New LIBDS CD Library and Runtime Data Decompression

LIBDS Overview

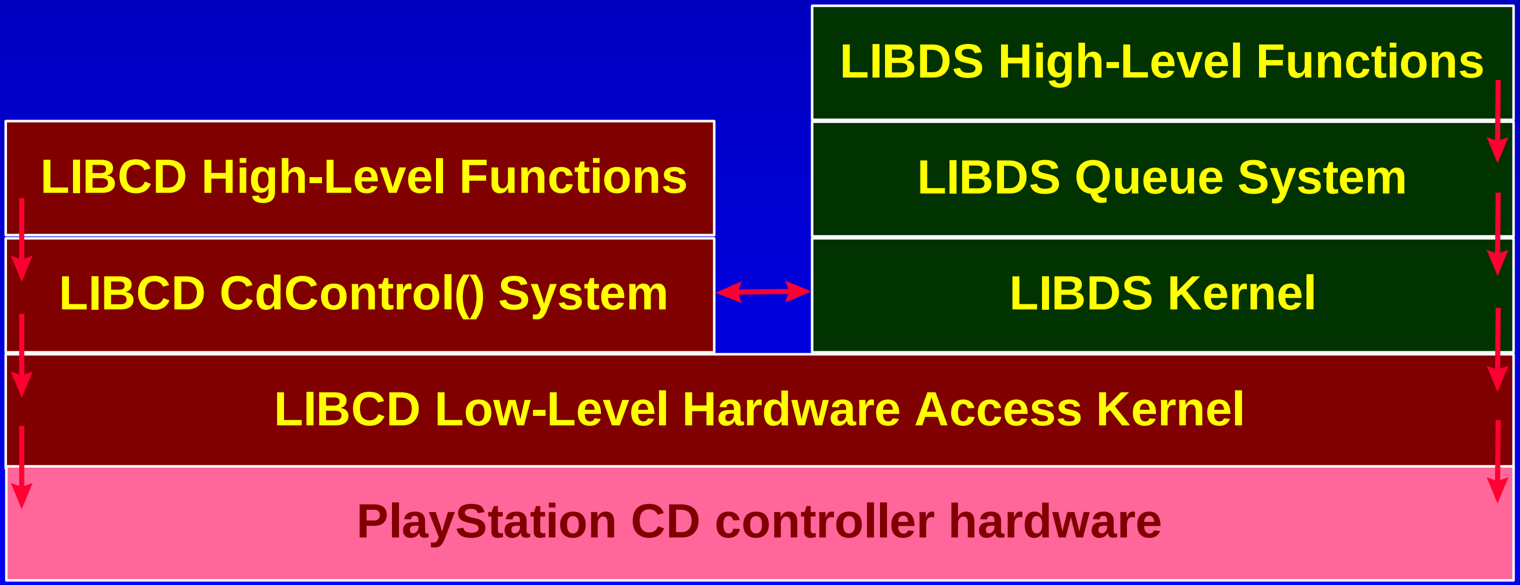
- What is LIBDS?
- Differences between LIBDS and LIBCD
- Using LIBDS

What is LIBDS?

- A new library for controlling the CD
- An API interface that supercedes LIBCD
- A library providing enhanced error recovery

What is LIBDS?

- Relationship between LIBDS and LIBCD
  - LIBDS uses low-level functions from LIBCD
  - Must use same version of LIBCD and LIBDS together


LIBDS High-Level Functions
LIBCD High-Level Functions
LIBDS Queue System
LIBCD CdControl() System
LIBDS Kernel
LIBCD Low-Level Hardware Access Kernel
PlayStation CD controller hardware

What is LIBDS?

- Relationship with other libraries
  - Streaming functions use LIBCD
    - Functions which start streaming CD access use LIBDS
      - Use DsRead2() instead of CdRead2().
    - When streaming, only LIBDS need be initialized

Differences between LIBDS & LIBCD

- LIBCD does not support command queuing
  - CdControl() from LIBCD always waits for previous command to complete.
  - Application must keep track of commands until each CD command is completed and CD subsystem is available again.

