RT-Druid Code generator Plugin reference manual

A tool for the design of embedded real-time systems

version: 1.4.9
March 5, 2009

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 64/A
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 the RT-Druid Code generator tool and the associated Erika Enterprise Kernel.

The Code generator tool is part of the RT-Druid design framework for architecture level modeling. The RT-Druid toolset 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 modular structure of RT-Druid is now fully integrated with the open Eclipse framework. The Eclipse environment can easily be extended with third party components and plug-ins, further improving the configurability and extensibility of RT-Druid.

This user guide document covers the code generation plugin. It consists of an overview, explaining the architectural concepts, the standards and the inputs and outputs of the code generator tool. In the first section: Overview of RT-Druid Code Generator and Erika Enterprise, the tool and the real-time Erika Enterprise OS are introduced, the code generation process is outlined and the relationships among the products and the Eclipse development environment are explained.

The second part contains the basic information for operating with the tool and providing the right configuration input for the following code generation phase. 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 multiprocessor 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.

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

RT-Druid is an open and extensible environment, based on XML and open standards (Java) allowing generation of portable OSEK C code from OIL definitions to create applications that run in real-time in a variety of environments, including dsPIC, AVR, ARM7, Altera Nios II, and others.

Generated code can run on any OSEK-compliant system, but the RT-Druid framework is optimized for running in conjunction with the Erika Enterprise kernel.

Because of its generic framework, the RT-Druid give a extensible modeling and analysis platform for modeling any hardware and software, providing compatibility with most of the model-based methodologies for functional design on the market. The tool 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 purchased and 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 2000/Xp, Linux, Macintosh).

Extensibility: future extensions include custom plug-ins and integration with third party production tools for code generation complying with industrial standards: such as the OSEK and its related standards (OIL, ORTI) in the automotive domain. In particuler, future enhancements include:

- Graphic interface for placement and configuration options
- Support for the ORTI standard for debugging
- Support for the Lauterbach tools, for tracing and measurement of time-related attributes.

2.1. Software design with RT-Druid

The architecture of the RT-Druid family of tools is shown in Figure 2.1. 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 for multiprocessors (including extensions for Altera programmable HW);

Future extensions include

- Trace viewer/Analyzer;
- Scheduling Simulator;
- Importers from Mathworks Simulink, ASCET and UML 2.0 models.

2.2. 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.

2.3. 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.

Similarly, operation for creating a new project, as shown in the following Chapter 3 and 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).

