AVR32709: AVR32 UC3 Audio Decoder Over USB

Features

- **Software MP3 Decoder**
- **FAT File System**
- **Music played over USB (USB Host mass storage class)**
- **Standalone - Low Memory Footprint (Code & RAM) and no Operating System dependency**
- **Audio output over I2S using SSC controller or internal Audio Bitstream DAC**
- **Local Control with a keypad**

1. Introduction

This application note will help the reader to use the Atmel® AVR® UC3 Audio Decoder over USB software. This software includes a software MP3 decoder, a file system, and a USB Host mass storage class support.

For more information about the AVR UC architecture, please refer to the appropriate documents available from http://www.atmel.com/avr.

2. License

The MP3 decoder MAD is distributed under the terms of the GPL.

The source-code for the UC3-specific parts of the libmad decoder are available from [1] and binaries can be built from [2]. For a complete audio application note, this is available under NDA only.
The JPEG decoder (IJG) license can be found in /SERVICES/PICTURES/JPG/IJG/license.txt.
The memory manager (dlmalloc) license can be found in /SERVICES/MEMORY/MEMORY_MANAGER/DLMALLOC/license.txt

Notes:
1. http://git.buildroot.net/buildroot/tree/package/multimedia/libmad/libmad-0.15.1b-optimization.patch.avr32
2. http://git.buildroot.net/buildroot/tree/package/multimedia/libmad

3. Requirements

The software provided with this application note requires several components:

- A computer running Microsoft® Windows® 2000/XP/Vista or Linux®
- AVR32Studio and the GNU toolchain (GCC) or IAR Embedded Workbench for AVR32 compiler.
- A JTAGICE mkII or AVROne! debugger

32-bit Atmel Microcontroller

Application Note

7817D–AVR32–05/11

4. Theory of Operation

4.1 Overview

Today, embedded MP3 decoders are everywhere for consumers listening to audio content on mobile devices.

4.1.1 MP3

MPEG-1 Audio Layer 3, more commonly referred to as MP3, is a digital audio encoding format using a form of lossy data compression. Several bit rates are specified in the MPEG-1 Layer 3 standard: 32, 40, 48, 56, 64, 80, 96, 112, 128, 160, 192, 224, 256 and 320Kbit/s, and the available sampling frequencies are 32, 44.1 and 48KHz. A sample rate of 44.1KHz is almost always used. 128 Kbit/s bitrate files are slowly being replaced with higher bitrates like 192Kbit/s, with some being encoded up to MP3's maximum of 320Kbit/s.

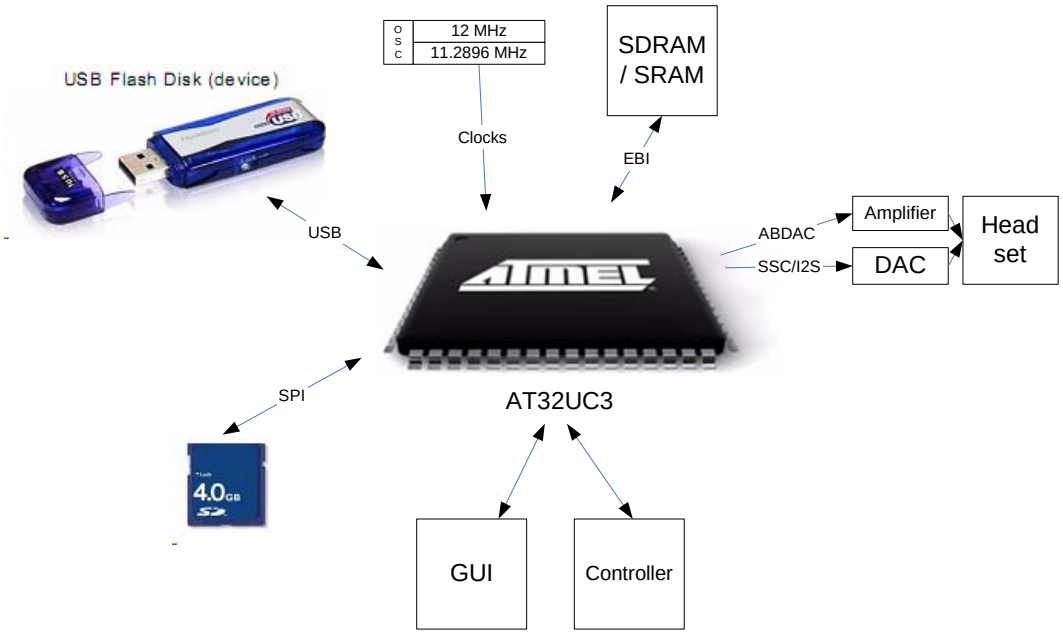
A tag in a compressed audio file is a section of the file that contains metadata such as the title, artist, album, track number or other information about the file's contents.

The chosen MP3 decoder here is MAD (libmad), a high-quality MPEG audio decoder. It currently supports MPEG-1 and the MPEG-2 extension to Lower Sampling Frequencies, as well as the so-called MPEG 2.5 format. All three audio layers (Layer I, Layer II, and Layer III a.k.a. MP3) are fully implemented. MAD does not yet support MPEG-2 multichannel audio (although it should be backward compatible with such streams).

4.2 Block diagram

The following block diagram describes the UC3 interfacing the USB stick and output the audio stream from the key to the external DAC. The user can control the player using a keypad, running a customisable Human-Machine Interface (HMI).

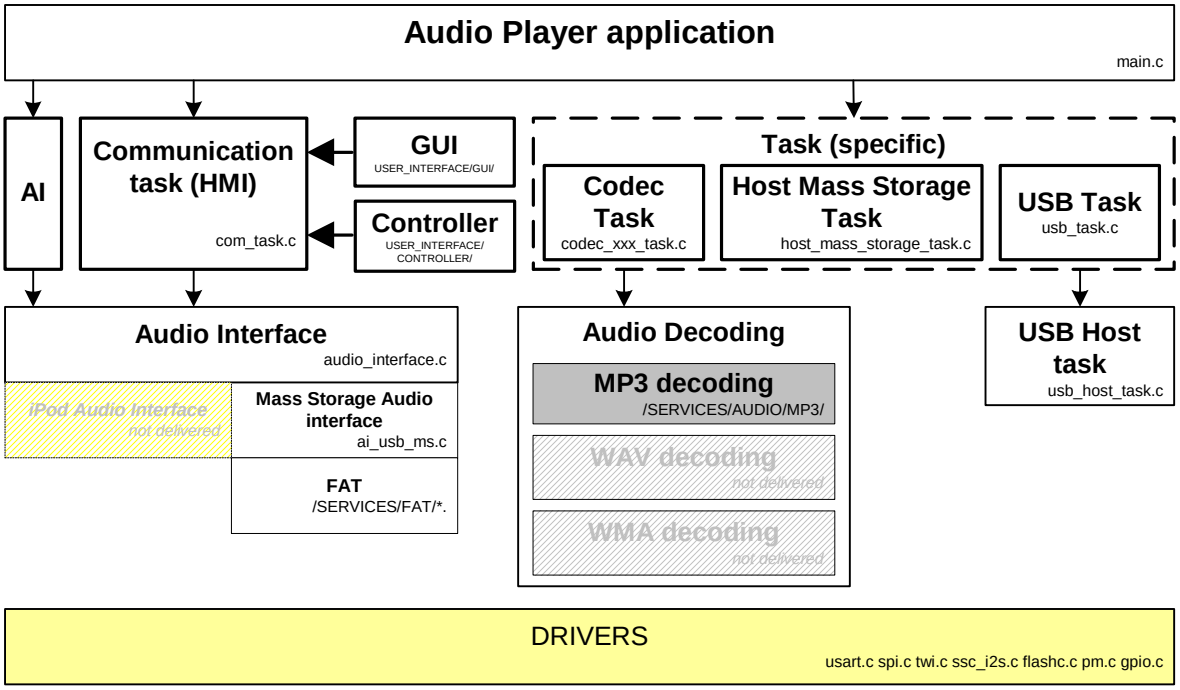
Figure 4-1. Block diagram



4.3 Software architecture

The following figure shows the basic software architecture of the application:

