CoreLinux++ Development

The Corelinux Consortium

Revision: 1.10
Created on June 04, 2000
Revised on October 7, 2000

Abstract

This document describes how to build applications with CoreLinux++ Standards. This is the CoreLinux++ Development Guide.

Contents

Copyright notice

`CoreLinux++ Copyright © 1999, 2000 CoreLinux Consortium`
`Revision 1.10 Last Modified: October 7, 2000`

This material may be distributed only subject to the terms and conditions set forth in the Open Publication License.

1 Introduction

This guide is broken into two (2) parts: The first is meant to familiarize the developer with the standards, guidelines, and build process for the CoreLinux++ class libraries, frameworks, and applications, and the second providing a insiders view into the workings of the class library.

If you are ***just*** interested in using the binaries, refer to the Developing with CoreLinux++ section. For CoreLinux++ developers, before attempting to build any of the CoreLinux++ code refer to the Setting Up section. It is probably a good idea after that to reference the Standard Development Process.

To assist all developers the following references are available:

- CoreLinux C++ Standards and Guidelines
- CoreLinux Object Oriented Analysis and Design
- CoreLinux Functional Requirements
- CoreLinux Class Reference

For all questions or bug reports please go to the CoreLinux++ homepage. and then to the project links. There you will find forums for comments or discussions and also the ability to enter defect reports. If you are really in a jam, you can e-mail `frankc_at_users.sourceforge.net`.

2 Setting Up for CoreLinux++ builds

2.1 Getting Things Rolling

The configuration, building, and installation of the CoreLinux++ libraries is now handled by `automake` [FSF, 2000a], `autoconf` [FSF, 1999], and `libtool` [FSF, 2000b]. Please read the INSTALL file for instructions on how to get started with the system.

2.2 Make Environment

With the new build system, the libcorelinux++ header files can be found in *install-path*`/corelinux`. This means that you will need to prefix includes:

```
#include <corelinux/Common.hpp>
```

for example.

2.3 Setting up for CoreLinux++ Execution

During development the libcorelinux++ and/or libcoreframework++ shared libraries are put into *install-path*`/lib` when you '`make install`'.

2.4 Directory Structure

This section is geared mostly to developers and maintainers who are working on `libcorelinux++` or `libcoreframework++`.

The directory structure for libcorelinux++ source is very straight forward, assuming you have checked out the CVS corelinux module:

```
\corelinux                ; The root corelinux directory
    \corelinux            ; Root include directory (libcorelinux++ includes)
        \coreframework    ; Includes for libcoreframework++
    \src                  ; Root source directory
        \classlib         ; Class Library project source
            \corelinux    ; libcorelinux++ source(s)
        \testdrivers      ; Test driver project source(s)
            \ex1          ; Various test drivers
            \ex[n]
```

where the contents of each directory will be discussed.

2.4.1 The Root Directory

The `Corelinux++` root directory is the canonical tip for the development libraries. Besides the directories that follow this description, it is the home of the configuration and build macros. It is recommended that you do your development work with debug versions of the libraries, and testing with both debug and optimized code.

We have added a few macros which help setup for these modes. You need to specify your compiler using the `--with-cxx` assignment. Here are examples using the GNU compiler:

```
./configure --with-cxx=g++ --enable-debug
./configure --with-cxx=g++ --enable-optimization
```

use

```
./configure --help
```

for other options.

2.4.2 The Root Include

The CoreLinux++ root include directory behaves in a similar fashion to `/usr/include`. The base directory contains headers that are libcorelinux++ classes and macros. When developing libcorelinux++ or libcoreframework++, the includes should resolve to here. When developing applications to a specific libcorelinux++ and/or libcoreframework++, the include tree will most likely reside in `/usr/include/corelinux`.

2.4.3 The Root Source Root

All source code for CoreLinux++ projects are located from this point in the development tree.

2.4.4 Class Library Source Root

Source code, the result of which are Class Libraries, are located from this point in the development tree.

2.4.5 CoreLinux Class Library Source

Source code specific to libcorelinux++, the result of which is a Class Libraries, is located here.

2.4.6 Additional Class Library Source

Other support or free form class library source project(s).

