Dharma Software Development Kit

Documentation Set

Interactive Objects, Inc.

November 2001

Author: John Locke

Table of Contents

Introduction

The Dharma™ SDK (also referred to as Dharma™ Application Stack or simply Dharma™) provides a diverse and well defined set of APIs and components necessary for commercial digital audio playback. This release of the Dharma OS, as an external SDK for external parties to develop with, is the first source release of Dharma in any form, and, naturally, is our best. Dharma is a "living" system, in that we are constantly adding new features and functionality into the core system, so this SDK is a snapshot of Dharma that has been polished for release.

Documentation structure

The full documentation is available both online and in PDF format. The documentation is divided into the following areas:

- Architectural Overview – A description of the Dharma Operating System.
- Player Tutorial – How to use the Dharma SDK to develop a media player.
- Dharma Board documentation – Instructions for using the Dharma board, including downloading and image creation, hardware specifications, and data sheets.
- API Reference – Reference for each class in the Dharma SDK.

Documentation conventions

Filenames are written in UNIX form and are relative to the "sdk" folder. So if the documentation refers to a file `./player/util/registry/include/Registry.h` and you installed in c:\dadio, the full unix path would be: `/dadio/sdk/player/util/registry/include/Registry.h` and the DOS path might be: `c:\dadio\sdk\player\util\registry\include\Registry.h`.

Resources are defined as modules that a given component relies on.

Glossary

Cygwin

A UNIX environment for Windows. The free Cygwin package is on the Dharma SDK CD-ROM, and includes the `bash` shell and several UNIX tools.

Dadio™

Obsolete. The former name of the software application stack included with Dharma.

DCL

Dharma Configuration Language. A custom mini-language for generating different options in makefiles used to build Dharma-based images.

Dharma™

A hardware board used for developing Dharma objects, and the software application stack provided in this SDK for jump-starting your development. The hardware consists of a motherboard, CPU, and many peripherals that may be plugged in or removed to test various configurations. The software consists of a set of embedded objects written in C++ that are designed for the Cirrus 7312, 7409, and 9312 chips.

eCos™

"Embedded Configurable Operating System." An open-source real-time operating system used by Dharma. Dharma does not include the source for eCos – instead, it provides object code for three different configurations: minimum RAM configuration, network-enabled, and USB-enabled. If these configurations do not meet your needs, contact Interactive Objects.

PCM

Pulse Code Modulation. A digital scheme for transmitting analog data.

Symbolic Links

Similar to Windows shortcuts, symbolic links are a way of having one file or directory appear in multiple places in the file system. A symbolic link is sort of a soft link that points to a file actually located somewhere else. The convenience of this is that you can have a single copy of a file, but use it from several locations (when compiling, for example).

Quick Start Guide

This section describes how to quickly get the Dharma Board up and running.

System Requirements

Before starting, make sure you have the minimum system requirements listed below:

PII-class processor (or better)

Windows 2000 or later

At least 600 MB of free disk space

Available serial (COM) port

The SDK installs the full Cygwin UNIX environment for Windows 2000, version 1.3.2, and uses several packages bundled into the SDK installer. **Interactive Objects strongly recommends that you install the package on the CD-ROM without modifying any of the options.** The SDK has only been tested with the bundled version of Cygwin – newer versions may not work correctly.

Documentation Overview

There are two versions of the documentation: printed and online. All of the SDK documentation is available in both forms.

The online documentation is installed on your system when you run the installer, and is accessible from your Start -> Programs menu.

The printed documentation is in the Documentation directory on the CD-ROM.

The readme.txt file contains this page only, in the root directory. The QuickStart.pdf file contains this page and the Operation Guide which describes building an image and transferring the image to the device.

Hardware Setup

Setting up the hardware is easy. Follow the steps below:

1. Carefully unpack the Dharma board and put it on a flat surface.
2. Connect a null-modem cable between the serial port on the device and an available COM port on your computer.
3. Connect the power supply to the board.

Software Setup

The software setup is similarly simple:

1. Read the ReleaseNotes.txt file on the CD-ROM for specific information about this release.
2. Double-click the **IObjects-sdk-current.exe** on the CD-ROM.
3. When prompted to install Cygwin, click **Yes**.
4. When given the choice of where to install Cygwin from, click **Install from Local Directory**.
5. You can specify different locations for the install, and where to put shortcuts, but otherwise, accept all of the other default settings.

You now have the Cygwin environment set up, and the SDK installed. The next step is to build an image, and download it to the device. See the Dharma Operation Guide on page 72 to continue.

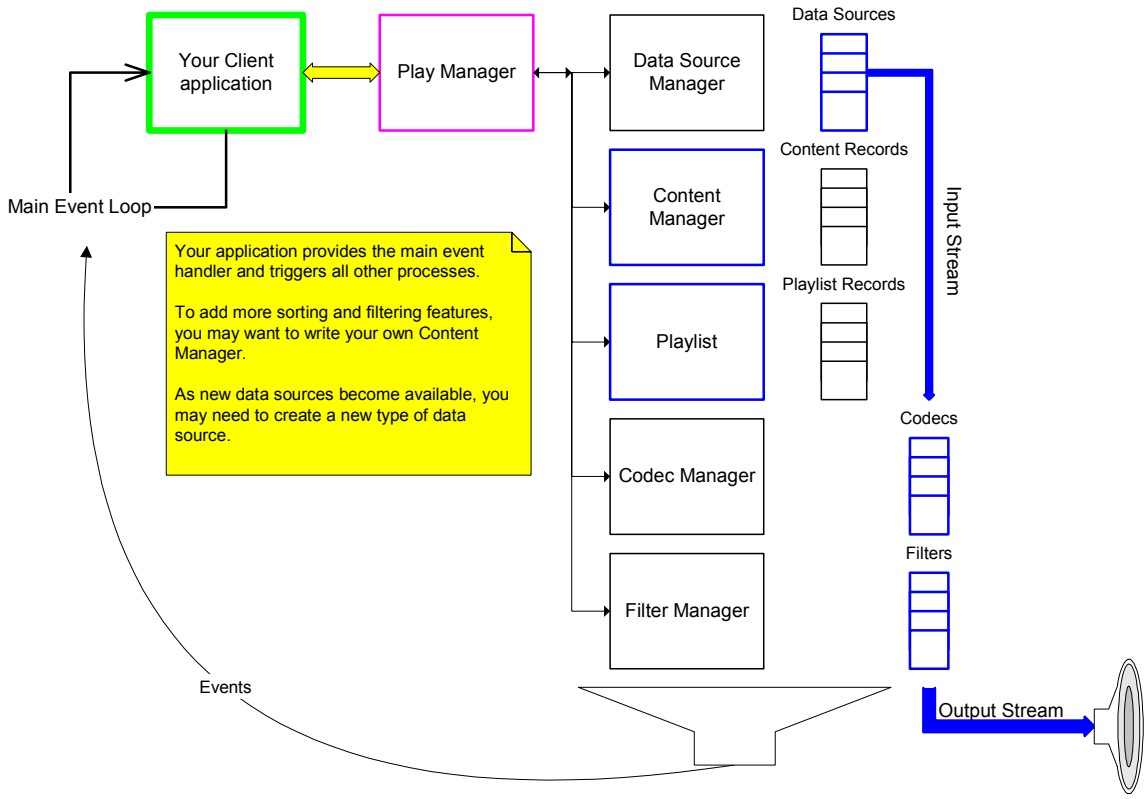
Architectural Overview

The Dharma SDK provides an object-oriented framework to help you quickly build the firmware for your media player.

This document describes the architecture of the objects provided as part of the software development kit, and gives you what you need to know to jump-start your development.

The Dharma SDK comes with working code with basic functionality. You will want to build on the set of features to make the device dance.

At the highest level, the client application controls the user interface, including the display, buttons, and connections to devices and media. The Dharma SDK includes the low-level drivers for many devices, but your client application must define which ones are being used. The main event handling routines are all in your client code. Your client code interacts almost exclusively with the Play Manager, which routes requests to other appropriate modules:



The SDK provides several abstract classes. You may extend these classes as necessary to add functionality to the device. These abstract classes include: Content Manager, Playlist, Metadata, Data Source, Data Streams, Codec, and Filter.

Event Loop

The first task for your new application will be to create an event loop. The example event loop simply grabs the oldest event off the queue and passes to the default event handler in the Play Manager.

Events are triggered by all of the user interface elements: buttons, switches, jog dials, media insertion/removal, etc. In addition, the SDK objects generate events such as:

- Media Player track progress
- Content Enumeration information
- Errors.

Your main event loop can pass all events to the default event handler in the Play Manager, but you will probably want to intercept certain events and add your own handler.

User Interface

The demo applications provide examples of two user interfaces: a serial text-based interface, and a simple bit-mapped interface. To make a more sophisticated interface, you might want to use a GUI toolkit, such as PEG, Eyelet, or MicroWindows. These are available from individual vendors.

Main Application

Your main application will mostly consist of handlers for the various events. Let's take a closer look at each type of event, and how you can use the SDK to perform the necessary tasks. Here's a list of the basic types of activities the SDK supports:

- Power on/initialization
- Adding/removing media
- Content enumeration
- Adding/removing/reordering content into playlists
- Starting/stopping/rewinding/skipping tracks on the media player
- Applying/adjusting filters
- Storing and retrieving playlists and state.

The objects in the SDK, for the most part, track their state internally.

We will now walk through each of these scenarios.

Power on and initialization

Your main application will need to create a few objects before using the Play Manager. The main objects you need to create to pass into the Play Manager are the Data Sources, the Playlist, and the Content Manager. All available codecs and filters are registered at startup. The SDK includes Data Source objects for the following types of data sources:

- CD-ROM (ISO 9660)
- FAT Filesystem – hard drives, compact flash, etc.

- Network data – an Internet or LAN server for streaming media.

Interactive Objects has data source objects for Data Play and Clik media in development. You can define and write other data source objects for new types of media.

For each actual device, you must instantiate a corresponding data source object. Your application must call the `CPlayManager::AddDataSource()` method, passing a Data Source object for each data source available to the system.

You must also pass a Content Manager object to the Play Manager using the `CPlayManager::SetContentManager()` method, and a Playlist using the `CPlayManager::SetPlaylist()` method. The SDK provides two Content Manager objects:

1. `CSimpleContentManager`: A content manager that isn't. This example content manager returns the full content list on all queries.
2. `CMetakitContentManager`: A more complex content manager that uses the open-source Metakit database to manage each content record. This Content Manager can return a subset of titles by artist, genre, or album.

You may want to create your own Content Manager to add more sophisticated features. The Play Manager uses the pointer you pass in to access the playlist during playback. The SDK provides a single Playlist class, `CSimplePlaylist`.

Optional initialization tasks include refreshing the content list from the data sources, or restoring state of your content manager objects and playlists.

Adding or removing media

The default event handler notifies the Play Manager when the user adds or removes media from a data source. The `CPlayManager::NotifyMediaRemoved()` method stops the track playing on the media player if its data source is the removed media, and then removes all of the content records for that data source from the Content Manager. The `CPlayManager::NotifyMediaInserted()` method automatically tells the Data Source Manager to enumerate all of the tracks on the new media, sending data for each track asynchronously to the Event Queue.

Content enumeration

Content Enumeration is the process of creating a list of tracks on all available media, and optionally, collecting metadata for those tracks. There are three modes of content enumeration, differing on when and what metadata is retrieved:

- Single-pass

- Single-pass with metadata
- Double-pass.

Single-pass enumeration collects only a list of tracks on each media, with no metadata. Single-pass with metadata collects all available metadata for all tracks in a single pass. To do this, however, each track must be opened and processed by a codec.

Double-pass enumeration separates metadata collection from the track enumeration. The second pass (the metadata pass) can then only be performed for tracks that do not already have metadata available. For example, if you have restored the state of the Content Manager, its media content records may already have metadata for the tracks, but if there are new tracks on the media, the Content Manager can send the tracks for which it needs metadata to the second pass, and skip the tracks it already knows about.

The content enumeration mode can be different for each data source. Each data source can specify a default content enumeration mode, as well as a default update chunk size. Your application can set these default values.

The `CPlayManager::RefreshAllContent()` method calls each Data Source object to start enumerating its content. Each data source sends the list of tracks in batches to the event queue. The default event handler will send these content updates to the Content Manager.

In each of these modes, the Data Source Manager awakens its own thread to handle the actual content enumeration and returns. All data is sent to the event queue as it is processed.

In the single pass mode, the Data Source Manager cycles through each data source, collecting the file name for each available track and sending it to the Event Queue. This type of content refresh is good for small FAT data sources, such as Compact Flash, where you may not need the metadata and you just want the player to play a pre-set sequence.

In the single-pass with metadata mode, the Data Source Manager goes through each data source, creates a Codec object for each file, and runs each file through the Codec to extract the metadata from the file. This metadata generally includes information such as Artist, Album, and Song title, but may also include such details as the length of the track, etc. This type of content refresh is good for data sources that take a long time to seek to different tracks, such as a CD player.

In the double-pass mode, the Data Source Manager performs a single pass as above to retrieve the filesystem information for each track. These files are sent to the Event Queue, and if you use the default handler in the Play Manager, the file names are passed into the Content Manager. Any files that do not have metadata already associated with them are then sent to the Data Source Manager for the second pass. The second pass opens a Codec object for each file name and uses it to extract the metadata as above, passing it to the event queue. This type of content refresh is good for when you have stored the state of the Content Manager (see Storing and retrieving state on page 19), and may only need to update the metadata for specific tracks. It can also be used to periodically check for new content.

Each Data Source has a default mode of content refreshing, that will be used if no mode is specified. You can, of course, over-ride the mode by supplying one.

See the Data Sources topic on page 21 for more.

