Dadio Software Development Kit

Dadio Object Tutorial

<1st Draft> -

Interactive Objects, Inc.

October 2001

Author: John Locke

Table of Contents

Creating Your Media Player

This section describes how to build the main client application for your media player, as well as details about extending the provided abstract classes.

Global Definitions

This section describes definitions, macros, and types used globally in the Dadio 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

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.

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.

TCHAR

The TCHAR data type is defined to provide you with the ability to compile both single-byte and double-byte character encoding from the same source. If you define the constant UNICODE in your makefile, all TCHAR variables will be defined as wide strings, containing double-byte characters, and using the appropriate functions. Otherwise, all TCHAR variables will be strings composed of single-byte characters.

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.

Demo Applications

The Dadio SDK includes two 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. There are icons for Play, Pause, and Stop. All other functions display a simple text message to indicate their status.

The main play mode shows information about the currently playing track, including metadata if available, the time into the track, the data source, and the demo title.

After certain buttons are pressed, the display changes to show different messages, and then returns to the main play mode.

Demo #1

Local playback from hard drive and CD

The hard drive/CD 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 Dadio SDK. This demo focuses on player controls and the user interface.

Features

Demo #1 supports the following features:

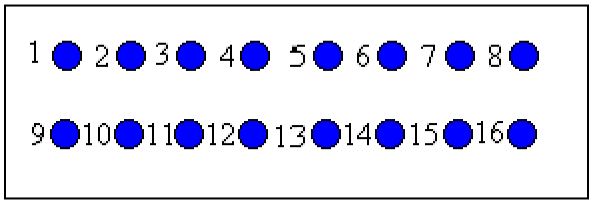
- **Play files from a hard drive or CD.** Access a drive or CD and create a file database that allows the media player to navigate and play tracks from the drive.
- **Simple UI.** Display player functions; File name or Track ID and Artist name and album name for the playing track, Volume level, Bass level, Treble level, Play-pause-stop mode, play list mode, Time code for track, Hard drive or CD 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).
- **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/ Save device state/ Query drive-refresh content DB/ Playlist mode.

- **Playlist mode.** Support normal, random, repeat, random repeat 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
1	Play / Pause	The play/pause button toggles between playing the next cued track and pausing currently playing track. The UI displays the play, pause, and stop icons on the main play mode screen depending on what state the player is in.
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. The

		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 11 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 LEVEL" and a numerical level number for steps 1-10. Level 11 displays "MAX" to indicate that the demo is at Max volume. 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 11 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 LEVEL" and a numerical level number for steps 1-10. Level 11 displays "MAX" to indicate that the demo is at Max bass. 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 11 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 LEVEL" and a numerical level number for steps 1-10. Level 11 displays "MAX" to indicate that the demo is at Max treble. Pressing and holding the treble buttons will increase/decrease a step every second until the maximum or minimum level is reached.
6	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 will display "SAVING SETTINGS" on the screen for 3 seconds then revert to the standard play mode screen.
7	Not used	
8	Playlist mode	The playlist mode button modifies the behavior of the current playlist. There are four modes: Random, Repeat, Random Repeat, and

		Normal. When you press this button, the UI will display the words "PLAY LIST" and the next mode for three seconds. Each time you press the button, the next mode appears on the UI, rolling through the modes in this sequence: RANDOM=>REPEAT=>RANDOM REPEAT=>NORMAL. After three seconds of idle the UI returns to the main play mode display, the device state is saved and the playlist mode is changed.
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 "PLAY" with "STOP".
10	Next track	The next track button navigates to the next track in the system. If the device is at the end of the play list it will navigate back to the start of the play list. The UI will display "NEXT TRACK" for three seconds then revert to the main play mode.
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	Pressing this button causes the unit to spider all available drives and create a content database of

	content DB	all tracks found on all media types active. The UI will display "GETTING TRACK INFORMATION" for three seconds on the UI status line, and then start playing the first track in the database.
15	Not used	
16	Not used	

Demo #2

Local Playback with Metadata Sorting

The hard drive #2 demo builds on demo #1. 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. The infrastructure that supports this functionality is a Metakit embedded database.

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, Hard drive or CD 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).
- **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 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 #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.

	Album	and build a playlist of all tracks from that album on all available media. The demo then plays the first track in the list. Pushing the button a second time starts the process again with the next available album.
6	Generate List By Genre	Pressing the Sort by Genre button causes the demo to load the first available track, extract the genre, and build a playlist of all tracks from that genre on all available media. The demo then plays the first track in the list. Pushing the button a second time starts the process again with the next available genre.
7	Save Playlist	The save Playlist button saves the songs of the content DB into a Playlist file. This functionality illustrates Playlist creation. It does not provide any playlist management due to UI and time constraints. The UI displays "SEARCHING FOR PLAYLIST" on the UI status line while saving the playlist. The demo saves a playlist on the hard drive in the Dadio proprietary *.dpl format.
8	Playlist mode	The playlist mode button modifies the behavior of the current playlist. There are four modes: Random, Repeat, Random Repeat, and Normal. When you press this button, the UI will display the words "PLAY LIST"

		and the next mode for three seconds. Each time you press the button, the next mode appears on the UI, rolling through the modes in this sequence: RANDOM=>REPEAT=>RANDOM REPEAT=>NORMAL. After three seconds of idle the UI returns to the main play mode display, the device state is saved and the playlist mode is changed.
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 "PLAY" with "STOP".
10	Next track	The next track button navigates to the next track in the system. If the device is at the end of the play list it will navigate back to the start of the play list. The UI will display "NEXT TRACK" for three seconds then revert to the main play mode.
11	Volume Down	See Volume Up (button #3).
12	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 will display "SAVING SETTINGS" on the screen for 3 seconds then revert to the standard play mode screen.
13	Not used	