Figure 4-2. Software architecture



The application does not require any operating system to run. The main() function is in charge of calling the software «tasks» (using a scheduler) that make audio decoding, HMI, and USB management possible. There are 5 tasks:

- The communication task (shown in bold on the graph) contains the HMI of the application. **This task holds the real intelligence of the user application and interfaces directly with the Audio Interface. This is the task that the user should modify for his own application.**
- Audio Interface task: This task handles dynamic support for any new plugged device types. For example, if a new device using another class than the mass storage class is plugged, and of course if this class is supported by the application, then, this task will tell the audio interface to link the "Audio Interface" API to a low level API that supports this class. All the mechanism in place to dynamically link an API to this abstract API is located in the file: /SERVICES/AUDIO/AUDIO_PLAYER/audio_interface.[c|h]
  In this application, only the USB mass storage class is implemented, therefore, this function is not used.
- The last task is macro used for feature tasks. In the MP3 audio player over USB, the modules involved will be the MP3 decoder, the USB stack and the host mass storage task:
  - The MP3 codec task: This task is in charge of the Audio Decoding management.
  - The USB task: This task handles the USB stack and events.
  - The Host Mass Storage task: This task will check for new devices connection and initialize them using the USB Mass Storage class.

The main loop of the application is a simple free-running task scheduler:

```
while (TRUE)
{
  ai_task();
  com_task();
  task();
}
```

Note that the Audio Interface can also support iPod® audio decoding through the USB Audio class. More information can be provided to MFi certified customers..

5. Audio Interface API

Note that the audio player is a generic audio player interface and is designed to support multiple audio formats besides MP3.

5.1 Overview

The Audio Interface (AI) manages disk navigation, audio navigation and audio control (see below). Thus, the user does not have to directly interface with the File System and audio control APIs. This greatly simplifies the software architecture.

The Audio Interface can be used in 2 different ways:

- Using "**asynchronous**" functions, which result/effect may not be produced in one single iteration. Using these functions usually leads to the use of state-machines in the user firmware (since one must wait for the completion of a command before launching a new one), and has the advantage of reducing the risks of audio underrun. Asynchronous functions always have the "ai_async" prefix. Note that only one asynchronous function can be used at a time.
- Using "**synchronous**" functions, which are executed immediately. This drastically simplifies the user firmware architecture (no use of state-machines since the synchronous AI functions are immediately executed) but *may* produce audio underrun since the execution time of these functions may be too long. Synchronous functions just have the "ai_" prefix.

All functions of the Audio Interface have a synchronous and asynchronous interface. For example, the command which returns the number of drives is:

- ai_**async**_nav_drive_nb() in the asynchronous interface.
- ai_nav_drive_nb() in the synchronous interface.

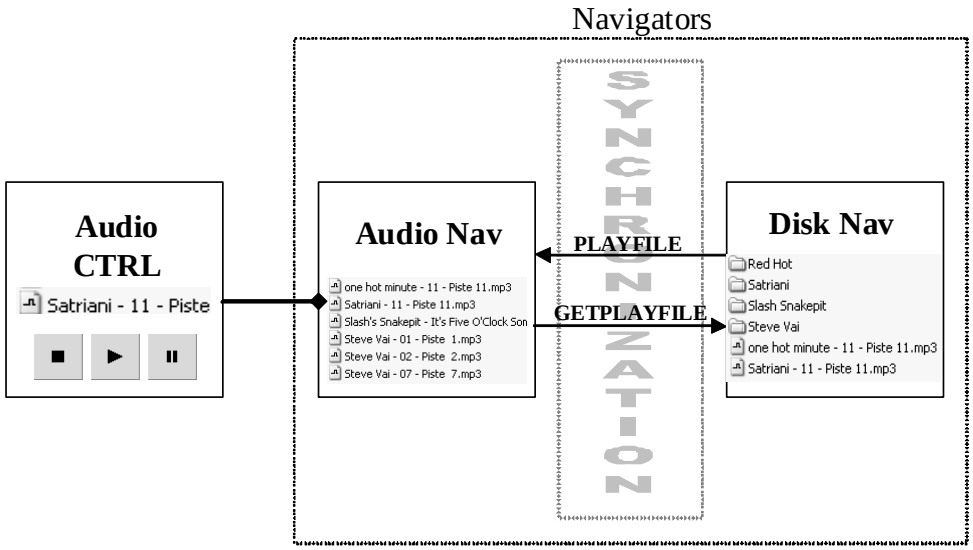
Using asynchronous functions shall be the preferred solution in order to avoid audio underruns.

5.2 Audio Interface Architecture

The AI commands to interface the audio player are divided into three categories:

- **Disk navigation (Disk Nav)** to browse into the tree architectures of the USB device.
- **Audio navigation (Audio Nav)** to manage a list of playable songs.
- **Audio Control (Audio CTRL)** to control the audio stream (play/pause/fast forward/...).

Figure 5-1. Audio Player Navigators Overview



5.2.1 Disk Navigator: “browse into the tree architectures of the mass storage device”

The “Disk Navigator” is similar to a file explorer. It is used to navigate in the tree architectures of the connected USB device. It will hide all files whose extension differs from *.mp3, *.pls, *.smp or *.m3u. Note that the file extension filtering is automatically updated according to the modules selected (MP3, playlists, ...).

The “Disk Navigator” is totally independent from the “Audio Navigator”.

The commands associated with this module are used to navigate into the disks/directories and to synchronize the selected file/folder with the “Audio Nav” module. This synchronization is made with these two commands **ai_nav_getplayfile** and **ai_audio_nav_playfile**. See “Disk Navigation” on page 8. for a complete list of commands.

5.2.2 Audio Navigator: “manage a list of playable songs”

The “Audio Navigator” deals with a list of files. This list is defined once a **ai_audio_nav_playfile** command is executed. This command sets the list of files to be played according to the current selection in the “Disk Navigator“ and to the “Audio Navigator” **ai_audio_nav_expmode_set** option.

When the **ai_audio_nav_playfile** command is executed:

- if the “Disk Navigator” is currently pointing on a playlist (*.m3u), then only the files of this playlist will be included in the “Audio Navigator” file list.
- if the “Disk Navigator” is currently pointing on a disk or a file, then the file list will depend on the audio explorator mode (set by the **ai_audio_nav_expmode_set** command):

Table 5-1. **ai_audio_nav_expmode_set** command behavior

Explorer mode	Behavior
AUDIO_EXPLORER_MODE_DISKS	Builds a file list off all playable songs of all disks, and start playing from the selected file.
AUDIO_EXPLORER_MODE_DISK	Builds a file list off all playable songs of the current disk only, and start playing from the selected file.
AUDIO_EXPLORER_MODE_DIRONLY	Builds a file list off all playable songs that are contained in the current directory, and start playing from the selected file. Sub directories are ignored.
AUDIO_EXPLORER_MODE_DIRSUB	Builds a file list off all playable songs that are contained in the current directory and its sub-directories, and start playing from the selected file.