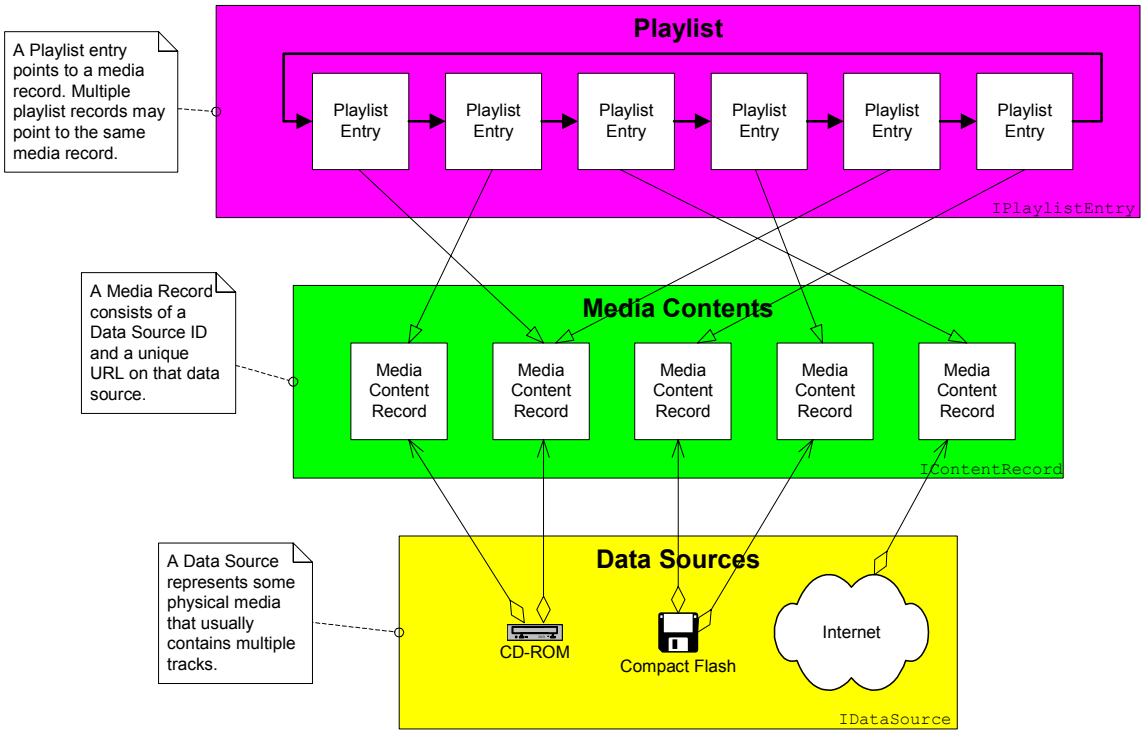
Working with Playlists

Playlists are defined as separate objects, but they are tightly integrated with the Content Manager.

Before you can create a playlist, the Content Manager must have a complete list of available tracks (see Content enumeration on page 14). This list is stored as a collection of *Media Content Records*.

Then, before the media player can play any tracks, you need to create a playlist. You will probably want to get some sort of user input to determine what tracks to add to the playlist, and in what sequence.

The purpose of separating the media content records from the playlist is so that your users can play their music in any sequence. They can play a single track multiple times, or omit available tracks entirely.



When you've populated the media records and playlist records, you pass a pointer for the playlist, and another for the Content Manager, into the Play Manager if you haven't already. You can also set a Playlist Mode, from among the following options:

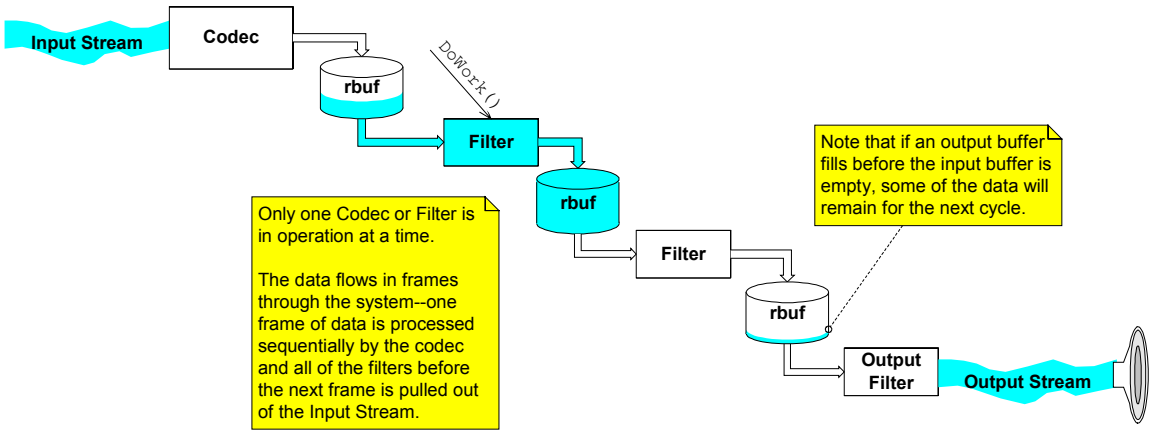
- NORMAL: Play the entire list from start to finish, and then stop.
- RANDOM: Play each track once, in random order.
- REPEAT_ALL: Play the entire list from start to finish, and then restart.
- REPEAT_RANDOM: Play a random track, and then play another random track, until the user intervenes…
- REPEAT_TRACK: Keep playing the current track until you run out of power…

Controlling the Media Player

The other half of the Dharma SDK is the code for the actual media player. You will probably not need to deal with the Media Player much at all, unless you need to add a new Codec or output stream. Your application can simply call the Play Manager with `Play()`, `Pause()`, `Seek(seconds)`, `Stop()`, `NextTrack()`, or `PreviousTrack()`, and the media player will execute, using the current playlist to determine the sequence of tracks.

When playback is started, the Play Manager first determines the current Playlist Entry. Several methods on the Playlist object assist by returning a Playlist Entry object. The Play Manager then calls `SetSong` on the Media Player object with the Playlist Entry.

The Media Player then takes over. The SetSong method gets the media content record from the playlist entry. Then, it creates a data source object associated with the media content record. Finally, the Media Player creates Codec and Filter objects, and passes an input stream into the Codec using the `SetSong()` method.



When playback is to start, the Media Player spawns a separate thread and returns. The new thread calls the `DecodeFrame()` method on the Codec. The Codec decodes data from the input stream, and writes to an output buffer until either the buffer is full, or the input stream is empty.

The Media Player then calls each filter, one at a time, using the `DoWork()` method. Each filter reads from its input buffer and writes to its output buffer until it's either out of data or the output buffer is full, and returns.

The Output Filter writes the raw output to the Output Stream. When its buffer is full, the Media Player starts the cycle again, sending another `DecodeFrame()` call to the Codec.

Applying and adjusting filters

Filters are a way of manipulating raw sound before it reaches the output stream. This SDK does not include any filters, but does provide examples that illustrate how you would write a filter.

Filters can be used to change the speed, pitch, or other characteristics of the track before it reaches the output device.

Note that volume and treble/bass controls are usually hardware-based. The SDK includes a VolumeControl object that controls the hardware settings for volume, treble, and bass.

Storing and retrieving state

Most of the objects in the SDK can be reconstructed at startup. For your users' convenience, however, you might want to store the current volume level and tone settings, or the current playlist.

The SDK provides methods to assist storing and retrieving the state of four objects:

- Playlists
- Playlist Entries
- Volume Control

You can, of course, add similar code to your other objects.

Your client application will need to call the appropriate methods on each object you want to store, and then save them appropriately. The second demo application provides an example of storing and retrieving state.

The SDK provides Input and Output stream objects for reading and writing to FAT-formatted media, simplifying the task. You can also store state information in the registry (which is generally stored to FAT-formatted media). Demo #2 provides an example of storing and retrieving the state of the Content Manager.

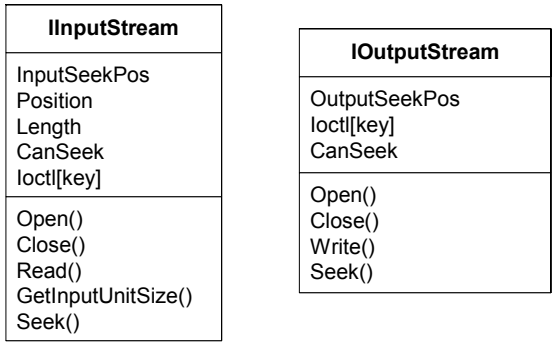
System Components

This section describes each of the major components of the Dharma SDK in greater detail.

Data Streams

Dharma includes several different types of data streams, grouped into input streams and output streams. Input streams are readable, while output streams are writable. Input data streams are typically managed and opened by a data source. Output stream objects need to be created and managed directly in your application.

Note that the media player handles the basic output for you, so you only need an output stream to write to other devices or media.



The Ioctl property is a set of key/value pairs that provide direct access to metadata about a particular data stream.

Data streams are used for all access to data sources, for metadata, raw media streams, and any other type of data file you want to use. For example, you can use a Fat file data stream to store and retrieve playlists on a compact flash card, or when serializing objects and saving state.

The following input streams are available:

- `CBufferedInputStream` – a memory buffer that is parsed as a stream.
- `CCDDAInputStream` – a CD metadata stream from a CDDA provider.
- `CFatFileInputStream` – any file stored on a FAT-12 file system.
- `CHTTPInputStream` – any file received from a server using HTTP. Not seekable.
- `CIsoFileInputStream` – a CD image as a single raw file.
- `CLineInputStream` – a digital raw signal. Not seekable.

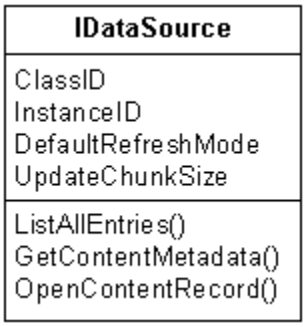
The following output streams are available:

- `CFatFileOutputStream` – a stream that writes to a file on a FAT-12 file system.
- `CWaveOutputStream` – a stream that writes to the wave output device.

You can define your own data streams for other data source types.

Data Sources

Each data source used by your device is represented by a Data Source object. The Data Source object provides a standard interface so that the other parts of your system can use the same calls for different data sources.



Each Data Source object essentially wraps drivers that interact with the device itself, and provides a common interface used by the Data Source Manager.

The Dharma SDK includes Data Source objects for the following types of devices:

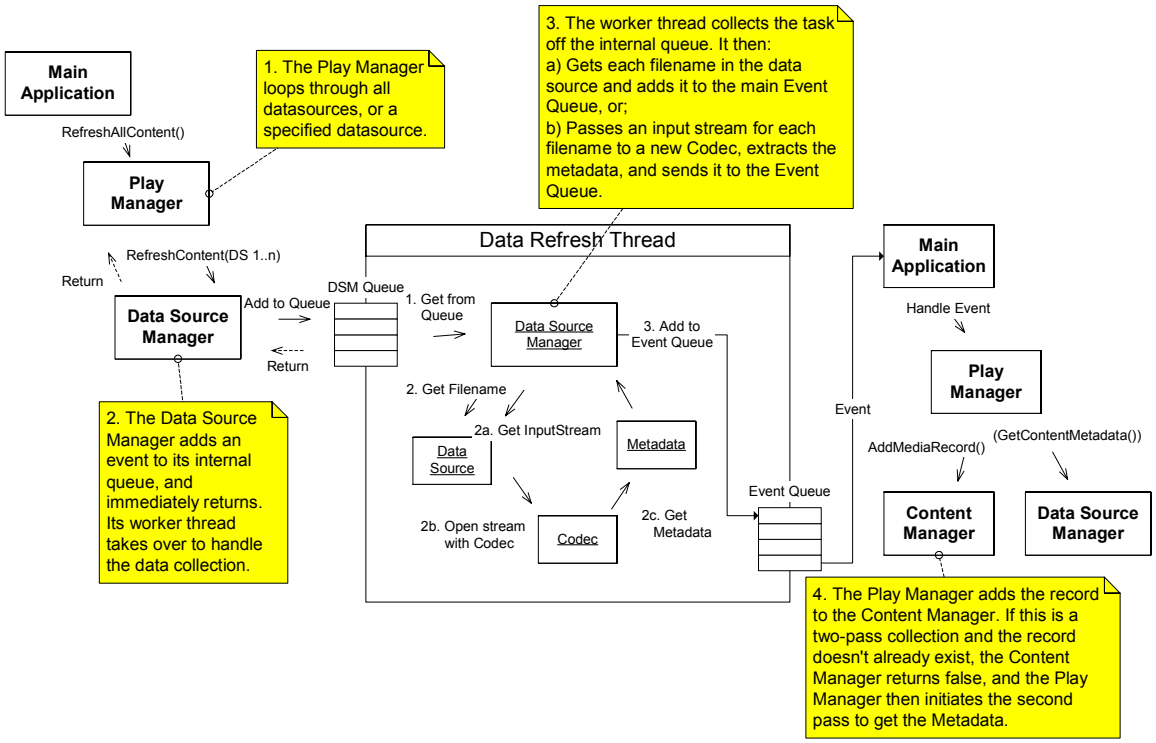
- CD Player
- FAT Device
- Serial Input
- Network Interface

Data Source Manager

The Data Source Manager handles registration of all of the data sources.

Collecting metadata is a major function of the Data Source Manager. While reporting the filenames on a given device is not expensive, most data sources do not provide easy access to metadata, such as Artist, Title, Album, track length, etc. To get this information, you generally need to open the file with its codec, a time-consuming, expensive process.

To help optimize the collection of the Metadata, the Data Source Manager uses a separate worker thread. This thread handles all of the file I/O functions so that your device can handle other tasks without waiting. The worker thread has its own internal queue, and calls to the Data Source Manager simply add requests to this queue. The worker thread sends its results to the main event queue, where your application can forward them to the Play Manager, which, in turn, adds them to the Content Manager.



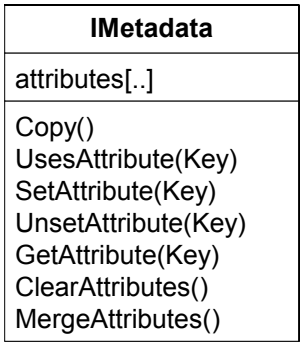
Depending on the type of content enumeration being used, the worker thread in the Data Source Manager collects a list of filenames on the data source (single-pass), a Metadata object for each track on the data source (single-pass with metadata), or a Metadata object for a specified track (second pass of double-pass). See Content enumeration on page 14 for more on the types of Content Enumeration, and Metadata below for more on Metadata objects.

