ALOE User Guide

Ismael Gomez, Vuk Marojevic, Antoni Gelonch

Universitat Politècnica de Catalunya

June 2010

Contents

This document contains the ALOE user guide. It features a quick start guide and presents several ALOE tools and describes how to use them. ALOE stands for abstraction layer and operating environement. It is an open-source SDR framework with cognitive computing resource management capabilities. For more information about ALOE, check the related publications, available online http://flexnets.upc.edu/. For hands-on execises, download and try the ALOE Sessions, which are also available on the FlexNets web site.

Index

1 ALOE Quick Start Guide

Section 1.1 describes how to download and install the ALOE framework on a Linux operating system (OS). Section 1.2 explains how to run an example waveform.

ALOE can run on any multiprocessing platform. The current version can be installed on:

- 32-bit x86 processors
- 64-bit x86 processors
- ARM processors (tested on MARVELL processors from SheevaPlug)

The next release will support PPC440 and TI DSPs.

1.1 ALOE Versions

We continuously evolve the ALOE framework and tools. Consult http://flexnets.upc.edu/downloads/source/ for the latest versions. This user guide was written for ALOE version 1.4. *$ALOE/* stands for the directory where you extracted and installed the ALOE source package on your PC. If you follow the insallation instructions of Section 1.2 it symbolizes aloe-<version_number>/, aloe-1.4, for instance.

1.2 Download and Installation

Download the ALOE source package from

http://flexnets.upc.edu/downloads/source/$ALOE.tar.gz

or check the last svn tag release at

http://flexnets.upc.edu/svn/phal/tags/

Before installing ALOE, make sure your system features the following libraries:

- readline (versión 4.0 or higher)
  - ftp://ftp.gnu.org/gnu/readline/
- termcap
  - ftp://ftp.gnu.org/gnu/termcap/

In a Debian based Linux distribution, type the following command in the shell:

```
sudo apt-get install libreadline5-dev
```

In a Fedora distribution, use yum to install the libraries:

```
sudo yum install readline-devel
```

Now you can untar, compile, and install ALOE as follows:

```
tar xzvf $ALOE.tar.gz
cd $ALOE
```

```
./configure
make
sudo make install
```

ALOE is now successfully installed on your system. Otherwise, contact us (http://flexnets.upc.edu/trac/wiki/People) and indicate the problems you experienced while trying to install ALOE.

Before running ALOE, you need to tune some kernel parameters to enable larger messages buffers. You may edit the *sysctl.conf* file to make these changes permanent. Therefore, add the following lines to */etc/sysctl.conf* file:

```
kernel.msgmnb=1048576
kernel.msgmax=10485760
kernel.msgmni=128
```

You need to log in as root for editing the file. Now run sysctl to make the changes effective:

```
sudo /sbin/sysctl -p
```

1.3 Running an Example Waveform

Launch ALOE from the ALOE installation directory (*$ALOE/*, the directory where you extracted and installed the ALOE source package on your PC):

```
runph
```

To run ALOE in real-time you need to launch it with root permissions:

```
sudo runph
```

Note that since ALOE executables are installed in /usr/local/bin, make sure this path is accessible from sudo (visudo command).

This will open the ALOE prompt where you can enter ALOE commands. Should you experience problems while trying to launch ALOE, the reason may be that a previous ALOE session is still running. If this is the case:

```
sudo runph –f
```

The following commands load, initialize, and run the ***example*** waveform:

```
runph$: phload example
runph$: phinit example
runph$: phrun example
```

The ***example*** waveform should be running now. You can obtain information about its execution status using the *execinfo* tool:

```
runph$: execinfo example
```

You can obtain the available waveform variables (statistics) with

```
runph$: statlist example
```

To obtain the value of a variable (statistics) you can use the `staget` and `statset` tools:

```
runph$: statget example source numbits
runph$: statset example source numbits 512
```

2 ALOE User Guide for Linux

This section describes the first steps for launching and running a waveform on the ALOE framework for Linux. Section 4 described how to develop waveforms for ALOE.

We assume that you have already downloaded and installed ALOE on your PC (Section 1).

2.1 Launching ALOE

If you downloaded the ALOE source package, take a look at the *$ALOE/example-repository/* directory. This is path that the Manager Daemons will examine for configuration files and waveform specifications and executables, among others.

The *example-repository/* contains the following mandatory subdirectories:

- *logs/* Module logs in a separate subdirectory for each waveform.
- *reports/* Module statistic reports in a separate subdirectory for each waveform.
- *statsman/* Module initialization parameters in a separate sub directory for each waveform.
- *swman_execs/* Module binaries in a separate subdirectory for each platform.
- *swman_apps/* Application descriptions.

Make sure that the platform directories under *swman_execs/* contain (or link to) your waveform binaries compiled for linux. If you are not crosscompiling (Section **¡Error! No se encuentra el origen de la referencia.**), objects are installed in */usr/local/bin*. Therefore, the local executable directories should point to it:

- swman_execs/linux --> /usr/local/bin
- swman_execs/linux_64 --> /usr/local/bin
- swman_execs/linux_arm --> /usr/local/bin

The ALOE framework for Linux needs to access the waveform repository. The path of these respositories can be selected permanently in a configuration file or as an argument in the Launcher. By default, it is defined in a configuration file and points to *$ALOE/example-repository/*. ALOE needs root privileges in order to set real-time priorities to processes. Start ALOE:

```
sudo runph
```

This will launch a default single-processor configuration.

The `runph` command is configured in the *platform configuration file*. The default configuration can be modified with a set of arguments:

```
runph -r [repository_path] -c [platform_cfg_file] -i [xitf_cfg_file] [--
daemon]|[--no-daemon] [--debug] [-o output_file]
```

The argument soptions are:

- *-r [repository_path]*: Path to the repository directory.

- *-c [platform_cfg_file]*: Platform parameters configuration file, see Section 2.2. *Default*: /usr/local/etc/daemons.conf
- *-i [xitf_cfg_file]*: External interfaces configuration file, see Section 2.2. *Default*: /usr/local/etc/xitf.conf
- *--daemon*: Run ALOE in the background. *Default*: Foreground.
- *--no-daemon*: Run ALOE in normal mode.
- *--debug*: Run in debug mode. Do not kill objects in the case of real-time failures.
- *-o [output_file]*: File to redirect the standard output when running as a background process. *Default*: output.log

2.2 Customizing your Processor

The ALOE Linux Launcher admits two configuration files as arguments for modifying certain platform parameters. In this section we describe how to configure these parameters and what implications they have.

Platform Configuration File: This file configures the processor options and selects which daemons will be launched when executing `runph`. You can add or remove them according to the daemons that you need for your system. By default, any processor should run the following daemons:

- frontend
- swload
- exec
- bridge
- sync
- stats

The following daemons can be optionally included:

- swman
- hwman
- statsman
- sync_master

A single-processor configuration should launch all the daemons except for **sync** and **sync_master** (synchronization is not needed). Section 7 explains how to configure multiprocessor execution environemnts. The *Platform Configuration File* contains the default configuration for typical scenarios.

Some of the parameters in the *Platform Configuration File* can be overwritten by the arguments in the runph command. The following parameters are available in the *Platform Configuration File*:

- *name*: The name of the processor,
- *arch*: Processor architecture. Supported architectures: x86, x86_64, arm
- *macs*: Total processing capacity in multiply-accumulate (MAC) operations per second (MAC/s),
- *nof_cores*: Number of symmetric multiprocessing (SMP) cores,
- *tslot*: Duration of the time slot in microseconds,
- *run_as_daemon*: Run as a background process,
- *output_file*: Redirects the standard output to this file,

- *work_path*: Path to the software repository,
- *printbt_rtfault*: Print backtrace of object's real time failures,
- *printbt_atexit*: Print backtrace when an object exits,
- *itfs_delay*: Object Interfaces default delay. Typical values are 0, 1, or 2,
- *debug_mode*: Disable real time monitoring (Debug mode).

External Interfaces Configuration File: This file has one section for each external interface of your processor. Before creating a network of ALOE processors, you should become familiar with the ALOE execution environment concepts. ALOE permits selecting which daemons will run on which processor, depending on your network architecture. The format of the file is a set of sections (*[xitf]*) with the following mandatory fields:

- *id*: Identification of the interface (hexadecimal 16-bit integer). Not any value is possible as it is used by daemons to discover the purpose of the interface. It must be one of the following:
  - 0x1: Master Control Interface,
  - 0x1n, where n=[0..F]: Slave Control Interface,
  - 0xpq, where p=[2..F],q=[0..F]: Data Interfaces.

- *address*: IP address of the interface. For input interfaces it will be the one to bind to, for output interfaces it will be the remote address.

- *bw*: Bandwidth of the interface (in bits per second, bps). Only for output interfaces.

- *port*: IP port for the connection.

- *type:* Interface protocol. Supported protocols: *tcp, udp*

- *mode*: Direction of the interface. It can be one of the following:
  - in: Input interface (data only),
  - out: Output interface (data only),
  - inout: Input/Output interface, listening socket, typically for slave control interfaces,
  - outin: Input/Output interface, output socket, typically for master control interface.

