = 1;
  }, HANDLER_IRQ6 { // external interrupt request 6
    STRING FUNCTION;
    INT32[1,2] TYPE = 1;
  }, HANDLER_IRQ7 { // external interrupt request 7
    STRING FUNCTION;
    INT32[1,2] TYPE = 1;

  }, HANDLER_T0_MATCH {    // Timer/Counter 0 Compare Match
    STRING FUNCTION;
    INT32[1,2] TYPE = 1;
  }, HANDLER_T0_OVERFLW {  // Timer/Counter 0 Overflow
    STRING FUNCTION;
    INT32[1,2] TYPE = 1;
  }, HANDLER_T1_EVENT {    // Timer/Counter 1 Capture Event
    STRING FUNCTION;
    INT32[1,2] TYPE = 1;
  }, HANDLER_T1_MATCH_A {  // Timer/Counter 1 Compare Match A
    STRING FUNCTION;
    INT32[1,2] TYPE = 1;
  }, HANDLER_T1_MATCH_B {  // Timer/Counter 1 Compare Match B
    STRING FUNCTION;
    INT32[1,2] TYPE = 1;
  }, HANDLER_T1_MATCH_C {  // Timer/Counter 1 Compare Match C
```

```
    STRING FUNCTION;
    INT32[1,2] TYPE = 1;
  }, HANDLER_T1_OVERFLW {  // Timer/Counter 1 Overflow
    STRING FUNCTION;
    INT32[1,2] TYPE = 1;
  }, HANDLER_T2_MATCH {    // Timer/Counter 2 Compare Match
    STRING FUNCTION;
    INT32[1,2] TYPE = 1;
  }, HANDLER_T2_OVERFLW {  // Timer/Counter 2 Overflow
    STRING FUNCTION;
    INT32[1,2] TYPE = 1;
  }, HANDLER_T3_EVENT {    // Timer/Counter 3 Capture Event
    STRING FUNCTION;
    INT32[1,2] TYPE = 1;
  }, HANDLER_T3_MATCH_A {  // Timer/Counter 3 Compare Match A
    STRING FUNCTION;
    INT32[1,2] TYPE = 1;
  }, HANDLER_T3_MATCH_B {  // Timer/Counter 3 Compare Match B
    STRING FUNCTION;
    INT32[1,2] TYPE = 1;
  }, HANDLER_T3_MATCH_C {  // Timer/Counter 3 Compare Match C
    STRING FUNCTION;
    INT32[1,2] TYPE = 1;
  }, HANDLER_T3_OVERFLW {  // Timer/Counter 3 Overflow
    STRING FUNCTION;
    INT32[1,2] TYPE = 1;

  }, HANDLER_SPI {  // SPI Serial Transfer Complete
    STRING FUNCTION;
    INT32[1,2] TYPE = 1;

  }, HANDLER_US0_RX {      // USART0 Rx complete
    STRING FUNCTION;
    INT32[1,2] TYPE = 1;
  }, HANDLER_US0_EMPTY {   // USART0 Data Register Empty
    STRING FUNCTION;
    INT32[1,2] TYPE = 1;
  }, HANDLER_US0_TX {      // Usart0 Tx complete
    STRING FUNCTION;
    INT32[1,2] TYPE = 1;

  }, HANDLER_US1_RX {      // USART1 Rx complete
    STRING FUNCTION;
    INT32[1,2] TYPE = 1;
```

```
      }, HANDLER_US1_EMPTY {   // USART1 Data Register Empty
        STRING FUNCTION;
        INT32[1,2] TYPE = 1;
      }, HANDLER_US1_TX {      // Usart1 Tx complete
        STRING FUNCTION;
        INT32[1,2] TYPE = 1;