Metadata

In the Dharma SDK, Metadata objects are associated with individual tracks on a data source. The interface for Metadata objects is generic, treating all metadata as key/value pairs in an associative array.

Your application can use Metadata objects to display information about a track, search for information, create catalogs, or whatever else you want. Your application can define a default Metadata creation function for the Media Player. The Content Manager may provide a Metadata object whenever a Data Source performs certain types of content enumeration. In both cases, the Metadata object is populated with information in the file decoded by the Codec.

If you don't want to use metadata, you need to specify a Single-Pass refresh mode for content enumeration, and set the Media Player metadata creation function to null (the default setting).



The `IMetadata` class is primarily used by Codec objects to populate Metadata objects. It provides methods for accessing any attribute by ID. Metadata attributes are designed as key/value pairs—you provide a key to set or get the value. Concrete extensions of this class provide other methods to access the data directly.

Metadata Object Lifecycle

Metadata objects are created at two different times:

1. By the Content Manager during content enumeration; and
2. By the Media Player when a new track is selected.

Content Manager Metadata objects

During content enumeration, the Data Source requests a Metadata object from the Content Manager to pass to the Codec. The Content Manager owns the Metadata object, and is responsible for creating, destroying, and storing it. Metadata objects are not requested during a single-pass enumeration, or the first pass of a double-pass enumeration.

See Content enumeration on page 14 for more information about content enumeration modes.

Media Player Metadata objects

When the Media Player sets a new track, it gets an input stream from the data source and passes it to the Codec. Your application can pass a default metadata creation function into the Media Player (`CMediaPlayer::SetCreateMetadataFunction()`). If you pass a function to the media player with this method, the media player will call whatever function you provide, expecting an IMetadata object in return. This object will then be passed to the Codec for data population, and finally, sent to the Event Queue.

Your application might use the Metadata object created by the media player for displaying information about the current track on the user interface.

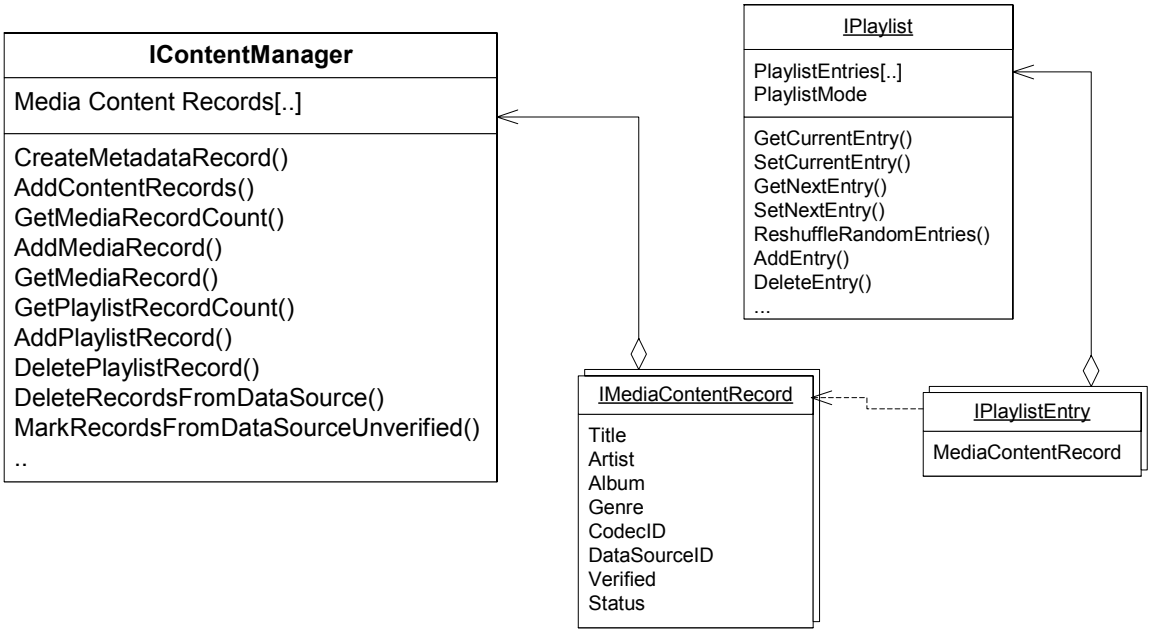
When your application has finished using the Metadata object from the Media Player, it can pass the event to the default event handler for destruction. If you do not use this event handler, you will have to destroy the object yourself.

See IMetadata in the API reference for more information about the specific Metadata objects provided in the SDK, and implementing your own metadata objects.

Content Manager

The Content Manager tracks all of the media content records from all of the data sources, and manages playlists. Of all of the SDK objects, the Content Manager and its children are the objects you may want to customize.

All Content Manager objects must extend the `IContentManager` class. The Play Manager calls several of these methods to determine what track to play next, and handle numerous other events.



Content Manager objects track all active content objects. Content objects include both playlist files and individual tracks.

The Content Manager manages a set of media content records. The Play Manager notifies the Content Manager when new media is available, when new Metadata has been retrieved, if a track was unplayable, or several other conditions. Your client application handles creating the Content Manager and passing the pointer to the Play Manager, as well as storing the serialized state.

Queryable Content Manager

The `CSimpleContentManager` class is a simple implementation of `IContentManager`. If you want to scan the list of content records using any metadata (by Artist, for example), you will probably want to use a class that derives from `IQueryableContentManager`. The `CMetakitContentManager` extends this class, allowing you to query it for lists of Artists, Albums, Genres, or everything.

The Queryable Content Manager class provides ways of filtering tracks based on metadata. Other than the demo applications, the SDK does not require you to use this class.

Content Records

A generic content record represents an individual file on a data source. `IContentRecord` is an abstract class that is extended by both `IMediaContentRecord` and `IPlaylistContentRecord`.

The `IPlaylistContentRecord` abstract class is primarily provided as a standard interface to save and load playlists to media. See Playlist Format Manager on page 28 for more about saving Playlists. A Playlist Content Record simply contains a URL corresponding to a data source and file location of a playlist file.

A Media Content Record, in addition to containing a URL pointing to the file on the data source (the implementation of IContentRecord), contains properties representing metadata of the track.

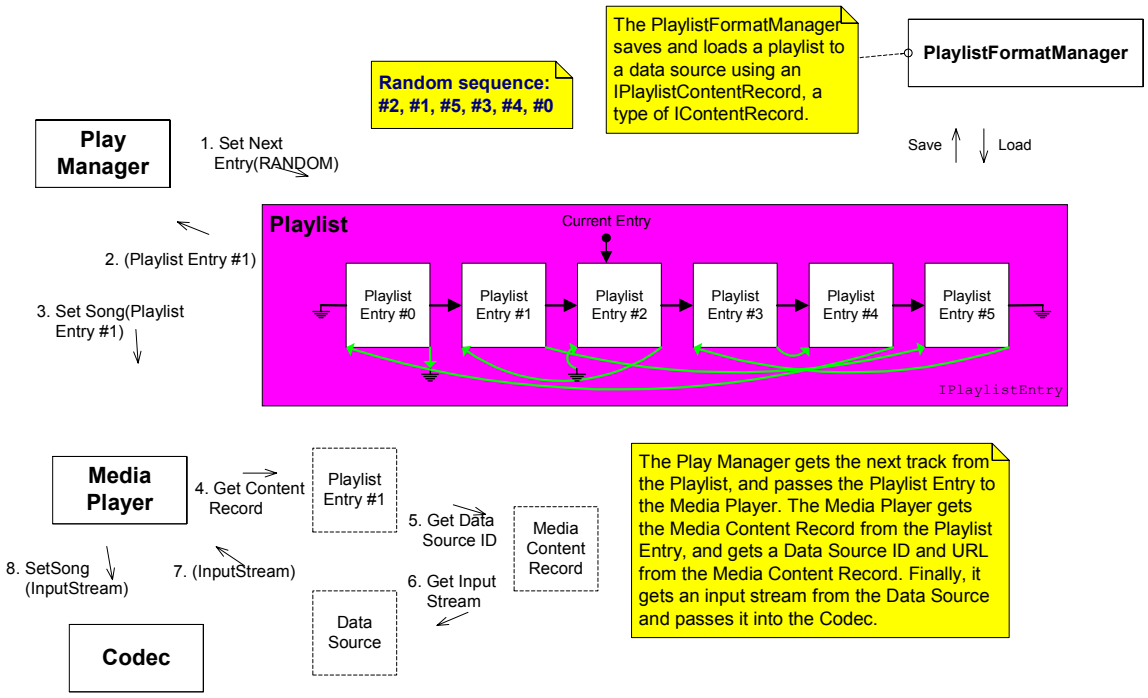
See IContentManager on page 56 for more information about Content Managers.

Playlists

A Playlist is an object representing a sequence of tracks. There are actually two sequences in each playlist: the normal indexed sequence, and the random sequence. When the Play Manager requests the next track from the Playlist object, it specifies a Playlist Mode that indicates whether to get the next track in the series, or the next random track.

The Playlist Mode may be one of the following:

- NORMAL – use the normal sequence when determining the next track.
- RANDOM – use the random sequence when determining the next track.
- REPEAT_ALL – use the normal sequence when determining the next track. When the current track is the last playlist entry, the next track becomes the first playlist entry.
- REPEAT_RANDOM – use the random sequence when determining the next track. If the current track is the last entry in the random sequence, use the first entry in the random sequence as the next track.
- REPEAT_TRACK – use the current track as the next track.



Playlist Format Manager

The CPlaylistFormatManager class maintains a list of registered playlist formats available in the system, and provides functions for saving and loading playlists to and from data streams. New playlist formats can be added through a registration interface. Two playlist formats come with the SDK: the Dadio playlist format (DPL) and Winamp's M3U format.

Loading and saving playlists

To load or save a playlist, at least three pieces of information are needed: the playlist format ID, the URL to save/load the playlist to/from, and a pointer to the IPlaylist to save/load. The playlist format ID can be found by either calling the FindPlaylistFormat function, which will return an ID based on filename extension, or by using the playlist format ID stored in a content manager's playlist content record.

The entries found in a playlist file are generally checked against the content manager before being added to the playlist; however, this check can be disabled by setting the bVerifyContent parameter of the LoadPlaylist function to false. In this case, entries in the playlist file that aren't already in the content manager will be added automatically. This is useful for quickly grabbing a playlist before doing a lengthy content update.

Adding playlist formats

New playlist formats can be added through the use of the REGISTER_PLAYLIST_FORMAT macro. Each playlist format must provide a unique format ID, a list of supported file extentions, and functions for loading and saving.

Loading a playlist

In general, your playlist function should follow these steps:

1. Open the URL for reading. This is easiest done by calling the data source manager's OpenInputStream function.
2. Add entries. Generate a URL and call the content manager's `GetMediaRecord()` function to get a pointer to the content record to add to the playlist. If no record exists and `bVerifyContent` is false, then populate a `media_record_info_t` structure and pass it to the content manager's `AddMediaRecord()` function.
3. Close the stream.

Saving a playlist

In general, the save playlist function should follow these steps:

1. Open the URL for writing. This is easiest done by calling the data source manager's `OpenOutputStream` function.
2. Traverse the playlist, saving each entry to file.
3. Close the stream.

Play Manager

The Play Manager is the core interface for the SDK, and includes the default event handler. You can pass events into the Play Manager and it will handle them with the appropriate methods. Or, you can intercept whatever events you like, and call the Play Manager to execute the appropriate code. The `HandleEvent(`*`key`*`,`*`data`*`)` method is designed to send events that bubble up from other threads to the appropriate handlers.

Events supported by the Play Manager

These events include:

- New media inserted
- Media removed
- Change of Media Player state
- Content enumeration completed
- Metadata update loaded

You will need to create your own event handlers to handle button presses, and any other event you want to handle differently than the default.

Objects not controlled by the Play Manager

Generally, your client code will call on the Play Manager to control the Media Player, and to interact with data sources. Your client code will interact with the following objects directly:

- Playlists
- Content Managers
- Volume Control
- Playlist Format Manager.

The Play Manager offers some fairly sophisticated features.

1. Automatic handling of media removal. When Play Manager receives a "Media Removed" event, or if the `NotifyMediaRemoved()` method is called, it removes the data source from the Data Source Manager, removes all of the media content records for that data source, and removes all playlist entries for the data source. If the current track is on that media, it stops playback and attempts to start the next track.
2. Automatic handling of media insertion. When the `NotifyMediaInserted()` method is called, the Play Manager asks the Data Source Manager to enumerate the content of that data source.
3. Content enumeration. The Data Source Manager sends event messages containing the contents to create a media content record. If these events are passed into the event handler in the Play Manager, the Play Manager will add them to the Content Manager.
4. Search for missing playlist entries. If the Media Player cannot resolve a playlist entry, it calls the `SetStatus()` method on the Playlist Entry, setting it to whatever the result was. The Play Manager then searches for a valid playlist entry, trying tracks one at a time until it finds one that works. When attempting to get a valid track, the Play Manager first searches forward in the list from the current track, and then searches backwards. Only if no entry in the entire playlist is present will the Play Manager return an error.
5. Caching of current settings for the Media Player. The Play Manager tracks the current playlist mode, and maintains pointers to the current Playlist and Content Manager objects. Your client code can call methods such as `Play()` and `NextTrack()` without sending any arguments, and the Play Manager will know what to do.

Media Player

The Media Player controls the actual playback on your device's speakers. The Play Manager handles all interaction with the Media Player, so you don't need to do anything to get it to work. You can, however, call methods on the Media Player to find out details about the currently playing track, including the length of the track, and how far into the track the media player currently is.

The Media Player has four states:

1. PLAYING
2. PAUSED
3. STOPPED
4. NOT_CONFIGURED.

Before starting the media player, it must be associated with a Playstream, which points to the output device. The media player starts out in the NOT_CONFIGURED state, until a playlist entry has been passed to the `SetSong()` method. When this method is called, the Media Player creates an appropriate Codec object and all of the registered filters, and transitions to the STOPPED state.

You can then add or remove filters to the data flow by using the `AddNode()` and `RemoveNode()` methods, adding as many filters as desired.

