CAPS Library and API Documentation

Revision 1.02

Table of Contents

Revision History

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 **CAPS** *support library* allows various programs to access ***I****nterchangeable* ***P****reservation* ***F****ormat* – **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, CD, tape images and so on.

Of CAPS and Men

CAPS makes it possible to archive floppy disks with all their content intact and functioning as expected on the real media. This allows copy protected media to be archived to virtual media - that is, files that are not subject to transfer and archiving limitations and the very limited lifetime of the original media - representing the original one and working perfectly.

Apart from preservation, it also has the nice side effect of making the preserved material fully workable without alteration in media specific players, such as playing original ancient games and programs - i.e. not "cracks" or "warez" - with computer emulators.

There are some simplistic solutions to this, but they do not guarantee or are not concerned with the integrity of the data, as e.g. in the world of classic games hundreds of disk formats had been used. Without proper analysation their content and layout cannot be determined, and as a consequence of that, their integrity cannot be checked.

Think of it as taking your most valuable moments of life on photos with a non-digital camera to film, but never having the time to develop them. Your descendants finally try to do it, and are shocked to discover most of the photos are badly taken, under exposed or blank, as you were not very good at doing them in the first place. Important memories of your family are lost forever; those moments never ever come back.

Now imagine an important part of digital history being lost forever the exact same way; while loosing your family memories can be sad, loosing the documents of say a whole era that is of interest to upcoming generations, historians, scientists, researchers or just those interested in the past is devastating.

While archiving "blindly", say by copying a VHS tape, you may get blurry picture, and faded colours. In digital data that is not designed to be redundant - such as the data stored on floppy disks - the consequences of losing content means losing non-redundant information, such as text, program code, video and audio data. On solid compressed data - such as levels of games - missing information renders the whole media unusable, causes memory or other corruptions as the data was expected to work in the first place. **CAPS** is also concerned with the authenticity of the data analysed, by using fingerprints of the signals read.

This is like seeing someone painting a few red spots into a book. We are looking for different fingerprints of signals on the disks, making it possible to differentiate authentic data, and data that was altered later, such as save positions or high scores saved to disk, or worse, altered program code, graphics sound or other information. To stick with our example: some people would find the modified art "cool", but you'd have a very hard time finding a librarian, gallery or museum considering it a worthwhile representation of the subject.

If that was the only painting available - likely - they'd have no choice, but keep it with other alterations (and write you into their blacklist of vandals).

If that was a very valuable book, chances are that somewhere there is still another copy exists without your marks, and they will try and do everything to locate and preserve that copy of the book, as in its unaltered form invaluable for the future generations to come.

Likewise we try and locate unaltered versions of digital media, not for the now - for the future.

User Manual

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 **CAPS** site:

www.caps-project.org

Should you have problems with the usage of some software supplied in **IPF** format, such as a program capable of using **CAPS IPF** files, and the image used is not recognized by the application, we advise to download and install 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 **CAPS**. Please consult with the authors and companies responsible for such products directly. Inquires like how to make game X work with emulator Y using the **IPF** file Z should be directly aimed to the product suppliers. In the CAPS database there are hints however that can help identifying the required settings for emulation.

Installation

Applications taking advantage of the functionality may require some preparation to get everything working properly.

Obtaining the latest library version

As a first step download the latest library from the download section at:

www.caps-project.org

Since the file is very small it is possible that it is being cached by proxy and/or browser software and an older version is being "downloaded" rather than the one offered on the site.

In this case we recommend cleaning the caches involved; please consult your browser and proxy system documentation for more details.

Deleting existing copies

In case you have multiple copies of the library installed on your system, it is recommended that you delete all of them before making use of a new version.

Some operating systems may cache the libraries used since the last boot, therefore it is a good idea to reboot your system after the removal of the old library.

Please note, that in case of library caching, re-booting is mandatory as just re-logging in may still use the same system components as before.

If you can't delete or replace the library, it is being used by an application or service.

Exit applications and/or stop the services known to be using it, until it can be deleted or replaced. Alternatively re-boot your system.

Installing the library

Copy the downloaded library to a place recognised by the operating system as a library access location. Use only one location, so later updates should only apply to that file.

Depending on the operating system used the location varies and some operating systems require that you have administrative privileges in order to add or change a library.

Please consult with your operating system manuals for more details.

Windows

Under Windows based operating systems if you are using the library only for one application – e.g. an emulator - normally you can place the library to the same path as the application executable as that location is searched for additional libraries by the system loader.