      }, HANDLER_ADC {  // ADC Conversion Complete
        STRING FUNCTION;
        INT32[1,2] TYPE = 1;
      }, HANDLER_EEPROM {// EEPROM Ready
        STRING FUNCTION;
        INT32[1,2] TYPE = 1;
      }, HANDLER_ANALOG_COMP { // Analog Comparator
        STRING FUNCTION;
        INT32[1,2] TYPE = 1;
      }, HANDLER_2WSI { // Two-wire serial Interface
        STRING FUNCTION;
        INT32[1,2] TYPE = 1;
      }, HANDLER_SPM_READY {   // Store program Memory Ready
        STRING FUNCTION;
        INT32[1,2] TYPE = 1;
      }

    ] HANDLER[];
  }

] CPU_DATA[];

ENUM [

  SAMSUNG_KS32C50100 {
    STRING IRQ_EXT0;                /* Ext 0 */
    STRING IRQ_EXT1;                /* Ext 1 */
    STRING IRQ_EXT2;                /* Ext 2 */
    STRING IRQ_EXT3;                /* Ext 3 */
    STRING IRQ_UART0TX;             /* UART0 Tx */
    STRING IRQ_UART0RX;             /* UART0 Rx & error*/
    STRING IRQ_UART1TX;             /* UART1 Tx */
    STRING IRQ_UART1RX;             /* UART1 Rx & error*/
    STRING IRQ_GDMA0;               /* GDMA ch. 0 */
    STRING IRQ_GDMA1;               /* GDMA ch. 1 */
    STRING IRQ_TIMER0;              /* Timer 0 */
    STRING IRQ_TIMER1;              /* Timer 1 */
    STRING IRQ_HDLCATX;             /* HDLC A Tx */
    STRING IRQ_HDLCARX;             /* HDLC A Rx */
    STRING IRQ_HDLCBTX;             /* HDLC B Tx */
    STRING IRQ_HDLCBRX;             /* HDLC B Rx */
    STRING IRQ_ETHBDMATX;           /* Ethernet BDMA Tx */
    STRING IRQ_ETHBDMARX;           /* Ethernet BDMA Rx */
    STRING IRQ_ETHMACTX;            /* Ethernet MAC Tx */
    STRING IRQ_ETHMACRX;            /* Ethernet MAC Rx */
    STRING IRQ_I2C;                 /* I2C-bus */
    STRING IRQ_NO_PENDING;          /* No Pending Interrupt */
     },
  unibo_mparm {
    STRING IRQ_EXT0;                // Ext 0
    STRING IRQ_EXT1;                // Ext 1
    STRING IRQ_EXT2;                // Ext 2
    STRING IRQ_EXT3;                // Ext 3
    STRING IRQ_UART0TX;             // UART0 Tx
    STRING IRQ_UART0RX;             // UART0 Rx & error
    STRING IRQ_UART1TX;             // UART1 Tx
    STRING IRQ_UART1RX;             // UART1 Rx & error
    STRING IRQ_GDMA0;               // GDMA ch. 0
    STRING IRQ_GDMA1;               // GDMA ch. 1
    STRING IRQ_TIMER0;              // Timer 0
    STRING IRQ_TIMER1;              // Timer 1
    STRING IRQ_HDLCATX;             // HDLC A Tx
    STRING IRQ_HDLCARX;             // HDLC A Rx
    STRING IRQ_HDLCBTX;             // HDLC B Tx
    STRING IRQ_HDLCBRX;             // HDLC B Rx
    STRING IRQ_ETHBDMATX;           // Ethernet BDMA Tx
    STRING IRQ_ETHBDMARX;           // Ethernet BDMA Rx
    STRING IRQ_ETHMACTX;            // Ethernet MAC Tx
    STRING IRQ_ETHMACRX;            // Ethernet MAC Rx
    STRING IRQ_I2C;                 // I2C-bus
    STRING IRQ_NO_PENDING;          // No Pending Interrupt
     },
     PIC30 {
       ENUM [
         CUSTOM {
           STRING MODEL;
           STRING LINKERSCRIPT;
           STRING DEV_LIB;
           STRING INCLUDE_C;
           STRING INCLUDE_S;
         },
PIC24FJ128GA006, PIC24FJ128GA008,
PIC24FJ128GA010, PIC24FJ32GA002,
PIC24FJ32GA004, PIC24FJ64GA002,
PIC24FJ64GA004, PIC24FJ64GA006,
PIC24FJ64GA008, PIC24FJ64GA010,
PIC24FJ96GA006, PIC24FJ96GA008,
PIC24FJ96GA010, PIC24HJ128GP206,
PIC24HJ128GP210, PIC24HJ128GP306,
PIC24HJ128GP310, PIC24HJ128GP506,
PIC24HJ128GP510, PIC24HJ256GP206,
PIC24HJ256GP210, PIC24HJ256GP610,
PIC24HJ64GP206, PIC24HJ64GP210,
PIC24HJ64GP506, PIC24HJ64GP510,