Codecs

The Dharma OS provide an abstract interface to codecs (typically decoders). Arbitrary codecs can be linked in with the library and be automatically available to applications. Codecs are associated with a range of file extensions, and can optionally be probed to determine if they can decode a given bitstream. Codecs are dynamically created and released.

Components

There are 3 basic pieces to the Dharma(tm) codec system, each of which are covered in more detail later:

1. ICodec interface
   (`./player/codec/common/include/Codec.h`)
   The ICodec interfaces describes the necessary APIs that a codec must implement to operate in the Dharma environment.
2. CCodecManager class
   (`./player/codec/codecmanager/include/CodecManager.h`)
   The CCodecManager class provides a mechanism for locating a given codec by file extensions it supports (such as ".mp3"), by a specific codec ID in situations where the type of codec is well known (as is the case with CD audio), and also by probing available codecs to determine if they can decode the bitstream.
3. Registration interface
   (`./player/codec/common/include/Codec.h`)
   The registration interface provides a way for arbitrary objects to make themselves globally visible to Dharma(tm). The Codec interface defines the necessary macros to register your class and define certain basic routines needed for the CCodecManager class to properly work with your codec.

Program flow

During startup, all codecs register themselves with the system (specifically, with the system registry). Later on, during player initialization, the CCodecManager singleton is instantiated, and it builds a list of supported file extensions based on the registered codec list. At this point the CCodecManager is available to fulfill the system's codec needs.

The goal of the codec system is to allow third party codec writers to easily integrate their codec into the Dharma(tm) OS. This is accomplished by providing a single interface (ICodec) to write to, and a registration system to incorporate their codec into the main system.

Codec Manager

The Codec Manager is a singleton object that the system can use to access available codecs. It keeps a list of supported extensions, in addition to table of pointers to creation routines for the available codecs. The assembly of this list is facilitated through the registration interface.

When the Media Player gets an input stream, it passes the codec ID stored in the media content record to the Codec Manager to get a codec.

If the codec cannot decode the input stream, or the codec ID is missing or invalid, the Codec Manager first attempts to load a codec associated with the file extension. If that doesn't work, the Codec Manager will step through all of the registered codecs to find one that works. If none work, the Media Player returns an error.

For more information about the Codec Manager, see the CodecManager.h file.

Volume Control

The Volume Control object provides methods to change the hardware settings on the device itself. The Volume Control controls volume, bass, and treble. For more sophisticated sound control, you can use filters.

Filters

Filters provide the ability to enhance, distort, or otherwise modify the raw output of the media player. Systems such as Sound Retrieval System (SRS) can be added using filters. Filters are always downstream of the codec.

The SDK provides the interfaces for creating filters, but there are no filters included.

The `IFilter` class behaves in a similar way to the `ICodec` class. It's designed as a wrapper to the actual code. You implement the methods for the `IFilter` class that call the appropriate methods in the actual filter.

The SDK includes a ring buffer system (RBUF) for buffering data between codecs and filters. Your filters will generally read from a dedicated input RBUF, and write to a dedicated output RBUF. Filters and RBUFs are strung together in a daisy-chain manner. The last RBUF becomes the input buffer for the Output Filter, which writes directly to the Output Stream.

Filter Manager

The Filter Manager creates filter objects on request from the Media Player. To work correctly, you must register your filters, and provide input and output RBUF sizes.

The Filter Manager can provide any filter previously registered using the registration interface. See the Filter.h documentation for more details on how to register a filter.

Creating Your Media Player

This section describes the demos available as part of the Dharma SDK, global macros and types you can use in your application, and details about extending the provided abstract classes.

The best way to create a player application is to start with one of the demo applications, and extend them as necessary to accomplish what you want.

Read through the Global Definitions section on page 44 before you start, and then dig into the source for the demo applications. If you need to implement one of the abstract classes, see the appropriate section below.

Demo Applications

The Dharma SDK includes three demo applications you can use to model your application. These demos are completely functional, and all of the source code is provided so you can build off of existing work.

Each demo is available in both compiled and source forms.

Demo User Interface

All of the demos share a very simple user interface (UI). There are icons for Play, Pause, and Stop. All other functions display a simple text message to indicate their status.

The UI consists of an 11-line text display. When the Dharma Board is operating, it displays a banner indicating which demo is playing, the play state and track time of the current track, the title, artist, album, and genre of the current track, the Playlist Mode, the Volume, Bass, and Treble settings, and a status message.

Demo Excluded Features

Due to time and developer constraints several features are not supported in the demo applications. The following features are not demonstrated:

- **Track seek** – Seeking within a track is not demonstrated in the demo applications.
- **Power management** – No power management features are used in the demo applications.

Demo 1: Local playback from a hard drive

The hard drive demo highlights several features of a simple digital audio device such as a Jukebox or a portable player. You can access, navigate and play content with limited user interface and player controls from a spinning media storage device, and get an idea of the capabilities and flexibility of the Dharma platform with Dharma SDK. This demo focuses on player controls and the user interface.

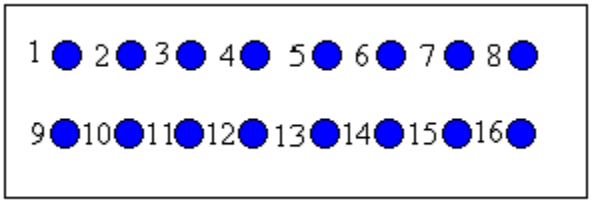
Features

Demo #1 supports the following features:

- **Play files from a hard drive.** Access a drive and create a file database that allows the media player to navigate and play tracks from the drive.
- **Simple UI.** Display player functions; File name or Track ID and Artist name and album name for the playing track, Volume level, Bass level, Treble level, Play-pause-stop mode, playlist mode, Time code for track.
- **Save and load device state information.** Save and load the current settings of the device (Volume level, bass level, and treble level). The last saved state of these settings are loaded from a settings file at startup.
- **Automatic playback.** At start-up, this demo scans the contents of the hard drive and builds a simple playlist. It then automatically starts playing the first track.
- **Support button events.** Play-Pause/Stop/Next track/Previous track/ Volume up/ Volume down/ Bass up/ Bass down/ Treble up/ Treble down/ Save device state/ Playlist mode.
- **Playlist mode.** Support normal, random, repeat, random repeat, and repeat track modes for playlists.

Demo button configuration and description

The Dharma bard has 16 buttons for user input to control the demos. Once loaded and running the demo will respond to input from button array.



The hard drive demo supports the following button functions:

Button #	*Function*	*Notes*

2	Previous track	The previous track button operates in two modes depending on where you are in the currently playing track. If the current track is playing and is in the first five seconds of play, pressing the button will navigate to the previous track. If the player is at the beginning of the playlist and not in a repeat mode, this button has no effect. If the playing track has played longer then the first five seconds pressing the previous track button will navigate to the start of the current track.
3	Volume Up	The volume up and down buttons have 21 volume steps. Pressing the volume up/down button will increase/decrease the volume on the device from its current step location one step per press until it's at the maximum or minimum level. The UI displays "Volume" and a numerical level number between 0 and 20. Pressing and holding the volume buttons will increase/decrease a step every second until the maximum or minimum level is reached.
4	Bass up	The bass up and down buttons have 12 bass steps. Pressing the bass up/down button will increase/decrease the bass on the device from its current step location one step per press until it's at the maximum or minimum level. The UI displays "Bass" and a numerical level number between 0 and 11. Pressing and holding the bass buttons will increase/decrease a step every second until the maximum or minimum level is reached.
5	Treble up	The treble up and down buttons have 12 treble steps. Pressing the treble up/down button will increase/decrease the treble on the device from its current step location one step per press until it's at the maximum or minimum level. The UI displays "Treble" and a numerical level number between 0 and 11. Pressing and holding the treble buttons will increase/decrease a step every second until the maximum or minimum level is reached.
6	Playlist Mode	The playlist mode button modifies the behavior of the current playlist. There are five modes: Repeat All, Repeat Random, Repeat Track, Normal, and Random. The UI displays the words "Playlist Mode" and the current mode. Each time you press the button, the next mode appears on the UI, rolling through the modes in the above order.
7	Not used	

8	Save Device State	The Save Device State saves the current volume, bass and treble settings for the device. This feature shows how a device state file is saved and loaded next time the power is cycled. After pushing this button the demo status line displays "Saving State" followed by "Saved State" if the operation is successful.
9	Stop	The stop button stops the playing of the current track. The track will not restart until play is pushed. This button changes the User Interface in the main play mode by replacing the "play" icon with the "stop" icon.
10	Next track	The Next Track button navigates to the next track in the system. If the device is at the end of the playlist and not in a repeat mode, this button will do nothing.
11	Volume Down	See Volume Up (button #3).
12	Bass down	See Bass Up (button #4).
13	Treble down	See Treble Up (button #5).
14	Query drive-refresh content DB	Pressing this button causes the unit to spider the hard drive and create a content database of all tracks found on all media types active. When finished, the status line will display "Content update finished", and then start playing the first track in the database. If there was an error, the status line will read "Error updating content."
15	Not used	
16	Load Device State	This button loads the previously saved volume, bass, and treble settings.

Demo 2: Local Playback with Metadata Sorting

The hard drive #2 demo builds on demo #1. This demo reads tracks from the CD-ROM player and Compact Flash, as well as the hard drive. Several of the player controls are changed to support selecting a group of tracks based on artist, album, or genre of the content. This demo highlights the ability of the player to manage large amounts of content. It demonstrates saving and loading the state of the content manager to keep content update times to a minimum. The infrastructure that supports this functionality is a Metakit embedded database.

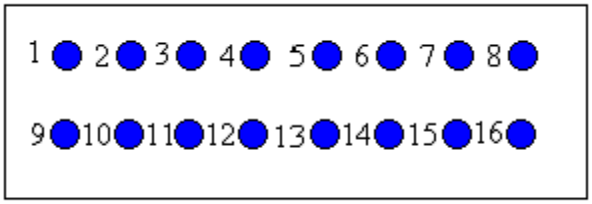
This demo does not start playing immediately; you must first populate the content manager by starting a content refresh, or by loading state. To get the most from this demo, you need to copy several CDs of diverse music to the hard drive.

Features

Demo #2 supports the following features:

- **Metakit embedded database.** To sort content, a database is installed and all file records are stored in the content manager with associated metadata.
- **Play files from a hard drive or CD.** Access a drive or CD and create a file database that allows the media player to navigate and play tracks from the drive.
- **Simple UI.** Display player functions; File name or Track ID and Artist name and album name for the playing track, Volume level, Bass level, Treble level, Play-pause-stop mode, play list mode, Time code for track.
- **Save and load device state information.** Save and load the current settings of the device (Volume level, bass level, treble level, Playlist track selection, content database).
- **Playlist support.** Create, save and load a Playlist
- **Support button events.** Play-Pause/Stop/Next track/Previous track/ Volume up/ Volume down/ Bass up/ Bass down/ Treble up/ Treble down/ Sort by Album/ Sort by Artist/ Sort by Genre/ Save device state/ Query drive-refresh content DB/ Save Playlist/ Load Playlist/ Playlist mode.
- **Playlist mode.** Support normal, random, repeat, random repeat, and repeat track modes for playlists.
- **Searching by genera, artist, track, album.** Searching and filtering is not provided in the demo. This demo only sorts the current playlist by these categories.
- **Playlist Management.**This demo allows you to save and restore playlists using the database.

Demo button configuration and description



Demo #2 uses many of the same buttons as Demo #1. Here's a rundown:

Button #	***Function***	***Notes***
1	Play / Pause	Same as Demo #1.
2	Previous track	Same as Demo #1.
3	Volume Up	Same as Demo #1.
4	Generate List By Artist	Pressing the Sort by Artist button causes the demo to generate a list of all tracks by a single artist. The demo then plays the first track in the list. Pushing the button again cycles through the list of artists in alphabetical order.
5	Generate List By Album	Pressing the Sort by Album button causes the demo to generate a list of all tracks on a single album on all available media. The demo then plays the first track in the list. Pushing the button again cycles through the list of albums in alphabetical order.
6	Playlist Mode	Same as Demo #1.
7	Save Playlist	The save Playlist button saves the current playlist to a Playlist file on the hard drive. The UI displays "Saved Playlist" on the status line if the operation is successful. The demo saves the playlist on the hard drive in the Dharma proprietary *.dpl format.
8	Save Device State	Same as Demo #1, but it also saves the state of the content manager (the database containing the track information).
9	Stop	Same as Demo #1.
10	Next track	Same as Demo #1.
11	Volume Down	Same as Demo #1.
12	Generate List By Genre	Pressing the Sort by Genre button causes the demo to generate a list of all tracks from a single genre on all available media. The demo then plays the first track in the list. Pushing the button again cycles through the list of genres in alphabetical order.

13	Generate List by Playlist	This button cycles through all playlists (*.m3u and *.dpl files) found on all available media during content refresh. When you press this button, a single playlist is loaded and the first track starts playing.
14	Query drive-refresh content DB	Pressing this button causes the unit to spider all available drives and create a content database of all tracks found on all media types active. For each data source, the message line displays the messages, "Starting content update on data source X", "Scanning content on data source X", "Content update finished on data source X", "Scanning metadata on data source X", and finally, "Metadata update finished on data source X", where X is the ID of the data source. The Dharma board then starts playing the first track in the database.
15	Load Playlist	The Load Playlist button loads the Playlist saved using button #7. If no playlist has been saved with button #7, this button does nothing.
16	Load Device State	Same as Demo #1, but also loads the state of the Content Manager.

Demo 3: Streaming technologies

The streaming technologies demo uses Shoutcast URLs to connect to a streaming data source. This demo uses the same player controls as the Demo 1: Local playback from a hard drive, except for loading and saving device state (buttons #8 and #16). The data source consists of several URLs hard-coded into the `main.cpp` file that the device accesses on startup and streams through the player. A Next Shoutcast URL button (mapped to button #7) lets you cycle through all of the URLs mapped to the demo.

