Installing Xenomai 2.6.1

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

1 Introduction **1**

2 Installation steps **1**

2.1 Preparing the target kernel . . . 1
2.2 Configuring and building the target kernel . . . 2
2.3 Building the user-space support . . . 2
2.3.1 Feature conflict resolution . . . 2
2.3.2 Generic configure options . . . 2
2.3.3 Arch-specific configure options . . . 2
2.3.4 Cross-compilation . . . 3

3 Typical installation procedures **3**

3.1 Building for x86_32/64bit . . . 4
3.2 Building for the PowerPC architecture . . . 5
3.3 Building for the Blackfin . . . 6
3.4 Building for ARM . . . 6
3.5 Building for NIOS II . . . 7
3.5.1 Minimum hardware requirements . . . 8
3.5.2 Xenomai compilation for NIOS II . . . 8

4 Testing the installation **9**

4.1 Testing the kernel . . . 9
4.2 Testing the user-space support . . . 9

The latest version of this document is available at this address.

For questions, corrections and improvements, write to the mailing list.

1 Introduction

Starting with version 2.1, Xenomai follows a split source model, decoupling the kernel space support from the user-space libraries used in accessing the former.

To this end, kernel and user-space Xenomai components are respectively available under the `ksrc/` and `src/` sub-trees.

The `ksrc/` sub-tree providing the kernel space support is seen as a built-in extension of the Linux kernel, and no more as a collection of separate out-of-tree modules. A direct benefit of such approach is the ability to build the Xenomai real-time subsystem statically into the target kernel, or as loadable modules as with earlier versions. therefore, the usual Linux kernel configuration process will be normally used to define the various settings for the Xenomai kernel components. Sections "Preparing the target kernel" and "Configuring and building the kernel" document the installation process of this kernel space support.

The `src/` sub-tree contains the various user-space libraries and commands provided by the Xenomai framework. This tree can be built separately from the kernel support, even if the latter is absent from the build system. Section "Building the user-space support" documents the installation process of this user-space support.

If you are using a Debian based distribution, it is also possible to install, and even build Xenomai as a set of Debian packages. For further details, see this page.

2 Installation steps

2.1 Preparing the target kernel

Xenomai provides a real-time sub-system seamlessly integrated to Linux, therefore the first step is to build it as part of the target kernel. To this end, `scripts/prepare-kernel.sh` is a shell script which sets up the target kernel properly. The syntax is as follows:

```
$ scripts/prepare-kernel.sh --linux=<linux-srctree>
[--adeos=<adeos-patch>] [--arch=<target-arch>]
```

`--linux`
: specifies the path of the target kernel source tree. Such kernel tree being configured or not makes no difference and is valid either way.

`--adeos`
: specifies the path of the Adeos patch to apply against the kernel tree. Suitable patches are available with Xenomai under `ksrc/arch/<target-arch>/patches`. This parameter can be omitted if Adeos has already been patched in or the script shall suggest an appropriate one. In any case, the script will not try to apply it again whenever a former patch is detected.

`--arch`
: tells the script about the target architecture. If unspecified, the build system architecture is detected and suggested as a reasonable default.

For instance, the following command would prepare the Linux tree located at `/usr/src/linux-2.6.23-ipipe` in order to include the Xenomai support:

```
$ cd xenomai-2.4
$ scripts/prepare-kernel.sh --linux=/usr/src/linux-2.6.23-ipipe
```

Note: The script will infer the location of the Xenomai kernel code from its own location within the Xenomai source tree. In other words, if `/usr/src/xenomai-2.4/script/prepare-kernel.sh` is executing, then Xenomai's kernel support available from `/usr/src/xenomai-2.4/ksrc` will be bound to the target kernel.

2.2 Configuring and building the target kernel

Once the target kernel has been prepared, the kernel should be configured following its usual configuration procedure. All Xenomai configuration options are available from the "Real-time subsystem" toplevel menu.

There are several important kernel configuration options, some are documented in the TROUBLESHOOTING guide, others in the "Typical installation procedures" for the architecture you are using.

Once configured, the kernel should be built as usual.

If you want several different configs/builds at hand, you can reuse the same source by adding `O=../build-<target>` to each make invocation. See section "Building for the PowerPC architecture" for an example

In order to cross-compile the Linux kernel, pass an ARCH and CROSS_COMPILE variable on make command line. See sections "Building for the PowerPC architecture", "Building for the Blackfin", "Building for ARM" and "Building for NIOS II" for examples.