2.3 Launching ALOE as a Daemon

Running ALOE as a background process is often interesting when long execution times are expected. If not launching ALOE in the background mode, disconnecting a user terminal (or a ssh session) will kill the ALOE processes. To run ALOE in the background, launch it with the *daemon* option (provided as an argument) or configure the platform configuration file:

1. Modify the *platform configuration file*: set "**run_as_daemon**=*yes*" at the CPU section or launch ALOE with the **--daemon** argument.
2. Modify the *platform configuration file*, adding "**output_file**=*hwapi_log_file*" to the CPU section or launch ALOE with the **-o** *hwapi_log_file* argument.
3. Launch ALOE from any directory:

```
sudo runph
```

4. To issue commands you need to connect to the *cmdman daemon* (from a local or remote machine). Launch *cmdman* in the remote mode, specifying the IP address of the processor that runs ALOE:

```
cmdman -c 127.0.0.1
```

   where 127.0.0.1 should be replaced by the IP address of the ALOE host.

   Now you can type commands as usual and exit with Ctrl-C. ALOE will keep running in the background. When you wish to log in again, connect with the previous cmdman command. Also, keep track of the output files (specified in the *platform configuration file*) for errors or unusual behaviour.

5. Finally, you can kill ALOE by sending an interrupt signal to the ALOE parent process:, just type:

```
killph
```

   The script is installed in */usr/local/bin/*.

2.4 Running a Waveform – Basic ALOE Commands

ALOE can be controlled from a shell using text-based commands. The waveform execution is divided into three execution stages (steps): *load*, *initialize* and *run*. We will exemplify the waveform execution procedures for the ***example*** waveform.

Once ALOE has been started we can load, initialize, and run the ***example*** waveform in three consecutinve steps:

```
runph$: phload example
runph$: phinit example
runph$: phrun example
```

Once the waveform is in the run state, we can retrieve the execution statistics of its modules:

```
runph$: execinfo example
```

To get the waveform variables (statistics):

```
runph$: statls example
```

You also can pause the execution of the waveform and run it for a finite number of steps:

```
runph$: phpause example
runph$: phrun example 1000
```

2.5 List of ALOE Commands

Here we list all currently avaialable ALOE commands:

- `help` Displays list of commands.
- `phload` Loads the avefroms. Args: `phload app_name`.
- `phinit` Initializes the waveform. Args: `phinit app_name`.

- `phrun` Starts the resal-time waveform execution. Args: `phrun app_name`.
- `phpause` Pauses the waveform. Args: `phpause app_name`.
- `phstep` Runs the waveform during a single time slot. Args: `phstep app_name`.
- `phstop` Stops the waveform execution. Args: `phstop app_name`.
- `statlist` Lists the available statistics. Args: `statlist app_name [obj_name]`.
- `statset` Modifys a statistics value. Args: `statset app_name obj_name stat_name new_value`.
- `statget` Gets a statistics value. Args: `statget app_name obj_name stat_name`.
- `statreport` Starts/stops reporting statisitcs. Args: `statreport start/stop app_name obj_name stat_name window_len period`.
- `applist` List the loaded waveforms.
- `pelist` Lists the processing elements (PEs).
- `execinfo` Executing information. Args: execinfo app_name [obj_name].
- `exit` Exits the terminal.

2.6 Performance Notes

To get the best performance out of your waveform, the following hints may be useful:

- Compile your modules with the **highest optimization** level and specify your processor architecture. If you have a Pentium or Athlon processor with streaming SIMD extensions (SSE), you can pass CFLAGS to the default configure script:

```
./configure CFLAGS='-O3 -march=native -mfpmath=sse'
make clean && make && sudo make install
```

  To optimize for an ARMv5te, on the other hand:

```
./configure CFLAGS='-O3 -march=armv5te'
make clean && make && sudo make install
```

2.7 Cross compilation

In a multiprocessor execution environment with different processor architectures, the processor running the SWMAN daemon (usually the manager processor, specified in the *Platform Configuration File*) needs access to the implementation of each component for any available processor architecture. There are two options to achieve this: perform cross compilation for all components from the manager processor or copy the binaries from the local */usr/local/bin/* directories (compiled in normal compilation mode).

To be able to perform cross compilation you first need to obtain a tool-chain for the desired target. For example, to compile for the Marvell ARM processor with the gnueabi tool-chain, execute `./configure` with the following arguments (you can alternatively run *scripts/arms_configure* from the *$ALOE/* directory):

```
ac_cv_func_malloc_0_nonnull=yes ./configure --host=arm-none-linux-gnueabi \
CFLAGS='-march=armv5te' --prefix=/usr/local/arm/
```

Pay attention to the prefix argument. Here you should specify a different directory for each different PE target architecture. The binaries of the waveform components will be installed there. There should be a link to these directories from the *swman_execs/* directory. For the above example make sure that

- swman_execs/linux_arm --> /usr/local/arm/bin

Then you can simply recompile your source files and install them:

```
make clean
make
sudo make install
```

Currently only one target PE architecture is supported for eachplatform. For example, you can either have ARM5 processors and cross compile the modules for this target or have ARM9 processors and compile for them. However, you currently cannot have a mix of ARM5 and ARM9 processors. Future ALOE versions will have a customizable set of executables directories allowing for each architecture.

3 ALOE Module Development

ALOE generates waveforms by concatenating a set of digital signal processing blocks. We refer to these blocks as modules, components, or objects. A proper concatenation of modules generates the desired transmitter or receiver digital signal processing chain. A component must be written in ANSI C while following a set of rules to be able to interact with the other components and with the ALOE framework.

A digital signal processing block or **"ALOE Module"** has the following properties:

- A set of *input* interfaces,
- A set of *output* interfaces,and
- A set of *initialization parameters.*

An ALOE Module

- reads the initialization parameters once for configuration purposes and
- on each invocation generates data for output data streams as a function of the input data if any.

3.1 Defining a Module

The table indicates how to organize the module files.

Table I – Module files.

Directory	Files	Comment
mymod/platform_make/	...	One directory for each supported target processor. For example: *lnx_make* may contain the Makefile for Linux and *c6000_make* the Code Composer Studio .pjt files.
mymod/interfaces/	Inputs.h outputs.h stats.h	Definition of the input and output interfaces and variables (initialization parameters and statistics)
mymod/src/	module_imp.c mymod.c	mymod.c: Standalone function implementing the algorithm. module_imp.c define the skeleton frontend, a tool that simplifies the module design processs. This module calls the signal processing function implemented in mymod.c
mymod/bin/	mymod	Binaries after compilation and linking

3.2 Implementing a Module

Once the functionality of your module has been defined, you can start implementing it. Here we assume the reader is familiar with the basic ALOE concepts. This section describes how to implement an ALOE Module and provides some examples.

3.2.1 Process Structure

The first order a loaded module receives is the initialization (INIT) order.The module will then read a set of initialization parameters, activate the communications interfaces, and define a set of statistics variables (to be monitored during the real-tim data processing). This procedure is executed just once in the execution lifecycle of the module.

Once the execution order is received (RUN), the module will switch to the RUN state, receive messages, dispatch the required tasks (for example process a block of data received from an interface and send the result to another interface), and return to the initial point.

When the module receives the STOP order, it notifies ALOE to close all the allocated resources and terminate its execution (unload the module).

Note how the CPU is allocated to the module during the time it process the message and dispatches the corresponding tasks. When finished, the module returns to the idle state, returning the CPU to ALOE. In case where a modules violates the timing rule (for example, exceeding the allocated time slot for execution), ALOE may force the CPU control and set the module back to idle.

3.2.2 Programming a Module

The programming language used to describe a module may from device to device and depends on the available tools that are provided by the device manufacturer. If we could always use the same language for designing a module, the code could be directly reused. This, however, is unrealistic, because some programmable devices can use C/C++ (GPPs, DSPs) while others cannot (FPGAs, for instance).

We have chosen the following standard programming language for the ALOE framework:

- C Language for GPPs and DSPs and
- VHDL for FPGAs.

The use of a standard programming language is, though, not the only requirement for enabling the whole set of ALOE functionalities. The use of the ALOE Software Library is another requirement. At this point we recommend checking the ALOE SW-API Documentation to get an insight on the available functions and their behaviour.

A module designer does not design modules for a particular device of device set. This means that important concepts, such as sampling frequency, time slot duration, and memory distribution or architecture are unknown during the module design process. Thus, it the module needs to support a wide sampling frequency range and different architectures. A special function provided by the ALOE Software Library serves for this purpose. The number of samples to be generated every time the module is executed (each time slot) can be obtained with the following function:

```
int GetTempo(float freq);
```

This function receives as a parameter the frequency at which the module should operate. Since only ALOE knows the execution interval associated to each module, a simply operation will calculate the number of samples to be processed per module and time slot. The amount of samples that the module needs at its inputs will depend on the algorithm and, in particular, on the ratio between the input and output rate frequency.The module will then wait until all the data it needs is available. Note that this could lead to timing violations if the frequency relations between neighboring objects are not carefully defined.

A skeleton has been designed to simplify the module's implementation procedure. It provides some common functionalities so that the user will only have to define the interfaces following a predefined structure and implement the custom digital signal processing algorithm. The skeleton takes care of the initialization of interfaces and the buffer management.

This skeleton is provided in the *libskeleteon.a* library (located at sw_api/lib directory). The skeleton needs three header files: *inputs.h*, *outputs.h* and *stats.h*. Interfaces and statistics are defined in structures (defined in *swapi_utils.h*) whose names need to be mantained (*input_itfs*, *output_itfs*, *params* and *stats*). The below figure introduces these structures.