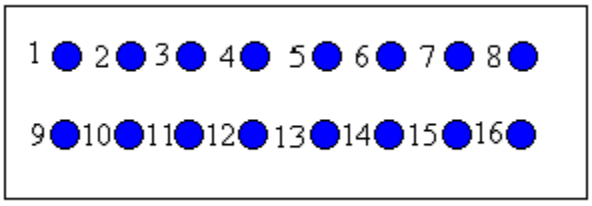
Features

Demo #3 supports the following features:

- **Play files from a Shoutcast URL.** Access the internet and stream content from the server.
- **Simple UI.** Display player functions; File name or Track ID and Artist name and album name for the playing track, Volume level, Bass level, Treble level, Play-pause-stop mode, play list mode, Time code for track, Internet source.

- **Save and load device state information.** Save and load the current settings of the device (Volume level, bass level, treble level, Playlist track selection, content database).
- **Support button events.** Play-Pause/Stop/Next track/Previous track/ Volume up/ Volume down/ Next Internet URL/ Save device state.

Demo button configuration and description



The streaming media demo supports the following button functions:

Button #	*Function*	*Notes*
1	Play / Pause	Same as Demo #1.
2	Previous track	Five Shoutcast URLs are stored in the demo, each pointing to a streaming media source carrying a different genre – Classical, Rock,
3	Volume Up	Same as Demo #1.
4	Bass up	Same as Demo #1.
5	Treble up	Same as Demo #1.
6	Not used	
7	Not used	
8	Not used	
9	Stop	Same as Demo #1.
10	Next track	Five Shoutcast URLs are stored in the demo, each pointing to a streaming media source carrying a different genre – Classical, Rock,
11	Volume Down	Same as Demo #1.

11	Volume Down	Same as Demo #1.
12	Bass down	Same as Demo #1.
13	Treble down	Same as Demo #1.
14	Not used	
15	Not used	
16	Not used	

Shoutcast URLs

The Net Data Source object supports DHCP and DNS. You can provide a named URL or a static IP address to access a media stream.

Global Definitions

This section describes definitions, macros, and types used globally in the Dharma SDK. Take advantage of them to simplify your client application development. The list includes:

- Data types
  - ERESULT
  - TCHAR
- Containers
  - Simple List
  - Simple Vector
  - Simple Map
- Macros
  - Debugging system
  - Events
  - Registry
  - System Timer

Data Types

ERESULT

An ERESULT is a 32-bit code divided into Severity (8 bits), Zone (8 bits), and Code (16 bits). To use ERESULT in your own code, you will need to define a unique zone and include the `util/eresult/eresult.h` header. You can then define any integer as a return code.

ERESULT is a standard return data type used extensively in the SDK. It provides a way of quickly evaluating whether an operation completed successfully, and if not, what the problem was.

You can simply check `succeeded()` or `failed()` on the ERESULT to quickly determine whether the call was successful. These methods return boolean values.

TCHAR

The TCHAR data type is a double-byte character encoding, basically a wide text format.

Containers

Simple List

The Simple List container is a double-linked list. Each element in the container contains a data item and two pointers: one to the previous item, and one to the next item.

To use the Simple List container, include the `util/datastructures/include/SimpleList.h` file.

Simple Vector

The Simple Vector container is a basic indexed array. It has functions for adding, removing, and inserting values using an index value.

To use the Simple Vector container, include the `util/datastructures/include/SimpleVector.h` file.

Simple Map

The Simple Map container is an associative array. You can store and retrieve data associated with a key value.

To use the Simple Map container, include the `util/datastructures/include/SimpleMap.h` file.

Macros

Debugging System

The SDK includes debugging macros. You will mostly use three functions to provide debugging information:

1. DEBUG_MODULE() – use to specify a module for debugging. You should define your own modules using this function.
2. DEBUG_USE_MODULE() – use this to specify an existing module to send the messages to.
3. DEBUG() – use this function to send a message of a specified severity to the debug log.

Debugging can be turned on or off by module, and you can use a single switch at compile time to turn off all debugging. Each message has a severity level, and for each module, you can specify what level of messages you want debugged.

The macro is in `util/debug/include/debug.h`.

Event Queue

All events are handled by the Event Queue API. To use this API, include the `util/eventq/include/EventQueueAPI.h` file in your header. This header defines methods for both C and C++ code.

To send an event to the Event Queue, you simply provide an ID and data to the PutEvent() method.

Registry

The Registry is a place where you can store data you want to be persistent. Your application should restore the registry on startup, and save it on shutdown to a FAT device, or provide another data source type capable of storing it.

The Registry provides a simple key/value database system. To use it, include the `util/registry/include/Registry.h` file.

System Timer

The System Timer provides a way to execute functions at specific intervals. The timer uses a tick interval of 100 ms, or 1/10th of a second. To use it, you include the `util/timer/include/Timer.h` file, and register a function with the interval and number of iterations.

Dharma Abstract Classes

This section describes the abstract classes you may implement to extend functionality of the applications you develop. Interactive Objects provides basic implementations of these classes, but by extending these classes, you can create additional functionality that can be managed by the higher-level objects in the system.

The abstract classes in this release of Dharma include:

- DataSource
- DataStreams
- Metadata
- Codec
- Filter
- ContentManager
- Playlist

We'll now take a closer look at each of these classes.

ICodec

The ICodec class is a wrapper class for your actual codecs. The SDK includes full codecs for the WMA and PCM formats, and pre-compiled object code for the ARM MP3 codec.

Note: The included MP3 codec is licensed from ARM, Inc.

You can add a new codec by simply writing a new wrapper object that extends the ICodec class, calling the appropriate methods on the codec from the wrapper.

See the Codec.h file for the specific structures and methods for the ICodec class.

Codec Registration System

Dharma provides an abstract way for codecs to be available to the playback system through codec registration. The actual registration process has been simplified to a pair of easy to use macros, which take care of the actual work.

DEFINE_CODEC Macro

Fills out the codec class definition with additional members necessary for codecs to work in Dharma. This satisfies the underlying requirements of the ident (IIdentifiableObject) baseclass.

Arguments

`codecname`	A string name for the codec
`codecID`	A unique identifier for the codec
`can_probe`	Either true or false indicating whether the system can probe bitstreams against this codec. In the case of PCM and similar pass through codecs, this should be false
`extensions...`	A variable length list of strings that indicate which extensions are supported by this codec. This is case insensitive.

Example usage

```
DEFINE_CODEC( "iObjects MP3 Codec", 783, true, "mp3" )
DEFINE_CODEC( "iObjects PCM Codec", 784, false, "pcm",
"raw" )
```

REGISTER_CODEC Macro

To be used in your implementation file. This macro defines helper data structures and has function implementations necessary for your codec to be registered with Dharma.

Arguments

`classname`	The name of your class, such as 'CMP3Codec'
`codecID`	The same codecID used in the DEFINE_CODEC macro

IFilter

Filters provide access to the current PCM playback stream. Filters have the ability to analyze and modify the PCM data, and can even resize the amount of data being passed through the system. Examples of filters would be a sample rate converter, a PCM disk writer, or a wrapper for a PCM enhancement library. Filters are chained together, so it is possible to have multiple instances of such filters. The chain is terminated by a special filter called the OutFilter. The OutFilter reads all its input data and writes it to an Output Stream, typically the `CWaveOutputStream` for audio playback.

Filters are chained together through RBUFs, a special type of ring buffer that provides write and read handles (see Controlling the Media Player on page 18). A given filter writes data to its write handle, which maps down to a ring buffer. The next filter in the chain has a read handle which maps to the previously stated ring buffer, giving it access to the generated audio data. In this way, RBUFs are fairly similar to pipes, with a few exceptions and some special properties and considerations.

The chaining between codec -> filters -> outfilter is managed by the MediaPlayer, which sets up the appropriate components on track changes. When it is creating items in the playstream, it queries each object for its unit sizes. Since filters read and write data to the stream, they are asked for input and output unit sizes. At the moment, these unit sizes are used only to ensure that the buffers are large enough to support the reader and the writer of a given RBUF. Hence, filters are expected to return the minimal amount of input data and output space required for them to perform work.

Filters that need special configuration can implement custom routines through an `Ioctl()` interface. The `ioctl` interface provides a generic method to query data from the filter and set custom parameters. Other components in the system can then issue `ioctls` to the filter through the mediaplayer, which will locate the instance of the filter and apply the appropriate settings.

Filters are loaded abstractly based on their unique ID. This ID is passed to the filter manager, which then instantiates the requested filter and passes back a pointer to it. This allows customization of standard components, such as the OutFilter or the sample rate converter. Filters must register with the system registry at startup to be available to the media player. See Filter Manager on page 33 for more about filter management.

IMetadata

The SDK includes two concrete Metadata classes:

- `CSimpleMetadata` – provides simple methods for accessing Title, Artist, Album, and Genre only.
- `CConfigurableMetadata` – tracks whatever metadata you add to the attribute list.

Simple Metadata

If you only need Artist, Album, Genre, and Title information, use the `CSimpleMetadata` class. This class, in addition to the `IMetadata` interface, provides direct methods for accessing these data: `GetTitle()`, `GetArtist()`, `GetAlbum()`, and `GetGenre()`. All other metadata associated with the track is ignored.

Configurable Metadata

This object contains no set metadata properties; it must be configured with all properties you wish to track. You can use the `AddAttribute()` method to add each type of metadata you want to track to the object as an attribute. When you add an attribute to any `CConfigurableMetadata` object, it is added to all instances of the object.

Attributes are identified using an integer ID. The Metadata.h file defines all metadata types currently known to all codecs in the SDK. You can use these metadata types with no further work.

However, if you want to define other metadata types that are not populated directly from the track, you need to define it in your header and register it in the Metadata Table using `CMetadataTable::AddMetadataAttribute()`. You also need to provide the data type, which may be an integer or a TCHAR.

Custom Metadata Objects

If you want a clean, optimized Metadata object with only a small set of metadata types, and the Simple Metadata Object does not meet your needs, you can define your own. The Configurable Metadata Object is capable of handling any metadata, but you may be able to improve performance by hard-coding the specific metadata types you need.

Like the Configurable Metadata object, if you want to define your own type of metadata, you need to register it in the Metadata Table. Otherwise, implementing your own Metadata object is straightforward. See the code for the existing Metadata objects for ideas.

See Metadata on page 23 for more information about Metadata objects.

IDataSource

Data Source objects represent physical data sources, including: CD-ROM drives, hard drives, compact flash cards, and network locations. The Data Source object is a wrapper that provides a standard interface for accessing media that uses different drivers.

The Dharma SDK includes data source objects for all hardware devices supported by the Dharma board. However, if you want to add a new device that does not use the same format as an existing data source, you will need to create a new Data Source class.

The basic role of the Data Source object is three-fold: provide a unique URL for every media track available on the object; provide an input or output stream in response to a request for a particular track; and collect metadata from specific tracks on request. For the actual device, you will also need to consider how to intercept an event such as media removal.

See IDataSource in the API reference for more information.

Data Source URLs

Content records from a data source must be identified by a URL that is unique across the system. The URL has three parts: a prefix, an identifier for the specific data source, and a name for the stream.

The only rigid criterion for the URL is that it must be unique across the entire system, and the Data Source Manager must know what Data Source to forward the request to. To simplify this, Interactive Objects recommends using URLs that have the same structure as normal Internet URLs. So while the prefix looks like a profile, it is really only a pattern identifying what Data Source object to route the request to.

For example, the CFatDataSource object prefixes its URLs with `file://`.

The rest of the URL is up to you to define, but if your Data Source class handles multiple objects, you will want to define a unique path to each one. For example, the FAT data source uses filenames to identify each stream. So the file `a:\bach.mp3` has a complete URL of `file://a:\bach.mp3`.

Enumerating URLs

The Data Source Manager calls the `IDataSource::ListAllEntries()` method to update the list of available tracks. When you implement this method, you will need to observe the following sequence to provide the events and memory management the Data Source Manager and Content Manager are expecting:

1. `put_event(EVENT_CONTENT_UPDATE_BEGIN, (void*)GetInstanceID());` to indicate that you have begun processing a content update.
2. Create a new `content_record_update_t` struct using `new`. This object will be destroyed by the default event handler. Set the `iDataSourceID` to your instance ID and the `bTwoPass` field to TRUE if in DSR_TWO_PASS mode or FALSE otherwise.
3. Loop through the available media tracks and playlists on the device, until the count of tracks equals the batch size provided in `iUpdateChunkSize`. For each track/playlist, add a media content record or playlist content record (see below).
4. Send an EVENT_CONTENT_UPDATE event with a pointer to the content_record_update_t struct, and repeat at step 2 until you're out of tracks and playlists.
5. `put_event(EVENT_CONTENT_UPDATE_END, (void*)GetInstanceID());` to notify that you're done, and return.

If at any point you get an unrecoverable error, send `put_event(EVENT_CONTENT_UPDATE_ERROR, (void*)GetInstanceID());` before terminating.

If `iUpdateChunkSize` is zero, you are to return all records in a single `content_record_update_t` struct.

Adding Media Content Records

This section describes how to add a media content record to the content_record_update_t struct, while going through the loop described above.

1. For a file system, check to see if the file extension is recognized by the Codec Manager.
2. Populate a `media_record_info_t` struct for each record to be added. You will push this object on to the `content_record_update_t` struct at the end of the loop.
3. Use `malloc` to allocate space for the URL. This space will be freed by the Content Manager.
4. Set `szURL` to a unique URL.
5. Set `iDataSourceID` to your instance ID.

6. Set `bVerified` to TRUE. By definition, records coming from a data source are verified. This field is FALSE when media content records are restored from other sources.
7. Set `iCodecID` to the value returned by the Codec Manager's `FindCodecID()` function. If set to zero, the Media Player will determine the codec during playback.
8. If the refresh mode is DSR_ONE_PASS_WITH_METADATA, call `CreateMetadataRecord()` on the current Content Manager (`CPlayManager::GetContentManager()`). If the Content Manager returns zero, no metadata retrieval is necessary (though if the application is properly coded, we shouldn't be in DSR_ONE_PASS_WITH_METADATA). Otherwise, set values on the Metadata object of any attributes your data source can find, then open an input stream and pass it to the appropriate codec's `GetMetadata()` function.
9. Call `PushBack()` on the `content_record_update_t` struct's media field to add the record to the list.

Adding Playlist Content Records

This section describes how to add a playlist content record to the content_record_update_t struct, while going through the loop described above.

