



GNSDK Developer's Guide

Release Number 3.2.0

Published: 1/8/2013 7:41 AM

Gracenote, Inc.
2000 Powell Street, Suite 1500
Emeryville, California
94608-1804
www.gracenote.com

Getting Started with GNSDK

This topic covers key concepts and guidelines for GNSDK application development. Most of the examples in this document are based on C. However, the concepts and general processes apply to all languages supported by GNSDK.

About Gracenote

A pioneer in the digital media industry, Gracenote combines information, technology, services and applications to create ingenious entertainment solutions for the global market.

From media management, enrichment and discovery products to content identification technologies, Gracenote allows providers of digital media products and the content community to make their offerings more powerful and intuitive, enabling superior consumer experiences. Gracenote solutions integrate the broadest, deepest, and highest quality global metadata and enriched content with an infrastructure that services billions of searches a month from thousands of products used by hundreds of millions of consumers.

Gracenote customers include the biggest names in the consumer electronics, mobile, automotive, software and Internet industries. The company's partners in the entertainment community include major music publishers and labels, prominent independents and movie studios.

Gracenote technologies are used by leading online media services, such as Apple iTunes®, Pandora® and Sony Music Unlimited, and by leading consumer electronics manufacturers, such as Pioneer, Philips and Sony.

For more information about Gracenote, please visit: www.gracenote.com.

Resources for GNSDK

Other useful resources provided in the SDK package are:

- Sample applications you can compile and run. Find these in the samples folder of the SDK package.
- Reference applications, including source code, that show how to implement several GNSDK features. Find these in the reference_apps folder of the SDK package.
- A Data Dictionary that describes GNSDK data and metadata fields
-
- GNSDK API Reference documentation:
 - GNSDK C API Reference (see the /docs folder in the SDK package)
- A data model reference table that maps Gracenote Data Object (GDO) types to child keys and value keys (see Data Model in the GNSDK C API Reference)
- A list of possible GNSDK error codes (see Error Codes in the GNSDK C API Reference)

Key Concepts

This topic describes key concepts and terminology for developing GNSDK applications.

What is Gracenote SDK?

Gracenote SDK (GNSDK) is a platform that delivers Gracenote technologies to devices, desktop applications, web sites, and backend servers. GNSDK enables easy integration of Gracenote technologies into customer applications and infrastructure—helping developers add critical value to digital media products—while retaining the flexibility to fulfill almost any customer need.

GNSDK is designed to meet the ever-growing demand for scalable and robust solutions that operate smoothly in high-traffic servers and multithreaded environments. GNSDK is also lightweight – made to run in desktop applications and even to support popular portable devices.

GNSDK and Gracenote Products

GNSDK provides convenient access to Gracenote Services (Gracenote's suite of advanced media recognition, enriched content, and discovery services paired with robust infrastructure), enabling easy integration of powerful Gracenote products into customer applications and devices. Gracenote products provided by GNSDK are summarized in the table below.

Gracenote Products	GNSDK Modules	Module Description
MusicID TM	MusicID, MusicID-File	Enables MusicID recognition for identifying CDs, digital music files, and streaming audio and delivers relevant metadata such as track titles, artist names, album names, and genres. Also provides library organization and direct lookup features.
Playlist	Playlist, MoodGrid	Provides functionality to generate and manage playlists and playlist collections. Also supports creation of MoodGrids to visually group content based on mood metadata.

System Requirements

GNSDK provides thread-safe technology for the following common platforms.

Platform	Architecture	Bit	S/S	OS	Compiler/Note
Android	ARM	32-bit	Shared only		arm-linux-androideabi-gcc (GCC) 4.4.3
iOS	ARMV7	32-bit	Static only	iOS 4.3 and above	i686-apple-darwin11-llvm-gcc-4.2 (GCC) 4.2.1

Platform	Architecture	Bit	S/S	OS	Compiler/Note
iOS	x86 Emu-lator	32-bit	Static only	iOS 4.3 and above	i686-apple-darwin11-llvm-gcc-4.2 (GCC) 4.2.1
Linux	x86	32-bit 64-bit	Shared only	2.6 kernel and above	gcc (GCC) 3.4.6 20060404 (Red Hat 3.4.6-11) glibc-2.3.4-2.43 glibc 2.3.2 or newer*
MacOS	x86	32-bit	Shared only	OSX 10.4 and above	i686-apple-darwin10-llvm-gcc-4.2 (GCC) 4.2.1
Windows	x86	32-bit	Shared only	Windows XP or above	Microsoft Visual Studio 2010 (Version 10.0.40219.1 SP1Rel)

*The GNU C Library (glibc), revision 2.3.2 and newer, is typically available on Linux distributions released during or after 2003.



NOTE: Please contact Gracenote if the platform you need is not listed above.

Gracenote Media Elements

Gracenote Media Elements are the software representations of real-world things like CDs, Albums, Tracks, Contributors, and so on. The following is a partial list of the higher-level media elements represented in the GNSDK:

Music

- Music CD
- Album
- Track
- Artist
- Contributor

Core and Enriched Metadata

All Gracenote customers can access core metadata from Gracenote for the products they license. Optionally, customers can retrieve additional metadata, known as *enriched metadata*, by purchasing additional metadata entitlements.

The following diagram shows the core and enriched metadata available for Music.

Music Core Metadata	Music Enriched (Link) Metadata
Album Title	Album Cover Art
Track Title	Artist Images
Artist Name	...
Album Genre	
Track Genre	
Artist Origin	
Album Era	
Track Era	
Artist Type	
External IDs (XIDs)	
...	

Genre and other List-Dependent Values

Some Gracenote metadata is grouped into hierarchical lists. Some of the more common examples of list-based metadata include genre, artist origin, artist era, and artist type. Other list-based metadata includes mood, tempo, roles, and others.

Lists-based values can vary depending on the locale being used for an application. That is, some values will be different depending on the chosen locale of the application. Therefore, these kinds of list-based metadata values are called *locale-dependent*. For a list of locale-dependent metadata values, See "Locales" on page 32.

Genres and List Hierarchies

One of the most commonly used kinds of metadata are genres. A genre is a categorization of a musical composition characterized by a particular style.

The Gracenote Genre System contains more than 2200 genres from around the world. To make this list easier to manage and give more display options for client applications, the Gracenote Genre System groups these genres into a relationship hierarchy. The current hierarchy consists of three levels: level-1, level-2, and level-3.

For example, the partial genre list below shows two level-1 genres, Alternative & Punk and Rock. Each of these genres has two level-2 genres. For Rock, the level-2 genres shown are Heavy Metal and 50's Rock. Each level-2 genre has three level-3 genres. For 50's Rock, these are Doo Wop, Rockabilly, and Early Rock and Roll. This whole list represents 18 genres.

- Alternative & Punk
 - Alternative
 - Nu-Metal
 - Rap Metal
 - Alternative Rock
 - Punk
 - Classic U.K. Punk
 - Classic U.S. Punk
 - Other Classic Punk

- Rock
 - Heavy Metal
 - Grindcore
 - Black Metal
 - Death Metal
 - 50's Rock
 - Doo Wop
 - Rockabilly
 - Early Rock & Roll

Simplified and Detailed Hierarchical Groups

In addition to hierarchical levels for some metadata, the Gracenote Genre System provides two general kinds of hierarchical groups (also called list hierarchies). These groups are called *Simplified* and *Detailed*.

You can choose which hierarchy group to use in your application for any of the locale/list-dependent values. Your choice depends on how granular you want the metadata to be.

The Simplified group retrieves the coarsest (largest) grain, and the Detailed group retrieves the finest (smallest) grain, as shown below.



Below are examples of a Simplified and Detailed Genre Hierarchy Groups. The values below are for documentation purposes only.

Example of a Simplified Genre Hierarchy Group

- Level 1: 10 genres
 - Level 2: 75 genres
 - Level 3: 500 genres

Example of a Detailed Genre Hierarchy Group

- Level 1: 25 genres
 - Level 2: 250 genres
 - Level 3: 800 genres

Basic Application Design

This topic describes how to get started with GNSDK development.

Application Flow

Every GNSDK application follows this general design and application flow:

1. Include the GNSDK header files for the modules your application requires. These should be based on the Gracenote products you licensed, such as MusicID, MusicID-File, and so on.

2. Include other application headers your application requires.
3. Initialize GNSDK Manager.
4. Initialize other modules as needed, such as MusicID (for example).
5. If you are using a Gracenote local database, initialize it.
6. Enable Logging. Gracenote recommends that the logging functionality of GNSDK be enabled to some extent. This aids in application debugging and issue reporting to Gracenote.
7. Create a User (or deserialize an existing User) to get a User handle. All queries require a User handle with correct Client ID information to perform the query.
8. Create a query handle with lookup criteria and any query options.
9. Perform the query. Gracenote returns a response GDO with full or partial results. The GDO may contain a single match or multiple matches, requiring additional queries to refine the results.
10. Process the query result to get to the desired element and retrieve its metadata
11. Perform clean up by releasing all GDO and query handles.
12. Release the User handle.
13. Shutdown GNSDK modules. Shut down GNSDK Manager last.

GNSDK Modules

GNSDK consists of several modules that support specific Gracenote products. The principal module required for all applications is the GNSDK Manager. Others are optional, depending on the functionality of the applications you develop and the products you license from Gracenote.

General GNSDK Modules

GNSDK Manager: Required. GNSDK Manager provides core functionality necessary to all Gracenote modules.

Link: Optional. The Link module provides access to enriched metadata. This metadata is linked to but not part of a Response GDO. Instead, it is accessed separately via the Link module APIs. The Link module can also return any custom information that a customer has linked to Gracenote metadata, such as their own media identifiers. The application provides Gracenote identifiers (derived from previous Gracenote identification results) to the Link module to access the additional content. To use this module, your application requires special metadata entitlements.

DSP: Optional. Provides digital signal processing functionality required for fingerprint generation.

Local Database : Optional. Used for embedded system applications, such as car stereo head units, that require a local database that contains pre-defined Gracenote metadata.

SQLite: The SQLite module provides a local storage solution for GNSDK using the SQLite database engine. This module is primarily used to manage a local cache of queries and content that the GNSDK modules make to Gracenote Service. SQLite is a software module that implements a self-contained, server-less, zero-configuration, transactional SQL database engine. SQLite is the most widely deployed SQL database engine in the world. The source code for SQLite is in the public domain. Information on SQLite can be found at <http://www.sqlite.org>.

Product-Specific Modules

MusicID: Provides CD TOC, text, and fingerprint identification for music.

MusidID-File: Provides file-based music identification using LibraryID, AlbumID, and TrackID processing functionality.

Playlist: Provides functionality to generate and manage playlists and playlist collections. Also supports creation of MoodGrids to visually group content based on mood metadata.

Header Files and Libraries

GNSDK consists of a set of shared modules. The GNSDK Manager module is required for all applications. All other modules are optional. Your application's feature requirements determine which of the optional modules should be used.

As shown in the following table, each GNSDK module has one or more corresponding header files. Other header files are also provided for public types and error codes.



NOTE: The libraries can be linked in any order for all modules, except for the GNSDK Manager library. The GNSDK Manager library must be last in the linking order.

Module	Header Files	Libraries
General	<p>Required.</p> <p>gnsdk.h. This header file also includes these required header files:</p> <ul style="list-style-type: none"> • gnsdk_defines.h: GNSDK types • gnsdk_platform.h: GNSDK platform specific types (one header for each supported platform) • gnsdk_errors.h: GNSDK error macros • gnsdk_error_codes.h: GNSDK error definitions • gnsdk_version.h: GNSDK current version definition (this header is optional) 	
GNSDK Manager	<p>Required. These header files provide core functionality necessary to all Gracenote modules.</p> <ul style="list-style-type: none"> • gnsdk_manager.h • gnsdk_manager_gdo.h • gnsdk_manager_lists.h • gnsdk_manager_list_values.h • gnsdk_manager_locales.h • gnsdk_manager_logging 	<ul style="list-style-type: none"> • libgnsdk_manager.so • libgnsdk_manager.dylib • gnsdk_manager.dll

Module	Header Files	Libraries
MusicID	<p>Optional. This header file provides CD TOC, text, and fingerprint identification for music.</p> <ul style="list-style-type: none"> • gnsdk_muscid.h 	<ul style="list-style-type: none"> • libgnsdk_muscid.so • libgnsdk_muscid.dylib • gnsdk_muscid.dll
MusicID-File	<p>Optional. Provides music file-based identification using LibraryID, AlbumID, and TrackID processing functionality.</p> <ul style="list-style-type: none"> • gnsdk_muscid_file.h 	<ul style="list-style-type: none"> • libgnsdk_muscid_file.so • libgnsdk_muscid_file.dylib • gnsdk_muscid_file.dll
DSP	<p>Optional. Provides digital signal processing functionality required for fingerprint generation.</p> <ul style="list-style-type: none"> • gnsdk_dsp.h 	<ul style="list-style-type: none"> • libgnsdk_dsp.so • libgnsdk_dsp.dylib • gnsdk_dsp.dll
Link	<p>Optional. Provides enriched metadata (non-core content and third party metadata.).</p> <ul style="list-style-type: none"> • gnsdk_link.h 	<ul style="list-style-type: none"> • libgnsdk_link.so • libgnsdk_link.dylib • gnsdk_link.dll
MoodGrid	<p>Optional. A feature of Playlist to visually group content on a grid based on mood metadata.</p> <ul style="list-style-type: none"> • gnsdk_moodgrid.h 	<ul style="list-style-type: none"> • libgnsdk_moodgrid.so • libgnsdk_moodgrid.dylib • gnsdk_moodgrid.dll
Playlist	<p>Optional. Provides functionality to generate and manage playlists and playlist collections.</p> <ul style="list-style-type: none"> • gnsdk_playlist.h 	<ul style="list-style-type: none"> • libgnsdk_playlist.so • libgnsdk_playlist.dylib • gnsdk_playlist.dll

Module	Header Files	Libraries
Local Database	<p>Optional. Used for embedded system applications that require a local database that contains pre-defined Gracenote metadata.</p> <ul style="list-style-type: none"> • gnsdk_lookup_local.h 	<ul style="list-style-type: none"> • libgnsdk_lookup_local.so • libgnsdk_lookup_local.dylib • gnsdk_lookup_local.dll
SQLite	<p>Optional. Provides local caching implementation based on SQLite, and provides the default storage implementation to support local databases.</p> <ul style="list-style-type: none"> • gnsdk_sqlite.h 	<ul style="list-style-type: none"> • libgnsdk_sqlite.so • libgnsdk_sqlite.dylib • gnsdk_sqlite.dll

Authorizing a GNSDK Application

Gracenote manages access to metadata using a combination of product licensing and server-side metadata entitlements.

When developing a GNSDK application, you must include a client ID and license file to authorize your application with Gracenote. In general, the License file enables your application to use the Gracenote products (and their corresponding GNSDK modules) that you purchased. The client ID is used by Gracenote Media Services to enable access to the metadata your application is entitled to use.

Client IDs

Each GNSDK customer receives a unique client ID string from Gracenote. This string uniquely identifies each application to Gracenote Media Services and lets Gracenote deliver the specific metadata the application requires.

A client ID string has the following format: 123456-789123456789012312. It consists of a six-digit client ID, and a 17-digit client ID Tag, separated by a hyphen (-).

When creating a new User with `gnsdk_manager_user_create_new()`, you must pass in the client ID and client ID tag as separate parameters. For more information, See "Users" on page 11.

License Files

Gracenote provides a license file along with your client ID. The license file notifies Gracenote to enable the GNSDK products you purchased for your application. Below is an example license file, showing enabled and disabled Gracenote products for Customer A. These products generally map to corresponding GNSDK modules.

```
-- BEGIN LICENSE v1.0 1D0D9110 --
licensee: Customer A
name:  license_file_name
start_date: 2012-06-29
client_id: a_client_id
musicid_file: enabled
musicid_search: enabled
musicid_stream: disabled
musicid: enabled
submit: enabled
playlist: disabled
videoid: enabled
videoid_explore: enabled
-- SIGNATURE 1D0D9110 --
hashed_client_id_string
-- END LICENSE 1D0D9110 --
```

Users

To perform queries, an application must first register a new User and get its handle. A User represents an individual installation of a specific client ID string. This ensures that each application instance is receiving all required metadata entitlements.

Users are represented in GNSDK by User handles. You can create a User handle using either `gnsdk_manager_user_create_new()` or `gnsdk_manager_user_create()`.

When creating a new User with `gnsdk_manager_user_create_new()`, you must pass in the client ID and client ID tag as separate parameters. For more information about client IDs and client ID tags, See "Client IDs" on page 10.

Each application installation should only request a new User once and store the serialized representation of the User for future use. For example:

```
gnsdk_user_handle_t user_hdl = GNSDK_NULL; // User handle
gnsdk_cstr_t        ser_hdl  = GNSDK_NULL; // Serialized user handle

/** Get serialized user handle if one has been saved
userHandleSaved = get_serialized_user_data(&ser_hdl); // NOT a Gracenote
API

/** Call API to register and create new user
if (userHandleSaved)
{
    // This API takes a serialized user handle ('ser_hdl') and creates a
    // non-serialized user handle in 'user_hdl'
    error = gnsdk_manager_user_create(ser_hdl, &user_hdl);
}
else
{
    // This API creates a new, non-serialized user handle in 'user_hdl'
```

```

    error = gnsdk_manager_user_create_new(clientid, clienttag, clientver,
&user_hdl);
}

/**
/** Perform queries - this changes the user handle
/** ...

/*******
/**
/** R E L E A S E   U S E R
/**
/** This API MAY serialize the user handle to 'ser_hdl':
/** If you called gnsdk_manager_user_create_new(), then
/** gnsdk_manager_user_release() will ALWAYS set 'ser_hdl' to a serialized
/** user handle. If you used gnsdk_manager_user_create(), then 'ser_hdl'
will
/** ONLY contain data if the user handle has changed between 'create' and
'release'.
/**

/*******
****
error = gnsdk_manager_user_release(user_hdl, &ser_hdl);

/**
/** Save serialized user handle before exiting
/** It is recommended that a serialized user handle be saved only upon
close as
/** the user handle can change during the course of the app
/**
if (!error && ser_hdl)
{
    save_serialized_user_data(ser_hdl); // NOT a Gracenote API
}

// Close GNSDK

```

Managing User Handles

In general, all User handles are thread-safe and can be simultaneously used by multiple queries.

Basic User management process:

1. First application run: create new User and get handle.
2. Provide User handle to GNSDK APIs that require one.

3. Release User handle when finished and store it as serialized data.
4. Subsequent application runs: create User from stored serialized data.

If an application registers a new User on every use instead of storing a serialized User, then the User quota maintained for the client ID is quickly exhausted. Once the quota is reached, attempting to create new Users will fail. If an application uses a single User registration across multiple installations of the application—in short, forcing all the installations to use the same User—the application risks exhausting Gracenote per-user query quota.

To maintain an accurate usage profile of your application, and to ensure that the services you are entitled to are not being used unintentionally, your application should register new Users only when needed, and then store that User for future queries.

Online and Local Database Lookup Modes

Your application must set an option that indicates whether lookups should be performed locally or online. The following table shows the possible options:

Option	Description
GNSDK_LOOKUP_MODE_ONLINE	This is the default mode. If a cache exists, the query checks it first for a match. If a no match is found in the cache, then the online Gracenote Service is queried. If a result is found there, it is stored in the local cache. If no online provider exists, the query will fail.
GNSDK_LOOKUP_MODE_ONLINE_NOCACHE	This mode forces the query to be done online only and will not perform a local cache lookup first. If no online provider exists, the query will fail. If a storage provider has been initialized, queries and lists are written to local storage, but are never read unless the lookup mode is changed.
GNSDK_LOOKUP_MODE_LOCAL	This mode forces the lookup to be done against the local database only. Local lookups and local caches are used. If no local database exists, the query will fail.

For example, the following call sets the user lookup option so that lookups are performed locally:

```
gnsdk_manager_user_option_set(
    user_handle,
    GNSDK_USER_OPTION_LOOKUP_MODE,
    GNSDK_LOOKUP_MODE_LOCAL
);
```



Your application can use GNSDK_USER_OPTION_CACHE_EXPIRATION option to set the length of time before a cache lookup times out.

Overriding the Lookup Mode for a Specific Query

Setting the GNSDK_USER_OPTION_LOOKUP_MODE option for a user handle applies to *all queries using the user handle*. You can override this for a specific query by setting the equivalent *query handle option*.

For example, you can override the setting for a music query by setting the GNSDK_MUSICID_OPTION_LOOKUP_MODE option. The query handle option uses the same option value keys as the user handle option.

Using Both Local and Online Lookup Modes

Your application can switch between local and online lookups. To do this, you need to explicitly switch modes when needed. For example, the following pseudocode shows how to do a local Album TOC lookup using MusicID APIs, followed by an online lookup for cover art using Link APIs:

```
/* Local TOC lookup using MusicID
musicid_option_set(MODE_LOCAL)
musicid_toc_set(toc)
musicid_find_album(&album_gdo)
/* Online cover art lookup using Link
link_option_set(MODE_ONLINE)
link_set_gdo(album_gdo)
link_retrieve_content(cover)
```

Initializing an Application

Before using a module, the application must initialize it. All applications must first initialize the GNSDK Manager module by calling `gnsdk_manager_initialize()`. In this call, the application must include the client ID and a Gracenote license file. Initializing GNSDK Manager returns a handle that is required to initialize other modules.

Depending on your application logic, you may need to retain the GNSDK Manager handle to initialize other modules from different locations in your application. One option is to globally manage the GNSDK manager handle so it is always available. Alternatively, you can call `gnsdk_manager_initialize()` multiple times when needed and avoid shutting down GNSDK Manager prematurely.

Example: Using GNSDK Manager Handles.

The following example shows how to initialize GNSDK Manager and maintain its handle for multiple initializations of other modules.

Code Snippet: [initialization_multiple.c](#)

Application Steps:

1. Initialize GNSDK Manager with license data and get GNSDK Manager handle.
2. Use GNSDK Manager handle to initialize MusicID.

3. Use GNSDK Manager handle to initialize MusicID-File.
4. Shut down MusicID and MusicID-File.
5. Shut down GNSDK Manager.

Specifying the License File

Each application provides the license file on the first (and likely only) call to `gnsdk_manager_initialize()`. Your application has a choice of methods of providing the license file to the GNSDK Manager. These methods are controlled through the `license_data` and `license_data_len` parameters of the `gnsdk_manager_initialize()` API. The methods are:

- **Memory buffer**—Set the `license_data` parameter to the memory buffer pointer, and the `license_data_len` to the size of the memory buffer.
- **Null-terminated string**—Set the `license_data` parameter to the string buffer pointer, and the `license_data_len` to `GNSDK_MANAGER_LICENSEDATA_NULLTERMSTRING`.
- **Filename**—Set the `license_data` parameter to a string buffer containing the relative filename of a file containing the license data, and the `license_data_len` to `GNSDK_MANAGER_LICENSEDATA_FILENAME`.
- **Stdin**—Set the `license_data` parameter to `GNSDK_NULL`, and the `license_data_len` to `GNSDK_MANAGER_LICENSEDATA_STDIN`.



