Name

xdc - eXpress DSP Component package build command

Synopsis

```
xdc [-n] [-h] [-k] [goal ...] [-P[RD] package-dir ...]
```

Description

The `xdc` command is used to build packages and executables that use packages. In its current implementation, `xdc` is nothing more than a command shell that invokes the GNU `make` utility with makefiles that are either part of the XDC toolset or generated as part of processing a package's build script (`package.bld`).

Options

`-h`	display usage help and exit
`-k`	if an error occurs during a build, do not stop (keep building as much as possible)
`-n`	show the `make` command but don't execute it
`-P[RD] pkg ...`	build specified goal(s) in all directories named after `-P` that contain a build script (i.e., `package.bld`); if `-PR` is specified, `xdc` recursively descends into all specified directories and builds in any package directory that contains a build script; if `-PD` is specified, `xdc` builds in the specified package directories and the package directories of all other packages "required" by these packages.

Usage

Any option passed on the `xdc` command line that is not listed above is passed directly to the underlying invocation of GNU `make`. In this way, one can control the build process with as much flexibility as a normal GNU `make`. In particular, the goals specified on the command line may be any file that `make` knows how to build.

Although it is always possible to name one or more specific buildable files on the `xdc` command line, it is often desirable to build collections of executables or libraries or run collections of tests. For example, building a test suite consisting of hundreds of executables would be difficult if each program had to be explicitly named on an `xdc` command line.

The `xdc` command supports a number of build goals that facilitate the building of collections of files. The following table summarizes some standard build goals supported by the `xdc` command.

Goal	Description
`all`	builds all package files and all executables; this is the default goal if no goals are specified on the command line.
`clean`	removes all generated files
`release`	create the default release; a tar file containing all files that are to be distributed to a consumer of the package.
`test`	runs all package tests declared in the package's build script
`.make`	builds the generated makefiles only
`.libraries`	builds all libraries exported by this package
`.executables`	builds all executables declared in the package's build script
`.interfaces`	builds the package's schema and generates all header files for interfaces exported by the package

In addition to the target independent goals above, target specific goals are also supported. These goals allow one to restrict a build to just the libraries for a particular target, for example. In addition, these goals are used internally to prevent unnecessarily building libraries for all targets before building a specific executable. The following table summarizes the target specific goals currently supported.

Goal	Description
`all,`*trg*	builds all libraries and all executables for the target whose suffix is *trg*.
`clean,`*trg*	removes all generated files for the target whose suffix is *trg*.
`release,`*name*	create the release named *name*.
`test,`*trg*	run all tests for all executables built using the target whose suffix is *trg*.
`.libraries,`*trg*	builds all libraries for the target whose suffix is *trg*
`.executables,`*trg*	builds all executables for the target whose suffix is *trg*

The `-P` and `-PR` options are very useful when working with multiple packages. Not only does it allow one to avoid building each package using a separate command it automatically handles any package inter-dependencies. A package may depend on another package; for example, any package that contains a program depends on at least one platform package. If two or more dependent

packages are under development at the same time, it is important that the interfaces for the packages referenced are built before any libraries referencing these packages are built and that any libraries referenced are built before linking any executable. Thus, when building multiple packages, the `xdc` command builds the packages in several "phases"; it first builds all interfaces for all packages, then all libraries for all packages, and finally, all executables for all packages. By making several passes over the packages, it is possible to simultaneously develop multiple packages without having to worry about package build order due to package dependencies (which are subject to change).

Examples

Building a Package

The following command will remove all generated files from the package located in the current working directory.

```
xdc clean
```

To build all files for the package in the current working directory:

```
xdc
```

Building Specific Files

While it is valuable to (re)build a package in its entirety, during development, it is often more convenient to build a specific file. The `xdc` command allows one to specify any generated file as the goal to be built. For example, the following command will build the program `Hello.x62` and files unrelated to this program will not be built.

```
xdc Hello.x62
```

It is also possible to name several goals and, again, `xdc` will only build the files required to create the build goals named. For example, the following command will build both executables `Hello.x62` and `Hello.x62e`.

```
xdc Hello.x62 Hello.x62e
```

Running Package Tests

The XDC build environment supports the ability to not only build executables and packages but also run executables. The XDC environment defines a test as an executable and a set of command line arguments for the executable. Tests may be specified in the package's build script and every program defined in a package has one implicitly

created test; the executable with no command line arguments. The `xdc` command allows one to easily run a package's tests.

