

SPS Decoder Library and API Documentation

Revision 5.20

Table of Contents

COPYRIGHT NOTICE	5
REVISION HISTORY	5
PURPOSE	6
SOFTWARE PRESERVATION SOCIETY (SPS) AND IPF FORMAT	6
USER NOTES	9
DEVELOPER MANUAL	10
BACKGROUND	10
INTERFACE	10
READING DISK IMAGES	10
<i>Emulation</i>	11
<i>Virtual Drive</i>	12
WRITING DISK IMAGES	12
USING THE TRACK DATA	14
DISK ROTATION	14
DISK INDEX	14
TRACK DATA ALIGNMENT	15
CELL DENSITY MAP	15
MULTI-REVOLUTION TRACKS	16
DRIVE PROPERTIES	16
USING THE LIBRARY	18
COMPILING THE LIBRARY	19
COMPILING FOR WINDOWS	19
<i>To build using Visual Studio</i>	19
<i>To build from the command line</i>	20
COMPILING FOR MACOS AND LINUX	20
<i>Linux</i>	20
<i>macOS</i>	20
USING THE API	21
PROCESS CONTEXTS	21
MULTI-THREADING	21
POINTERS AND DATA PERSISTENCE	21
FILE SHARING	22
ERROR HANDLING	22
FREEING MEMORY	22
PROGRAMMING TASKS	23
OPENING AND CLOSING THE LIBRARY	23
CREATING AND DESTROYING IMAGE CONTAINERS	23
LOCKING AND UNLOCKING IMAGES	23
GETTING IMAGE INFORMATION	24
LOCKING AND UNLOCKING TRACKS	24
CAPSLIB FDC EMULATOR AND MFM FORMATTING USAGE	26
FDC EMULATOR USAGE	26

FORMATTING DATA TO MFM _____	30
API REFERENCE _____	33
UPDATING CODE FROM THE OLD API _____	33
LEGACY TYPE NAME MAPPING _____	34
DEFINITIONS AND CONSTANTS _____	35
FILE NAME DEFINITIONS _____	35
BUILD IMPORT/EXPORT DEFINITIONS _____	36
LOCKING FLAGS _____	37
ENUMERATIONS _____	40
CAPSGETINFO.INFTYPE _____	40
RECOGNIZED IMAGE TYPES _____	41
IMAGE ERROR STATUS _____	42
FDC EMULATOR INFORMATION _____	44
FDC RESET STATES _____	45
FUNCTIONS _____	46
CAPSINIT _____	46
CAPSEXIT _____	47
CAPSADDIMAGE _____	48
CAPSREIMAGE _____	49
CAPSLOCKIMAGE _____	50
CAPSLOCKIMAGEMEMORY _____	51
CAPSUNLOCKIMAGE _____	52
CAPSLOADIMAGE _____	53
CAPSGETIMAGEINFO _____	54
CAPSLOCKTRACK _____	55
CAPSUNLOCKTRACK _____	57
CAPSUNLOCKALLTRACKS _____	58
CAPSGETPLATFORMNAME _____	59
CAPSGETVERSIONINFO _____	60
CAPSFDCGETINFO _____	61
CAPSFDCINIT _____	62
CAPSFDCRESET _____	63
CAPSFDCRESETSTATE _____	64
CAPSFDCEMULATE _____	65
CAPSFDCREAD _____	66
CAPSFDCWRITE _____	67
CAPSFDCINVALIDATETRACK _____	68
CAPSFORMATDATATOMFM _____	69
CAPSGETINFO _____	70
CAPSSETREVOLUTION _____	71
CAPSGETIMAGETYPE _____	72
CAPSGETIMAGETYPEMEMORY _____	73
CAPSGETDEBUGREQUEST _____	74
STRUCTURES AND TYPES _____	75
CAPSDATETIMEEXT _____	75
CAPSVERSIONINFO _____	76
LIBRARY VERSION DEFINITIONS _____	77
CAPSIMAGEINFO _____	78
CAPSIMAGEINFO.TYPE _____	79
CAPSIMAGEINFO.PLATFORM _____	80
CAPSTRACKINFO _____	81

CAPSTRACKINFOT1	83
CAPSTRACKINFOT2	85
CAPSTRACKINFO.TYPE	87
PUBLIC SIZE DEFINITIONS	88
CAPSECTORINFO	89
CAPSECTORINFO.BITCELL TYPE	91
CAPSECTORINFO.ENCODER TYPE	92
CAPSECTORINFO.GAP SIZE MODE	93
CAPSDATAINFO	94
CAPSDATAINFO.DATA TYPE	95
CAPSREVOLUTIONINFO	96
CAPSDRIVE	97
DRIVE DEFAULTS	100
DISK AND DRIVE ATTRIBUTE FLAGS	101
CAPSFDCHOOK	101
CAPSFDC	102
FDC MODELS	108
FDC OUTPUT LINE FLAGS	109
FDC STATUS FLAGS AND MASKS	110
FDC END REQUEST FLAGS	111
FDC AM INFO FLAGS	112
FDC RUN MODES	113
FDC DATA MODES	114
CAPSFMTBLOCK	115
FORMAT BLOCK TYPES	117
CAPSFMTTRACK	118

Copyright Notice

This document is copyright © 2002-2026 KryoFlux Products & Services Limited

<https://www.kryoflux.com>

For licensing and policy please refer to the enclosed document with the library revisions.

Some acronyms, names and expressions may be the trademark of various companies; all such material is acknowledged.

Please visit our forum:

<https://forum.kryoflux.com>

Revision History

5.20	2026.6.7	Major update from 1.02 to 5.2, Aki Sivula, István Fábián
1.02	2004.2.22	Minor corrections, Christian Sauer Changed: CAPSRemImage, CAPSGetPlatformName, AmigaOS Specific Drive Properties, István Fábián
1.01	2004.2.20	Proof read, Kieron Wilkinson
1.0	2004.2.17	Initial Release

Purpose

The **SPS Decoder Library** (aka *CAPS Library*) allows various programs to access *Interchangeable Preservation Format – IPF* files in a uniform way.

IPF files can represent various types of media in a common “virtual” format, regardless of the physical form originally used. The files are currently used to provide authentic representation of floppy disk images and ROM contents, though future uses may include more possibilities, such as information on dongle (aka hardware key) protections, tape images and so on.

Software Preservation Society (SPS) and IPF format

This library is part of our (*Software Preservation Society*) software preservation work; it helps researchers, developers and users to study and experience our history of computer games in their original format, intact - yet available on modern, non-legacy platforms represented by authentic, original data.

Without an open-source release, using preserved data in any format is not future proof, portable or usable for media shifting. Access to IPF file contents and understanding its data might get lost over time.

We are committed to release the library sources to make sure end users as well as libraries, archives and museums can make use of it. If you appreciate what we have been doing and giving and what we intend to do, please encourage us to carry on.

Please visit our forum:

<https://forum.kryoflux.com>

SPS is a privately funded association of art collectors and computer enthusiasts striving for the preservation of computer art, namely computer games.

Art is an important cultural asset. Thousands of museums and archives all over the world preserve and restore pictures, books, movies and audio recordings and information in general for generations to come. To accomplish their assignment, national libraries are backed by law which, varying from country to country, forces production companies to deliver copies of publications, books, audio recordings and movies to the archives for long term preservation. It seems that as of today, nobody has ever thought of or actively cared about the true, unmodified and verified preservation of computer games. Without any action taken, time will run out, very quickly.

Unlike games from the 1970s (delivered on solid state ROM-modules) and games from and after the mid-1990s (delivered on optical media like CD-ROMs and DVDs which are supposed to last for decades), computer games from the 1980s and early 1990s were delivered on magnetic media like tapes or floppy disks and are now at the brink of extinction.

From a preservation point of view, tapes and floppy disks are a nightmare for several reasons:

1. Tapes and floppy disks constantly degrade, in two ways. First is the physical degradation of the orientation of the metal particles which form the magnetic field and store the data. This process is slow and given the fact that the data is encoded digitally, it may be too late to do anything when reading errors occur. Reading errors happen when it has become difficult to decide if a particular bit is 0 or 1. Preservation should occur before it becomes a gamble to get a good read.

2. Second is the chemical degradation. The metal particles bound to the plastic platter of a floppy disk, or the surface of a tape can come off the surface. In fact, in most cases the bonding will simply fall apart after years of temperature changes, moisture and other issues of improper storage. Record companies struggle with this problem when remastering old recordings and have developed a process called baking where the original master tape is put in an oven to rebind the coating to the transport material. After baking, playback is a one try only process because the media will fall apart after passing the playback head of the machine. While similar recording as the original is sufficient for analogue material, even a single misinterpreted bit in the digital world means instant failure.

3. While no user can press industry standard vinyl recordings, CDs or DVDs at home (recordable media can be spotted by simply looking at it), tapes and floppies can actually be written and modified with consumer-grade equipment. It takes a lot of expertise to distinguish a professionally replicated medium from a home-made copy. Even if a disk was produced by a commercial replicator, it does not necessarily mean that the disk is still authentic and appropriate for preservation. Apart from a game possibly being copied over the original (that we have seen many times as a "fix" to a broken disk), many games themselves persist some kind of save state or high score, thus changing or erasing data that was available on the disk in the first place. As soon as the disk has been modified in any way, the authenticity of that copy is put into serious doubt.

SPS has successfully mastered these challenges and developed software and hardware technology to deal with the problems arising during the preservation process. Founded by computer expert and preservation pioneer István Fábíán in 2001 as **CAPS** (the *Classic Amiga Preservation Society*), our highly specialised team has more than two decades of field experience. SPS members have not only been involved in playing games on the machines which are regarded retro today, but were programmers and designers also responsible for some of the games and programs available on these platforms.

While our original disk imaging tools (working on e.g. a standard Amiga 1200 with a compact flash adapter) are still good and easy to use, we are now using our own completely self-contained floppy disk controller "**KryoFlux**" developed by SPS that works with any modern PC via a USB connection. This does not only speed up imaging of disks but also enables physical media restoration of any title preserved so far.

Preservation at SPS usually is a two-step process. Contributors from all over the world can help imaging disks with our unique technology. At SPS, our experts then use the Softpres Analyser to investigate the disk structure and create an IPF file. Scripting allows a flexible, even game-specific way of representing data when read by a tool, or when rewritten to disk. Often rather different methods are required to represent various disk formats or copy protection methods when intended to be read by e.g. an emulator or to be written back when restoring an original disk.

Due to the high quality of the preservation technology, IPFs have become the de facto standard demanded by Amiga users when looking for unmodified images true to the original.

While disks themselves are the problem that needs to be addressed quickly while they are still readable, SPS is also striving for complete archival of manuals and boxes in the form of physical products as well as digital scans.

As of today, SPS has digitally archived over 10,000 games produced for the Commodore Amiga, C64, Atari ST and other classic computing platforms. This is a race against time to protect the software gems of yesterday from fading into oblivion.

User notes

There is no GUI or other options accessible to the users of the library; if a program can make use of the library, it will make automatic use of the features available.

The latest library version is always to be found at the download section of the **KryoFlux** site:

<https://www.kryoflux.com>

Should you have problems with the usage of some software supplied in **IPF** format, such as a program capable of using **IPF** files, yet the image used is not recognized by the application, we advise downloading and installing the latest version of the library for the specific platform.

The library is naturally not involved in the working or the settings of applications using it, just like a generic compression library is not involved with the applications using it.

Considering the above: please note, that we cannot answer support questions regarding applications not issued by **KryoFlux** or **SPS**.

Please consult with the authors and companies responsible for such products directly.

Classic computing platforms have many dedicated forums where members are happy to help with problems of emulation or even using a real machine.

Developer Manual

Please note, readers are expected to be familiar with programming languages, programming techniques and other involved material.

This part of the manual is not intended for users, casual readers or beginners.

Background

The access library has been designed to act like a virtual device for removable media.

Think of it as your card reader with cards, a CD player with CDs or any other storage device using removable media.

The library itself is the device capable of accessing the media and the **IPF** files are the removable media.

The internal format of **IPF** files can and does change according to needs, but the library completely hides the complexities involved with interpreting the data in its original format as used by the files.

Therefore, if you plan to use the files without the library you are on your own:

- There will be no support whatsoever
- Internal changes will affect your program
- There is no legitimate usage of the files that is not accessible through the library functions
- The built-in library implementation is efficient, highly optimised and already very fast even on older CPU architectures

If a library feature is not available, and you think it should be, please post on our forums explaining your use-case.

Interface

The library interface is simple by design so it is easy to implement **IPF** functionality in any application wishing to do so.

Please note that implementing functionality that takes advantage of data supplied by the library may not be a trivial task, especially when dealing with floppy disk images.

Reading Disk Images

Tracks read from **IPF** disk images are returned as raw data read by the floppy disk controller. The data is pre-processed in a way so there is no need to emulate a complete FDC compensating for bit cell width changes, jitter, data window, etc. - however the data must be dealt with the same way the target system FDC interprets the pre-processed raw bit cell data.

Emulation

You may want to implement aligning the data as real FDCs do.

Basically, synced data output from an FDC is always aligned to byte or word boundaries - or other alignment normally associated with DMA boundaries.

During the alignment of data, the FDC may strip a few bits once the proper syncing is detected and the internal data shifter is re-aligned.

Luckily, so far only two games are known to take advantage of this on Amiga, both using the Ordilogic disk format: “Agony” and “Unreal”.

Amiga

The Amiga is unique in a way that it does not have a hardware or firmware-based data decoder, pretty much everything should be done by software on the host machine.

The raw bit cell data and timing supplied by the library can be used to properly feed programs with disk data as expected by them when reading from the real media.

Some programs may require proper implementation of the sync re-aligning/stripping functionality of the FDC shifter.

Obviously, the emulator should be able to emulate other functionality by translating various hardware register changes into FDC states when necessary, like keeping track of the head position of the floppy disk drives and so on.

Generic Controllers

Generic FDCs as used by loads of computers and other systems out there normally implement decoding of the pre-processed data on their own using hardware or firmware.

Hardware Based Decoding

When emulating hardware-based controllers the decoding algorithm used by the hardware should be implemented, such as MFM decoding in addition to translating FDC commands to internal FDC states, like head position, motor spin and others.

Firmware Based Decoding

Firmware based controllers can be emulated by intercepting the firmware commands and translating them into something more suitable to the emulator and decoding the raw bit cell data as the firmware would. However, when the firmware is available and the device running the firmware can be emulated, it is recommended to emulate the firmware and feed the hardware registers of the device with raw bit cell data available from the library. That way every patch and quirk that is due to firmware can be perfectly represented, as things like that are quite often hard to replicate properly under all conditions and disk protection systems may take advantage of them.

Virtual Drive

When using raw bit cell data supplied by the library for other purposes than emulation, you can get away with much simpler implementation of the needed functionality.

Basically, your application will choose which track to read, get the library to read it and interpret it any way suitable for the program.

The application is of course still expected to understand how to develop the data into something useful for itself.

Applications like browsing an **IPF** image with some expected formats certainly don't need any complex functionality implemented regarding the use of the disk content.

Just like emulation you may want to use the bit cell data as is or decoded by some simple functionality like the generic MFM decoding used by most FDCs available.

Naturally there will be no need to emulate drives and their internal working and states in these cases, unless you really want to for some other reason.

Writing Disk Images

The library by design implements a “read-only” device.