NOTE: When necessary, an existing license file can be updated with new license file by calling `gnsdk_manager_initialize()` again. The new file overwrites the existing license data.

Shut Down Guidelines

Shutdown Every Initialization

Calls to the initialize API on any GNSDK module are counted. The first call to initialize the module does the actual work, and every subsequent call to initialize only increments an initialization count. So, calling initialize multiple times is safe and not resource-intensive.

However, for every initialize call, your application must call shutdown an equal number of times. Shutdown decrements the initialization count, and it is only when the count reaches zero that the actual shutdown occurs.

Therefore, each successful call to an initialize function must be paired (eventually) with a call to a shutdown function.



NOTE: Do not call a shutdown function if the corresponding initialize function returns an error.

Shutdown GNSDK Manager Last

As a best practice, your application should shut down other modules before shutting down GNSDK Manager. You do not need to release the GNSDK Manager handle. The resources for the handle are freed when the GNSDK Manager is shut down.

Example: Initializing and Shutting Down

Code Snippet: [initialization.c](#)

Application Steps:

1. Initialize GNSDK Manager.
2. Initialize MusicID (for example).
3. Register User.
4. Perform queries.
5. Release User.
6. Shutdown MusicID.
7. Shutdown GNSDK Manager.

Gracenote Data Objects (GDOs)

The primary goal of any GNSDK application is to recognize media elements and access their metadata. When an application performs a query, Gracenote returns metadata about the target query element, such as the title and genre of an album element. In addition, information about the query operation is returned, such as its timestamp, the start and end range of the query results, and the number of additional results available from Gracenote.

GNSDK stores the information returned by a query within containers known as Gracenote Data Objects (GDOs). The contents of a GDO depends on:

- The kind of query and the search criteria, such as a CD TOC lookup, a fingerprint lookup, text lookup, and so on
- The query target element
- Information about the target element available from Gracenote

A GDO can contain:

- String values and/or
- Other GDOs

GDOs have two purposes:

1. Act as a container for accessing and navigating metadata returned from Gracenote
2. Act as an input for retrieving data from Gracenote

GDOs facilitate a key feature of GNSDK – interoperability between all of the Gracenote products and services. Results from one Gracenote query can be used as an input for another. For example, a MusicID result can immediately be used as an input for the Link module without the need for any application intervention. This interoperability is possible for nearly all combinations of Gracenote Services.

GDO Types

Every GDO has a type. For example, when an application performs a query to identify an Album, Gracenote returns a GDO of type Album. Therefore, for most applications, you can *infer* a GDO's type based on the target element of the query and knowing the underlying data model for the element.

If needed, your application can get the type of a GDO using `gnsdk_manager_gdo_get_type()`. For example, your application might request a GDO's type to confirm it matches the intended type.

Another use case is analyzing the results of a general text lookup. This kind of query can return multiple GDOs of *different* types. The application needs to process the results to determine which GDO is the best response to the query.

Child GDOs and Values

GDO responses can contain other GDOs or references to them. These related "sub-GDOs" are called *child GDOs*.

You can think of any GDO response as a "parent GDO" if it contains or references other GDOs. The contained GDOs are its children.

A child GDO is just like any other GDO once it is retrieved. It is not dependent on its parent GDO and has the same behaviors and features of other GDOs. The fact that it was once a child of another GDO does not matter.

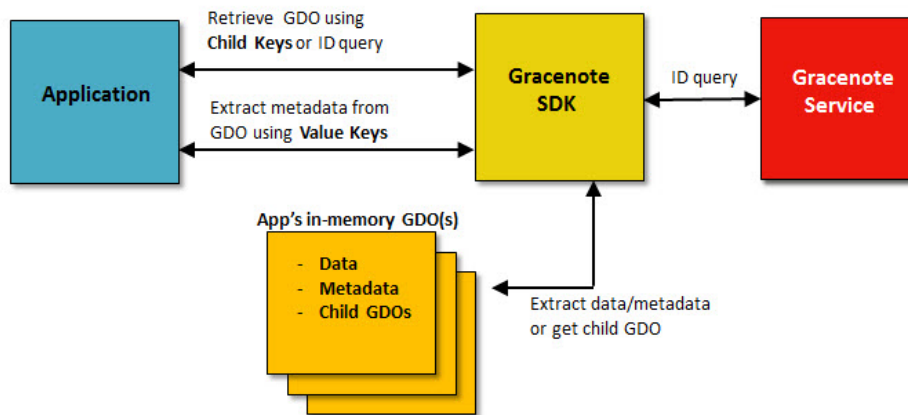
For example, a "parent" Album GDO response can contain child GDOs of type Track, or Artist, other Albums, and so on. A child GDO response can contain its own child GDOs, such as Tracks, Artists, or Contributors, and so on.

A GDO's child objects are enumerated with an ordinal index, starting from 1 (not 0) for the first child object. Queries for child objects take this index as input.

Child Keys and Value Keys

To extract metadata from a GDO, or get a child GDO, your application must use defined keys. There are two kinds of keys: Value and Child.

- Value Key—Used to extract a specific piece of metadata from a GDO, for example `GNSDK_GDO_VALUE_ALBUM_LABEL`.
- Child Key—Used to get a child GDO, for example, `GNSDK_GDO_CHILD_TRACK`.



Full and Partial Results

A GDO response contains either *partial* or *full* results. A partial (or non-full) result contains a subset of information about the match, but enough to perform additional processing. One common use of partial results is to present the information to an end user to make a selection and then perform a secondary query for full results.

To test if a GDO is a full or partial result, use the `GNSDK_GDO_VALUE_FULL_RESULT` value key. A return value of zero indicates a partial result. A non-zero return value indicates a full result.



For a list of values returned in a partial result, See "GNSDK Data Model for GNSDK" on page 54.

Online and Local Database Queries

Applications that have an online connection can query Gracenote for information. Applications without an online connection, such as embedded applications used for stereo head units in cars, can instead query a Gracenote local database.

In general, local database queries return full GDO results, even when the response contains multiple matches. Most online queries return partial GDO results containing just enough information to identify the matches. Using this information, the application can perform additional queries to obtain more information. In general, your application should *always* test GDO results to determine they are full or partial.

Matches that Require Decisions

When an application performs an identification query, Gracenote returns one of the following:

- no GDO match
- single GDO match
- multiple GDO matches

In all cases, Gracenote returns high-confidence results based on the identification criteria.

However, even high-confidence results may require additional decisions from an application or end user. For example, *all* multiple GDO match responses require the application (or end user) make a decision about which GDO to use and process to retrieve its metadata. Therefore, each multiple match GDOs response is flagged as "*needs decision*" by GNSDK.

A single GDO response can also need a decision. In this case, the match result is a good candidate, but by Gracenote's determination, may not be perfect. If a single response does not need a decision, Gracenote has determined that the response is as accurate as possible based on the criteria used for the identification query.

Responses that require a decision are:

- any response containing multiple GDO matches
- any single match response that Gracenote determines needs a decision from the application or end user, based on the quality of the match and/or the mechanism used to identify the match (such as text, TOC, fingerprints, and so on),

To test if a GDO needs a decision, use the GDO value key `GNSDK_GDO_VALUE_RESPONSE_NEEDS_DECISION`. This key returns a boolean `TRUE` value if a decision is needed.

To resolve a decision, an application typically presents the end user with the matches and lets the user select the correct match or reject them all. Alternatively, the application may automatically select the first match or reject them all .



In all cases, match results can be partial or full, so after selecting a match, the application should test it using `GNSDK_GDO_VALUE_FULL_RESULT`.

About the Text Match Score

GNSDK provides a score for text matches based on comparing the input text to the corresponding field in the match text. In this case, the text score does not indicate the *quality* of the text match.

For example, the result text could be substantially different from the input text because the input text contained a lot of incorrect information. Such a response indicates that the results have an amount of ambiguity in them that the application must resolve.

GDO Workflows

A Gracenote identification query can return no GDO match, a single GDO match, or a multiple GDO match. The workflow for managing single and multiple GDO matches is similar, but not identical. The following sections describe these two workflows.

GDO Workflow for a Single GDO Match

The simplest example of a GDO workflow is when an identification query returns a single match. For example, suppose there is a query to look up a track by name and find its containing album. If that Track only exists on one album, GNSDK identifies the single album and returns a response GDO that contains the core metadata for the identified album. The application can then access the metadata using value keys.

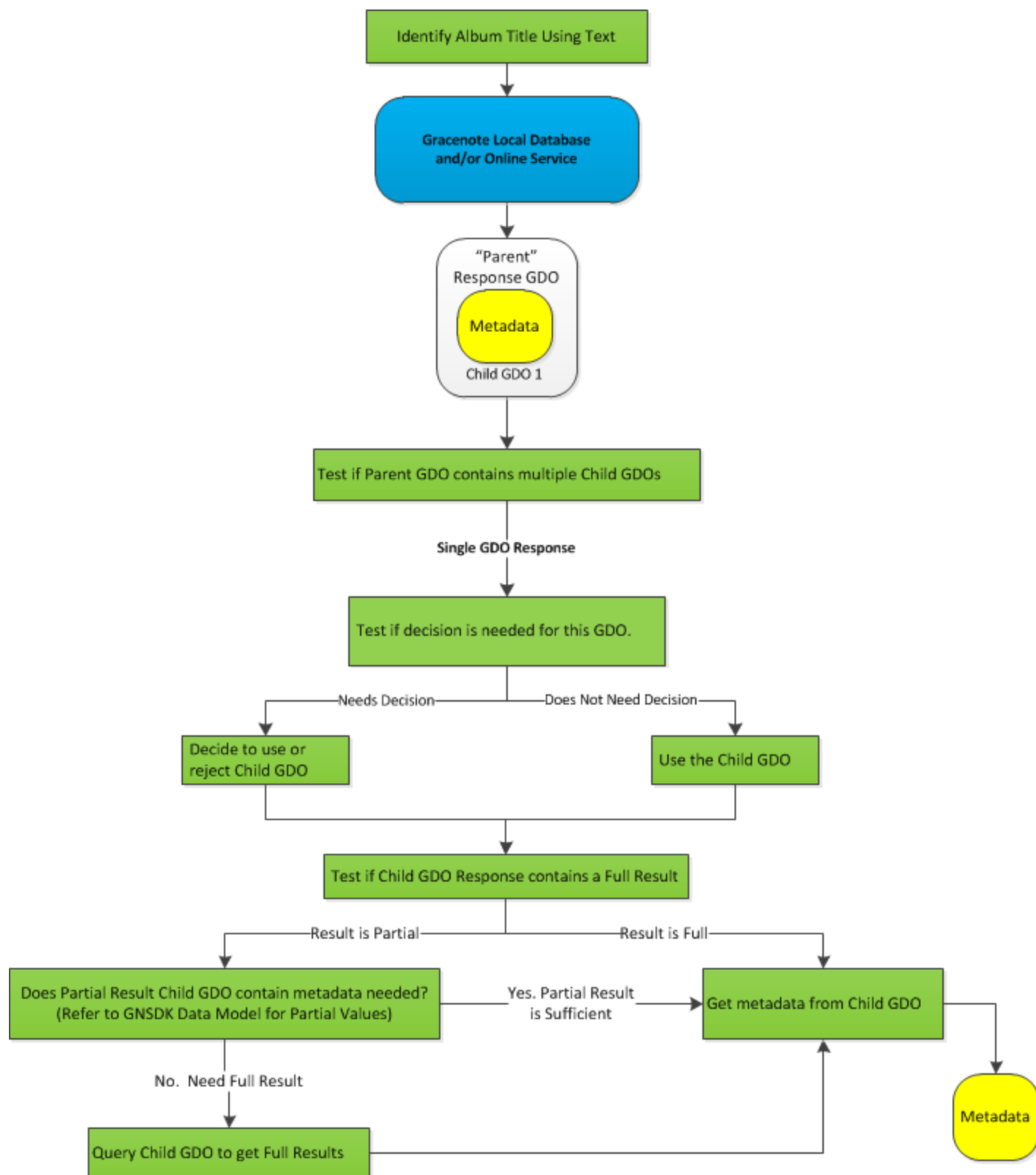
As described in See "Matches that Require Decisions" on page 18, some single GDO responses may require a decision by the application or end user. To address this possibility, the GDO workflow should include a test query using the GNSDK_GDO_VALUE_RESPONSE_NEEDS_DECISION value key to determine if Gracenote has pre-determined that the GDO response needs a decision.

The application should also test the GDO to determine if it contains partial or full results. Use the GNSDK_GDO_VALUE_FULL_RESULT value key for this test. If the response is partial, the application can either use the partial information or perform an additional query to get the full results. In some cases, the information returned in a partial response may be sufficient for the purpose of the query. If so, the application can simply get the values from the partial response.



For a list of values returned in a partial result, See "GNSDK Data Model for GNSDK" on page 54.

The following diagram shows the basic workflow, followed by the application steps in detail.



GDO Workflow steps for Album title text lookup - Single GDO Response

1. Call `gnsdk_musid_query_create()` to create a query handle.
2. Call `gnsdk_musid_query_set_text()` with input text and input field `GNSDK_MUSICID_FIELD_ALBUM` to set the text query for an album title.
3. Call `gnsdk_musid_query_find_albums()` to perform the query.
4. Test if Parent GDO response has multiple Child GDOs using `gnsdk_manager_gdo_value_get()` with `GNSDK_GDO_VALUE_RESPONSE_RESULT_COUNT`. For this example, assume a single GDO response was returned.
5. Test if a decision is needed for the Parent GDO using `gnsdk_manager_gdo_value_get()` with value key `GNSDK_GDO_VALUE_RESPONSE_NEEDS_DECISION`.
6. If decision is not needed, use the Child GDO. If a decision is needed, choose to use the GDO or reject it.
7. If using the GDO, call `gnsdk_manager_gdo_child_get()` with Child Key `GNSDK_GDO_CHILD_ALBUM` and its ORD value of 1.
8. Test if the Child GDO Response contains a full result using `gnsdk_manager_gdo_value_get()` with Value Key `GNSDK_GDO_VALUE_FULL_RESULT`.
9. If the GDO contains a full result, jump to the last step. If the GDO contains partial results, decide if it contains the metadata needed for the query.
10. If partial result is sufficient, jump to the last step. If full results are needed, query the *Child GDO* to get the full results:
 - a. Set the handle to the *selected Child GDO* using `gnsdk_musid_query_set_gdo()`.
 - b. Re-query using `gnsdk_musid_query_find_albums()`.
11. Get metadata from Child GDO using `gnsdk_manager_gdo_value_get()` and value keys.

GDO Workflow for Multiple GDO Matches

Gracenote identification queries often return multiple matches. For example, suppose that a Track exists on multiple Albums, such as the original Album, a compilation Album, and a greatest hits Album. In this case, the query returns a Response GDO that contains multiple child Album GDOs. Each GDO represents a possible Album match for the query. When this happens, the end-user needs to select which Album they want. Each Response GDO returns enough metadata for this purpose. Based on the user's selection, the application can then send another query to Gracenote to return the GDO for the chosen Album.

The diagram below shows the general application GDO workflow for multiple GDO responses. As described in See "Matches that Require Decisions" on page 18, *all* multiple GDO match responses require a decision by the application or end user. Therefore, it is optional to test if a multiple GDO match needs a decision.

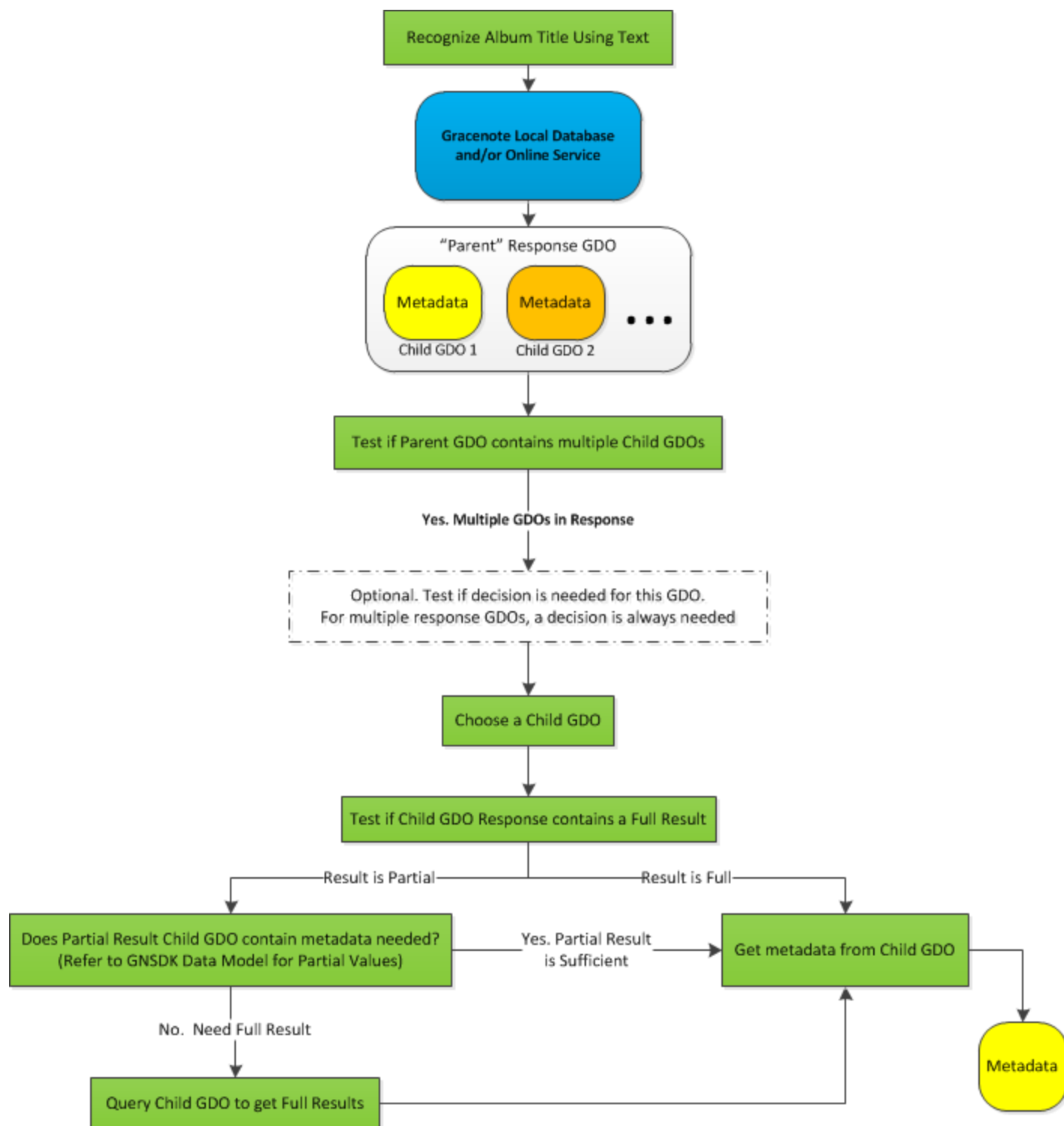
In general, after choosing a Child GDO from the multiple matches, the application should test it to determine if it contains partial or full results. Use the `GNSDK_GDO_VALUE_FULL_RESULT` value key for this test. If the response is partial, the application can either use the partial information or perform an additional query to get the full results. In some cases, the information returned in a partial response

may be sufficient for the purpose of the query. If so, the application can simply get the values from the partial response.



For a list of values returned in a partial result, See "GNSDK Data Model for GNSDK" on page 54.

The following diagram shows the basic workflow, followed by the application steps in detail.



GDO Workflow steps for Album title text lookup - Multiple GDO Response

1. Call `gnsdk_musicid_query_create()` to create a query handle.
2. Call `gnsdk_musicid_query_set_text()` with input text and input field `GNSDK_MUSICID_FIELD_ALBUM` to set the text query for an album title.
3. Call `gnsdk_musicid_query_find_albums()` to perform the query.
4. Test if Parent GDO response has multiple Child GDOs using `gnsdk_manager_gdo_value_get()` with Value Key: `GNSDK_GDO_VALUE_RESPONSE_RESULT_COUNT`. For this example, assume a multiple GDO response was returned.
5. *Optional.* Test if a decision is needed for the Parent GDO using `gnsdk_manager_gdo_value_get()` with Value Key: `GNSDK_GDO_VALUE_RESPONSE_NEEDS_DECISION`. For multiple response GDOs, this always returns TRUE.
6. Choose a specific Child GDO using `gnsdk_manager_gdo_child_get()` with Child Key `GNSDK_GDO_CHILD_ALBUM` and its ORD value (1-based).
7. Test if the Child GDO Response contains a full result using `gnsdk_manager_gdo_value_get()` with Value Key `GNSDK_GDO_VALUE_FULL_RESULT`.
8. If the GDO contains a full result, jump to the last step. If the GDO contains partial results, decide if it contains the metadata needed for the query.
9. If partial result is sufficient, jump to the last step. If full results are needed, query the Child GDO to get the full results:
 - a. Set the handle to the *selected Child GDO* using `gnsdk_musicid_query_set_gdo()`.
 - b. Re-query using `gnsdk_musicid_query_find_albums()`.
10. Get metadata from Child GDO using `gnsdk_manager_gdo_value_get()` and value keys.

Serializing a GDO

You can serialize a GDO to save it for later use in an application. Serializing a GDO retains only key information needed for common GDO functions. Other information is discarded.

To restore the discarded information, your application must request the GDO again. An application can reconstitute (de-serialize) a serialized GDO and use it for subsequent processing.



For a list of values returned in a partial result, See "GNSDK Data Model for GNSDK" on page 54.



Response GDOs are not serialized. The serialization API (`gnsdk_manager_gdo_serialize()`) does not return an error in this case, but no serialized output is generated.



Important: A serialized GDO is not a static identifier. You cannot use it as a comparison identifier for any purposes, including straight comparisons, caching, indexing, and so on.

Rendering a GDO as XML

GNSDK supports rendering GDOs as XML. This is an advanced feature, but may be useful to track user choices, retain query history, or provide data to other applications. See "Rendering a GDO as XML" on page 90

GDO Navigation Examples

Gracenote Data Objects (GDOs) are the primary identifiers used to access Gracenote metadata.

Example: Looking Up an Album by a TOC

Sample Application: [musicid_lookup_album_toc/main.c](#)

Example: Accessing Album and Track GDOs

Code Snippet: [gdo_get_metadata.c](#)

Description: This snippet shows how to get and display metadata from one album, including the title of each album track.

Application Steps:

1. Initialize SDK, MusicID library, User handle, and load locale
2. Perform a MusicID query and get an album response GDO
3. Get album child GDO from album response GDO
4. Get album title GDO from album child GDO
5. Get and display album title metadata
6. Get and display album metadata in album child GDO
7. Get track child GDOs from album child GDO
8. Get track title GDO from track child GDOs
9. Get and display track title metadata
10. Release resources and shutdown SDK

Example: Accessing Album and Track Metadata Using Album GDO

This example uses an Album GDO to access Album metadata: artist, credits, title, year, and genre, as well as basic Track metadata: artist, credits, title, track number, and genre.