PIC30F1010, PIC30F2010,
PIC30F2011, PIC30F2012,
PIC30F2020, PIC30F2021,
PIC30F2022, PIC30F2023,
PIC30F3010, PIC30F3011,
PIC30F3012, PIC30F3013,
PIC30F3014, PIC30F4011,
PIC30F4012, PIC30F4013,
PIC30F5011, PIC30F5013,
PIC30F5015, PIC30F5016,
PIC30F6010, PIC30F6010A,
PIC30F6011, PIC30F6011A,
PIC30F6012, PIC30F6012A,
PIC30F6013, PIC30F6013A,
PIC30F6014, PIC30F6014A,
PIC30F6015,

PIC33FJ128GP206, PIC33FJ128GP306,
PIC33FJ128GP310, PIC33FJ128GP706,
PIC33FJ128GP708, PIC33FJ128GP710,
PIC33FJ128MC506, PIC33FJ128MC510,
PIC33FJ128MC706, PIC33FJ128MC708,
PIC33FJ128MC710, PIC33FJ256GP506,
PIC33FJ256GP510, PIC33FJ256GP710,
PIC33FJ256MC510, PIC33FJ256MC710,
PIC33FJ64GP206, PIC33FJ64GP306,
PIC33FJ64GP310, PIC33FJ64GP706,
PIC33FJ64GP708, PIC33FJ64GP710,
PIC33FJ64MC506, PIC33FJ64MC508,
PIC33FJ64MC510, PIC33FJ64MC706,
        PIC33FJ64MC710
      ] MODEL;
    }
  ] MCU_DATA;

  ENUM [
NO_BOARD,
EVALUATOR7T,
EE_FLEX {
     BOOLEAN USELEDS = FALSE;
     BOOLEAN USELCD = FALSE;

     ENUM [
       DEMO {
          ENUM [
           ACCELEROMETER,
           ADC_IN,
           BUTTONS,
           BUZZER,
           DAC,
           ENCODER,
           IR,
           LCD,
           LEDS,
           PWM_OUT,
           PWM_MOTOR,
           SENSORS,
           TRIMMER,
           USB,
           ZIGBEE,

           ALL
         ] OPTIONS[];
       },
       MULTI {
         ENUM [
           ETHERNET,
           EIB,
           ALL
         ] OPTIONS[];
       },
       STANDARD {
         ENUM [
           LEDS, LCD, ALL
         ] OPTIONS[];
       }
     ] TYPE = STANDARD;
   },
MICROCHIP_EXPLORER16 {
        BOOLEAN USELEDS;
        BOOLEAN USEBUTTONS;
        BOOLEAN USELCD;
        BOOLEAN USEANALOG;
      }
MICROCHIP_DSPICDEM11PLUS {
        BOOLEAN USELEDS;
        BOOLEAN USEBUTTONS;
        BOOLEAN USELCD;
        BOOLEAN USEANALOG;
        BOOLEAN USEAUDIO;
      },
ATMEGA_STK50X,
XBOW_MIB5X0
 ] BOARD_DATA = NO_BOARD;

 ENUM [
   ENABLE {
     STRING NAME[];
   }
 ] LIB[];