IPF is a preservation format, and by using **IPF** files the user expects the data to be the original, unchanged representation of the original media content archived. By supplying direct write functionality, the preservation aspect of the format – as is suggested even by the naming of the format – would be circumvented.

Application authors wishing to make changes to the media represented by **IPF** files – such as saving back to “disk” - are advised to use incremental difference/delta files that can be accessed by the application at the same time as opening the original **IPF** file.

If a delta file is not present that of course means all tracks should be read from the **IPF** file and the “disk” is write-protected

Each data track that is present in the difference file should be accessed from that file, while tracks not present should be accessed from the **IPF** file.

When “writing” of the media occurs the diff file should be built, changed or appended as needed by the specific platform represented by the media.

When a disk is set to be write enabled by the user and the delta file is not present it should be created and only upon successful creation of a read/write delta file should the disk being presented write enabled to the user.

It is a good idea to keep a quick index of all the tracks up to the maximum tracks supported by the system hardware at the beginning of the file and changing data in the quick index whenever needed.

It may be also helpful to make delta tracks that are able to hold the largest possible written track by the system for simplicity.

All “changes” made to the “disk” can be undone by simply deleting the delta file, and the authenticity of the **IPF** content is maintained in a clean way.

Naturally disks represented by **IPF** files should be presented as read-only disks, unless delta writing functionality is implemented and/or understood by the application. According to this, when a “disk” does not have a delta file it is always write protected, otherwise it is write enabled if the user sets the delta file to be write enabled by any means allowed by the application, and write protected if the delta file cannot be written by the application. If the delta file is present, the application can read it but cannot write for whatever reason the media should be presented as write protected.

Using the Track Data

The track data returned by the library is divided into a data area representing the bit cells on the disk surface and a density map representing the cell widths in packs of 8 bits, i.e. every group of 8-bit cells (a complete byte) has one density value assigned to it. The index of a density value in the density map buffer is the same as the index that should be used to retrieve the data byte from the cell buffer.

It is worth noting as an optimisation that sync values that lead the first block on a track are always aligned to byte boundaries, in other words normally byte comparisons can be used by most applications searching for marks (aka syncs) on a track.

Most MFM recorded tracks always have all the marks starting on a byte boundary, as gaps between blocks are byte sized.

Using this knowledge is not recommended practice for emulation or with more precise reading, where protection data may consist of blocks slightly shifted to each other on non-byte boundaries. The first sync of such a protection data still can be found by only using byte comparisons – since the library will align the very first bit of valid, non gap data to start on a byte boundary -, however subsequent marks will not be found this way. Protections depending on FDC shifter re-alignment on sync values will give valid, but different sync values back depending on the position used to read the track data stream, and syncs cannot be found on such tracks using byte comparisons.

Disk Rotation

Keep in mind that disk tracks are normally circular entities. Thus, once the last data bit of a track is read and the read operation is not finished by the application, the disk data should be read again from the starting position previously used on the buffer (usually index 0, but this may vary depending on the detail of emulation).

Disk Index

The only absolute position that can be used in mapping the geometry of a disk track is the position of the index hole on the track.

The data buffer returned by the library has the index hole at exactly before index/offset 0 of the buffer. That is, a disk is “rotated” a complete revolution if the buffer contents are read sequentially and the highest index in the buffer plus one is reached; the buffer index/offset should be changed to 0 and a disk index signal issued by the emulated hardware.

Alternatively, a much more precise way of emulation is timing the reading of the track for all data to be read (a complete track data buffer) between each disk index hole signalled by the emulated hardware.

If developing a non-emulation related application, the only thing worth keeping in mind is that buffer position 0 holds the start of the data track. In fact, it can be more convenient to align the track data to the start of the buffer through a locking flag provided for this purpose, making the track always “index-synced” regardless of the real geometry and timing used on the track.

Track Data Alignment

Normally buffers can be of any arbitrary length, however a locking flag is provided for aligning the data buffer size to be of even length – that is, a multiple of machine words (16 bits).

Recent revisions of the library also support tracks that have lengths defined in bit cells – the tracks are stored using the expected alignment, but emulating rotation should consider the number of valid bit cells.

The data area is a bit stream that always starts at the very first byte of the buffer (offset/index 0) - and the leftmost bit of that (bit 7).

Bits are traditionally numbered from right to left, i.e. bit 7 represents the first (leftmost) bit of a byte in the data buffer, and bit 0 is the rightmost bit of the bit stream represented by the buffer. The next byte of the stream again should read from the leftmost to the rightmost bit.

A simple piece of C code that reads one bit from a bit stream like the one explained:

```
bytebuffer[bitpos>>3] >> ((bitpos&7) ^7) & 1
```

Cell Density Map

The density map represents the cell width for each pack of 8 bits, i.e. every group of 8 data bit cells on the disk (a complete byte) has one density value assigned to it. The index of a density value in the density map buffer is the same as the index that should be used to retrieve the data byte from the cell buffer.

The density value supplied by the library is relative to the complete cell time of a track. Each step represents a 1/1000th difference from the default cell density used by normal speed cell groups of the whole track. A value of 1000 represents a cell group in normal - 100% - width; a higher value is a wider cell group (slower/takes more time to read); a lower value is a narrower cell group (faster/takes less time to read).

It is normal to have variable cell density within the same track as a protection measure.

The normal cell density timing of each bit on a track can be calculated by dividing the amount of time available for one complete revolution of the disk by the number of bits/cells present on the track. The number of cells on a track is always the track data length for one revolution multiplied by 8.

For example, if you have a data buffer of 12500 bytes in a 300 rpm drive each bit cell should take exactly 2us (microseconds) to read.

A 300 rpm drive has exactly 300 **Revolutions Per Minute (RPM)**.

One second has $300 / 60 = 5$ revolutions. The complete time of 1 revolution is therefore $1/5$ or 0.2 second. 12500 bytes is $12500 * 8 = 100\ 000$ bits. $0.2s / 100\ 000 = 0.000002$ s, or 2us.

Recent library versions also support a fraction based representation of the density timings.

Most applications should not be concerned with cell density maps and therefore should not specify any of the related locking flags for the tracks to save on memory usage.

Multi-revolution Tracks

There are protections relying on “random” data (weak bits) read from the same disk area each revolution.

The library provides a convenient way of dealing with this: multi-revolution tracks.

Such tracks have more than one revolution of the data stream generated when the track is decoded, flakey (aka weak) data bits are generated with a simple pseudo random generator algorithm at the correct positions producing different random values for each disk revolution.

Once reading crosses the track size boundary – that is the disk index – the data buffer pointer should be read from the next buffer pointer of the track structure returned by the library starting at index/offset 0 as usual. The number of valid track pointer entries in the array is given by the *trackcnt* variable of the structure.

Note, that the cell density map is still generated for one revolution only regardless of the number of revolutions generated for data.

Normal track data not containing random elements is always decoded as one revolution of stream data. Unless there is an error during decoding, the first pointer in the related array - *trackdata[0]* and the size - *tracksize[0]* - should always be valid.

Applications not interested in multiple revolutions should not use the *tracklen* data from the track structure, as it holds the complete size of the allocated buffer area for all the track revolutions decoded, hence giving an incorrect value for just one track. Use *trackdata[0]* and *tracksize[0]* instead.

An alternative to using multi-revolution tracks is using the track locking features to update the weak bit areas of a track after each disk revolution processed by the emulation. In this case, there is only a single track decoded by the library, but the track contents keep on changing by subsequent locking calls.

Drive Properties

Cylinders are often referred as tracks when discussing drive mechanics as the head positions are ignored - upper and lower head on a double-sided drive that can access both sides of a disk. Logically a track is accessed using the cylinder and the head position of the drive, even though the drive may only support one side of the disk.

Physical drive limitations should never be derived from the information contained in the **IPF** files, like setting the drive “hard stops” from *mincylinder*/*maxcylinder*. These values help to identify the data available in the IPF image, not the physical characteristics of the drive used to read the image.

Always use the drive parameters associated with a specific drive type.

One example is that an image may contain only 80 cylinders, but the program tries to write over that limit. If the drive limits were taken from the **IPF** file, the drive would stop at cylinder 79 and overwrite the data there, instead of writing to cylinder 80.

A floppy disk drive (**FDD**) normally has a “hard stop” on cylinder 0 – commonly referred as “track0” - as the minimum allowed cylinder. If a drive attempts to step out

from cylinder 0 the head does not move, however other signals may get changed as if the step happened.

Cylinders are numbered from the outside towards the inner rings of the disk: the outermost ring is cylinder 0.

A **FDD** normally has a *hard stop* on the maximum allowed cylinder as well. It is safe that a program always pretends there is a *hard stop* there.

The maximum cylinder a drive can access depends on the specifications, model etc of the drive.

A drive that supports 80 cylinders can normally access 82 cylinders; some models can access 84 cylinders.

A drive that supports 40 cylinders can normally access 40 or 42 cylinders.

For compatibility it is best to allow the access of the maximum possible cylinder when emulating a drive.

Cylinders are numbered from 0, so an 84-cylinder drive can access cylinders in the range of 0...83.

Some drives – specifically some Commodore drives – use an 80-track drive mechanism for 40 track operations. In that case only every second track is used. The in-between steps are referred to as “*half-tracks*”. They may hold protection or other data.

Using the library

For most projects, *PLEASE* use the library as a dynamic library and do not statically link to it. This will help end-users to upgrade the library independently of the update cycle of its host program. If a feature is not available, and you think it should be, please post on our forums including explaining your use-case. We are active users of the library as well; our own tools, like DTC do not use to library sources directly either - they load the library and call the functions as needed.

Adding the library to projects written in any language should be possible if the necessary data types and pointers or references to buffers are supported in some way and the language can call C-style library functions.

The structures and their packing/alignment should not be altered, and the same packing alignment functionality must be used when converting the headers to be used for other languages, otherwise access violations will happen as the library will use different structure offsets from the ones used by the application. Languages able to link to C libraries have alignment settings – or use completely packed structures by default when calling C libraries.

As an example, using the library from a host code written in C++ is as follows.

1, Setup the project include directories as needed by your project

```
// external definitions
#include "CommonTypes.h" // Core/CommonTypes.h

// IPF library public definitions
#define CAPS_USER
#include "CapsLibAll.h" // LibIPF/CapsLibAll.h
```

Unlike previous library codebases, legacy, custom datatypes are no longer used - you should include `<stdint>` before using any of the library headers.

2, Use the library functions according to their documentation and source code.

Compiling the library

The compiler used should be capable of compiling for C++17 or later. It is expected that the Standard Library implementation is also compliant.

While the code might compile to earlier C++ standards and libraries, this is not guaranteed, verified or getting fixed.

Compiling for Windows

The library uses a Visual Studio solution file and is intended to be built with a Visual Studio C++ build environment, such as Visual Studio 2017 or later, or Visual Studio Build Tools 2017 or later.

A C++17-capable compiler, matching Standard Library, runtime libraries, Platform Toolset and Windows SDK are required for compilation.

Please note that the project is configured to build binaries compatible with older Windows releases.

This may become more restricted in future official library releases.

In theory (not validated), the compiled library should be compatible with:

- Windows XP or later for 32-bit builds
- Windows 8 or later for 64-bit builds

The project currently uses the following toolsets:

- Win32 builds: Platform Toolset v141_xp
- x64 builds: Platform Toolset v141

Older Windows SDKs may also be required depending on the selected build configuration. If the required SDK is not installed, you can either install the matching SDK or retarget the project to an installed SDK version.

Retargeting may reduce compatibility with older Windows releases.

For example, when building with a newer Windows SDK, the SDK version can be overridden from the command line:

```
msbuild .\SPStudio_Dev.sln /p:Configuration=Release  
/p:Platform=x64 /p:WindowsTargetPlatformVersion=10.0.26100.0
```

To build using Visual Studio

- 1, Open SPStudio_Dev.sln
- 2, Optionally retarget the Platform Toolset and Windows SDK, keeping the compatibility notes above in mind
- 3, Select the desired configuration:
 - Debug or Release
 - Win32 or x64
- 4, Build the solution, or use Batch Build for multiple configurations

To build from the command line

Open a Developer PowerShell or Developer Command Prompt for Visual Studio and run for example:

```
msbuild .\SPStudio_Dev.sln /p:Configuration=Release  
/p:Platform=x64
```

If the build fails because an older Windows SDK is missing:

- either install the required SDK or
- specify an installed SDK version with `WindowsTargetPlatformVersion`

Compiling for macOS and Linux

1, Go to the folder with the sources, then

```
mkdir build  
cd build  
cmake ../CMakeLists.txt -B .  
make
```

Linux

If you want to install, then

```
make install
```

If you want to create packages (RPM/DEB)

```
make package
```

macOS

Either use the `.dylib` in your own projects, or copy the file (not the links)

to `/usr/lib` (needs root access).

For universal build use the `CAPSIMAGE_MACOS_UNIVERSAL` option, e.g.

```
cmake ../CMakeLists.txt -B . -D CAPSIMAGE_MACOS_UNIVERSAL=ON
```

Using the API

Process contexts

Different processes can safely use the library at any given time.

Each process must open its own instance of the library and should only use data supplied by the library in its own context.

“Sharing” the library through invalid means – such as passing pointers and function pointers among processes – is discouraged and will not work.

Multi-threading

The library functions should only be called by one thread at a time, but any thread can access the API within the same process context.

While existing data is accessed in a safe way, manipulating some part of the data may involve functionality that is not safe when another call is in progress, such as adding new image containers to existing ones.

Therefore, for complete safety it is recommended that the caller implement a thread-locking mechanism if multiple threads can call the API within the same process.

Pointers and Data Persistence

Pointers are only valid within the process context using the library; sharing the pointers with other processes is an access violation. If you really want to share data between processes, copy the results to a shared memory file, but it’s much better to just open and use the library whenever needed.

If more than one process will use the library, simply open it with each one of them.

Pointers supplied by the API can and do change, but it is guaranteed that between each API call involved with the creation or destruction of the data, the data remains unchanged and the pointers are valid.

Data is not “moved around” by the library, but it may be destroyed by specific API calls.

If you possibly use the same data – such as track descriptors – by different threads at the same time, and you explicitly invalidate the data by an API call, you should not attempt to use the data by any of the other threads after deletion. One way to avoid any such problems is to always call the necessary API functions to retrieve the base data descriptors supplied by the library and obtain your pointers from the descriptors returned. If the data is already available, the library caches it and the function practically returns within a few cycles, if the data is not available yet or already destroyed it gets re-built and cached until it is destroyed through an API call.

Since pointers may change depending on the internal state of the library, it is a good idea to retrieve the referenced pointers and structures after each API call and not to try statically “cache” them. Although this should be trivial, we recite here just for safety.

Do not expect using the same image and the same tracks to twice return the same pointers as data could be created or destroyed in any way.

It is not recommended that data returned by the library calls be altered. If you plan to change the buffer contents just copy them into your private data buffers.

File Sharing

The library allows shared access to the same file for any thread or process when using **IPF** files. However, the sharing mechanisms and permissions involved may vary among the various operating systems and user privileges.

You should normally be able to open the same image file several times, but it is a good idea to check the error codes returned.

The library will fail to function correctly if the contents of an open image file changes while it is being used.

Error Handling

Unless otherwise stated, after successful completion of any API call *imageOk* is returned. Any other value means there was a problem during execution.

