Dadio Software Development Kit

Architectural Overview

<1st Draft> -

Interactive Objects, Inc.

October 2001

Author: John Locke

Table of Contents

Introduction

The Dadio™ Operating System (also referred to as Dadio™ OS or simply Dadio™) provides a diverse and well defined set of APIs and components necessary for commercial digital audio playback. This release of the Dadio OS, as an external SDK for external parties to develop with, is the first source release of Dadio in any form, and, naturally, is our best. Dadio 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 Dadio 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 Dadio Operating System.
- Player Tutorial – How to use the Dadio 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 Dadio 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 Dadio SDK CD-ROM, and includes the GCC compiler, the GNU Make tool, a development environment, the `bash` shell, and several other UNIX tools.

Dadio

A C++-based operating system that provides embedded objects that run on the Cirrus 7312, 7409, and 9312 chips.

DCL

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

Dharma

A hardware board used for developing Dadio objects. It consists of a motherboard, CPU, and many peripherals that may be plugged in or removed to test various configurations.

PCM

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

Quick Start Guide

This section describes how to get an image up and running on the Dharma board.

The first step is to install the SDK. The SDK installer is an executable that first installs the Cygwin environment, and then installs the files that make up the Dadio SDK.

For more information about Cygwin, including how to open a `bash` shell, see http://cygwin.com.

The basic process to creating and loading a Dadio image is:

1. Build the image
2. Transfer the image to the device.

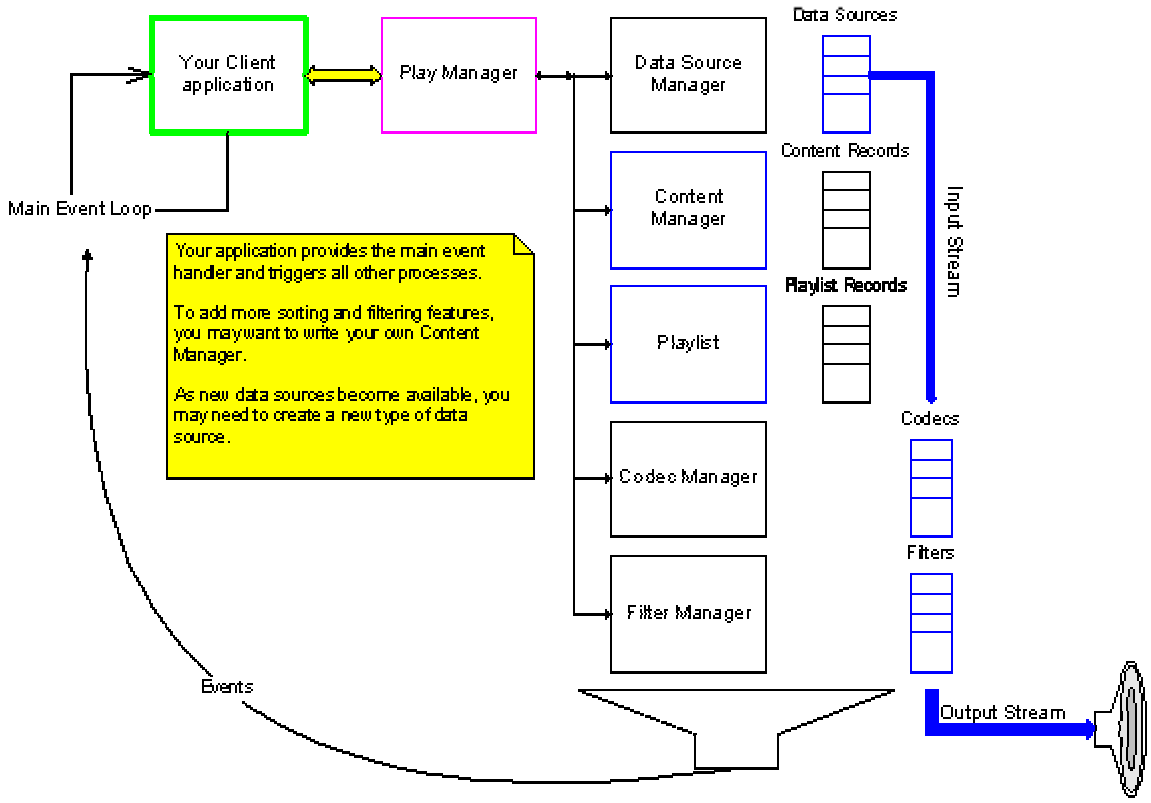
Architectural Overview

The Dadio 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 Dadio 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 Dadio 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. Follow the links for more detailed information about each of these abstract classes.