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

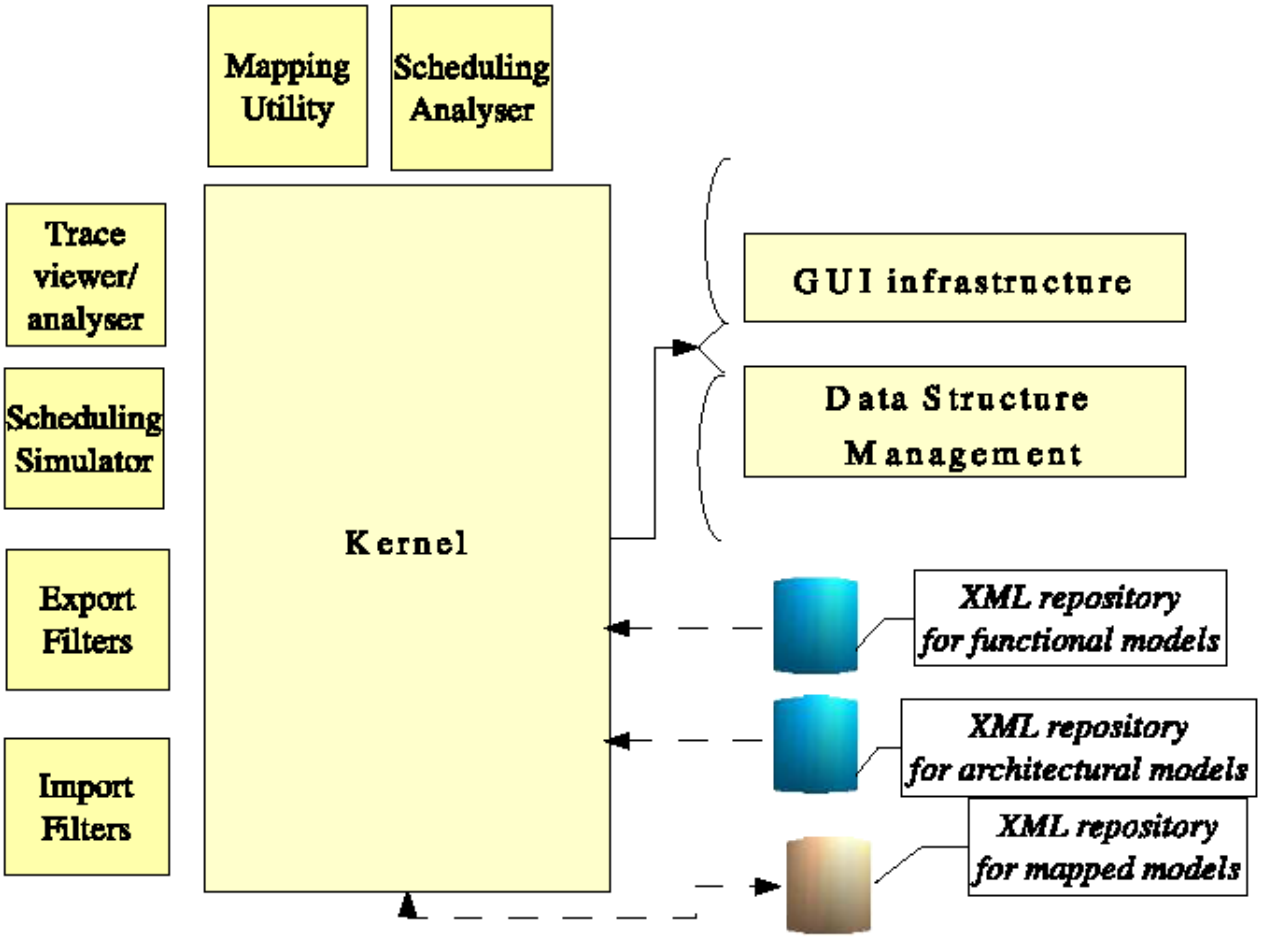

Figure 2.1.: The plug-in architecture of RT-Druid.

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.

2.4. Multiprocessor Version

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

- Multiprocessor systems with shared memory.
- Support for code placement in a multiprocessor 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

2.5. Multiprocessor extensions for Altera Nios II (Target specific info)

The multiprocessor extensions allow for processor specific features including compatibility with multicore targets like Altera Nios II, allowing reuse of standard Altera peripherals and device drivers. Integration with Altera's development tools, including Nios II IDE is also supported: basically, Eclipse CDT is extended by RT-Druid in order to allow creation and manegement of projects for multiprocessor systems based on Erika Enterprise.

2.5.1. Integration with Nios II IDE

The required commands and action sequences for editing, compiling, debugging and running code produced in the RT-Druid framework are identical to those common to Nios II IDE projects.

Integration with the Nios II IDE is not yet fully automated. In the description of the system, RT-Druid requires information related to the HW architecture running the system, such as the number of available CPUs and the available RAM addresses. In the

current version, this information needs to be provided by the user by writing suitable entries inside the configuration file, and it needs to be consistent with the definitions provided inside Altera's SOPCbuilder.

The compilation stage, however, offers full integration with the Nios II IDE environment. The compilation stage is handled by Eclipse, and in particular by CDT, starting from the scripts generated by RT-Druid which, in turn, exploits the compilation environment made available by Altera Nios II IDE. The end-result is a sequence of steps identical to those performed when commanding from the Eclipse interface the compilation of a standard Nios II IDE project, with the only difference that an RT-Druid project allows handling code distributed among multiple processors, whereas standard, automatically handled Nios II IDE projects assume deployment of a project for each CPU.

2.6. 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[2]. 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.

On a multicore system, the 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.

Creation of the CPU build directories. Following the creation of the project main directory, the RT-Druid Code Generator creates a directory for each CPU inside the it. If the system for which the project is compiled is a single-processor, only one folder is created. The name of the cpu folders follows the name/ID definitions provided in the OIL file for the CPUs[3].

All CPU folders contain the same files, with the exception of the folder for the Master CPU[4], which contains the extra file `common.c`, containing information common to all

[2]For Altera Nios II users: That directory has the same meaning of those created by Altera Nios II IDE projects for the parameter configurations, like `Debug`, `Release`, and so on.