2.4.7 The Test Driver and Example Code Root

Source code, the result of which are applications which exercise or validate the class libraries or framework libraries are located from this point in the development tree.

2.4.8 Example application source

Source code for the examples supplied as part of the default CoreLinux++ hierarchy.

2.4.9 CoreLinux++ Documentation for Standards, Class Reference, Requirements, Analysis, and Design

Most of the documentation is in HTML format. You have found it because you are reading this!

3 Standard Development Process

Refer to From Design to Implementation

4 Developing with CoreLinux++

This section is to include the steps taken by developers who are not interested in contributing to the library itself.

4.1 Using autoconf

For the developer who wants to check `Corelinux++` existence using `autoconf` and its `m4` macro `AC_CHECK_LIB`, here is the code to add to the configure.in:

```
#
# check corelinux presence
#
AC_CHECK_LIB(cl++, ACCheckCoreLinux,[
INCLUDES="${INCLUDES} -I/usr/include/corelinux"
LIBS="${LIBS} -lcl++"
],[
 echo "You need to install corelinux. see http://corelinux.sourceforge.net"
 exit;
],)
```

4.2 Macros

Two macros are used heavily in `Corelinux++`. They are macros to provide a uniform way to declare types and classes. Don't be shy at using them and what they provide.

- `DECLARE_TYPE` declares a new type as in code 4.1

```
    #include <corelinux/Common.hpp>

    // declares newType as type Type
    DECLARE_TYPE( Type, newType );

    // it will be expanded by the C preprocessor
    // as follows:
    typedef Type             newType;
    typedef newType*         newTypePtr;
    typedef const newType*   newTypeCptr;
    typedef newType&       newTypeRef;
    typedef const newType& newTypeCref;

    // for example
    DECLARE_TYPE( char, Char );

    // will define the types Char,
    // CharPtr, CharCptr, CharRef and CharCref
```

Code 4.1: `DECLARE_TYPE` macro

- `DECLARE_CLASS` declares a new class as in code 4.2 this macro is expecially used for forward

```
    #include <corelinux/Common.hpp>

    // forward declaration of class foo
    DECLARE_CLASS( foo );

    // it will be expanded as follows:
    class   foo;
    typedef foo*        fooPtr;
    typedef const foo*  fooCptr;
    typedef foo&        fooRef;
    typedef const foo&  fooCref;
```

Code 4.2: `DECLARE_CLASS` macro

class declaration or to ensure the declaration of the associated pointer and reference types.

Corelinux++ uses STL containers and algorithm. In order to provide the same services DECLARE_TYPE and DECLARE_CLASS, several macros are provided :

- CORELINUX_LIST defines a STL std::list as in code 4.3

```
#include <corelinux/List.hpp>
CORELINUX_LIST( int, ListIntType );

// will declare:
typedef std::list<int>  ListIntType;

// and as always, ListIntTypePtr, ListIntTypeRef
// ListIntTypeCptr, ListIntTypeCref will be also
// defined
```

Code 4.3: CORELINUX_LIST macro

- CORELINUX_VECTOR defines a STL std::vector as in code 4.4

```
#include <corelinux/Vector.hpp>
CORELINUX_VECTOR( int, VectorIntType );

// will declare:
typedef std::vector<int>  VectorIntType;

// and as always, VectorIntTypePtr, VectorIntTypeRef
// VectorIntTypeCptr, VectorIntTypeCref will be also
// defined
```

Code 4.4: CORELINUX_VECTOR macro

- CORELINUX_STACK defines a STL std::stack as in code 4.5

```
#include <CORELINUX/Stack.hpp>

CORELINUX_STACk( int, StackIntType );

// will declare:
typedef std::stack<int>  StackIntType;

// and as always, StackIntTypePtr, StackIntTypeRef
// StackIntTypeCptr, StackIntTypeCref will be also
// defined
```