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. Your application must call the `CPlayManager::AddDataSource()` method, passing a Data Source object for each data source available in 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 play list 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 play list. 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.

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 file names for each track. These file names 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 file names 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), and may only need to update the metadata for specific tracks.

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 for more.

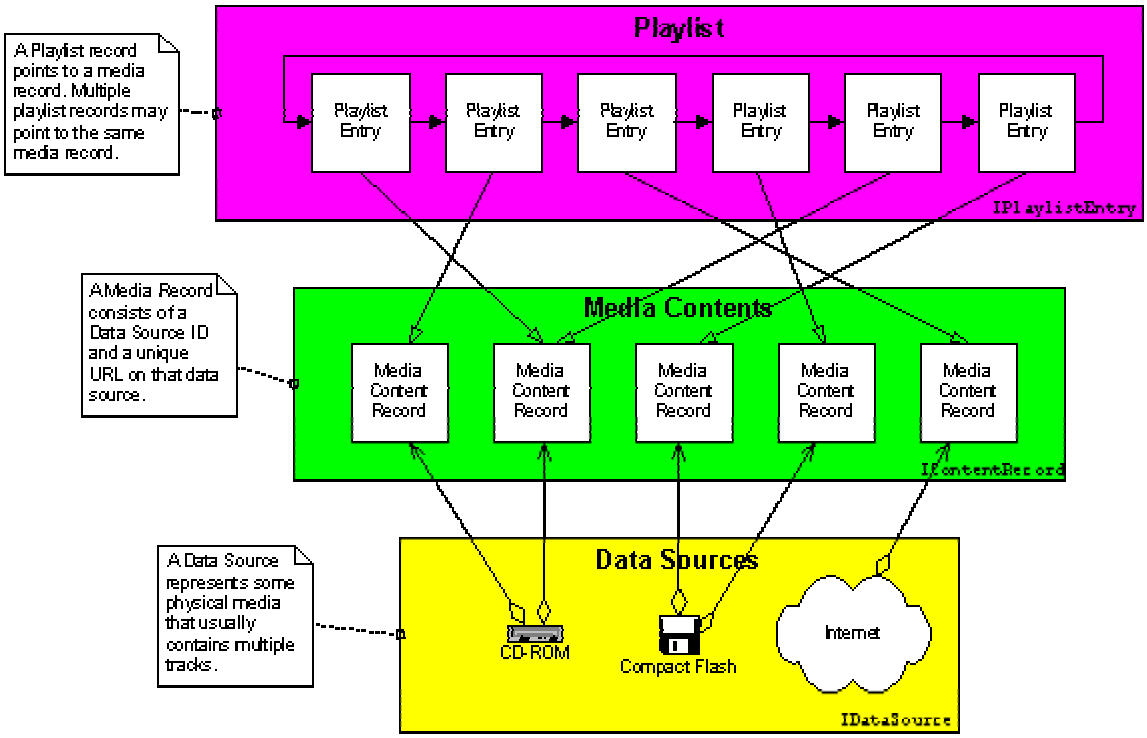
Working with Playlists

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). 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:

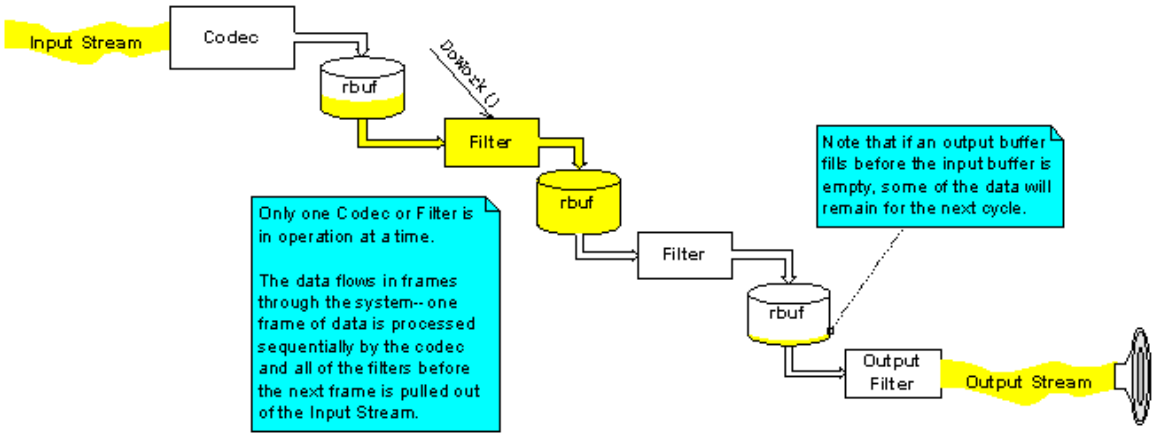
- NORMAL: Play the entire list from start to finish, and then stop.
- RANDOM: Play each track once, in random order.
- REPEAT_ALL: Play the entire list from start to finish, and then restart.
- REPEAT_RANDOM: Play a random track, and then play another random track, until the user intervenes…
- REPEAT_TRACK: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

Controlling the Media Player

The other half of the Dadio 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
- Content Manager
- 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.

Implementing and using these methods is completely up to you. 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).

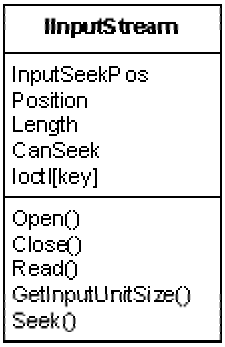
System Components

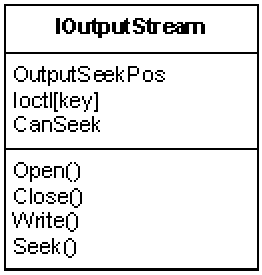
This section describes each of the major components of the Dadio SDK in greater detail.

Data Streams

Dadio 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.

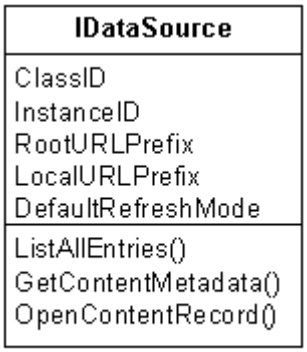
The following output streams are available:

- `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 Dadio SDK includes Data Source objects for the following types of devices:

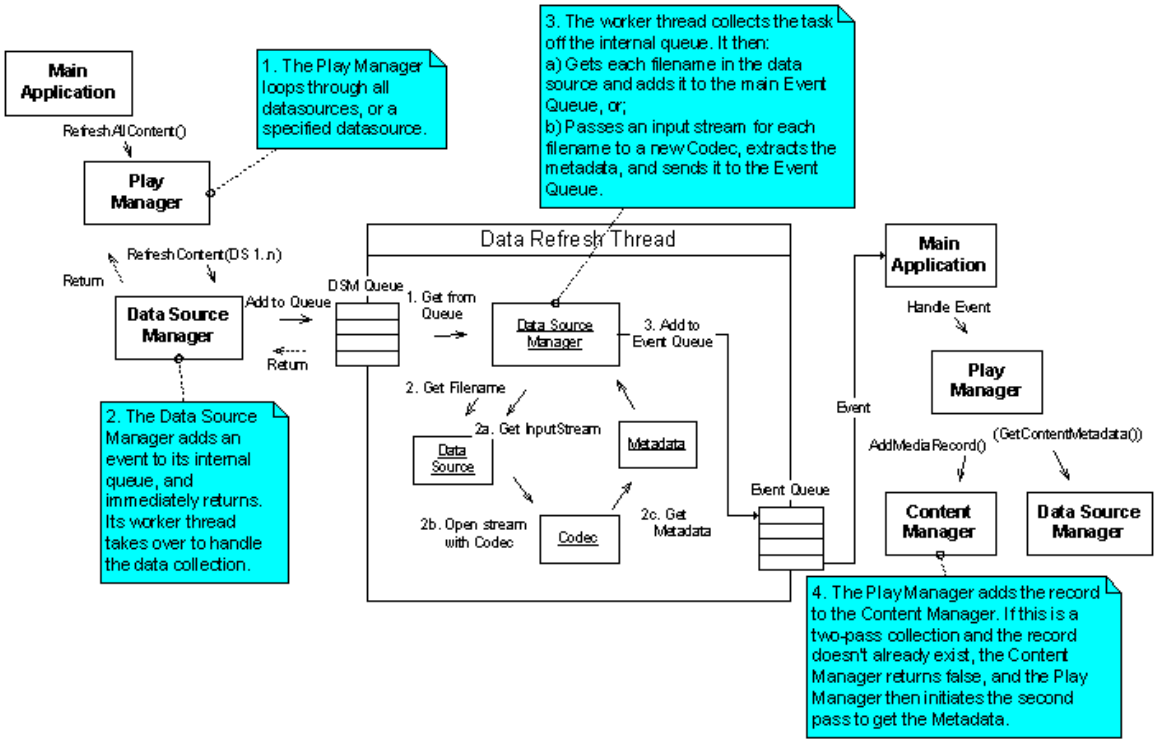
- CD Player
- FAT Device
- Serial Input
- Network Interface