If any function returns with an error, the program is expected to handle that.

Functions that fill a typed output structure first identify the requested structure type, then clear the identified structure before filling it.

If type identification, validation, or later processing returns an error, do not expect the output structure to be cleared, complete, or meaningful unless a function explicitly documents valid results even in this case.

Freeing Memory

Freeing memory returned through API calls should not be attempted by the application using the library. It will lead to access violations or other malfunctions.

Memory is only to be freed using the appropriate calls described for unlocking or destroying data.

Programming Tasks

Opening and Closing the Library

The library must be initialised with the *CAPSInit* function before any other function is called.

The library must be closed with *CAPSExit* after all activity is complete.

All pointers and values previously returned by any of the functions are invalid after *CAPSExit*.

Creating and Destroying Image Containers

Each **IPF** image in active use is accessed through an image device or “container”. Rather than accessing the files directly, each container acts as a virtual device for the file assigned to it.

One container always holds one file at a time. You should create as many containers as is needed for the files open at the same time.

Images are assigned or “locked” into the containers – devices – during their use and can be ejected or “unlocked” after use.

The containers do not share their internal state with each other; therefore, the same **IPF** file locked into different containers can have different states and certainly have completely different pointers.

There is no need to destroy a container after its use as it takes minimal memory, but if an image no longer needs to be locked, unlocking it will free the memory used by the internal caches of the container as well as remove access to the file itself allowing, e.g. deletion of the file. Of course, an empty container should be re-cycled by the application to conserve resources used by the library.

Image containers are created with *CAPSAddImage* and destroyed with *CAPSRemImage*.

Before a container can be used it must be created with *CAPSAddImage*.

Once a container is no longer in use it must be destroyed with *CAPSRemImage*. Destroying a container unlocks its image.

CAPSExit destroys all the valid containers.

All pointers and values previously returned by any of the functions about a container or its content are invalid after *CAPSRemImage*.

Locking and Unlocking Images

Images are assigned or “locked” into the containers – devices – during their use and can be ejected or “unlocked” after use.

An **IPF** image must be locked to be accessible by the library using the *CAPSLockImage* or *CAPSLockImageMemory* functions.

Locking an image from file using *CAPSLockImage* does not load or decode its contents or allocate memory, all the relevant information about contents and how to access them is cached when locking occurs. The file itself is locked into a read-only, shared state to allow subsequent reads to be performed.

Locking from memory is done by calling *CAPSLockImageMemory*, and performs the same steps with *CAPSLockImage* but of course file locking is not involved. The whole **IPF** file must be accessible from the supplied memory buffer at the time of calling *CAPSLockImageMemory* otherwise access violations can occur.

When the file is no longer in use *CAPSUnlockImage* should be called.

Destroying a container unlocks the image previously locked.

Locking another image into the same container does an implicit *CAPSUnlockImage* first.

All pointers and values previously returned by any of the functions about an image and its contents – like tracks - are invalid after *CAPSUnlockImage*.

Getting Image Information

It is possible to obtain the information stored about the **IPF** file contents through the API using the *CAPSGetImageInfo* call.

The information can be used to restrict accessing only to valid areas of a disk image, display information about the contents, and so on.

The *platform* array contains up to CAPS_MAXPLATFORM platform identifiers.

The *CAPSGetPlatformName* function can be used to retrieve the symbolic name assigned to a platform ID value, like “Amiga” or “Atari ST” etc.

Locking and Unlocking Tracks

Disk images are divided into tracks.

Each track can be decoded from its **IPF** format into raw bit cell data by calling *CAPSLockTrack* and destroyed after use with *CAPSUnlockTrack*.

Once the track data is decoded with *CAPSLockTrack* it is cached until a subsequent *CAPSUnlockTrack* is called for the same track or it is indirectly invalidated through unlocking or replacing the image or destroying its container.

If you make any subsequent use of the track contents using the original buffers supplied by the lock, it is advised not to unlock the track to prevent performance hits. If memory is at a premium unlocking the unneeded tracks can be used to free the buffer areas used.

The locking of tracks is done using some attributes – flags – supplied with the call. Some flags can result in different data being returned by the call. As long as the track is not unlocked directly or indirectly, each subsequent lock on the same track returns the very same data generated the first time the track was originally locked, regardless

of the flags later supplied. Therefore, if changing the locking attributes of a track is desired, it must be unlocked first.

Locked tracks containing weak bits however can be updated in their weak bit areas with subsequent lock calls.

There is no reference counting on track locking, the programmer is free to lock or unlock its contents at any time.

Locking an already locked track returns the previously cached state of the track, unlocking a track always free the resources – memory - associated with the track contents.

It is possible to lock all the tracks of an image with just one call, *CAPSLoadImage*. If memory is of no concern but performance is – like real time emulation -, this call is recommended for use, e.g. when changing disks for emulation. Note, that previously locked tracks are not unlocked first, therefore they reflect the locking attributes used when they had been locked for the first time. Remember locking already locked tracks is practically free costing no execution time; performance hits due to decoding of the **IPF** contents are not an issue that way.

For convenience it is possible to unlock all tracks by one call, *CAPSUnlockAllTracks*.

All pointers and values previously returned by any of the functions about a track and its contents are invalid after *CAPSUnlockTrack* or calls of the same functionality.

CAPSLib FDC Emulator and MFM Formatting Usage

This section gives practical examples for using the CAPSLib FDC emulator and MFM encoder parts of the library.

These APIs are not required for ordinary IPF file reading; the image and track APIs described above are enough for that.

The functions mentioned here are convenience and reference functionality for emulator and tool authors, who can use the supplied WD-style emulator instead of implementing similar controller behaviour themselves. Some applications use the supplied emulator directly.

The examples focus on host-application responsibilities, data ownership, and call sequencing. They are not a replacement for the API reference; use the reference sections for complete function, structure, constant, and error details.

FDC Emulator Usage

The FDC emulator exposes a WD1772-style controller through a `CapsFdc` state block and one or more `CapsDrive` drive blocks. The host application owns these structures and supplies disk data when the emulator asks for a track.

Hardware register access should normally go through `CAPSFdcRead` and `CAPSFdcWrite` so the same side effects occur as they would on an emulated machine.

1. Prepare the CAPSLib image and the FDC workspace

Create and load the image before the FDC needs track data. The image container ID is later used by the track-change callback.

```
CAPSInit();
int32_t imageId = CAPSAddImage();
CAPSLockImage(imageId, imageName);
CAPSLoadImage(imageId, DI_LOCK_DENALT | DI_LOCK_DENVAR |
DI_LOCK_UPDATEFD);
```

Allocate or embed a `CapsFdc` structure and an array of `CapsDrive` structures. Clear them before use. Set the fields that `CAPSFdcInit` preserves: structure sizes, FDC model, FDC clock frequency, attached drive pointer, drive count, active drive count, and optional host user data.

```
CapsFdc fdc = {};
CapsDrive drives[1] = {};

fdc.type = sizeof(CapsFdc);
fdc.model = cfdcmWD1772;
fdc.clockfrq = 8000000;
fdc.drive = drives;
fdc.drivcnt = 1;
fdc.drivemax = 1;

drives[0].type = sizeof(CapsDrive);
```

CAPSFdcGetInfo can be used before allocation to query `cfdcSize_Fdc` and `cfdcSize_Drive`, or later with a valid `CapsFdc` pointer to inspect command, status, track, sector, and data registers without normal bus-read side effects.

2. Prepare callback handlers

The host must provide IRQ, DRQ, and track-change callback functions for FDC emulation. A track-change callback supplies disk data from an IPF image or from a generated MFM buffer. The callback receives the `CapsFdc` pointer and the drive index that needs track data. Install the callback pointers after `CAPSFdcInit` because initialization clears the callback fields.

The callback uses the selected drive `buftrack` and `bufside` values. For IPF data, request `CapsTrackInfoT1` by setting `type` to 1 and pass `DI_LOCK_TYPE` with the density flags used by the emulator.

```
void __cdecl OnFdcTrackChange(PCAPSFDC pc, uint32_t driveIndex)
{
    PCAPSDRIVE drive = pc->drive + driveIndex;
    CapsTrackInfoT1 trackInfo = {};
    trackInfo.type = 1;

    int32_t result = CAPSLockTrack(&trackInfo, imageId,
        drive->buftrack, drive->bufside,
        DI_LOCK_DENALT | DI_LOCK_DENVAR | DI_LOCK_UPDATEFD |
        DI_LOCK_TYPE);

    if (result != imgeOk)
        return;

    drive->ttype = trackInfo.type;
    drive->trackbuf = trackInfo.trackbuf;
    drive->timebuf = trackInfo.timebuf;
    drive->tracklen = trackInfo.tracklen;
    drive->overlap = trackInfo.overlap;
}
```

Track pointers returned by `CAPSLockTrack` remain owned by the library. They are valid only while the corresponding track and image state remains locked. If the host invalidates or reloads image data, refresh the drive track pointers through the callback path.

3. Initialize, install callbacks, insert media, and reset as needed

Call `CAPSFdcInit` after preserved preset fields are in place. Initialization clears the FDC state block, restores the documented presets, and leaves drives empty and write-protected. After a successful call, set all three callbacks before running the emulator. Mark a disk inserted by setting `CAPSDRIVE_DA_IN` and clear `CAPSDRIVE_DA_WP` if the emulated disk should be writable. Use the documented drive defaults unless the host has platform-specific values.

```
if (CAPSFdcInit(&fdc) != imgeOk)
```

```

return;

fdc.cbirq = OnFdcIrq;
fdc.cbdrq = OnFdcDrq;
fdc.cbtrk = OnFdcTrackChange;

```

All three callback pointers must be valid before calling `CAPSFdcEmulate` or using related FDC paths that can run emulation or request track data. The emulator does not check these callback pointers for null during cycle execution; the host is responsible for fully initialized state before it runs the emulator.

```

drives[0].rpm = CAPSDRIVE_35DD_RPM;
drives[0].maxtrack = CAPSDRIVE_35DD_HST;
drives[0].diskattr = CAPSDRIVE_DA_IN;

```

Use `CAPSFdcReset` for a cold hardware reset. Use `CAPSFdcResetState` with `cfdcrs_Warm` when the emulated machine performs a warm reset and the FDC track and data registers should be preserved.

4. Run the emulation loop

The host advances the FDC by calling `CAPSFdcEmulate` with elapsed FDC clock cycles. Between calls, the emulated CPU accesses FDC registers through `CAPSFdcRead` and `CAPSFdcWrite`. Address lines are masked by `CapsFdc.addressmask`, so addresses 0-3 select status or command, track, sector, and data respectively for the WD1772 model.

```

CAPSFdcWrite(&fdc, 1, trackRegisterValue);
CAPSFdcWrite(&fdc, 2, sectorRegisterValue);
CAPSFdcWrite(&fdc, 0, commandByte);

while (machineIsRunning)
{
    CAPSFdcEmulate(&fdc, elapsedFdcCycles);

    if (fdc.lineout & CAPSFDC_LO_DRQ)
        data = CAPSFdcRead(&fdc, 3);

    if (fdc.lineout & CAPSFDC_LO_INTRQ)
        status = CAPSFdcRead(&fdc, 0);
}

```

Reading the status register clears `INTRQ`. Reading or writing the data register clears `DRQ` when it is set. Writing the command register starts a command when the FDC is not busy; type 4 force-interrupt commands are accepted even while busy.

5. Refresh track data when the host changes disk state

Call `CAPSFdcInvalidateTrack` when the host changes the image, replaces generated track data, changes media state, or otherwise needs the next data access to request the track again. The next read path forces the track-change callback and refreshes `CapsDrive.trackbuf`, `timebuf`, `tracklen`, and `overlap`.

```
CAPSFdcInvalidateTrack(&fdc, 0);
```

6. Shut down cleanly

Before unloading or replacing an image, stop using any drive pointers that came from locked track data. Unlock cached tracks, unlock the image, remove the image container, and call `CAPSExit` when the host is done with the library.

```
CAPSUnlockAllTracks (imageId);  
CAPSUnlockImage (imageId);  
CAPSRemImage (imageId);  
CAPSExit ();
```

Formatting Data to MFM

CAPSFormatDataToMFM converts a logical track description into an MFM byte stream. The host describes the output track with CapsFormatTrack and describes each sector or index block with CapsFormatBlock. The generated buffer can be used directly as a CapsDrive.trackbuf source for FDC emulation.

1. Describe the track and sectors

Clear the track descriptor and the block array before filling them. Set CapsFormatTrack.type to 0 for the currently supported structure revision. Set gap values, output buffer information, starting position, block count, and block pointer. Then fill one cfrmbtData block per sector.

```
CapsFormatTrack formatTrack = {};
CapsFormatBlock blocks[MAX_SECTORS] = {};

formatTrack.type = 0;
formatTrack.gapacnt = 60;
formatTrack.gapavalue = 0x4e;
formatTrack.gapbvalue = 0x4e;
formatTrack.trackbuf = outputBuffer;
formatTrack.tracklen = outputLength;
formatTrack.buflen = outputCapacity;
formatTrack.startpos = 0;
formatTrack.blockcnt = sectorCount;
formatTrack.block = blocks;

for (int sector = 0; sector < sectorCount; ++sector)
{
    CapsFormatBlock *block = &blocks[sector];
    block->gapacnt = 12;
    block->gapavalue = 0x00;
    block->gapbcnt = 22;
    block->gapbvalue = 0x4e;
    block->gapccnt = 12;
    block->gapcvalue = 0x00;
    block->gapdcnt = 40;
    block->gapdvalue = 0x4e;
    block->blocktype = cfrmbtData;
    block->track = cylinder;
    block->side = head;
    block->sector = sector + 1;
    block->sectorlen = sectorSize;
    block->databuf = sectorData + sector * sectorSize;
}
```

If databuf is null, datavalue is used as the repeated data byte for the sector payload. Valid sector lengths are 128, 256, 512, and 1024 bytes.

2. Size the buffer when required

If `trackbuf`, `tracklen`, or `buflen` is zero, `CAPSFormatDataToMFM` does not write output. It validates the descriptors and sets `CapsFormatTrack.bufreq` to the minimum required MFM byte count. Allocate a buffer at least that large, set `trackbuf`, `tracklen`, and `buflen`, then call the function again.

```
formatTrack.trackbuf = nullptr;
formatTrack.tracklen = 0;
formatTrack.buflen = 0;
int32_t sizing = CAPSFormatDataToMFM(&formatTrack,
DI_LOCK_TYPE);
if (sizing != imgeOk)
    return sizing;

uint32_t requiredBytes = formatTrack.bufreq;
```

3. Convert the track

Call `CAPSFormatDataToMFM` with `DI_LOCK_TYPE` so the structure revision field is checked. On success, `trackbuf` contains the generated MFM stream, `size` contains the generated byte count, and `startpos` has advanced to the final write position. The conversion wraps around inside `tracklen` when writing the track buffer.

```
int32_t result = CAPSFormatDataToMFM(&formatTrack,
DI_LOCK_TYPE);
if (result != imgeOk)
    return result;

drive->ttype = ctitAuto;
drive->trackbuf = formatTrack.trackbuf;
drive->tracklen = formatTrack.tracklen;
drive->overlap = formatTrack.startpos;
```

4. Handle common errors

`imgeUnsupportedType` means the structure revision requested through `DI_LOCK_TYPE` is not supported. The function writes the highest supported revision back to the first `uint32_t` field.