Code 4.5: CORELINUX_STACK macro

- CORELINUX_QUEUE defines a STL std::dequeue as in code 4.6 on the next page
- CORELINUX_SET defines a STL std::set as in code 4.7 on the following page

```
#include <CORELINUX/Queue.hpp>

CORELINUX_QUEUE( int, QueueIntType );

// will declare:
typedef std::dequeue<int>  QueueIntType;

// and as always, QueueIntTypePtr, QueueIntTypeRef
// QueueIntTypeCptr, QueueIntTypeCref will be also
// defined
```

Code 4.6: `CORELINUX_QUEUE` macro

```
#include <CORELINUX/Set.hpp>

CORELINUX_SET( int, less<int>, SetIntType );

// will declare:
typedef std::set<int, less<int> >  SetIntType;

// and as always, SetIntTypePtr, SetIntTypeRef
// SetIntTypeCptr, SetIntTypeCref will be also
// defined
```

Code 4.7: `CORELINUX_SET` macro

```
#include <CORELINUX/Set.hpp>

CORELINUX_MULTISET( int, less<int>, SetIntType );

// will declare:
typedef std::multiset<int, less<int> >  MultiMultiSetIntType;

// and as always, MultiSetIntTypePtr, MultiSetIntTypeRef
// MultiSetIntTypeCptr, MultiSetIntTypeCref will be also
// defined
```

Code 4.8: `CORELINUX_MULTISET` macro

- `CORELINUX_MULTISET` defines a STL `std::multiset` as in code 4.8 on the page before
- `CORELINUX_MAP` defines a STL `std::map` as in code 4.9

```
#include <CORELINUX/Map.hpp>

CORELINUX_MAP( std::string,
               std::string,
               less<string>,
               MetStringType );

// will declare:
typedef std::map<string,
                 string,
                 less<string> >  MapStringType;

// and as always, MapStringTypePtr, MapStringTypeRef
// MapStringTypeCptr, MapStringTypeCref will be also
// defined
```

Code 4.9: `CORELINUX_MAP` macro

- `CORELINUX_MULTIMAP` defines a STL `std::multimap` as in code 4.10

```
#include <CORELINUX/Map.hpp>

CORELINUX_MULTIMAP( string,
                    string,
                    less<string>,
                    MultiMapStringType );

// will declare:
typedef std::multimap<string,
                      string,
                      less<string> >  MultiMapStringType;

// and as always, MultiMapStringTypePtr, MultiMapStringTypeRef
// MultiMapStringTypeCptr, MultiMapStringTypeCref will be also
// defined
```

Code 4.10: `CORELINUX_MULTIMAP` macro

Note that for all these macro, except `CORELINUX_STACK`, the `Iterator` type is defined also, see for example the code 4.11 on the following page

5 Class Library Internals

The Class Library Internals is your guide into the inner workings of libcorelinux++. This is a useful section for those interested in how it works, or how they can contribute to make it better.

```
    // if we define VectorType as follows
    CORELINUX_VECTOR( int, VectorType );

    // then CORELINUX_VECTOR macros will define
    // also the following Iterator types
    typedef VectorType::iterator VectorTypeIterator;
    typedef VectorType::iterator& VectorTypeIteratorRef;
    typedef VectorType::iterator* VectorTypeIteratorPtr;
    typedef VectorType::const_iterator VectorTypeConstIterator;
    typedef VectorType::const_iterator& VectorTypeConstIteratorRef;
    typedef VectorType::const_iterator* VectorTypeConstIteratorPtr;
```

Code 4.11: `Iterator` declaration

5.1 Foundation Classes

This section covers the foundational classes of the class library.

5.2 Inter-Process Communication

Every library should have them, and we do. This section covers the various inter-process communication types of the class library.

5.2.1 Semaphore and SemaphoreGroup

Semaphores and SemaphoreGroups encapsulate the use of the semaphore IPC functions, and adds features useful to real world applications. At the very least you will be able to use Semaphores and SemaphoreGroups as you would with Linux semaphore groups.

5.2.2 Threads

5.3 Design Patterns

6 Frameworks

A framework is defined as a facilitating backbone which combines sets of related components for a specific purpose or domain. A distinguishing feature of a framework is that it defines a generic design that supports a bi-directional flow of control between the application and itself.