[3]CPU names/IDs **must not** contain spacing characters

[4]For informations about the Master CPU, please refer to Section 4.11.2

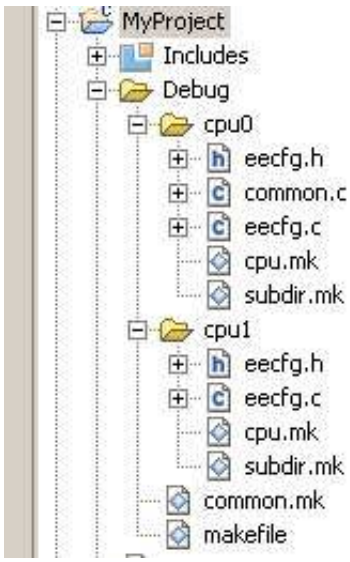

Figure 2.2.: Folders and files created by RT-Druid for an Altera Nios II multicore design.

the CPUs in the system. In the examples shipped with Erika Enterprise `cpu0` is the predefined ID for the Master CPU.

For each CPU the Code generator produces the following files (see Figure 2.2):

`eecfg.h`. This file contains the declarations of all the RTOS symbols (tasks, resources, alarms, events, and so on) that are visible from the given CPU. The objects visible from a CPU are the objects allocated on it, plus the objects on other CPUs that may be referred by the code running on the CPU itself.

`eecfg.c`. This file contains the configuration data structures of the Erika Enterprise kernel, providing information on the OIL file local objects options.

`cpu.mk`. This file contains the rules used to compile the source code allocated to the CPU. Together with the `common.mk` file it provides information equivalent to the contents of the `makefile` created by the Nios II IDE C/C++ Application Projects.

`subdir.mk`. This file contains the list of the files that must be compiled and linked in order to generate the executable to be run on the CPU. The files depend on the partitioning configuration defined in the OIL file.

> **Warning:** To be included inside `subdir.mk`, a file needs to be listed inside the OIL declaration. The behavior is different from normal Altera Nios II Projects where the fact that a file is inserted in a project implies that it will be automatically added to the `subdir.mk` and compiled.

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 in in the Navigation sidebar, and a dedicated OIL Editor will appear.

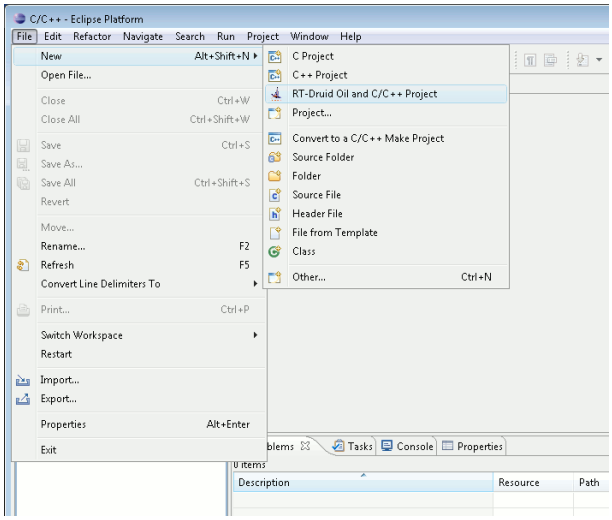

Figure 3.1.: Activating the "New Project" Wizard.

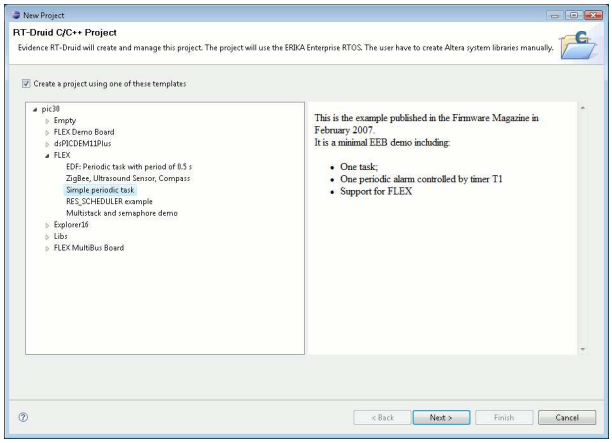

Figure 3.2.: Choosing a template application for your new RT-Druid Project.