- if the “Disk Navigator” is currently pointing on a folder, the audio navigator will not enter into it. It will look for the first file that is in the current directory and build its file list according to the previous rules.
- if the “Disk Navigator” is not pointing on any files or folders (which is the case after a mount or a goto), a directory or a playlist is selected, then it will select the first file of the file list.

Note that the playing order can be changed at compilation time by enabling a define. This is fully explained in Section 5.8.3 “Example 3 - Change the playing order” on page 17.

Once the file list is set, two main commands are available to navigate into it:

- **ai_audio_nav_next** (select next file) and
- **ai_audio_nav_previous** (select previous file).

Options can also be defined to the “Audio Navigator” (to change the repeat mode, enable/disable the shuffle mode).

To synchronize back the “Disk Navigator” with the “Audio Navigator”, i.e. make the “Disk Navigator” selecting the file played by the “Audio Navigator”, the command **ai_nav_getplayfile** can be used. See “Audio Navigation” on page 10. for a complete list of commands.

5.2.3 Audio Control: “control the audio stream (play/pause/fast forward/...)”

This module controls the audio stream of the selected file (selected by the “Audio Navigator”). Commands like play/pause/stop/fast forward/fast rewind… are available. See “Audio Control” on page 13. for a complete list of commands.

5.3 Limitations

5.3.1 Speed

Speed navigation (such as file browsing) in directories may be affected if:

- A High-bitrate file is played at the same time.
- Directories have many files.
- The playlist includes many files.

5.4 Disk Navigation

The exploration is based on a selector displacement. The **file list** is the list of the files in the current directory according to the extension filter (.mp3, .m3u, .pls, .smp, ...)

The “file list”:

- is updated when you exit or enter a directory or a disk.
- starts with the directories then the files.
- is sorted in the creation order.

Table 5-2. Disk Navigator Commands

Command Name	Input	Output		Description
		Return	Extra result	
ai_get_product_id		Product ID		Returns the product identifier (USB PID) of the connected device (if available).
ai_get_vendor_id		Vendor ID		Returns the vendor identifier (USB VID) of the connected device (if available).
ai_get_serial_number		Length of the serial number in bytes	Serial number	Returns the serial number of the connected device (if available).
ai_nav_drive_nb		Number of drive		Returns the number of disks available.
ai_nav_drive_set	Drive number	True or false		Selects the disk but does not mount it: (0 for drive 0, 1 for drive 1...). Returns false in case of error.
ai_nav_drive_get		Drive number		Returns the selected disk number.
ai_nav_drive_mount		True or false		Mounts the selected disk. Returns false in case of error.
ai_nav_drive_total_space		Capacity of the drive		Returns the total space, in bytes, available on the device.
ai_nav_drive_free_space		Free space on the drive		Returns the free space, in bytes, available on the device.
ai_nav_dir_root		True or false		Initializes the file list on the root directory. Return false in case of error.
ai_nav_dir_cd		True or false		Enters in the current directory selected in file list. Return false in case of error.
ai_nav_dir_gotoparent		True or false		Exits current directory and goes to parent directory. Then selects the folder from which it just exits, rather than selecting the first file of the parent directory. This simplifies navigation since the user firmware does not have to memorize this information. Returns false in case of error.
ai_nav_file_isdir		True or false		Returns true if the selected file is a directory, otherwise returns false.
ai_nav_file_goto	Position in file list	True or false		Goes to a position in file list (0 for position 0, 1 for position 1...). Returns false in case of error.
ai_nav_file_pos		File position		Returns the file position of the selected file in the current directory. Returns FS_NO_SEL if no file is selected.

Table 5-2. Disk Navigator Commands

Command Name	Input	Output		Description
		Return	Extra result	
ai_nav_file_nb		Number of audio files		Returns the number of audio files in the file list. Audio files are all the files which extensions matches *.mp3 or *.wma (it depends on the file format supported). There is a specific command for playlist files, see below.
ai_nav_dir_nb		Number of directories		Returns the number of directories in the file list.
ai_nav_playlist_nb		Number of playlists		Returns the number of playlists in the file list. The playlists are all the files matching with the extension *.m3u, *.pls and *.smp.
ai_nav_dir_name		Length of the string in bytes	UNICODE name	Returns the name of the current directory.
ai_nav_file_name		Length of the string in bytes	UNICODE name	Returns the name of the selected file.
ai_nav_file_info_type		File type		Returns the type of the audio file selected.
ai_nav_file_info_version		Version number		Returns the version of the metadata storage method used for the selected audio file if available, otherwise, returns 0.
ai_nav_file_info_title		Length of the string in bytes	UNICODE title	Returns the title of the selected audio file if available, otherwise, returns an empty string.
ai_nav_file_info_artist		Length of the string in bytes	UNICODE artist	Returns the artist's name of the selected audio file if available, otherwise, returns an empty string.
ai_nav_file_info_album		Length of the string in bytes	UNICODE album	Returns the album's name of the selected audio file if available, otherwise, returns an empty string.
ai_nav_file_info_year		Year		Returns the year of the selected audio file if available, otherwise, returns 0.
ai_nav_file_info_track		Length of the string in bytes	UNICODE info	Returns the track information of the selected audio file if available, otherwise, returns an empty string.
ai_nav_file_info_genre		Length of the string in bytes	UNICODE genre	Returns the genre of the selected audio file if available, otherwise, returns an empty string.
ai_nav_file_info_duration		Duration of the track		Returns the total time of the selected audio file in milliseconds if available, otherwise, returns 0.
ai_nav_file_info_image	Max image size	Image size	-True of false - A pointer on the image data	Returns information and a pointer on a bitmap of the embedded cover art of the selected audio file. Return false in case of error.
ai_nav_getplayfile		True or false		Selects the file selected by the audio navigator. This command is the only link between these two navigators. Return false in case of error.

5.5 Audio Navigation

This navigator sets the list of files to be played. It can be seen as a «playlist».
Before accessing this navigator, an **ai_audio_nav_playfile** command must be issued.