`imgeBufferShort` means `tracklen` is larger than `buflen` or smaller than the required generated size.

`imgeBadBlockType` means a block has an unsupported `blocktype` value.

`imgeBadBlockSize` means a sector length is not one of the supported WD-style sector sizes.

`imgeBadDataStart` means `startpos` is outside the supplied track buffer.

Except for documented values such as the supported revision written for `imgeUnsupportedType`, do not rely on output fields or generated data after an error return.

5. Use generated MFM with the FDC emulator

A generated MFM buffer can be returned from the same track-change callback used for image-backed tracks. When the requested cylinder and head are valid, build the `CapsFormatTrack` description, convert it, and assign the output fields to the selected `CapsDrive`. The FDC then reads the generated bytes through the normal data path during `CAPSFdcEmulate`.

```
void SupplyGeneratedTrack(PCAPSDRIVE drive, uint8_t *buffer,
uint32_t bufferSize)
{
    CapsFormatTrack formatTrack = {};
    CapsFormatBlock blocks[MAX_SECTORS] = {};

    /* Fill formatTrack and blocks as shown above. */
    formatTrack.trackbuf = buffer;
    formatTrack.tracklen = bufferSize;
    formatTrack.buflen = bufferSize;

    if (CAPSFormatDataToMFM(&formatTrack, DI_LOCK_TYPE) !=
imgOk)
        return;

    drive->ttype = ctitAuto;
    drive->trackbuf = formatTrack.trackbuf;
    drive->tracklen = formatTrack.tracklen;
    drive->overlap = formatTrack.startpos;
}
```

The host owns generated buffers and must keep them alive while the drive points to them.

If the buffer contents are replaced or reallocated, call `CAPSFdcInvalidateTrack` before the FDC continues reading from that drive.

API Reference

The API reference is given in C style for easier reading.

The reader must be familiar with either the C language or similar pseudo code in order to make use of the library functions.

Although most of the library is written in C++, the glue code and the interface are provided in C in order to help those using C compilers or projects or other languages for their application.

The headers should be safe to include by any C++ project as correct linking is selected.

Since the header files have been adjusted to C usage, namespaces and other features available for better typing and data isolation could not be used in the public interface of the library.

Updating Code from the Old API

This library has a long history of development, and started way before standardized integer types were available for C++, thus custom types were defined for using the API.

These legacy types are no longer defined or used by the current library code or API.

The following legacy mappings are provided to make updating code from using the old API types to the current public API types easier.

New code should use the current public API names and types used throughout this reference.

The standard types used by the library now are technically equivalent to the previously used user defined types, however compilers may issue warnings unless the types used by the API calls and data manipulation are changed to use the correct definitions.

Legacy Type Name Mapping

Legacy headers use custom type names.

The current LibIPF headers use fixed-width C/C++ **stdint** types for the public API.

Values

SDWORD	old signed 32-bit value; maps to <code>int32_t</code> in the current public headers.
UDWORD	old unsigned 32-bit value; maps to <code>uint32_t</code> in the current public headers.
UBYTE	old unsigned 8-bit value; maps to <code>uint8_t</code> in the current public headers.
PUBYTE	old pointer to UBYTE; maps to <code>uint8_t *</code> in the current public headers.
PCHAR	old <code>char *</code> string pointer. Current string input parameters use <code>const char *</code> where the API treats the string as read-only.
PVOID	old generic pointer; maps to <code>void *</code> in the current public headers.

Definitions and Constants

Although it might be tempting, never use hard coded values for constants found in the API headers. They may change with future library revisions.

Use the correct data types found in the header files supplied with the library; they resolve to the correct alignment, packing and size requirements of the library internals.

File Name Definitions

Default IPF file name suffix values.

Values

CAPS_FILEEXT	Default IPF file extension without the dot: "ipf". Use this when adding an extension to a file name.
CAPS_FILEPFX	Default IPF file suffix with the dot: ".ipf". Use this when a complete suffix including the dot is needed.

Build Import/Export Definitions

Conditional macros used by CapsLib.h and ComLib.h for public declarations. Define CAPS_USER before including CapsLib.h when building a library client. CapsLib.h then defines LIB_USER; ComLib.h resolves DllSub and DllVar to imported declarations. Without LIB_USER, ComLib.h resolves them to exported declarations. ExtSub and ExtVar add C linkage to public functions and variables.

Values

CAPS_USER	Define before including CapsLib.h when using the library as a client. CapsLib.h then defines LIB_USER before including ComLib.h so public declarations are imported rather than exported.
LIB_USER	Defined by CapsLib.h when CAPS_USER is defined. ComLib.h uses it to select imported public declarations instead of exported declarations; library builds leave it unset and export the same public API symbols.

Locking Flags

Flags used when locking images, tracks and formatter structures. The values are bit flags combined with bitwise OR.

The `DI_LOCK_DEN...` flags request density maps; without a density flag the library does not generate a density map. `DI_LOCK_INDEX` repositions the decoded track buffer as if it were synchronized to index, which is useful for simpler disk logic but changes the timing position from the original recording. `DI_LOCK_ALIGN` pads decoded track data to a 16-bit word boundary; without it, odd byte lengths can be returned exactly as recorded.

Some flags can request generated data, alternate density representation, bit-addressed lengths or weak-data handling. Such generated or repositioned data may be useful for browsers, virtual drives or format conversion, but callers that emulate timing-sensitive protections should choose only the flags they require. Some flags may result in data generated not representing faithfully the data originally recorded on the disk and such data may not be suitable for emulation use – especially for games protected by timing and track geometry properties.

Values

<code>DI_LOCK_INDEX</code>	<p>Re-aligns data as an index-synced recording. Track data is re-aligned in the buffer as if it was index-synced originally, starting at the beginning of the buffer. Normally decoded track data is positioned as it was found on the original disk, starting at any position or distance from the disk index.</p> <p>Setting this flag changes the returned track position and therefore changes timing relative to the original recording.</p>
<code>DI_LOCK_ALIGN</code>	<p>Decodes the track to a word-aligned size. The decoded track data is aligned to a 16-bit word boundary. If the flag is not set, locked tracks may return odd byte lengths when that is how the original disk data was represented.</p> <p>Setting this flag may make the returned track slightly larger than the original decoded size.</p>
<code>DI_LOCK_DENVAR</code>	<p>Generates a cell density map for variable density tracks, such as variable speed protection tracks, like Copylock or Speedlock. Normally only variable density tracks should request this map to save memory and improve application performance. Constant-speed and unformatted tracks can usually be handled by simpler density handling.</p>
<code>DI_LOCK_DENAUTO</code>	<p>Generates a density map for automatically sized, constant-speed cells. The generated map describes constant timing values matching the decoded track size.</p>

DI_LOCK_DENNOISE	Generates a density map for unformatted cells. The generated map describes the timing used for synthetic unformatted or noise data.
DI_LOCK_NOISE	Generates unformatted data. An unformatted track is algorithmically filled with noise-pattern data when this flag is set; otherwise, no track buffer is allocated for an unformatted track.
DI_LOCK_NOISEREV	Generates unformatted data that changes each revolution. The returned buffer can contain multiple revolutions with different generated data, which is useful when the caller needs changing unformatted data across revolutions.
DI_LOCK_MEMREF	<p>Uses the source memory buffer supplied with the function directly in functions where this is supported, such as CAPSLockImageMemory. If set, the library keeps using the caller-supplied buffer until the image is unlocked directly or indirectly, so the caller must keep that buffer valid while the lock is valid.</p> <p>If clear, the library copies the supplied buffer to private storage and the caller may release the original buffer after the call returns. The private data area allocated by the library is automatically freed once the associated lock is deleted.</p>
DI_LOCK_UPDATEFD	Creates flakey or weak data on one revolution and updates it with each lock. This lets the returned weak-bit areas change when a track is locked or updated. Without this flag, tracks containing weak bits may be returned as multiple generated revolutions instead of one changing revolution.
DI_LOCK_TYPE	The first uint32_t in the supplied information structure holds the expected structure type. The library uses it to select the structure layout it fills. If the requested type is unsupported, the library writes back the highest supported type and returns imgeUnsupportedType.
DI_LOCK_DENALT	Generates the alternate density map as fractions. When a density map is generated, this flag converts it from per-cell timing values to the alternate cumulative fractional form.
DI_LOCK_OVLBIT	Reports the overlap position in bits instead of bytes. Use this when bit-level overlap precision is required.
DI_LOCK_TRKBIT	Reports tracklen in bits, and treats the track buffer as bit-sized. Without this flag, tracklen is byte-sized and the decoded track is rounded to a byte or word boundary.
DI_LOCK_NOUPDATE	Track overlap or weak data is initialized but not automatically updated. The lock operation does not advance the next revolution counter and does not update overlap or weak-bit state.

DI_LOCK_SETWSEED	Sets the weak-bit generator seed before locking a type 2 track information block. Set CapsTrackInfoT2.wseed before calling CAPSLockTrack with DI_LOCK_TYPE and DI_LOCK_SETWSEED; the filled CapsTrackInfoT2 returns the resulting wseed value.
------------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Enumerations

CAPSGetInfo.inftype

Information type selector passed to CAPSGetInfo.

Values

cgiiNA	Illegal CAPSGetInfo selector.
cgiiSector	Return CapsSectorInfo for the sector index in infid. Valid sector indices are below CapsTrackInfo.sectorcnt.
cgiiWeak	Return CapsDataInfo for the weak data index in infid. Valid indices are below CapsTrackInfoT2.weakcnt when type 2 track information is available.
cgiiRevolution	Return CapsRevolutionInfo for the selected image and track, describing previous, next, real and maximum revolution selection.

Recognized Image Types

Image type values returned by CAPSGetImageType and CAPSGetImageTypeMemory.

Values

citError	Error preventing image type identification.
citUnknown	Unknown image type.
citIPF	IPF image.
citCTRaw	CT Raw image.
citKFStream	KryoFlux stream files.
citKFStreamCue	KryoFlux stream cue file.
citDraft	Draft image.

Image Error Status

Status values returned by image, track and format API functions. Unless otherwise stated, functions return `imgeOk` after successful completion; any other value indicates a problem during execution.

Values

<code>imgeOk</code>	Function completed successfully.
<code>imgeUnsupported</code>	Requested operation or option is unsupported.
<code>imgeGeneric</code>	Generic error, including invalid pointer arguments where the API checks for them.
<code>imgeOutOfRange</code>	Data supplied to a function call is outside the valid range, such as image id, drive number, track, head or information index, or the referenced data has been invalidated.
<code>imgeReadOnly</code>	Operation attempted on read-only data.
<code>imgeOpen</code>	File or memory image could not be opened.
<code>imgeType</code>	Image type is unknown or not accepted by the image factory.
<code>imgeShort</code>	Input data is shorter than required; when locking from memory this can indicate that the supplied buffer is shorter than the image file data.
<code>imgeTrackHeader</code>	Track header error while decoding an image.
<code>imgeTrackStream</code>	Track stream error while decoding an image.
<code>imgeTrackData</code>	Track data error while decoding an image.
<code>imgeDensityHeader</code>	Density header error while decoding an image.
<code>imgeDensityStream</code>	Density stream error while decoding an image.
<code>imgeDensityData</code>	Density data error while decoding an image.
<code>imgeIncompatible</code>	Image data is incompatible with the requested operation, such as when decoded track or density metadata does not match the requested operation.
<code>imgeUnsupportedType</code>	Requested public structure type or structure size is unsupported. Functions using <code>DI_LOCK_TYPE</code> write back the highest supported type when they can report it.
<code>imgeBadBlockType</code>	<code>CAPSFormatDataToMFM</code> found an unsupported <code>CapsFormatBlock.blocktype</code> .
<code>imgeBadBlockSize</code>	<code>CAPSFormatDataToMFM</code> found an invalid formatter sector size.

imgeBadDataStart	CAPSFormatDataToMFM found CapsFormatTrack.startpos outside the supplied track buffer.
imgeBufferShort	The supplied formatter track buffer is shorter than the required size. CAPSFormatDataToMFM reports the required size in CapsFormatTrack.bufreq.

FDC Emulator Information

Information selectors for CAPSFdcGetInfo.

Values

cfhciNA	Invalid CAPSFdcGetInfo selector.
cfhciSize_Fdc	Return sizeof(CapsFdc). Use this to allocate or validate the FDC state block size expected by the library.
cfhciSize_Drive	Return sizeof(CapsDrive). Use this to allocate or validate the drive state block size expected by the library.
cfhciR_Command	Return the FDC command register.
cfhciR_ST	Return the current FDC status register after applying the active status mask.
cfhciR_Track	Return the FDC track register.
cfhciR_Sector	Return the FDC sector register.
cfhciR_Data	Return the FDC data register.

FDC Reset States

Reset state values passed to CAPSFdcResetState.

Values

cfdcrs_Cold	Cold reset. CAPSFdcReset performs this reset path, and CAPSFdcResetState can request the same state reset.
cfdcrs_Warm	Warm reset. The emulator resets command state while preserving and restoring the track and data registers. Some systems may perform a warm reset on the FDC hardware, e.g. using the M68000 RESET instruction on an Atari ST hardware should be emulated as warm reset for the FDC.

Functions

CAPSInit

The function initialises the library internals and starts CAPS image support. It must be called before any other CAPS image API calls are made.

```
int32_t CAPSInit(void);
```

Parameters

-

Return Values

imgeOk if successful.

Remarks

The program should attempt no further library calls if the function fails, however CAPSExit must be called to free resources that might have been allocated during the call.

Related Functions

CAPSExit

CAPSExit

The function closes the library and frees all resources allocated by it.

```
int32_t CAPSExit(void);
```

Parameters

-

Return Values

imgeOk if successful.

Remarks

The program should not attempt any library calls after calling the function if the function succeeds. All image containers held by the library are destroyed. All pointers previously returned by the library become invalid after this call and must not be used.

Related Functions

CAPSInit, CAPSAddImage, CAPSRemImage

CAPSAddImage

The function allocates an image container to be used by image manipulation functions.

```
int32_t CAPSAddImage(void);
```

Parameters

-

Return Values

A container ID greater or equal to 0 if successful.

Remarks

A negative return value means error, usually resource related. If an error occurs, the function returns no image id and does not allocate a container.

After freeing a container, its ID will be recycled eventually. The program should not assume the ID values returned by the library.

Related Functions

CAPSRemImage, CAPSLockImage, CAPSLockImageMemory

CAPSRemImage

The function frees an image container used by image manipulation functions.

```
int32_t CAPSRemImage(  
int32_t id /* container ID */  
);
```

Parameters

id: [in] the container ID returned by CAPSAddImage.

Return Values

The supplied container ID if successful, otherwise -1.

Remarks

A negative return value means error; usually the ID was invalid.

After freeing a container its ID will be recycled eventually. The program should not assume the ID values returned by the library.

Related Functions

CAPSAddImage, CAPSUnlockImage

CAPSLockImage

The function locks an image into a container device.

```
int32_t CAPSLockImage(  
int32_t id, /* container ID */  
const char *name /* filename */  
);
```

Parameters

id: [in] the container ID returned by CAPSAddImage.

name: [in] the name of the image file to be opened.

Return Values

imgeOk if successful or related imge error code.