Differences between LIBDS & LIBCD

- LIBDS Supports Command Queuing
  - LIBDS features a command queue that allows non-blocking execution of CD commands
    - Reduces blocking time when commands are issued
    - When CD subsystem becomes available, stored commands are executed in the same order they were issued.

Differences between LIBDS & LIBCD

- The LIBDS Command Packet
  - Allows you to combine the four commands typically required to do a read operation.
  - Deals with retries as specified by your application.

Differences between LIBDS & LIBCD

- Enhanced Error Recovery Features
  - Retry count may be specified, allows unlimited retries.
  - Command packets given unique ID for later identification
  - Library support provided for error checking

Differences between LIBDS & LIBCD

- Opening & Closing of the CD cover is automatically detected
  - Recovery processing is automated.
  - Queue processing put on hold until recovery completed.
    - Restarts at VBLANK period following recovery
- Changes in CD transfer speed are automatically detected
  - CD Command execution is automatically blocked for 3/60ths to allow speed change to complete.
    - Incoming CD commands diverted to queue

Differences between LIBDS & LIBCD

- Constants
  - LIBDS.H has all of the same constants defined by LIBCD.H
    - 1st 3 letters changed from “Cdl” to “Dsl”
      - For example, CdlPause changed to DslPause
- Functions
  - LIBDS has most of the same commands of LIBCD
    - 1st 2 letters changed from “Cd” to “Ds”.
      - For example CdSync changed to DsSync
    - Arguments for some functions may be different
    - Execution timing for functions may be different

Using LIBDS

- Initializing LIBDS
- Resetting LIBDS
- Exiting LIBDS

Initializing LIBDS

- Use DsInit() function
- Call after ResetGraph(), InitPAD(), and InitCARD()
- Cannot mix LIBCD and LIBDS calls
    - Use LIBDS calls for streaming
        - DsRead2()
        - DslReadS

Resetting LIBDS

- Use DsFlush() call.
  - Flushes the CD subsystem
  - Clears the command queue
- Use DsReset() call.
  - Similar to DsFlush
  - Also resets callback routines set by your program
- These calls do not stop ongoing read / playback operations
  - Issue a DslPause command

Exiting LIBDS

- Use DsClose() call
  - Always exit LIBDS prior to using Load() or LoadExec() for child process or overlay.
- Call DsInit() again to restart LIBDS

Using LIBDS

- The Command Queue
  - Issuing Commands
  - Command Packets
  - Confirming Command Completion
  - Checking Queue & System Status
- Simplified Data Ready Callback System

The Command Queue

- Controls the issuing and completion of CD primitive commands & automates the processes required to operate the CD subsystem
  - Commands are immediately executed if CD subsystem is available.
  - Otherwise, commands are placed into queue.
    - Eliminates blocking time when commands are issued
    - Queue processing is completely callback driven.
    - When CD subsystem becomes available, stored commands are executed in the same order they were issued.

Issuing Commands

- The DsCommand function is used to place primitive commands into the command queue.
  - Multiple processes cannot enter commands in the queue.
  - When you start to issue a command, the queue is closed until the command is successfully issued.

```
int command_id = DsCommand( u_char command_code,
                            u_char *parameters,
                            DslCB  *callback_function,
                            int    retry_count );
```

Issuing Commands

- The *command_code* argument of DsCommand().
  - Specifies command to be entered into queue.
    - Most commands cannot be placed in queue immediately after a read or play command.
      - OK commands are DslNop, DslGetlocP, DslGetlocL, DslPause, DslStandby, DslStop
  - Defined in LIBDS.H, same as CdControl commands:

DslNop
DslSetloc
DslPlay
DslForward
DslBackward
DslReadN
DslStandby
DslStop
DslPause
DslMute
DslDemute
DslSetfilter
DslSetmode
DslGetparam
DslGetlocL
DslGetlocP
DslGetTN
DslGetTD
DslSeekL
DskSeekP
DslReadS

Issuing Commands

- The *parameters* argument of DsCommand().
  - Most commands do not take parameters
    - Pass a NULL value
  - Others take a pointer to a data structure:
    - DslATV
    - DslFILE
    - DslFILTER
    - DslLOC
      - Correspond to LIBCD structures