Table 5-3. Audio Navigator Commands

Command Name	Input	Output		Description
		Return	Extra result	
ai_audio_nav_playfile		True or false		This command sets the audio file list according to the current mode of the audio navigator and plays (or sets to pause) the file selected in the disk navigator. In other words, it synchronizes the audio navigator with the disk navigator and plays (or sets to pause) the selected file from the disk navigator. Note that this command returns immediatly and does not wait until the current track is completly played. This commands does not change the current options (repeat/random/expmode). It is the opposite of the command **ai_nav_getplayfile**. Return false in case of error.
ai_audio_context_save		Structure Context	- True or false - The length of the structure in bytes	Gives complete audio context (player state, play time, repeat, random, file played, explorer mode).
ai_audio_context_restore	Structure Context	True or false		Restores an audio context (eventually restart playing).
ai_audio_nav_next		True or false		Jump to next audio file available in the list. The next song file is chosen according to the current options (repeat/random/mode).
ai_audio_nav_previous		True or false		Jumps to previous audio file available in the list. The next song file is chosen according to the current options (repeat/random/mode).
ai_audio_nav_eof_occur		True or false		This routine must be called once a track ends. It will choose, according to the options (repeat/random/expmode), the next file to play.
ai_audio_nav_nb		Number of audio files present in the list		Returns the number of audio files present in the list.
ai_audio_nav_getpos		File position		Returns the file position of the selected file in the list.
ai_audio_nav_setpos	File position	True or false		Goes to a position in the list.
ai_audio_nav_repeat_get		Ai_repeat_mode		Returns the current repeat mode (no repeat, repeat single, repeat folder, repeat all).
ai_audio_nav_repeat_set	Ai_repeat_mode			Sets the current repeat mode (no repeat, repeat single, repeat folder, repeat all).
ai_audio_nav_shuffle_get		Ai_shuffle_mode		Returns the current shuffle mode (no shuffle, shuffle folder, shuffle all).
ai_audio_nav_shuffle_set	Ai_shuffle_mode			Sets the current shuffle mode (no shuffle, shuffle folder, shuffle all).
ai_audio_nav_expmode_get		Ai_explorer_mode		Returns the current explorer mode (all disks, one disk, directory only, directory + sub directories).

Table 5-3. Audio Navigator Commands

Command Name	Input	Output		Description
		Return	Extra result	
ai_audio_nav_expmode_set	Ai_explorer_ mode			Sets the current explorer mode (all disks, one disk, directory only, directory + sub directories). This mode cannot be changed while an audio file is being played.
ai_audio_nav_getname		Length of the string in bytes	UNICODE name	Returns the name of selected file.
ai_audio_nav_file_info_type		File type		Returns the type of the audio file selected.
ai_audio_nav_file_info_version		Version number		Returns the version of the metadata storage method used for the selected audio file if available, otherwise, returns 0.
ai_audio_nav_file_info_title		Length of the string in bytes	UNICODE title	Returns the title of the selected audio file if available, otherwise, returns an empty string.
ai_audio_nav_file_info_artist		Length of the string in bytes	UNICODE artist	Returns the artist's name of the selected audio file if available, otherwise, returns an empty string.
ai_audio_nav_file_info_album		Length of the string in bytes	UNICODE album	Returns the album's name of the selected audio file if available, otherwise, returns an empty string.
ai_audio_nav_file_info_year		Year		Returns the year of the selected audio file if available, otherwise, returns 0.
ai_audio_nav_file_info_track		Length of the string in bytes	UNICODE info	Returns the track information of the selected audio file if available, otherwise, returns an empty string.
ai_audio_nav_file_info_genre		Length of the string in bytes	UNICODE genre	Returns the genre of the selected audio file if available, otherwise, returns an empty string.
ai_audio_nav_file_info_duration		Duration of the track		Returns the total time of the selected audio file in milliseconds if available, otherwise, returns 0.
ai_audio_nav_file_info_image	Max image size	Image size	-True of false - A pointer on the image data	Returns information and a pointer on a bitmap of the embedded cover art of the selected audio file. Returns false in case of error.

5.5.1 Modes used by the interface

5.5.1.1 Ai_repeat_mode

Defines the repeat modes:

- AUDIO_REPEAT_OFF: no repeat.
- AUDIO_REPEAT_TRACK: repeat the current track.
- AUDIO_REPEAT_FOLDER: repeat the current folder.
- AUDIO_REPEAT_ALL: repeat the list of files.

5.5.1.2 Ai_shuffle_mode

Defines the shuffle/random mode:

- AUDIO_SHUFFLE_OFF: no shuffle.
- AUDIO_SHUFFLE_FOLDER: shuffle into the current folder.
- AUDIO_SHUFFLE_ALL: shuffle into the list of files.

5.5.1.3 *Ai_explorer_mode*

Defines the explorer mode (see Section 5.8.3 “Example 3 - Change the playing order” on page 17 for more information.

5.6 Audio Control

The audio controller is used to control the audio stream of the audio file selected by the audio navigator.

Table 5-4. Audio Control Commands

Command Name	Input	Output		Description
		Return	Extra result	
ai_audio_ctrl_stop		True or false		Stops the audio.
ai_audio_ctrl_resume		True or false		Plays or resumes play after an **ai_audio_ctrl_pause** or an **ai_audio_ctrl_stop** command.
ai_audio_ctrl_pause		True or false		Pauses the audio.
ai_audio_ctrl_time		Elapsed time		Returns the elapsed time of the audio track being played.
ai_audio_ctrl_status		Status		Returns the status of the audio controller (stop, play, pause, a new audio file is being played, the current folder has changed).
ai_audio_ctrl_ffw	Skip time	True or false		Fast forwards the audio until the skip time has been reached. Then, it will continue to play the rest of the audio file. The skip time passed in parameter is in second.
ai_audio_ctrl_frw	Skip time	True or false		Fast rewinds the audio until the skip time has been reached. Then, it will set the audio player in play mode. The skip time passed in parameter is in second.
ai_audio_ctrl_start_ffw		True or false		Sets the audio player into fast forward mode. ***Function not implemented yet.***
ai_audio_ctrl_start_frw		True or false		Sets the audio player into fast rewind mode. ***Function not implemented yet.***
ai_audio_ctrl_stop_ffw_frw		True or false		Stops fast forwarding/rewinding and set the audio player into the previous mode (play or pause). ***Function not implemented yet.***

5.7 Using Asynchronous or Synchronous API

Using synchronous function is straightforward. Once finished, synchronous functions returns, with or without a result.

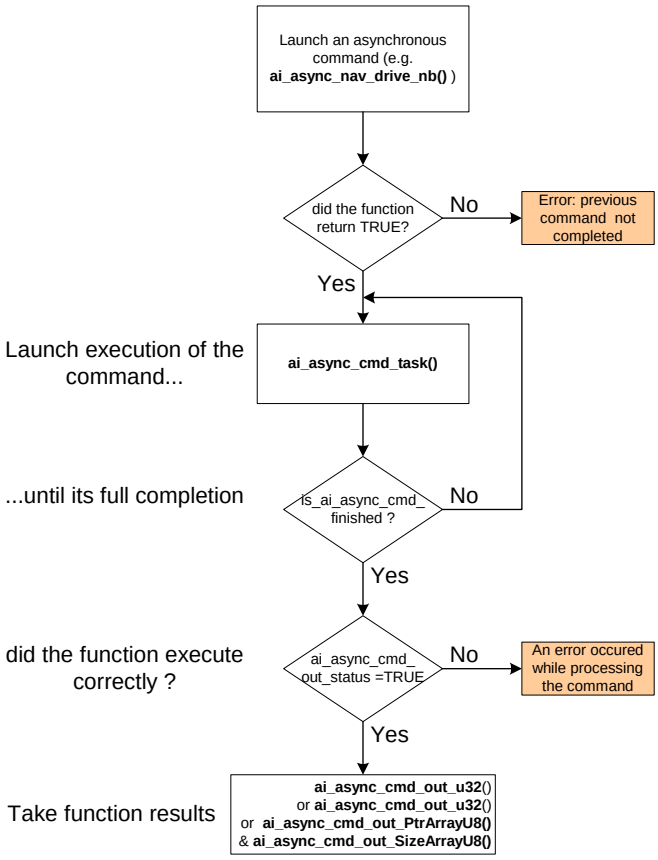
Using asynchronous function is more complicated: they *may* not produce the requested task in a single shot. Thus these functions need some other functions to properly operate:

- void ai_async_cmd_task(void): This function is the entry point of all asynchronous commands. It must be called to execute the current state of the internal state machine of the current asynchronous function.
- Bool is_ai_async_cmd_finished (void): This function returns TRUE if the last command is finished.
- U8 ai_async_cmd_out_status(void): Returns the status of the last executed command (CMD_DONE, CMD_EXECUTING or CMD_ERROR).