Remarks

The image and the file are only locked if the function succeeds, otherwise the container is unlocked and empty and CAPSUnlockImage has no effect on it.

The function can return imgeOutOfRange for an invalid container, imgeOpen for file open or file identification errors, and imgeType for unknown file content.

Related Functions

CAPSLockImageMemory, CAPSUnlockImage, CAPSGetImageType

CAPSLockImageMemory

The function locks an image into a container device. The image is supplied in a memory buffer rather than a file reference.

```
int32_t CAPSLockImageMemory(  
int32_t id, /* container ID */  
const uint8_t *buffer, /* memory buffer */  
uint32_t length, /* buffer length */  
uint32_t flag /* locking flags */  
);
```

Parameters

id: [in] the container ID returned by CAPSAddImage.

buffer: [in] pointer to the buffer area where the image in memory starts.

length: [in] length of the supplied buffer. It must cover the complete image file data in memory.

flag: [in] locking flags. `DI_LOCK_MEMREF` directly uses the supplied source memory buffer.

Return Values

imageOk if successful or related image error code.

Remarks

This function is useful for retrieving images from archive files by first decompressing the image file to a memory buffer then calling the lock function.

The image is only locked if the function succeeds, otherwise the container is unlocked and empty and CAPSUnlockImage has no effect on it.

If `DI_LOCK_MEMREF` is used the caller must keep the source memory valid for as long as the image may refer to it.

Related Functions

CAPSLockImage, CAPSUnlockImage, CAPSGetImageTypeMemory

CAPSUnlockImage

The function unlocks - "ejects" - an image from a container device.

```
int32_t CAPSUnlockImage(  
int32_t id /* container ID */  
);
```

Parameters

id: [in] the container ID returned by CAPSAddImage.

Return Values

imgeOk if successful or related imge error code.

Remarks

Any resources allocated for the image are freed and the image file is unlocked once the function completes.

Related Functions

CAPSLockImage, CAPSLockImageMemory, CAPSRemImage

CAPSLoadImage

The function locks all unlocked tracks of an image. Already locked tracks remain unchanged. The function is useful for decoding and pre-caching track data for very fast retrieval.

```
int32_t CAPSLoadImage(  
int32_t id, /* container ID */  
uint32_t flag /* locking flags */  
);
```

Parameters

id: [in] the container ID returned by CAPSAddImage.

flag: [in] locking flags.

Return Values

imgeOk if successful or related imge error code.

Remarks

Should only be used when memory usage is not an issue.

Related Functions

CAPSLockTrack, CAPSUnlockAllTracks

CAPSGetImageInfo

The function reads the image information data from a locked image file.

```
int32_t CAPSGetImageInfo(  
PCAPSIMAGEINFO pi, /* pointer to CapsImageInfo */  
int32_t id /* container ID */  
);
```

Parameters

pi: [out] pointer to the CapsImageInfo that receives the image data from the library.

id: [in] the container ID returned by CAPSAddImage.

Return Values

imgeOk if successful or related imge error code.

Remarks

CapsImageInfo has a fixed output type; the result structure is cleared before it is filled. If the function returns an error, callers should not use the output data.

Related Types

CapsImageInfo, CapsDateTimeExt

CAPSLockTrack

The function locks - reads and decodes - a track from a locked image file.

```
int32_t CAPSLockTrack(  
void *ptrackinfo, /* pointer to track information block */  
int32_t id, /* container ID */  
uint32_t cylinder, /* cylinder to read */  
uint32_t head, /* head to read */  
uint32_t flag /* locking flags */  
);
```

Parameters

ptrackinfo: [out] pointer to the track information block that receives the track data from the library.

id: [in] the container ID returned by CAPSAddImage.

cylinder: [in] the cylinder number for the track.

head: [in] the head number for the track.

flag: [in] locking flags.

Return Values

imgeOk if successful or related imge error code.

Remarks

Without `DI_LOCK_TYPE`, ptrackinfo is treated as a CapsTrackInfo structure.

With `DI_LOCK_TYPE`, the first `uint32_t` of ptrackinfo holds the expected structure type: 0 for CapsTrackInfo, 1 for CapsTrackInfoT1, and 2 for CapsTrackInfoT2.

If an unsupported type is requested, the first `uint32_t` is set to the highest supported version and `imgeUnsupportedType` is returned.

The function reads the requested structure type before clearing the output block. If the call returns an error, do not expect ptrackinfo to contain a cleared or complete structure, except for the documented unsupported-type version value.

Subsequent calls locking the same track return the same cached decoded data, regardless of locking flags used, until the track is unlocked directly or indirectly.

CapsTrackInfoT1 includes the track timing buffer and overlap position for callers that need timing data.

Related Functions

CAPSLockTrack, CAPSLockAllTracks, CAPSGetInfo, CAPSSetRevolution

Related Types

CapsTrackInfo, CapsTrackInfoT1, CapsTrackInfoT2

CAPSUnlockTrack

The function unlocks - frees all the resources allocated - a track from the buffers associated with a locked image file.

```
int32_t CAPSUnlockTrack(  
int32_t id, /* container ID */  
uint32_t cylinder, /* cylinder to unlock */  
uint32_t head /* head to unlock */  
);
```

Parameters

id: [in] the container ID returned by CAPSAddImage.

cylinder: [in] the cylinder number for the track.

head: [in] the head number for the track.

Return Values

imgeOk if successful or related imge error code.

Remarks

In order to apply different locking to the same track it must be unlocked first.

Related Functions

CAPSLockTrack, CAPSUnlockAllTracks

CAPSUnlockAllTracks

The function unlocks - frees all the resources allocated - all tracks from the buffers associated with a locked image file.

```
int32_t CAPSUnlockAllTracks(  
int32_t id /* container ID */  
);
```

Parameters

id: [in] the container ID returned by CAPSAddImage.

Return Values

imgeOk if successful or related imge error code.

Remarks

-

Related Functions

CAPSUnlockTrack, CAPSLoadImage

CAPSGetPlatformName

The helper function gets the symbolic name assigned to a platform ID at CapsImageInfo.platform[].

```
const char *CAPSGetPlatformName(  
uint32_t pid /* platform ID */  
);
```

Parameters

pid: [in] the platform ID available from CapsImageInfo.platform[] array members.

Return Values

The return value is a pointer to the symbolic name of the platform or nullptr for an invalid platform ID.

Remarks

ciipNA value should be skipped, it is an unused entry in the platform array.

Related Types

CapsImageInfo

CAPSGetVersionInfo

The function gets the library version and capability information.

```
int32_t CAPSGetVersionInfo(  
void *pversioninfo, /* pointer to CapsVersionInfo */  
uint32_t flag /* information flags */  
);
```

Parameters

pversioninfo: [out] pointer to the version information block.

flag: [in] information flags. If DI_LOCK_TYPE is set, the first uint32_t of the structure holds the expected structure type.

Return Values

imgeOk if successful or related imge error code.

Remarks

Supported structure type: 0, using CapsVersionInfo. Unsupported structure types return imgeUnsupportedType.

The function reads the requested structure type before clearing the output block. If the call returns an error, do not expect pversioninfo to contain a cleared or complete structure, except for the documented unsupported-type version value.

The flag member of CapsVersionInfo receives the supported DI_LOCK_... flags.

Related Types

CapsVersionInfo

CAPSFdcGetInfo

The function gets FDC emulator information.

```
uint32_t CAPSFdcGetInfo(  
int32_t iid, /* information ID */  
PCAPSFDC pc, /* pointer to CapsFdc */  
int32_t ext /* extension parameter */  
);
```

Parameters

iid: [in] information ID, one of the cfdci... values.

pc: [in] pointer to the FDC state. Required for register queries.

ext: [in] reserved extension parameter; pass 0.

Return Values

The requested information value or 0 for an unsupported information ID.

Remarks

cfdciSize_Fdc returns the required size of CapsFdc.

cfdciSize_Drive returns the required size of CapsDrive.

Register information IDs return the current command, status, track, sector or data register value.

Related Functions

CAPSFdcInit

Related Types

CapsFdc, CapsDrive

CAPSFdcInit

The function initialises the FDC emulator.

```
int32_t CAPSFdcInit(  
PCAPSFDC pc /* pointer to CapsFdc */  
);
```

Parameters

pc: [in, out] pointer to the FDC state block.

Return Values

imgeOk if successful or related imge error code.

Remarks

Before calling the function the host program must set the FDC structure size, FDC model, clock frequency, drive pointer and drive counts. Each drive must also have a valid structure size.

Supported FDC model: cfdcmWD1772.

The function clears the FDC state and initialises it to the default emulation state.

CAPSFdcInit clears the state block, restores the documented preset fields, and then sets FDC timing and reset state.

Related Functions

CAPSFdcGetInfo, CAPSFdcReset, CAPSFdcResetState

CAPSFdcReset

The function performs an FDC cold hardware reset.

```
void CAPSFdcReset(  
PCAPSFDC pc /* pointer to CapsFdc */  
);
```

Parameters

pc: [in, out] pointer to the FDC state block.

Return Values

-

Remarks

The function recalculates timing and resets FDC internal state and drive lines. The function expects a valid CapsFdc pointer.

Related Functions

CAPSFdcResetState, CAPSFdcInit

CAPSFdcResetState

The function performs an FDC cold or warm hardware reset.

```
int32_t CAPSFdcResetState(  
PCAPSFDC pc, /* pointer to CapsFdc */  
int32_t reset_state /* reset state */  
);
```

Parameters

pc: [in, out] pointer to the FDC state block.

reset_state: [in] cfders_Cold or cfders_Warm.

Return Values

imgeOk if successful or related imge error code.

Remarks

cfders_Warm preserves the track and data registers across the reset.

Related Functions

CAPSFdcReset

CAPSFdcEmulate

The function runs the FDC emulator for the selected number of clock cycles.

```
void CAPSFdcEmulate(  
PCAPSFDC pc, /* pointer to CapsFdc */  
uint32_t cyclecnt /* clock cycles */  
);
```

Parameters

pc: [in, out] pointer to the FDC state block.

cyclecnt: [in] requested clock cycles to complete.

Return Values

-

Remarks

CapsFdc.clockact receives the clock cycles completed by the emulator call. The function expects a valid initialised FDC state.

The host normally calls this repeatedly with elapsed FDC clock cycles. The emulator stores the requested cycle count in clockreq and the completed cycle count in clockact.

The WD1772 register addresses used by the emulator are 0 for command, 1 for track, 2 for sector and 3 for data. Data is masked by datamask before it is stored on the emulated data bus.

Related Functions

CAPSFdcRead, CAPSFdcWrite

CAPSFdcRead

The function reads from an FDC register as an emulated machine would, affecting the internal FDC state.

```
uint32_t CAPSFdcRead(  
PCAPSFDC pc, /* pointer to CapsFdc */  
uint32_t address /* register address */  
);
```

Parameters

pc: [in, out] pointer to the FDC state block.

address: [in] FDC register address.

Return Values

The register data value.

Remarks

The address is masked by CapsFdc.addressmask and the returned data is masked by CapsFdc.datamask.

Address 0 reads the status register and clears CAPSFDC_LO_INTRQ. Address 1 reads the track register. Address 2 reads the sector register. Address 3 reads the data register and clears CAPSFDC_LO_DRQ.

The WD1772 register addresses used by the emulator are 0 for status, 1 for track, 2 for sector and 3 for data. Reading the status register clears INTRQ, and reading the data register clears DRQ.

Related Functions

CAPSFdcWrite, CAPSFdcGetInfo

CAPSFdcWrite

The function writes to an FDC register as an emulated machine would, affecting the internal FDC state.

```
void CAPSFdcWrite(  
PCAPSFDC pc, /* pointer to CapsFdc */  
uint32_t address, /* register address */  
uint32_t data /* register data */  
);
```

Parameters

pc: [in, out] pointer to the FDC state block.

address: [in] FDC register address.

data: [in] FDC register data.

Return Values

-

Remarks

The address is masked by CapsFdc.addressmask and the data is masked by CapsFdc.datamask.

Address 0 writes the command register. Type 4 commands are processed always; type 1 to type 3 commands are processed only if the FDC is not busy.

Address 1 writes the track register. Address 2 writes the sector register. Address 3 writes the data register and clears CAPSFDC_LO_DRQ.

Track and sector register writes are not protected during a busy state.

Related Functions

CAPSFdcRead, CAPSFdcEmulate

CAPSFdcInvalidateTrack

The function invalidates track data for an FDC drive.

```
int32_t CAPSFdcInvalidateTrack(  
PCAPSFDC pc, /* pointer to CapsFdc */  
int32_t drive /* drive number */  
);
```

Parameters

pc: [in, out] pointer to the FDC state block.

drive: [in] drive number.

Return Values

imgeOk if successful or related imge error code.

Remarks

The next read cycle will request a track change callback as well as re-lock the stream.

Related Types

CapsFdc, CapsDrive, CAPSFDCHOOK

CAPSFormatDataToMFM

The function performs MFM format conversion.

```
int32_t CAPSFormatDataToMFM(  
void *pformattrack, /* pointer to CapsFormatTrack */  
uint32_t flag /* conversion flags */  
);
```

Parameters

pformattrack: [in, out] pointer to a CapsFormatTrack structure.

flag: [in] conversion flags. If DI_LOCK_TYPE is set, the first uint32_t of the structure holds the expected structure type.

Return Values

imgeOk if successful or related imge error code.

Remarks

Supported structure type: 0, using CapsFormatTrack.

The function reads the requested structure type before using the CapsFormatTrack block. If the call returns an error, do not expect pformattrack fields such as bufreq to be meaningful unless the specific return path documents them, such as the size-query path.

If trackbuf, tracklen or buflen is zero, the function calculates the required buffer size in CapsFormatTrack.bufreq.

Supported data sector lengths are 128, 256, 512 and 1024 bytes.

The caller should clear or otherwise initialise CapsFormatTrack and the CapsFormatBlock array before filling the sector descriptors and converting the track.

Related Types

CapsFormatTrack, CapsFormatBlock

CAPSGetInfo

The function gets various information after decoding.

```
int32_t CAPSGetInfo(  
void *pinfo, /* information block */  
int32_t id, /* container ID */  
uint32_t cylinder, /* cylinder */  
uint32_t head, /* head */  
uint32_t inftype, /* information type */  
uint32_t infid /* information ID */  
);
```

Parameters

pinfo: [out] pointer to the information block that receives the data.

id: [in] the container ID returned by CAPSAddImage.

cylinder: [in] the cylinder number for the track.

head: [in] the head number for the track.

inftype: [in] information type, one of the cgiit... values.

infid: [in] information ID within the selected information type.

Return Values

imgeOk if successful or related imge error code.

Remarks

cgiitSector fills CapsSectorInfo.

cgiitWeak fills CapsDataInfo.

cgiitRevolution fills CapsRevolutionInfo.

The inftype selector identifies the output structure type before the selected structure is cleared. If the call returns an error, do not expect pinfo to contain a cleared or complete structure.

Related Functions

CAPSLockTrack, CAPSSetRevolution

Related Types

CapsSectorInfo, CapsDataInfo, CapsRevolutionInfo

CAPSSetRevolution

The function sets the next revolution to be used by track lock.

```
int32_t CAPSSetRevolution(  
int32_t id, /* container ID */  
uint32_t value /* revolution */  
);
```

Parameters

id: [in] the container ID returned by CAPSAddImage.

value: [in] revolution number.