Sample Application: [musicid_gdo_navigation/main.c](#)

Example: Serializing an Album GDO

An application can serialize a GDO to minimize its size. The process serializes key pieces of metadata needed to access additional information or related GDOs. Other information is discarded but can be restored by requesting the metadata again.

Serialized GDOs are not static. They include a timestamp, which indicates that the exact identifier value varies even though the metadata it refers to remains the same. Because of this, the GDO cannot be used as a comparison identifier for any purposes, including straight comparison, caching, indexing, and so on.

Sample Application: [gdo_serialize.c](#)

Application Steps:

1. Initialize SDK, MusicID library, User handle, and load locale
2. Deserialize input album GDO
3. Perform a MusicID query and get an album response GDO
4. Get album child GDO from album response GDO
5. Serialize album GDO
6. Display serialized album GDO string
7. Release resources and shutdown SDK

GNSDK Storage and Caching

The GNSDK SQLite module provides a local storage solution using the SQLite database engine. This module is used to manage a local cache of content and Gracenote Service queries. Note that this is for GNSDK use only - your application cannot use this database for its own storage.

NOTE: For information on using SQLite, see <http://www.sqlite.org>

In the future, other database modules will be made available, but currently, the only option is SQLite. Besides APIs specific to SQLite, there is a set of general storage APIs that apply, now and in the future, to whatever database module is implemented. These general APIs cover setting cache location and various cache maintenance operations (cleanup, validate, compact, flush, etc.). See the API Reference for a complete list.

Specifically, API calls are provided to manage 3 *stores* or *caches* (as indicated by the following defines):

1. GNSDK_MANAGER_STORAGE_CONTENTCACHE—Stores cover art and related information.
2. GNSDK_MANAGER_STORAGE_QUERYCACHE—Stores media identification requests.
3. GNSDK_MANAGER_STORAGE_LISTSCACHE—Stores Gracenote locale lists.

To begin, your application needs to make the following call to initialize SQLite (after initializing GNSDK Manager and getting an SDK handle).

```
gnsdk_storage_sqlite_initialize(sdkmgr_handle);
```



Important: It is possible to initialize this library at any time before or after other libraries have been operating. However, to ensure that all queries are properly cached, it should be initialized immediately after the GNSDK Manager and before any other libraries.

As with all GNSDK "initialize" calls, there should be a corresponding "shutdown" call before your application exits:

```
gnsdk_storage_sqlite_shutdown();
```

Following initialization, your code **must** make the following API call to establish a valid storage folder path for these caches. The path must be writeable. The following sample call sets this folder to the current directory ('.').

```
gnsdk_storage_sqlite_option_set(GNSDK_STORAGE_SQLITE_OPTION_STORAGE_FOLDER, ".");
```



Folder paths can be either relative or absolute. The SDK is path convention agnostic in that, given either direction for slashes (e.g., "\" or "/"), it will correct to the native OS's standard.

In addition to setting the location for all 3 caches, your application has the option to set a location for each cache with the `gnsdk_manager_storage_location_set()` API. Note that this is a general storage API call and not specific to SQLite.

```
gnsdk_manager_storage_location_set(GNSDK_MANAGER_STORAGE_QUERYCACHE, "./querycache");
```

You might want to set your cache stores to different locations to improve performance and/or tailor your application to specific hardware. For example you might want your locale list store in flash memory and your image store on disk.



Important: Neither `gnsdk_storage_sqlite_option_set()` nor `gnsdk_manager_storage_location_set()` returns an error if the passed location does not exist or is not writeable. In this case, no caching is done. Your application can create this folder at a later time to turn on caching.

Other SQLite API calls cover setting cache file and memory size, journaling, and synchronous operations. See the API Reference for a complete list.

Example: Caching of Queries and Content using SQLite

Code Snippet: [cache.c](#)

Prerequisite: Create a writeable `./querycache` folder where you are running the sample.

Application Steps:

1. Initialize GNSDK Manager.
2. Initialize SQLite.
3. Set a valid storage folder path for all 3 stores with `gnsdk_sqlite_option_set()`. (See code sample above.)

4. Set a folder path for QUERYCACHE (see code sample above) with `gnsdk_manager_storage_location_set()`.
5. Initialize MusicID.
6. Get User handle.
7. Perform MusicID query (this will be cached in `./querycache`).
8. Shutdown SQLite cache.
9. Shutdown other GNSDK modules.

After the program runs, you should see a `gn_cachq.gdb` file in `./querycache`.

Logging

To enable Gracenote SDK logging in your application, use the `gnsdk_manager_logging_enable()` function. For example:

```
/**
/** Enable logging
/**
gnsdk_manager_logging_enable(
    "sample.log",          // File path
    GNSDK_LOG_PKG_GNSDK,  // Include entries for higher level GNSDKs as
well as your app
    GNSDK_LOG_LEVEL_ALL,  // Include all level entries
    GNSDK_LOG_OPTION_ALL, // All logging options
    0,                    // Max size of log: 0 means a new log file
will be created each run
    GNSDK_FALSE           // GNSDK_TRUE = old logs will be renamed and
saved
);
```

The GNSDK logging system can manage multiple logs simultaneously. Each call to the enable API can enable a new log, if the provided log file name is unique. Additionally, each log can have its own filters and options.

The SDK allows an application to register one or more of its own package IDs into the GNSDK logging system with the `gnsdk_manager_logging_register_package()` call. The application can then enable, disable or filter its own logging messages based on the registered package IDs. For example:

```
/**
/** Register our package ID with the logging system - this will make our
logging
/** entries show up with the package description of "Sample App"
/**
gnsdk_manager_logging_register_package(MYAPP_PACKAGE_ID, "Sample App");
```

To write to a log file, use `gnsdk_manager_logging_write()` or `gnsdk_manager_logging_vwrite()`.

The logging system determines the applicability of a log message for each enabled log, and writes a log message to multiple enabled logs, if appropriate. In cases where filters have overlapping functionality, the system writes the log message(s) to all applicable enabled logs.

The most typical use case for GNSDK logging is to configure a single log file to capture all logged messages and errors. You can control the detail level of the run time logs to include specific information (such as logging only errors or full debug information).

However, you can also direct GNSDK to allow a logging callback, where you can determine how best to capture and disseminate specific logged messages. For example, your callback function could write to its own log files or pass the messages to an external logging framework, such as the Unix Syslog or the Windows Event Log.

After an SDK API call fails, your application can access additional error information from the SDK with the `gnsdk_manager_error_info()` API:

```
/**  
/** Make call to get system error information for logging  
/**  
const gnsdk_error_info_t* err_info = GNSDK_NULL;  
err_info = gnsdk_manager_error_info();
```

This structure contains both an 'error_code' field and a 'source_error_code' field. The 'error_code' is what the call returns, while 'source_error_code' is internal to the SDK and may help Gracenote diagnose any difficulties your application may be experiencing. In most cases though, these fields will be the same.

Example: Logging

Code Snippet: [logging.c](#)

Application Steps:

1. Initialize GNSDK Manager.
2. Register package ID with logging system so entries show up with the package description of "Sample App".
3. Enable logging - indicate what level, packages to log.
4. Initialize MusicID.
5. Get User handle.
6. Use SDK to make MusicID query.
7. During step 6, deliberately create `gnsdk_musicid_query_set_gdo()` error.
8. Check for errors at each step and write error to log, if any.
9. Release resources and shutdown SDK.

After program completes, check "sample.log" for logged messages. Among various informational messages, you should see an entry for the deliberately created error:

```
Thu Nov 01 14:16:00 2012 ERROR Sample App 0x00001F28 main.c[151]  
API:gnsdk_musicid_query_set_gdo,  
SDK error code: 0x90810001, Description: Invalid argument, Internal  
error code: 0x90810001
```

Callbacks

GNSDK provides callback functions to help you get updates on different transactions and handle them. For example, when creating any query, you can register a status callback, which is called by GNSDK at each stage of the query (sending query metadata, receiving query metadata, and so on). You can also use the status callback to abort the query if needed.

Example: Making Callbacks

This example demonstrates setting a status callback for a MusicID query.

Code Snippet: [callback.c](#)

Application Steps

1. Initialize GNSDK.
2. Create MusicID query.
3. Register a callback.
4. Set query input.
5. Set query options.
6. Perform query.
7. Display query status updates using callback.

Example: Logging to a Callback

This example shows setting up a logging callback that echoes the messages to the system log (Syslog). It shows how to configure the GNSDK logging functionality with a custom callback that enables the application complete control over how logging messages are captured.

The most typical use case for GNSDK logging is to configure a single log file to capture all logged messages and errors. You can control the detail level of the run time logs to include specific information (such as logging only errors or full debug information).

However, you can also direct the GNSDK to allow a logging callback, enabling you to determine how best to capture the specific logged message(s). In this callback, the application can write to its own log files or pass the messages to an external logging framework, such as the Unix Syslog or the Windows Event Log.

You can configure the GNSDK to have any number of logging destinations (0-n log files and 0-n log callbacks). You may decide to send important error messages to the Syslog while specifying a run-time-enabled file to capture more detailed information during development or bug tracking.

Code Snippet: [logging_to_callback.c](#)

Application Steps:

1. Open the Syslog.
2. Initialize GNSDK.

3. Register callback with the GNSDK Manager.
4. Perform the application function.
5. Shut down GNSDK.
6. Close the Syslog.

Locales

GNSDK provides *locales* as a convenient way to group locale-dependent metadata specific to a region (such as Europe) and language that should be returned from the Gracenote service. A locale is defined by a group (such as Music), a language, a region and a descriptor (indicating level of metadata detail), which are identifiers to a specific set of *lists* in the Gracenote Service.

Using locales is relatively straightforward for most applications to implement. However, it is not as flexible or complicated as accessing lists directly - most locale processing is handled in the background and is not configurable. For most applications though, using locales is more than sufficient. Your application should only access lists directly if it has a specific reason or use case for doing so. For information about lists, See "Using Lists" on page 91.

Loading a Locale

To load a locale, use the `gnsdk_manager_locale_load()` function. As can be seen in the sample below, Locale properties are:

- **Group** Group type of locale such as Music or Playlist that can be easily tied to the application's use case
- **Region** Region the application is operating in, such as US, China, Japan, Europe, and so on, possibly specified by the user configuration
- **Language** Language the application uses, possibly specified by the user configuration
- **Descriptor** Additional description of the locale, such as Simplified or Detailed for the list hierarchy group to use, usually determined by the application's use case

For example:

- A locale defined for the USA of English/ US/Detailed returns detailed content from a list written in English for a North American audience.
- A locale defined for Spain of Spanish/Global/Simplified returns list metadata of a less-detailed nature, written in Spanish for a global Spanish-speaking audience (European, Central American, and South American).

To configure the locale:

- Set the group key to the respective `GNSDK_LOCALE_GROUP_*`.
- Set the language key (`GNSDK_LANG_*`) to the required language.
- Set the region and descriptor keys to the respective `GNSDK_*_DEFAULT` key.

For example:


```

gnsdk_manager_locale_load(
    GNSDK_LOCALE_GROUP_MUSIC,          // Group - Music (others include EPG,
    Playlist and Video)
    GNSDK_LANG_ENGLISH,                // Language - English
    GNSDK_REGION_DEFAULT,              // Default is US (others include China,
    Japan, Europe, and so on)
    GNSDK_DESCRIPTOR_DETAILED,         // Default music descriptor is
    'detailed' (versus 'simplified')
    user_handle,                       // User handle
    GNSDK_NULL,                        // No status callback
    GNSDK_NULL,                        // No status userdata
    @locale_handle                     // Locale handle to be set
);

```

Locale Groups

Setting the locale for a group causes the given locale to apply to a particular media group, such as Music or Playlist. For example, setting a locale for the Music group applies the locale to all music-related objects. When a locale is loaded, all lists necessary for the locale group are loaded into memory.

The locale group property can be set to one of the following values:

- GNSDK_LOCALE_GROUP_MUSIC: Sets the locale for all music-related objects
- GNSDK_LOCALE_GROUP_PLAYLIST: Sets the locale for playlist generation

Once a locale has been loaded, you must call one of the following functions to set the locale before retrieving locale-dependent values from a GDO:

- gnsdk_manager_locale_set_group_default(): This function sets a default locale. When a locale is set to be the default, it becomes the default locale for its inherent group. So, you can set a default for each locale group, such as Music, Playlist, etc. The default locale is automatically applied to each new GDO (that is relevant for that locale group). Setting a locale manually for a GDO (using gnsdk_manager_gdo_set_locale) overrides the default locale.
- gnsdk_manager_gdo_set_locale(): This function sets the locale of the locale-dependent data for a specific GDO handle.

Locale-Dependent Values and List Types

The table below summarizes locale-dependent value keys and their corresponding list types. The list type values actually returned depend on the type of GDO you are working with. You can load lists using gnsdk_manager_gdo_set_locale().

List types are categorizations of related list metadata. For example, GNSDK_LIST_TYPE_MOODS contains a hierarchical list of moods for audio metadata, such as Blue (Level 1) and Earthy/Gritty/Soulful (Level 2).

Locale-Dependent Genre Levels

The Gracenote Genre System provides a locale-dependent view of the genre hierarchy based on the user's geographic location or cultural preference. This allows you to deliver localized solutions for consumers in different parts of the world. Localized solutions allow representation and navigation of music in a manner that is expected in that region.

For example, consumers in the U.S. would expect to find Japanese or French Pop music in a World genre category, while North American Pop would be expected to be labeled as Pop. In Japan, consumers would expect to find Japanese Pop under Pop and French and North American Pop under Western Pop. In a solution shipped globally, all Pop music would be categorized as Pop, regardless of the origin of the music.

Music Locale-Dependent Values and List Types

Locale/List-Dependent Values	Locale/List Types
GNSDK_GDO_VALUE_ARTISTTYPE_LEVEL1	GNSDK_LIST_TYPE_ARTISTTYPES
GNSDK_GDO_VALUE_ARTISTTYPE_LEVEL2	
GNSDK_GDO_VALUE_COMPOSITION_FORM	GNSDK_LIST_TYPE_COMPOSITION_FORM
GNSDK_GDO_VALUE_ENTITY_TYPE	GNSDK_LIST_TYPE_CON- TRIBUTORENTITYTYPES
GNSDK_GDO_VALUE_ERA_LEVEL1	GNSDK_LIST_TYPE_ERAS
GNSDK_GDO_VALUE_ERA_LEVEL2	
GNSDK_GDO_VALUE_ERA_LEVEL3	
GNSDK_GDO_VALUE_GENRE_LEVEL1	GNSDK_LIST_TYPE_GENRES
GNSDK_GDO_VALUE_GENRE_LEVEL2	
GNSDK_GDO_VALUE_GENRE_LEVEL3	
GNSDK_GDO_VALUE_INSTRUMENTATION*	GNSDK_LIST_TYPE_INSTRUMENTATION
GNSDK_GDO_VALUE_MOOD_LEVEL1	GNSDK_LIST_TYPE_MOODS
GNSDK_GDO_VALUE_MOOD_LEVEL2	
GNSDK_GDO_VALUE_ORIGIN_LEVEL1	GNSDK_LIST_TYPE_ORIGINS
GNSDK_GDO_VALUE_ORIGIN_LEVEL2	
GNSDK_GDO_VALUE_ORIGIN_LEVEL3	
GNSDK_GDO_VALUE_ORIGIN_LEVEL4	

Locale/List-Dependent Values	Locale/List Types
GNSDK_GDO_VALUE_PACKAGE_LANGUAGE_DISPLAY	GNSDK_LIST_TYPE_LANGUAGES
GNSDK_GDO_VALUE_ROLE	GNSDK_LIST_TYPE_CONTRIBUTORS
GNSDK_GDO_VALUE_ROLE_CATEGORY	
GNSDK_GDO_VALUE_ROLE	GNSDK_LIST_TYPE_ROLES
GNSDK_GDO_VALUE_ROLE_CATEGORY	
GNSDK_GDO_VALUE_TEMPO_LEVEL1	GNSDK_LIST_TYPE_TEMPOS
GNSDK_GDO_VALUE_TEMPO_LEVEL2	
GNSDK_GDO_VALUE_TEMPO_LEVEL3	

*GNSDK_GDO_VALUE_INSTRUMENTATION will be removed in future releases.

Multi-Threaded Access

Since locales and lists can be accessed concurrently, your application has the option to perform such actions as generating a Playlist or obtaining result display strings using multiple threads.

Typically, an application loads all required locales at start up, or when the user changes preferred region or language. To speed up loading multiple locales, your application can load each locale in its own thread.

Updating Locales and Lists

GNSDK includes support for storing locales and their associated lists locally. Doing this improves access times and performance. Your application needs to include a database module (such as SQLite) to implement local storage. See "GNSDK Storage and Caching" on page 27 for more information.



Periodically, your application should update any locale lists you are storing locally. Currently, Gracenote lists are updated no more than twice a year. However, Gracenote recommends that applications run an update with the `gnsdk_manager_locale_update()` function *every 14 days*.

If the SDK infers your locale lists are out of date, it will return a `GNSDKERR_ListUpdateNeeded` error code. This error is only returned if your application attempts to access metadata via a response GDO that cannot be resolved.



Updates require the user option `GNSDK_USER_OPTION_LOOKUP_MODE` to be set to `GNSDK_LOOKUP_MODE_ONLINE` (default) or `GNSDK_LOOKUP_MODE_ONLINE_ONLY`. This allows the SDK to retrieve lists from the Gracenote service. You may need to toggle this option value for the update process.

Best Practices

Practice	Description
Applications should use locales.	Locales are simpler and more convenient than accessing lists directly. An application should only use lists if there are specific circumstances or use cases that require it.
Applications can deploy with pre-populated list stores and reduce startup time.	<p>On startup, a typical application loads locale(s). If the requested locale is not cached, the required lists are downloaded from the Gracenote service and written to local storage. This procedure can take time.</p> <p>Customers should consider creating their own list stores that are deployed with the application to decrease the initial startup time and perform a locale update in a background thread once the application is up and running.</p>
Use multiple threads when loading or updating multiple locales.	Loading locales in multiple threads allows lists to be fetched concurrently, reducing overall load time.
Update locales in a background thread.	<p>Locales can be updated while the application performs normal processing. The SDK automatically switches to using new lists as they are updated.</p> <div>  <p>If the application is using the GNSDK Manager Lists interface directly and the application holds a list handle, that list is not released from memory and the SDK will continue to use it.</p> </div>
Set a <i>persistence</i> flag when updating. If interrupted, repeat update.	<p>If the online update procedure is interrupted (such as network connection/power loss) then it must be repeated to prevent mismatches between locale required lists.</p> <p>Your application should set a persistence flag before starting an update procedure. If the flag is still set upon startup, the application should initiate an update. You should clear the flag after the update has completed.</p>
Call <code>gnsdk_manager_storage_compact()</code> after updating lists or locales.	<p>As records are added and deleted from locale storage, some storage solutions, such as SQLite, can leave empty space in the storage files, artificially bloating them. You can call <code>gnsdk_manager_storage_compact()</code> to remove these.</p> <div>  <p>The update procedure is not guaranteed to remove an old version of a list from storage immediately because there could still be list element references which must be honored until they are released. Therefore, your application should call <code>gnsdk_manager_storage_compact()</code> during startup or shutdown after an update has finished.</p> </div>

Example: Accessing a Locale-Dependent Value

The basic steps to access locale-specific values from query results are:

1. Load the desired locale(s) to ensure all necessary lists are in memory.
2. (Optional) Set a default locale.
3. (Optional) Set a locale on a GDO.
4. Access the locale-specific values.

The example below demonstrates loading a locale to access an album genre, which is a locale-specific value. It shows the results of querying for a locale-specific value with the correct and incorrect locale. If locale is not loaded when trying to get a locale dependent value (for example, an album genre), GNSDK returns an error (locale not loaded).

Code Snippet: [locale_mgmt_1.c](#)

Application Steps:

1. Initialize GNSDK Manager, MusicID library, and register a User.
2. Perform Music ID query to an the album response GDO, and the child album GDO.
3. Grab a locale-specific value (genre) without first loading a locale (generates error).
4. Load an English locale and set the GDO locale.
5. Load the same locale-specific value and display it (no error this time).
6. Shutdown GNSDK Manager, MusicID, and release User handle.

Example: Overriding the Default Locale

This example demonstrates overriding a default locale to access an album primary genre in a secondary locale.

Code Snippet: [local_mgmt_2.c](#)

Application Steps:

1. Initialize GNSDK Manager, MusicID Library, and register a User.
2. Load two music locales - Music/English/Detailed and Music/Chinese/Simplified.
3. Perform a Music ID query.
4. Get album primary genre (a locale dependent value) from work GDO and display in English.
5. Set child album GDO locale to Chinese and get/display same value.
6. Shutdown GNSDK Manager, MusicID Library and release User handle.

Best Practices for Images

Gracenote images – in the form of cover art, artist images and more – are integral features in many online music services, as well as home, automotive and mobile entertainment devices. Gracenote maintains a comprehensive database of images in dimensions to accommodate all popular applications, including a growing catalog of high-resolution (HD) images.

Gracenote carefully curates images to ensure application and device developers are provided with consistently formatted, high quality images – helping streamline integration and optimize the end-user experience. This topic describes concepts and guidelines for Gracenote images including changes to and support for existing image specifications.

Image Formats and Dimensions

Gracenote provides images in several dimensions to support a variety of applications. Applications or devices must specify image size when requesting an image from Gracenote. All Gracenote images are provided in the JPEG (.jpg) image format.

Standard Image Dimensions

In general, Gracenote provides images to fit within the following square dimensions.

Image Dimension Name	Pixel Dimensions
75	75 x 75
170	170 x 170
300	300 x 300
450*	450 x 450
720*	720 x 720
1080*	1080 x 1080

*Images with dimensions 450 x 450 and higher are available online only, or by special request.



Source images are not always square, and may be proportionally resized to fit within the specified square dimensions. Images will always retain their original aspect ratio.

Album Cover Art

While album cover art is often represented by a square, it is more accurately a bit wider than it is tall. The dimensions of these cover images vary from album to album. Some CD packages, such as a box set, might even be a radically different shape.

Using a Default Image When Cover Art Is Missing

Occasionally, an Album result might have missing cover art. If the cover art is missing, Gracenote recommends trying to retrieve the artist image, and if that is not available, trying to retrieve the genre image. If none of these images are available, your application can use a default image as a substitute. Gracenote distributes an clef symbol image to serve as a default replacement. The images are in jpeg format and located in the /images folder of the package. The image names are:

- music_75x75.jpg
- music_170x170.jpg
- music_300x300.jpg

Artist and Contributor Images

Artist and Contributor images, such as publicity photos, come in a wide range of sizes and both portrait and landscape orientations.