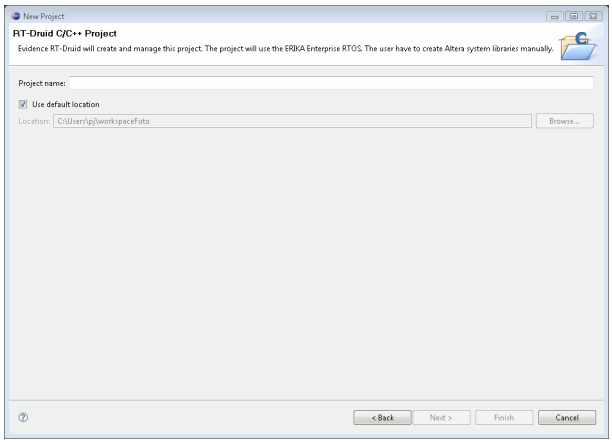

Figure 3.3.: Choosing a meaningful name for your new RT-Druid Project.

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 exists 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 `TASK`s.

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 A. 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 `#define`s 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 4.5.9), 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. Basicaly, 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 which 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;
  },
  BCC1,
  BCC2,
  ECC1,
  ECC2
] KERNEL_TYPE;
```

For the EDF kernel, it is possible to specify the tick length. given the tick length for the circular timer, which is then used to compute the vaues to put on 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.

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

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`.

4.3.10. 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.

RT-Druid provides the possibility to automatically generate an ORTI[1] 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[2] 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;
  }
  ...
}
```

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

[2] 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).

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.

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 whithout any time unit (e.g. "1234" and not "1234ms"), then the number is taken "as is" and then 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 or non preemptive.

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 belongs 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. 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;
    };
  };
  ...
}
```

4.5.9. 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";
    ...
  };
  ...
}
```

4.5.10. 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";
    ...
  }
  ...
}
```

4.5.11. 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 4.5.9 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"[3]. Here is the definition of the `MASTER_CPU` attribute:

```
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.

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

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.

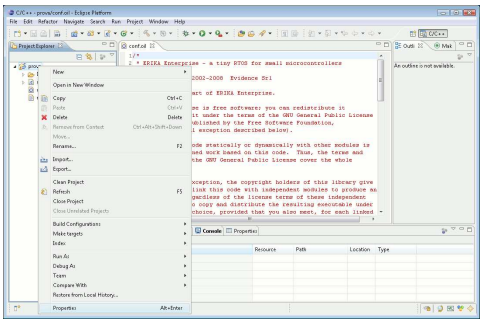

Figure 5.1.: Opening the Project Properties.

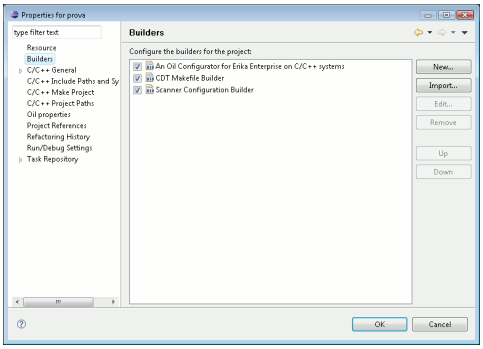

Figure 5.2.: The Builders property page. All checkboxes are selected by default.

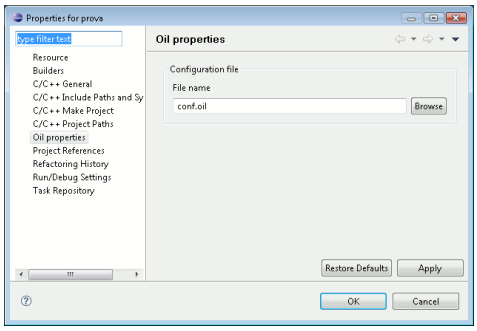

Figure 5.3.: Changing the current OIL file for code generation.

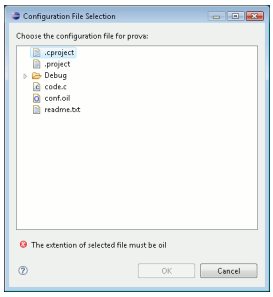

Figure 5.4.: Choosing an existing OIL file by browsing the filesystem.