Return Values

imgeOk if successful or related imge error code.

Remarks

This function is used with images where multiple revolutions or changing track data can be selected by the API. The exact meaning of the revolution number is format dependent.

Related Functions

CAPSLockTrack, CAPSGetInfo

CAPSGetImageType

The function gets the image type using a file name.

```
int32_t CAPSGetImageType(  
const char *name /* filename */  
);
```

Parameters

name: [in] the name of the image file to be checked.

Return Values

One of the cit... recognized image type values. citError means an error prevented the type identification.

Remarks

The function identifies the image type without locking the image into a container.

Related Functions

CAPSGetImageTypeMemory, CAPSLockImage

CAPSGetImageTypeMemory

The function gets the image type using the specified memory buffer.

```
int32_t CAPSGetImageTypeMemory(  
const uint8_t *buffer, /* memory buffer */  
uint32_t length /* buffer length */  
);
```

Parameters

buffer: [in] pointer to the buffer area where the image in memory starts.

length: [in] length of the supplied buffer.

Return Values

One of the cit... recognized image type values. citError means an error prevented the type identification.

Remarks

The function identifies the image type without locking the image into a container.

Related Functions

CAPSGetImageType, CAPSLockImageMemory

CAPSGetDebugRequest

This emulator helper function returns the state of the host debug request coming from the library and resets the state.

```
int32_t CAPSGetDebugRequest(void);
```

Parameters

-

Return Values

The previous debug request state.

Remarks

No public debug request value constants are declared in the public headers.

Structures and Types

CapsDateTimeExt

The structure is used to retrieve date/time information from the packed format used by IPF files. It expands the value into separate calendar and time fields and is used by CapsImageInfo.crdt to report the image creation date and time.

```
struct CapsDateTimeExt {  
    uint32_t year;  
    uint32_t month;  
    uint32_t day;  
    uint32_t hour;  
    uint32_t min;  
    uint32_t sec;  
    uint32_t tick;  
};  
typedef struct CapsDateTimeExt *PCAPSDATETIMEEXT;
```

Members

year	Year component of the image creation date, decoded from the packed date/time stored in the image metadata.
month	Month component of the image creation date, decoded from the packed date/time stored in the image metadata.
day	Day component of the image creation date, decoded from the packed date/time stored in the image metadata.
hour	Hour component of the image creation time, decoded from the packed date/time stored in the image metadata.
min	Minute component of the image creation time, decoded from the packed date/time stored in the image metadata.
sec	Second component of the image creation time, decoded from the packed date/time stored in the image metadata.
tick	Tick value decoded from the packed source time. This preserves the sub-second part available in the image metadata.

Related Types

CapsImageInfo

CapsVersionInfo

The structure is used to retrieve library version information from CAPSGetVersionInfo. It reports the library type, release, revision and DI_LOCK flag bits supported by the library.

```
struct CapsVersionInfo {  
    uint32_t type;  
    uint32_t release;  
    uint32_t revision;  
    uint32_t flag;  
};  
typedef struct CapsVersionInfo *PCAPSVERSIONINFO;
```

Members

type	Library type identifier. When CAPSGetVersionInfo is called with DI_LOCK_TYPE, set this field to the requested CapsVersionInfo structure type; on an unsupported request the library writes back the highest supported type and returns imgeUnsupportedType.
release	Library release number returned by CAPSGetVersionInfo.
revision	Library revision number returned by CAPSGetVersionInfo.
flag	Supported flags. CAPSGetVersionInfo fills this with the DI_LOCK_* flags supported by the library build.

Related Functions

CAPSGetVersionInfo

Library Version Definitions

Compile-time release and revision definitions from CapsLibVersion.h. They identify the library build and correspond to the release and revision fields reported through CapsVersionInfo.

CAPS_LIB_RELEASE: compile-time library release number.

CAPS_LIB_REVISION: compile-time library revision number.

CapsImageInfo

The structure is used to retrieve generic information about the IPF image through CAPSGetImageInfo. The values describe the image type and release identifiers, the cylinder/head range that contains valid data, the image creation date/time, and the intended platform identifiers.

Tracks are addressed as cylinder.head in track-related functions. Double-sided Amiga or Atari ST images often start at 0.0 and end at 79.1, but callers must not assume those values; use mincylinder, maxcylinder, minhead and maxhead from this structure.

```
struct CapsImageInfo {
uint32_t type;
uint32_t release;
uint32_t revision;
uint32_t mincylinder;
uint32_t maxcylinder;
uint32_t minhead;
uint32_t maxhead;
struct CapsDateTimeExt crdt;
uint32_t platform[CAPS_MAXPLATFORM];
};
typedef struct CapsImageInfo *PCAPSIMAGEINFO;
```

Members

type	Image type. CapsImageInfo.type receives a ciit* value, such as ciitFDD for a floppy disk image or ciitNA when the image type is unavailable.
release	Image release identifier stored in the image metadata. Each IPF file is designed to have a unique release ID that can be used as a database key for information about the file or about a group of files. More than one file can share the same release ID when all of them belong to the same release, such as a game supplied on multiple disks. ID 0 is invalid, only used by test images.
revision	Image release revision stored in the image metadata. More than one revision of the same release can exist; later revisions replace older ones. The normal revision number is 1.

mincylinder	Lowest cylinder number valid for the image when calling track-related functions. The value is inclusive and should be used instead of assuming common disk geometry.
maxcylinder	Highest cylinder number valid for the image when calling track-related functions. The value is inclusive and should be used instead of assuming common disk geometry.
minhead	Lowest head number valid for the image when calling track-related functions. Use together with maxhead to determine the valid head range.
maxhead	Highest head number valid for the image when calling track-related functions. Use together with minhead to determine the valid head range.
crdt	Creation date and time of the IPF image, expanded into a CapsDateTimeExt structure.
Platform [CAPS_MAXPLATFORM]	<p>Intended platform identifiers. The array contains up to CAPS_MAXPLATFORM valid ciip* entries for the platforms the image is naturally linked with; dual or tri-format disks may have more than one valid entry. Unused or invalid entries use ciipNA.</p> <p>These entries represent only the intended use, e.g. it is possible to read an Atari disk using an Amiga, but this is not intended platform, therefore only an Atari entry will be found.</p>

Related Functions

CAPSGetImageInfo, CAPSGetPlatformName

CapsImageInfo.type

Values allowed for the type member of CapsImageInfo. Additional image type values may be available with later library versions.

Values

ciitNA	Invalid or unavailable image type.
ciitFDD	Floppy disk image.

CapsImageInfo.platform

Values allowed for the platform member of CapsImageInfo. Additional platform IDs may be available with later library versions.

Values

ciipNA	Invalid platform, used as a dummy entry in platform arrays; skip this entry when interpreting CapsImageInfo.platform.
ciipAmiga	Commodore Amiga platform identifier.
ciipAtariST	Atari ST platform identifier.
ciipPC	IBM PC compatible platform identifier.
ciipAmstradCPC	Amstrad CPC platform identifier.
ciipSpectrum	Sinclair ZX Spectrum platform identifier.
ciipSamCoupe	SAM Coupe platform identifier.
ciipArchimedes	Acorn Archimedes platform identifier.
ciipC64	Commodore 64 platform identifier.
ciipAtari8	Atari 8-bit platform identifier.

Remarks

These platform IDs are not about configuration but intended use.

CapsTrackInfo

The structure is used to retrieve generic information about a track of a disk image from an IPF file through CAPSLockTrack. It identifies the selected cylinder/head, decoded sector information, the track buffer and its length, optional per-track data buffers, multi-revolution track variants, and optional timing or density-map data.

```
struct CapsTrackInfo {  
    uint32_t type;  
    uint32_t cylinder;  
    uint32_t head;  
    uint32_t sectorcnt;  
    uint32_t sectorsize;  
    uint32_t trackcnt;  
    uint8_t *trackbuf;  
    uint32_t tracklen;  
    uint8_t *trackdata[CAPS_MTRS];  
    uint32_t tracksize[CAPS_MTRS];  
    uint32_t timelen;  
    uint32_t *timebuf;  
};  
typedef struct CapsTrackInfo *PCAPSTRACKINFO;
```

Members

type	Track type. The value is a ctit* base type and may include CTIT_FLAG_FLKEY; use CTIT_MASK_TYPE to extract the base type.
cylinder	Cylinder number requested from CAPSLockTrack or described by the returned track information.
head	Head number requested from CAPSLockTrack or described by the returned track information.
sectorcnt	Number of sector descriptors available for the locked track. Use CAPSGetInfo with cgitSector and an index below this count to query each sector.
sectorsize	Default sector size in bytes for the track when it is available from the image. Individual sector sizes are available through CapsSectorInfo.

trackcnt	Number of revolutions or data-track variants decoded for this track. A locked ordinary track normally reports 1; multi-revolution tracks can report values greater than 1. The trackdata and tracksize arrays contain the per-variant buffers and sizes.
trackbuf	Pointer to the decoded track data buffer. For multi-revolution tracks it points to the first returned buffer. The pointer can be null if the track is unlocked or decoding did not produce a buffer.
tracklen	Total track buffer length. For multi-revolution data this is the sum of the returned track lengths; use tracksize[] for an individual revolution or variant. With DI_LOCK_TRKBIT the value is in bits; otherwise, it is byte-sized and may be rounded by locking flags.
Trackdata [CAPS_MTRS]	Track data pointers as decoded by the library. Up to CAPS_MTRS entries can be valid, with the count in trackcnt and the first valid entry at trackdata[0]. Entries can be null when a variant is unavailable.
Tracksize [CAPS_MTRS]	Track data sizes paired with trackdata. Up to CAPS_MTRS entries can be valid, with the count in trackcnt and the first valid entry at tracksize[0]. Values can be 0 when a variant is unavailable; with DI_LOCK_TRKBIT the unit follows bit-sized track data.
timelen	Number of entries in the timing buffer or cell density map. It can be 0 when timing or density data was not requested or is not available for the selected track and locking flags.
timebuf	Pointer to the timing buffer or cell density map. It can be null when timing or density data was not requested or is not available for the selected track and locking flags; when present, contents and units depend on DI_LOCK_DEN* and DI_LOCK_DENALT.

Remarks

With DI_LOCK_TRKBIT, tracklen is in bits and the track buffer is bit sized.

Related Functions

CAPSLockTrack

CapsTrackInfoT1

CapsTrackInfoT1 is a type 1 track information block returned by CAPSLockTrack when DI_LOCK_TYPE is used. It provides the decoded track buffer, optional timing or density-map data, and the overlap position for callers that need track wrap information.

```
struct CapsTrackInfoT1 {  
    uint32_t type;  
    uint32_t cylinder;  
    uint32_t head;  
    uint32_t sectorcnt;  
    uint32_t sectorsize;  
    uint8_t *trackbuf;  
    uint32_t tracklen;  
    uint32_t timelen;  
    uint32_t *timebuf;  
    int32_t overlap;  
};  
typedef struct CapsTrackInfoT1 *PCAPSTRACKINFOT1;
```

Members

type	Track type. The value is a ctit* base type and may include CTIT_FLAG_FLAKEKEY; use CTIT_MASK_TYPE to extract the base type.
cylinder	Cylinder number requested from CAPSLockTrack or described by the returned track information.
head	Head number requested from CAPSLockTrack or described by the returned track information.
sectorcnt	Number of sector descriptors available for the locked track. Use CAPSGetInfo with cgiitSector and an index below this count to query each sector.
sectorsize	Default sector size in bytes for the track when it is available from the image. Individual sector sizes are available through CapsSectorInfo.
trackbuf	Pointer to the decoded track data buffer. The pointer remains valid only while the image or track lock that produced it

	remains valid, and it can be null if decoding did not produce a buffer.
tracklen	Track buffer length. With DI_LOCK_TRKBIT the value is in bits; otherwise it is byte-sized and may be rounded by locking flags.
timelen	Number of entries in the timing buffer or cell density map. It can be 0 when timing or density data was not requested or is not available for the selected track and locking flags.
timebuf	Pointer to the timing buffer or cell density map. It can be null when timing or density data was not requested or is not available; when present, contents and units depend on DI_LOCK_DEN* and DI_LOCK_DENALT.
overlap	Overlap position where the decoded track wraps. With DI_LOCK_OVLBIT the value is in bits; otherwise it is byte-oriented.

Remarks

With DI_LOCK_OVLBIT, overlap is in bits.

Related Functions

CAPSLockTrack

CapsTrackInfoT2

CapsTrackInfoT2 is a type 2 track information block returned by CAPSLockTrack when DI_LOCK_TYPE is used. It extends CapsTrackInfoT1 with the decoded start bit, weak-bit generator seed and weak data area count so callers can inspect weak or flakey regions.

```
struct CapsTrackInfoT2 {
    uint32_t type;
    uint32_t cylinder;
    uint32_t head;
    uint32_t sectorcnt;
    uint32_t sectorsize;
    uint8_t *trackbuf;
    uint32_t tracklen;
    uint32_t timelen;
    uint32_t *timebuf;
    int32_t overlap;
    uint32_t startbit;
    uint32_t wseed;
    uint32_t weakcnt;
};
typedef struct CapsTrackInfoT2 *PCAPSTRACKINFOT2;
```

Members

type	Track type. The value is a ctit* base type and may include CTIT_FLAG_FLKEY; use CTIT_MASK_TYPE to extract the base type.
cylinder	Cylinder number requested from CAPSLockTrack or described by the returned track information.
head	Head number requested from CAPSLockTrack or described by the returned track information.
sectorcnt	Number of sector descriptors available for the locked track. Use CAPSGetInfo with cgitSector and an index below this count to query each sector.

sectorsize	Sector size field retained for layout compatibility in CapsTrackInfoT2. Use CapsSectorInfo for individual sector sizes.
trackbuf	Pointer to the decoded track data buffer. The pointer remains valid only while the image or track lock that produced it remains valid, and it can be null if decoding did not produce a buffer.
tracklen	Track buffer length. With DI_LOCK_TRKBIT the value is in bits; otherwise it is byte-sized and may be rounded by locking flags.
timelen	Number of entries in the timing buffer or cell density map. It can be 0 when timing or density data was not requested or is not available for the selected track and locking flags.
timebuf	Pointer to the timing buffer or cell density map. It can be null when timing or density data was not requested or is not available; when present, contents and units depend on DI_LOCK_DEN* and DI_LOCK_DENALT.
overlap	Overlap position where the decoded track wraps. With DI_LOCK_OVLBIT the value is in bits; otherwise it is byte-oriented.
startbit	Start position of the decoding in bits. DI_LOCK_INDEX realigns the track to the index and reports a zero start bit.
wseed	Weak bit generator seed data. With DI_LOCK_SETWSEED, the caller supplies this value before locking and receives the resulting seed after locking.
weakcnt	Number of weak or flakey data areas reported for the track.

Remarks

With DI_LOCK_SETWSEED, wseed supplies the weak bit generator seed value.

Related Functions

CAPSLockTrack, CAPSGetInfo

CapsTrackInfo.type

Values allowed for the type member of CapsTrackInfo. The value describes the track density type returned by CAPSLockTrack. Additional track type values may be available with later library versions.