Figure 2 – Structure definitions (*swapi_utils.h*).

```
/** Structure for logic interfaces
 *
 * Regardless the direction of the interface, the user must provide
 * a buffer for storing data of size max_buffer_len elements of size sample_sz
(in bytes).
 *
 * For input interfaces. Set the variable name to input_itf:
 *  process_fnc() will be called as soon as the required amount of
 *    samples is available. This number is provided by the function
get_block_sz()
 *  which will be called at the beginning of every timeslot (if needed).
 *   Setting get_block_sz() to NULL is equivalent to returning 0 from this
function.
 *  As soon as any data is available the processing function will be called.
 *
 * For output interfaces. Set the variable name to output_itf:
 *    If data needs to be generated, point get_block_sz() to a function
returning
 *   the amount of samples to be generated (you can use GetTempo to convert
from
 *  frequency to number of samples). Then, process_fnc() will be called to
 *  generate such data.
 *   If data is being provided synchronously to any input interface, set
get_block_sz
 *  to NULL and use SendItf() function to send the data.
 *
 * Example:
 *
 * Configure an input interface which can process any amount of samples (not
block oriented)
 *
 *  typedef int input1_t;
 *
 *  struct utils__itf input_itfs[] = {
 *                   {"myInputInterface",
 *                   sizeof(input1_t),
 *                   1024,
 *                   input_buffer,
 *                   NULL,
 *                   process_input},
 *
 *                   {NULL, 0, 0, 0, 0, 0}};
 *
 *
 */
struct utils_itf {
        char *name;                     /**<< Name of the interface */
        int sample_sz;                  /**<< Size of the sample, in bytes */
        int max_buffer_len;             /**<< Max buffer length in samples */
        void *buffer;                   /**<< Buffer where to store data */
        int (*get_block_sz) ();            /**<< Returns number of samples to
read/generate. Returning <0 interrupts execution*/
        int (*process_fnc) (int); /**<<  Function  to  process/generate  data.
Returning 0 interrupts execution */
};


struct utils_param {
        char *name;             /**< Name of the parameter */
        char type;              /**< Type of the parameter*/
```

```
        int size;               /**< Size of the parameter value */
        void *value;            /**< Pointer to the value to obtain.
                                    Make sure the buffer is big enough
(len) */
};


/** Structure for statistics
 *
 * You can automatically initialize a variable if you set the configuration
 * in this structure.
 * The id of the initialized variable is stored in the id pointer and you can
 * also initialize the variable with a value if the pointer value is set to
any
 * non-NULL value.
 *
 * Set update_mode to READ to automatically read the contents of the variable
 * at the beggining of every timeslot (with the contents at value with fixed
length size).
 *
 * Set update_mode to WRITE to automatically set the contents of the variable
 * at the end of every timeslot (with the contents at value with fixed length
size).
 *
 * Set update_mode to OFF to disable automatic updates.
 */
struct utils_stat {
        char *name;                     /**< Name of the variable */
        char type;                      /**< Type of the variable */
        int size;                       /**< Size of the variable, in elements
(of type) */
        int *id;                          /**< Pointer to store the id for the
stat */
        void *value;                    /**< Initial value for stat */
        enum stat_update update_mode;    /**< Mode for automatically updating
*/
};
```

The other skeleton file, *module_imp.c*, is the bridge between the skeleton and the digital signal processing functions. It implements the functions that are called when a new packet is received or needs to be generated. Although the signal processing algorithms could be implemented here, good coding practices recommend doing it in another file.

Figure 3 – Object implementation (*module_imp.c*).

```
/** ALOE headers
 */
#include <phal_sw_api.h>
#include <swapi_utils.h>

#include <phal_hw_api.h>
#include <sys/resource.h>

#include "inputs.h"
#include "outputs.h"
#include "stats.h"

char str[128];

int get_output_length()
{
    /* use gettempo to get number of samples */
    return GetTempo(freq);
}

/** Run function.
* @return 1 if ok, 0 if error
```

```
*/
int generate_output(int len)
{
        int i;

        if (!len)
            return 1;

        sprintf(str, "Generating sampfreq %.1f sin freq %.1f\n", freq,fsin);
        WriteLog(1,str);

        /* generate sinusoid */
        generate_sinusoid(output_data,fsin,fsamp);

        SetStatsValue(stat_signal, output_data, len);

        return 1;
}

int InitCustom()
{
    return 1;
}

int RunCustom()
{
    return 1;
}
```

The last two functions, `InitCustom()` and `RunCustom()`,need also be defined. The `InitCustom` function is used to implement custom initialization routines that are not related with interfaces or statistics. Computing filter coefficients, for example, would be done by this function. The skeleton will provide the necessary parameter(s) for the computation and call the `InitCustom` function.

Analogously, the `RunCustom` function is useful for performing some background processing tasks that are not directly related with the data processing. These tasks are asynchronous with the data flow.

Finally, the actual algorithm will be implemented in a standalone file, for example, *myobj.c* (Figure 4).

Figure 4 – Signal processing algorithm (*myobj.c*).

```
/** ALOE headers
 */
#include <phal_sw_api.h>
#include <math.h>

#define PI 3.1415

void generate_sinusoid(int *output_data, float fsin, float fsamp)
{
        int i;
        for (i=0;i<len;i++) {
                output_data[i]    =    (int)    ((float)    1000.0*cosf((double)
2.0*PI*fsin*i/freq));
        }
}
```

3.3 Compiling and Linking

Compiling an ALOE module does not require any other considerations than linking with the ALOE SW and HW Libraries. Obviously, we need to pick the HW Library of our system. After installing ALOE on Linux, the libraries *libsw_api.a* and *libhw_api.a* should be at */usr/local/lib/*.

If you prefer using *autoconf* or *automake* tools you may find Makefile.am useful. This file is included in the ALOE source package (under *lnx_make/* of a module's directory path).

4 ALOE Waveform Development

We model SDR applications, or waveforms, as directed acyclic graphs, where nodes represent data processing and arcs data flows. Data is propagated through the processing chain. Each processing block performs some data processing. We use *module*, *component*, and *object* interchangeably for referring to a digital signal processing block.

This Secion explains how to design, debug, and test SDR applications for ALOE.

4.1 Creating Waveforms

ALOE waveforms are created by correctly linking the interfaces of the involved objects. This implies editing the *application description file*, which defines how interface connections. Future ALOE releases will provide a graphical user interface (GUI) for performing this task.

The application description file is a simple text, which indicate the modules (and their executables) that are to be loaded, the interfaces and how they should be interconnected. Listing 1 shows an example application description file. In this example, a single executable (*test_exec*) is launched as two instances, where each is assigned a different object name and different interfaces.

Figure 5 – Waveform definition.

```
object {                    # begins a description of a new obje t
        obj_name=test_w     # object name used in ALOE (must be unique)
        exe_name=test_exec   # executable name in the system (as SWMAN will
access)
        proc=100            # processing demand (in MIPS)

        outputs {                                   # begins output interfaces
definitions
                name=test_w_itf                 # name of the output interface
(as the object will use in its code)
                remote_itf=test_r_itf         # name of the remote interface
                remote_obj=test_r             # name of the remote objec
        }
        outputs {                                  # another output could just be
added...
                name=output_2
                remote_itf=...
                remote_obj=...
        }


}

object {
        obj_name=test_r         # Note how we instantiate another object with a
different name, although the executable is the same
        exe_name=test_exec
        proc=100

        inputs {
                name=test_r_itf
                remote_itf=test_w_itf
                remote_obj=test_w
        }
}
```

Once you have created this file, you need to save it as *your_waveform_name.app* under the *swman_apps/* directory. Make sure that the executables are accessible under the swman_execs/platform/ path.

4.2 Debugging Waveforms

Once ALOE is launched and a waveform loaded, you can attach a debugger (gdb, ddd, eclipse, etc.) to the running process. The following steps explain how to debug a waveform component:

1. Launch ALOE with the debug argument. This will prevent the components from being killed when stopped by the debugger:

```
sudo runph --debug
```

2. On the prompt, load your waveform as usual:

```
runph$: phload my_app
```

3. Open your debugger and attach it to the waveform component execution process.

4. Go back to the runph prompt and:

```
runph$: phinit my_app
```

5. If you placed a breakpoint at the initialization part, you will see how the program stops. You can continue the execution (step by step or free running) until the component arrives to the Status() call. At this point, you can start debugging the waveform's RUN phase. Place a breakpoint at any line of the RUN part of the code and use the `phstep` tool to execute it for one time slot.

```
runph$: phstep my_app
```

   The program will stop at the breakpoint.

6. Once the module gets back to the Status() point, type `phstep` again for running it for another time slot. You can also remove breakpoints and run it for a finite number of time slots, for example 100:

```
runph$: phrun my_app 100
```

5 Creating Modules from Simulink

The ALOE modules code can be automatically generated from MATLAB/Simulink. This high-level simulation tool allows testing the functionality of your module. It also permits synthesizing C code that is compatible with ALOE. We created a Simulink Workshop Target for automatically generating the interface to the ALOE SW-API.

This Section explains by means of an example how to create ALOE modules from MATLAB/Simulink:

5.1 System Requirements

1) MATLAB 2008/2009 and Simulink
2) PC running Linux (kernel 2.6.21 or above)
3) ALOE version 1.4 or above

5.2 Installation

You need to configure MATLAB and ALOE to be able to generate ALOE modules from MATLAB.