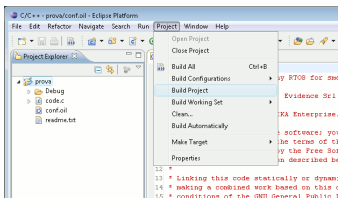

Figure 5.5.: The “Build Project” option from the “Project” menu. Please note that the “Build Automatically” option is not selected.

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.

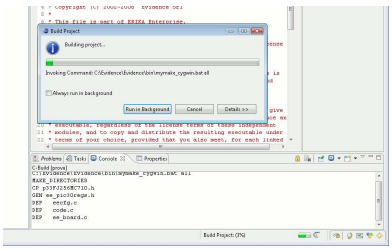

Figure 5.6.: The Build progress window.

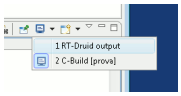

Figure 5.7.: Displaying the RT-Druid Console.

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.

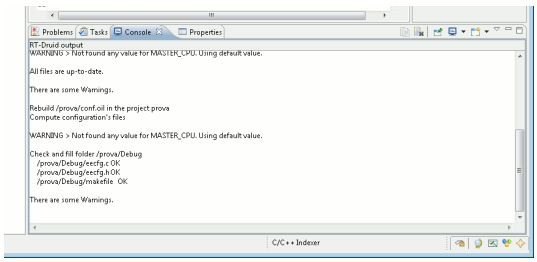

Figure 5.8.: The RT-Druid console.

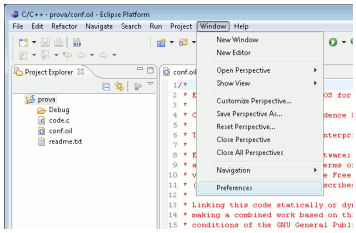

Figure 5.9.: The "Windows/Preferences" menu item.

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.

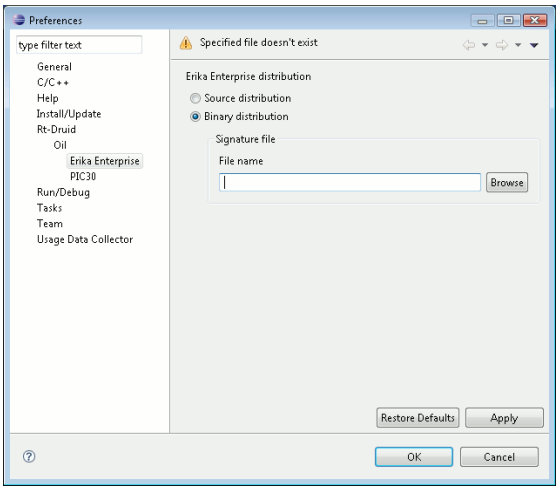

Figure 5.10.: The OS Configurator dialog box.

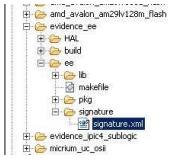

Figure 5.11.: The default location of the signature file.

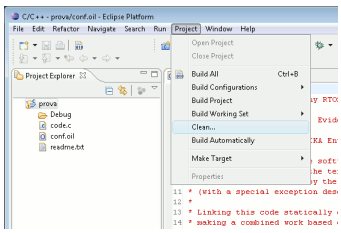

Figure 5.12.: The “Clean...” command on the Project menu.

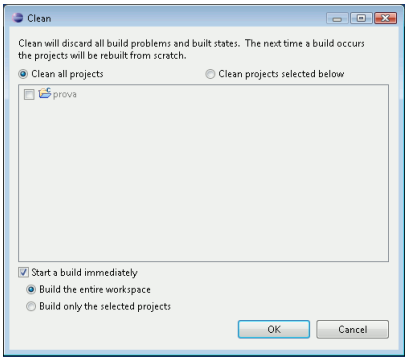

Figure 5.13.: The Clean dialog box. Please note the bottom left checkbox.

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.

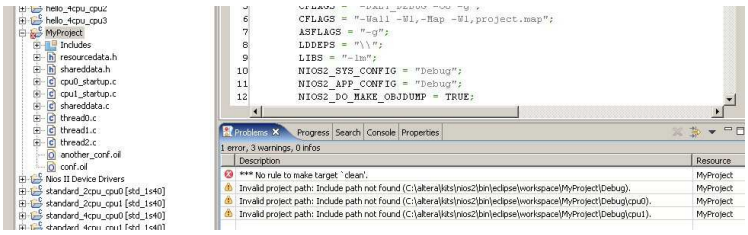

Figure 5.15.: The Navigation toolbar after the Project clean. The errors in the Problem window can be ignored, because they refer to files that have been removed and that will be automatically created by RT-Druid at Build time.