Values

ctitNA	Invalid or unavailable track type.
ctitNoise	The track is not formatted; cells are random-sized. Data or density for unformatted tracks is not generated unless requested when locking the track, such as with DI_LOCK_NOISE, DI_LOCK_NOISEREV or DI_LOCK_DENNOISE. When data or density is not generated, related pointers and values can be null or 0.
ctitAuto	Each cell group has the same width or timing. The cell density can be generated by evenly distributing the cell groups over the time it takes to read a track. Density for these tracks is not generated unless requested when locking the track; empty density maps leave related pointers and values null or 0.
ctitVar	The track contains cell groups with variable density. When requested, such as with DI_LOCK_DENVAR, the library generates a cell density map containing the density values for the track. Density is not generated unless requested when locking the track; empty density maps leave related pointers and values null or 0.

Remarks

The value may be combined with CTIT_FLAG_FLKEY. Use CTIT_MASK_TYPE to mask the base type.

Public Size Definitions

Fixed public array sizes and track type masks used by the structures above. CAPS_MAXPLATFORM is the number of platform slots in CapsImageInfo.platform, CAPS_MTRS is the number of track data pointer/size slots in CapsTrackInfo, and CTIT_MASK_TYPE removes CTIT_FLAG_FLAKEY from a track type value.

Values

CAPS_MAXPLATFORM	Maximum number of platform IDs stored in CapsImageInfo.platform.
CAPS_MTRS	Maximum number of track data pointers and sizes stored in CapsTrackInfo.
CTIT_FLAG_FLAKEY	Track type flag indicating flakey or weak data. The flag may be ORed into CapsTrackInfo.type; use CTIT_MASK_TYPE to recover the base ctit* value.
CTIT_MASK_TYPE	Mask for the base CapsTrackInfo track type value. Apply it to remove CTIT_FLAG_FLAKEY before comparing against ctitNA, ctitNoise, ctitAuto or ctitVar.

CapsSectorInfo

The structure is returned by CAPSGetInfo with cgiitSector for a sector index below CapsTrackInfo.sectorcnt. It reports descriptor and decoded data/gap sizes and positions in bits, write-splice gap sizes and modes, and cell/encoder metadata for the selected sector.

```
struct CapsSectorInfo {  
    uint32_t descdatasize;  
    uint32_t descgapsize;  
    uint32_t datasize;  
    uint32_t gapsize;  
    uint32_t datastart;  
    uint32_t gapstart;  
    uint32_t gapsizews0;  
    uint32_t gapsizews1;  
    uint32_t gapws0mode;  
    uint32_t gapws1mode;  
    uint32_t celltype;  
    uint32_t enctype;  
};
```

```
typedef struct CapsSectorInfo *PCAPSSECTORINFO;
```

Members

descdatasize	Data size in bits from the IPF descriptor before decoding adjustments.
descgapsize	Gap size in bits from the IPF descriptor before decoding adjustments.
datasize	Data field size in bits after decoding.
gapsize	Gap size in bits after decoding.
datastart	Data start position in bits from the start of the decoded track.
gapstart	Gap start position in bits from the start of the decoded track.
gapsizews0	Gap size before the write splice point.
gapsizews1	Gap size after the write splice point.

gapws0mode	Gap size mode before the write splice point. The value is a csiegm* mode describing whether the size is fixed, automatic or scripted.
gapws1mode	Gap size mode after the write splice point. The value is a csiegm* mode describing whether the size is fixed, automatic or scripted.
celltype	Bitcell type used by this sector descriptor, stored as a csic* value.
enctype	Encoder type used by this sector descriptor, stored as a csie* value.

Related Functions

CAPSGetInfo

CapsSectorInfo.bitcell type

Bitcell type stored in CapsSectorInfo.celltype.

Values

csicNA	Invalid bitcell type.
csic2us	2us bitcells.

CapsSectorInfo.encoder type

Encoder type stored in CapsSectorInfo.encype.

Values

csieNA	Undefined encoder.
csieMFM	MFM encoder.
csieRaw	No encoder is used; raw sector data only.

CapsSectorInfo.gap size mode

Write splice gap size mode stored in CapsSectorInfo.gapws0mode and gapws1mode.

Values

csiegmFixed	Fixed gap size; it cannot be changed by resize processing.
csiegmAuto	Gap size can be changed and resize information is calculated automatically.
csiegmResize	Gap size can be changed and resize information is scripted.

CapsDataInfo

The structure is returned by CAPSGetInfo with cgitWeak to describe a weak or flakey data area in the decoded track. The start and size fields are bit positions and counts in the locked track data.

```
struct CapsDataInfo {  
    uint32_t type;  
    uint32_t start;  
    uint32_t size;  
};  
typedef struct CapsDataInfo *PCAPSDATAINFO;
```

Members

type	Data type for a CapsDataInfo entry, such as cditWeak for weak-bit areas.
start	Start position of the described data area in bits from the start of the decoded track.
size	Size of the described data area in bits.

Related Functions

CAPSGetInfo

CapsDataInfo.data type

Data type stored in CapsDataInfo.type.

Values

cbitNA	Undefined data type.
cbitWeak	Weak bits. CapsDataInfo entries of this type describe weak or flakey bit areas in the decoded track.

CapsRevolutionInfo

The structure is returned by CAPSGetInfo with cgiitRevolution for images where multiple revolutions or changing track data can be selected by the API. It describes the next, previous, real and maximum revolution values; the exact revolution numbering is format dependent.

```
struct CapsRevolutionInfo {  
    int32_t next;  
    int32_t last;  
    int32_t real;  
    int32_t max;  
};  
typedef struct CapsRevolutionInfo *PCAPSREVOLUTIONINFO;
```

Members

next	The revolution number used by the next track lock call. CAPSSetRevolution can change this value for the selected track.
last	The revolution number used by the last track lock call.
real	The real revolution number used by the last track lock call after the library has resolved generated, random or wrapped revolution selection.
max	The maximum revolution available for the selected track. A value below zero means unlimited or randomized generated data, and zero means no revolution data is available.

Related Functions

CAPSGetInfo, CAPSSetRevolution

CapsDrive

CapsDrive is the drive state block used by the FDC emulator. It describes drive geometry, selected track and side, disk attributes, index timing, track/timing buffers and caller-owned user fields used while CAPSFdcEmulate processes a drive.

```
struct CapsDrive {
uint32_t type;
uint32_t rpm;
int32_t maxtrack;
int32_t track;
int32_t buftrack;
int32_t side;
int32_t bufside;
int32_t newside;
uint32_t diskattr;
uint32_t idistance;
uint32_t clockrev;
int32_t clockip;
int32_t ipcnt;
uint32_t ttype;
uint8_t *trackbuf;
uint32_t *timebuf;
uint32_t tracklen;
int32_t overlap;
int32_t trackbits;
int32_t ovlmin;
int32_t ovlmax;
int32_t ovlcnt;
int32_t ovlact;
int32_t nact;
uint32_t nseed;
void *userptr;
uint32_t userdata;
```

```
};
```

```
typedef struct CapsDrive *PCAPSDRIVE;
```

Members

type	Structure size; set to sizeof(CapsDrive) before CAPSFdcInit so the emulator can validate the caller-supplied drive block.
rpm	Drive speed in revolutions per minute. The 3.5 DD default is CAPSDRIVE_35DD_RPM.
maxtrack	Highest track reachable before the drive hard stop. CAPSDRIVE_35DD_HST is the 3.5 DD default.
track	Actual physical track position of the drive head.
buftrack	Track number currently represented by the loaded track buffer.
side	Actual side selected for processing.
bufside	Side number currently represented by the loaded track buffer.
newside	Side to select after processing; host code may set this before the emulator updates the active side.
diskattr	Disk attributes. Set CAPSDRIVE_DA_IN to insert a disk, CAPSDRIVE_DA_WP to mark write protection, CAPSDRIVE_DA_MO when the motor is on, and CAPSDRIVE_DA_SS for a single-sided drive.
idistance	Distance from the index pulse in FDC clock cycles.
clockrev	FDC clock cycles per disk revolution, derived from drive RPM and FDC clock frequency during initialisation.
clockip	FDC clock cycles for which the index pulse remains active.
ipcnt	Index pulse clock counter. Values below zero mean initialisation state and zero means the pulse counter is stopped.
ttype	Track type for the currently loaded drive buffer, using the same ctit* values as CapsTrackInfo.type. The value describes how the FDC emulator should interpret the drive track buffer.
trackbuf	Track buffer memory supplied by CAPSLockTrack or CAPSFormatDataToMFM before FDC emulation reads the track. Keep the buffer valid while the FDC drive state refers to it.
timebuf	Timing buffer supplied with locked IPF track data when available; used by the emulator for variable density timing.

	Keep the timing buffer valid while the FDC drive state refers to it.
tracklen	Track buffer memory length copied from locked track information or a formatted track. The unit follows the source track buffer setup.
overlap	Overlap position copied from locked track information or the formatter start position. It defines where circular track data wraps.
trackbits	Used track size in bits after the emulator has prepared the track for processing.
ovlmin	First bit position of the active overlap area.
ovlmax	Last bit position of the active overlap area.
ovlcnt	Overlay bit counter used while the emulator crosses the overlap area.
ovlact	Active overlay phase while processing circular track overlap.
nact	Active noise phase for generated unformatted or noisy track data.
nseed	Noise generator seed used when generated unformatted or noisy data is required.
userptr	Host application pointer preserved by the library. Use it to associate emulator or callback state with this structure.
userdata	Host application data word preserved by the library. Use it to associate small caller state with this structure.

Related Functions

CAPSFdcInit, CAPSFdcInvalidateTrack

Drive Defaults

Default drive geometry values for a 3.5 inch double density drive. Use them as starting values for CapsDrive.rpm and CapsDrive.maxtrack when emulating that drive type.

Values

CAPSDRIVE_35DD_RPM	3.5 DD drive speed in RPM. Use as the default CapsDrive.rpm value for a 3.5 inch double-density drive.
CAPSDRIVE_35DD_HST	3.5 DD hard stop track value. Use as the default CapsDrive.maxtrack value for a 3.5 inch double-density drive.

Disk and Drive Attribute Flags

Bits used in CapsDrive.diskattr to describe whether media is inserted, write protected, motor enabled, or single sided. The index-pulse mask combines the inserted and motor-on states required for index pulse availability.

Values

CAPSDRIVE_DA_IN	Disk inserted. If no disk is inserted, the drive is treated as unavailable and write protected by status logic.
CAPSDRIVE_DA_WP	Disk write protected. The FDC status reports write protect while this bit is set.
CAPSDRIVE_DA_MO	Motor on. The index pulse is only available when this bit and CAPSDRIVE_DA_IN are both set.
CAPSDRIVE_DA_SS	Single-sided drive. When set, side 1 cannot be accessed as a separate disk side.
CAPSDRIVE_DA_IPMASK	Index pulse availability mask. The index pulse is available only when all bits in this mask, CAPSDRIVE_DA_IN and CAPSDRIVE_DA_MO, are set.

CAPSFDCHOOK

FDC hook function type.

```
typedef struct CapsFdc *PCAPSFDC;
```

```
typedef void(__cdecl *CAPSFDCHOOK)(PCAPSFDC pfdc, uint32_t state);
```

Parameters

pfdc	Pointer to the CapsFdc state block passed to an FDC callback.
state	Hook state. For IRQ and DRQ callbacks this is the new line state; for the track callback this is the drive number that needs track data.

Remarks

The hook function pointer and any host state it uses must remain valid for as long as the FDC state can call it.

Related Types

CapsFdc

CapsFdc

CapsFdc is the FDC emulator state block. It contains the controller model, clocking, bus masks, output lines, command/status registers, data separator state, attached drives and callbacks used by CAPSFdcInit, CAPSFdcReset, CAPSFdcEmulate, CAPSFdcRead and CAPSFdcWrite.

```
struct CapsFdc {  
    uint32_t type;  
    uint32_t model;  
    uint32_t endrequest;  
    uint32_t clockact;  
    uint32_t clockreq;  
    uint32_t clockfrq;  
    uint32_t addressmask;  
    uint32_t dataline;  
    uint32_t datamask;  
    uint32_t lineout;  
    uint32_t runmode;  
    uint32_t runstate;  
    uint32_t r_st0;  
    uint32_t r_st1;  
    uint32_t r_stm;  
    uint32_t r_command;  
    uint32_t r_track;  
    uint32_t r_sector;  
    uint32_t r_data;  
    uint32_t seclenmask;  
    uint32_t seclen;  
    uint32_t crc;  
    uint32_t crccnt;  
    uint32_t amdecode;  
    uint32_t aminfo;  
    uint32_t amisigmask;
```

```
int32_t amdatadelay;
int32_t amdataskip;
int32_t ammarkdist;
int32_t ammarktype;
uint32_t dsr;
int32_t dsrent;
int32_t datalock;
uint32_t datamode;
uint32_t datacycle;
uint32_t dataphase;
uint32_t datapcnt;
int32_t indexcount;
int32_t indexlimit;
int32_t readlimit;
int32_t verifylimit;
int32_t spinupcnt;
int32_t spinuplimit;
int32_t idlecnt;
int32_t idllelimit;
uint32_t clockcnt;
uint32_t steptime[4];
uint32_t clockstep[4];
uint32_t hstime;
uint32_t clockhs;
uint32_t iptime;
uint32_t updatetime;
uint32_t clockupdate;
int32_t drivecnt;
int32_t drivemax;
int32_t drivenew;
int32_t drivesel;
int32_t driveact;
```

```

PCAPSDRIVE drivepre;
PCAPSDRIVE drive;
CAPSFDCHOOK cbirq;
CAPSFDCHOOK cbdrq;
CAPSFDCHOOK cbtrk;
void *userptr;
uint32_t userdata;
};

```

Members

type	Structure size; set to sizeof(CapsFdc) before CAPSFdcInit so the emulator can validate the caller-supplied FDC state block.
model	FDC model selected before CAPSFdcInit. Use cfdcmWD1772 for the WD1772-compatible emulator.
endrequest	End-request flags. CAPSFDC_ER_COMEND marks command completion and CAPSFDC_ER_REQEND requests execution to stop at the current emulation cycle.
clockact	Clock cycles completed by the last CAPSFdcEmulate call.
clockreq	Clock cycles requested for the current CAPSFdcEmulate call.
clockfrq	FDC clock frequency in Hz. Set this before CAPSFdcInit so timing values can be converted to clock cycles.
addressmask	Valid address-line mask. CAPSFdcInit sets this to 0x03 for WD1772 register addresses 0...3; CAPSFdcRead and CAPSFdcWrite mask the supplied register address with this value.
dataline	Current data bus value used by the emulated register interface.
datamask	Valid data-line mask. CAPSFdcInit sets this to 0xff for an 8-bit data bus; CAPSFdcRead and CAPSFdcWrite mask data with this value.
lineout	Output line flags exposed by the emulator, including DRQ, INTRQ, motor, direction and internal command state bits.
runmode	Current FDC run mode, using cfdcrm* values to identify the command processing loop.
runstate	Local run state inside the active command processing loop.