5.2.1 MATLAB

The ALOE Simulink Real-Time Workshop Target files are available in the *matlab* directory in the ALOE package, which features:

- lnx_callback.m
- lnx_install_dir.m
- aloe_main.c
- lnx_unix.tmf
- make_lnx.m
- aloe.tlc
- rtwmakecfg.m

First copy these files to the Matlab path *%MATLAB%/rtw/c/aloe*. Then start Matlab and add the new directory to the Matlab path (File->Set path...).

5.2.2 ALOE

To compile the Matlab models for Linux, we have to copy the Matlab libraries and headers to a Linux path. This is as easy as copying the directory structure from the Matlab directory to the Linux file system.

We recommend creating a new Linux directory, *matlab-files/*, for example, and copying the following directories from the Matlab installation path:

```
%MATLAB%/extern
%MATLAB%/rtw
%MATLAB%/simulink
%MATLAB%/toolbox
```

Note that the ALOE target directory rtw/c/ also needs to be copied.

The amount of files to copy may be large. Although it would suffice to copy only those toolboxes that will be used, we recommend copying all of them.

5.3 C Code Generation for ALOE

Before generating the C code for ALOE, we should specify ALOE which Simulink signals will be used as inputs and which as outputs. A Simulink signal will not be accessible from the generated C code unless we select the signal option "Test point".

The second issue is indicating ALOE the signal type (input/output). We, therefore, add the prefix "in_" or "out_" before the signal name. The name that ALOE will use to access this interface (as specified by the *application description file*) is

- *in_itfName_re*/ *out_itfName_re* for the real part and
- *in_itfName_im*/ *out_itfName_im* for the imaginary part.

Finally, we have to select the interface we want to use to access this signal, which will be global export.

All these options can be selected under the Signal Properties menu (right-click on the signal and select **Signal Properties**). Then, select the **Test point** option and the name of the signal, as indicated below.

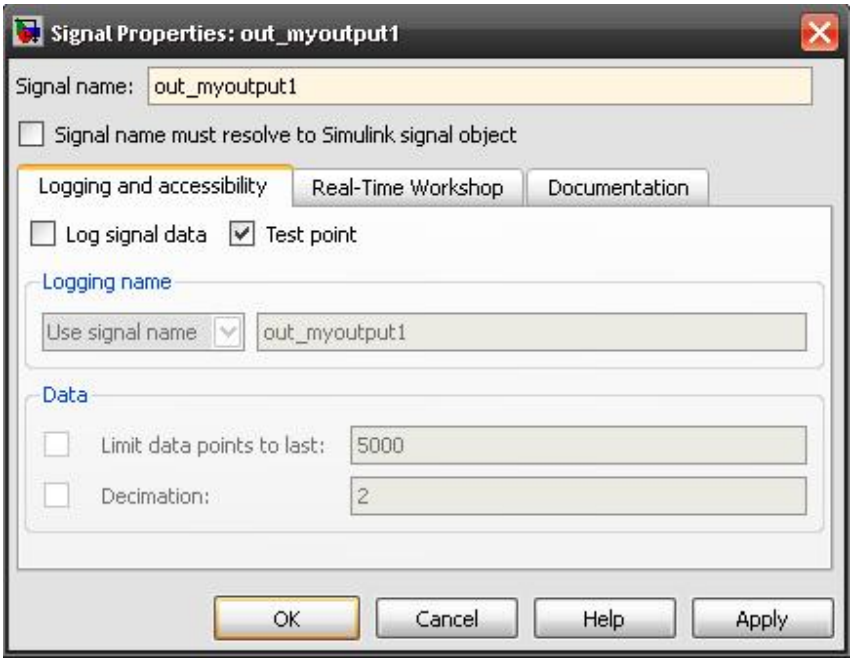

Figure 6 – Signal configuration.

Now select the storage class "ExportedGlobal" under the Real-Time Workshop tab (Figure 7).

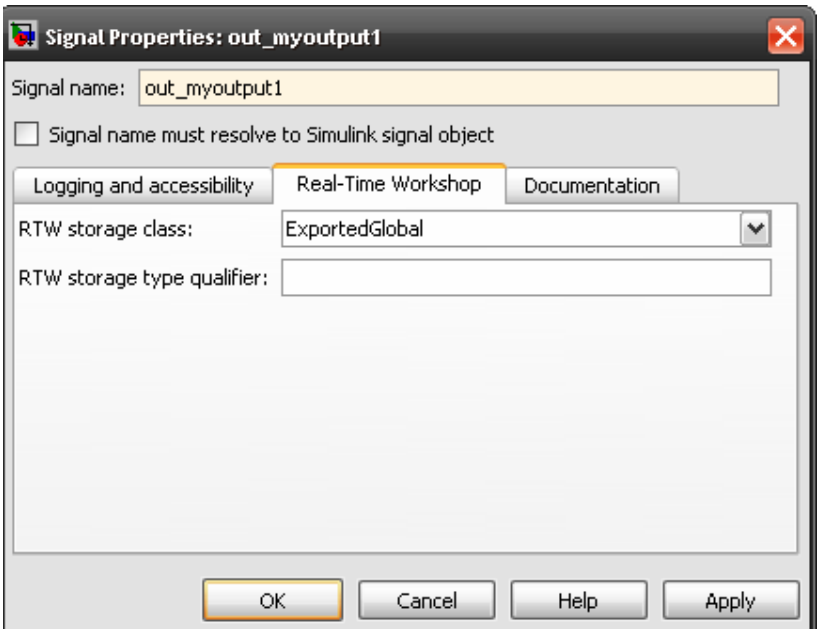

Figure 7 – Signal storage class.

5.3.1 Code Generation

Next we configure the Simulink model to generate an implementation for the ALOE target. In the model window, choose **Tools->Real Time Workshop->Options...** form the main toolbar (Figure 8).

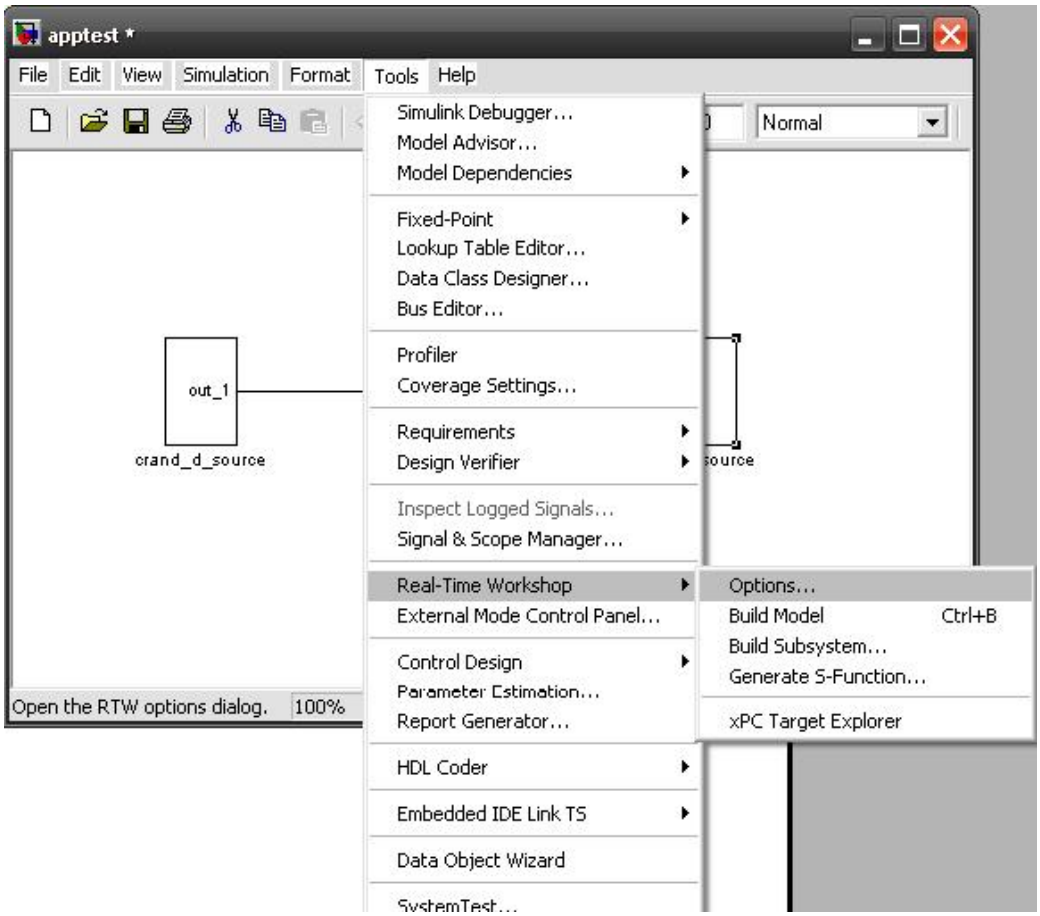

Figure 8 – Real-Time Workshop options.

As the “System target file” select “aloe.tlc” from the browse options (Figure 9).