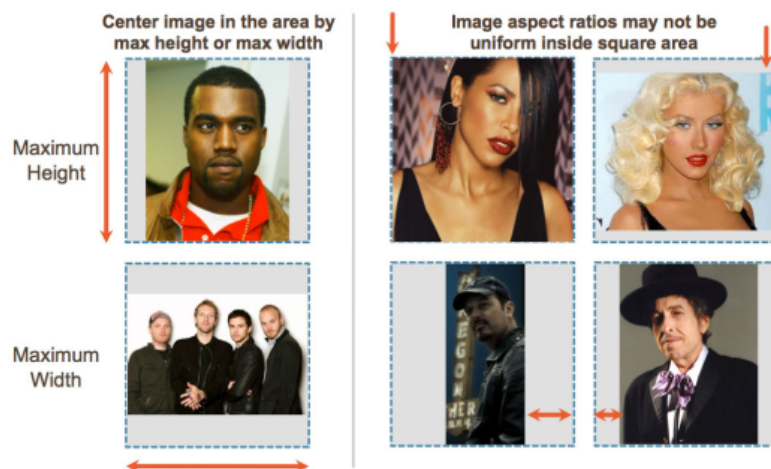
Image Resizing Guidelines

Gracenote images are designed to fit within squares as defined by the available image dimensions. This allows developers to present images in a fixed area within application or device user interfaces. Gracenote recommends applications center images horizontally and vertically within the predefined square dimensions, and that the square be transparent such that the background shows through. This results in a consistent presentation despite variation in the image dimensions. To ensure optimum image quality for end-users, Gracenote recommends that applications use Gracenote images in their provided pixel dimensions without stretching or resizing.

Gracenote resizes images based on the following guidelines:

- **Fit-to-square:** images will be proportionally resized to ensure their largest dimension (if not square) will fit within the limits of the next lowest available image size.
- **Proportional resizing:** images will always be proportionally resized, never stretched.
- **Always downscale:** smaller images will always be generated using larger images to ensure the highest possible image quality

Following these guidelines, all resized images will remain as rectangles retaining the same proportions as the original source images. Resized images will fit into squares defined by the available dimensions, but are not themselves necessarily square images.



Using the Sample Applications and Databases

GNSDK provides working, command-line C-based sample applications that demonstrate common queries and application scenarios.

The package also provides sample databases you can use when developing your applications.

Gracenote recommends stepping through the sample applications with a debugger to observe module usage and API calls.



The sample applications are designed for simplicity and clarity, and not for use in real-world applications.

Code Snippets, Sample Applications, and Reference Applications

Below is a list of code snippets, sample applications, and reference applications for GNSDK. Sample and reference applications are included in the release package in the /samples and /reference_applications folders. Code snippets are part of the documentation and are not included.

- *Code snippets* demonstrate specific tasks to reinforce documented processes.
- *Sample applications* demonstrate common queries to identify media elements and access meta-data.
- *Reference applications* are more extensive and demonstrate specialized or more comprehensive use cases.

Code Snippets

Description	Link/Name
Look up Album cover art	samples/code_snippets/album_content_lookup.c
Set up caching	samples/code_snippets/cache.c
Set up callbacks	samples/code_snippets/callback.c
Navigate an Album's GDO and get its metadata	samples/code_snippets/gdo_get_metadata.c
Serialize GDOs	samples/code_snippets/gdo_serialize.c
Render GDOs as XML	samples/code_snippets/gdo_xml_render.c
Initialization	samples/code_snippets/initialization.c
	samples/code_snippets/initialization_multiple.c
Use Lists	samples/code_snippets/list.c

Description	Link/Name
Use Locales	samples/code_snippets/locale_mgmt_1.c
	samples/code_snippets/locale_mgmt_2.c
Set up logging	samples/code_snippets/logging.c
	samples/code_snippets/logging_to_callback.c

Sample Applications

Modules Used	Description	Link/Name
Link	Retrieves an Album cover art image using Link starting with a serialized GDO as source.	samples/link_coverart/main.c
MusicID	Uses MusicID to look up Album GDO content, including Album artist, credits, title, year, and genre. It demonstrates how to navigate the album GDO that returns basic track information, including artist, credits, title, track number, and genre.	samples/musicid_gdo_navigation/main.c
	Looks up an Album using a TUI in the local database, if not found then performs an online lookup.	samples/musicid_lookup_album_local_online/main.c
	Looks up an Album based on its TOC using either a local database or online.	samples/musicid_lookup_album_toc/main.c
MusicID-File	Uses AlbumID for advanced audio recognition. The audio file's folder location and its similarity to other audio files are used to achieve more accurate identification.	samples/musicid_file_albumid/main.c
	Uses LibraryID to perform additional scanning and processing of all the files in an entire collection. This enables LibraryID to find groupings that are not captured by AlbumID processing	samples/musicid_file_libraryid/main.c
	Uses TrackID, the simplest MusicID-File processing method to process each audio file independently, without regard for any other audio files in a collection.	samples/musicid_file_trackid/main.c

Reference Applications

Modules Used	Description	Name
Playlist	Demonstrates the MoreLikeThis use-case using Playlist and MoodGrid.	reference_apps/MoodGrid_MoreLikeThis

Building a Sample Application

GNSDK provides sample applications for many common use cases. You can find the sample applications in the `samples/<module>` folders of your GNSDK package, where `<module>` corresponds to GNSDK modules and features, like `musicid_lookup_album_toc`, `musicid_file_albumid`, and so on.

Each sample application folder contains the following files:

- `main.c`: C source for the sample application
- `makefile`: GNU Make makefile for building the sample application

Some modules may contain a `/data` subfolder if metadata exists for the sample application.

The GNSDK package also contains following makefiles:

- GNSDK Sample Makefile, `makefile`, an introductory makefile that includes `makefile_platforms.inc`. Type `make` to view its contents.
- GNSDK Build file, `makefile_platforms.inc`
- Common GCSL module Sample Makefile, `makefile_sample.inc`

Each sample application makefile includes `makefile_platforms.inc` and `makefile_sample.inc`.

You can build sample applications on any of the GNSDK supported platforms. See "System Requirements"

Build Environment Requirements

Building the sample applications requires the following environment:

- GNU Make 3.81 or above
- A command-line environment supported by the target platform

Build Steps

To build and run a sample application:

1. Open a shell and navigate to the sample application folder.
2. Type `make`. This creates a **sample.exe** file (and generally other output files) in the same folder. Object files from the compiler are stored in an output folder.
3. Type the platform-specific command to run the `sample.exe` and include the ClientID, ClientID Tag, and License file information as command-line arguments as described below.

Supported make commands

Command	Description
<code>make</code>	Builds a debug version
<code>make release</code>	Builds a release (non-debug) version
<code>make release CFLAGS=-DUSE_LOCAL</code>	Enables local database lookups

Including Client and License Information to Run a Sample

Running the sample application generally requires the client ID, client ID tag, and license file provided with the SDK. Gracenote SDKs and sample applications cannot successfully execute without these values. You must supply them as command-line arguments to the sample application, in the following format:

```
./sample.exe <clientid> <clientid_tag> <license_file>
```

For example,

```
./sample.exe 12345678 ABCDEF0123456789ABCDEF012345679 "C:/gn_license.txt"
```

Directing Output and Log Files

Each sample application output varies, but in general, you should see the application performing online querying and displaying scrolling output. Redirect the output to a file to capture it for viewing. The application also creates a `sample.log` file, which logs any errors that may have occurred.

Platform-Specific Build Instructions

This section contains important build notes for specific platforms.

Windows

Building on a Windows platform requires the following:

- Cygwin: For more information, see the file `README_windows.txt`, included in the package.
- Visual Studio 2005 or Visual Studio 2008

To build the sample application on a Windows platform:

1. Edit the cygwin.bat file to configure the correct variables for the installed Visual Studio version. (The default location for this file is generally C:\cygwin, but this is dependent on your specific configuration.) See examples below.
2. Follow the general instructions: See "Build Steps" on page 42.
3. Be sure to navigate to the sample folder using the proper Cygwin path prefix of /cygdrive/c <or correct drive>/<path to sample>. For more information on using Cygwin, see the [Cygwin's User's Guide](#).

This example shows the original code delivered by Cygwin in the cygwin.bat file:

```
@echo off  
  
C:  
  
chdir C:\cygwin\bin  
  
bash --login -i
```

This example shows how the code must be edited to call the correct variables for the installed Visual Studio version. Note that this code specifically calls Visual Studio 2008 and configures the variables before initializing Cygwin:

```
@echo off  
  
:: Change this call based on the version of Visual Studio being used  
:: If using Visual Studio 2008, call "%VS90COMNTOOLS%\.." "  
:: If using Visual Studio 2005, call "%VS80COMNTOOLS%\.." "  
call "%VS90COMNTOOLS%\..\VC\vcvarsall.bat" x86  
  
C:  
  
chdir C:\cygwin\bin  
  
bash --login
```

Windows CE

Because Windows CE does not have a relative path concept, the samples provided with GNSDK will not run without modifications. The following modifications are necessary:

- You must use an absolute path for the log files.
- You must set a path name for the database after SQLite is initialized, for example:

```
gnsdk_storage_sqlite_option_set(GNSDK_STORAGE_SQLITE_OPTION_STORAGE_ FOLDER, "/storage card/samples");
```

- You must set a path name for the database after Playlist is initialized, for example:

```
gnsdk_playlist_storage_location_set( "/storage card/samples");
```

You also might need to hard-code the full path name for license.txt:

```
error = gnsdk_manager_initialize(  
    &sdkmgr_handle,  
    "/storage card/license.txt",  
    GNSDK_MANAGER_LICENSEDATA_FILENAME  
);
```

The recommended way to run the samples is to create a .bat file with a line similar to this:

```
/storage card/samples/playlist/sample.exe 12345678  
ABCDEF0123456789ABCDEF012345679 "\Storage card\samples\license.txt"
```

To run the samples:

1. Launch a command prompt.
2. Change directories (cd) to the running directory.
3. Invoke the .bat file to run the sample.

Optionally, you can redirect the output: ./sample.bat >sample.out



NOTE: Redirecting the output will not capture the leak statistics.

Linux ARM

To build samples when using the Linux ARM-32 binaries:

1. Open a shell and navigate to the sample application folder:../gnsdk_<version>_<date>/_sample/<module>.
2. Type make clean ARCH=arm.
3. Type make ARCH=arm to build the executable file.
4. Navigate to the../gnsdk_<version>_<date>/_sample/linux_arm-32 folder to access the gnsdk_<module>_sample.exe and run the sample.

Linux and Solaris

Follow the general instructions in See "Build Steps" on page 42.

MacOS

Follow the general instructions in See "Build Steps" on page 42.

To build samples when using Mac OS X x86_64 binaries, use the command:

make ARCH=x86_64

MusicID Sample Application Walkthrough

GNSDK MusicID sample application

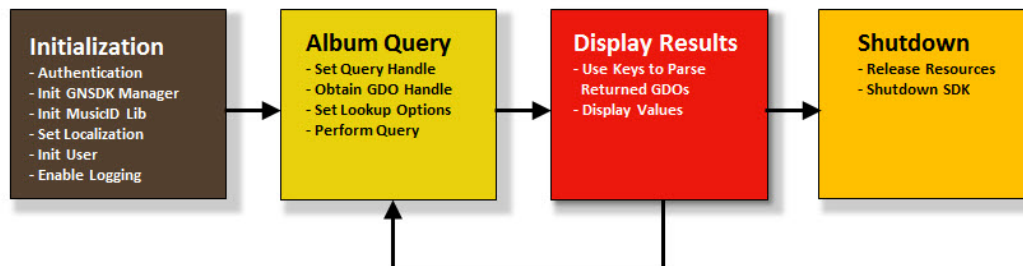
This topic steps through the GNSDK sample application called `musicid_gdo_navigation`.

Sample Application: [musicid_gdo_navigation/main.c](#)

This application uses MusicID to look up Album content, navigate the hierarchy of data returned, and extract and display metadata results. As shown in the following image, most of the sample applications follow a similar pattern—authenticate/initialize, query Gracenote Service, parse and display results, and shut down.

This sample application:

- Initializes the GNSDK Manager, the MusicID library and local storage, enables logging, sets localization options, and registers the application user with the Gracenote Service.
- Performs album lookup queries using internal Gracenote identifiers.
- Parses returned Gracenote Data Objects (GDOs) and displays the results.
- Releases resources and shuts down the GNSDK when done.



In this topic, error handling is minimized and function calls are sometimes shown without some of their parameters. Refer to the sample application code to see how to implement error handling, and see the actual function call syntax.

Prerequisites

To build and run any of the sample applications, See "Building a Sample Application" on page 42. In general, you need three things from Gracenote:

1. License File—Details your permissions and access to GNSDK libraries.
2. Client ID—Your specific GNSDK access identifier.
3. Client Tag—A hashed representation of the Client ID that guarantees its authenticity.

These three items are used to authenticate your application's access to MusicID as well as other GNSDK libraries. You pass them when you invoke a sample application program at the command-line:

```
> sample.exe <clientid> <clientidtag> <licensefile>
```

Initialization

After `main()` does some initial error checking on the Client ID, Client Tag and License File parameters, it calls `_init_gnsdk()` which makes a number of initialization calls.

Initialize the GNSDK Manager

Your application must initialize the GNSDK Manager prior to calling any other GNSDK library. The GNSDK Manager is the central controller for your application's interaction with the Gracenote Service. The initialization call returns a GNSDK Manager handle for use in subsequent SDK calls, including any other initialization calls. This is a required call.

```
/**
/**  Initialize the GNSDK Manager
/**
gnsdk_manager_handle_t sdkmgr_handle= GNSDK_NULL;
gnsdk_manager_initialize(&sdkmgr_handle, license_path, GNSDK_MANAGER_
LICENSEDATA_FILENAME);
```



The `GNSDK_MANAGER_LICENSEDATA_FILENAME` define is passed for the parameter that is supposed to indicate the license string length but, instead, indicates that the license data should be read from a file.

Initialize the MusicID Library Manager

After getting an SDK Manager handle, the application then initializes its interaction with the Gracenote MusicID library. Every GNSDK library must be initialized before an application can successfully call any of its APIs.

```
//Initialize the MusicID Library
gnsdk_musicid_initialize(sdkmgr_handle);
```

Initialize Storage and Local Lookup

The following two calls initialize the application's interaction with local storage. Both take the GNSDK Manager handle as a parameter. The SDK uses storage to cache queries (which improves online performance) and to power local lookups. Currently, SQLite is the only database provided for local storage and requires initialization. This must be done before the local lookup initialization call.

```
// Initialize the Storage SQLite Library
gnsdk_storage_sqlite_initialize(sdkmgr_handle);

// Initialize the storage folder (/sample_db) for Local Lookup
gnsdk_storage_sqlite_option_set(GNSDK_STORAGE_SQLITE_OPTION_STORAGE_FOLDER,
"./sample_db");

// Initialize the Lookup Local Library
gnsdk_lookup_local_initialize(sdkmgr_handle);
```

Initialize the User Handle

Every application user is required to register with the Gracenote Service. To perform queries, an application must first register a new user and get its handle. A user represents an individual installation of a specific Client ID. This ensures that each application instance is receiving all required metadata entitlements. Users are represented in GNSDK by their handles. These handles contain the Client ID string. The `_init_gdsdk()` function calls `_get_user_handle()` to either create a new user handle or restore a user handle from serialized storage.



After your application creates a new user, it should save its handle to serialized storage, where it can be retrieved every time your application needs it to use again. If an application registers a new user on every use instead of storing a serialized user, then the user quota maintained for the Client ID is quickly exhausted. Once the quota is reached, attempting to create new users will fail. To maintain an accurate usage profile of your application, and to ensure that the services you are entitled to are not being used unintentionally, it is important that your application registers a new user only when needed, and then stores that user for future use.

To register as a new user, your application can call `gnsdk_manager_user_create_new()`

For user serialization, your application can use the following two calls:

- `gnsdk_manager_user_create()`—Takes a serialized user string and creates a user handle.
- `gnsdk_manager_user_serialize()`—Saves user handle to serialized storage so it can be re-used in the future without having to register as a new user. Generally, this should be done when the application exits as the user handle can change during the course of the app. The user handle release call - `gnsdk_manager_user_release()` - will also serialize the user handle.

Initialize Localization

Finally, this sample application makes the following localization calls. Note that these calls can be done at anytime, but must be done *after* user registration, since they require a user handle parameter.

- `gnsdk_manager_locale_load()`—Sets locale and creates a locale handle for subsequent calls. GNSDK locales are identifiers to a specific set of lists in the Gracenote Service. By using a locale, an application instructs the Gracenote Service to return only the data contained in a specific list. A locale is defined by a language and (optionally) a list region, a list descriptor, and a group. This sample application sets language to English (`GNSDK_LANG_ENGLISH`) and the region to default (`GNSDK_REGION_DEFAULT`).
- `gnsdk_manager_locale_set_group_default()`—Sets a locale's global group default; all GDOs will use this locale unless specifically overwritten by `gnsdk_manager_gdo_set_locale()`. If the default is not set, no locale-specific results would be available. The locale group was set in the local handle with the previous call when the `GNSDK_LOCALE_GROUP_MUSIC` was passed.

MusicID Queries

An application can make a MusicID identification query in several ways, including text lookups, TOC lookups, fingerprint lookups, and so on. For a complete list of these options and examples, See "About MusicID" on page 57.

After identifying an element, Gracenote recommends using GDOs (Gracenote Data Objects) to do additional navigation and retrieve metadata. For information about GDOs, See "Gracenote Data Objects (GDOs)" on page 16.

However, in addition to GDOs, there are several unique identifier types that can access Gracenote media elements. To learn more about these non-GDO Gracenote identifiers, See "Working with Non-GDO Identifiers" on page 87 .



This sample application (`musicid_gdo_navigation`) uses unique identifiers called TUIs to perform the initial identification (lookup) query. TUI stands for "Title Unique Identifier" and is used for internal Gracenote identification. Compared to text lookups, TOC lookups, and fingerprint lookups, TUIs are rarely used. They are used here as a convenient way to find a specific Album.

Album Lookup

After initialization, this sample application calls `_do_sample_tui_lookups()` to perform a number of album lookups.

In `_perform_sample_album_tui_lookup()`, the sample application makes the following GNSDK calls to query a specific album:

- `gnsdk_musicid_query_create()`—Create the query handle.
- `gnsdk_manager_gdo_create_from_id()`—Obtain GDO handle. This takes the TUI and TUI tag parameters that uniquely identify an album. Under the hood, this creates a stub GDO that is used, in turn, for the query method.
- `gnsdk_musicid_query_set_gdo()`—Add the stub GDO from the last call to the query handle.
- `gnsdk_musicid_query_option_set()`—Add the options for local lookup to the query handle.
- `gnsdk_musicid_query_find_albums()`—Perform the query.

GDO Navigation and Display

Each GNSDK identification query returns a GDO. GDO fields indicate how many matches were returned (none, single match, or multiple matches), which varies based on query criteria.

GDO metadata results can be either partial or full. Partial metadata may be sufficient for applications that only need to display basic information to the end user. Other applications may need full results. In all cases, applications should check to see if the GDO contains partial or full results and then determine if the partial is sufficient. For more information about full and partial results, See "Gracenote Data Objects (GDOs)" on page 16. For a list of partial results returned in GDOs, see the GNSDK Data Dictionary.

Parsing the Returned Response GDO

A query returns a Response GDO containing a child GDO for each match. Since we are querying on a specific Gracenote Identifier (TUI and TUI tag), we are only going to get one child GDO containing full metadata results. However, the code parses the returned GDO to handle the multiple matches possibility. This is done in the `perform_sample_album_tui_lookup()` function:

```
/**
/** Find number of matches using the GNSDK_GDO_VALUE_RESPONSE_RESULT_COUNT
value key
/**
gnsdk_manager_gdo_value_get(result_gdo,GNSDK_GDO_VALUE_RESPONSE_RESULT_
COUNT, 1, &value_string);
result_count = value_string ? atoi(value_string) : 0;

/**
/** Get child if result count > 0 using the GNSDK_GDO_CHILD_ALBUM child
key
/**
gnsdk_manager_gdo_child_get(result_gdo,GNSDK_GDO_CHILD_ALBUM, 1, &album_
result_gdo);

/**
/** Find if child GDO contains full or partial results using the GNSDK_
GDO_VALUE_FULL_RESULT value key.
/**
gnsdk_manager_gdo_value_get(album_result_gdo, GNSDK_GDO_VALUE_FULL_RESULT,
1, &value_string);

/**
/** If this GDO only contains partial metadata (atoi(value_string) == 0),
then re-query for the full results.
/** First, however, set match back to the existing query handle
/**
gnsdk_musid_query_set_gdo(query_handle, album_result_gdo);

/**
/** Query for this match in full
```

```
/**  
gnsdk_musid_query_find_albums(query_handle, &match_type, &result_gdo);
```

Parsing the Child GDO

Once we have a Response GDO with a child GDO containing full results for our album query, we can then extract the child GDO values and display them. This is done in the `_navigate_album_match_result_gdo()` function.

The query to get full results also returns a Response GDO. All queries return Response GDOs. This means that the `_navigate_album_match_result_gdo()` function has to repeat the call to get the child album GDO (see code snippet below). This time, however, the application is aware that the child GDO contains full results and does not need to re-test for this.

```
/**  
/** Get the child album GDO containing full results from the album  
Response GDO  
/**  
error = gnsdk_manager_gdo_child_count(result_gdo, GNSDK_GDO_CHILD_ALBUM,  
&child_count);  
  
if (child_count > 0)  
{  
    error = gnsdk_manager_gdo_child_get(result_gdo, GNSDK_GDO_CHILD_ALBUM,  
1, &album_gdo);  
    ...
```

After getting the child GDO, `_navigate_album_match_result_gdo()` does one of two things:

1. Extracts and displays values (metadata) from the GDO using value keys.
2. Calls functions to navigate additional child GDOs (sub-objects). These functions extract and display metadata values from those objects.

Extract Metadata and Display

The navigate album function calls `_display_gdo_value()` to extract and display a metadata value. The display function calls `gnsdk_manager_gdo_value()` to extract values from the GDO using defined value keys, for example, `GNSDK_GDO_VALUE_YEAR`, `GNSDK_GDO_VALUE_GENRE_LEVEL_1`, `GNSDK_GDO_VALUE_ALBUM_TRACK_COUNT`, and so on.



Your application does not need to free GDO values and they will remain valid until the GDO handle is released.

Navigate and Parse Additional Child GDOs

Functions are also called to parse child GDOs (sub-objects), for example, `_navigate_track_official_gdo()` (tracks), `_navigate_credit_gdo()` (credits), `_navigate_contributor_gdo()` (contributor), and so on. These

functions go through a similar process of parsing data and extracting and displaying results.

Releasing Resources and Shutting Down

Before the program exits, it calls `_shutdown_gnsdk()`, which releases the user handle and shuts down the MusicID library, local storage and the GNSDK:

```
gnsdk_manager_user_release(user_handle, &serialized_user_string);

// The sample app code (not shown) saves the serialized user handle for
// later use. Note that the
// release call serializes the user handle to the serialized_user_string
// parameter.

gnsdk_manager_string_free(serialized_user_string);
#if USE_LOCAL
    gnsdk_lookup_local_shutdown();
#endif
gnsdk_storage_sqlite_shutdown();
gnsdk_musicid_shutdown();
gnsdk_manager_shutdown();
```



Your application needs to pair each call to an `"_initialize"` function with a matching call to a `"_shutdown()"` function.