2.3 Building the user-space support

A regular autoconf script is provided in order to prepare for building the user-space support. The options listed below can be passed to this script. Those options only affect the libraries compiled as part of Xenomai's user-space support, but in any case, they never impact the kernel-based support.

2.3.1 Feature conflict resolution

Because of the strong decoupling between the kernel and user-space build procedures, Xenomai needs to make sure that all user-space options selected at configuration time will be consistent with the actual support the runtime libraries will get from the target kernel. For instance, enabling TSC support in user-space for x86 albeit the kernel has been compiled with `CONFIG_X86_TSC` disabled would certainly lead to runtime problems if uncaught, since Xenomai and the application would not agree on the high precision clock to use for their timings. Furthermore, most of these issues cannot be probed for during compilation, because the target generally has different features than the host, even when they are the same arch (ex 386 vs 686).

In order to solve those potential issues, each Xenomai architecture port defines a set of critical features which is tested for consistency, each time a user-space application binds itself to a real-time interface in kernel space. Unresolvable conflicts are reported and the execution stops immediately in such a case.

Options that need perfect matching between both sides are marked as "strong" in the following lists, others that may differ are marked as "weak". The way Xenomai deals with tolerated discrepancies is decided on a case-by-case basis, depending on the option considered. When not applicable, the binding type remains unspecified.

For instance, a kernel providing SMP support can run either UP or SMP user-space applications since the SMP option's binding is weak. On the other hand, x86-based applications linked against Xenomai libraries which have been compiled with the x86-tsc option on, must run on a kernel built with `CONFIG_X86_TSC` set, since the x86-tsc option's binding is strong.

2.3.2 Generic configure options

NAME	DESCRIPTION	[BINDING,] DEFAULT [1]
`--prefix`	Installation directory	`/usr/xenomai`
`--enable-debug`	Enable debug symbols (`-g`)	disabled
`--enable-smp`	Enable SMP support	weak, disabled

2.3.3 Arch-specific configure options

NAME	DESCRIPTION	[BINDING,] DEFAULT [1]
`--enable-x86-sep`	Enable x86 SEP instructions for issuing syscalls. You will also need NPTL.	strong, disabled

NAME	DESCRIPTION	[BINDING,] DEFAULT [1]
`--enable-x86-tsc`	Enable x86 TSC for timings You must have TSC for this.	strong, enabled
`--enable-arm-tsc`	Enable ARM TSC emulation. [2]	weak, kuser
`--enable-arm-quirks`	Enable quirks for specific ARM SOCs Currently sa1100 and xscale3 are supported.	weak, disabled

2.3.4 Cross-compilation

In order to cross-compile Xenomai user-space support, you will need to pass a `--host` and `--build` option to the configure script. The `--host` option allow to select the architecture for which the libraries and programs are built. The `--build` option allows to choose the architecture on which the compilation tools are run, i.e. the system running the configure script.

Since cross-compiling requires specific tools, such tools are generally prefixed with the host architecture name; for example, a compiler for the power PC architecture may be named `powerpc-405-linux-gnu-gcc`.

When passing the option `--host=powerpc-405-linux-gnu` to configure, configure will automatically use `powerpc-405-lin` as a prefix to all compilation tools names and infer the host architecture name from this prefix. If configure is unable to infer the architecture name from the cross-compilation tools prefix, you will have to manually pass the name of all compilation tools using at least the CC and LD, variables on configure command line. See sections "Building for the PowerPC architecture" and "Building for the Blackfin" for an example using the CC and LD variable, or "Building for ARM" for an example using the `--host` argument.

The easiest way to build a GNU cross-compiler might involve using crosstool-ng, available here.

If you want to avoid to build your own cross compiler, you might if find easier to use the ELDK. It includes the GNU cross development tools, such as the compilers, binutils, gdb, etc., and a number of pre-built target tools and libraries necessary to provide some functionality on the target system. See here for further details.

Some other pre-built toolchains:

- Mentor Sourcery CodeBench Lite Edition, available here;
- Linaro toolchain (for the ARM architecture), available here.

3 Typical installation procedures

The examples in following sections use the following conventions:

`$linux_tree`
: path to the target kernel sources

`$xenomai_root`
: path to the Xenomai sources

`$build_root`
: path to a clean build directory

`$staging_dir`
: path to a directory that will hold the installed file temporarily before they are moved to their final location; when used in a cross-compilation setup, it is usually a NFS mount point from the target's root directory to the local build host, as a consequence of which running `make DESTDIR=$staging_dir install` on the host immediately updates the target system with the installed programs and libraries.