r_st0	Primary FDC status register. CAPSFdcRead returns status through the active status mask.
r_st1	Secondary FDC status register used for command-specific status bits.
r_stm	Status mask register; one bits select r_st1 bits and zero bits select r_st0 bits when composing the visible status register.
r_command	FDC command register written through CAPSFdcWrite.
r_track	FDC track register.
r_sector	FDC sector register.
r_data	FDC data register used for register reads, writes and data transfer.
seclenmask	Sector length mask used while processing sector-size fields.
seclen	Current sector length used by the active command.
crc	CRC accumulator used while reading or writing address and data fields.
crcnt	CRC bit counter used while the CRC logic is active.
amdecode	Address-mark detector decoder and shifter state.
aminfo	Address-mark information flags. The CAPSFDC_AI_* bits describe detector, CRC and data-shift-register state.
amisigmask	Enabled address-mark signal bits used to decide which AM information changes should be signalled.
amdatadelay	Address-mark data delay in FDC clock cycles.
amdataskip	Address-mark data skip interval in FDC clock cycles.
ammarkdist	Invalid distance from the last address mark in bitcells, or zero when the mark distance is valid.
ammarktype	Last address-mark type detected by the AM decoder.
dsr	Data shift register that assembles bytes and address-mark values from incoming bitcells.
dsrent	Data shift register bit counter.
datalock	Data lock bit position. Values below zero mean the data separator is not locked.
datamode	Data access mode, using cfdcdm* values to identify whether the emulator is reading no line, noise, plain data or density-mapped data.
datacycle	Clock-cycle remainder for the current bitcell.

dataphase	Data access phase inside the current read or write operation.
datapcnt	Data phase counter used by the active data access mode.
indexcount	Index pulse counter used by commands that wait for or count index pulses.
indexlimit	Index pulse abort point for commands that stop after a maximum number of index pulses.
readlimit	Read abort point used when a read command cannot find the requested data before the configured limit.
verifylimit	Verify abort point used by commands that must verify track position or data before continuing.
spinupcnt	Counter used to model spin-up status.
spinuplimit	Spin-up point at which the emulated drive is considered ready.
idlecnt	Idle counter used while the FDC is not actively transferring data.
idlelimit	Idle point used to control wait timing between operations.
clockcnt	General FDC clock counter.
steptime[4]	Stepping rates in microseconds for the WD1772 step-rate selections.
clockstep[4]	Stepping rates converted to FDC clock cycles.
hstime	Head-settling delay in microseconds.
clockhs	Head-settling delay converted to FDC clock cycles.
iptime	Index pulse hold time in microseconds.
updatetime	Update delay between short operations in microseconds.
clockupdate	Update delay converted to FDC clock cycles.
drivecnt	Number of valid CapsDrive entries supplied in drive.
drivemax	Number of active emulated drives; must not exceed drivecnt.
drivenew	Drive to select after processing. Host code sets this value to choose the active drive before the emulator updates drive selection.
drivesel	Drive selected for processing. Values below zero mean no valid drive is selected.
driveact	Drive used for processing. Values below zero mean no valid drive is active.
driveprc	Pointer to the drive currently processed by the emulator.

drive	Pointer to the host-supplied CapsDrive array. The array must contain at least drivecnt entries and remain valid while the FDC state refers to it.
cbirq	IRQ line change callback. The emulator calls this hook when the emulated IRQ output line changes. The function pointer must remain valid while the FDC state can call it.
cbdrq	DRQ line change callback. The emulator calls this hook when the emulated DRQ output line changes. The function pointer must remain valid while the FDC state can call it.
cbtrk	Track-change callback. It provides track data for the selected drive, normally by locking an image track or generating formatted data, then copying trackbuf, timebuf, tracklen, and overlap to the selected CapsDrive. The function pointer must remain valid while the FDC state can call it.
userptr	Host application pointer preserved by the library. Use it to associate emulator or callback state with this structure.
userdata	Host application data word preserved by the library. Use it to associate small caller state with this structure.

Related Functions

CAPSFdcInit, CAPSFdcReset, CAPSFdcEmulate, CAPSFdcRead, CAPSFdcWrite

FDC Models

FDC model values stored in CapsFdc.model before CAPSFdcInit.

Values

cfdcNA	Invalid FDC model.
cfdcWD1772	WD1772 FDC model. Select this model before CAPSFdcInit to use the WD1772-compatible emulator.

FDC Output Line Flags

Bits used in CapsFdc.lineout to expose emulated FDC output lines such as DRQ, INTRQ, forced interrupt, motor state, direction, command-in-progress and DRQ-set state.

Values

CAPSFDC_LO_DRQ	DRQ line state. The emulator sets this bit when the data request output line is asserted and calls the DRQ callback on line changes.
CAPSFDC_LO_INTRQ	INTRQ line state. The emulator sets this bit when the interrupt request output line is asserted and calls the IRQ callback on line changes.
CAPSFDC_LO_INTFRC	Forced interrupt, internal. When set, line processing forces INTRQ to the asserted state.
CAPSFDC_LO_MO	Motor line state. This bit mirrors the emulated controller motor output.
CAPSFDC_LO_DIRC	Direction line state. This bit indicates the current head step direction.
CAPSFDC_LO_INTIP	Interrupt on index pulse. The emulator uses this bit to request an INTRQ pulse when the next index pulse is detected.
CAPSFDC_LO_DRQSET	Set DRQ, internal. The emulator uses this transitional bit before promoting it to the visible CAPSFDC_LO_DRQ line state.

FDC Status Flags and Masks

Status bits and command masks used by the FDC emulator. The SR values mirror WD1772-style status register bits, while the mask values describe which bits are cleared, set or selected for different command classes.

Values

CAPSFDC_SR_BUSY	Busy status bit. Set while the FDC is executing a command.
CAPSFDC_SR_IP_DRQ	Index pulse or DRQ status bit, depending on command type. Type 1 commands use it for index pulse; read/write commands use it for data request state.
CAPSFDC_SR_TR0_LD	Track 0 or lost data status bit, depending on command type. Type 1 commands use it for track zero; read/write commands use it for lost data.
CAPSFDC_SR_CRCERR	CRC error status bit. Set when the active command detects a CRC mismatch in address or data fields.
CAPSFDC_SR_RNF	Record-not-found status bit. Set when the requested sector, address mark or record cannot be found before the command limit.
CAPSFDC_SR_SU_RT	Spin-up or record type status bit, depending on command type. Type 1 commands use it for spin-up; read commands use it for record type.
CAPSFDC_SR_WP	Write-protect status bit. Set when the selected drive is write protected or no writable disk is inserted.
CAPSFDC_SR_MO	Motor-on status bit. Mirrors the active motor state in the status register.
CAPSFDC_SR_NCCLR	Status flags cleared before a new command. The emulator clears these bits when command execution begins.
CAPSFDC_SR_NCSET	Status flags set before a new command. The emulator sets these bits when command execution begins.
CAPSFDC_SM_TYPE1	Type 1 status mask for index, track 0 and spin-up fields. It selects which secondary status fields are visible for type 1 commands.
CAPSFDC_SM_TYPE2R	Type 2 read status mask. It selects DRQ, lost data and record type fields for read commands while preserving the command-specific interpretation of status bits.
CAPSFDC_SM_TYPE2W	Type 2 write status mask. It selects DRQ, lost data and record type fields for write commands while preserving the command-specific interpretation of status bits.

FDC End Request Flags

Bits used in CapsFdc.endrequest to indicate command completion or to request that execution stops at the current emulation cycle.

Values

CAPSFDC_ER_COMEND	Command complete. The emulator sets this end-request bit when the active command has reached its completion condition.
CAPSFDC_ER_REQEND	Request to stop execution at the current cycle. Callback code may bitwise set this flag when it needs CAPSFdcEmulate to return promptly.

FDC AM Info Flags

Bits used in CapsFdc.aminfo for address mark detection, CRC handling and DSR synchronization state. These values expose whether address-mark detection and CRC logic are enabled or active, whether A1/C2 marks were detected, and whether DSR contains a complete byte or mark-synchronized byte.

Values

CAPSFDC_AI_AMDETENABLE	Address-mark detector enabled. When set, the emulator scans incoming bitcells for address marks.
CAPSFDC_AI_CRCENABLE	CRC logic enabled. When set, detected marks and data can feed the CRC accumulator.
CAPSFDC_AI_CRCACTIVE	CRC logic active after the first relevant mark has been found.
CAPSFDC_AI_AMACTIVE	Data being assembled in the data shift register is the last byte of an address mark.
CAPSFDC_AI_MA1ACTIVE	Data being assembled in the data shift register is an A1 mark.
CAPSFDC_AI_AMFOUND	Address mark found in the decoder, valid for one cell.
CAPSFDC_AI_MARKA1	A1 mark found in the decoder, valid for one cell.
CAPSFDC_AI_MARKC2	C2 mark found in the decoder, valid for one cell.
CAPSFDC_AI_DSRREADY	The data shift register holds a complete byte. The flag remains valid until the data shift register changes.
CAPSFDC_AI_DSRAM	The data shift register is synchronized to an address mark. The flag remains valid until the register changes; the next DSRREADY indicates a data byte.
CAPSFDC_AI_DSRMA1	The data shift register is synchronized to an A1 mark.

FDC Run Modes

Internal FDC run mode values exposed in CapsFdc.runmode.

Values

cfdrmNop	No-operation run mode.
cfdrmIdle	Idle or wait-loop run mode.
cfdrmType1	Type 1 command loop, used for restore, seek and step-style commands.
cfdrmType2R	Type 2 read command loop, used for sector read commands.
cfdrmType2W	Type 2 write command loop, used for sector write commands.
cfdrmType3R	Type 3 read command loop, used for read-track style commands.
cfdrmType3W	Type 3 write command loop, used for write-track style commands.
cfdrmType3A	Type 3 address command loop, used for read-address style commands.
cfdrmType4	Type 4 command loop, used for force-interrupt handling.

Remarks

These values are exposed by the public structure but are normally controlled by the emulator. Host applications should not need to set them directly.

FDC Data Modes

Internal FDC data access mode values exposed in CapsFdc.datamode.

Values

cfdcdmNoline	No line input. The data separator is not currently reading a usable data line.
cfdcdmNoise	Noise input. The emulator generates noisy bit input rather than reading stable encoded data.
cfdcdmData	Data input. The emulator reads ordinary track data without a density map.
cfdcdmDMap	Data input with a density map. The emulator uses timing data while reading variable-density track data.

Remarks

These values are exposed by the public structure but are normally controlled by the emulator. Host applications should not need to set them directly.

CapsFormatBlock

CapsFormatBlock is a formatter block descriptor consumed by CAPSFormatDataToMFM. Each entry describes the gaps, address fields, sector identity and source data used to generate an index or data block in an MFM track.

```
struct CapsFormatBlock {
uint32_t gapacnt;
uint32_t gapavalue;
uint32_t gapbcnt;
uint32_t gapbvalue;
uint32_t gapccnt;
uint32_t gapcvalue;
uint32_t gapdcnt;
uint32_t gapdvalue;
uint32_t blocktype;
uint32_t track;
uint32_t side;
uint32_t sector;
int32_t sectorlen;
uint8_t *databuf;
uint32_t datavalue;
};
typedef struct CapsFormatBlock *PCAPSFORMATBLOCK;
```

Members

gapacnt	Count of bytes in the gap before the first address mark generated by CAPSFormatDataToMFM.
gapavalue	Byte value used for the gap before the first address mark generated by CAPSFormatDataToMFM.
gapbcnt	Count of bytes in the gap after the first address mark generated by CAPSFormatDataToMFM.
gapbvalue	Byte value used for the gap after the first address mark generated by CAPSFormatDataToMFM.
gapccnt	Count of bytes in the gap before the second address mark generated by CAPSFormatDataToMFM.

gapcvalue	Byte value used for the gap before the second address mark generated by CAPSFormatDataToMFM.
gapdcnt	Count of bytes in the gap after the second address mark generated by CAPSFormatDataToMFM.
gapdvalue	Byte value used for the gap after the second address mark generated by CAPSFormatDataToMFM.
blocktype	Formatter block type. cfrmbtData writes a sector header, data field and CRC fields; cfrmbtIndex writes index address-mark data.
track	Track number written into generated sector ID fields.
side	Side number written into generated sector ID fields.
sector	Sector number written into generated sector ID fields.
sectorlen	Sector length in bytes. CAPSFormatDataToMFM accepts only valid FDC sector sizes that can be represented in the generated sector header.
databuf	Source data buffer for cfrmbtData sector contents. If it is null, CAPSFormatDataToMFM fills the sector data field with datavalue.
datavalue	Source data value used to fill a cfrmbtData sector when databuf is null.

Related Functions

CAPSFormatDataToMFM

Format Block Types

Block type values stored in CapsFormatBlock.blocktype.

Values

cfmibtNA	Invalid formatter block type.
cfmibtIndex	Index block. CAPSFormatDataToMFM writes index address-mark data for this block type.
cfmibtData	Data block. CAPSFormatDataToMFM writes a sector header, data field and CRC fields for this block type.

CapsFormatTrack

CapsFormatTrack is the formatter track descriptor passed to CAPSFormatDataToMFM. It supplies the output track buffer, index-gap values, start position and ordered CapsFormatBlock array; if trackbuf, tracklen or buflen is zero, the function reports the required size in bufreq.

```
struct CapsFormatTrack {  
    uint32_t type;  
    uint32_t gapacnt;  
    uint32_t gapavalue;  
    uint32_t gapbvalue;  
    uint8_t *trackbuf;  
    uint32_t tracklen;  
    uint32_t buflen;  
    uint32_t bufreq;  
    uint32_t startpos;  
    int32_t blockcnt;  
    PCAPSFORMATBLOCK block;  
    uint32_t size;  
};  
typedef struct CapsFormatTrack *PCAPSFORMATTRACK;
```

Members

type	Structure type. Set to 0 when DI_LOCK_TYPE is used with the current CapsFormatTrack layout; unsupported type requests return imgeUnsupportedType.
gapacnt	Count of bytes in the gap after the index area when generating a formatted MFM track.
gapavalue	Byte value used in the gap after the index area when generating a formatted MFM track.
gapbvalue	Byte value used in the gap before the next index area when generating a formatted MFM track.
trackbuf	Track buffer memory used as the destination for generated MFM data. If trackbuf, tracklen or buflen is zero, CAPSFormatDataToMFM only calculates the required buffer size in bufreq.

tracklen	Track buffer memory length used by the formatter. CAPSFormatDataToMFM requires tracklen to fit within buflen and startpos to be inside tracklen.
buflen	Track buffer allocation size; tracklen must not exceed it.
bufreq	Track buffer requested size filled by CAPSFormatDataToMFM. Use this value to allocate a sufficiently large trackbuf before generating data.
startpos	Start position in the circular track buffer. It must be less than tracklen and is used as the formatter wrap/overlap position.
blockcnt	Number of block descriptors in the block array.
block	Pointer to an array of CapsFormatBlock descriptors. CAPSFormatDataToMFM processes blockcnt entries in order.
size	Converter counter used while generating the track. Initialise the structure before use so this working value starts from a known state.

Remarks

If trackbuf, tracklen or buflen is zero, CAPSFormatDataToMFM calculates the required buffer size in bufreq.

If tracklen is smaller than bufreq, CAPSFormatDataToMFM returns imgeBufferShort. If startpos is outside the supplied track buffer, it returns imgeBadDataStart.

For cfrmbtData blocks, provide one CapsFormatBlock entry per sector and point databuf to that sector's data. The generated track buffer can be used as CapsDrive track data for FDC emulation.

Related Functions

CAPSFormatDataToMFM