1. For a file system, check to see if the file extension is recognized by the Playlist Format Manager.
2. Create a `playlist_record_t` struct for each record to be added. You will push a copy of this object on to the `content_record_update_t` struct at the end of the loop, so this is a local object you will need to destroy.
3. Use `malloc` to allocate space for the URL. This space will be freed by the Content Manager.
4. Set `szURL` to a unique URL.
5. Set `iDataSourceID` to your instance ID.
6. Set `bVerified` to TRUE. By definition, records coming from a data source are verified. This field is FALSE when playlist content records are restored from other sources.
7. Set `iPlaylistFormatID` to the value returned by the Playlist Format Manager's `FindPlaylistFormat()` function.
8. Call `PushBack()` on the `content_record_update_t` struct's media field to add the record to the list.

Getting Content Metadata

If the refresh mode is DSR_TWO_PASS, then some records may be sent to the `GetContentMetadata()` method for the second pass.

All content enumeration happens using a single thread. Events are queued seperately to the Play Manager thread and the Data Refresh thread. So in two-pass mode, you will have already sent all of the batches of media content records to the event queue before this method gets called.

At the beginning of the second pass, the Data Source must send an EVENT_CONTENT_METADATA_UPDATE_BEGIN message, sending the instance ID of the data source as the data parameter. When the metadata update is complete, the system will automatically send an EVENT_CONTENT_METADATA_UPDATE_END message.

Meanwhile, on the Play Manager thread, the Content Manager receives each `content_record_update_t`, merges the media content records with its current set, removes media content records that it already has metadata for, and then sends the remaining media content records, in the same `content_record_update_t` struct to the data refresh queue for metadata collection.

So the `GetContentMetadata()` function will get a pointer to a variable-length `content_record_update_t` struct. For each content recored in the struct, you need to collect metadata.

To do so, call `CreateMetadataRecord()` on the current Content Manager (`CPlayManager::GetContentManager()`). If the Content Manager returns zero, no metadata retrieval is necessary (though if the application is properly coded, we shouldn't be in DSR_TWO_PASS). Otherwise, set values on the Metadata object of any attributes your data source can find, then open an input stream and pass it to the appropriate codec's `GetMetadata()` function.

Providing Streams

Data is retrieved from (and sometimes sent to) the data source through input and output streams. The Data Source is responsible for creating streams appropriate to the content type. You need to implement the `OpenInputStream()` and `OpenOutputStream()` functions to provide this ability. Both functions accept a URL argument and return a pointer to the input stream or output stream for the content. If the URL cannot be opened for whatever reason, return a zero.

Capturing Events

If the device your data source represents does anything unusual (such as having a removable media system like a CD-ROM or compact flash), you need to provide a way to send and handle an appropriate event.

Your driver system may provide a mechanism for sending such an event. You might also choose to register a function in the system timer (see System Timer on page 46) that polls the media at regular intervals to see if it is still present.

However you detect a media change, you can send an EVENT_MEDIA_REMOVED or an EVENT_MEDIA_INSERTED event to the Event Queue and the default handler will either remove all entries in the content manager that came from that data source, or trigger the `ListAllEntries()` function on the data source using the default update mode for the data source.

For other behavior, or other types of events, you will need to create your own event handler.

Data Streams

Dharma provides interfaces for generic input and output of data in the form of IInputStream and IOutputStream classes. Implementations of these classes are required for playback: specifically, an IInputStream derived class is used to read input data into the playstream, and an IOutputStream derived class is used to write audio out to a destination. In terms of actual implementation, the difference between the two is that IInputStream has a `Read()` routine, and IOutputStream has a `Write()` routine.

Input and Output Streams can also be used to read or write any other type of data. For example, the Playlist Format Manager writes playlists to FAT media using an output stream, and Content Managers load playlists from input streams.

The following Input Streams are provided:

- `CCDDAInputStream` – Tracks on a music CD
- `CFatFileInputStream` – Files on FAT media
- `CHTTPInputStream` – Files received over http
- `CIsoFileInputStream` – Files on a CD-ROM
- `CLineInputStream` – Wave input over an analog line-in port

The following Output Streams are provided:

- `CFatFileOutputStream` – File handle on a FAT media
- `CWaveOutStream` – Output device such as a speaker.

If you need to send data to another device, or read it from an unsupported device, you will need to write your own data stream.

Although these are labeled as streams, there are two routines implemented for seek support: `Seek()` and `CanSeek()`. Calling `CanSeek()` will indicate if the current stream supports the seek operation, and `Seek()` will actually seek in the stream (if seeking is supported).

Input streams are typically created through data sources. A URL can be passed into the data source manager, which will locate the appropriate data source to open the input stream. Input streams are closely bound with data sources, but can be instantiated directly. Output streams must be instantiated directly.

In situations where a stream can be both input and output, it may prove useful to implement a wrapper class and have `IInputStream` and `IOutputStream` derived classes that use the wrapper instead of the underlying API. This will help ensure a consistent usage of the underlying API, and will also allow programs that need simultaneous read/write access to instantiate a class tailored to their needs.

IContentManager

The primary responsibility of the Content Manager is to maintain a list of content available on all of the data sources. Content is divided into two types: media content records, and playlist content records. Content records have two unique identifiers: an ID for the record provided by the Content Manager, and the URL provided by the data source.

The SDK comes with two different Content Managers:

1. Simple Content Manager
2. Metakit Content Manager

The Simple Content Manager is a very basic implementation of the `IContentManager` class. It returns the same list of media content records, no matter which method is called.

The Metakit Content Manager uses the open source Metakit database to store metadata for each track, allowing basic sorting and filtering by any metadata you register in the Content Manager. It also implements IQueryableContentManager, which provides methods for accessing content records based on Artist, Album, and Genre.

The `CSimpleContentManager` allows you to specify a Metadata object creation function. The `CMetakitContentManager` is written to use a corresponding `CConfigurableMetadata` object.

Content Updates

The Play Manager starts a content update whenever the default event handler gets a New Media Inserted event, or whenever a content refresh is requested. The Play Manager calls the Content Manager in the following sequence:

1. Play Manager calls `MarkRecordsFromDataSourceUnverified()`, passing a data source ID.
1. If the refresh mode is DSR_ONE_PASS_WITH_METADATA, the data source calls `CreateMetadataRecord()` for each media record.
2. The Play Manager calls `AddContentRecords()`, passing a `content_record_update_t` struct filled with content records created and populated by the Data Source.
3. If the refresh mode is DSR_TWO_PASS:

The Content Manager should remove content records from the struct that it already has metadata for, and return the struct containing only content records it needs metadata for.

The data source calls `CreateMetadata()` for each content record.

The Play Manager passes the updated records to AddContentRecords().

4. When the content refresh is complete, the Play Manager calls `DeleteUnverifiedRecordsFromDataSource()`.

Create Metadata Record()

This function is primarily called by a data source during a content update. It should return a pointer to a metadata object that will store the attributes of interest. If the Content Manager doesn't need any metadata, this function can return zero.

Add Content Records()

The default handler calls this function after receiving an EVENT_CONTENT_UPDATE or EVENT_CONTENT_METADATA_UPDATE, passing a pointer to a `content_record_update_t` struct created or modified by a data source during content enumeration.

For playlist content records, simply traverse the array stored in the "playlists" field of the struct and call `AddPlaylistRecord()` for each one. The URL of each record must be deallocated by calling `free()` after the call to `AddPlaylistRecord()`.

For media content records, the procedure is a bit more involved. If this is the first part of a two-pass content update, the media list should be pruned of records that already exist in the database, so they won't be passed back to the data source for metadata retrieval.

For each record in the "media" field of the `content_record_update_t` struct `AddMediaRecord()` should be called. If the `bTwoPass` field of the struct is false or if the media record already exists in the manager (determined by passing TRUE or FALSE back through the `pbAlreadyExists` parameter in the `AddMediaRecord` call) then the entry in the media array can be deleted. Call `free()` on the URL and `Remove()` on the media array to dispose of the record. Otherwise leave the record in the array and go to the next one. In two-pass mode, records left in the media array after the function finishes will be sent to the data source for metadata retrieval.

AddMedia Records()

This function is used to add or update a single media record in the content manager.

If it's a new record to be added, then:

1. Create a media content record to store the incoming data.
2. The media content record should make a copy of the URL, since that memory is managed by the caller.
3. The metadata pointer is the responsibility of the function, since it was created by the content manager's `CreateMetadataFunction`. Its data should be stored somewhere, either merged into the media content record (which is a subclass of IMetadata) or a pointer to the metadata object kept around. The metadata object should be deleted when no longer in use.
4. If `pbAlreadyExists` is non-zero, then set the contents to false.
5. Return a pointer to the new media content record.

If a record with a matching URL already exists in the content manager, then:

1. Merge the metadata from the `media_record_info_t`'s `pMetadata` field with the metadata in the media content record.
2. Delete the passed-in metadata object.
3. If the media content record isn't verified and the `media_record_info_t struct`'s `bVerified` is true, then set the record as verified.
4. If `pbAlreadyExists` is non-zero, then set the contents to true.
5. Return a pointer to the matching media content record.

AddPlaylist Record

This function is used to add or update a single playlist content record in the content manager.

If it's a new record to be added, then:

1. Create a playlist content record to store the incoming data.
2. The playlist content record should make a copy of the URL, since that memory is managed by the caller.
3. Return a pointer to the new playlist content record.

If a record with a matching URL already exists in the content manager, then:

1. If the playlist content record isn't verified and the `playlist_record_info_t struct`'s `bVerified` is true, then set the record as verified.
2. Return a pointer to the matching playlist content record.

Record verification

Content records exist in two states: verified and unverified. A record is considered verified if it has been found on a data source. A record is unverified if it was added in some other manner (e.g., loading the content manager's state from file).

When a content update starts, the play manager will call `MarkRecordsFromDataSourceUnverified()`. The content manager is responsible for marking all records (both media and playlist content records) from the specified data source as unverified.

During a content update, content records passed to AddMediaRecord and AddPlaylistRecord will be marked as verified. If the record already exists in the manager then its status should be set to verified.

After a content update is completed, the play manager will call `DeleteUnverifiedRecordsFromDataSource()`. The content manager should then delete all records (both media and playlist content records) still marked as unverified.

Getting content records

The content manager provides four functions for retrieving records: GetAllMediaRecords, GetMediaRecordsByDataSourceID, GetAllPlaylistRecords, and GetPlaylistRecordsByDataSourceID.

The first argument in each of these functions is a reference to an array of pointers to records. It is the responsibility of the functions to add content to that array. Call PushBack() on the array to add either media content records or playlist content records to the end. The functions shouldn't alter the array in any other way (such as clearing the list or removing records).

IPlaylist

A Playlist object is a sequential collection of Media Content Records accessible using a zero-based index. Its main responsibility is to provide the current Playlist Entry, and set the next and previous playlist entries according to a provided mode.

The client application creates a Playlist object and loads it into the Play Manager. If you want to change the playlist, you can either add or delete entries in the current playlist, or load a new playlist (stored on FAT-formatted media as a file, for example) and set it in the Play Manager.

The main reasons you may want to implement your own Playlist class might be if you don't like the shuffle mechanism in `CSimplePlaylist`, or if you want to add some sort of tracking or caching strategy.

See Playlist in the API reference for more information.

CSimple Playlist Random Sequence

There is always a current track in a playlist. When you call the reshuffle entries method, whatever the current track is, becomes the first entry in the random list. If you wanted to start playback with a random song, and still hear all songs in a playlist, you would have to reshuffle the playlist, move to the next playlist entry, and then reshuffle again (to get the first entry forward in the list).

If you add new entries to a playlist that already exists and is already shuffled, the new entries are inserted at the end of the indexed sequence but inserted randomly into the random list. If the current track is not at the beginning of the random list, it is likely that some of the new entries will get inserted earlier in the random list and will not be heard until the list is reshuffled or repeated.

IPlaylistEntry

A Playlist Entry is a simple interface used to get a pointer to the media content record it represents.

Configuration and Build System

This section describes how to configure and build a target image to download to the Dharma board.

Dharma is designed to be flexible and to maximize reuse of code. To that end, we have designed a custom build and configuration system suited to our needs. This involves organizing the source code into modules and writing custom configuration scripts for each module, in addition to parsing configuration scripts and generating appropriate build trees. Our build system uses perl, GNU make, and GNU toolchain to create an image using a combination of pre-existing and generated makefiles.

Build a Target Image

To create an already-configured image, all you need to do is:

```
cd /c/iobjects/dadio/
make <target>
```

... where <*target*> is the name of an existing configuration. The SDK comes with three pre-built configurations:

- simple
- hdcdcf
- net

These correspond to Demo #1, Demo #2, and Demo #3 described elsewhere in the documentation.

Invoking `make simple` from the "dadio" directory generates the build tree, creates a list of modules, generates make files for each module, and links in a build Makefile. It then switches to the build directory and automatically starts compiling the build. If successful, you should have a finished image ready for downloading in the "builds/simple" directory.

To download the image to the Dharma board, see Download an Image on page 75.

Once you have run the top-level Makefile, you have a build tree for the configuration. To rebuild the configuration, you can `cd` into the build directory and type `make`. If you have updated any of the configuration files, the whole tree will be regenerated. You can clean out existing object dependencies using `make clean` or `make depclean`.

Create a New Target Configuration

When you're creating a new configuration, these are the basic steps:

1. Edit the DCL file in the source directory for each module.
2. In the "config/" tree, create a *<target>*`_modules` file listing all of the modules used in your configuration.
3. In the "config/" tree, create a *<target>*`.mk` file specifying the name of the target, the eCos Build name, and any environment variables or macros to execute during the build process.
4. Add the target name to the `config/targets` file.
5. Perform the build steps as described above.

These steps are described in greater detail in following sections.

SDK Product Directory Files

The base for all files in the sdk is the 'dadio' directory. For this section, references to full paths assume you installed the SDK in "C:\iobjects\dadio", which has a corresponding cygwin path of "/c/iobjects/dadio".