When you initialize other GNSDK libraries, be sure to shut them down before shutting down the GNSDK Manager. This ensures the application is in control of when individual libraries are shutdown. Otherwise, the other libraries will be forcibly shutdown as well. Generally, this is not an issue, but it may not be what your application is expecting.

Prior to calling `_shutdown_gnsdk()`, the locale handle is released:

```
/**
/** Clean up and shutdown
/**
gnsdk_manager_locale_release(locale_handle);
_shutdown_gnsdk(user_handle);
```

It is a best practice to release resources when they are no longer needed, for example:

```
/** Release GDO (in _navigate_name_official_gdo() )
gnsdk_manager_gdo_release(name_official
gdo);
/** Release Query Handle (in _perform_sample_album_tui_lookup() )
gnsdk_musicid_query_release(query_handle);
```

Logging

Note: The sample application has changed and no longer logs errors to a file. Now, it simply uses `printfs` to write them to the console. However, this topic has been left unchanged, to show how

logging using GNSDK calls can be done.

The `_init_gnsdk()` function calls `_enable_logging()`, which makes the following GNSDK logging call:

```
gnsdk_manager_logging_enable(  
    "sample.log",                // Log file path  
    GNSDK_LOG_PKG_ALL,          // Include entries for all  
    packages and subsystems  
    GNSDK_LOG_LEVEL_ERROR|GNSDK_LOG_LEVEL_WARNING, // Include only error and  
    warning entries  
    GNSDK_LOG_OPTION_ALL,        // All logging options:  
    timestamps, thread IDs, etc  
    0,                          // Max size of log: 0 means  
    a new log file will be created each run  
    GNSDK_FALSE                 // GNSDK_TRUE = old logs  
    will be renamed and saved  
);
```

As can be seen, your application has a number of logging options. The GNSDK logging system can manage multiple logs at the same time. Each call to this API can enable a new log, if the new log's provided file name is unique. Additionally, each log can have its own filters and options.

The most typical use case for GNSDK logging is to configure a single log file to capture all logged messages and errors. You can control the detail level of the run time logs to include specific information (such as logging only errors or full debug information).

However, you can also direct the GNSDK to allow a logging callback, where you can determine how best to capture and disseminate specific logged messages. For example, your callback function could write to its own log files or pass the messages to an external logging framework, such as the Unix Syslog or the Windows Event Log.

The GNSDK logging system writes log messages to all appropriate enabled logs, as applicable. This includes scenarios where filters have overlapping functionality; in this case, the logging system writes a log message to multiple logs.

Error Information

This sample application has a function—`_log_app_error()`—for logging error messages. This gets called when the Gracenote Service returns an error (for example, Return Code `!= GNSDK_SUCCESS`). The function first makes a call to get error information from GNSDK:

```
const gnsdk_error_info_t* err_info = GNSDK_NULL;  
err_info = gnsdk_manager_error_info();           /** Get the error info
```

To retrieve any error information, your application must call this function immediately after receiving an error from a GNSDK API. Though not all APIs provide error information, at a minimum, they return an error description string.

Writing to the Log File

You can use the following GNSDK API to log errors to a file:

```
gnsdk_manager_logging_write(
    line,                                // Source line number of
    this call (optional)                  // Source file name of
    file,                                // Source file name of
    this call (optional)                  // Package ID of appli-
    MYAPP_PACKAGE_ID,                    // Package ID of appli-
    cation making call                    // Error mask for this
    GNSDK_LOG_LEVEL_ERROR,               // Error mask for this
    logging message                       // Error message format
    "%s: 0x%x - %x. error info: %s: 0x%x - %s.", // Error message format
    info,                                // Error message format
    arguments                             // Error message format
    error,
    err_info->source_error_code,
    err_info->error_description,
    err_info->error_code,
    err_info->error_api
);
```

Log entries are identified by the 'package' that logs them using a self-defined package ID, for example:

```
#define MYAPP_PACKAGE_ID    GNSDKPKG_ID_APP_START+0x01
```

The package ID must be in the range GNSDKPKG_ID_APP_START to MAX_GNSDK_PKG_ID

For more information about logging, see the API Reference documentation for `gnsdk_manager_logging.h`

GNSDK Data Model for GNSDK

The GNSDK data model represents Gracenote media elements and metadata. The model establishes interrelationships among Gracenote Data Objects (GDOs) and the metadata they contain or reference.

A table-based version of the data model can be found in the online help system provided with GNSDK, in the GNSDK C API Reference, under GDO Type Mappings.

This table maps GDO types to their corresponding Child GDOs and values. The table also links directly to the corresponding GDO and value APIs documented in the GNSDK API Reference.

In addition to the main Data Model table, there are two other tables:

- Locale-Dependent Values
- GDO Partial Results

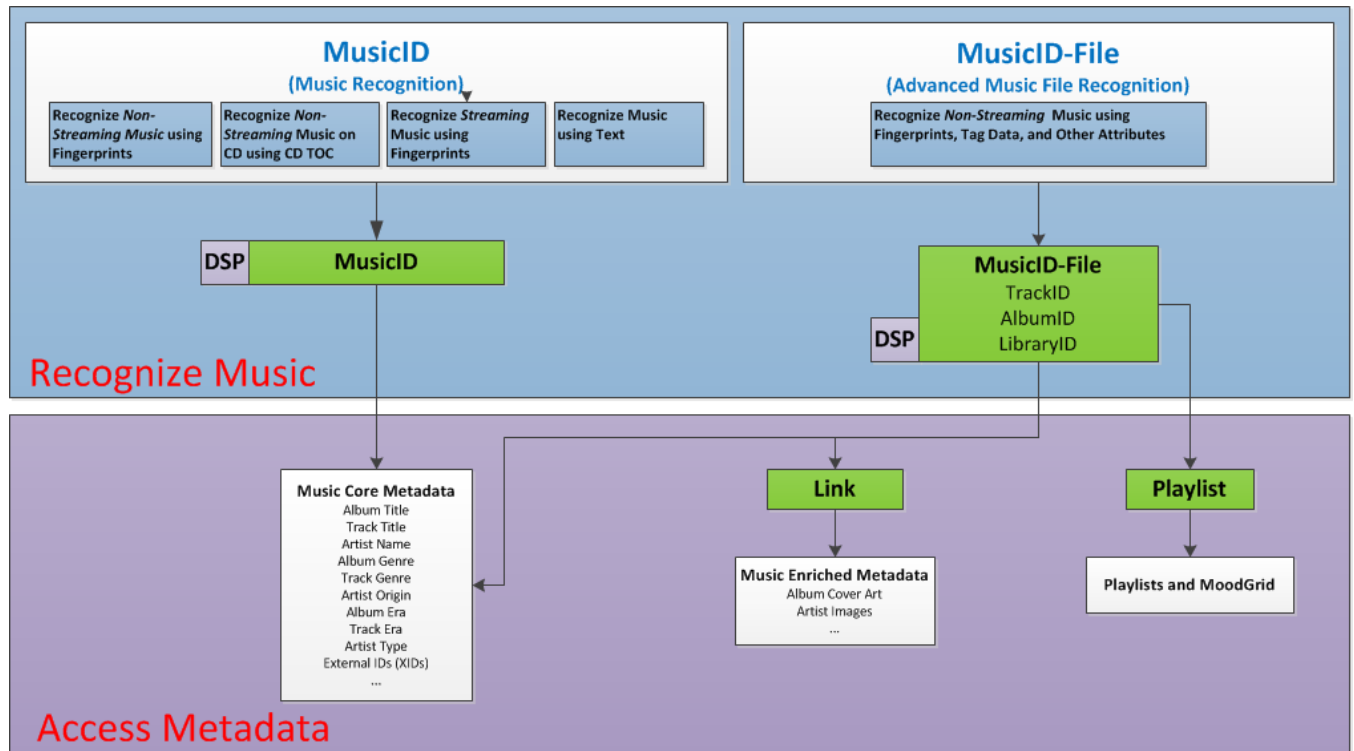
The Locale-Dependent Values table shows metadata values, like Genre, artist Origin, artist Era, and artist Type (collectively known as GOET values), as well as mood, tempo, contributor lists, roles, and others. All of these values are locale (or list) dependent. For more information about locale-dependent values and their corresponding List types, See "Locales" on page 32

The GDO Partial Results table shows values that are returned when a GDO result is not full. Your application can test if a GDO result is full using the `gnsdk_manager_gdo_value_get()` with Value Key `GNSDK_GDO_VALUE_FULL_RESULT`. For information about GDO partial and full results, See "Gracenote Data Objects (GDOs)" on page 16

Implementing Music Features

Working with Music Modules

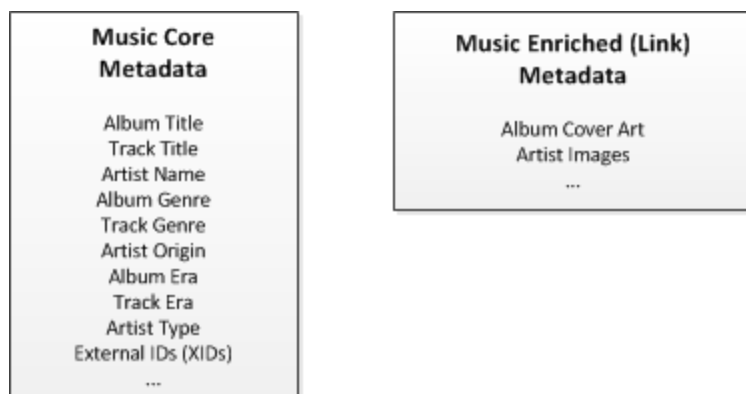
The following diagram shows the relationship between various Music modules and the queries they can perform and the metadata they can return.



Core and Enriched Music Metadata

All Gracenote customers can access core metadata from Gracenote Services for the products they license. Optionally, customers can access additional metadata, known as enriched metadata by purchasing additional metadata entitlements.

The following diagram lists the core and enriched metadata available for Music.



About MusicID

MusicID allows application developers to deliver a compelling digital entertainment experience by giving users tools to manage and enjoy music collections on media devices, including desktop and mobile devices. MusicID is the most comprehensive identification solution in the industry with the ability to recognize, categorize and organize any music source, be it CDs, digital files, or audio streams. MusicID also seamlessly integrates with Gracenote's suite of products and provides the foundation for advanced services such as enriched content and linking to commerce.

Media recognition using MusicID makes it possible for applications to access a variety of rich data available from Gracenote. After media has been recognized, applications can request and utilize:

- Album, track, and artist names
- Genre, origin, era and type descriptors

GNSDK accepts the following types of inputs for music recognition:

- CD TOCs
- File fingerprints
- Stream fingerprints
- Media element identifiers
- Audio file and folder information (for advanced music recognition)

MusicID-CD

MusicID-CD is the component of GNSDK that handles recognition of audio CDs and delivery of information including artist, title, and track names. The application provides GNSDK with the TOC from an audio CD and MusicID-CD will identify the CD and provide album and track information.

TOC Identification

The only information that is guaranteed to be on every standard audio CD is a Table of Contents (TOC). This is a "header" at the beginning of the disc giving the precise starting location of each track on the CD, so that CD players can locate the tracks and compute the track length information for their display panels. This information is given in frames, where each frame is 1/75 of a second. Because this number is so precise, the likelihood that two unrelated CDs would have the same TOC is extremely low – allowing GNSDK to use the TOC as a relatively unique identifier.

The following figure shows a typical TOC for a CD containing 13 tracks. The first number, 150, indicates the starting location of the first track, in frames, from the beginning of the disc (150 frames = 2 seconds). Each of the remaining 13 numbers is the length, in frames, of the corresponding track on the CD. The last number is the offset of the "lead out", which marks the end of the CD program area.

150 26670 52757 74145 95335 117690 144300 163992 188662 209375 231320 253150 281555 337792
--

The Gracenote MusicID module provides functions for sending a TOC to the Gracenote Media Database for identification, and receiving data back from the service in a form the client application can easily display for the end-user. The client application is responsible for extracting TOCs from CDs loaded in the end-user's CD-ROM drive.

Multiple TOCs and Fuzzy Matching

Gracenote MusicID utilizes several methods to perform TOC matches. This combination of matching methods allows client applications to accurately recognize media in a variety of situations.

- Exact Match – when there is only one Product match for a queried CD TOC
- Multi-Exact Match – when there are multiple Product matches for a queried CD TOC
- Fuzzy Match – allows identification of media that has slight known and acceptable variations from well-recognized media.

MusicID-Stream

MusicID-Stream uses similar waveform fingerprinting technology to that in MusicID-File, but can identify music using short, recorded samples from anywhere within a song. Fingerprints can be generated from a variety of audio sources, including recorded and degraded sources such as radios and televisions. This enables music identification using arbitrary audio sources – including sampling music via mobile devices.

Waveform Recognition

A fingerprint is generated from a short audio sample of about six seconds. The fingerprint is sent to Gracenote Services for identification. Accessing fingerprints requires an Internet connection, which must be provided by the application.

Music-ID Text

Provides text-based recognition for Gracenote albums.

Album Lookup

Text-based Album lookup will query for audio albums matching the given input text. Input fields supported are: album and track artist name, album and track title, and composer.

Multiple Results and Partial Objects

Partial objects allow for disambiguation when more than one album matches the input criteria. This commonly occurs when 1) ambiguous text input terms are used and 2) when queried TOCs match to multiple albums in the Gracenote database – which occurs frequently when discs have a small number of tracks. When this happens, partial objects (partials) are returned to allow the application or user to select the desired album.

For example, if an album title lookup for "Rent" returns many products with "Rent" in the title, partial objects such as artist or year can be returned and referenced by the application or user to select the correct album. The ability to retrieve partials for identical results allows an application to minimize the amount of metadata returned until the correct result is selected. Once the correct album is determined, a full data set can be queried and returned for use by the application.

About MusicID File

MusicID-File provides advanced file-based identification features not included in the MusicID module. MusicID-File can perform recognition using individual files or leverage collections of files to provide advanced recognition. When an application provides decoded audio and text data for each file to the library, MusicID-File identifies each file and groups the files into albums.

The following is a high-level summary of the APIs that fulfill the services for MusicID-File.

- Waveform fingerprinting for files
- Media file data population
- Advanced processing methods
- Result Management

MusicID-File can be used with a local database, but it only performs text-matching locally. Fingerprints are not matched locally.



NOTE: MusicID-File queries never return partial results. They always return full results.

Waveform and Metadata Recognition

The MusicID-File module utilizes both audio data and existing metadata from individual media files to produce the most accurate identification possible.

Decoded audio data is provided to MusicID-File, which processes it to retrieve a unique audio fingerprint. The application can also provide any metadata available for the media file, such as file Tag, file-name, and perhaps any application metadata. MusicID-File can use a combination of fingerprint and text lookups to determine a best-fit match for the given data.

The MusicID module also provides basic file-based media recognition using only audio fingerprints. The MusicID-File module is preferred for file-based media recognition, however, as its advanced recognition process provides significantly more accurate results.

Processing Methods

MusicID-File provides three processing methods enabling advanced file-based media recognition. Each method utilizes the same population and result management APIs, so which method to use is determined by the application's requirement at the time of processing. MusicID-File returns a GDO for each result – providing as many as needed.

TrackID

TrackID provides the simplest processing of media files. With this method, MusicID-File will process each media file independently without regard for other media files that are provided.

This method is best used for small sets of media files. It is also appropriate for retrieving all possible results for a single media file. Consequently, TrackID is useful when getting an answer is more important than getting the best answer.

AlbumID

AlbumID provides an advanced method of media file recognition. Context of the media files, such as their file-system (folder) location and similarity to other media files, can all be leveraged to achieve more accurate media recognition.

This method is best used for groups of media files – where the grouping of the results matters as much as the accuracy of individual results.

LibraryID

LibraryID adds another level of processing beyond AlbumID and is useful for very large collections of media files. Applying the same techniques used in AlbumID, LibraryID further scans and processes files in an entire collection – allowing it to identify groupings otherwise missed by AlbumID.

This method is highly recommended for use when it is necessary to identify a large number of files (hundreds to thousands), but is also effective when used with only a few files. This method takes most of the guesswork out of MusicID-File and lets the SDK do all the work for the application.

About Link

Link allows applications to access and present enriched content related to media that has been identified using GNSDK identification features. Link delivers third-party content identifiers matched to the identified media, which can then be used to retrieve enriched content from Gracenote. Link can also return any custom information that a customer has linked to Gracenote data – such as their own media identifiers.

Without Link, developers themselves must match and integrate data from various sources to present a targeted, relevant content experience to end-users during music playback. By providing developers with the ability to retrieve enriched content, Gracenote enables its customers to generate new revenue streams while improving the music listening experience.

The following is a high-level summary of the APIs that fulfill the services for Link.

- Set input using result GDO; content retrieved is based on results specified in a GDO returned from previous queries
- Retrieve enriched content; multiple pieces of content may be requested without having to set the GDO each time
- Retrieve third-party content IDs

Available enriched content offered through Gracenote Link includes:

- Music Enrichment
 - Album cover art
 - Artist images
 - Album reviews
 - Artist biographies
- Third-party IDs of preferred partners

Music Enrichment

Link provides access to Gracenote Music Enrichment – a single-source solution for enriched content including cover art, artist images, biographies, and reviews. Link and Music Enrichment allow applications to offer enriched user experiences by providing high quality images and information to complement media.

Gracenote provides a large library of enriched content and is the only provider of fully licensed cover art including a growing selection of international cover art. Music Enrichment cover art and artist images are provided by high quality sources that include all the major record labels.

Third-Party Identifiers and Preferred Partners

Link can also match identified media with third-party identifiers. This allows applications to match media to IDs in stores and other online services – facilitating transactions by helping connect queries directly to commerce.

Gracenote has preferred partnerships with several partners and matches preferred partner content IDs to Gracenote media IDs. Entitled applications can retrieve IDs for preferred partners through Link.

Playlist and MoodGrid

Provides functionality to create media sub-groups from a media collection. To use the MoodGrid feature, your application requires special metadata entitlements.

Mood and Tempo (Sonic Attributes)

Gracenote provides two metadata fields that describe the sonic attributes of an audio track. These fields, mood and tempo, are track-level descriptors that capture the unique characteristics of a specific recording.

Mood is a perceptual descriptor of a piece of music, using emotional terminology that a typical listener might use to describe the audio track.

Tempo is a description of the overall perceived speed or pace of the music. Gracenote mood and tempo descriptor systems include hierarchical categories of increasing granularity, from very broad parent categories to more specific child categories.



NOTE: Tempo metadata is available online-only.

Identifying Music

GNSDK supports identification of both non-streaming and streaming music. Non-streaming music generally refers to music stored as a file or on a CD. Streaming music refers to music that is delivered in real-time as an end user listens. Listening to a song on the radio or playing a song from a media player are both examples of streaming music.

GNSDK allows you to identify non-streaming music using a CD TOC, information extracted from an audio file, or Gracenote identifiers.

GNSDK allows you to identify streaming music by generating fingerprints from a streaming audio source, which are then used to identify the music.

Identifying Music Using a CD TOC

The only information that is guaranteed to be on every standard audio CD is a Table of Contents, or TOC. This is a header at the beginning of the disc giving the precise starting location of each track on the CD so that CD players can locate the tracks and compute the track length information for their display panels.

This information is given in frames, where each frame is 1/75 of a second. Because this number is so precise, it is relatively unlikely that two unrelated CDs would have the same TOC. This lets Gracenote use the TOC as a relatively unique identifier.

The example below shows a typical TOC for a CD containing 13 tracks:

```
150 26670 52757 74145 95335 117690 144300 163992 188662 209375 231320  
253150 281555 337792
```

The first number, 150, indicates the starting location of the first track, in frames, from the beginning of the disc (150 frames = 2 seconds). Each of the remaining 13 numbers is the length, in frames, of the corresponding track on the CD. The last number is the offset of the lead out, which marks the end of the CD program area. The first number is greater for extended CDs (CDs with an audio session and one or more metadata sessions). Extended CDs usually have an initial offset of 182 or greater.

Multiple TOC Matches

An album will often have numerous matching TOCs in the Gracenote database. This is because of CD manufacturing differences. More popular discs tend to have more TOCs. Gracenote maintains a catalog of multiple TOCs for many CDs, providing more reliable matching.

For more information on handling multiple matches, See "Gracenote Data Objects (GDOs)" on page 16.

The following is an example of multiple TOCs for a single CD album. This particular album has 22 popular TOCs and many other less popular TOCs.

```
150 26670 52757 74145 95335 117690 144300 163992 188662 209375 231320  
253150 281555 337642
```

```
150 26670 52757 74145 95335 117690 144300 163992 188662 209375 231320
253150 281555 337792
```

```
182 26702 52790 74177 95367 117722 144332 164035 188695 209407 231362
253182 281587 337675
```

```
150 26524 52466 73860 94904 117037 143501 162982 187496 208138 230023
251697 279880 335850
```

Example: Identifying an Album Using a CD TOC

The example below illustrates a simple TOC lookup for local and online systems. The code for the local and online lookups is the same, except for two areas. If you are performing a local lookup, you must initialize the SQLite and Local Lookup libraries, in addition to the other GNSDK libraries:

```
gnsdk_storage_sqlite_initialize(sdkmgr_handle);
gnsdk_lookup_local_initialize(sdkmgr_handle);
```

You must also set the query option to do a local lookup:

```
gnsdk_muscid_query_option_set(
    query_handle,
    GNSDK_MUSCID_OPTION_USE_LOOKUP_LOCAL,
    "true"
);
```

Other than these additions, setting up queries and processing results works in the same way for both local and online lookups.

Sample Application: [muscid_lookup_album_toc/main.c](#)

Identifying Music Using Text

You can identify music by using a lookup based on text strings. The text strings can be extracted from an audio track's file path name and from text data embedded within the file, such as mp3 tags. You can provide the following types of input strings:

- Album title
- Track title
- Album artist
- Track artist
- Track composer

Text-based lookup attempts to match these attributes with known albums, artists, and composers. The text lookup first tries to get an album match. If that is not possible, it next tries to get an artist match. If

that does not succeed, a composer match is tried. Adding as many input strings as possible to the lookup improves the results.

Text-based lookup returns “best-fit” objects, which means that depending on what your input text matched, you might get back album matches or contributor matches.