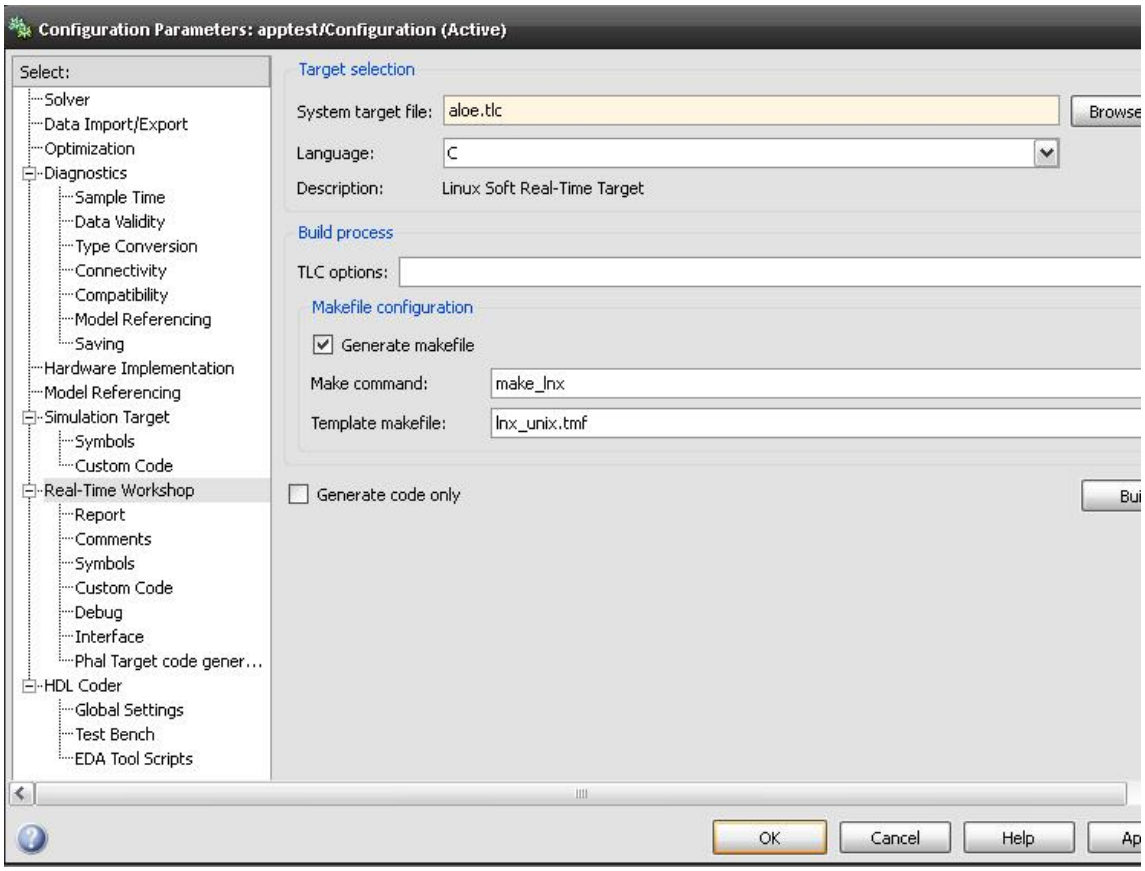

Figure 9 – System target file selection.

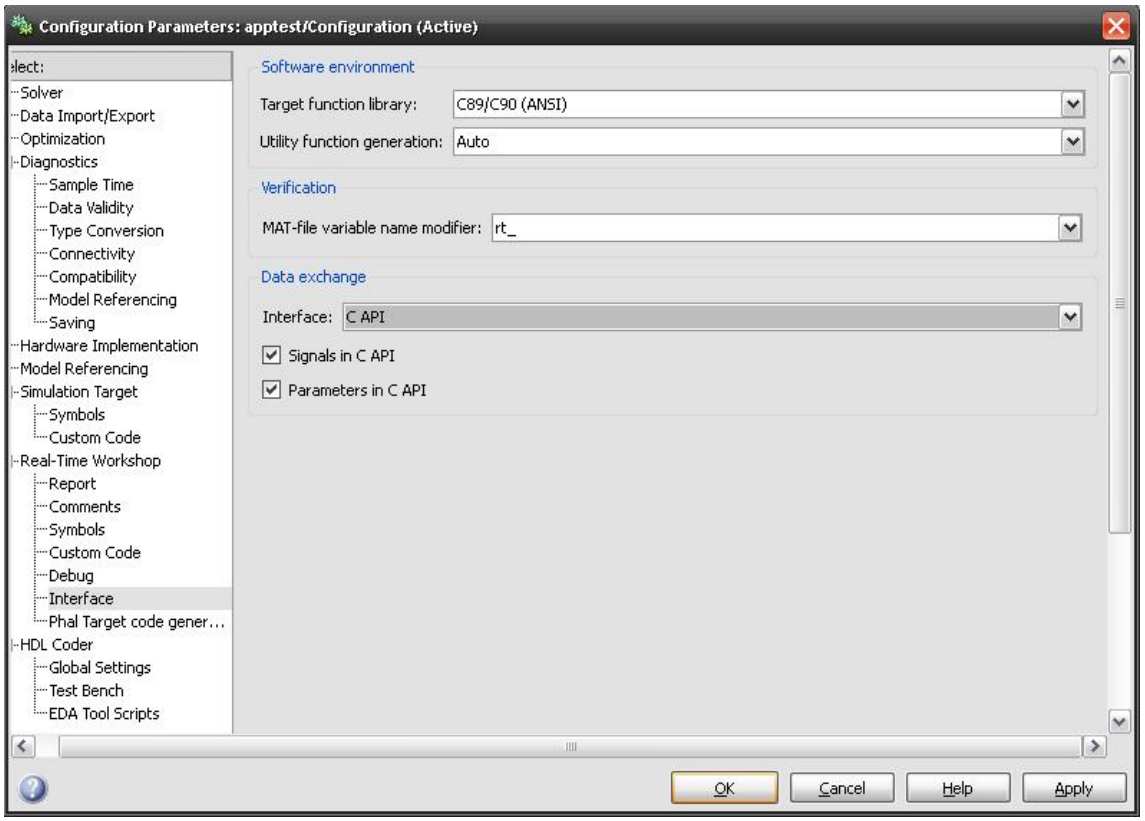

Figure 10 – System target signal interface configuration.

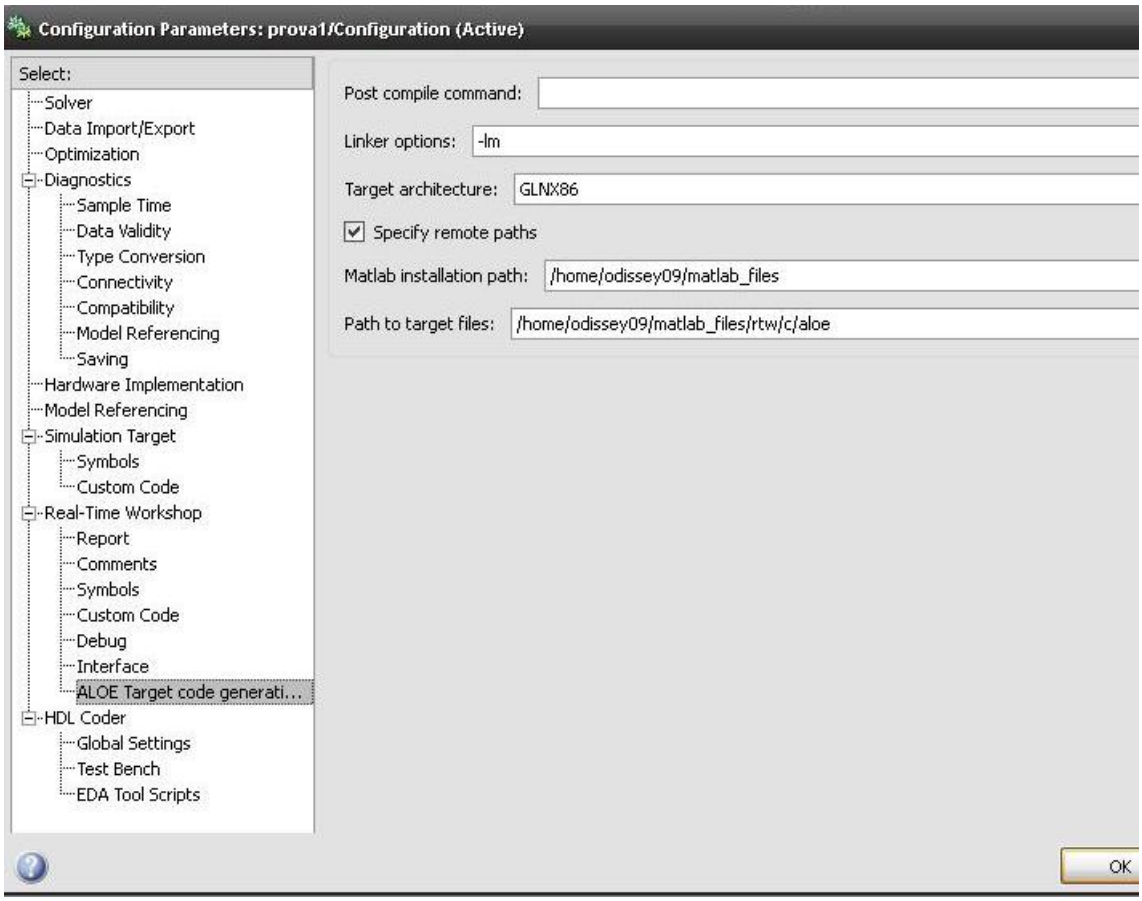

Figure 11 – Matlab paths configuration.