6.1 Framework Support

CoreLinux++ enables framework development via the libclfw++ library. Included in the library are common objects that can be re-used by many of the framework implementations, in addition to abstractions of frameworks that have been discovered, much like design patterns, to be common across domain application design.

The framework library (libclfw) provides a number of support objects and framework abstractions as defined below.

6.2 Meta-class MetaType

Meta-classes were originally introduced for languages such as CLOS and Smalltalk to assist in the construction of instances whose layout was entirely defined at run-time. More efficient languages, such as C++ or Eiffel, define object layouts at compile-time and compiler writers have no need to provide meta-classes. The available facilities in C++ are limited but they do exist.

Therefore a meta-class instance is used to capture (model) the type information about a class, and subsequently, it's instances. For example, we can consider that "aaaaa" is an instance of a ascii string, and that string is of type MetaTypeAsciiString. Within MetaTypeAsciiString we can convey information about constraints (upper and lower bound, each character is 8 bits, must terminate with null, etc.) as well as it's relationship to other types. In the example, MetaTypeAsciiString can be modeled as a child of MetaTypeString.

6.3 MetaType Macros

The following macros, and their use, are detailed below:

6.4 Ontology

A ontology is an explicit specification of some topic. It is a formal and declarative representation which includes the vocabulary (or names) for referring to terms in that subject area and the logical statements that describe what the terms are, how they are related to each other, and how they can or cannot be related to each other. Ontologies therefore provide a vocabulary for representing and communicating knowledge about some topic and a set of relationships that hold among the terms in that vocabulary.

The CoreLinux++ support for ontologies are in the form of MetaType object declarations and definitions joined through associations into a hierarchy, or tree. It is through access to this tree that components or applications can reason with the class types involved in the ontology.

6.5 MetaType Ontology

The base framework library (libclfw++) contains a number of types that, via the MetaType macros, forms a “starter kit” type ontology. The types defined include:

6.6 Common franework abstractions

The following are the abstract frameworks that come with the libclfw library.

6.6.1 Library Load

Whether managing graphic libraries, supporting plug-ins for extension, or dynamically calling shared library functions, applications often have a need to load information at run-time through the support of some library or operating system facility. The Library Load Framework supplies the extensible object types and behvioral semantics for such activities.

6.7 Schemas And Persistence

6.7.1 Premise

The persistence abstraction is built on the premise that the actual storage service code, that which implements concrete persist services beneath the abstraction, should be intelligent enough to take guidance from domain descriptions (concepts). Said concepts define the types to be stored, in addition to other information, and it is the job of the storage service to manage the data layout and service execution as defined by it's implementation.

It is, after all, the 21st century.

This benefits the application developer from having to worry about the details of a particular storage service, as well as having the flexibility to change persist implementations at will, or by user choice.

6.7.2 Overview

There are actually two (2) aspects to the CoreLinux++ Abstract Persist framework: Schemas and Services. This release focuses on the newly provided Schema constructs. * Please refer to the current implementation restrictions at the end of this document.

A Schema is an organization of concepts to model an idea. In this case, what is being modeled are the types that the application will want to have made persistent. When a schema has been modeled, it is then usable as a roadmap, in effect, to guide the storage implementation. For example, the types may represent tables in a relational database storage service, or a file type for a flat file service. It is ultimatley up to the service to decide how to "plan the trip".

There are two (2) workflows around Schemas and their concepts: Model and Run. Lets talk turkey first:

6.7.3 Schema Modeling

Schema modeling is supported through built-in services and functions of the library. All modeling changes are persisted to gdbm databases, and the domain modeler need only focus on making their schema correct. While there is a tremendous opportunity for adventerous open source developers to build tools around this (GUI, etc.), the library contains the fundemental capabilities to Create, Request, Update, and Delete Schemas or their artifacts (concepts).

The example test driver (clfw/src/testdrivers/exf2) shows some of the following capabilities.

1. Typically, start by resolving the SchemaSponsor and creating an instance of it.
2. From the Sponser, get the Catalog object instance.
3. Create, Edit, Save, Close, Delete Schemas using the interface of the Catalog instance.