In the 'dadio' directory you will find the following items:

Makefile

The top level product Makefile. This allows you to create and remove build trees for a specified target.

configs/

A directory with files for each target, and a makefile include (targets.mk) which specifies the available targets.

images/

Pre-built utility and demo images that you can burn onto Dharma and experiment with, including documentation describing how to use them.

docs/

Documentation for the SDK.

ecos/

A directory with builds of various kernel configurations.

player/

All the code and libraries shipped with the SDK.

scripts/

Support utilities and scripts used by the build and configuration system. You shouldn't need to modify the contents of this directory.

support/

Various images and utilities for downloading images to and configuring the board.

As a general rule, all filenames that start with an underscore ('_') are generated automatically. Generated files live only in the build tree.

The Role of Make

To streamline the configuration and build process, Dharma uses two completely different Makefiles. The first one, in the base "dadio" directory, creates a build directory, along with creating the second makefile. The second one, in the generated build directory, makes the actual object code. We'll call the first one the "Configuration Make" and the second one the "Build Make."

Configuration Make

The configuration make requires you to specify a target configuration. Target configurations are specified in the `config/` directory, listed in the `target.mk` file in the TARGETS variable. Each configuration should have corresponding `*.mk` and `*_modules` files.

When `make` is invoked at the top level, it searches the `TARGETS` variable to see if it can build the requested build target. If it is found, it attempts to load the file `configs/<target>.mk`, then invokes the DCL script parser (`scripts/parse_dcl.pl`) on the target.

The `*.mk` file uses standard Makefile format, and specifies default parameters for the build, including target executable name, the name of the eCos configuration to link to, and any macros to execute during compilation. See Configuration .mk Files on page 66 for more information about these files.

The `*_modules` file is simply a listing pointing to configuration files for each module that this configuration uses. These secondary configuration files are actually in the source directory, and use a specially created language, Dharma Configuration Language (DCL), to define what happens during the configuration make. See Configuration _modules Files on page 67 for more about the `*_modules` file, and Configuration .dcl Files on page 68 for more about the individual module DCL files.

So the configuration make walks through all of the DCL files specified in the *_modules file, and builds a tree under the builds/ directory. It uses symbolic links to link to the appropriate header files, generates make files for each module, and creates a symbolic link to the main Makefile in the source to be used for the build. Finally, it starts the Build Make, described below.

Output of Configuration Make

The resulting make files for each module are named `_module.mk`. The configuration make also creates a pair of top level files for the build system: `_modules.h` and `_config.mk`.

`_modules.h` is a standard C header with macros defined for each module present in the system. The naming for modules is derived from their type and module name. This allows code to be conditional on whether or not a given module is available.

`_config.mk` is a makefile include that tells the player makefile which modules will be built into the system, which configuration files were used to generate those modules, and provides some other simple information to the main makefile.

Note: You should never need to edit any of these target files. If you want to customize the behavior of the build process, edit the corresponding DCL file for the module, or the `*.mk` file for the configuration, and the configuration make will generate the build files according to your needs. This system prevents you from having to customize your build files every time you need to regenerate the build tree.

Build Make

The Build Make runs within the `builds/<target name>` directory. The same Makefile is used for all builds, but it is symbolically linked to the build tree, and runs from there. When make is invoked in the build tree, the build Makefile first includes the `_config.mk` file. From this file it is able to determine a list of `_module.mk` files to include and which configuration files were used to generate the build tree. If any of the configuration files are newer than the `_config.mk` file, it reruns the DCL parser to update the build tree.

Next, it includes all `_module.mk` files. From the list of compiled sources that is assembled in the `_module.mk` files, it generates a list of object files (OBJS) and a list of dependency files (DEPS). From this point on, the behavior of the makefile depends on the target:

- If a test target was specified ('`make TEST=<path_to_test>`'), the makefile includes the dependency information for that test, builds all object files in the build, and links the test.
- If a test group was specified ('`make fs_tests`'), the makefile recursively invokes itself on each test in the test grouping. When it recurses, it does not check to see if any configuration files have been updated.
- If the `config`, `clean`, or `depclean` target was specified, the makefile does not include any dependency information and will perform the requested operation - '`config`' will force the DCL parser to run on the tree, '`clean`' will delete all object files, '`depclean`' will delete all object files and all dependency files.
- If no target is specified, the makefile includes all dependency information and builds the default target.

When a target is being built, `make` looks to make sure all the object files listed in OBJS are up to date. On a freshly generated tree, no object files or dependencies will exist. This forces make to search for sources and rebuild them. Since all source files reside in the `player/` directory, we use `VPATH` to force make to search that directory. The result is that the source is allowed to reside in the `player/` directory while object files and dependencies are placed in the build directory.

GCC has extensions that allow dependency generation and compilation in a single pass. As a result of this, when a file is being compiled a new dependency file will automatically be generated for it. GCC 2.9x has a bug that causes dependency files to be generated without path information for the target, but this bug seems to have been resolved in GCC 3.x. As such, the makefile will attempt to determine the current GCC version and compensate for this bug, at a small performance decrease.

After all object files and dependency files are up to date, the makefile will attempt to link a player with the eCos libraries. This is the final step in the build. After the build is complete, you may notice a file named "`_build_list`" in the build directory. This file lists target files that were generated during the build process. This can be useful when writing scripts to package builds, or when you are creating multiple images, as is usually the case when building tests.

Configuration System

As described earlier, the configuration system uses three types of files that you must edit to create a configuration:

- *<target>*.mk
- *<target>*_modules
- *.dcl

The first two define a target configuration, listing all included modules and setting up the environment. The last is a configuration file associated with each individual module, and must be set up in the source tree. This section describes the format of each of these files.

Configuration .mk Files

The target makefile include specifies a handful of parameters and extra configuration that is independent of any given module. This file uses standard makefile syntax, and is included into other makefiles during the build process. Although many options can be configured in this file, the standard options are as follows:

Required variables:	
`ECOS_BUILD_NAME`	The name of the eCos build to link against:

	■ `usb-ram` – USB-enabled image.
`MAIN_MODULE`	The path to the module that `cyg_user_start()` is defined in. This is the entry point from the eCos bootstrap environment. This module is excluded from the build when generating tests, since the `cyg_user_start()` symbol would be defined again during the link step.
`TARGET_FILE_NAME`	The name of the executable image to generate from this build.
Optional variables:	
`COMPILER_FLAGS`	Additional flags to pass to the compiler.
Optional macros:	
`pre_build_step`	A macro to run prior to building sources. This macro is always executed, even if the target is up to date.
`pre_compile_step`	A macro to run prior to compiling a file.
`post_compile_step`	A macro to run after compiling a file if the compilation succeeded.
`pre_link_step`	A macro to run prior to linking the sources. This macro is only executed if a relink is required.
`post_link_step`	A macro to run after linking the sources. This macro is only executed if a relink is required.

Build system extensions

A brief note was made earlier about `pre_build_step`, `pre_link_step`, `post_link_step`, `pre_compile_step`, and `post_compile_step`. These macros can optionally be defined to perform steps at certain parts of the build sequence. The syntax of these macros is best left to make documentation, such as that available for GNU make:

http://www.gnu.org/manual/make-3.79.1/html_mono/make.html

Configuration _modules Files

Module lists associate a target with a list of modules and module configurations. A module list is simply a flat text file with 2 columns, delimited by whitespace. The left column indicates the path to the module that should be built in; the right column indicates the name of the configuration to use when generating a build tree for this target. If the specified configuration script is not found, the configuration "default.dcl" will be searched for. If that configuration script is not found, then the module will be excluded from the build.

It is possible to comment out a line in a module list by starting the line with the '#' character. When the list is processed, all text after the '#' character will be ignored.

Path Column

The left column of this file is a search path for the module, and begins in the "player/" directory. There is a special convention used for versioning in this path: if the last directory in the path is either "current" or matches "v[0-9]_[0-9]" (representing a version number in the format "v#_#" where each "#" is a single digit), then the search path includes both the specified directory and its parent. See Module Versioning on page 70 for more information about this system.

File Column

The right column of this file is the name of the .dcl file to load. ".dcl" is automatically appended to this name. If this file is not found in the search path in the left column, the parser looks for a file called "default.dcl". If this file doesn't exist, the module will be omitted.

This system allows you to create different configurations for each module in the source tree, and specify which one to build in the target configuration.

Configuration .dcl Files

Configuration files are written in DCL. DCL is a very simple language. commented lines begin with the # character. The text for any directive must fit on a single line. Blank lines are ignored.

There are a few main keywords in DCL:

`name <module_name>`	Specifies a name for this module. Other modules can then explicitly use this module as a dependency using the name/type pair.
`type <module_type>`	Specifies a type for this module. The currently allowed types are: `storage` Storage (block) device `net` Network device `dev` Generic device `input` Input wrapper `output` Output wrapper `filter` Filter/codec wrapper `fs` Filesystem `playlist` Playlist `other` Unspecified type

`requires <packagename>`	Specifies that this module needs the package <*packagename*> to be built in order to function. <*packagename*> can either be a specific package, or a wildcard. An example is that a fat module (of type `fs`) would require a module of type `any_storage` to be built in. An `iso9660` module would require a module of type `ata_storage` be built (since the ata module includes the atapi support for CD drives).
`include <dcl list>`	Include the specified dcl file(s) into the current one. This allows a single module to have a base configuration file with features common to all configurations of that module, and for individual configurations to be limited to their custom information.
`build_flags <build options>`	A string to be added to the build line for `gcc` and `g++`.
`link_flags <link options>`	A string to be added to the final link step.
`tests <file list>`	Specifies test targets that can be built for this module.
`export <file list>`	Specifies the headers in this module that should be exported to the build tree. Headers not listed in the export list will not be accessible to other modules.
`compile <file list>`	Specifies the source files in this module that must be compiled for the module to function.
`link <file list>`	Specifies objects or libraries to be linked at the final link step.
`arch <object list>`	Specifies a list of object files to archive into a library.

`header <headername> <start | end>`	Specifies the start (or end) of a block of text to be generated and deposited into the header file specified by <headername>. This header exists in the build tree under the directory for this module.
	An example of this would be as follows:
	`header hw_cfg.h start`
	`#include <cyg/hal/hal_edb7xxx.h>`
	`#define HW_TARGET_NAME "dharma"`
	`#define HW_GPIO PDDR`
	`#define HW_RESET 0x04`
	`header hw_cfg.h end`
	This would generate the header hw_cfg.h in the build tree, with the following contents:

```
// generated file, do not edit !!
#ifndef __HW_CFG_H__
#define __HW_CFG_H__

#include <cyg/hal/hal_edb7xxx.h>
#define HW_TARGET_NAME "dharma"
#define HW_GPIO PDDR
#define HW_RESET 0x04

#endif // __HW_CFG_H__
```

The most basic DCL file must have a name and a type specified, but does not need to use any other directives. The simplest sensible DCL file would have at least one file that it compiled, or at least one header that it exported.

Module Versioning

One of the problems posed by the GNU make system is that dependency (.d) and object (.o) files in the build tree must have the exact same path within the build tree as their corresponding source file within the source tree. However, since modules refer to headers in other modules, it is less than ideal to have version numbers in the path to the headers.

Accommodating differing versions of different modules is one of the main reasons the configuration system is set up the way it is. The DCL language was set up to create a symbolic link to the correct header file in a non-version-specific directory so that other modules may link to it.

During compilation, however, `make` de-references the symbolic links, and builds the actual modules in directories that match the path in the source tree.

What This Means For You

Versioning is completely optional. You can omit version directories in your source tree, or move all of the source files out of each version directory into its parent directory, and you will not need to change any of the configurations.

If you want to use versioning, you must use the following "magic" directory names:

- current – represents the latest development version.
- v# – "#" represents a version number. For example, "v3".
- v#_# – represents a version with a sub-version. For example, "v3_2".

If any of the above are present in the `<target>_modules` path, the DCL parser will put a symbolic link to the header file in the parent directory of the listed directory.

If none of the above are in the path, the path will be used exactly as listed in the `<target>_modules` file.

Example

Say you have the following module:

```
player/playlist/simpleplaylist/v1_0
```

... that has one header and one source file:

```
player/playlist/simpleplaylist/v1_0/include/SimplePlayli
st.h
player/playlist/simpleplaylist/v1_0/src/SimplePlaylist.c
pp
```

When the DCL parser generates the build tree, it will generate a symbolic link as follows:

```
builds/simple/playlist/simpleplaylist/SimplePlaylist.h -
>
../../player/playlist/simpleplaylist/v1_0/include/Simple
Playlist.h
```

However, when the source file is compiled, the output will be in:

```
builds/simple/playlist/simpleplaylist/v1_0/src/SimplePla
ylist.o
```

... since compiling uses `make`, and `make` requires that the path to the source file mirror the path to the object file.