[1] Each option enabled by default can be forcibly disabled by passing `--disable-<option>` to the configure script.

[2] In the unusual situation where Xenomai kernel support for the target SOC does not support the kuser generic emulation, pass this option to use another tsc emulation. See `--help` for a list of valid values.

3.1 Building for x86_32/64bit

Since Linux 2.6.24, x86_32 and x86_64 trees are merged. Therefore, building Xenomai for 2.6.24 or later is almost the same, regardless of the 32/64bit issue. You should note, however, that it is not possible to run xenomai libraries compiled for x86_32 with a kernel compiled for x86_64.

Assuming that you want to build natively for a x86_64 system (x86_32 cross-build options from x86_64 appear between brackets), you would typically run:

```
$ $xenomai_root/scripts/prepare-kernel.sh --arch=x86 \
  --adeos=$xenomai_root/ksrc/arch/x86/patches/adeos-ipipe-2.6.29.4-x86-X.Y-ZZ.patch \
  --linux=$linux_tree
$ cd $linux_tree
$ make [ARCH=i386] xconfig/gconfig/menuconfig
```

. . . configure the kernel (see also the recommended settings here).

Enable Xenomai options, then install as needed with:

```
$ make [ARCH=i386] bzImage modules
$ mkdir $build_root && cd $build_root
$ $xenomai_root/configure --enable-x86-sep (*) \
  [--host=i686-linux CFLAGS="-m32 -O2" LDFLAGS="-m32"]
$ make install
```

(*) Make sure to pass --enable-smp as well if building for a SMP-capable system.

Now, let's say that you really want to build Xenomai for a Pentium-based x86 32bit platform running a legacy 2.6.23 kernel, using the native host toolchain; the typical steps would be as follows:

```
$ $xenomai_root/scripts/prepare-kernel.sh --arch=i386 \
  --adeos=$xenomai_root/ksrc/arch/x86/patches/adeos-ipipe-2.6.23-i386-X.Y-ZZ.patch \
  --linux=$linux_tree
$ cd $linux_tree
$ make xconfig/gconfig/menuconfig
```

. . . configure the kernel (see also the recommended settings here).

Enable Xenomai options, then install as needed with:

```
$ make bzImage modules
$ mkdir $build_root && cd $build_root
$ $xenomai_root/configure --enable-x86-sep
$ make install
```

Similarly, for a legacy kernel on a 64bit platform, you would use:

```
$ $xenomai_root/scripts/prepare-kernel.sh --arch=x86_64 \
  --adeos=$xenomai_root/ksrc/arch/x86/patches/adeos-ipipe-2.6.23-x86_64-X.Y-ZZ.patch \
  --linux=$linux_tree
$ cd $linux_tree
$ make xconfig/gconfig/menuconfig
```

. . . configure the kernel (see also the recommended settings here).

Enable Xenomai options, then install as needed with:

```
$ make bzImage modules
$ mkdir $build_root && cd $build_root
$ $xenomai_root/configure
$ make install
```

Once the compilation has completed, /usr/xenomai should contain the user-space librairies and header files you would use to build applications that call Xenomai's real-time support in kernel space.

The remaining examples illustrate how to cross-compile Xenomai for various architectures. Of course, you will have to install the proper cross-compilation toolchain for the target system first, in order to build Xenomai.

3.2 Building for the PowerPC architecture

PowerPC has a legacy `arch/ppc` branch, and a newer, current `arch/powerpc` tree. Xenomai supports both, but using `arch/powerpc` is definitely recommended. To help the preparation script to pick the right one, you have to specify either `--arch=powerpc` (current) or `--arch=ppc` (legacy). Afterwards, the rest should be a no-brainer:

A typical cross-compilation setup, in order to build Xenomai for a lite5200 board running a recent 2.6.29.4 kernel. We use DENX's ELDK cross-compiler:

```
$ $xenomai_root/scripts/prepare-kernel.sh --arch=powerpc \
  --adeos=$xenomai_root/ksrc/arch/powerpc/patches/adeos-ipipe-2.6.29.4-powerpc-2.6-00.patch ↩
      \
  --linux=$linux_tree
$ cd $linux_tree
$ make ARCH=powerpc CROSS_COMPILE=ppc_6xx- xconfig/gconfig/menuconfig
```