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. The UI will display "GETTING TRACK INFORMATION" for three seconds on the UI status line, and then start playing the first track in the database.
15	Load Playlist	The Load Playlist button loads the first Playlist found on the drive in Dadio (*.dpl) or M3U format and starts playing the first track found in the Playlist. The UI displays "LOADING PLAYLIST" on the UI status line.
16	Not used	

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 hard drive #1 demo. The data source consists of five Shoutcast URL's 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	Same as Demo #1.

	track	
3	Volume Up	Same as Demo #1.
4	Bass up	Same as Demo #1.
5	Treble up	Same as Demo #1.
6	Save device state	Same as Demo #1.
7	Next Internet URL	Five Shoutcast URLs are stored in the demo, each pointing to a streaming media source carrying a different genre – Classical, Rock, Country, Jazz, and Alternative. Pressing this button causes the demo to access the next URL in the list, returning to the first one after the end of the list is reached.
8	Not used	
9	Stop	Same as Demo #1.
10	Next track	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. The following URLs are hard-coded into the demo:

1. Alternative: http://205.188.245.131:8042/
2. Classical: http://66.9.50.228:8400
3. Rock: http://128.121.239.88:10530
4. Country: http://66.119.198.120:8000
5. Jazz: http://205.188.234.34:8052

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.
- **Multiple Playlist support/Playlist Manangement** – Demo #2 provides code for saving and loading playlists from media. However, it does not allow choosing which playlist to load, or any ability to name the saved playlist.
- **Power management** – No power management features are used in the demo applications.

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 Dadio 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.

Files

```
./player/codec/common/include/Codec.h
```

Resources

The ICodec interface uses the following modules:

```
./player/datastream/input
./player/util/eresult
./player/util/registry
./player/util/ident
./player/util/rbuf
./player/playlist/common
```

Error codes

The ICodec interface uses the ERESULT error system. The following ICodec error results are defined and interpreted by the Dadio(TM) OS:

CODEC_NO_ERROR	No error was encountered during the operation
CODEC_DRM_SUCCESS	DRM Authentication succeeded for the input stream
CODEC_NO_WORK	The codec was asked to decode a frame, but either no input data is available, or no output space is available

CODEC_BAD_FORMAT	The input stream is not in the expected format
CODEC_DECODE_ERROR	The decoder had an error on the bitstream
CODEC_END_OF_FILE	An EOF was received from the input stream
CODEC_FAIL	General purpose failure
CODEC_DRM_FAIL	DRM Authentication failure

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

Codec Registration System

Dadio 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.

Files

```
./player/codec/common/include/Codec.h
```

Resources

The Codec Registration system uses the following modules:

```
./player/util/registry
./player/util/ident
```

Macros

```
#define DEFINE_CODEC( codecname, codecID,
can_probe, extensions... )
```

Purpose

Fill out the codec class definition with additional members necessary for codecs to work in Dadio(tm). Additionally, 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" )
```

```
#define REGISTER_CODEC( classname, codecID )
```

Purpose

To be used in your implementation file. Defines helper data structures and has function implementations necessary for your codec to be registered with Dadio.

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). 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 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. At startup, you create a `CConfigurableMetadata` object and 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 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 Dadio 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. Create a `media_record_info_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 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.

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) 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

Dadio 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
- `CDPInputStream` (in development) – Files on a DP device.

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, 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.

Each of these content managers has a corresponding `Metadata` object, as well as concrete implementations of `IMediaContentRecord` and `IPlaylistContentRecord`.

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

    1. 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.
    2. The data source calls `CreateMetadata()` for each content record.
    3. The Play Manager passes the updated records to AddContentRecords().

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

CreateMetadataRecord()

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.

AddContentRecords()

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.

AddMediaRecords()

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:

- Create a media content record to store the incoming data.
- The media content record should make a copy of the URL, since that memory is managed by the caller.
- 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.

- If pbAlreadyExists is non-zero, then set the contents to false.
- Return a pointer to the new media content record.

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

- Merge the metadata from the media_record_info_t's pMetadata field with the metadata in the media content record.
- Delete the passed-in metadata object.
- 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.
- If pbAlreadyExists is non-zero, then set the contents to true.
- Return a pointer to the matching media content record.

AddPlaylistRecord

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:

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

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

- 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.
- 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.

CSimplePlaylist 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.