Note that if you are placing the library on a search path specific for the application, other applications will not be able to use it – it has to be placed into a generic library access path to be accessible by any programs. Such locations are [windows]/system32, [windows]/system, [windows] etc. [windows] is the path where your OS is installed.

The library is only tested and supported on Windows NT kernel based systems. Although it may work correctly on Windows98 or earlier Win32.implementations or Win32 API/OS emulators/layers – none of them is tested or supported officially.

Linux

You can install the library as: cp libcapsimage.so.1.1 /usr/lib/ and running ldconfig

Currently, the internal version string (the soname) is libcapsimage.so.1

ldconfig automatically creates symbolic links from installed libraries to their sonames.
Manual linking: ln -s libcapsimage.so.1.1 libcapsimage.so.1

Updates of libcapsimage can be installed simply by copying libcapsimage.so.1.2 (or whatever the file is named) to /usr/lib and running ldconfig again - or manually changing the symbolic link:

ln -s /usr/lib/libcapsimage.so.1.X /usr/lib/libcapsimage.so.1

AmigaOS, classic68k

The default directory used for:

- Devices: **Devs**:
- Libraries: **Libs**:

The normal search path for such items, such as the application executable location should work as well, but other applications will not be able to open the device if the location is not in their search path.

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.

You have been warned.

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 ancient M68k CPUs.

Interface

The library interface is fairly 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 data.

Emulation

You may want to implement aligning the data as the 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 format 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 with 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).

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 **R**evolutions **P**er **M**inute (**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.

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 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 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 actually 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.

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.

Always use the drive parameters set for the 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 as "*half-tracks*". They may hold protection or other data.

Using the API

Compiling and Linking

The example files supplied with the developer library downloads should give a good overview on how to use the header files and compile simple projects under MSCVC6 or other MSVC plug-in compilers like Intel's compiler on Win32, or GCC on different platforms. Register usage of parameters passed to the library is documented through the associated header files where applicable, namely the Amiga platform for those who wish to call the API functions from assembly.

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 is capable of calling library functions in C style.

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.

The library must be dynamically linked or opened – i.e. runtime, not compile time – by any application.

Static, compile time linking where the object code of the library is merged with the application is not allowed.

This way whenever a new library is released the user of the application can instantly use the new release without having to wait for upgrades of the applications.

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.

Generally although it should be safe, though 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.

Generally speaking you should normally be able to open the same IPF file several times, but it is a good idea to check the error codes returned.

Error Handling

Unless otherwise stated, after successful completion of any API call *imgeOk* 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.

Structures passed to a function that fills the structure are cleared upon entering the function therefore placing safe values into them before API calls is not needed. If an error occurs during the execution of such function the structure may only be partially filled or the entire content invalid. Therefore data returned by functions emitting errors should not be used, or only used with caution testing elements against 0 or NULL.

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*.

AmigaOS specific

For the Amiga environment the library is supplied as a standard Amiga device, hence it must be opened with exec.library/OpenDevice() first. Afterwards, the base address of the capsimage library must be obtained in order to use the capsimage functions.

The library should be opened as a valid device first, using the library functions afterwards, and finally freeing the resources allocated for the device.

CAPSInit should only be called after successfully allocating the resources and opening the device; *CAPSExit* must be called before closing the device and freeing the resources allocated.

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.

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.

API Reference

Functions

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 stuck with C compilers or projects.

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 goodies available for better typing and data isolation could not be used in the public interface of the library.

Constants, Data Types and Definitions

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

Data types and naming may vary from platform to platform, but their functionality remains the same.

Just use the correct data types to be found in the header files supplied with the platform specific libraries, they resolve to the correct alignment, packing and size requirements of the library internals.

Locking Flags

Flags are bit values combined using the bitwise OR operation to provide the various attributes desired for locking the images.

E.g. in C style: (DI_LOCK_INDEX| DI_LOCK_ALIGN)

If no *DI_LOCK_DEN…*flag is given, the library does not generate a density map for the track.

Note that using 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 -, however data generated such a way may be useful for other applications, like **IPF** file browsers or virtual drives with simpler disk logic implemented. One particularly helpful flag is DI_LOCK_INDEX for implementing simple disk logic.