. . . select the kernel and Xenomai options, save the configuration

```
$ make ARCH=powerpc CROSS_COMPILE=ppc_6xx- uImage modules
```

. . . manually install the u-boot image and modules to the proper location

```
$ cd $build_root
$ $xenomai_root/configure --host=powerpc-unknown-linux-gnu \
  CC=ppc_6xx-gcc AR=ppc_6xx-ar LD=ppc_6xx-ld
$ make DESTDIR=$staging_dir install
```

Another cross-compilation setup, in order to build Xenomai for a powerpc64 PA-Semi board running a recent 2.6.29.4 kernel:

```
$ $xenomai_root/scripts/prepare-kernel.sh --arch=powerpc \
  --adeos=$xenomai_root/ksrc/arch/powerpc/patches/adeos-ipipe-2.6.29.4-powerpc-2.6-00.patch ↩
      \
  --linux=$linux_tree
$ cd $linux_tree
$ make ARCH=powerpc CROSS_COMPILE=powerpc64-linux- xconfig/gconfig/menuconfig
```

. . . select the kernel and Xenomai options, save the configuration

```
$ make ARCH=powerpc CROSS_COMPILE=powerpc64-linux-
```

. . . manually install the vmlinux image and modules to the proper location

```
$ cd $build_root
$ $xenomai_root/configure --enable-smp --host=powerpc64-linux \
  CC=powerpc64-linux-gcc AR=powerpc64-linux-ar LD=powerpc64-linux-ld
$ make DESTDIR=$staging_dir install
```

Yet another cross-compilation setup, this time for building Xenomai for a PowerPC-405-based system running a legacy `arch/ppc` 2.6.14 kernel (we do support recent ones as well on this platform):

```
$ $xenomai_root/scripts/prepare-kernel.sh --arch=ppc \
  --adeos=$xenomai_root/ksrc/arch/powerpc/patches/adeos-ipipe-2.6.14-ppc-1.5-*.patch \
  --linux=$linux_tree
$ mkdir -p $build_root/linux
$ cd $linux_tree
$ make ARCH=ppc CROSS_COMPILE=ppc_4xx- O=$build_root/linux xconfig/gconfig/menuconfig
```

. . . select the kernel and Xenomai options, save the configuration

```
$ make ARCH=ppc CROSS_COMPILE=ppc_4xx- O=$build_root/linux bzImage modules
```

. . . manually install the kernel image, system map and modules to the proper location

```
$ make $build_root/xenomai && cd $build_root/xenomai
$ $xenomai_root/configure --build=i686-pc-linux-gnu --host=ppc-unknown-linux-gnu \
  CC=ppc_4xx-gcc LD=ppc_4xx-ld
$ make DESTDIR=$staging_dir install
```

3.3 Building for the Blackfin

The Blackfin is an MMU-less, DSP-type architecture running uClinux.

```
$ $xenomai_root/scripts/prepare-kernel.sh --arch=blackfin \
  --adeos=$xenomai_root/ksrc/arch/blackfin/patches/adeos-ipipe-bf53x-*.patch \
  --linux=$linux_tree
$ cd $linux_tree
$ make ARCH=blackfin CROSS_COMPILE=bfin-uclinux- xconfig/gconfig/menuconfig
```

. . . select the kernel and Xenomai options, then compile with:

```
$ make linux image
```

. . . then install as needed

```
$ cp images/linux /tftpboot/...
```

. . . build the user-space support

```
$ mkdir $build_root && cd $build_root
$ $xenomai_root/configure --host=blackfin-unknown-linux-gnu \
  CC=bfin-linux-uclibc-gcc AR=bfin-linux-uclibc-ar LD=bfin-linux-uclibc-ld
$ make DESTDIR=$staging_dir install
```

You may also want to have a look at the hands-on description about configuring and building a Xenomai system for the Blackfin architecture, available at this address.

Note
Xenomai uses the FDPIC shared library format on this architecture. In case of problem running the testsuite, try restarting the last two build steps, passing the `--disable-shared` option to the "configure" script.