Creating and Executing a Query for a Text-based Lookup

The first step in performing a text-based lookup is to create a music query, which returns a query handle that you will use in subsequent calls:

```
gnsdk_muscid_query_create( user_handle, GNSDK_NULL, GNSDK_NULL, &query_handle );
```

The next step in creating the query is to set the input text fields, based on the information you have available. The possible input fields are:

- GNSDK_MUSCID_FIELD_ALBUM (album title)
- GNSDK_MUSCID_FIELD_TITLE (track title)
- GNSDK_MUSCID_FIELD_ALBUM_ARTIST
- GNSDK_MUSCID_FIELD_ARTIST (track artist, if different from the album artist)
- GNSDK_MUSCID_FIELD_COMPOSER (track composer; only supported for classical music)

Call the `gnsdk_muscid_query_set_text()` function to set each input field. For example, the following call sets the album title “Dark Side of the Moon” to be used as an input field:

```
gnsdk_muscid_query_set_text( query_handle, GNSDK_MUSCID_FIELD_ALBUM, "Dark Side of the Moon" );
```

Finally, to execute the query, call:

```
gnsdk_muscid_query_find_matches( query_handle, &response_gdo );
```

Processing Text-based Lookup Results

After executing the text-based query, you need to determine which “best-fit” objects were returned. To do this, iterate through the objects, and get the match GDO and then the GDO type, using the following functions:

```
gnsdk_manager_gdo_child_get( response_gdo, GNSDK_GDO_CHILD_MATCH, ord, &match_gdo );
```

```
gnsdk_manager_gdo_get_type( match_gdo, &gdo_type );
```

The GDO type will be one of the following types:

- GNSDK_GDO_TYPE_ALBUM
- GNSDK_GDO_TYPE_CONTRIBUTOR

Compare the GDO type to these types, as shown in the following example:


```
if (0 == strcmp(gdo_type, GNSDK_GDO_TYPE_ALBUM))
{
    printf( "Album match\n" );
    // Display album information.
}
else if (0 == strcmp(gdo_type, GNSDK_GDO_TYPE_CONTRIBUTOR))
{
    printf( "Contributor match\n" );
    // Display contributor information.
}
```



NOTE: If an album GDO is returned and you need to make a follow-up query, use `gnsdk_musicid_query_find_albums()`. If the result is a contributor GDO, no follow-up query is needed.

For more information about navigating through the GDO type hierarchy, see Gracenote Data Objects (GDOs)".

Example: Text Lookup for a Track

This examples performs a sample text query with album, track and artist inputs.

Sample Application: [musicid_lookup_matches_text/main.c](https://github.com/gracenote/gnsdk/blob/master/examples/musicid_lookup_matches_text/main.c)

Identifying Music Using Fingerprints

You can use MusicID or MusicID-File to identify music using an audio fingerprint. An Audio fingerprint is data that uniquely identifies an audio track. The Gracenote Media Service uses audio fingerprints to match the audio from a client application to the Gracenote Music Database.

There are two basic types of fingerprints:

- MusicID-File
- MusicID-Stream

MusicID-File recognizes audio files such as those generated when ripping a CD. It uses the first portion of the audio file for recognition.

MusicID-Stream can recognize a snippet of audio that can be taken from anywhere within the track making it suitable for identifying streamed audio (such as an internet stream) or ambient audio (such as that recorded from a microphone).



Your application can retain fingerprints for a collection of audio files so they can be used later in queries. For example, your application can fingerprint an entire collection of files in a background thread and reuse them later.

PCM Audio Format Recommendations

The following PCM Audio Formats are supported by GNSDK fingerprinting:

Sample Sizes:

- 16-bit

Channels:

- 2 or 1 (stereo or mono)

Sample Rates:

- 48000 Hz
- 44100 Hz
- 32000 Hz
- 24000 Hz
- 22050 Hz
- 16000 Hz
- 11025 Hz

Applications should use the highest quality audio possible to ensure the best results. Lower quality audio will result in less accurate fingerprint matches. Gracenote recommends at least 16-bit, stereo, 22050 Hz.



Do not resample or downsample audio to target these frequencies. Send the best quality audio that you have available.

Example: Identifying Music from an Audio Stream

Sample Application: [musicid_stream/main.c](#)

Advanced Music Identification and Organization

The MusicID-File module provides APIs that enable advanced music identification and organization. These APIs are grouped into three general categories: TrackID, AlbumID, and LibraryID.

TrackID

TrackID processing provides the simplest processing of media files. With this method, MusicID-File processes each media file independently, without regard for any other provided media files. This method is best used for small sets of media identification, where getting an answer is more important than getting the best answer. It is also appropriate to use for accessing all possible results for a single media file. The `gnsdk_musicidfile_query_do_trackid()` API provides TrackID processing.

Example: Implementing TrackID

Sample Application: [musicid_file_trackid/main.c](#)

AlbumID

AlbumID processing provides an advanced method of media file identification. The media file's folder location and its similarity to other media files are used to achieve more accurate identification. This method is best used for groups of media files where the grouping of the results matters as much as accessing accurate, individual results. The `gnsdk_musicidfile_query_do_albumid()` API provides AlbumID processing.

Example: Implementing AlbumID

Sample Application: [musicid_file_albumid/main.c](#)

LibraryID

LibraryID processing adds another level of processing above AlbumID for very large collections of media files. LibraryID extends AlbumID functionality by performing additional scanning and processing of all the files in an entire collection. This enables LibraryID to find groupings that are not captured by AlbumID processing. This method is highly recommended for use when there are a large number (hundreds to thousands) of files to identify, though it is also equally effective when processing only a few files. This method takes most of the guesswork out of MusicID-File and lets the module do all the work for the application. The `gnsdk_musicidfile_query_do_libraryid()` API provides LibraryID processing.

Response GDOs from LibraryID queries are only available via Status and Result callbacks. This is because all LibraryID GDOs are freed after the process finishes.

Example: Implementing LibraryID

Sample Application: [musicid_file_libraryid/main.c](#)

Accessing Music Metadata

This topic describes how to access metadata from music GDOs.

Using Music GDOs

Top-level music GDOs generally represent Albums and Tracks. Album and Track queries can return GDOs containing one or more child GDOs. For example, because a track may exist on multiple albums, the top-level GDO returned from a Track query may contain multiple albums results. Your application should select the child GDO representing a specific album. For more information on managing multiple results, See "Gracenote Data Objects (GDOs)" on page 16.



A music query can return either a full or partial set of metadata. Generally, most queries return partial results containing just enough information to identify the matches. Using



this information, the user can further refine the query, and access full results. For more information, see

The following code assumes that an initial query has already been performed and has returned a result in the variable `result_gdo`. A GDO's child objects are enumerated with an ordinal index, starting from 1 (not 0) for the first child object. The code accesses the GDO representing the first album from the query result in variable `album_gdo`.

```
gnsdk_gdo_handle_t album_gdo = GNSDK_NULL;
gnsdk_manager_gdo_child_get(
    result_gdo,
    GNSDK_GDO_CHILD_ALBUM,
    1,
    &album_gdo
);
```

With the album accessed, the application can now access the GDO for an individual track:

```
gnsdk_gdo_handle_t track_gdo = GNSDK_NULL;
gnsdk_manager_gdo_child_get(
    album_gdo,
    GNSDK_GDO_CHILD_TRACK,
    1,
    &track_gdo
);
```

The ordinal of a track's GDO within the album GDO has no relationship to its track number within the album itself, as not all tracks may be represented within an album. The following code demonstrates access of the actual track number.

```
gnsdk_cstr_t value = GNSDK_NULL;
gnsdk_manager_gdo_value_get(
    track_matched_gdo,
    GNSDK_GDO_VALUE_TRACK_NUMBER,
    1,
    &value
);
```

Code Snippet: [musicid_album_gdo.c](#)

Accessing Mood and Tempo Metadata

In GNSDK, access sonic attribute metadata from Gracenote Service by setting the `GNSDK_MUSICID_OPTION_ENABLE_SONIC_DATA` option to True when performing MusicID Album queries. Setting this option causes the mood and tempo metadata contained in a GDO to be automatically rendered as XML output:

An application must be entitled to implement sonic attribute metadata. Contact your Gracenote Professional Service representative for more information.

Mood

GNSDK provides up to two levels of granularity for mood metadata: Level 1 and Level 2.

Use the GNSDK_GDO_VALUE_MOOD_* value keys to access available mood information from an audio track GDO, as shown below in the accessor code sample:

- GNSDK_GDO_VALUE_MOOD_LEVEL1: Value key to access the Level 1 mood classification, such as Blue.
- GNSDK_GDO_VALUE_MOOD_LEVEL2: Value key to access the Level 2 mood classification, such as Gritty/Earthy/Soulful.

For example:

```
gnsdk_manager_gdo_value_get(track_gdo, GNSDK_GDO_VALUE_MOOD_LEVEL1, 1,
&mood_meta);
gnsdk_manager_gdo_value_get(track_gdo, GNSDK_GDO_VALUE_MOOD_LEVEL2, 1,
&mood);
printf("TRACK %s MOOD:\t\t\t%s (%s)\n", trknum_str, mood_meta, mood);
```

Tempo



NOTE: Tempo metadata is available online-only.

GNSDK provides up to three levels of granularity for tempo metadata; in order of increasing granularity, they are meta, micro, and sub.

Use the following GNSDK_GDO_VALUE_TEMPO_* value keys to access available tempo information from an audio track GDO, as shown below in the accessor code sample:

- GNSDK_GDO_VALUE_TEMPO_LEVEL1: Value key to access the Level 1 tempo classification, such as Fast Tempo.
- GNSDK_GDO_VALUE_TEMPO_LEVEL2: Value key to access the Level 2 tempo classification, such as Very Fast.
- GNSDK_GDO_VALUE_TEMPO_LEVEL3: Value key to access the Level 3 tempo classification, which may be displayed as a numeric tempo range, such as 240-249, or a descriptive phrase.

For example:

```
gnsdk_manager_gdo_value_get(track_gdo, GNSDK_GDO_VALUE_TEMPO_LEVEL1, 1,
&tempo_meta);
gnsdk_manager_gdo_value_get(track_gdo, GNSDK_GDO_VALUE_TEMPO_LEVEL2, 1,
&tempo);
gnsdk_manager_gdo_value_get(track_gdo, GNSDK_GDO_VALUE_TEMPO_LEVEL3, 1,
&tempo_micro);
```

```
printf("TRACK %s TEMPO:\t\t\t%s (%s/%s/%s)\n", trknum_str, value, tempo_
meta, tempo, tempo_micro);
```

Accessing Enriched Music Metadata with the Link Module

All Gracenote customers can access core metadata from Gracenote Services for the products they license. Optionally, customers can access additional metadata, known as enriched metadata by purchasing additional metadata entitlements. Applications generally access enriched metadata using the Link module APIs.

The general process to access linked music metadata is:

1. Pass in a source GDO.
2. Pass in any necessary options (such as a track ordinal number, or an image size).
3. Access the desired content, using the `gnsdk_link_query_content_access()` API.
4. When an application has finished using the content, free the content through the `gnsdk_link_query_content_free()` API.

The following table lists the specific function and option used to access Link metadata in a particular GNSDK module.

GNSDK module	Link Metadata accessed for	Function	Option
MusicID	Albums and Tracks	<code>gnsdk_musicid_query_option_set()</code>	<code>GNSDK_MUSICID_OPTION_ENABLE_LINK_DATA</code>
MusicID-File	Albums and Tracks	<code>gnsdk_musicidfile_query_option_set()</code>	<code>GNSDK_MUSICIDFILE_OPTION_ENABLE</code>

Accessing Classical Music Metadata

This topic discusses the Gracenote Classical Music Initiative (CMI) Three-Line-Solution (TLS), and GNSDK's support for accessing classical music metadata from Gracenote Service. Gracenote TLS provides the four basic classical music metadata components—composer, album name, artist name, and track name—in a standard three-field media display for albums, artists, and tracks, as follows:

- **Composer:** The composer(s) that are featured on an album are listed by their last name in the album title (where applicable). In the examples noted below, the composer is Beethoven.
- **Album Name:** In most cases, a classical album's title is comprised of the composer(s) and work(s) that are featured on the product, which yields a single entity in the album name display. However, for albums that have formal titles (unlike composers and works), the title is listed as it is on the product.
- **General title example:** Beethoven: Violin Concerto
- **Formal title example:** The Best Of Baroque Music

- **Artist Name:** A consistent format is used for listing a recording artist(s)—by soloist(s), conductor, and ensemble(s)—which yields a single entity in the artist name display. For example: Hilary Hahn; David Zinman: Baltimore Symphony Orchestra
- **Track Name:** A consistent format is used for listing a track title—composer, work title, and (where applicable) movement title—which yields a single entity in the track name display. For example: Beethoven: Violin Concerto In D, Op. 61 – 1. Allegro Ma Non Troppo

Accessing Additional Classical Music Metadata

To access additional classical music metadata (when available), set a query handle with the applicable query option set to True to access the metadata and automatically render it to the XML output.

The following table lists the applicable query functions and options for the GNSDK music modules.

Product	Query Function	Query Option
MusicID	gnsdk_musicid_query_option_set gnsdk_musicid_query_option_get	GNSDK_MUSICID_OPTION_ENABLE_CLASSICAL_DATA
MusicID-File	gnsdk_musicidfile_query_option_get	GNSDK_MUSICIDFILE_OPTION_ENABLE_CLASSICAL_DATA

Additional Results in the XML Output

The additional classical music metadata may enhance or replace existing metadata in the XML output of a classical music query result.

GNSDK_GDO_VALUE_CLASSICAL_DATA contains a boolean value to indicate whether classical music metadata exists for the query record.

For the following fields, existing metadata may be updated with classical music metadata, at the specified level(s):

- GNSDK_GDO_VALUE_ARTIST_DISPLAY: At the Album level, lists the Composer name; at the Track level, lists the Artist Name.
- GNSDK_GDO_VALUE_ARTIST_TYPE: At the Composer credit level.
- GNSDK_GDO_VALUE_COMPOSITION_FORM: At the Track level.
- GNSDK_GDO_VALUE_ERA: At the Album and Track credit level, and Composer credit level.
- GNSDK_GDO_VALUE_INSTRUMENTATION: At the Track level.
- GNSDK_GDO_VALUE_GENRE: At the Track level.
- GNSDK_GDO_VALUE_ORIGIN: At the Album, Track, and Composer credit level.
- GNSDK_GDO_VALUE_TITLE_DISPLAY: At the Album level, lists the Album Name; at the Track level, lists the Track Name.
- GNSDK_GDO_VALUE_TITLE_TLS: At the Track level; and the metadata may be different that the metadata that displays in the GNSDK_GDO_VALUE_TITLE_DISPLAY field.

Accessing Album Cover Art

Many GNSDK music applications retrieve Album cover art and display it to the end user. Album cover art is part of the enriched metadata set available through the Link module.



If cover art is unavailable for an album, you can use the default cover art symbol instead. See "Using a Default Image When Cover Art Is Missing" on page 38.

Example: Accessing Album Cover Art

This example shows how to access album cover art.

Sample Application: [link_coverart/main.c](#)

Application Steps:

1. Initialize GNSDK Manager.
2. Initialize Link.
3. Create a Link query.
4. Pass in an album GDO.
5. Set the desired size.
6. Set content type to be cover art.
7. Access the cover art.
8. Free the cover art.

Implementing Playlist Features

Working with Playlists

Playlist provides advanced playlist generation enabling a variety of intuitive music navigation methods. Using Playlist, applications can create sets of related media from larger collections – enabling valuable features such as More Like This™ and custom playlists – that help users easily find the music they want. Playlist functionality can be applied to both local and online user collections. Playlist is designed for both performance and flexibility – utilizing lightweight data and extensible features.

Playlist builds on the advanced recognition technologies and rich metadata provided by Gracenote through GNSDK to generate highly relevant playlists.

The following is a high-level summary of the APIs that fulfill the services for Playlist.

- Create, populate, and manage collection summaries
- Store collection summaries within a local storage solution.
- Generate More Like This playlists
- Generate, validate, and execute custom literal or seed-based playlists
- Manage playlist results

Collection Summaries

Collection summaries store attribute data to support all media in a candidate set and are the basis for playlist generation. Collection summaries are designed for minimal memory utilization and rapid consumption, making them easily applicable to local and server/cloud-based application environments.

Playlists are generated using the attributes stored in the active collection summary. Collection summaries must, therefore, be refreshed whenever media in the candidate set or attribute implementations are modified. Playlist supports multiple collection summaries, enabling both single and multi-user applications.

More Like This

More Like This is a powerful and popular navigation feature made possible by Gracenote and GNSDK Playlist. Using More Like This, applications can automatically create a playlist of music that is similar to given seed music. More Like This is commonly applied to an application's currently playing track to provide users with a quick and intuitive means of navigating through their music collection.

Gracenote recommends using More Like This to quickly implement a powerful music navigation solution. Functionality is provided via a dedicated API to further simplify integration. If you need to create custom playlists, you can use the Playlist Definition Language. See "Playlist PDL Specification" on page 94 for more information.

Playlist Requirements and Recommendations

This topic discusses requirements and recommendations for your Playlist implementation.

Simplified Playlist Implementation

Gracenote recommends streamlining your implementation by using the provided More Like This function, `gnsdk_playlist_generate_morelikethis()`. It uses the More Like This algorithm to generate optimal playlist results and eliminates the need to create and validate Playlist Definition Language statements.

Playlist Content Requirements

Implementing Playlist has these general requirements:

- The application integrates with GNSDK's MusicID or MusicID-File (or both) to recognize music media and create valid GDOs.
- The application uses valid unique identifiers. A unique identifier must be a valid UTF-8 string of unique media identifier data. See "Unique Identifiers" on page 76 for more information.



Unique identifiers are essential to the Playlist generation process. The functionality cannot reference a media item if its identifier is missing or invalid.

Playlist Storage Recommendations

GNSDK provides the SQLite module for applications that may need a storage solution for collections. You can dynamically create a collection and release it when you are finished with it. If you choose this solution, you must store the GDOs or recognize the music at the time of creating the collection.

Your application can also store the collection using the serialization and deserialization functions.

Playlist Resource Requirements

The following table lists resource requirements for Playlist's two implementation scenarios:

Use Case	Typical Scenario	Number of Collection Summaries	Application Provides Collection Summary to Playlist	Required Computing Resources
Single user	Desktop user Mobile device user	Generally only one	Once, normally at start-up	Minimal-to-average, especially as data is ingested only once or infrequently

Use Case	Typical Scenario	Number of Collection Summaries	Application Provides Col-lection Summary to Playlist	Required Com-puting Resources
Multiple users	Playlist server Playlist - in-the-cloud system	Multiple; requires a unique collection summary for each user who can access the system	Dynamically and multiple times; typically loaded with the playlist criteria at the moment before playlist generation	Requires more computing resources to ensure an optimal user experience

Playlist Level Equivalency for Hierarchical Attributes

Gracenote maintains certain attribute descriptors, such as Genre, Era, Mood, and Tempo, in multi-level hierarchies. See "Mood and Tempo (Sonic Attributes)" on page 61 for a description of the hierarchies. As such, Playlist performs certain behaviors when evaluating tracks using hierarchical attribute criteria.

Track attributes are typically evaluated at their equivalent hierarchy list-level. For example, Rock is a Level 1 genre. When evaluating candidate tracks for a similar genre, Playlist analyzes a track's Level 1 genre.

However, Seeds contain the most granular-level attribute. When using a SEED, Playlist analyzes tracks at the respective equivalent level as is contained in the Seed, either Level 2 or Level 3.

Key Components

Playlist operates on several key components. The GNSDK Playlist module provides functions to implement and manage the following key components within your application.

Media metadata: Metadata of the media items (or files)

The media may be on MP3s on a device, or a virtual collection hosted on a server. Each media item must have a unique identifier, which is application-defined.

Playlist requires recognition results from GNSDK for operation, and consequently must be implemented with one or both of GNSDK's music identification modules, MusicID and MusicID-File.

Attributes: Characteristics of a media item, such as Mood or Tempo

Attributes are Gracenote-delivered string data that an application can display; for example, the Mood attribute Soulful Blues.

When doing recognition queries, if the results will be used with Playlist, set the 'enable playlist' query option to ensure proper data is retrieved for the result (GNSDK_MUSICID_OPTION_ENABLE_PLAYLIST or GNSDK_MUSICIDFILE_OPTION_ENABLE_PLAYLIST).

Unique Identifiers

When adding media to a collection summary, an application provides a unique identifier and a GDO, which contains the metadata for the identifier. The identifier is a value that allows the application to identify the physical media being referenced. The identifier is not interpreted by Playlist; it is only returned to the application in Playlist results. An identifier is generally application-dependent. For example, a desktop application typically uses a full path to a file name for an identifier, while an online application typically uses a database key. The media GDO should contain relevant metadata for the media referenced by the identifier. In most cases the GDO comes from a recognition event for the respective media (such as from MusicID). Playlist will take whatever metadata is relevant for playlist generation from the given GDO. For best results, Gracenote recommends giving Album Type GDOs that have matched tracks to this function; Track Type GDOs also work well. Other GDOs are supported, but most other types lack information for good Playlist generation.

Collection Summary

A collection summary contains the distilled information GNSDK Playlist uses to generate playlists for a set of media.

Collection summaries must be created and populated before Playlist can use them for playlist generation. Populating collection summaries involves passing a GDO from another GNSDK identification event (for example, from MusicID) along with a unique identifier to Playlist. Collection Summaries can be stored so they do not need to be reconstructed before every use.

Storage

The application can store and manage collection summaries in local storage.

Seed

A seed is the GDO of a media item in the collection summary used as input criteria for playlist generation. A seed is used to generate More Like This results, or in custom PDL statements. For example, the seed could be the GDO of a particular track that you'd like to use to generate More Like This results.

Playlist generation

GNSDK provides two Playlist generation functions: a general function for generating any playlist, `gnsdk_playlist_generate_playlist()`, and a specific function for generating a playlist that uses the Gracenote More Like This algorithm, `gnsdk_playlist_generate_morelikethis()`.



NOTE: Gracenote recommends streamlining your Playlist implementation by using the provided More Like This™ function, `gnsdk_playlist_generate_morelikethis()`, which uses the More Like This algorithm to generate optimal playlist results and eliminates the need to create and validate PDL statements.

Generating a Playlist

Generating a Playlist involves the following general steps.

1. Create a collection summary.
2. Populate the collection summary with unique identifiers and GDOs.
3. (Optional) Store the collection summary.
4. (Optional) Load the stored collection summary into memory in preparation for Playlist results generation.
5. Generate a playlist, using the More Like This function with a seed.
6. Access and display playlist results.

To generate a custom playlist, use the Playlist Definition Language. See "Playlist PDL Specification" on page 94 for more information.

Creating a Collection Summary

To create a collection summary, call the following function:

```
gnsdk_playlist_collection_create("sample_collection", &playlist_col-  
lection);
```

This creates a new, empty collection summary. The function returns a pointer to a collection handle, which can be used in subsequent calls.



NOTE: Each new collection summary that is created must have a unique name. Although it is possible to create more than one collection summary with the same name, if these collection summaries are then saved to local storage, one collection will override the other. To avoid this, ensure that collection summary names are unique.

Populating a Collection Summary

The main task in building a Playlist collection summary is to provide all possible data to Playlist for each specific media item. The API to provide data for a collection summary is `gnsdk_playlist_collection_add_gdo()`. This API takes a media identifier (any unique string determined by the application) and a GDO to acquire data from. The GDO should come from a recognition event from the other GNSDK modules (such as MusicID or MusicID-File).