Dharma Operation Guide

This section describes the mechanics of using the Dharma board, including:

- Installing a boot loader
- Building an image
- Downloading an image to the Dharma board
- Debugging an image
- Burning an image to flash
- Debugging an image in flash

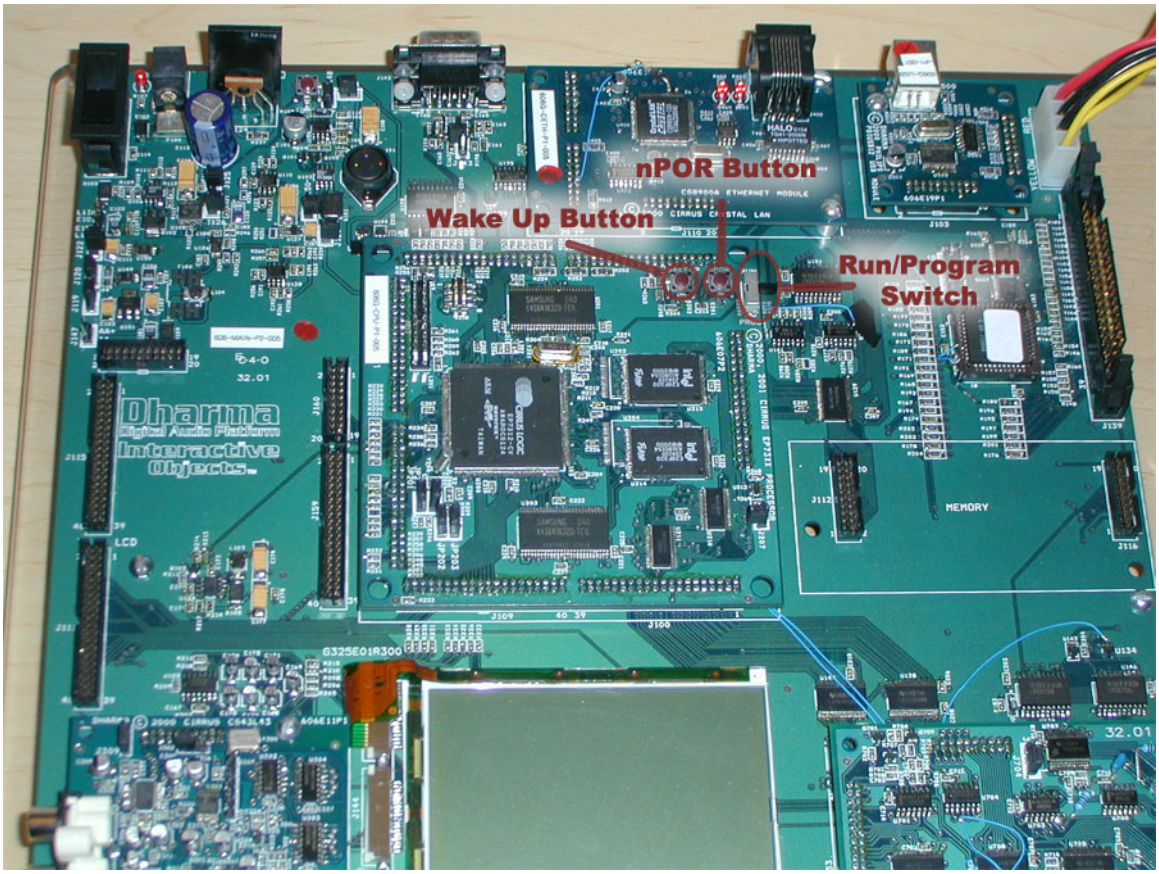
Installation of the software was discussed in the Quick Start Guide on page 9, and is also available on the CD in the Readme.txt file.

To perform the steps in this section, you must connect the Dharma board to a COM port on your PC using a null modem cable.

Install the Boot Loader

Running the image requires the use of a gdb stub or a bootstrap environment on the board. Interactive Objects has created a combination gdb stub/bootstrap environment for Dharma using RedBoot™, a product from RedHat. The Dharma board you received should have a Redboot stub on it already. You can verify this by connecting the Dharma board, starting up a term program, and configuring it for your serial port and setting it to 115200 8N1. Next, verify that the run/program switch is in the "run" position, power on the dharma board, wait approximately 1 second, and hit the wakeup button (located on the processor mainboard). You should see some status information regarding the flash, a banner, and a prompt:

```
RedBoot>
```



If you do not see the RedBoot> prompt, you may need to burn a redboot stub to the device. This can be done as follows:

1. Power on the board.
2. Switch the board into "program" mode.
3. Hit the nPOR button on the board to induce a power-on-reset.
4. On the desktop side, open a bash shell, and cd into the sdk directory:
   ```
   cd /c/iobjects/dadio
   ```
5. On the desktop side, start the download application. If your null modem cable is plugged into COM1, the syntax is as follows:
   ```
   support/downloader.exe images/redboot_stub.bin
   ```
6. If you are using a COM port other than 1, you need to specify the port number (X):
   ```
   support/downloader.exe -pX images/redboot_stub.bin
   ```
7. Once this command has been issued, you should see the text:
8. Waiting for the board to wakeup...

9. At this point you can hit the WAKEUP button on the Dharma processor board, and the download sequence should start.
10. When the download has completed, switch the board back into the "run" mode, start your term program as described above, then hit nPOR, wait one second, and hit WAKEUP. you should see a RedBoot> prompt at this point.

RedBoot provides an interactive shell with flash management and image handling capabilities. Discussing the exact feature set is beyond the scope of this document, but is well covered at the following location:

http://sources.redhat.com/ecos/docs-latest/redboot/redboot.html

For term programs, we highly recommend TeraTerm 2.3 (Win95, 98, NT, 2k compatable), available from the following location:

http://hp.vector.co.jp/authors/VA002416/teraterm.html

Create an Image

1. Start a bash shell and `cd` into your installation directory:

```
cd /c/iobjects/dadio
```

2. Determine which target you want to build by looking at the available targets in configs/targets.mk, then build it. This generates the build tree, and creates a player image:

```
make simple
```

3. After the build is completed, you can cd into the build directory to find the generated image:

```
cd builds/simple
ls -l simple.exe
```

Note: Since the build tree was created in step #2, you can build directly from the build directory and save a little time from this point on:

```
cd /c/iobjects/dadio/builds/simple
make
```

The build system is described more fully in Configuration and Build System on page 61.

Download an Image

Now that you have created an image, you can transfer this image to the Dharma board and start debugging it, or simply transfer and execute it.

There are two methods for downloading an image to the device:

1. Using `gdb` (the debugger) to download the image into memory.
2. Using RedBoot to download the image to flash memory.

These options are described in the following sections.

Download and Debug the Image

Once a gdb stub has been installed on the board, it is possible to use `gdb` to load and debug an image. Since our image has been cross compiled, we need to use a cross debugger. Assuming you have built the target 'simple', you can perform the following steps to debug the image:

1. Open a bash shell and `cd` into the build directory:

   ```
   cd /c/iobjects/dadio/builds/simple
   ```

2. Prepare the Dharma(tm) board by powering it on, waiting one second, then hitting the WAKEUP button.
3. At the shell, load `gdb`:

   ```
   arm-elf-gdb simple.exe
   ```

   This should bring you to the `(gdb)` prompt.

4. Configure `gdb` for the remote target:

   ```
   (gdb) set height 0
   (gdb) set remotebaud 115200
   (gdb) target remote com1
   ```

The **height** variable controls the pager – setting it to 0 disables the pager. The **remotebaud** variable controls the baud rate of the serial target. **target remote com1** indicates that `gdb` should attach to a remote target on `com1` – if you are using a different serial port, you should specify `com`*X* where *X* is your serial port number.

At this point `gdb` will try to connect to the Dharma board. If the Dharma board is not properly connected or does not have a debug stub on it, `gdb` on the desktop side will hang. You can interrupt `gdb` by pressing CTRL-C, and then type "quit" to exit the console. After the board has been properly connected and set up, repeat steps 3-4 to attach the debugger.

Note: `gdb` will look for and execute a startup script if one is available. This allows highly repetitive commands, such as the three listed above, to be executed automatically. To use a startup script, create a file named `gdb.ini` in the same directory as your image, and put the commands in that file exactly as you would type them at the `gdb` prompt.

---

5. Transfer the image from the host to the device:

```
(gdb) load
```

This may take a while, depending on the size of the image. `gdb` will provide feedback describing which section is currently transfering. When the transfer is complete, it will return to the `(gdb)` prompt.

6. Set any breakpoints you may have:

```
(gdb) break cyg_user_start
(gdb) break main/demos/simple/src/main.cpp:84
```

`gdb` will come back with the line number and file associated with the breakpoint and a breakpoint number. You can disable or delete this breakpoint using the **disable** and **delete** commands.

7. Execute the image:

```
(gdb) continue
```

Most of our demo applications provide some level of feedback over the serial port. Most images will attempt to draw to the lcd, or access peripherals on the board. This should give you feedback about the execution of the application. If you have breakpoints set, `gdb` will return you to the `(gdb)` prompt and indicate the breakpoint that has been hit.

Complete documentation regarding `gdb` syntax and features is available at:

http://www.gnu.org/manual/gdb-4.17/gdb.html

Download an Image to Flash

For demos and repetitive testing, you may want to burn the image to the flash memory rather than transfer it over a serial connection every time. RedBoot provides flash management and can be used to do this as follows:

1. Start a bash shell, and `cd` to the build directory:

   ```
   cd /c/iobjects/dadio/builds/simple
   ```

2. Convert your image to the srec format using the following syntax:

   ```
   arm-elf-objcopy -O srec simple.exe simple.srec
   ```

3. This will generate the file "simple.srec" in the current directory.

   Note: The existing targets do this automatically after linking, by setting a `post_link_step` in the build system to automatically perform this copy step. Refer to the Configuration .mk Files on page 66 for more information.

4. Start your terminal program and wake up the board. Your terminal program must support xmodem, ymodem, or zmodem for the transfer.
5. At the RedBoot prompt, initiate a transfer on the device side:

   ```
   RedBoot> load -m xmodem -b 0x20000 simple
   ```

   This tells redboot to start an xmodem transfer, and to place the transfered image at 0x20000 in RAM. The image is given a name, "simple", but this is not used at this point.

6. Initate a transfer on the host side. If you are using TeraTerm, this can be done by going to File-> Transfer-> XMODEM-> Send.... Send the previously created simple.srec image to the device
7. When the transfer completes, RedBoot will print some gathered information about the image. For example:

```
Entry Point: 0x00020040, address range:
0x00020000-0x000ae1c4, length: 0x0008e1c4
```

   You will need this information to burn this image to the flash.

8. Tell RedBoot to burn the image to the flash using the following command:

```
RedBoot> fis create -b 0x20000 -l 0x8e1c4 -e
0x20040 -r 0x20000 simple
```

   "fis create" tells redboot we want to create a new image in the Flash Image System (FIS).

   "-b 0x20000" indicates the base address in RAM of the image to burn, and corresponds directly to the "-b 0x20000" argument we used in the "load" operation.

   "-l 0x8e1c4" tells RedBoot the length of the image – this is taken from the info line in step 6.

   "-e 0x20040" tells RedBoot the entry point to the image – again, taken from the info line in step 6.

   "-r 0x20000" indicates the address to load the image to. This is important since all of our images will need to be loaded to this address.

   "simple" assigns a name to this image.

   At this point, RedBoot will begin burning the flash with the image. when it is done, it will return to the RedBoot prompt.

9. Verify the image was burned correctly:

```
RedBoot> fis list
```

   The listing of the flash contents should have an entry for "simple".

10. Load and execute the image. It is possible to execute the image already in RAM (from step 4) without reloading it from the FIS. However, to properly verify the image, we should reset the board and reload the image:

```
RedBoot> reset
```

The board should reset and return to the RedBoot> prompt.

```
RedBoot> fis load simple
RedBoot> go
```

This should start executing the image.

Debug an Image in Flash

It is possible to debug a flash-resident image. This is useful when an image has a condition that is difficult to trace or requires multiple runs. To debug a flash-resident image, first make sure that the image burned to flash and the image on your host are the same build. Follow the steps in the previous section to burn your image to flash, and then proceed as follows:

1. Reset the board and load the image:

```
RedBoot> reset
RedBoot> fis load simple
```

2. Close the terminal program and start gdb from your shell:

```
cd /c/iobjects/dadio/builds/simple
arm-elf-gdb dadio.exe
```

3. At the gdb prompt, execute the following commands:

```
(gdb) set height 0
(gdb) set remotebaud 115200
(gdb) target remote com1
```

At this point, gdb should attach to the remote device. the trick is to point gdb at the image to be debugged rather than the debug stub itself:

```
(gdb) set $pc=0x20040
```

This modifies the program counter to point to the entry point of the image we loaded from flash.

At this point all the standard gdb commands are available, and the image can be debugged as normal. To debug the image again, the board will have to be reset and the sequence 1-3 repeated.

USB Test Application

The Dharma SDK comes with a test application that demonstrates USB bulk storage functionality. This test program works for the hard drive and for compact flash cards, but must be compiled specifically for each target. The bulk storage driver has been configured for single Logical Unit Number (LUN) only. The purpose of this test app is to allow developers to transfer content via USB from the host to the device. The driver has not been fully tested against all hosts, and may not be suitable for shipping products.

Althought USB bulk storage is a standardized protocol, not all operating systems come with driver support for it. Internally we have only used Windows 2000 Professional with this driver, although it should work with Windows ME and XP.

Loading the pre-built USB test app

The RedBoot stub image should contain two versions of the USB test app: one for transfering files to a compact flash card, and one for transfering files to the hard drive. The hard drive based image assumes the hard drive is jumpered as a slave. Before loading the application, make sure the usb module is present on the board, and the USB cable is unplugged from the device.

To load the pre-built USB test app, refer to the section titled Download an Image to Flash on page 77. At the RedBoot> prompt, enter the following command:

```
RedBoot> fis list
```

This should print a list of available images in the flash. To load the hard drive test app, enter the following command:

```
RedBoot> fis load usb_hd
```

Or, for the compact flash test app, use the following:

```
RedBoot> fis load usb_cf
```

To start executing the device side of the code, use the "go" command:

```
RedBoot> go
```

At this point the device will be ready to initialize itself on the USB bus. Make sure the host end of your usb cable is plugged into your computer, and connect the device end into the Dharma board. The host should enumerate the hard drive, which may take some time depending on the drive capacity. The device should show up as a "Removable Disk" under "My Computer" on Windows machines.

When you are done transferring files to the device, be sure to stop the device and unplug it prior to resetting the board.

Building your own USB test app

To build your own USB test app, you must configure a product build tree to link against the USB enabled kernel. You can modify an existing product configuration, or generate a new one specifically for USB enabled software. This step simply requires modifying the ECOS_BUILD_NAME value in the file `configs/<target>.mk` to read

```
ECOS_BUILD_NAME := usb-ram
```

and rebuilding your image against the new kernel according to the documentation in the section Create an Image on page 74. Multiple variations of the ATA driver have been packaged, allowing USB to access various physical layouts. In the event that an ATA configuration supporting both the hard drive and compact flash interfaces is used, the USB test will default to the hard drive interface.

Once you have recreated the build tree against the USB enabled kernel, rebuild the USB test app by going to the build directory:

```
cd builds/simple
```

And issuing the following "make" command:

```
make ata_storage_tests
```

This will build all the available ATA tests, one of which is the usb test. The tests will be put in the ATA module directory under the build directory, `devs/storage/ata/<version>/tests/*`. The generated image can be loaded via gdb, or transfered through RedBoot and burned to the device.

Hardware Information

The documentation for the Dharma Board is all shipped in PDF format. If you need a viewer for this, install the free Acrobat Reader software included on this CD-ROM.

You can find all hardware documentation in the Documentation directory on the CD-ROM.