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

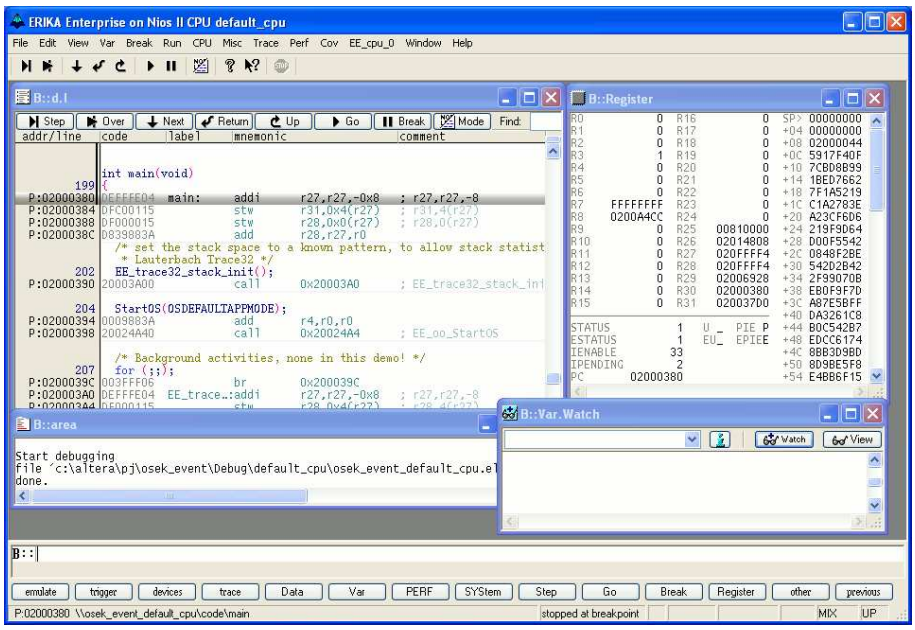

Figure 5.16.: The Lauterbach Trace32 for Altera Nios II.

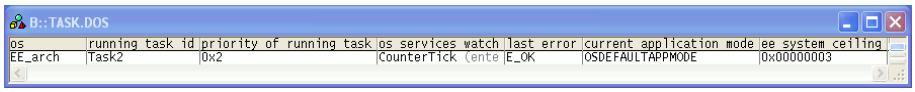

Figure 5.17.: General information about the Erika Enterprise status.

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.

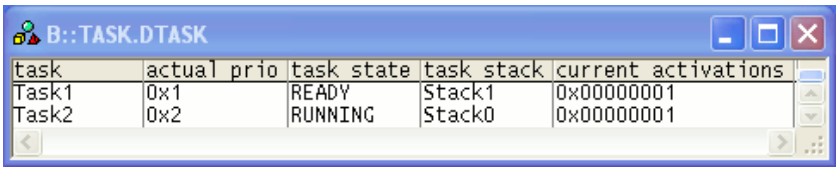

Figure 5.18.: Information about the tasks in the system.

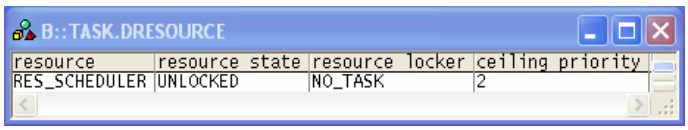

Figure 5.19.: Information about the resources in the system.

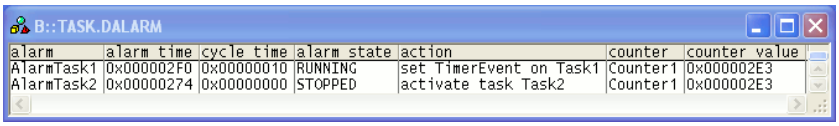

Figure 5.20.: Information about the alarms in the system.

- 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.
- Finally, 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.