Enabling Playlist Attributes

When creating a MusicID or MusicID-File query to populate a playlist, you must set the following query options to ensure that the appropriate Playlist attributes are returned (depending on the type of query):

- GNSDK_MUSICID_OPTION_ENABLE_SONIC_DATA or GNSDK_MUSICIDFILE_OPTION_ENABLE_SONIC_DATA
- GNSDK_MUSICID_OPTION_ENABLE_PLAYLIST or GNSDK_MUSICIDFILE_OPTION_ENABLE_PLAYLIST

Adding Data to a Collection Summary

After retrieving GDOs with a query, use the `gnsdk_playlist_collection_add_gdo()` function to add the GDOs (and unique identifiers) to the collection summary. For example, you might call the function with an album or track GDO (or both), depending on your use case:

```
error = gnsdk_playlist_collection_add_gdo(h_playlist_collection, unique_
ident, album_gdo);
```

```
error = gnsdk_playlist_collection_add_gdo(h_playlist_collection, unique_
ident, track_gdo);
```



NOTE: You can add multiple GDOs for the same identifier by calling the `gnsdk_playlist_collection_add_gdo()` API multiple times with the same identifier value. The data gathered from all the provided GDOs is stored for the given identifier.



When adding an album GDO to Playlist that resulted from a CD TOC lookup, Gracenote recommends adding all tracks from the album to Playlist individually. To do this, call `gnsdk_playlist_collection_add_gdo()` twice for each track on the album. The first call should pass the album GDO for the identifier (to allow Playlist to gather album data), and the second call should pass the respective track GDO from the album using the same identifier used in the first call. This will ensure Playlist adds full album and track data to the collection summary for each track on the album.

How Playlist Gathers Data

When given an album GDO, Playlist gathers any necessary album data and then traverses to the matched track and gathers the necessary data from that track. This data is stored for the given identifier. If no matched track is part of the album GDO, no track data is gathered. Playlist also gathers data from both the album and track contributors as detailed below.

When given a track GDO, Playlist gathers any necessary data from the track, but it is not able to gather any album-related data (such as album title). Playlist also gathers data from the track contributors as detailed below.

When given a contributor GDO (or traversing into one from an album or track), Playlist gathers the necessary data from the contributor. If the contributor is a collaboration, data from both child contributors is gathered as well.

For all other GDOs, Playlist attempts to gather the necessary data, but no other specific traversals are done automatically.

Adding List Elements to Collection Summaries

You can add genre and other list data (origin, era, and tempo) to playlists, by matching by string and adding the result to a collection summary. To search for list elements by string, use the `gnsdk_manager_list_get_element_by_string()` function. This API returns a `gnsdk_list_element_handle_t`, which can be passed to the `gnsdk_playlist_collection_add_list_element()` function to add the value to a collection summary.

Working with Local Storage

You can store and manage collection summaries in local storage. To store a collection summary, use the following function:

```
gnsdk_playlist_storage_store_collection(playlist_collection);
```

To retrieve a collection summary from local storage, use the `gnsdk_playlist_storage_count_collections()` and `gnsdk_playlist_storage_enum_collections()` functions to retrieve all collections.

Before Playlist can use a collection summary that has been retrieved from storage, it must be loaded into memory:

```
error = gnsdk_playlist_storage_load_collection(collection_name, &playlist_collection);
```

Generating a Playlist Using More Like This

You can streamline your Playlist implementation by using the `gnsdk_playlist_generate_morelikethis()` function, which use the "More Like This" algorithm to generate optimal playlist results and eliminates the need to create and validate Playlist Definition Language statements.

You can set the following options when generating a More Like This Playlist, by using the `gnsdk_playlist_morelikethis_option_set()` function:

Option	Description
GNSDK_PLAYLIST_MORELIKETHIS_OPTION_MAX_TRACKS	The maximum number of results returned.
GNSDK_PLAYLIST_MORELIKETHIS_OPTION_MAX_PER_ARTIST	The maximum number of results per artist returned.

Option	Description
GNSDK_PLAYLIST_MORELIKETHIS_OPTION_MAX_PER_ALBUM	The maximum number of results per album returned; the value for this key must evaluate to a number greater than 0.
GNSDK_PLAYLIST_MORELIKETHIS_OPTION_RANDOM	The randomization seed value used in calculating a More Like This playlist. The value for this key must evaluate to a number greater than 0 (the recommended range is 1 - 100, but you can use any number). This number will be the seed for the randomization. You can re-create a playlist by choosing the same number for the randomization seed; choosing different numbers will create different playlists.

The following example demonstrates setting the More Like This options:

```
/* Change the possible result set to contain a maximum of 30 tracks */
gnsdk_playlist_morelikethis_option_set(
    h_playlist_collection,
    GNSDK_PLAYLIST_MORELIKETHIS_OPTION_MAX_TRACKS,
    "30"
);

/* Change the possible result set to contain a maximum of 10 tracks per artist */
gnsdk_playlist_morelikethis_option_set(
    h_playlist_collection,
    GNSDK_PLAYLIST_MORELIKETHIS_OPTION_MAX_PER_ARTIST,
    "0"
);

/* Change the possible result set to contain a maximum of 5 tracks per album */
gnsdk_playlist_morelikethis_option_set(
    h_playlist_collection,
    GNSDK_PLAYLIST_MORELIKETHIS_OPTION_MAX_PER_ALBUM,
    "5"
);

/* Change the random result to be 1, so that there is no randomization */
gnsdk_playlist_morelikethis_option_set(
    h_playlist_collection,
    GNSDK_PLAYLIST_MORELIKETHIS_OPTION_RANDOM,
    "1"
);
```

To generate a More Like This playlist, call the `gnsdk_playlist_generate_morelikethis()` function with a user handle, a playlist collection summary, and a seed GDO. A seed GDO can be any recognized media GDO, for example, the GDO of the track that is currently playing.


```
error = gnsdk_playlist_generate_morelikethis(  
    user_handle,  
    h_playlist_collection,  
    h_gdo_seed,  
    &h_playlist_results  
);
```

Accessing Playlist Results

Once you have generated a playlist, you can iterate through the results using the `gnsdk_playlist_results_count()` and `gnsdk_playlist_results_enum()` functions. The results consist of a set of unique identifiers, which match the identifiers given to Playlist during the population of the collection summary. The application must match these identifiers to the physical media that they reference.

Joining Collection Summaries

The functions that generate playlists take a single collection summary handle as input. If you wish to create a playlist based on multiple collection summaries, you must join the collections, and pass the handle of the joined collection to the generation function. For example, if your application has created a playlist based on one device (collection 1), and another device is plugged into the system (collection 2), you might want to create playlists based on both of these devices. This can be accomplished using the `gnsdk_playlist_collection_join()` function:

```
gnsdk_playlist_collection_join(collection_handle1, collection_handle2);
```

Calling this function joins collection 2 into collection 1. The collection 1 handle now represents the joined collection, and collection 1 now contains a reference to collection 2. You can pass the collection 1 handle to the playlist generation functions, and the joined collection will be used to generate the playlist.



If you are working with multiple collection summaries, Gracenote recommends creating an empty parent collection summary, and then joining all additional collection summaries into the parent. This way, all collection summaries will be children of the parent, rather than one collection being the child (or parent) of another.

Joins are created in memory only, and cannot be saved in local storage. If you try to save a joined collection in local storage, only the parent collection will be stored. Collections that have been joined into the parent will not be stored.

To remove a collection from a join, call the `gnsdk_playlist_collection_join_remove()` function.

Synchronizing Collection Summaries

Collection summaries must be refreshed whenever items in the user's media collection are modified. For example, if you've created a collection summary based on the media on a particular device, and the media on that device changes, your application must synchronize the collection summary.

Synchronization of a collection summary to physical media involves two steps:

1. Adding all existing (current and new) unique identifiers, using the `gnsdk_playlist_collection_sync_ident_add()` function, which Playlist collates.
2. Calling `gnsdk_playlist_collection_sync_process()` to process the current and new identifiers and using the callback function to add or remove identifiers to or from the collection summary.

Iterating the Physical Media

The first step in synchronizing is to iterate through the physical media, calling the `gnsdk_playlist_collection_sync_ident_add()` function for each media item. For each media item, pass the unique identifier associated with the item to the function. The unique identifiers used must match the identifiers that were used to create the collection summary initially.

Processing the Collection

After preparing a collection summary for synchronization using `gnsdk_playlist_collection_sync_ident_add()`, call `gnsdk_playlist_collection_sync_process()` to synchronize the collection summary's data. During processing, the callback function `gnsdk_playlist_update_callback_fn()` will be called for each difference between the physical media and the collection summary. So the callback function will be called once for each new media item, and once for each media item that has been deleted from the collection. The callback function should add new and delete old identifiers from the collection summary.

Example: Implementing a Playlist

This example demonstrates using the Playlist APIs to create More Like This and custom playlists.

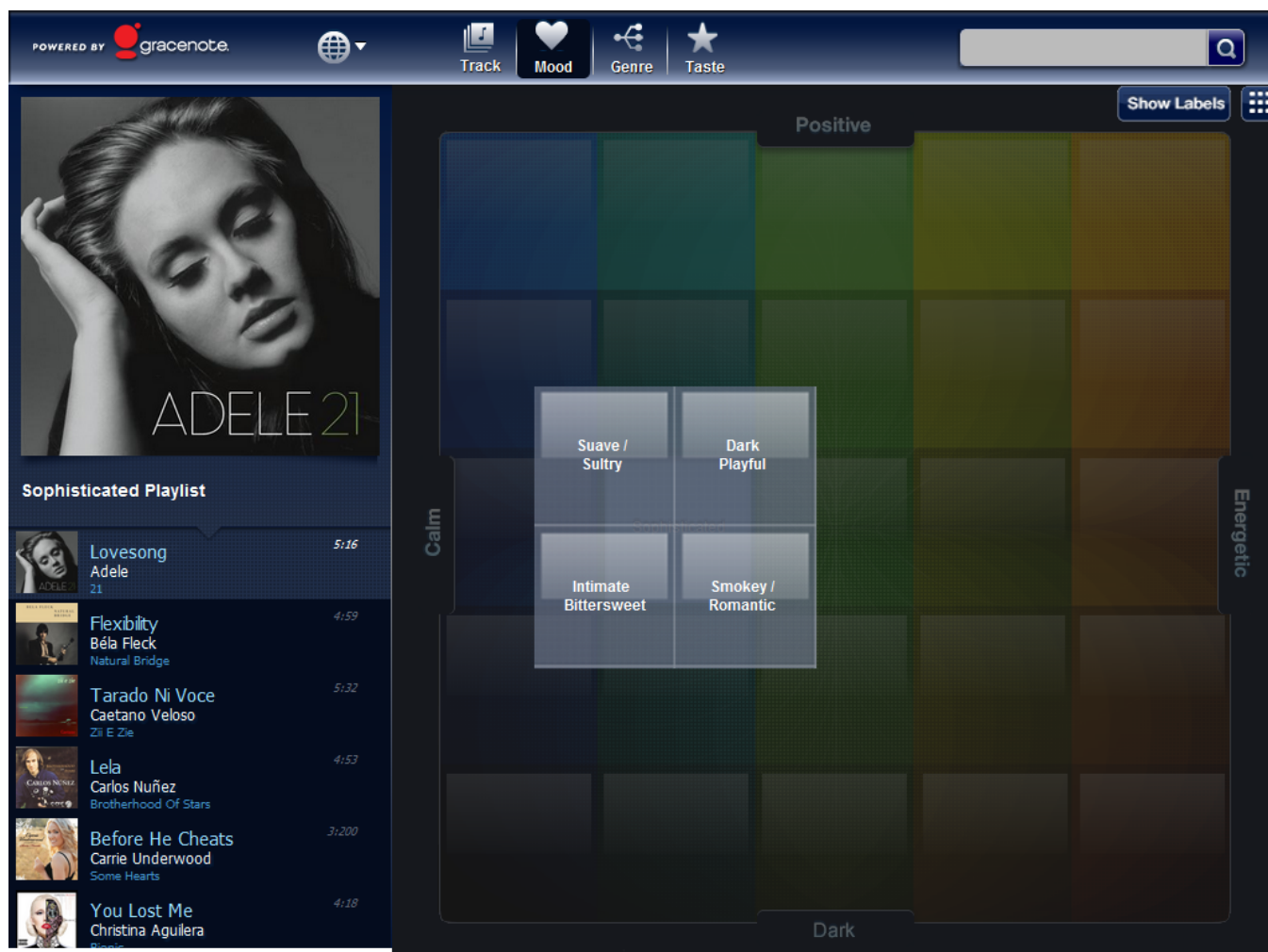
Sample Application: [playlist/main.c](#)

Working with MoodGrid

The MoodGrid library allows applications to generate playlists and user interfaces based on Gracenote Mood descriptors. MoodGrid provides Mood descriptors to the application in a two-dimensional grid that represents varying degrees of moods across each axis. One axis represents energy (calm to energetic) and the other axis represents valence (dark to positive). When the user selects a mood from the grid, the application can provide a playlist of music that corresponds to the selected mood. Additional filtering support is provided for genre, origin, and era music attributes.

Mood Descriptors are delivered with track metadata from queries to MusicID, MusicID-File, or MusicID-Stream services.

The MoodGrid data representation is a 5x5 or 10x10 grid. Each element of the grid corresponds to a Gracenote Mood category ID (Level 1 for 5x5 and Level 2 for 10x10), which can be used to create a list of playable songs in the user's collection characterized by that mood. Each Level 1 Mood maps to four, more granular, Level 2 Moods. For example, a Level 1 Mood might be "Romantic", and the corresponding Level 2 Moods might be "Suave/Sultry", "Dark/Playful", "Intimate/Bittersweet", and "Smoky/Romantic", providing a greater level of detail within the "Romantic" Mood.



Note: The pictured implementation of MoodGrid is centered upon the collection and arrangement of track Mood descriptors across the dimensions of energy (calm to energetic) and valence (dark to positive). The actual layout and navigation of these moods is entirely dependent upon the implementation of the application's user interface. Gracenote's MoodGrid does not necessarily need to be presented in the square representation demonstrated above.

The MoodGrid APIs:

- Encapsulate Gracenote's Mood Editorial Content (mood layout and ids).
- Simplify access to MoodGrid results through x,y coordinates.
- Allow for multiple local and online data sources through MoodGrid Providers.
- Enable pre-filtering of results using genre, origin, and era attributes.
- Support 5x5 or 10x10 MoodGrids.
- Provide the ability to go from a cell of a 5x5 MoodGrid to any of its expanded four Moods in a 10x10 grid.

Implementing MoodGrid

Implementing MoodGrid in an application involves the following steps:

1. Initializing the MoodGrid module.
2. Enumerating the data sources using MoodGrid Providers.
3. Creating and populating a MoodGrid Presentation.
4. Filtering the results, if needed.

Initializing the MoodGrid Module

Before using the MoodGrid APIs, follow the usual steps to initialize GNSDK. The following GNSDK modules must be initialized:

- GNSDK Manager
- SQLite (for local caching)
- MusicID
- Playlist
- MoodGrid

For more information on initializing the GNSDK modules, see “Initializing an Application.”

To initialize MoodGrid, use the `gnsdk_moodgrid_initialize()` function.

```
error = gnsdk_moodgrid_initialize(sdkmgr_handle);
```



If you are using MusicID to recognize music, you must enable Playlist and DSP data in your query. You must be entitled to use Playlist—if you are not, you won't get an error, but MoodGrid will return no results.

Enumerating Data Sources using MoodGrid Providers

GNSDK automatically registers all local and online data sources available to MoodGrid. For example, if you create a playlist collection using the Playlist API, GNSDK automatically registers that playlist as a data source available to MoodGrid. These data sources are referred to as Providers. MoodGrid is designed to work with multiple providers. You can iterate through the list of available Providers using the `gnsdk_moodgrid_provider_count()` and `gnsdk_moodgrid_provider_enum()` functions. For example, the following call returns a handle to the first Provider on the list (at index 0):

```
gnsdk_moodgrid_provider_enum(0, &provider);
```

You can use the handle to the Provider to retrieve the following information, by calling the `gnsdk_moodgrid_provider_get_data()` function:

- Name (GNSDK_MOODGRID_PROVIDER_NAME key)
- Type (GNSDK_MOODGRID_PROVIDER_TYPE key)
- Network Use (GNSDK_MOODGRID_PROVIDER_NETWORK_USE key)

Creating and Populating a MoodGrid Presentation

Once you have a handle to a Provider, you can create and populate a MoodGrid Presentation with Mood data. A Presentation is a data structure that represents the MoodGrid, containing the mood

name and playlist information associated with each cell of the grid.

To create a MoodGrid Presentation, use the `gnsdk_moodgrid_presentation_create()` function, passing in the user handle, the type of the MoodGrid, and the provider handle. The type can be one of the enumerated values in `gnsdk_moodgrid_presentation_type_t`: either a 5x5 or 10x10 grid. The function returns a handle to the Presentation:

```
gnsdk_moodgrid_presentation_create(user, type, NULL, NULL, &presentation);
```

Each cell of the Presentation is populated with a mood name and associated playlist. You can iterate through the Presentation to retrieve this information from each cell. The following pseudo-code demonstrates this procedure:

```
gnsdk_moodgrid_presentation_type_dimension(type, &max_x, &max_y);
for (every grid cell [x,y] from [1,1] to [max_x, max_y])
{
    gnsdk_moodgrid_presentation_get_mood_name(presentation, x, y, &name);
    gnsdk_moodgrid_presentation_find_recommendations(presentation, provider, x, y, &results);
    gnsdk_moodgrid_results_count(results, &count);
    for ( every item i in results )
        gnsdk_moodgrid_results_enum(results, i, &ident);
}
```

Calling the `gnsdk_moodgrid_presentation_type_dimension()` function obtains the dimensions of the grid, which allows you to iterate through each cell. For each cell you can get the mood name by calling the `gnsdk_moodgrid_presentation_get_mood_name()` function with the coordinates of the cell. The locale of the mood name is based on the playlist group locale, and if that is not defined, it is based on the music group locale. You can also get the list of recommended tracks (playlist) for that particular mood by calling the `gnsdk_moodgrid_presentation_find_recommendations()` function with the same coordinates. Finally, you can get the identifier for each track in the playlist, by iterating through the results and calling the `gnsdk_moodgrid_results_enum()` function.

Filtering MoodGrid Results

If you wish, you can filter the MoodGrid results. MoodGrid provides filtering by genre, origin, and era. If you apply a filter, the results that are returned are pre-filtered, reducing the amount of data transmitted. For example, the following call sets a filter to limit results to tracks that fall within the Rock genre.

```
gnsdk_moodgrid_presentation_filter_set(presentation, "FILTER", GNSDK_MOODGRID_FILTER_LIST_TYPE_GENRE, GNSDK_LIST_MUSIC_GENRE_ROCK, GNSDK_MOODGRID_FILTER_CONDITION_INCLUDE);
```

Shutting Down MoodGrid

When you are finished using MoodGrid, you can use the `gnsdk_moodgrid_shutdown()` function to shut it down. The GNSDK modules necessary for running MoodGrid should be shut down in the following order:

1. MoodGrid
2. Playlist
3. MusicID
4. SQLite
5. GNSDK Manager

Example: Working with MoodGrid

This example provides a complete MoodGrid sample application.

Sample Application: [moodgrid/main.c](#)

Advanced Topics

This section covers topics for advanced GNSDK development.

Working with Non-GDO Identifiers

In addition to GDOs, Gracenote supports other media element identifiers, as shown in the following table. All of these represent TUI/Tag pairs.

Identifier	Equivalent to...	Description
TUI/Tag	N/A	A TUI (Title Unique Identifier) is the core identifier for Gracenote. This identifier has a direct relationship to specific metadata within Gracenote. A TUI Tag is a special hash that must accompany the TUI whenever the TUI is used. The TUI Tag ensures the TUI was received from Gracenote and was not created by a non-Gracenote program.
GNID	TUI-Tag	
GNUID	Shortened representation of a TUI/Tag pair	
TAGID and ProductID	DATA<TUI>FOO<TAG>DATA, where DATA is random data	TagID and ProductID are the same. TagID was the original name for this identifier. These IDs are created from TUI and TUI Tag pairs. They are commonly included as tag metadata within MP3 files so the files can be quickly identified by Gracenote. These IDs are static.

Examples of Non-GDO Identifiers

The following table shows examples of non-GDO identifiers.

Identifier	Example
TUI/Tag	12345567/ABCDEF1234567890ABCDEF1234567890
GNID	12345567-ABCDEF1234567890ABCDEF1234567890
GNUID	ABCD1234567890
TAGID and ProductID	DATA12345567FOOABCDEF1234567890ABCDEF1234567890DATA

GNSDK Identifiers Comparison Matrix

The following table lists the strengths and weaknesses of the different Gracenote identifiers.

Identifier	Pros	Cons
Serialized GDO	<p>Good for getting back to the original metadata.</p> <p>Good for joining responses and requests between different Gracenote products</p> <p>Does not require knowledge of the object type</p>	<p>Cannot be used for comparison to other serialized GDO values</p> <p>Not useful for integrating GNSDK to other metadata sources, as only</p> <p>GNSDK supports GDOs</p> <p>Relatively large and requires much storage space</p>
TUI /TUI Tag Pair	<p>Good for comparison</p> <p>Good for integrating GNSDK with other Gracenote metadata sources</p>	<p>Cannot be used for getting back to the original metadata, due to the lack of a Tag ID to perform the query</p> <p>Cannot be efficiently used for joining different Gracenote products, as it does not contain enough metadata.</p> <p>Requires knowledge of the object type</p> <p>Requires an accompanying tag, as it is not a single input</p>
TagID (ProductID)	<p>Good for comparison</p> <p>Good for getting back to original metadata.</p> <p>Concise and does not require large amounts of storage space</p> <p>Single input and does not require an accompanying tag</p>	<p>Cannot be used among different Gracenote products as a Product ID/Tag ID</p> <p>contains only enough information for certain SDKs</p> <p>Cannot be efficiently used for integrating the GNSDK to other metadata sources, as Product IDs/Tag IDs are not generally supported (however, they can be deconstructed to a TUI)</p> <p>Requires knowledge of the object type</p>

Converting Non-GDO Identifiers to GDOs

This section shows examples that create GDO identifiers from Non-GDO identifiers.