DI_LOCK_INDEX	Track data is re-aligned in the buffer as if it was index synced recording originally, starting at the beginning of the buffer. Normally track data decoded is properly positioned as was found on the original disk, starting at any position or distance from the disk index. Setting the flag results in a track differently positioned therefore data differently timed from the original.
DI_LOCK_ALIGN	The decoded track data is aligned to be word – 16 bits - size. If unset buffer lengths of odd bytes can be returned for locked tracks, as is on the disk originally. Setting the flag may result in a track of a slightly different size than the original.
DI_LOCK_DENVAR	Cell density map is generated for a variable speed track, like Copylock or Speedlock protection tracks. Normally only variable density tracks should be prompted for a cell density map in order to save on memory and enhance application performance. Other cell densities might be generated and processed faster by programming workarounds.
DI_LOCK_DENAUTO	Cell density map is generated for a constant speed track
DI_LOCK_DENNOISE	Cell density map is generated for an unformatted track
DI_LOCK_NOISE	An unformatted track is algorithmically filled with “noise patterns” if set, otherwise no buffer is allocated for it.
DI_LOCK_NOISEREV	An unformatted track is algorithmically filled with “noise patterns” if set, otherwise no buffer is allocated for it. The returned buffer will contain multiple revolutions of

	different data.
DI_LOCK_MEMREF	Only used by locking functions that accept a memory reference as a parameter, like *CAPSLockImageMemory*. If set, the library uses the buffer supplied by the caller of the function, until the image is unlocked directly or indirectly. The program should not free the buffer supplied with the locking call as long as the lock is valid - i.e. not unlocked in any way direct or indirect. If the flag is clear the library allocates a private data buffer and copies the content of the supplied buffer to its private data area. The program can free the buffer given after the called function returns; the private data area allocated is automatically freed once the lock is deleted.

CapsDateTimeExt

The structure is used to retrieve date/time information from the packed format used by the **IPF** files.

```
struct CapsDateTimeExt {
        UDWORD year;
        UDWORD month;
        UDWORD day;
        UDWORD hour;
        UDWORD min;
        UDWORD sec;
        UDWORD tick;
};
```

```
typedef struct CapsDateTimeExt *PCAPSDATETIMEEXT;
```

Members:

Year	Year value
Month	Month value
Day	Day value
Hour	Hour value
Min	Minutes value
Sec	Seconds value
Tick	Ticks value (counter within a second, OS dependent)

CapsImageInfo

The structure is used to retrieve generic information about the **IPF** image.

Tracks are addressed as cylinder.head in functions. E.g. the data on double-sided disk for Amiga or AtariST image usually starts on 0.0 and ends on 79.1, though it is likely that the image contains more tracks than those. Never assume these values, always use the values from this structure.

```
struct CapsImageInfo {
        UDWORD type;
        UDWORD release;
        UDWORD revision;
        UDWORD mincylinder;
        UDWORD maxcylinder;
        UDWORD minhead;
        UDWORD maxhead;
        struct CapsDateTimeExt crdt;
        UDWORD platform[CAPS_MAXPLATFORM]; // intended platform(s)
};

typedef struct CapsImageInfo *PCAPSIMAGEINFO;
```

Members:

Type	Image type
Release	Release ID. Each IPF file has a unique release ID by design that can be used as a unique key for database, holding arbitrary information about the file or files belonging to the same release. More than one file can have the same ID if all of them belong to the same group of files, like games that are supplied on more than one disk. ID 0 is invalid, only used by test images.
Revision	Revision of the file. It is possible that more than one revision - having higher revision numbers than 1 - of the very same file exists. Later revisions always replace older ones. Normally the revision number is 1.

	0 is used for test images.
Mincylinder	Lowest cylinder number valid for the image when calling track related functions. (inclusive)
Maxcylinder	Highest cylinder number valid for the image when calling track related functions. (inclusive)
Minhead	Lowest head number valid for the image when calling track related functions.
Maxhead	Highest head number valid for the image when calling track related functions.
Crdt	The creation date and time of the **IPF** image
Platform[]	This array contains up to CAPS_MAXPLATFORM number of valid entries of intended platforms the image is naturally linked with. I.e. it is possible to read an Atari disk using an Amiga, but this is not intended use, therefore only an Atari entry will be found. Dual or tri-format disks will have more than one valid entry. Invalid entries are set to *ciipNA*

CapsImageInfo.type

The values allowed for the type member of CapsImageInfo

There may be additional types available with upcoming library versions.

```
enum {
	ciitNA=0,
	ciitFDD
};
```

Values:

CiitNA	Invalid image type
CiitFDD	Floppy disk

CapsImageInfo.platform

The values allowed for the platform member of CapsImageInfo

There may be additional platforms available with upcoming library versions.

```
enum {
        ciipNA=0,    // invalid platform (dummy entry)
        ciipAmiga,   // Amiga
        ciipAtariST, // Atari ST
        ciipPC       // PC
};
```