The RT-Druid and Erika Enterprise Trace32 support also includes support for the Nios II tracer module. As an example, Figure 5.23 shows the execution of an interrupt handler as recorded by the tracer module. Figure 5.24 shows an interpretation of the context changes and task status values using the ORTI information.

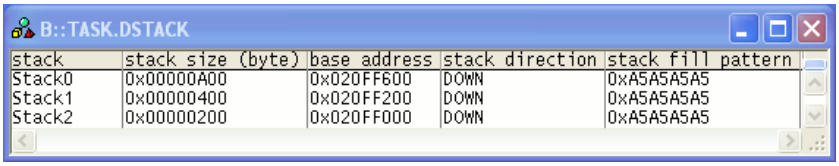

Figure 5.21.: The application stack list.

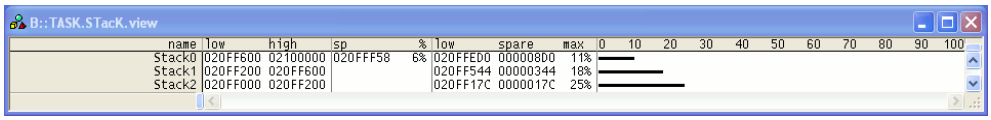

Figure 5.22.: A graphical view of the application stack usage.

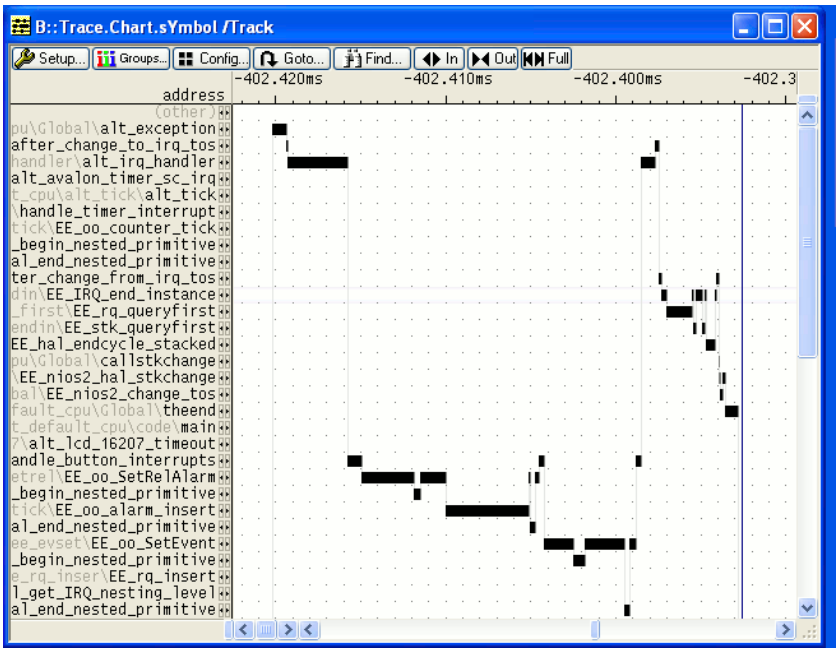

Figure 5.23.: The execution of the Button IRQ as recorded by Lauterbach Trace32.

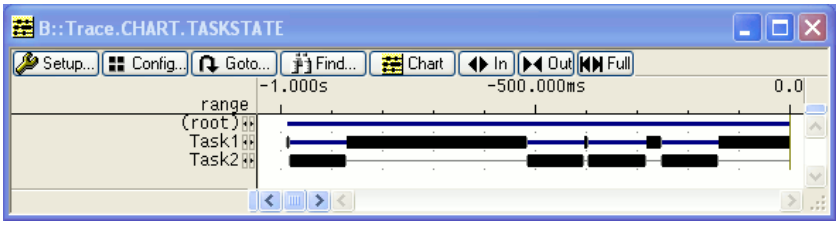

Figure 5.24.: The interpretation of a trace recorded with Lauterbach Trace32 showing the context changes happened in the system.

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. Script File

6.1. 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.

6.2. 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;

6.3. 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`.

6.4. 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.

6.5. 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"
    />
...
```

7. Standalone version of RT-Druid

7.1. 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.

7.2. 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`.

7.3. 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.

8. 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.

A. 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 [
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;

   },
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
    */

] CPU_DATA[];

ENUM [
```

```
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,
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;
   },
```

```
        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[];

      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.