Data Source Manager

The Data Source Manager handles registration of all of the data sources, and provides a data source object of the appropriate type when requested.

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.



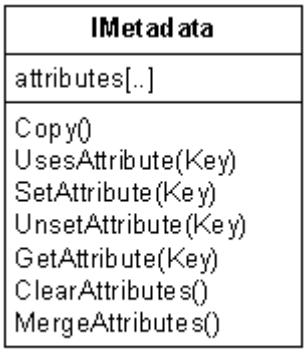
Depending on the type of content enumeration being used, the worker thread in the Data Source Manager collects a list of filenames on the data source (single-pass), a Metadata object for each track on the data source (single-pass with metadata), or a Metadata object for a specified track (second pass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

Metadata

In the Dadio 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 must define a default Metadata class for the Media Player, and the Content Manager must 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 object 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 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.

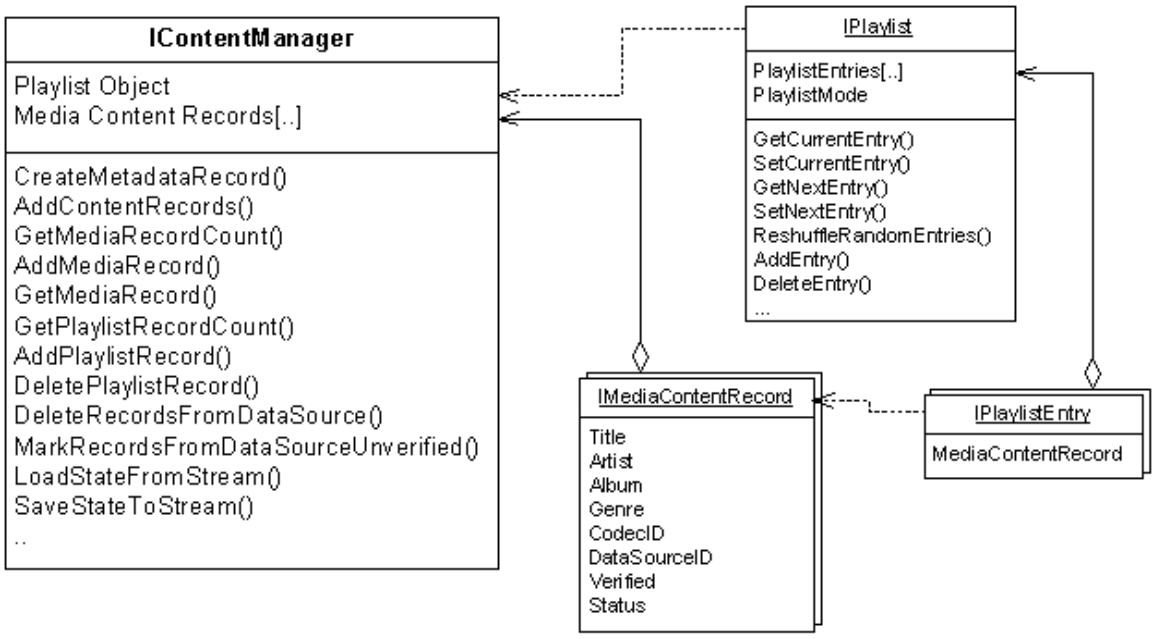
When your application has finished using the Metadata object from the Media Player, it can pass the event to the default event handler for destruction. If you do not use this event handler, you will have to destroy the object yourself.

