# Installing Xenomai 3.x

| REVISION HISTORY | | | |
|---|---|---|---|
| NUMBER | DATE | DESCRIPTION | NAME |
| | | | |

# Contents

# 1 Introduction

Xenomai 3 is the new architecture of the Xenomai real-time framework, which can run seamlessly side-by-side Linux as a co-kernel system like Xenomai 2, or natively over mainline Linux kernels. In the latter case, the mainline kernel can be supplemented by the PREEMPT-RT patch to meet stricter response time requirements than standard kernel preemption would bring.

This new architecture therefore exhibits two real-time cores, selected at build time. The dual kernel core nicknamed *Cobalt*, is a significant rework of the Xenomai 2.x system. The native linux version, an enhanced implementation of the experimental Xenomai/SOLO work, is called *Mercury*.

This magic works with the introduction of the *Copperplate* interface, which mediates between the real-time API/emulator your application uses, and the underlying real-time core. This way, applications are able to run in either environments without visible code change.

# 2 Installation steps

Xenomai follows a split source model, decoupling the kernel space support from the user-space libraries.

To this end, kernel and user-space Xenomai components are respectively available under the `kernel/` and `lib/` sub-trees. Other top-level directories, such as `scripts/`, `testsuite/` and `utils/`, provide additional scripts and programs to be used on either the build host, or the runtime target.

The `kernel/` sub-tree which implements the in-kernel support code is seen as a built-in extension of the Linux kernel. Therefore, the standard Linux kernel configuration process should be used to define the various settings for the Xenomai kernel components. All of the kernel code Xenomai currently introduces implements the *Cobalt* core (i.e. dual kernel configuration). As of today, the *Mercury* core needs no Xenomai-specific code in kernel space.

The `lib/` sub-tree contains the various user-space libraries exported by the Xenomai framework to the applications. This tree is built separately from the kernel support. Libraries are built in order to support the selected core, either *Cobalt* or *Mercury*.

# 3 Installing the *Cobalt* core

## 3.1 Preparing the *Cobalt* kernel

*Xenomai/cobalt* provides a real-time extension kernel 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>]
[--ipipe=<ipipe-patch>] [--arch=<target-arch>]
```

**`--linux`**
: specifies the path of the target kernel source tree. Such kernel tree may be already configured or not, indifferently. This path defaults to $PWD.

**`--ipipe`**
: specifies the path of the interrupt pipeline (aka I-pipe) patch to apply against the kernel tree. Suitable patches are available with *Xenomai/cobalt* under `kernel/cobalt/arch/<target-arch>/patches`. This parameter can be omitted if the I-pipe has already been patched in, or the script shall suggest an appropriate one. The script will detect whether the interrupt pipeline code is already present into the kernel tree, and skip this operation if so.

**`--arch`**
: tells the script about the target architecture. If unspecified, the build host architecture suggested as a reasonable default.

For instance, the following command would prepare the Linux tree located at `/home/me/linux-3.10-ipipe` in order to patch the Xenomai support in:

```
$ cd xenomai-3
$ scripts/prepare-kernel.sh --linux=/home/me/linux-3.10
```

Note: The script will infer the location of the Xenomai kernel code from its own location within the Xenomai source tree. For instance, if `/home/me/xenomai-3/scripts/prepare-kernel.sh` is executing, then the Xenomai kernel code available from `/home/me/xenomai-3/kernel/cobalt` will be patched in the target Linux kernel.

## 3.2 Configuring and compiling the *Cobalt* kernel

Once prepared, the target kernel can be configured as usual. All Xenomai configuration options are available from the "Xenomai" toplevel Kconfig menu.

There are several important kernel configuration options, documented in the TROUBLESHOOTING guide.

Once configured, the kernel can be compiled as usual.

If you want several different configs/builds at hand, you may reuse the same source by adding `O=../build-<target>` to each make invocation.

In order to cross-compile the Linux kernel, pass an ARCH and CROSS_COMPILE variable on make command line. See sections "Building a Cobalt/arm kernel", "Building a Cobalt/powerpc kernel", "Building a Cobalt/blackfin kernel", "Building a Cobalt/x86 kernel", for examples.

## 3.3 Examples of building the *Cobalt* kernel

The examples in following sections use the following conventions:

**`$linux_tree`**
: path to the target kernel sources

**`$xenomai_root`**
: path to the Xenomai sources

### 3.3.1 Building a *Cobalt/x86* kernel (32/64bit)

Building *Xenomai/cobalt* for x86 is almost the same for 32bit and 64bit platforms. You should note, however, that it is not possible to run Xenomai libraries compiled for x86_32 on a kernel compiled for x86_64, and conversely.

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:

```
$ cd $linux_tree
$ $xenomai_root/scripts/prepare-kernel.sh --arch=x86 \
  --ipipe=$xenomai_root/kernel/cobalt/arch/x86/patches/ipipe-core-X.Y.Z-x86-NN.patch
$ make [ARCH=i386] xconfig/gconfig/menuconfig
```

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

Enable Xenomai options, then build with:

```
$ make [ARCH=i386] bzImage modules
```

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

```
$ cd $linux_tree
$ $xenomai_root/scripts/prepare-kernel.sh --arch=i386 \
  --ipipe=$xenomai_root/kernel/cobalt/arch/x86/patches/ipipe-core-X.Y.Z-x86-NN.patch
$ make xconfig/gconfig/menuconfig
```

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

Enable Xenomai options, then build with:

```
$ make bzImage modules
```

Similarly, for a 64bit platform, you would use:

```
$ cd $linux_tree
$ $xenomai_root/scripts/prepare-kernel.sh --arch=x86_64 \
  --ipipe=$xenomai_root/kernel/cobalt/arch/x86/patches/ipipe-core-X.Y.Z-x86-NN.patch
$ make xconfig/gconfig/menuconfig
```

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

Enable Xenomai options, then build with:

```
$ make bzImage modules
```

The remaining examples illustrate how to cross-compile a *Cobalt*-enabled kernel for various architectures. Of course, you would have to install the proper cross-compilation toolchain for the target system first.

#### 3.3.2 Building a *Cobalt/powerpc* kernel (32/64bit)

A typical cross-compilation setup, in order to build Xenomai for a ppc-6xx architecture running a 3.10.32 kernel. We use the DENX ELDK cross-compiler:

```
$ cd $linux_tree
$ $xenomai_root/scripts/prepare-kernel.sh --arch=powerpc \
  --ipipe=$xenomai_root/kernel/cobalt/arch/powerpc/patches/ipipe-core-3.10.32-powerpc-1. ↩
      patch
$ make ARCH=powerpc CROSS_COMPILE=ppc_6xx- xconfig/gconfig/menuconfig
```

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

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

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

#### 3.3.3 Building a *Cobalt/blackfin* kernel

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

```
$ cd $linux_tree
$ $xenomai_root/scripts/prepare-kernel.sh --arch=blackfin \
  --ipipe=$xenomai_root/kernel/cobalt/arch/blackfin/patches/ipipe-core-X.Y.Z-x86-NN.patch
$ 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/...
```

#### 3.3.4 Building *Cobalt/arm* kernel

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

```
$ cd $linux_tree
$ $xenomai_root/scripts/prepare-kernel.sh --arch=arm \
  --ipipe=$xenomai_root/kernel/cobalt/arch/arm/patches/ipipe-core-X.Y.Z-x86-NN.patch
$ 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

# 4 Installing the *Mercury* core

For *Mercury*, you need no Xenomai-specific kernel support so far, beyond what your host Linux kernel already provides. Your kernel should at least provide high resolution timer support (`CONFIG_HIGH_RES_TIMERS`), and likely complete preemption (*PREEMPT_RT*) if your application requires short and bounded latencies.

Kernels with no real-time support can be used too, likely for basic debugging tasks, and/or running applications which do not have strict response time requirements.

Therefore, unlike with *Cobalt*, there is no additional steps for preparing and/or configuring the kernel for *Mercury*.

# 5 Installing the Xenomai libraries and tools

## 5.1 Prerequisites

### 5.1.1 Generic requirements (both cores)

- GCC must have support for legacy atomic builtins (__sync form).
- GCC should have a (sane/working) support for TLS preferably, although this is not mandatory if building with `--disable-tls`.

### 5.1.2 *Cobalt*-specific requirements

- The kernel version must be 3.10 or better.
- An interrupt pipeline (I-pipe) patch must be available for your target kernel. You can find the official patches issued by the Xenomai project there. Only patches from the **ipipe-core** series are appropriate, legacy patches from the **adeos-ipipe** series are not.
- A timestamp counter (TSC) is required from running on a x86_32 hardware. Unlike with Xenomai 2.x, TSC-emulation using a PIT register is not available.