- U32 ai_async_cmd_out_u32(void): if the last executed command should return a 32-bit result or less, this function will return this value.
- U64 ai_async_cmd_out_u64(void): if the last executed command should return a 64-bit result, this function will return this value.
- U16 ai_async_cmd_out_SizeArrayU8 (void): if the last executed command should return an extra result (e.g. a song name), this function returns the size in bytes of the extra result (no size limit).
- U8* ai_async_cmd_out_PtrArrayU8 (void): Returns a pointer to the extra result (assuming that the last executed command returns an extra result). This pointer can be freed by the application with the ai_async_cmd_out_free_ArrayU8() function.
- void ai_async_cmd_out_free_ArrayU8 (void):This function may be called to free the allocated buffer which holds the extra-result. Note that the Audio Interface will automatically do this before executing a new command that need extra-results. This ensures that the application will not have memory leakage. Allowing the application calling this function will free the extra-results sooner and improve allocated memory usage.

The following picture shows the flow of the asynchronous function use.

Figure 5-2. Asynchronous function flow



5.8 Examples

The following examples are using a disk with the following contents:

Let's take this disk as disk number 0 for the system.

5.8.1 Example 1 - Play “file1.mp3”

Table 5-5. Example: play “file1.mp3”

Command Order	Command Name
0	**ai_nav_drive_nb**(): returns **1** disk.
1	**ai_nav_drive_set**(0): selects the disk 0.
2	**ai_nav_drive_mount**(): mounts the select disk 0.
3	**ai_nav_file_goto**(0): goes to file position 0
4	**ai_nav_file_name**(): returns the name dir1
5	**ai_nav_file_isdir**(): returns true, the current file is a directory.
6	**ai_nav_file_goto**(1): goes to file position 1
7	**ai_nav_file_name**(): returns the name dir2
8	**ai_nav_file_goto**(2): goes to file position 2
9	**ai_nav_file_name**(): returns the name file1.mp3
10	**ai_audio_nav_playfile**(): selects the file selected by the file navigator (file1.mp3)
11	**ai_audio_ctrl_resume**(): plays the selected file.
12	*Wait for 10 seconds to listen to the begining of the playback.*
13	**ai_nav_file_goto**(3): goes to file position 3
14	**ai_nav_file_name**(): returns the name file2.wma

5.8.2 Example 2 - Play while browsing

Table 5-6. Example: play while browsing

Command Order	Command Name
0	**ai_nav_drive_nb**(): returns **1** disk.
1	**ai_nav_drive_set**(0): selects the disk 0.
2	**ai_nav_drive_mount**(): mounts the select disk 0.
3	**ai_audio_nav_playfile**(): selects a file in the audio navigator. By default it will seek inside the directories to play the first file which is “file5.wma” in our case.
4	**ai_audio_ctrl_resume**(): plays the selected file.
5	**ai_nav_getplayfile**(): synchronizes the disk navigator with the audio navigator. Now the disk navigator is pointing on the “file5.wma” file.
6	**ai_nav_file_nb**() + **ai_nav_dir_nb**() + **ai_nav_playlist_nb**(): gets the total number of entries (files+folder+playlist) in the current directory (dir3).
7	**ai_nav_file_goto**(0): goes to file position 0
8	**ai_nav_file_name**(): returns the name “file5.wma“. (Notice the difference with Figure 5.8.1 - step #4)
9	**ai_nav_file_goto**(1): goes to file position 1
10	**ai_nav_file_name**(): returns the name “file6.mp3“
11	**ai_audio_ctrl_stop**(): stops the audio

5.8.3 Example 3 - Change the playing order

The playing order can be changed at compilation time by enabling the NAV_AUTO_FILE_IN_FIRST define (Section 6.5 “Project Configuration” on page 25).

Table 5-7. Playfile sequence

Command Order	Command Name
0	**ai_nav_drive_nb**(): returns **1** disk.
1	**ai_nav_drive_set**(0): selects the disk 0.
2	**ai_nav_drive_mount**(): mounts the select disk 0.
3	**ai_audio_nav_playfile**(): selects a file in the audio player.
4	**ai_audio_ctrl_resume**(): plays the selected file.

If the NAV_AUTO_FILE_IN_FIRST define is **not** set, the sequence will play audio files in the following order:

Table 5-8. Playfile sequence with NAV_AUTO_FILE_IN_FIRST **undefined**

Order	File name	Parent directory path
0	file5.wma	/dir1/dir3/
1	file6.mp3	/dir1/dir3/
2	file4.mp3	/dir1/
3	file7.mp3	/dir2/
4	file1.mp3	/
5	file2.wma	/
6	file3.wma	/

Otherwise, if this define is set, the sequence will play audio files starting with files on the root:

Table 5-9. Playfile sequence with NAV_AUTO_FILE_IN_FIRST **defined**

Order	File name	Parent directory path
0	file1.mp3	/
1	file2.wma	/
2	file3.wma	/
3	file4.mp3	/dir1/
4	file5.wma	/dir1/dir3/
5	file6.mp3	/dir1/dir3/
6	file7.mp3	/dir2/

6. Source Code Architecture

6.1 Package

The AUDIO-PLAYER-<board(s)>-<feature(s)>-<version>.zip contains projects for GCC (or AVR32Studio) and IAR.

Default hardware configuration of the project is to run on Atmel AVR32 UC3 Evaluation Kits, although any board can be used (refer to section 6.4.7 “Board Definition Files” on page 21).

6.2 Documentation

For full source code documentation, please refer to the auto-generated Doxygen source code documentation found in:

- /APPLICATIONS/AUDIO-PLAYER/readme.html

6.3 Projects / Compiler

The **IAR™** project is located here:

- /APPLICATIONS/AUDIO-PLAYER/<part>-<board>-<feature(s)>/IAR/

The **GCC** makefile is located here:

- /APPLICATIONS/AUDIO-PLAYER/<part>-<board>-<feature(s)>/GCC/

An **AVR32Studio** project can be easily created by following the steps from the “AVR32769: How to Compile the standalone AVR32 Software Framework in AVR32 Studio V2” application note.

6.4 Implementation Details

The following describes the code implementation of the MP3 audio player running on the EVK1105. Other available packages are similar so you will find usefull information here that applies to every project configurations.