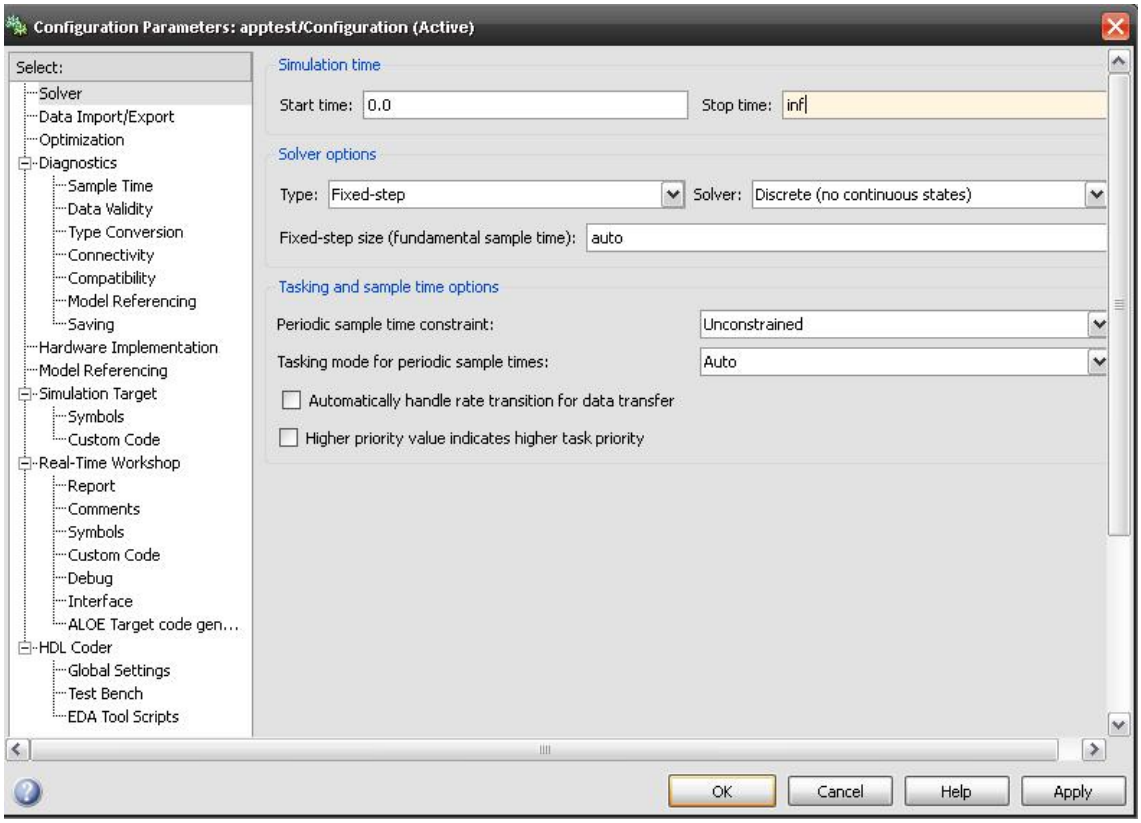

Figure 12 – Solver configuration.

Go to the Interface tag in the menu on the left of the window and select the C API Interface (Figure 10).

Make sure that target files are correctly specified for the Linux environment. Therefore, configure the ALOE Target code generation options following the model of Figure 11.

We finally need to select the discretization model for converting continuous signals to a discrete representation. In the same screen and under the Solver menu select "fixed-step" **Type** and "Discrete (no continuous states)" **Solver** (Figure 12). Set also the simulation time to infinite: "inf" as the **Stop time** option. Make sure that all other parameters are set according to Figure 12.

We can now generate the C code that specifies the signal processing functionality of the waveform component. Click on Tools->Real Time Workshop->Build Model and the component source code will be generated. (See the Matlab's main output screen in the case of errors.)

5.3.2 Compilation for Linux

Before compilation, we first need to copy the output files (generated by the Simulink model generator) to a Linux directory, *$ALOE/modules/my_matlab_component/*, for instance. These files are available in the *modelname_aloe_rtw* folder on your Matlab model path.

From the Linux directory that now contains the Simulink output files we can compile the component:

```
make -f modelname.mk
```

This will create the executable file *modelname*.

5.3.3 Execution

We should copy the executables of all models of our waveform to the ALOE SWMAN executable repository (usually located at *%example-repository%/swman_execs/linux/*) to be accessible by the ALOE framework. Then we can create the *application description file*, specifying the module interconnections. Pay attention to the interface name convention for input and output interfaces and real and imaginary parts. Figure 13 shows an example *application description file*.

Figure 13 – Example application description file.

```
object {
        obj_name=crand_d_source
        exe_name=crand_d_source
        proc=1000
        outputs {
                name=out_1_re
                remote_itf=in_1_re
                remote_obj=prova1
                bw=1000
        }
        outputs {
                name=out_1_im
                remote_itf=in_1_im
                remote_obj=prova1
                bw=1000
        }
}
```

6 ALOE User Interface (aloeUI)

This Section describes how to setup and use the graphical user interface (GUI) for ALOE (aloeUI). The GUI is written in JAVA.

6.1 Starting the aloeUI

The aloeUI is distributed with the ALOE download package in the *jar* format (see QuickStartGuide).

To start the GUI, open a Linux shell, switch to the *$ALOE/* directory and type

```
./aloeui.run
```

The aloeUI initial sreen (Figure 14) will show up.

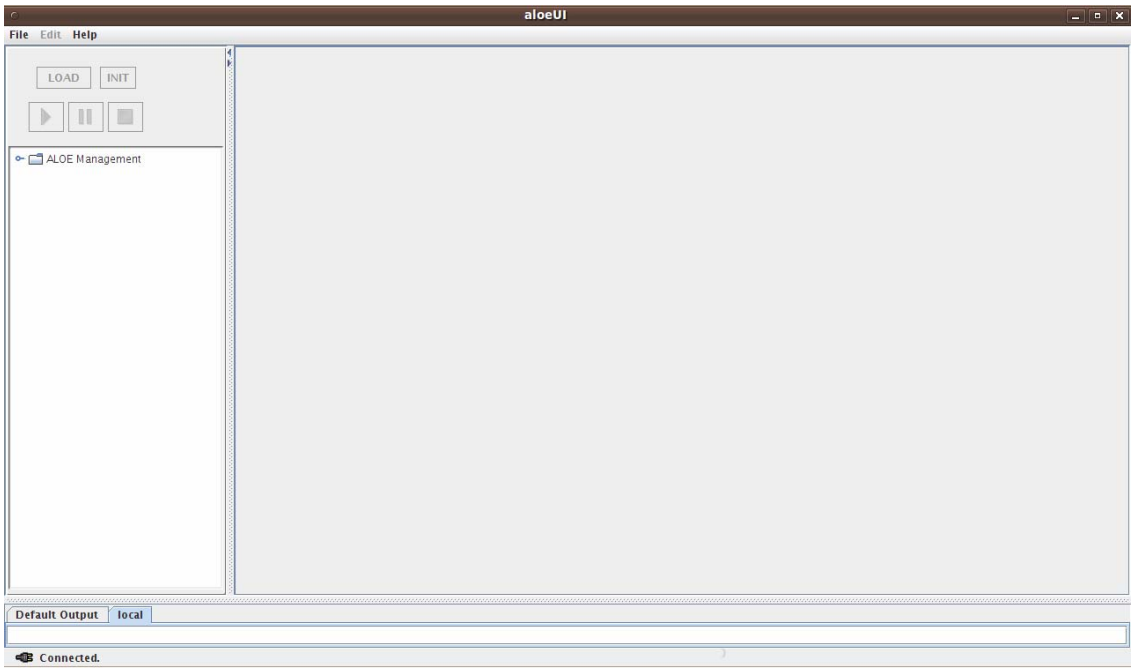

Figure 14 – aloeUI initial screen.

6.2 Setup

Click on "ALOE Management" in the left subwindow to expand a menu containing three items:

- Waveforms: Available waveforms in the active repository,
- Platforms: Available computing platforms,
- Panels: Statistics panels for viewing and modifying statistics.

The following steps configure the local platform:

1. Right-click on Platforms and select **New Platform**.
2. Enter a platform name: "localPlatform", for example.

3. Right-click on the new platform and select **Edit**. A tab will appear in the right window, where you can select how to connect to the platform. For a default local connection, configure the options shown in Figure 15.

Click on the **Browse** button next to the "Output log path" and select the path of the *output.log* file that will be generated by ALOE.