### 5.1.3 *Mercury*-specific requirement

- There is no particular requirement for Mercury setups, although using a NPTL-based glibc or uClibc is recommended.

## 5.2 Configuring

A common autoconf script prepares for building the libraries and programs, for both the *Cobalt* and *Mercury* cores. The core-specific code which may be needed internally is automatically and transparently selected at compilation-time by the build process.

The options listed below can be passed to this script.

### 5.2.1 Generic configuration options (both cores)

**--with=core=<type>** Indicates which real-time core you want to build the support libraries for, namely *cobalt* or *mercury*. This option defaults to *cobalt*.

**--prefix=<dir>** Specifies the root installation path for libraries, include files, scripts and executables. Running `$ make install` installs these files to `$DESTDIR/<dir>`. This directory defaults to /usr/xenomai.

**--enable-debug[=partial]** This switch controls the debug level. Three levels are available, with varying overhead:

- *symbols* enables debug symbols to be compiled in the libraries and executables, still turning on the optimizer (-O2). This option has no overhead, it is useful to get meaningful backtraces using gdb while running the application at nominal speed.
- *partial* includes *symbols*, and also turns on internal consistency checks within the Xenomai code (mostly present in the Copperplate layer). The `CONFIG_XENO_DEBUG` macro is defined, for both the Xenomai libraries and the applications getting their C compilation flags from the `xeno-config` script (i.e. `xeno-config --cflags`). The partial debug mode implicitly turns on `--enable-assert`. A measurable overhead is introduced by this level. This is the default level when `--enable-debug` is mentioned with no level specification.
- *full* includes *partial* settings, but the optimizer is disabled (-O0), and even more consistency checks may be performed. In addition to `__XENO_DEBUG__`, the macro `CONFIG_XENO_DEBUG_FULL` is defined. This level introduces the most overhead, which may triple the worst-case latency, or even more.

  Over the *Mercury* core, enabling *partial* or *full* debug modes also causes the standard malloc interface to be used internally instead of a fast real-time allocator (TLSF). This allows debugging memory-related issues with the help of *Valgrind* or other dynamic memory analysers.

**--disable-debug** Fully turns off all consistency checks and assertions, turns on the optimizer and disables debug symbol generation.

**--enable-assert** A number of debug assertion statements are present into the Xenomai libraries, checking the internal consistency of the runtime system dynamically (see *man assert(3)*). Passing `--disable-assert` to the *configure* script disables built-in assertions unconditionally. By default, assertions are enabled in partial or full debug modes, disabled otherwise.

**--enable-pshared** Enable shared multi-processing. When enabled, this option allows multiple processes to share real-time objects (e.g. tasks, semaphores).

**--enable-registry** Xenomai APIs can export their internal state through a pseudo-filesystem, which files may be read to obtain information about the existing real-time objects, such as tasks, semaphores, message queues and so on. This feature is supported by FUSE, which must be available on the target system. Building the Xenomai libraries with the registry support requires the FUSE development libraries to be installed on the build system.
When this option is enabled, the system creates a file hierachy under `/mnt/xenomai/<session>.<pid>` (by default), where you can access the internal state of the active real-time objects. The session label is obtained from the --session runtime switch. E.g. looking at the properties of a VxWorks task could be done as follows:

```
$ cat /mnt/xenomai/anon.12656/vxworks/tasks/windTask
name       = windTask
errno      = 0
status     = ready
priority   = 70
lock_depth = 0
```

You may override the default root of the registry hierarchy by using the `--registry-root` runtime option (see below).

---

**Note**
When running over *Xenomai/cobalt*, the `/proc/xenomai` interface is also available for inspecting the core system state.

---

**--enable-lores-clock**
Enables support for low resolution clocks. By default, libraries are built with no support for tick-based timing. If you need such support (e.g. for pSOS ™ or VxWorks ™ APIs), then you can turn it on using this option.

---

**Note**
The POSIX API does not support tick-based timing. Alchemy may use it optionally.

---

**--enable-clock-monotonic-raw**
The Xenomai libraries requires a monotonic clock to be available from the underlying POSIX interface. When `CLOCK_MONOTONIC_RAW` is available on your system, you may want to pass this switch, otherwise `CLOCK_MONOTONIC` will be used by default.