The following command runs all tests defined by the package in the current working directory.

```
xdc test
```

As with building individual files, it is often desirable to run individual tests. The following command runs the implicitly created test for the `Hello.x62` executable.

```
xdc Hello.x62.test
```

Building Multiple Packages

The following command will remove all generated files from the packages `pkg1`, `pkg2`, and `pkg3` located in the current working directory.

```
xdc clean -P pkg1 pkg2 pkg3
```

To build all files in multiple packages:

```
xdc all -P pkg1 pkg2 pkg3
```

Because the build goal `all` is the default goal, the following command is equivalent to the one above.

```
xdc -P pkg1 pkg2 pkg3
```

Note that the `xdc` command will silently ignore directories that do not contain a package build script (`package.bld`). Thus, even if a directory contains sub-directories that are not package directories it is possible to build all packages contained in this directory using a wildcard. Suppose, for instance, that the directory named `examples` contains multiple sub-directories and only some are package directories. It is possible to build all packages in the `examples` directory with the following command.

```
xdc -P examples/*
```

Since a package's name must match the directory names containing the package, it is not uncommon for packages to be located at different levels of a directory tree or even inside other packages. In these cases, it is desirable to be able to build all packages contained under a specified root directory. The following command builds all packages located under the examples directory.

```
xdc -PR examples
```

Rather than build all packages found at or below a specified directory, it is sometimes more efficient to build a specified package and (recursively) any of its prerequisites. The following command builds the package in the specified directory *and* any prerequisite packages declared by the "`requires`" statement in the package's specification file (i.e., `package.xdc`).

```
xdc -PD examples/basic/vers/app
```

By default, if an error occurs during the build of any package the `xdc` command will terminate the build and not attempt to build other packages. While this is convenient during interactive builds, during an overnight build of many packages it is preferable to continue building as much as possible. In the morning, one can correct the errors and re-execute the command. Only goals that failed to build or those that depend on the fixed files will be rebuilt.

To prevent the `xdc` command from stopping on the first error in the example above, use the `-k` option.

```
xdc -k -PR examples
```

Running Tests in Multiple Packages

In addition to building multiple packages at a time, it is also valuable to run all tests for multiple packages using a single command. A regression test suite can be structured as a collection of packages, for example. The following command runs all tests for the packages `pkg1`, `pkg2`, and `pkg3` located in the current working directory.

```
xdc test -P pkg1 pkg2 pkg3
```

Since tests have a tendency to fail (otherwise they are not good tests), it is often valuable to continue running all tests even if a test fails. This is especially true when running over-night regression test suites containing hundreds of tests. The following command runs all tests and continues even if one or more tests fail.

```
xdc -k test -P pkg1 pkg2 pkg3
```

Building and Running for a Particular Target

In the examples above, we built and ran executables for all targets. Recall a target defines a CPU ISA and a compiler runtime model (big endian, little endian, near, far, etc.). Since packages often need to support multiple targets, it is often desirable to restrict a build or a test to a particular target. The `xdc` command supports target specific versions of

the build goals `all`, `clean`, `test`, `.libraries`, and `.executables`.

The following command runs only the tests for the target whose suffix is "`62`" for all packages in the `examples` directory and tries to run each test even if a test fails.

```
xdc -k test,62 -P examples/*
```

The following command removes all generated files related to the target whose suffix is "`62`" for all packages in the `examples` directory.

```
xdc clean,62 -P examples/*
```

The following command builds all generated files related to the target whose suffix is "`62`" for all packages in the `examples` directory.

```
xdc all,62 -P examples/*
```

Environment Variables

In addition to the command line options, the `xdc` command also uses the following environment variables to control its behavior. Except for `XDCPATH`, `XDCARGS`, and `XDCBUILDCFG`, no environment variable changes the contents of any goal produced by `xdc`; the results of a build are unaffected by environment variables (unless a user-specified tool invoked by `xdc` is affected by an environment variable).

`XDCARGS` This variable names arguments that are passed to the package's build script, package.bld. The package's build script references the arguments from the global array `arguments`. For example, the command

```
xdc XDCARGS="foo bar"
```

causes the `arguments` array (in the package build script) to be initialized as follows:

```
arguments[0] = "foo";
arguments[1] = "bar";
```

`XDCBUILDCFG` if defined *and* the file "`./config.bld`" does not exist, this variable names a file that will be used in-lieu of the `config.bld` file found along the "import" path (i.e., "`.;$(XDCPATH);`***`xdcroot`***`;`***`xdcroot`***`/etc`") to configure the build environment prior to running a build script.