6.4.1 Main()

The main() function of the program is located in the file:

- /APPLICATIONS/AUDIO-PLAYER/main.c

This function will:

- Initialize audio output - refer to section 6.4.8 “Audio Rendering Interface” on page 21
- Do the clock configuration
- Call the USB task -refer to section 6.4.5.1 “USB” on page 20
- Call the USB host Mass-Storage task. This task will check for new devices connection and initialize them using the USB Mass Storage protocol.
- Call the communication task (HMI) - refer to 6.4.4 “HMI Communication Task Example” on page 19.
- Call the decoder task to perform MP3 decoding - refer to sections 6.4.2 “MP3 Decoder” on page 19

The /APPLICATIONS/AUDIO-PLAYER/ contains the following files:

- ./main.c: contains the main() function.
- ./com_task.c,h: HMI core. See Section 6.4.4 “HMI Communication Task Example” on page 19 for more details.
  - ./USER_INTERFACE/CONTROLLER/joystick_controller.c: HMI using a joystick interface as a controller.
  - ./USER_INTERFACE/GUI/et024006dhu_gui.c: HMI using a LCD screen as a display.
- ./codec_mp3_task.c: handle the MP3 decoding task.
- ./CONF/*.h and ./<part>-<board>-<features>/conf_audio_player.h: configuration file for audio, communication interface, memory and navigation explorer. Please refer to section 6.5 “Project Configuration” on page 25 for more information on the configuration files.

6.4.2 MP3 Decoder

The MP3 decoder source files are located in:

- /SERVICES/AUDIO/MP3/LIBMAD/: AVR32 port of LibMAD MP3 decoder

A library of the decoder is provided in /UTILS/LIBS/LIBMAD/AT32UC/.

ID3 is supported up to version 2.4. The ID3 reader source is located in:

- /SERVICES/AUDIO/MP3/ID3/reader_id3.c,h

6.4.3 Audio Player API

The Audio Interface API is located in:

- /SERVICES/AUDIO/AUDIO_PLAYER/audio_interface.h

The Mass Storage Audio Interface can be found in:

- /SERVICES/AUDIO/AUDIO_PLAYER/AI_USB_MS/
  - ./ai_usb_ms.c,h: Mass Storage Audio interface.
  - ./ai_usb_ms_mp3_support.c,h: add support to the MP3 file format in the audio interface.
  - ./host_mass_storage_task.c,h: USB host mass storage task.

Refer to Section 5. “Audio Interface API” on page 5 for more details.

6.4.4 HMI Communication Task Example

The included firmware implements an HMI example using a joystick and a SPI-driven LCD: (source code is located under /APPLICATIONS/AUDIO-PLAYER-MASS-STORAGE/USER_INTERFACE/).

All the HMI is based on a pair of files, com_task.[c|h], which implements all mechanisms used to communicate between the internal APIs of the audio player and the user’s HMI. An abstraction layer is used to attach easily all kinds of controller and graphical user interface to this communication task. It has been done to easily port the application to another board.

6.4.4.1 Controller

The controller is the module that makes the link between the driver and the abstraction layer used to ensure compatibility with the communication task. It is based on mainly two API groups: 'setup' and 'events'. All the APIs are defined in the file /APPLICATIONS/AUDIO-PLAYER/USER_INTERFACE/CONTROLLER/controller.h.

The function *controller_init* is used to initialize the controller. It may be used or not, depending on the controller implementation.

All the other functions are boolean functions, returning **true** if an event has been raised or **false** otherwise. Their name is explicit and follows the following naming convention:

Bool controller_<view>_<event>(void);

Where *<view>* corresponds to the current view when the event should be taken into account. If no value is set to this field, the event is applied to all views.

The *<event>* is a short name describing the current event for which the function applies to.

This example uses a joystick controller. The source code relative to this module is available in the file /APPLICATIONS/AUDIO-PLAYER/USER_INTERFACE/CONTROLLER/joystick_controller.c.

6.4.4.2 Graphical User Interface

This module is used to display to the user the high level audio player data. It acts as a display and when combined with the controller it provides to the user a full control of the application.

This module is similar to the controller in terms of API groups.

The initialization function is called *gui_init* and takes into parameters all useful frequencies that can be used to initialize the display.

The 'event' group (to keep the same architecture as the controller), is composed of only one function: *gui_update*. This function is called at every occurrence of the main loop and takes into parameters flags, describing which part of the view have to be updated. The resetting of these flag is let to the GUI module so that is can achieve the update of the display asynchronously and does not have to update every component at once.

The example provides a graphical LCD display module that implements both a navigation and a playback view. All the code is based on the files /APPLICATIONS/AUDIO-PLAYER/USER_INTERFACE/GUI/[et024006dhu_gui.c|sdram_loader.c].

6.4.5 AT32UC3A Drivers

The firmware uses the AVR32 UC3 driver library available in

- /UTILS/LIBS/DRIVERS/AT32UC3A/.

The source code can be found into /DRIVERS.

6.4.5.1 USB

The USB low level driver is located in:

- /DRIVERS/USB/

The USB mass storage service is located in:

- /SERVICES/USB/CLASS/MASS_STORAGE/

6.4.6 FAT File System

The FAT12/16/32 files is located in the directory /SERVICES/FAT/.

6.4.7 Board Definition Files

The application is designed to run on Atmel Evaluation Kits. All projects are configured with the following define: BOARD=EVKxxxx. The EVKxxxx definition can be found in the /BOARDS/EVKxxxx directory.

6.4.7.1 Board customization

- For '**IAR**' project, open the project options (Project -> Options), choose the «C/C++ Compiler», then «Preprocessor». Modify the 'BOARD=EVKxxxx' definition by 'BOARD=USER_BOARD'.
- For '**GCC**', just modify in the 'config.mk' file (/APPLICATIONS/AUDIO-PLAYER/<part>-<board>-<feature(s)>/GCC/) the DEFS definition with '-D BOARD=USER_BOARD'.
- For '**AVR32 Studio**', open the project properties (Project -> Properties), go in the «C/C++ build», then «Settings», «tool settings» and «Symbols». Modify the 'BOARD=EVKxxxx' definition by 'BOARD=USER_BOARD'.

The HMI can be easily changed. See Section 6.4.4 "HMI Communication Task Example" on page 19 for more details.

6.4.8 Audio Rendering Interface

The audio DAC mixer source code is lodated in /SERVICES/AUDIO/AUDIO_MIXER/audio_mixer.c,h.

6.4.8.1 I2S using SSC module

The /COMPONENTS/AUDIO/CODEC/TLV320AIC23B/ directory contains the driver for the external DAC TLV320AIC23B.

The UC3 communicates with the TLV320AIC23B with the Two Wire Interface (TWI). The UC3 is the TWI master.

The AVR32 SSC module generates I2S frames using internal DMA (PDCA) to free CPU cycles for audio decoding.

Each time a new song is played, the SSC module is configured corresponding to the sample rate of the new song. The SSC clocks are composed of a bit clock and a frame sync:

- The bit clock is the clock used to transmit a bit from the audio stream. For a 44.1 KHz sample rate, the bit clock will be 44100 x 2 (stereo) x 16 (bits per channel) i.e. 1.411 MHz.
- The Frame sync is equal to the sample rate frequency, i.e. 44.1 KHz taking the same example.

To get accurate 44.1KHz audio frequency samples, an external 11.2896 MHz oscillator is used as input to an internal PLL and source the CPU/HSB/PBA/PBB frequency with 62.0928 MHz.

The TLV320AIC23B uses a master clock (MCLK) of 11.2896 MHz, outputed by the UC3 through a generic clock. Then, the generic clock output (PA07) is connected to the MCLK of the TLV320AIC23B.