---

**Note**
The *Cobalt* core implements `CLOCK_MONOTONIC_RAW`, so this switch is turned on by default when building with `--with-core=cobalt`. On the contrary, this option is turned off by default when building for the *Mercury* core, since we don't know in advance whether this feature does exist on the target kernel.

---

**--enable-tls**
Xenomai can use GCC's thread local storage extension (TLS) to speed up the retrieval of the per-thread information it uses internally. This switch enables TLS, use the converse `--disable-tls` to prevent this.

Due to GCC bugs regarding this feature with some release,architecture combinations, whether TLS is turned on by default is a per-architecture decision. Currently, this feature is enabled for x86 and powerpc by default, other architectures will require `--enable-tls` to be passed to the *configure* script explicitly.

Unless `--enable-dlopen-libs` is present, the *initial-exec* TLS model is selected.

When TLS is disabled, POSIX's thread-specific data management services are used internally (i.e. pthread_set/getspecific()).

**--enable-dlopen-libs**
This switch allows programs to load Xenomai-based libraries dynamically, using the `dlopen(3)` routine. Enabling dynamic loading introduces some overhead in TLS accesses when enabled (see `--enable-tls`), which might be noticeable depending on the architecture.

To support dynamic loading when `--enable-tls` is turned on, the *global-dynamic* TLS model is automatically selected.

Applications loading `libcobalt.so` dynamically may want to create the XENO_NOSHADOW environment variable prior to calling `dlopen()`, to prevent auto-shadowing of the calling context.

Dynamic loading of Xenomai-based libraries is disabled by default.

**--enable-async-cancel**
Enables fully asynchronous cancellation of Xenomai threads created by the real-time APIs, making provision to protect the Xenomai implementation code accordingly.

When disabled, Xenomai assumes that threads may exit due to cancellation requests only when they reach cancellation points (like system calls). Asynchronous cancellation is disabled by default.

**Caution**
Fully asynchronous cancellation can easily lead to resource leakage, silent corruption, safety issues and all sorts of rampant bugs. The only reason to turn this feature on would be aimed at cancelling threads which run significantly long, syscall-less busy loops with no explicit exit condition, which should probably be revisited anyway.

**--enable-smp**
Turns on SMP support for Xenomai libraries.

**Caution**
SMP support must be enabled in Xenomai libraries when the client applications are running over a SMP-capable kernel.

**--disable-sanity**
Turns off the sanity checks performed at application startup by the Xenomai libraries. This option sets a default, which can later be overriden using the --[no-]sanity options passed to a Copperplate-based Xenomai application. Sanity checks are enabled by default when configuring.

**--enable-fortify**
Enables support for applications compiled in `_FORTIFY_SOURCE` mode.

**--disable-valgrind-client**
Turns off the Valgrind client support, forcing `CONFIG_XENO_VALGRIND_API` off in the Xenomai configuration header.

**--enable-doc-build**
Causes the inline Xenomai documentation based on the Doxygen markup language to be produced as PDF and HTML documents. Additional documentation like manpages based on the Asciidoc markup language is produced too.

A pre-built copy of the documentation is present in the source tree, under the `doc/generated/` file hierarchy.

**--disable-doc-install**
Disables the copying of the pre-built documentation to the installation directory.

### 5.2.2 *Cobalt*-specific configuration options

| NAME | DESCRIPTION | DEFAULT |
|---|---|---|
| `--enable-x86-vsyscall` | Use the x86/vsyscall interface for issuing syscalls. If disabled, the legacy 0x80 vector will be used. Turning on this option requires NPTL. | enabled |
| `--enable-arm-tsc` | Enable ARM TSC emulation. [1] | kuser |
| `--enable-arm-quirks` | Enable quirks for specific ARM SOCs Currently sa1100 and xscale3 are supported. | disabled |

[2]

### 5.3 Cross-compilation

In order to cross-compile the Xenomai libraries and programs, 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 PowerPC architecture may be named `powerpc-linux-gcc`.

When passing `--host=powerpc-linux` to configure, it will automatically use `powerpc-linux-` 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.

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 required 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.