Issuing Commands

- The *callback_function* argument of DsCommand().
  - LIBDS allows a separate Sync callback function to be set for each command issued using DsCommand.
    - A NULL value indicates no specific callback routine, Otherwise, a pointer to a function of type DslCB.
  - If a particular callback routine is not specified in the *callback_function* argument of DsCommand(), then the routine specified for DsSyncCallback() is used instead.
  - For read commands, callbacks for each sector are issued through the DsReadyCallback() mechanism.

```
typedef void ( *DslCB )( u_char, u_char* );
```

Issuing Commands

- The *retry_count* argument of DsCommand().
  - The actual commands issued using DsCommand() are performed in the background.
  - If execution of a command fails, it will be retried automatically by LIBDS according to the *retry_count* argument.
    - If *retry_count* is -1, then it will do unlimited retries.
    - If *retry_count* is 0, t hen it will not do any retries.
  - Neither the DsSyncCallback or the *callback_function* callback routines will be triggered during a retry.
    - CdSyncCallback will be triggered and is used by LIBDS

Issuing Commands

- The *command_id* value returned by DsCommand()
  - A command ID code that uniquely identifies the particular instance of that command.
    - Completion status of specific commands can be obtained from DsSync() function.

Command Packet

- The command packet allows the multiple commands required for a read operation to be combined so that they may be issued together in a batch.
- Special feature of command queue
  - Packet commands are issued using a CdSync chain
  - When all commands succeed, or when the error retry count is exceeded, a callback is triggered.
  - Reliable retries may be performed when errors occur

Command Packet

- LIBDS creates four commands in the queue to process the packet request.
  1) DslPause
  2) DslSetMode
  3) DslSetloc
  4) The command specified by the *command* parameter:

     DslReadN, DslReadS, DslPlay, DslSeekP, or DslSeekL
  - The Command Queue must have four empty slots to successfully register a packet.
  - Packet ends when all commands have succeeded.
  - Packet not removed from queue until it has completed.

Command Packet

- To enter a packet in the queue, useDsPacket().

```
int packet_id = DsPacket( u_char mode, DslLOC *pos,
                          u_char command,
                          DslCB callback_function,
                          int retry_count );

packet_id =                 return code.
                            0 = command was not issued OK
                            <>0 = unique packet ID code
mode =                      DslSetmode parameter
pos =                       pointer to DslLOC timecode specification
command =                   Command (for example DslPlay)
callback_function =         DslCB containing pointer to a callback
                            function that will be called when this
                            specific packet has been processed
retry_count =               number of desired retries.
                             0 = no retry, -1 = unlimited
```

Command Packet

- The *packet_id* value returned by DsPacket()
  - A packet ID code that uniquely identifies the particular instance of that entire packet.
  - Completion status of the packet can be obtained from DsSync() function.
- The *mode* argument of DsPacket().
  - Specifies the mode value for a DslSetmode command.
- The *pos* argument of DsPacket().
  - Specifies the timecode position for the packet operation.
    - Read / Play location or Seek destination

Command Packet

- The *command* argument of DsPacket().
  - Specifies the desired read / play / seek command
    - DslReadN, DslReadS
    - DslPlay
    - DslSeekP, DslSeekL
- The *callback_function* argument of DsPacket().
  - Specifies the callback routine to be executed when all of the commands executed by the packet have been successfully processed, or if an error occurs.
  - Basically the same as with DsCommand().

Command Packet

- The *retry_count* argument of DsPacket().
  - Specifies the number of times the commands issued by DsPacket() will be retries if an error occurs.
  - If execution of any single packet command fails, all of the commands are retried from the start of the packet according to the *retry_count* argument.
    - If *retry_count* is -1, then it will do unlimited retries.
    - If *retry_count* is 0, t hen it will not do any retries.
  - If the number of retries is exceeded, the packet triggers the callback routine specified by the *callback_function* argument.

Command Packet

- The *packet_id* value returned by DsPacket()
  - A packet ID code that uniquely identifies the particular instance of that entire packet.
  - Completion status of the packet can be obtained from DsSync() function.