The SSC clock divider in CMR register is given by:

$$SSC.CMR.DIV = Round\ (F_{PBA} / (2 \times (F_{SampleRateSetPoint} \times NumberChannel \times BitsPerSamples))$$

The real frequency sample rate output by the SSC is given by:

$F_{ActualSampleRate} = F_{PBA} / (2 \times SSC.CMR.DIV \times NumberChannel \times BitsPerSamples)$

Note: Note: NumberChannel = 2, BitsPerSamples = 16, F_{PBA} = 62.0928 MHz

The music interval in semitones is:

$MusicInterval\ (semitones) = LOG((F_{ActualSampleRate} / F_{SampleRateSetPoint}) / (2^{1/12}))$

Table 6-1. Samples rate with SSC module

Sample rate set point (Hz)	Actual Sample Rate (Hz)	Relative Error (%)	Music Interval (semitones)
8000	8018	0.23	0.04
11025	11025	0.00	0.00
16000	15095	-0.59	-0.10
22050	22050	0.00	0.00
32000	32340	1.06	0.18
44100	44100	0.00	0.00
48000	48510	1.06	0.18

For further information, please refer to the "*AVR32788: AVR®32 How to use the SSC in I2S mode*" application note.

6.4.8.2 *ABDAC*

The ABDAC driver is located in /DRIVERS/ABDAC.

The external amplifier driver (TPA6130A) is located in /COMPONENTS/AUDIO/AMP/TPA6130A.

The AVR32 ABDAC module is using the internal DMA (PDCA) to free CPU cycles for audio decoding.

To get accurate audio frequency samples, the two external oscillators 12 MHz (OSC1) and 11.2896 MHz (OSC0) are used to source (directly or through a PLL) the ABDAC generic clock.

The PLL0 output frequency is 62.0928 MHz. The PLL1 output frequency is 48 MHz (used for USB).

When used, the ABDAC generic clock divider is given by:

$ABDAC\ generic\ clock\ divider = Round (F_{GCLKInput} / (2 \times 256 \times (F_{SampleRateSetPoint})) - 1)$

Refer to /DRIVERS/ABDAC/abdac.c for the configuration of the ABDAC generic clock.

Table 6-2. Samples rate with ABDAC module

Sample rate set point (Hz)	ABDAC Generic clock input frequency	Actual Sample Rate (Hz)	Relative Error (%)	Music Interval (semitones)
8000	PLL0	8085	1.06	0.18
11025	OSC1	11025	0.00	0.00
16000	PLL1	15625	-2.34	-0.41
22050	OSC1	22050	0.00	0.00

Sample rate set point (Hz)	ABDAC Generic clock input frequency	Actual Sample Rate (Hz)	Relative Error (%)	Music Interval (semitones)
32000	PLL1	31250	-2.34	-0.41
44100	OSC1	44100	0.00	0.00
48000	PLL1	46875	-2.34	-0.41

6.4.9 SDRAM Loader

Graphical data like images consume a lot of space in RAM and flash which could be used by the application instead. Because of that, the images that are used by the audio player application GUI are stored in DataFlash and are loaded to SDRAM upon startup. The reason for having the data in the SDRAM, rather than in flash, is the speed improvement that leads to faster updates of the display with new content.

The SDRAM loader module uses a memory manager to manage the SDRAM memory space. This memory manager can also be used in the application to allocate memory from SDRAM instead of the internal SRAM.

The following sections give a short introduction to the parts involved in the SDRAM loader module.

The SDRAM loader source code is located in /APPLICATIONS/AUDIO-PLAYER/USER_INTERFACE_GUI/sdram_loader.c,h.

6.4.9.1 Dataflash

The DataFlash is formatted with a FAT16 file system and it currently contains graphical data that is used by the audio player application in the GUI implementation. The graphical data is stored as BMP files and the used format is RGB565 which can be used directly on the display when swapped from little endian to big endian. Other BMP formats are currently not implemented but this can be extended if needed.

The FAT file system makes it easy for developers to upgrade the content with new files or even use it for other applications like a web-server. An upgrade of the content can be done by using a mass storage example application.

The BMP picture used in the application are stored into the /PICTURES directory.

To customize the GUI, the bitmap files can be updated and must be saved to the ‘Windows bitmap image’ format using a ‘16-bit R5 G6 B5’ encoding. This can be done using ‘GIMP’, the GNU Image Manipulation Program (see http://www.gimp.org/).

6.4.9.2 Loading Process

The SDRAM loader consists of two files, sdram_loader.c and sdram_loader.h, in the directory APPLICATIONS/AUDIO-PLAYER/USER_INTERFACE/GUI. In these files the images are specified that should be loaded to SDRAM and how they should be converted. A configuration that loads three images to SDRAM could look like this:

```
typedef struct {
  const wchar_t *name;
  void *start_addr;
} ram_file_t;
…
```

```
#define STARTUP_IMAGE       0
#define DISK_NAV_IMAGE      1
#define AUDIO_PLAYER_IMAGE  2
#define NUMBER_OF_IMAGES    13

ram_file_t ram_files[NUMBER_OF_IMAGES] = {
        { .name = L"/AVR32_start_320x240_RGB565.bmp"},
        { .name = L"/disk_nav_320x240_RGB565.bmp"},
        { .name = L"/audio_player.bmp"}
};
```

The ram_files array is used throughout the GUI to get access to the image data. In order to load other data than RGB565 BMP data to SDRAM the module needs to be modified.

The SDRAM loader module is called once during the initialization of the graphical user interface. When called it initializes the SDRAM interface and reads the raw image data from specified BMP files into SDRAM. To get the raw image data the BMP header must be read to get the image size and the offset of the data in the file. The copy process does also a conversion from the little endian to the big endian data ordering and because of that the final image data can be dumped directly into the display buffer.

A single call to the SDRAM module is enough to do the initialization and load process:

```
void load_sdram_data(int hsb_hz);
```

The sole parameter is the HSB frequency in hertz which is needed to initialize the SDRAM timings. The above function is called starting from main in the following order:

```
main() -> com_task() -> gui_init() -> sdram_load_data()
```

6.4.9.3 SDRAM Memory Management

The images could be placed at specified locations in SDRAM and thus make a memory management unneeded, but on the other hand it is often better to do memory management to remove the task of keeping track of the data in memory from the developer.

The audio player uses a separate memory manager to manage the SDRAM memory (this memory manager can also replace the default memory manager in the Newlib library if needed).

The source files of the memory manager are located in /SERVICES/MEMORY/MEMORY_MANAGER/DLMALLOC/ and the additional configuration of it is /APPLICATIONS/AUDIO-PLAYER/CONF/conf_dlmalloc.h.

The memory manager is initialized in the SDRAM loader module and it is configured to use the whole SDRAM memory. After the initialization memory can be allocated from the SDRAM with the mspace_malloc call. Memory from the internal SRAM can be allocated with the default malloc call.

6.4.10 JPEG Decoder

The JPEG decoder is used for the MP3 cover art file support.

The source code of the IJG JPEG decoder is located under /SERVICES/PICTURES/JPG/IJG/. Documentation of the library can be found in /SERVICES/PICTURES/JPG/IJG/libjpeg.doc and an overview in the README file.

The IJG license text can be found in /SERVICES/PICTURES/JPG/IJG/license.txt.

The JPEG decoder application source are located under /APPLICATIONS/AUDIO-PLAYER/JPG/. This is a layer that handles the data input and output of the JPEG library. It also handles the error handling to jump out of the library in case of a critical error. Since the JPEG decoding is done in the SDRAM, the memory management back-end of the library is located here too, which normally would be included in the library itself. The memory management back-end is modified to use the SDRAM with the DLMALLOC memory manager (in /SERVICES/MEMORY/MEMORY_MANAGER/DLMALLOC/) instead of the standard malloc implementation.