## 6 Examples of building the Xenomai libraries and tools

The examples in following sections use the following conventions:

**`$xenomai_root`**
: path to the Xenomai sources

**`$build_root`**
: path to a clean build directory

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

**Caution**
In the examples below, make sure to add `--enable-smp` to the *configure* script options if building for a SMP-enabled kernel.

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

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

### 6.1 Building the x86 libraries (32/64bit)

Assuming that you want to build the *Mercury* libraries natively for a x86_64/SMP system, enabling shared multi-processing support. You would typically run:

```
$ mkdir $build_root && cd $build_root
$ $xenomai_root/configure --with-core=mercury --enable-smp --enable-pshared
$ make install
```

Conversely, cross-building the *Cobalt* libraries from x86_64 with the same feature set, for running on x86_32 could be:

```
$ mkdir $build_root && cd $build_root
$ $xenomai_root/configure --with-core=cobalt --enable-smp --enable-pshared \
  --host=i686-linux CFLAGS="-m32 -O2" LDFLAGS="-m32"
$ make install
```

After installing the build tree (i.e. using "make install"), the installation root should be populated with the librairies, programs and header files you can use to build Xenomai-based real-time applications. This directory path defaults to `/usr/xenomai`.

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

### 6.2 Building the PPC32 libraries

A typical cross-compilation setup, in order to build the *Cobalt* libraries for a ppc-6xx architecture. In that example, we want the debug symbols to be generated for the executable, with no runtime overhead though. We use the DENX ELDK cross-compiler:

```
$ cd $build_root
$ $xenomai_root/configure --host=powerpc-linux --with-core=cobalt \
  --enable-debug=symbols
$ make DESTDIR=$staging_dir install
```

### 6.3 Building the PPC64 libraries

Same process than for a 32bit PowerPC target, using a crosstool-built toolchain for ppc64/SMP.

```
$ cd $build_root
$ $xenomai_root/configure --host=powerpc64-unknown-linux-gnu \
  --with-core=cobalt --enable-smp
$ make DESTDIR=$staging_dir install
```

### 6.4 Building the Blackfin libraries

Another cross-compilation setup, in order to build the *Cobalt* libraries for the Blackfin architecture. We use ADI's toolchain for this purpose:

```
$ mkdir $build_root && cd $build_root
$ $xenomai_root/configure --host=bfin-linux-uclibc --with-core=cobalt
$ make DESTDIR=$staging_dir install
```

---

**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.

---

### 6.5 Building the ARM libraries

Using codesourcery toolchain named `arm-none-linux-gnueabi-gcc` and compiling for a CSB637 board (AT91RM9200 based), a typical cross-compilation from a x86_32 desktop would look like:

```
$ 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- --with-core=cobalt
$ make DESTDIR=$staging_dir install
```

**Important**

Unlike 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=vfp3` [3] | |
| imx6q | `-march=armv7-a -mfpu=vfp3` [3] | `--enable-smp` |
| ixp4xx | `-march=armv5 -msoft-float` | `--enable-arm-tsc=ixp4xx` |
| omap3 | `-march=armv7-a -mfpu=vfp3` [3] | |
| omap4 | `-march=armv7-a -mfpu=vfp3` [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] Depending on the gcc versions the flag for armv7 may be `-march=armv7-a` or `-march=armv7a`

# 7 Testing the installation

## 7.1 Booting the *Cobalt* kernel

In order to test the Xenomai installation over *Cobalt*, you should first try to boot the patched kernel. Check the kernel boot log for messages like these:

```
$ dmesg | grep -i xenomai
I-pipe: head domain Xenomai registered.
[Xenomai] Cobalt vX.Y.Z enabled
```

If the kernel fails booting, or the log messages indicates an error status instead, see the TROUBLESHOOTING guide.

## 7.2 Testing the real-time system (both cores)

First, run the latency test:

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

The latency test should display a message every second with minimum, maximum and average latency values. If this 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 the `xeno-test` test in order to assess the worst-case latency of your system. Try:

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

# 8 Building and running Xenomai 3 applications

Once the latency test behaves as expected on your target system, it is deemed ready to run real-time applications.

You may want to have a look at this document for details about the application build process.

In addition, you should refer to this document to learn about the builtin command line switches available with any Xenomai 3 application.

# 9 Migrating applications to Xenomai 3

If you plan to port an existing application based on Xenomai 2.x to Xenomai 3.x, you should have a look at this migration guide.