3.4 Building for ARM

Using codesourcery toolchain named `arm-none-linux-gnueabi-gcc` and compiling for a CSB637 board (AT91RM9200 based), a typical compilation will look like:

```
$ $xenomai_root/scripts/prepare-kernel.sh --arch=arm \
  --adeos=$xenomai_root/ksrc/arch/arm/patches/adeos-ipipe-2.6.20-arm-* \
  --linux=$linux_tree
$ cd $linux_tree
$ mkdir -p $build_root/linux
$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi- O=$build_root/linux \
  csb637_defconfig
$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi- O=$build_root/linux \
  bzImage modules
```

. . . manually install the kernel image, system map and modules to the proper location

```
$ mkdir $build_root/xenomai && cd $build_root/xenomai
$ $xenomai_root/configure CFLAGS="-march=armv4t" LDFLAGS="-march=armv4t" \
  --build=i686-pc-linux-gnu --host=arm-none-linux-gnueabi-
$ make DESTDIR=$staging_dir install
```

Important

Contrarily to previous releases, Xenomai no longer passes any arm architecture specific flags, or FPU flags to gcc, so, users are expected to pass them using the CFLAGS and LDFLAGS variables as demonstrated above, where the AT91RM9200 is based on the ARM920T core, implementing the `armv4` architecture. The following table summarizes the CFLAGS and options which were automatically passed in previous revisions and which now need to be explicitely passed to configure, for the supported SOCs:

Table 1: ARM configure options and compilation flags

SOC	CFLAGS	configure options
at91rm9200	`-march=armv4t -msoft-float`	
at91sam9x	`-march=armv5 -msoft-float`	
imx1	`-march=armv4t -msoft-float`	
imx21	`-march=armv5 -msoft-float`	
imx31	`-march=armv6 -mfpu=vfp`	
imx51/imx53	`-march=armv7-a -mfpu=vfp` [3]	
imx6q	`-march=armv7-a -mfpu=vfp` [3]	`--enable-smp`
ixp4xx	`-march=armv5 -msoft-float`	`--enable-arm-tsc=ixp4xx`
omap3	`-march=armv7-a -mfpu=vfp` [3]	
omap4	`-march=armv7-a -mfpu=vfp` [3]	`--enable-smp`
orion	`-march=armv5 -mfpu=vfp`	
pxa	`-march=armv5 -msoft-float`	
pxa3xx	`-march=armv5 -msoft-float`	`--enable-arm-quirks=xscale3`
s3c24xx	`-march=armv4t -msoft-float`	
sa1100	`-march=armv4t -msoft-float`	`--enable-arm-quirks=sa1100`

It is possible to build for an older architecture version (v6 instead of v7, or v4 instead of v5), if your toolchain does not support the target architecture, the only restriction being that if SMP is enabled, the architecture should not be less than v6.

3.5 Building for NIOS II

NIOS II is a softcore processor developped by Altera and is dedicated to the Altera's FPGA circuits.

NIOS II with no MMU enabled is supported by the uClinux distribution.

[3] Depending on the gcc versions the flag for armv7 may be `-march=armv7-a` or `-march=armv7a`

3.5.1 Minimum hardware requirements

You have to start with a minimal system with at least:

- A Nios II processor in f or s core version, with hardware multiplier, (f-core suggested, s-core is slower) and with no MMU enabled.
- SDRAM (minimum requirement 8MB).
- One full featured timer named sys_clk_timer used for uClinux.
- A jtag/serial uart or a real serial uart (preferred).

Note in Linux, IRQ 0 means auto-detected, so you must not use IRQ 0 for ANY devices.

The Xenomai port for NIOS II uses extra hardware that you have to add in SOPC builder:

- A full featured 32-bit Timer named hrtimer with a 1 microsecond period.
- A full featured High Resolution 64-bit Timer named hrclock used for time stamping (1 microsecond period for example).

Important
Please respect hrtimer and hrclock names, the Xenomai port depends on them!

You have to use Altera's Quartus II version 9.0 at least for synthesis.

A good start for your design is to use reference design shipped with your target board.

For example, with an Altera's board, you may use the *standard* design. *Standard* reference designs for Altera's boards are available here.

3.5.2 Xenomai compilation for NIOS II

You should first verify that uClinux without Xenomai can run on the target board.