See IMetadata 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 playlists and individual tracks. Unlike the other objects provided by the SDK, content objects are persistable, and should provide a method of serializing their state in all but the simplest media playing devices.

The Content Manager should always contain a single Playlist object, and 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 loading and saving Playlists and 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 extends `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, apart from saving the current state of the Content Manager. See Playlist Format Manager for more about saving Playlists.

Media Content Records

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

See Content Manager class for more information about Content Managers.

Playlists

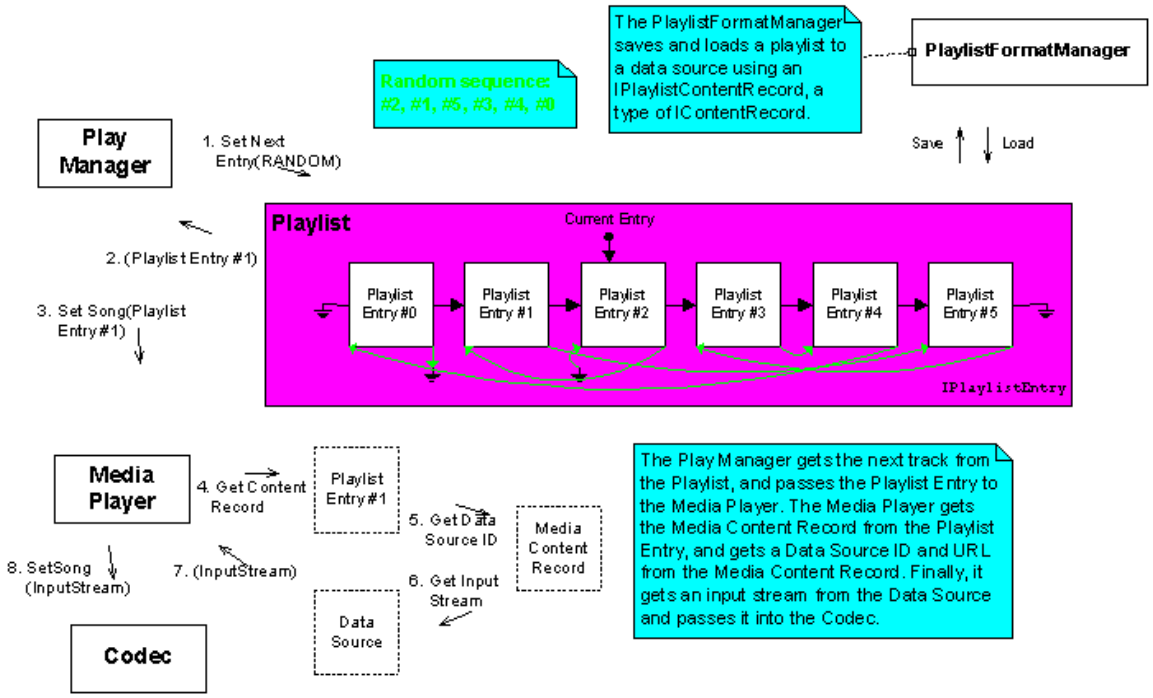
A Playlist is an object representing a sequence of tracks. There are actually two sequences in each playlist: the normal indexed sequence, and the random sequence. When the Play Manager requests the next track from the Playlist object, it specifies a Playlist Mode that indicates whether to get the next 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 is provided as part of the SDK to allow you to store and retrieve playlists. To store a playlist, you need to provide an output stream. To load the playlist, the Playlist Format Manager takes a Playlist Content Record, which extends the IContentRecord class and points to a URL on a data source. The Playlist Format Manager serializes the playlist into a specified format.

Supported formats include:

- M3U
- IO format (*.dpl).

You can add other formats to the system. The demo applications do not make use of the Playlist Format Manager.

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 adds it to the Data Source Manager, and starts a thread to enumerate the content.
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 Dadio 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 Dadio(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 Dadio 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 Dadio(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 Dadio(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.

Files

```
./player/codec/codecmanager/include/CodecManager.h
./player/codec/codecmanager/src/CodecManager.cpp
```

Resources

The Codec Manager uses the following modules:

```
./player/codec/common
./player/util/registry
```

Error codes

The Codec Manager does not use a standardized error interface.

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.