6.5 Project Configuration

There are two different configuration files. The one related to the audio player itself, located under /APPLICATIONS/AUDIO-PLAYER/<part>-<board>-<feature(s)>/conf_audio_player.h, is mostly a high level configuration file and can be customized to support certain configurations or enable/disable audio player features. The rest of the configuration files are located under the /APPLICATIONS/AUDIO-PLAYER/CONF/ directory and should be modified to change, separatly, audio player's module configurations.

Configuration files are not linked to 'IAR', 'GCC' or 'Avr32Studio' projects. The user can alter any of them, then rebuild the entire project in order to reflect the new configuration.

6.5.1 High level configuration file: /APPLICATIONS/AUDIO-PLAYER/<part>-<board>-<feature(s)>/conf_audio_player.h.

- **DEFAULT_DACS**, specifies the default audio DAC used for the audio output. 3 values are possible: AUDIO_MIXER_DAC_AIC23B for the I2S interface , AUDIO_MIXER_DAC_ABDAC for the internal DAC (ABDAC module) , and AUDIO_MIXER_DAC_PWM_DAC to use PWM channels (external low-pass filter is required).
- **USE_AUDIO_PLAYER_BUFFERIZATION**, set to 'ENABLED' to support navigation while playing feature, it will use extra memory to buffer decoded samples in order to prevent audio blips while the user navigates in the disk architecture for example. The memory address and size can be configured in the file '**conf_buff_player.h**' (see bellow). *Note that using the audio sample bufferization will not ensure that every audio blip will be covered. It will always depends on the speed of the USB device connected, its file system, the operations requested. The default buffer size is set to 128K.*
- **SUPPORT_MP3, SUPPORT_PLAYLISTS, SUPPORT_EMBEDDED_COVER_ARTS**: a set of defines that can be enabled or disabled to reduce audio player features. Note that the use of the embedded cover arts requires RAM in order to decode embedded JPEG pictures. Therefore it needs an external memory to handle this feature.

6.5.2 Low level configuration files: /APPLICATIONS/AUDIO-PLAYER/CONF/*.h.

6.5.2.1 conf_access.h

This file contains the possible external configuration of the memory access control. It configures the abstract layer between the memory and the application and specifies the commands used in order to access the memory. For example, this file will define the functions to be called for a SD/MMC memory access.

6.5.2.2 conf_audio_interface.h

A set of configuration flags to enable/disable internal features of the audio player.

6.5.2.3 *conf_audio_mixer.h*

Configures all parameters relative to the audio DACs. This file is made to support multiple configurations and can be easily upgraded to handle new DACs.

6.5.2.4 *conf_buff_player.h*

Defines the starting address and the size of the memory used to bufferize audio samples (if the feature is enabled).

6.5.2.5 *conf_dmalloc.h*

Configuration of malloc/free functions.

6.5.2.6 *conf_explorer.h*

It defines the configuration used by the FAT file system. The configuration is also applied to the playlist handler and the file navigation. The main parameters are:

- **NAV_AUTO_FILE_IN_FIRST**, must be define in order to play first the files at the root of a directory instead of the one inside the subdirectories.
- **FS_NAV_AUTOMATIC_NBFILE**, this flag can be set to DISABLE in order to speed up the response of the ai_audio_nav_playfile command. On the other hand, the three commands ai_audio_nav_getpos, ai_audio_nav_getpos and ai_audio_nav_nb will not be available anymore. It will also affect the use of the explorer modes, if different from “all disks” and “one disk”.

6.5.2.7 *conf_jepg_decoder*

Configuration of the JPEG decoder.

6.5.2.8 *conf_pwm_dac.h*

Configuration of the PWM DAC (which PWM channel is used, which pins are concerned).

6.5.2.9 *conf_tlv320aic23b.h*

Configuration of the external I2S DAC (which pins are used and which configuration interface).

6.5.2.10 *conf_usb.h*

Configuration file used for the USB.

6.5.2.11 *conf_version.h*

Internal version of the firmware.

6.6 Compiling the application

The following steps show you how to build the embedded firmware according to your environment.

6.6.1 If you are using AVR32 Studio

- Please refer to the application note “AVR32769: How to Compile the standalone AVR32 Software Framework in AVR32 Studio V2”

6.6.2 If you are using the standalone GCC with the AVR32 GNU Toolchain

- - Open a shell, go to the /APPLICATIONS/AUDIO-PLAYER/<part>-<board>-<feature(s)>/GCC/ directory and type:

  make rebuild program run

6.6.3 If you are using IAR Embedded Workbench® for Atmel AVR32

- - Open IAR and load the associated IAR project of this application (located in the directory /APPLICATIONS/AUDIO-PLAYER/<part>-<board>-<feature(s)>/IAR/).
- - Press the “Debug” button at the top right of the IAR interface.

  The project should compile. Then the generated binary file is downloaded to the microcontroller to finally switch to the debug mode.
- - Click on the “Go” button in the “Debug” menu or press F5.

7. FAQ

Q: What is the maximum number of playlist links supported?

A: The file system supports up to 65535 links inside a playlist.

Q: What are the supported text formats?

A: The file system supports the ASCII, UTF8 and UNICODE (UTF16LE & UTF16BE) text formats.

Q: How are the directories and files sorted inside the disk?

A: The logical structure is the same as an explorer view. Directory and files are sorted using their creation order.

Q: Which file systems are supported?

A: FAT 12/16/32.

Q: What is the maximum of files supported in a directory?

A: There is no limitation in the firmware for the supported number of files and directories. The only limitation is due to the FAT file system:

- for FAT12/16 root directory only: up to 256 files (short names),
- for FAT12/16/32 up to 65535 files (short names) per directory.

Q: What is the minimum RAM requirement to run this application?

A: The default application is using external SDRAM to support all features. This application can run with 64K of RAM (internal size on AT32UC3A0512) by disabling the audio bufferization and the cover art support. Audio blips will be present if the user try to navigate in the disk while a track is played. The SDRAM loader also uses external SDRAM memory for graphical data.

Q: What kind of license apply to the software?

A: Atmel provides the audio decoder software library free of charge. But some digital audio formats, including MP3 and WMA, contains technology that is patented, and a license for these patents must be obtained before the library can be used.

Atmel provides application note ”AVR32722 How to license audio and video codecs” on the www.atmel.com website. This serves as a guide to give Atmel® customers information about, as well as an overview over, the licensing of patents that cover technologies such as MP3 and WMA.

Atmel Corporation
2325 Orchard Parkway
San Jose, CA 95131
USA
Tel: (+1)(408) 441-0311
Fax: (+1)(408) 487-2600
www.atmel.com

Atmel Asia Limited
Unit 1-5 & 16, 19/F
BEA Tower, Millennium City 5
418 Kwun Tong Road
Kwun Tong, Kowloon
HONG KONG
Tel: (+852) 2245-6100
Fax: (+852) 2722-1369

Atmel Munich GmbH
Business Campus
Parkring 4
D-85748 Garching b. Munich
GERMANY
Tel: (+49) 89-31970-0
Fax: (+49) 89-3194621

Atmel Japan
9F, Tonetsu Shinkawa Bldg.
1-24-8 Shinkawa
Chuo-ku, Tokyo 104-0033
JAPAN
Tel: (+81)(3) 3523-3551
Fax: (+81)(3) 3523-7581