However, if XDCBUILDCFG is specified in the command line, any package specific "./config.bld" will be ignored. Thus, with respect to specifying which config.bld to use, the package has precedence over the environment variable but the command line has precedence over the package.

Why distinguish between setting XDCBUILDCFG on the command line verses setting it in the environment? Some packages need to override the setting of XDCBUILDCFG; e.g., in order to clean and rebuild a package of targets (which may be referenced by the file named by the environment variable XDCBUILDCFG), the package may define a "local" config.bld that does not reference *any* targets.

XDCOPTIONS — a string of options that affect the messages displayed by xdc while it runs. Only three options are currently supported: "v", "q", and "t".

If this string contains "-v" or "v", each command executed by xdc is displayed before execution. This makes it easy to create "shell scripts" that re-create the build without the need for the xdc command or make.

If this string contains "-q" or "q", banners normally displayed during multi-package builds are not displayed. Since the banners contain date and time information, this option is useful when the output from the xdc command must not vary between successive builds; e.g., when running regression tests.

If this string contains "-t" or "t", banners are displayed during multi-package builds but no dates or times are displayed. This option is useful when running regression tests on a fixed set of packages; when an error occurs, the banners make it easy to tell which package(s) failed.

XDCPATH — a string of ';' separated directories that contain packages. This path is used to locate packages that are used by the package being built.

It is usually a mistake to put a relative path in the XDCPATH environment variable. Relative paths in XDCPATH reference directories relative to the package being built rather than the directory where the xdc

command was invoked. Thus, a relative path will refer to a different repository for each package being built.

It is possible, however, to use the '^'character in the XDCPATH definition to refer to the current package's repository. So, if you have a repository that is always in a fixed location relative to all of your packages repositories, it is possible to create a single XDCPATH setting that does not include any absolute paths. Suppose, for example, that your build system places all prerequisite packages in an "imports" repository prior to building the packages in a "src" repository and the imports and src repositories are sibling directories in the file system. The following XDCPATH setting is sufficient to build all packages in the src repository.

```
set XDCPATH=^/../imports
```

Note that multiple versions of the same package can appear along the XDCPATH. The package path can name multiple "package repositories" which can contain a package directory with the same name. When searching for a package, the first repository that contains a directory matching the package's name will be used. Thus, even if two packages with the same name appear in the package path, only one will ever be found; i.e., the first one in the order specified in the package path.

XDCTARGETS — a string of white space separated target names that name all supported build targets. Each name is interpreted as a regular expression and is used to select from the set of all available targets included in the build "startup" script (see config.bld in the Files section below). This environment variable can be used to re-build packages with a subset of the available targets. The current set of targets include the following:

```
ti.targets.C54, ti.targets.C54_far

ti.targets.C55, ti.targets.C55_large

ti.targets.C28_large

ti.targets.C62

ti.targets.C62_big_endian

ti.targets.C64
```

`ti.targets.C64_big_endian`

If a specified target name does not match any available target, the prefix "`ti.targets.`" is added and the match is retried. If no match occurs, a warning is displayed and processing continues uninterrupted. Thus, the target "`ti.targets.C62`" may be abbreviated to just "`C62`"

Any change to an environment variable that may affect the results of the build will trigger a rebuild of the goals that may be affected (as well as some that may not be affected).

Note that these environment variables may be specified on the `xdc` command line. In this case, the value specified on the command overrides any value in the environment. For example, the following command causes the package in the current working directory to be built for just the `C62` and `C54` targets.

```
xdc XDCTARGETS="C62 C54"
```

Exit Status

The exit status of the `xdc` command is the exit status of the underlying `make` command whenever `make` is executed; otherwise, the following exit values are returned:

0 Successful completion.

1 An error occurred.

Files

`config.bld`

The build model "startup" script; this script, located along the import path "`.;$(XDCPATH);`***`xdcroot`***`;`***`xdcroot`***`/etc`", configures the build model's modules so that common settings can be shared among multiple package build scripts. It is possible to override this behavior using the `XDCBUILDCFG` environment variable.

`package.bld`

The package's build script; this script, located in the package's working directory, specifies all of the physical files (libraries, executables, etc.) that are part of the package.

`.xdcenv.mak`

This file is a generated file that captures the environment setting that can affect the contents of the generated makefile; changes to this file trigger a re-build of the makefile.

See Also

http://www.gnu.org/software/make/manual/html_mono/make.html