Values:

CiipNA	Invalid platform (dummy entry, skip it)
CiipAmiga	Amiga
CiipAtariST	Atari ST
CiipPC	PC

CapsTrackInfo

The structure is used to retrieve generic information about a track of a disk image from an **IPF** file.

```
struct CapsTrackInfo {
        UDWORD type;
        UDWORD cylinder;
        UDWORD head;
        UDWORD sectorcnt;
        UDWORD sectorsize;
        UDWORD trackcnt;
        PUBYTE trackbuf
        UDWORD tracklen;
        PUBYTE trackdata[CAPS_MTRS];
        UDWORD tracksize[CAPS_MTRS];
        UDWORD timelen;
        PUDWORD timebuf;
};

typedef struct CapsTrackInfo *PCAPSTRACKINFO;
```

Members:

Type	Track type
Cylinder	Cylinder position where the track is located
Head	Head position where the track is located
Sectorcnt	Number of sectors (blocks) used by the track
Sectorsize	Size of the sectors, normally meaningless since sectors can have any sizes within a track
Trackcnt	The number of revolutions (data tracks) decoded for this track. Unless something went wrong or the track has been freed this should be normally 1 for a locked track, and any value greater than 1 for multi-revolution tracks. For more information see the relevant topic.

Trackbuf	Pointer to the buffer of decoded track data. It always points to the first buffer if more than one present for multi-revolution tracks. If the track is unlocked or something went wrong during decoding this pointer can be NULL.
Tracklen	The length of the data buffer. It is the sum of all track length values and should not be used to determine the length of just one track revolution. Use tracksize[] for that purpose.
Trackdata[]	This array contains up to CAPS_MTRS number of valid entries of track buffer pointers as decoded by the library. The number of valid entries can be obtained from *trackcnt*. The first valid entry is always Trackdata[0] If the track is unlocked or something went wrong during decoding a pointer can be NULL.
Tracksize[]	This array contains up to CAPS_MTRS number of valid entries of data buffer sizes in bytes as decoded by the library. The number of valid entries can be obtained from *trackcnt*. The first valid entry is always Tracksize[0] If the track is unlocked or something went wrong during decoding values can be 0.
Timelen	The number of entries present in the cell density map. Can be 0 if the cell density map is not requested or not available for a track using its current locking.
Timebuf	Pointer to the buffer where the cell density map is stored. Can be NULL if the cell density map is not requested or not available for a track using its current locking.

CapsTrackInfo.type

The values allowed for the type member of CapsTrackInfo

There may be additional types available with upcoming library versions.

```
enum {
        ctitNA=0,
        ctitNoise,
        ctitAuto,  // automatic cell size, according to track size
        ctitVar    // variable density
};
```

Values:

CtitNA	Invalid type
CtitNoise	The track is not formatted; the cells should be random sized. Note that the data or density for unformatted tracks is not generated unless requested when locking the track. All the pointers and values for tracks not generated will be NULL or 0.
CtitAuto	Each cell group has the same width/timing. The cell density can be generated by evenly distributing the cell groups for the amount of time reading a track takes. Note that density for these tracks is not generated unless requested when locking the track. All the pointers and values for empty density maps are NULL or 0.
CtitVar	The track contains cell groups with variable density. The library must generate the cell density map; it will contain the values explained in the relevant section. Note that density for these tracks is not generated unless requested when locking the track. All the pointers and values for empty density maps are NULL or 0.

Error Codes

Generally each function returns with an *imgeOk* value upon successful completion, unless otherwise stated.

```
enum {
        imgeOk,
        imgeUnsupported,
        imgeGeneric,
        imgeOutOfRange,
        imgeReadOnly,
        imgeOpen,
        imgeType,
        imgeShort,
        imgeTrackHeader,
        imgeTrackStream,
        imgeTrackData,
        imgeDensityHeader,
        imgeDensityStream,
        imgeDensityData,
        imgeIncompatible
};
```

Values:

ImgeOk	Function completed
ImgeUnsupported	The requested function is not supported on the image
ImgeGeneric	Generic problem while executing the function such as bad parameters
ImgeOutOfRange	Data supplied to a function call is not within valid range, or the data referred to is invalidated.
ImgeReadOnly	Can not write to an image (not used)
ImgeOpen	There is a problem opening the image
ImgeType	Not an **IPF** image

ImgeShort	The image is shorter than the data expected, could be a programming problem when using memory buffers for locking an image.
ImgeTrackHeader	Problem with the data area of the track
ImgeTrackStream	Problem with the data area of the track
ImgeTrackData	Problem with the data area of the track
ImgeDensityHeader	Problem with the density area of the track
ImgeDensityStream	Problem with the density area of the track
ImgeDensityData	Problem with the density area of the track
ImgeIncompatible	The image contains data that cannot be decoded by the library used.