4. Create Concepts and add to Schema
5. Remove Concepts from Schema and delete

6.7.4 Schema Run Time

Once a Schema has been modeled, it can be used by a storage service to support the input and output operations of application objects.

1. Like Schema modeling, the activation of persistence in an application starts with the resolving and instance creation of a StoreSponser. This can be done be either using the the StoreSponser class name and resolving through the Ontology interfaces.

For example:

StoreSponsorPtr aSponsor (StoreSponsor::castDown(getInstanceOf("MySQLSponsor")));

or by walking the StoreSponser MetaClass hierarchy and testing each of the children nodes until you find the one you want. In latter releases we will have support for descriptive attributes on MetaClasses to help reason with sponser types and capabilities.

Another way to load a Sponser would be to work in conjunction with plug-in support to dynamically load types that are not part of the library compilation. Sounds like a job for the CoreLinux++ Library Load capability!

Note: In the above code example it presumes that someone has provided a MySQL persistence implementation.

2. From the Sponser, get the Catalog object instance.
3. Create a Store (which would require identifying the Schema to be used as the roadmap).
4. Fetch a Store to write and read application object instances from and to respectivley.

6.7.5 Current Restrictions

Name/Value Declarations

While ultimately Schemas and Concepts attributes won't be limited as to the depth or types supported, the current implementation assumes most attributes are of the form: Name:Value, where:

Name is the Key data member of the Attribute and should be of type FrameworkString

Value is the Value data member of the Attribute and should be of type FrameworkString

Primary Schema Declarations

For schemas, the following are the recognized attribute keys:

```
Attribute Key   : "Name"
Expected Value  : Unique name for this schema
Example : "Franks Schema"
Note : This is a required attribute

Attribute Key   : "Collection"
Expected Value  : The collection type literal (currently "Array" or "SetCollection")
Note  : SetCollection is safer for insuring unique name keys, although as new collection types
Note : This is a required attribute

Attribute Key   : "Location"
Expected Value  : Qualified path where schema should be stored
Note  : This is an optional attribute, default is ~/.clfw++/schemas/"Name"
```

```
Attribute Key   : "GUID"
Expected Value  : Stringified UniversalIdentifier to be used
Note : This is an optional attribute, default is the system will generate.
```

with the following attribute expected to be supported in later releases:

```
Attribute Key   : "SchemaClass"
Expected Value  : The Schema class type literal to be used when instantiating a new schema, thi
Example : "FranksWickedSchemaDerivation"
```

Primary Concept Declarations

```
Attribute Key   : "Name"
Expected Value  : Unique name for this schema
Example : "Concept A1"
Note : Even though concepts can be created for other reasons, this is a required attribute if a

Attribute Key   : "Class"
Expected Value  : Stringified UniversalIdentifier of the type that this Concept defines represe
Note : Even though concepts can be created for other reasons, this is a required attribute if a
```

with the following attributes expected to be supported in later releases:

```
Attribute Key  : "Uses"
Expected Value : Collection type, where the collection contains attributes that describe data f
Attribute Key  : "Data Member Name"
Expected Value : "Concept Name"

where:

"Data Member Name" is the name of the data member that has been
explicitly identified in the MetaClass definition of the entity
being stored.

"Concept Name" is the name of another concept in the Schema
(which one would have to presume is the same type of the Data
Member).

Attribute Key  : "Ignore"
Expected Value : Collection type, where the collection contains
single string values that identify (by name) the Data Members of
the Class that the storage management should ignore. In
otherword don't write or expect to read from persist.

Attribute Key : "Fence"
Expected Value : Collection type, where the collection contains
single string values that identify (by name) the Data Members of
the Class that the storage manager should recognize as an indicator
```

```
that the application whats instances of this data member to be handled
as a separate table/file/etc. according to it's (storage service)
domain.
```

Please note that these would be considered ”suggestions” that the declaration of which does not imply that a storage service would do anything with the information.

References

FSF. *GNU Autoconf Manual*. FSF, 2.13 edition, 1999.

FSF. *GNU Automake Manual*. FSF, 1.4 edition, 2000a.

FSF. *GNU Libtool Manual*. FSF, 1.3.4 edition, 2000b.