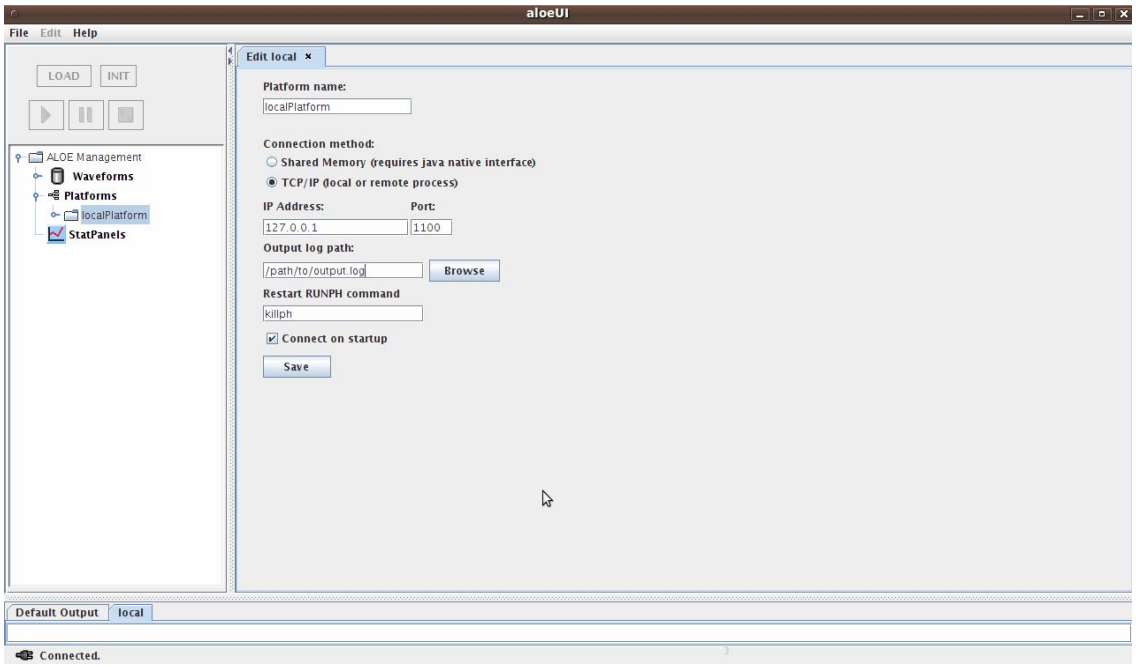

Figure 15 – localPlatform default configuration.

4) Click on **Save** and close the tab.

Open another Linux shell, switch to the ALOE installation directory and type

```
sudo runph
```

to launch ALOE.

Back to the aloeUI, right-click on the new platform and select **Connect**. "Connected" should appear on the status bar at the bottom.

Follow the steps below to configure the waveform repository:

1. Right-click on "Waveforms" and select **Edit**.
2. On the panel appearing in the main aloeUI window, browse to the waveform repository path *example-repository/* right under the ALOE installation directory.
3. Click on **Save** and close the tab. You should see how new applications appear under the Waveforms menu.

6.3 Running a Waveform

Now you can select a waveform and load, initialize, and run it. Therefore, click on the desired waveform under the Waveform menu and use the buttons on the upper-left part of the aloeUI to change the waveform execution status. When you select a waveform, its components are shown

below. You can see the log files or view/report statistics. To view a log file, for example, double-click on the Log icon.

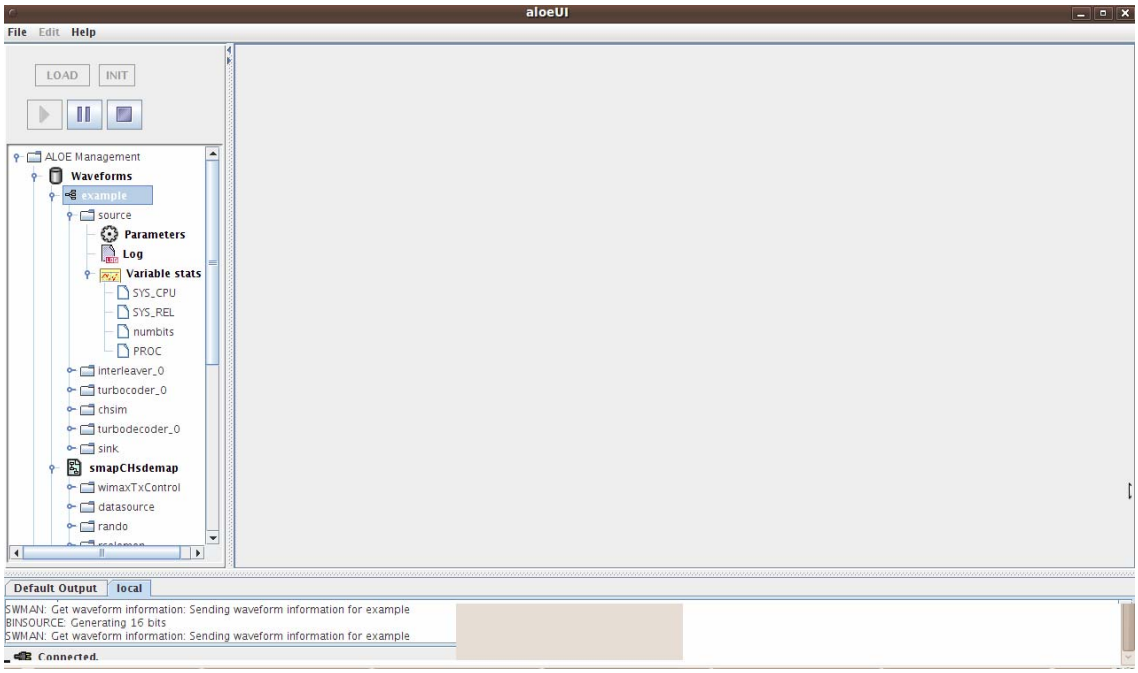

Figure 16 – Example waveform visualization options.

6.4 Viewing and Modifying Statistics

The evolution of statistics values can be captured and graphically visualized during runtime. To start capturing the values of a statistics right-click on the variable under the "Variable stats" menu and select **Start Reporting**. A panel will appear in the aloeUI main window. It may look like that of Figure 17.

You can modify two report options:

- *Report period* – the update interval in multiple time slots.
- *Report window* – the number of time slots capturing data samples. All data samples will be continuously reported if window is equal to period; otherwise (window < period) only a subset of samples will be captured.

You can also display multiple variables in the same graph: Just add another variable to the graph by dragging it to the graph display area.

You can also change the display option to see the histogram of the captured data during one period. Therefore, right-click on the grey part of the panel and select **Histogram**. The panel will switch and may look like that of Figure 18.

The histogram representation permits modifying the number of bins as well as the data intervals. In addition, you can choose between data accumulation and data update: The former allows refining the histogram, whereas the latter replaces the entire data set (with the new data report, updated each period).

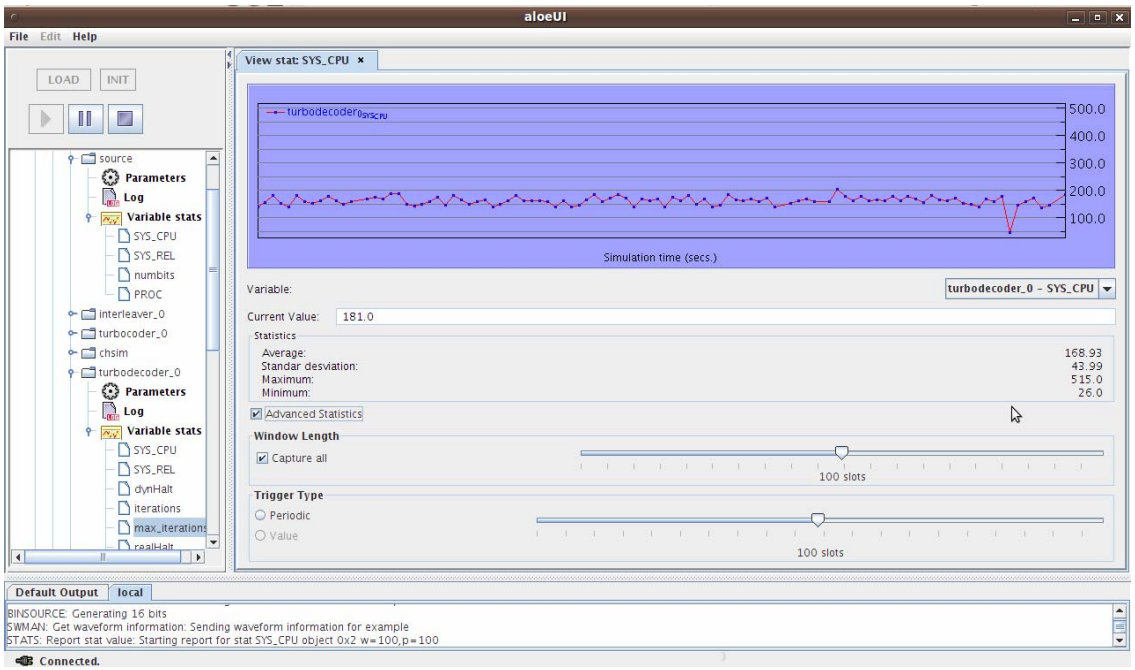

Figure 17 – Viewing statistics, display option 1.