CAPSInit

The function initialises the library internals. It must be called before any other calls are made.

```
SDWORD CAPSInit();
```

Parameters

-

Return Values

imgeOk if successful.

Remarks

The program should attempt no further library calls if the function does not succeed, however *CAPSExit* must be called in order to free resources that might have been allocated during the call.

CAPSExit

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

SDWORD CAPSExit();

Parameters

-

Return Values

imgeOk if successful.

Remarks

The program should not attempt any library calls after calling the function if the function succeeds.

CAPSAddImage

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

```
SDWORD CAPSAddImage();
```

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 result ID should not be used in further calls and *CAPSRemImage* should not be called with the ID.

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

CAPSRemImage

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

```
SDWORD CAPSRemImage(
        SDWORD id                     // container ID
);
```

Parameters

- Id: [in] the container ID returned by *CAPSAddImage*.

Return Values

The supplied container ID if successful, otherwise a negative value.

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.

CAPSLockImage

The function locks an IPF image into a container device.

```
SDWORD CAPSLockImage(
        SDWORD id,                  // container ID
        PCHAR name                  // filename
);
```

Parameters

- Id: [in] the container ID returned by *CAPSAddImage*.
- Name: [in] the name of the **IPF** file to be opened

Return Values

imgeOk if successful or related *imge* error code.

Remarks

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

CAPSLockImageMemory

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

```
SDWORD CAPSLockImageMemory (
        SDWORD id,                  // container ID
        PUBYTE buffer,              // memory buffer
        UDWORD length,              // buffer length
        UDWORD flag                 // locking flags
);
```

Parameters

- Id: [in] the container ID returned by *CAPSAddImage*.
- Buffer: [in] pointer to the buffer area where the **IPF** image in memory starts
- Length: [in] length of the supplied buffer. It must be the same with the size of the **IPF** image in file format.
- Flag: [in] only one flag is supported, DI_LOCK_MEMREF.

Return Values

imgeOk if successful or related *imge* error code.

Remarks

This function is useful for retrieving images from archive files by first decompressing the IPF 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 – *CAPSUnlockImage* has no effect on it.

CAPSUnlockImage

The function unlocks – “ejects” - an IPF image from a container device.

```
SDWORD CAPSUnlockImage (
        SDWORD 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 **IPF** file is unlocked (if a file was locked originally) once the function completes.

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.

```
SDWORD CAPSLoadImage (
        SDWORD id,                      // container ID
        UDWORD 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.

CAPSGetImageInfo

The function read image the image information data from a locked **IPF** file.

```
SDWORD CAPSGetImageInfo (
        PCAPSIMAGEINFO pi,          // pointer to CapsImageInfo
        SDWORD 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

-

CAPSLockTrack

The function locks – reads and decodes – a track from a locked **IPF** file.

```
SDWORD CAPSLockTrack (
        PCAPSTRACKINFO pi,          // pointer to CapsTrackInfo
        SDWORD id,                  // container ID
        UDWORD cylinder,            // cylinder to read
        UDWORD head,                // head to read
        UDWORD flag                 // locking flags
);
```

Parameters

- Pi: [out] pointer to the CapsTrackInfo 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

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

CAPSUnlockTrack

The function unlocks – frees all the resources allocated – a track from the buffers associated with a locked **IPF** file.

```
SDWORD CAPSUnlockTrack (
        SDWORD id,                  // container ID
        UDWORD cylinder,            // cylinder to read
        UDWORD head,                // head to read
);
```

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.

CAPSUnlockAllTracks

The function unlocks – frees all the resources allocated – all tracks from the buffers associated with a locked **IPF** file.

```
SDWORD CAPSUnlockAllTracks (
        SDWORD id,                  // container ID
);
```

Parameters

- Id: [in] the container ID returned by *CAPSAddImage*.

Return Values

imgeOk if successful or related *imge* error code.

Remarks

-

CAPSGetPlatformName

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

```
PCHAR CAPSGetPlatformName (
        UDWORD 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 NULL for an invalid platform ID.

Remarks

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