The typical actions for building the uClinux kernel for NIOS II (available here) are:

If `$uClinux-dist` is the path of NIOS II uClinux release, for example:

`/home/test/nios2-linux/uClinux-dist`

```
$ cd $uClinux-dist
$ make menuconfig
$ make vendor_hwselect SYSPTF=<path to your system ptf>
$ make
```

If the NIOS II cross-compiler is called `nios2-linux-gcc`, a typical compilation will look like:

```
$ $xenomai_root/scripts/prepare-kernel.sh --arch=nios2 \
  --adeos=$xenomai_root/ksrc/arch/nios2/patches/adeos-ipipe-2.6.26-rc6-nios2-* \
  --linux=$linux_tree
$ $xenomai_root/configure --host=nios2-linux
$ make install DESTDIR=$uClinux-dist/romf
$ cd $uClinux-dist
$ make
```

4 Testing the installation

4.1 Testing the kernel

In order to test the Xenomai installation, you should first try to boot the patched kernel. The kernel boot logs should show messages like:

```
I-pipe: head domain Xenomai registered.
Xenomai: hal/<arch> started.
Xenomai: scheduling class idle registered.
Xenomai: scheduling class rt registered.
Xenomai: real-time nucleus v2.6.1 (Light Years Away) loaded.
Xenomai: debug mode enabled.
Xenomai: starting native API services.
Xenomai: starting POSIX services.
Xenomai: starting RTDM services.
```

Where <arch> is the architecture you are using. If the kernel fails to boot, or the log messages indicates an error status instead, see the TROUBLESHOOTING guide.

4.2 Testing the user-space support

In order to test Xenomai user-space support, launch the latency test:

```
$ /usr/xenomai/bin/latency
```

The latency test should display a message every second with minimal, maximal and average latency values, such as:

```
# latency -T 25
== Sampling period: 100 us
== Test mode: periodic user-mode task
== All results in microseconds
warming up...
RTT|  00:00:01  (periodic user-mode task, 100 us period, priority 99)
RTH|----lat min|----lat avg|----lat max|-overrun|---msw|---lat best|--lat worst
RTD|      1.615|      1.923|      9.846|       0|     0|      1.615|      9.846
RTD|      1.615|      1.923|      9.692|       0|     0|      1.615|      9.846
RTD|      1.538|      1.923|     10.230|       0|     0|      1.538|     10.230
RTD|      1.615|      1.923|     10.384|       0|     0|      1.538|     10.384
RTD|      1.615|      1.923|     11.230|       0|     0|      1.538|     11.230
RTD|      1.615|      1.923|      9.923|       0|     0|      1.538|     11.230
RTD|      1.615|      1.923|      9.923|       0|     0|      1.538|     11.230
RTD|      1.615|      1.923|     11.076|       0|     0|      1.538|     11.230
RTD|      1.615|      1.923|     10.538|       0|     0|      1.538|     11.230
RTD|      1.615|      1.923|     11.076|       0|     0|      1.538|     11.230
RTD|      1.615|      1.923|     10.615|       0|     0|      1.538|     11.230
RTD|      1.615|      1.923|     10.076|       0|     0|      1.538|     11.230
RTD|      1.615|      1.923|      9.923|       0|     0|      1.538|     11.230
RTD|      1.538|      1.923|     10.538|       0|     0|      1.538|     11.230
RTD|      1.615|      1.923|     10.923|       0|     0|      1.538|     11.230
RTD|      1.538|      1.923|     10.153|       0|     0|      1.538|     11.230
RTD|      1.615|      1.923|      9.615|       0|     0|      1.538|     11.230
RTD|      1.615|      1.923|     10.769|       0|     0|      1.538|     11.230
RTD|      1.615|      1.923|      9.153|       0|     0|      1.538|     11.230
RTD|      1.538|      1.923|     10.307|       0|     0|      1.538|     11.230
RTD|      1.615|      1.923|      9.538|       0|     0|      1.538|     11.230
RTT|  00:00:22  (periodic user-mode task, 100 us period, priority 99)
RTH|----lat min|----lat avg|----lat max|-overrun|---msw|---lat best|--lat worst
RTD|      1.615|      1.923|     11.384|       0|     0|      1.538|     11.384
RTD|      1.615|      1.923|     10.076|       0|     0|      1.538|     11.384
```

```
RTD|      1.538|      1.923|      9.538|        0|     0|      1.538|     11.384
---|-----------|-----------|-----------|---------|------|--------------------------
RTS|      1.538|      1.923|     11.384|        0|     0|    00:00:25/00:00:25
#
```

If the latency test displays an error message, hangs, or displays unexpected values, see the TROUBLESHOOTING guide.

If the latency test succeeds, you should try next to run a latency test under load to evaluate the latency test of your system, the `xeno-test` script allows doing that. Try:

```
$ xeno-test --help
```