Confirming Command Completion

- The DsSync() routine can be used to obtain the results of an individual command or packet.
  - Execution results are saved in a ring buffer
    - Oldest results are overwritten by the newest results
    - Size specified by DslMaxRESULTS macro in LIBDS.H
      - Macro is for your information only and does actually affect size of ring buffer

```
int status = DsSync( int id, u_char* results );

status = Execution status of specified command
id = command ID returned by DsCommand or DsPacket
results = return value(s) from specified command (8 bytes)
```

Confirming Command Completion

- The *status* return value from DsSync()
  - Indicates the execution status of the specified command or packet.
    - DslComplete
      - Command executed normally
    - DslDiskError
      - Command generated an error
    - DslNoIntr
      - Command has not completed processing
    - DslNoResult
      - If requested results are no longer available in ring buffer.

Confirming Command Completion

- The *id* argument of DsSync()
  - A unique ID code that uniquely identifies a particular command or packet.
    - The *command_id* return value from DsCommand().
    - The *packet_id* return value from DsPacket().
- The *results* argument of DsSync()
  - A pointer to an array of 8 bytes which will receive the information returned from the specified command.
    - For example, the *DslLOC* timecode requested by DslGetlocL.

Checking Queue & System Status

- Checking the current queue status can be done using the DsQueueLen() function.
  - Returns the number of items which are currently waiting in the queue to be processed.
    - Includes any commands currently executing and not yet completed.
    - Maximum queue size specified by DslMaxCOMMANDS macro defined in LIBDS.H
      - Queue size is not configurable by application

```
int queue_length = DsQueueLen(void);

queue_length = Number of items currently in LIBDS queue
```

Checking Queue and System Status

- Checking the current CD subsystem status can be done using the DsSystemStatus() function.
  - Returns *status* code:
    - DslReady
      - Ready to execute command
    - DslBusy
      - Command being executed or command cannot be executed
    - DslNoCD
      - CD is not set (no CD loaded)

```
int status = DsSystemStatus(void);

status = Current status of CD subsystem
```

Simplified Data Ready Callback System

- LIBDS features a simplified Data Ready callback mechanism with automated error handling.
  - Subheader errors are checked
  - Library performs retry on errors automatically

```
int status = DsStartReadySystem( DslRCB func,
                       int retry_count );

status =Returns 1 if callback installed successfully,
                    0 if it failed (callback already installed
func = Pointer to the desired callback handler function
retry_count = # of times to retry read when errors occur.
                    0 = No retries
                    -1 = unlimited retries
```

Simplified Data Ready Callback System

- LIBDS does nothing when sector processing succeeds. Control is passed to specified callback so that sector data may be transfered using DsGetSector.
  - When error occurs, last performed read operation is retried according to the specified *retry_count*.
    - Sectors prior to the one which had the error are read again, but not passed to callback routine.
      - For best efficiency, avoid reading huge pieces of data in one chunk. Instead, break big reads into consecutive smaller reads.

Simplified Data Ready Callback System

- To shut down the callback, use the DsEndReadySystem function.
- Removes the currently installed callback previously setup with DsStartReadySystem.

```
void DsEndReadySystem(void);
```

Runtime Lossless Data Decompression

- Separate from LIBCD or LIBDS
- Uses Huffman coding for fast table-based runtime data decompression
  - Provides roughly 30-50% compression on average
  - Very fast decompression
  - Does not use static tables
    - Tables built at runtime
    - Requires about 2k temporary space

Runtime Lossless Data Decompression

- Commandline-based tool converts any desired data files into sector size chunks of Huffman-encoded data
  - Tool converts source files into sections which will compress into sector-size chunks.
  - As long as each sector is decompressed as it is received, only 1 sector of temporary buffer space is needed to store compressed data
- Runtime code decompresses data sector by sector as each one is read from CDROM

Runtime Lossless Data Decompression

- Advantages
  - Lossless compression suitable for code or data
  - Faster loading times
  - Less disc space required for data
  - Smaller memory footprint & faster decompression compared with other compression methods
- Disadvantages
  - Compression not as good as some other methods
  - Although fast, decompression is completely CPU dependent.

The End