Example: Accessing Locally-Stored Album using TUI

Sample Application: [musicid_lookup_album_local_online/main.c](https://github.com/Gracenote/musicid_lookup_album_local_online/main.c)

Example: Creating a GDO from an Album TUI

```
gnsdk_gdo_handle_t query_gdo;
gnsdk_cstr_t tui = "12345678";
gnsdk_cstr_t tui_tag = "ABCDEF1234567890ABCDEF1234567890";
error = gnsdk_manager_gdo_create_from_id(tui, tui_tag, GNSDK_ID_SOURCE_
ALBUM, &query_gdo);
if (!error)
error = gnsdk_musicid_query_set_gdo(query_gdo);
```

Example: Creating a GDO from a Track TagID

```
gnsdk_gdo_handle_t query_gdo;
gnsdk_cstr_t tagid =
"3CD3N43R15145217U37894058D7FCB938ADFA97874409F9531B2P3";
error = gnsdk_manager_gdo_create_from_id(tagid, GNSDK_NULL, GNSDK_ID_
SOURCE_TRACK, &query_gdo);
if (!error)
error = gnsdk_musicid_query_set_gdo(query_gdo);
```

Example: Creating a GDO from a CDDBID

```
gnsdk_gdo_handle_t query_gdo;
gnsdk_cstr_t cddbaid = "4+029C475EFCF8D469C78A644DEBFABF91+5795407";
error = gnsdk_manager_gdo_create_from_id(cddbaid, GNSDK_NULL, GNSDK_ID_
SOURCE_CDDBID, &query_gdo);
if (!error)
error = gnsdk_musicid_query_set_gdo(query_gdo);
```

Example: Creating a GDO from Raw CDDBID

A Raw CDDBID is a decomposition of the CDDBID above.

```
gnsdk_gdo_handle_t query_gdo;
gnsdk_cstr_t mui = "5795407";
gnsdk_cstr_t mediaid = "029C475EFCF8D469C78A644DEBFABF91";
error = gnsdk_manager_gdo_create_from_id(mui, mediaid, GNSDK_ID_SOURCE_
CDDBID, &query_gdo);
if (!error)
error = gnsdk_musicid_query_set_gdo(query_gdo);
```

Improving Matches Using Both CD TOC and Fingerprints

To help improve the accuracy of the results returned from Gracenote Service, you can use both a TOC and a fingerprint in a query. The use of a fingerprint for additional identification criteria can increase the number of single- and multi-exact matches and decreasing the number of fuzzy match results. This query method is especially useful for Albums that contain few (four or less) Tracks, such as CD singles. With this method:

- You do not need to set the TOC and fingerprint functions in a specific order on the query handle. Setting the TOC first and the fingerprint second, or vice versa, will work.
- The fingerprint can be of either metadata type: CMX or GNFPX.
- You can use existing fingerprint data, or generate fingerprint data. See the examples below.
- When the fingerprint metadata does not aid the match, Gracenote Service disregards it, and returns a result (or results) that match only the input TOC.
- Using this query method generally results in a longer query processing time to identify the additional fingerprint; on average, about 15 seconds per Track.
- You can alternately perform this query method using `gnsdk_musicid_query_add_toc_offset()`.

If experiencing issues when performing cumulative queries, check that the logs are recording both a TOC and a fingerprint as inputs for the specific query.

Using a TOC and an Existing Fingerprint

Use a TOC with an existing fingerprint that is accessed using `gnsdk_musicid_query_set_fp_data()`. In this case, the fingerprint does not have to be the first Track of the Album. A calling sequence example is:

1. `gnsdk_musicid_query_set_toc_string()`
2. `gnsdk_musicid_query_set_fp_data()`
3. `gnsdk_musicid_query_find_albums()`

Using a TOC and a Generated Fingerprint

Use a TOC with fingerprint that is generated using the `gnsdk_musicid_query_fingerprint_*` functions. Calling `gnsdk_musicid_query_fingerprint_end()` is optional, depending on whether the `pb_complete` parameter of `gnsdk_musicid_query_fingerprint_write()` receives enough audio data. Refer to these functions' Remarks for more information. A calling sequence example is:

1. `gnsdk_musicid_query_set_toc_string()`
2. `gnsdk_musicid_query_fingerprint_begin()`
3. `gnsdk_musicid_query_fingerprint_write()`
4. (Optional) `gnsdk_musicid_query_fingerprint_end()`
5. `gnsdk_musicid_query_find_albums()`

Rendering a GDO as XML

To present GDO metadata to a user, or to process its contents for other uses, an application can render it in XML format.

You can do this with the following call:

```
// Render the Album GDO as XML to the "xml" parameter
gnsdk_manager_gdo_render_to_xml(album_gdo, GNSDK_GDO_RENDER_XML_FULLL,
&xml);
```

Example: Rendering a GDO as XML

Code Snippet: [gdo_xml_render.c](#)

Application Steps:

1. Perform MusicID query and get album GDO.
2. Render album GDO as XML.
3. Display rendered GDO XML.

Using Lists

GNSDK uses list structures to store strings and other information that do not directly appear in results returned from the Gracenote Service. Lists generally contain information such as localized strings and region-specific information. Each list is contained in a corresponding *List Type*.

Lists are either *flat* or *hierarchical*. Flat lists contain only one level of metadata, such as languages. Hierarchical lists are tree-like structures with parents and children.

Typically, hierarchical lists contain general display strings at the upper levels (Level 1) and more granular strings at the lower levels (Level 2 and Level 3, respectively). For example, a parent Level 1 music genre of Rock contains a grandchild Level 3 genre of Rock Opera. The application can determine what level of list granularity is needed by using the list functions (`gnsdk_sdkmgr_list_*`) in the GNSDK Manager. For more information, See "Core and Enriched Metadata" on page 4.

Lists can be specific to a *Region*, as well as a *Language*. For example, the music genre known as J-pop (Japanese pop) in America is called pop in Japan.

In general, Lists provide:

- Mappings from Gracenote IDs to Gracenote Descriptors, in various languages
- Delivery of content that powers features such as MoreLikeThis, Playlist, and MoodGrid



MoreLikeThis, Playlist, and MoodGrid also use *Correlates*. These lists specify the correlations among different genres, moods, and so on. For example, Punk correlates higher to Rock than it does to Country. Using the MoreLikeThis feature when playing a Punk track will likely return more Rock suggestions than Country.

List and Locale Interdependence

GNSDK for Auto provides *Locales* as a way to group lists specific to a region and language. Using locales is relatively straightforward for most applications to implement. However, it is not as flexible as directly

accessing lists - most of the processing is done in the background and is not configurable. For more information, See "Locales" on page 32.

Conversely, the List functionality is robust, flexible, and useful in complex applications. However, it requires more implementation design and maintenance. It also does not support non-list-based metadata and the GDO value keys (GNSDK_GDO_VALUE_*).

For most applications, using locales is more than sufficient. In cases where the application requires non-locale functionality (for example, list display), implementing both methods may be necessary. The following sections discuss each method's advantages and disadvantages.

The GNSDK locale functionality includes:

- Loading a locale into the application using `gnsdk_manager_gdo_load_locale()`. GNSDK loads a locale behind the scenes, and loads only those lists that are not already in memory. When database (such as SQLite) cache is enabled, the GNSDK automatically caches the list. When caching is not enabled, GNSDK downloads the locale for each request.
- You can set a loaded locale as the application's default locale, and this can eliminate the need to call `gnsdk_manager_set_locale()` for a GDO. Instead, you call the default locale.
- Setting a locale for a GDO using `gnsdk_manager_gdo_set_locale()`. Doing this ensures that all language-specific metadata is returned with the language (and region and descriptor, if applicable) defined for the locale.

When using locales, be aware that:

- You can serialize locales and lists and save them within your application for later re-use. To store non-serialized locales and lists, you must implement a database cache. If you do not want to use a cache, you can instead download required locales and lists during your application's initialization.
- A list's contents cannot be displayed (for example, to present a list of available genres to a user).
- Performing advanced list functionality, such as displaying list items in a dropdown box for a user's selection, or accessing list elements for filtering lookups (for example, restricting lookups to a particular mood or tempo), is not possible.

The GNSDK list functionality includes:

- Directly accessing individual lists using the `gnsdk_manager_list_*` APIs.
- Accessing locale-specific list metadata.

When using lists, be aware that:

- There are numerous list handles.
- List handles must be released after use.
- Locale-specific non-list metadata is not supported.

- List metadata GDO keys are not supported.
- When using `gnsdk_manager_list_retrieve()` to get a list, you must provide a user handle.

Accessing Mood and Tempo Metadata Using Lists

Applications that have GNSDK list functionality implemented must use `GNSDK_LIST_TYPE_MOODS` and `GNSDK_LIST_TYPE_TEMPOS` with the GNSDK Manager list functions (`gnsdk_manager_list_*`) to access mood and tempo values.

Example: Accessing a Music Genre List

This example demonstrates accessing a list and displaying the list's elements.

Code Snippet: [list.c](#)

Application Steps:

1. Initialize GNSDK Manager and get User handle.
2. Get music genre list with `gnsdk_manager_list_retrieve()`.
3. Make SDK list calls (`gnsdk_manager_list_*`) to get list type, language, region, descriptor and # of levels and display.
4. Make SDK list calls for each level element and display list ID and name.
5. Release the list.
6. Release resources and shutdown SDK.

Best Practices for Audio Stream Recognition

The following practices are recommended when recognizing audio from the microphone:

- Provide clear and concise onscreen instructions on how to record the audio:
 - Most failed recognitions are due to incorrect operation by the user. Provide clear and concise instructions to help the user correctly operate the application to result in a higher match rate and a better user experience.
- While recording audio from the microphone display a progress indicator:
 - When recording from the microphone, the application can receive status updates. The status updates indicate what percentage of the recording is completed. Use this information to display a progress bar (indicator) to notify the user. When recording has finished use vibration or a tone (or both) to notify the user.
- Visual only notifications can hamper the user experience because:
 - The user may not see notifications if they are holding the recording device up to an audio source. Also, the user may pull the device away from an audio source to check if recording has completed. This may result in a poor quality recording.

In general, your application should display an animation to indicate the application is performing an operation. If the application appears to stop, the user may believe the application has crashed.
- If no match is found, redisplay the user instructions and ask the user to try again.

Playlist PDL Specification

The GNSDK Playlist Definition Language (PDL) is a query syntax, similar to Structured Query Language (SQL), that enables flexible custom playlist generation using human-readable text strings. PDL allows developers to dynamically create custom playlists. By storing individual PDL statements, applications can create and manage multiple preset and user playlists for later use.

PDL statements express the playlist definitions an application uses to determine what items are included in resulting playlists. PDL supports logic operators, operator precedence and in-line arithmetic. PDL is based on Search Query Language (SQL). This section assumes you understand SQL and can write SQL statements.



NOTE: Before implementing PDL statement functionality for your application, carefully consider if the provided More Like This function, `gnsdk_playlist_generate_morelikethis()` meets your design requirements. Using the More Like This function eliminates the need to create and validate PDL statements.

PDL Syntax

This topic discusses PDL keywords, operators, literals, attributes, and functions.

Note: Not all keywords support all operators. Use `gnsdk_playlist_statement_validate()` to check a PDL Statement, which generates an error for invalid operator usage.

Keywords

PDL supports these keywords:

Keyword	Description	Required or Optional	PDL Statement Order
GENERATE PLAYLIST	All PDL statements must begin with either GENERATE PLAYLIST or its abbreviation, GENPL	Required	1
WHERE	Specifies the attributes and threshold criteria used to generate the playlist. If a PDL statement does not include the WHERE keyword, Playlist operates on the entire collection.	Optional	2

Keyword	Description	Required or Optional	PDL Statement Order
ORDER	<p>Specifies the criteria used to order the results' display.</p> <p>If a PDL statement does not include the ORDER keyword, Playlist returns results in random order.</p> <p>Example: Display results in based on a calculated similarity value; tracks having greater similarity values to input criteria display higher in the results.</p> <p>The expression format is: <identifier> <operator> <identifier></p>	Optional	3
LIMIT	<p>Specifies criteria used to restrict the number of returned results.</p> <p>Also uses the keywords RESULT and PER.</p> <p>Example: Limiting the number of tracks displayed in a playlist to 30 results with a maximum of two tracks per artist.</p> <p>The expression format is: <identifier> <operator> <identifier></p>	Optional	4
SEED	<p>Specifies input data criteria from one or more ids. Typically, a Seed is the This in a More Like This playlist request.</p> <p>Example: Using a Seed of Norah Jones' track Don't Know Why to generate a playlist of female artists of a similar genre.</p> <p>The expression format is: <identifier> <operator> <identifier></p>	Optional	NA

Example: Keyword Syntax

This PDL statement example shows the syntax for each keyword. In addition to <att_imp>, <expr>, and <score> discussed above, this example shows:

- <math_op> is one of the valid PDL mathematical operators.
- <number> is positive value.
- <attr_name> is a valid attribute, either a Gracenote-delivered attribute or an implementation-specific attribute.

```

GENERATE PLAYLIST
WHERE <att_imp> [<math_op> <score>][ AND|OR <att_imp>]
ORDER <expr>[ <math_op> <expr>]
LIMIT [number RESULT | PER <attr_name>][,number [ RESULT | PER <attr_
name>]]

```

Operators

PDL supports these operators:

Operator Type	Available Operators
Comparison	>, >=, <, <=, ==, !=, LIKE LIKE is for fuzzy matching, best used with strings; see PDL Examples
Logical	AND, OR
Mathematical	+, -, *, /

Literals

PDL supports the use of single (') and double (") quotes, as shown:

- Single quotes: 'value one'
- Double quotes: "value two"
- Single quotes surrounded by double quotes: "'value three'"



You must enclose a literal in quotes or Playlist evaluates it as an attribute.

Attributes

Most attributes require a Gracenote-defined numeric identifier value or GDO Seed.

- Identifier: Gracenote-defined numeric identifier value is typically a 5-digit value; for example, the genre identifier 24045 for Rock. These identifiers are maintained in lists in Gracenote Service; download the lists using GNSDK Manager's Lists and List Types APIs
- GDO Seed: Use GNSDK Manager's GDO APIs to access XML strings for Seed input.

The following table shows the supported system attributes and their respective required input. The first four attributes are the GOET attributes.



The delivered attributes have the prefix GN_ to denote they are Gracenote standard attributes. You can extend the attribute functionality in your application by implementing custom attributes; however, do not use the prefix GN_.

Name	Attribute	Required Input
Genre Origin Era Artist Type Mood Tempo	GN_Genre GN_Origin GN_Era GN_ArtistType GN_Mood GN_Tempo	Gracenote-defined numeric identifier value GDO Seed XML string
Artist Name	GN_ArtistName	Text string
Album Name	GN_AlbumName	

Functions

PDL supports these functions:

- RAND(max value)
- RAND(max value, seed)

PDL Statements

This topic discusses PDL statements and their components.

Attribute Implementation <att_imp>

A PDL statement is comprised of one or more attribute implementations that contain attributes, operators, and literals. The general statement format is:

```
"GENERATE PLAYLIST WHERE <attribute> <operator> <criteria>"
```

You can write attribute implementations in any order, as shown:

```
GN_ArtistName == "ACDC"
```

or

```
"ACDC" == GN_ArtistName
```

WHERE and ORDER statements can evaluate to a score; for example:

```
"GENERATE PLAYLIST WHERE LIKE SEED > 500"
```

WHERE statements that evaluate to a non-zero score determine what ids are in the playlist results. ORDER statements that evaluate to a non-zero score determine how ids display in the playlist results.

Expression <expr>

An expression performs a mathematical operation on the score evaluated from a attribute implementation.

[<number> <math_op>] <att_imp>

For example:

3 * (GN_Era LIKE SEED)

Score <score>

Scores can range between -1000 and 1000.

For boolean evaluation, True equals 1000 and False equals 0.

Note: For more complex statement scoring, concatenate attribute implementations and add weights to a PDL statement.

Example: PDL Statements

The following PDL example generates results that have a genre similar to and on the same level as the seed input. For example, if the Seed genre is a Level 2: Classic R&B/Soul, the matching results will include similar Level 2 genres, such as Neo-Soul.

```
"GENERATE PLAYLIST WHERE GN_Genre LIKE SEED"
```

This PDL example generates results that span a 20-year period. Matching results will have an era value from the years 1980 to 2000.

```
"GENERATE PLAYLIST WHERE GN_Era >= 1980 AND GN_Era < 2000"
```

This PDL example performs fuzzy matching with Playlist, by using the term LIKE and enclosing a string value in single (') or double (") quotes (or both, if needed). It generates results where the artist name may be a variation of the term ACDC, such as:

- ACDC
- AC/DC
- AC*DC

```
"GENERATE PLAYLIST WHERE (GN_ArtistName LIKE 'ACDC')"
```

The following PDL example generates results where:

- The tempo value must be less than 120 BPM.
- The ordering displays in descending order value, from greatest to least (119, 118, 117, and so on).
- The genre is similar to the Seed input.

```
"GENERATE PLAYLIST WHERE GN_Tempo > 120 ORDER GN_Genre LIKE SEED"
```

Glossary

Term	Definition
Attribute	Characteristic of a media item, such as Genre, Origin, Era, and Artist Type (collectively known as GOET). Other attributes are Mood, Tempo, and standard meta-data like Names, Titles, and so on.
Client ID	<p>Each GNSDK customer receives a unique Client ID string from Gracenote. This string uniquely identifies each application and lets Gracenote deliver the specific features for which the application is licensed.</p> <p>The Client ID string has the following format: 123456-789123456789012312. The first part is a six-digit Client ID, and the second part is a 17-digit Client ID Tag.</p>
Contributor	A Contributor refers to any person who plays a role in an AV Work. Actors, Directors, Producers, Narrators, and Crew are all consider a Contributor. Popular recurring Characters such as Batman, Harry Potter, or Spider-man are also considered Contributors in Video Explore.
Credit	<p>A credit lists the contribution of a person (or occasionally a company, such as a record label) to a recording. Generally, a credit consists of:</p> <ul style="list-style-type: none"> The name of the person or company. The role the person played on the recording (an instrument played, or another role such as composer or producer). The tracks affected by the contribution. A set of optional notes (such as "plays courtesy of Capitol records"). See Role
Editable GDO	A GDO with editable metadata; this is an augmentation of the current GNSDK read-only GDO model. Note that only specific fields are editable to preserve the GDO's metadata integrity with the Gracenote Service.
Features	Term used to encompass a particular media stream's characteristics and attributes; this is metadata and information accessed from processing a media stream. For example, when submitting an Album's Track, this includes information such as fingerprint, mood, and tempo metadata (Attributes).
Filter Keys and Values	Filters are used to limit a result set. They follow the key/value paradigm. For example filter_set(FILTER_KEY_SEASON_NUMBER, "1") ensures that all of the results are for the first season of the show.

Term	Definition
Fingerprint	Fingerprint Types: Cantametrix Philips microFAPI nanoFAPI
Fuzzy Match	Allows identification of media that has slight known and acceptable variations from well-recognized media.
GDO	Gracenote Data Object
Generation Criterion	A selection rule for determining which tracks to add to a playlist.
Generator	Short for playlist generator. This is a term used to indicate that an artist has at least one of the extended metadata descriptors populated. The metadata may or may not be complete, and the artist text may or may not be locked or certified. See also: extended metadata, playlist optimized artist.
Genre	A categorization of a musical composition characterized by a particular style.
Key/Value Pairs	A key is used to access a value – either for getting or setting. Consider a dictionary. The key is the word you are looking up, the value is the definition. In a phone book, the name is the key, the phone number is the value. Like phone books and dictionaries, a single key may point to multiple values. So, keys are used to get or set to some piece of metadata. For GNSDK, Child keys are used to access children. Children are GDOs. Value keys are used to access values. Values are strings.
Link	A GNSDKmodule that allows applications to access and present enriched content related to media that has been identified using GNSDK identification features.
Manual Playlist	A manual playlist is a playlist that the user has created by manually selecting tracks and a play order. An auto-playlist is a playlist generated automatically by software. This distinction only describes the initial creation of the playlist; once created and saved, all that matters to the end-user is whether the playlist is static (and usually editable) or dynamic (and non-editable). All manual playlists are static; the track contents do not change unless the end-user edits the playlist. An auto-playlist may be static or dynamic.
Metadata	Data about data. For example, metadata such as the artist, title, and other information about a piece of digital audio such as a song recording.

Term	Definition
Mood	Track-level perceptual descriptor of a piece of music, using emotional terminology that a typical listener might use to describe the audio track; includes hierarchical categories of increasing granularity. See Sonic Attributes.
More Like This	A mechanism for quickly generating playlists from a user's music collection based on their similarity to a designated seed track, album, or artist.
Multiple Match	During CD recognition, the case where a single TOC from a CD may have more than one exact match in the MDB. This is quite rare, but can happen. For example with CDs that have only one track, it is possible that two of these one-track CDs may have exactly the same length (the exact same number of frames). There is no way to resolve these cases automatically as there is no other information on the CD to distinguish between them. So the user must be presented with a dialog box to allow a choice between the alternatives. See also: frame, GN media database, TOC.
MusicID-File	A feature of the MusicID product that identifies digital music files using a combination of waveform analysis technology, text hints and/or text lookups.
MusicId	Enables MusicID recognition for identifying CDs, digital music files and streaming audio and delivers relevant metadata such as track titles, artist names, album names, and genres. Also provides library organization and direct lookup features.
Options and Option Keys	<p>Option Keys are used to access (or set) option values. For example, use a 'TIMEOUT' key to set the network timeout option: <code>option_set(OPTION_KEY_TIMEOUT, "1000")</code> or <code>option_get(OPTION_KEY_TIMEOUT, &option_value)</code>.</p> <p>Option Values are some predefined values. For example, instead of setting the timeout to 1000 milliseconds like the above, you could use the <code>TIMEOUT_INFINITE</code> option value: <code>option_set(OPTION_KEY_TIMEOUT, OPTION_VALUE_TIMEOUT_INFINITE)</code></p>
Partial Object	Partial (high-level) metadata returned by Gracenote Services to provide a reference for selection between multiple results. Commonly occurs when input text strings are ambiguous.
Playlist	A set of tracks from a user's music collection, generated according to the criteria and limits defined by a playlist generator.

Term	Definition
Popularity	Popularity is a relative value indicating how often metadata for a track, album, artist and so on is accessed when compared to others of the same type. Gracenote's statistical information about the popularity of an album or track, based on aggregate (non-user-specific) lookup history maintained by Gracenote servers. Note that there's a slight difference between track popularity and album popularity statistics. Track popularity information identifies the most popular tracks on an album, based on text lookups. Album popularity identifies the most frequently looked up albums, either locally by the end-user, or globally across all Gracenote users. See Rating and Ranking
Ranking	For Playlist, ranking refers to any criteria used to order songs within a playlist. So a playlist definition (playlist generator) may rank songs in terms of rating values, or may rank in terms of some other field, such as last-played date, bit rate, etc.
Rating	Rating is a value assigned by a user for the songs in his or her collection. See Popularity and Ranking.
Role	A role is the musical instrument a contributor plays on a recording. Roles can also be more general, such as composer, producer, or engineer. Gracenote has a specific list of supported roles, and these are broken into role categories, such as string instruments, brass instruments. See Credit.
Seed Track, Disc, Artist, Genre	Used by playlist definitions to generate a new playlist of songs that are related in some way, or similar, to a certain artist, album, or track. This is a term used to indicate that an artist has at least one of the extended metadata descriptors populated. The metadata may or may not be complete, and the artist text may or may not be locked or certified. See also: extended metadata, playlist optimized artist, playlist-related terms.
Sonic Attributes	The Gracenote Service provides two metadata fields that describe the sonic attributes of an audio track. These fields, mood and tempo, are track-level descriptors that capture the unique characteristics of a specific recording. Mood is a perceptual descriptor of a piece of music, using emotional terminology that a typical listener might use to describe the audio track. Tempo is a description of the overall perceived speed or pace of the music. The Gracenote mood and tempo descriptor systems include hierarchical categories of increasing granularity, from very broad parent categories to more specific child categories.
Target Field	The track attribute used by a generation criterion for selecting tracks to include in a generated playlist.
Tempo	Track-level descriptor of the overall perceived speed or pace of the music; includes hierarchical categories of increasing granularity. See Sonic Attributes.

Term	Definition
TOC	Table of Contents. An area on CDs, DVDs, and Blu-ray discs that describes the unique track layout of the disc.
Type	Type indicates the kind of a GDO. Every GDO has a type, so developers know what metadata it contains (and what keys it supports for metadata access).