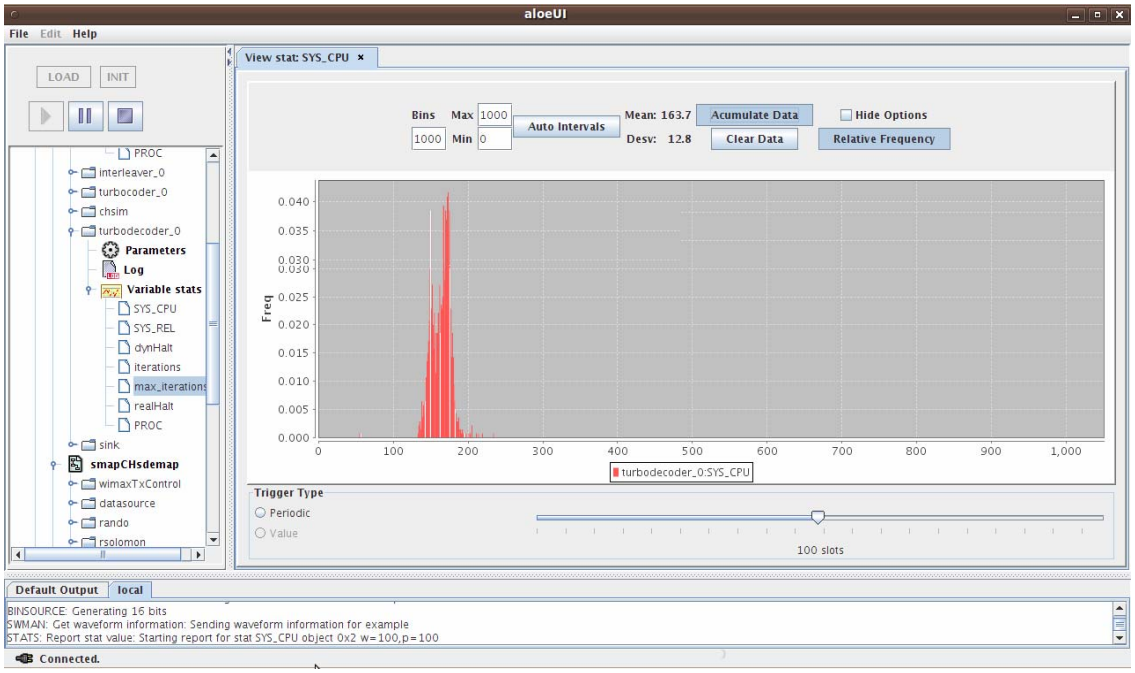

Figure 18 – Viewing statistics, display option 2: Histogram.

There are two predefined statistics variables for every ALOE module. These variables inform about the CPU usage:

- SYS_CPU: Consumed CPU time per module invocation (time slot).
- SYS_REL: CPU relinquish – the time stamp when the module releases the CPU each time slot.

Modules using the *module.c* or *module_imp.c* skeleton will, additionally, feature another statistics variable:

- PROC: Consumed CPU time per module invocation. This variable measures the module's data processing time, disregarding the time taken for receiving and sending data or similar operation that are not directly related with the actual digital signal processing. This information is useful for computing the time overhead of the ALOE services.

6.5 Viewing Execution Statistics

Execution information can bee seen in two formats:

- Numerically (Table)
- Graphically (Diagram)

To obtain the information in numerical format, right-click on a loaded waveform and select **RT Info**. This will open a panel showing a table with the waveform's real-time information (Figure 19).

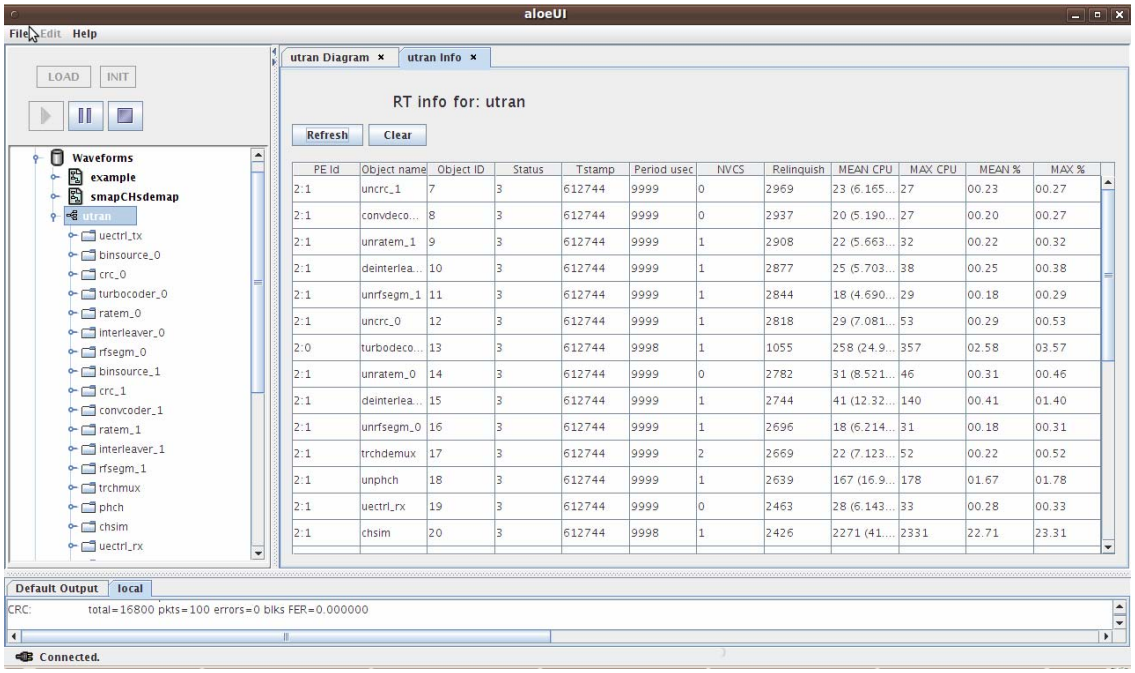

Figure 19 – Waveform real-time information.

For a graphical representation, select **RT Diagram** instead of **RT Info**. This will open a Gannt diagram of the modules' execution schedule on time slot basis (Figure 20). Note that since the execution order is constant, the graph does not change significantly from time slot to time slot (except for the execution time variations).

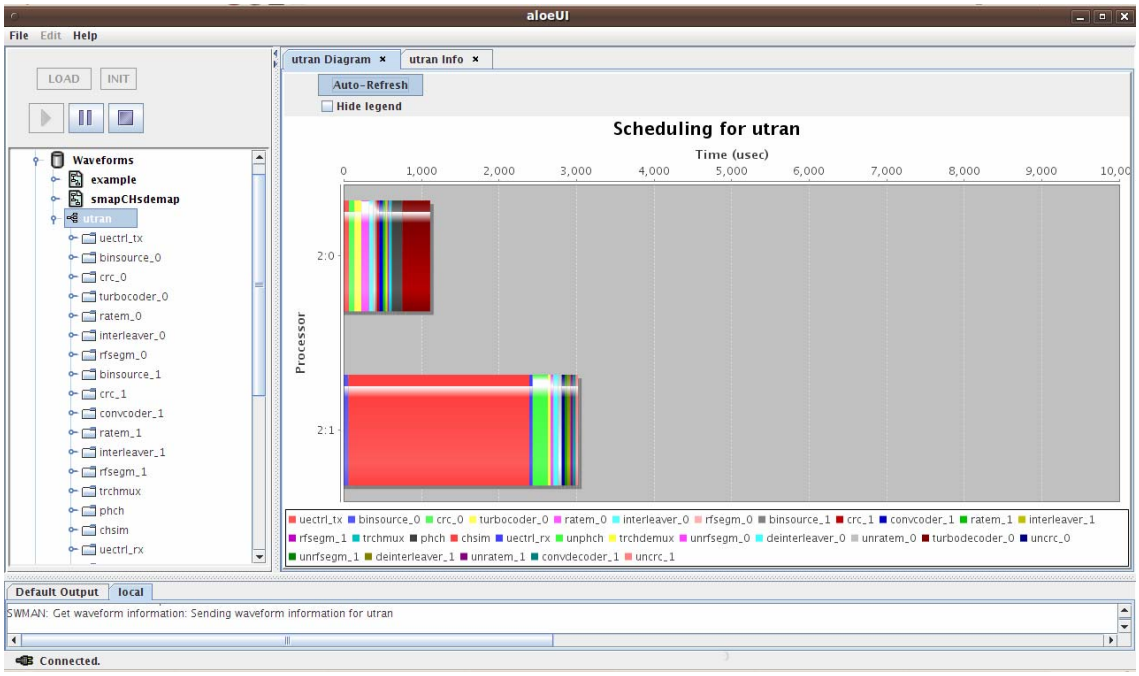

Figure 20 – Scheduling diagram.

7 Multiprocessor Platforms

This section describes how to deploy ALOE in a Linux multiprocessor execution environment.

ALOE currently supports TCP/IP interfaces, which lack of any QoS mechanism and may cause real-time violations when moving data between processors. To minimize this effect, make sure that the network load is low.

1) Decide your network topology: number of processors and connectivity.

2) Configure the *External Interfaces Configuration File* for each processor. Besides the data interfaces, you also need to configure the control and synchronization interfaces.

3) Configure the *Platform Configuration File* and select the Daemons that will be launched on every processor. A default configuration maps all Manager Daemons to a single processor, which contains the executables, whereas the others feature only the mandatory daemons:

- Manager Daemons: hwman, swman, statsman, sync_master, swload, frontend, stats, bridge, exec,
- Other Daemons: swload, frontend, stats, bridge, exec, sync.

4) Launch ALOE on each processor.

5) The hwman daemon at the Manager Processor will automatically detect the network of processors.

6) Load and run your application as usual from the Manager Processor cmdman shell (or from the GUI).

7) You can use the execinfo tool to obtain the mapping and the modules' execution statistics.

You should observe how the modules are distributed among the set of processors. The automatic mapping procedure processes the modules' resource requirements, specified in the waveform's *Application Description File*, and the platform's computing capacities due to the platform's *Platform Configuration File*.