 ENUM [
   FP {
     BOOLEAN NESTED_IRQ;
   },
   EDF {
     BOOLEAN NESTED_IRQ;
     STRING TICK_TIME;
     BOOLEAN REL_DEADLINES_IN_RAM = FALSE;
   },
   FRSH {
        ENUM [
          CONTRACT {
            STRING NAME;
            UINT32 BUDGET;
            UINT32 PERIOD;
            STRING CPU_ID;
          }
          ] CONTRACTS[];
          BOOLEAN USE_SYNC_OBJ;
        STRING TICK_TIME;
       }

       BCC1,
       BCC2,
       ECC1,
       ECC2
     ] KERNEL_TYPE;

     ENUM [
          NONE,
          ALL,
          OS_SECTION,
          TASK_SECTION,
          RESOURCE_SECTION,
          STACK_SECTION,
          ALARM_SECTION
     ] ORTI_SECTIONS[];

   };

APPMODE {
   };

TASK {
     BOOLEAN [
       TRUE {
         APPMODE_TYPE APPMODE[];
       },
       FALSE
     ] AUTOSTART;

     UINT32 PRIORITY;
     UINT32 RELDLINE;
     UINT32 ACTIVATION = 1;

     ENUM [NON, FULL] SCHEDULE;
     EVENT_TYPE EVENT[];
     RESOURCE_TYPE RESOURCE[];

     STRING CONTRACT;
```

```
      ENUM [
        SHARED,
        PRIVATE {
          UINT32 SYS_SIZE;
        }
      ] STACK = SHARED;
      STRING CPU_ID = "default_cpu";
      STRING APP_SRC[];

      TASK_TYPE LINKED[];
    };

RESOURCE {
      ENUM [
        STANDARD {
             STRING APP_SRC[];
        },
        LINKED {
          RESOURCE_TYPE LINKEDRESOURCE;
        },
        INTERNAL
      ] RESOURCEPROPERTY;
    };

EVENT {
      UINT32 WITH_AUTO MASK = AUTO;
    };

COUNTER {
      UINT32 MINCYCLE;
      UINT32 MAXALLOWEDVALUE;
      UINT32 TICKSPERBASE;

      STRING CPU_ID = "default_cpu";
    };

ALARM {
      COUNTER_TYPE COUNTER;
      ENUM [
        ACTIVATETASK {
          TASK_TYPE TASK;
        },
        SETEVENT {
          TASK_TYPE TASK;
          EVENT_TYPE EVENT;
        },
        ALARMCALLBACK {
          STRING ALARMCALLBACKNAME;
        }
      ] ACTION;

      BOOLEAN [
        TRUE {
          UINT32 ALARMTIME;
          UINT32 CYCLETIME;
          APPMODE_TYPE APPMODE[];
        },
        FALSE
      ] AUTOSTART;
    };
  };
```

Bibliography

[1] Eclipse Consortium. C/c++ development tools (CDT). http://www.eclipse.org/cdt, 2005.

[2] Eclipse Consortium. The eclipse modeling framework (EMF). http://www.eclipse.org/emf, 2005.

[3] Eclipse Consortium. The eclipse platform. http://www.eclipse.org, 2005.

[4] Eclipse Consortium. The graphical editing framework (GEF). http://www.eclipse.org/gef, 2005.

[5] OSEK/VDX Consortium. OSEK OIL standard. http://www.osek-vdx.org, 2005.

[6] Altera Corporation. Creating multiprocessor nios ii systems tutorial. Nios II literature page, http://www.altera.com/literature/lit-nio2.jsp, 2005.

[7] The Apache Software Foundation. The apache ant project. http://ant.apache.org, 2005.

[8] Lauterbach GMBH. The Lauterbach Trace32 Debugger for Nios II. http://www.lauterbach.com, 2005.

[9] Object Management Group, editor. *UML Profile for Schedulability, Performance, and Time*. OMG document ptc/02-03-02, 2002.