cubic player development kit documentation

version 0.5
for version 2.0α

Preface

Since cubic player has become quite well known, the mail support team including me has received tons of letters asking "couldn't you do/add ... in the next version?". The point is, that I simply don't have the time (nor the interest) to do so. A common request is the wish to have a new format or a new soundcard supported (yes... also old formats and old soundcards... :)). If I want to support a new soundcard, I need it for testing. If I want to support a new format, I need time for reserch and get still tons of bugreports, because I did something wrong. This cannot continue. One thing I could have done, is to release the complete source code, but there'd be too many disadvantages... I came to the conclusion that a DLL system is the solution. As many "modern" Operating Systems use that method, it can't be too wrong. You've seen the first part of it in releases 1.666 and 1.7 and there have been some drivers for new module types and soundcards since then (Which is another advantage: small add-ons rather than many new versions). I've worked on a concept to make add-ons possible on many different levels and finally split CP up into many (about 50) DLLs and one EXE. The EXE now only contains an INI reader, DLL linker and several basic functions. The rest is distributed among the DLLs. I hope that someone will do something for cp, and that I didn't write all this stuff for nothing.
If you decide to code something this documentation and the example sources will help you. As this is the first release of the CPDK I don't know about your needs. If you have problems, think twice, try, try again, think again, try again, and then ask me (mail business is quite time consuming, you know...). Study the examples carefully, I tried to include an example for everything you can do, you'll get more information from the examples than from this (very unfinished) documentation. Many things are not even documented here, so look at the example source that does something similar to what you want to do.
If you have done something you must decide what to do with it. You can distribute it without any restrictions, of course. You could also spread the source to help others. If you want your add-on to be distributed along with cubic player send it to me. Give me the source, if you like me to keep your code up to date (Only an option, I won't promise anything). Tell me your conditions as I might want to sell licenses for cubic player.
The internal structure for cp (since this is an DK, you can call it API) did change a lot all the time, and I don't expect that to end in the near future. Since this is my first try there might be many things I will have to change (Also because this is all done in a little hurry, I want to have it finished at TP6 (I wanted to have it finished to Wired '96, but that did not work... :( ), and I don't have the time to document everything and find the best solutions for all the problems). Anyway, I hope that the current structure is quite sufficient, try to make your add-ons with the current API, even if they might not work with future versions.
The good thing about CPDK is that you don't have to bother about most things you would have had to if you wanted to do a complete module player. If you want to write support for a new format, just write the loader and maybe a player, and have all the rest for free. No need to write a mixer, soundcard support, user interface, fileselector...

legal stuff

For this piece of software the same rules as for cubic player apply. That is it is forbidden to use this for commecial purposes or spread it through commercial media. You may not modify or reverse engineer anything. Exceptions to these rules are:
You may use the header and library files (*.h, *.lib) that come with the cpdk for every cubic player dll release without restriction. You may use and modify the source (*.cpp, *.asm) for your production, if you make clear, that you used/modified the files. You must leave the unchanged header on top of the file, add your headers

below, and mark every change you made. You may also release modified sources when it is clear that they were modified.

tracker guys

Make support for your format in cubic player and the users might switch to your tracker faster, because there is already a player to play their modules very accurately. Especially in music compos one player for all the modules is very useful, but there are always complaints about badly played modules... And: Who know's the effect processing better than you? I'm really fed up with people complaining about wrong effects!!!!

What you can do

You can change or add the following:

- change the main loop. ie. the fileselector (FORGET IT! I just said you can do it).
- new user interface (there should be no need to do that, unless you want to do something really different like an MPG player).
- new archive reading functions (.lha/lzh someone?).
- new module detection routines for the fileselector (preferrably in connection with a player/interface for it).
- new players. make a player using the standard interface, supplying a few callback functions to display data.
- new loaders. (uhoh... the internal module format is a little diffucult... :(and undocumented)
- new screens for the standard interface or for your player.
- new soundcard drivers (with keys/menus for speacial features if you like).
- postprocess the quality mixer output (ie. add filter, echos or effects)

- ### *What you need*

All you need to add something to cubic player is a compiler that generates 32 bit code and can be made to generate compatible function entry code (see below) and a linker that can create LE DLLs (LE format is used by Windows VxDs) As cubic player was written in Watcom C++ 10.6 this package is a perfect choice. If you want to do something in assembler I recommend Borland's TASM (I currently use 3.1), but you need a linker like like Watcom WLINK to make the DLLs. I'm not sure about any other languages, because I have never tried anything else on cp. I know that Watcom C++ 10.0 might be incompatible if you use classes, so if you experience crashes use 10.6.

contents

IV non sound programming

IV.A binfile (binfile.h)
IV.A.1 binfile
IV.A.2 examples
IV.A.3 sbinfile
IV.A.4 abinfile
IV.A.5 mbinfile
IV.A.6 binfilecache
IV.A.7 pakbinfile
IV.A.8 delbinfile

IV.B CP.EXE and initialization of components (pmain.h)

*IV.C Linker: *.DLL loader (plinkman.h)*

IV.D Configuration: cp.ini and commandline (psetting.h)

IV.E Fileselector general: (pfilesel.h)

IV.F Archive Reading: (pfilesel.h)

IV.G Interfaces: (pfilesel.h)

IV.H Cubic Player Interface (cpiface.h)
IV.H.1 players
IV.H.2 display modes
IV.H.3 textmode windows

V sound programming

V.A Sampler Devices (sampler.h)

V.B Player Devices (player.h)
V.B.1 creating new player devices

V.C Wavetable Devices (mcp.h)
V.C.1 playing sounds, sending commands, getting information
V.C.2 struct sampleinfo:
V.C.3 using wavetable devices
V.C.4 tick callback function, synchronous playing
V.C.5 asynchronous playing
V.C.6 getting information
V.C.7 dynamic channel allocation
V.C.8 creating new wavetable devices
V.C.9 postprocessing the quality mixer output (devwmixq.h)

VI contact information

VII closing words

programming considerations

If you use Watcom C++ use the register calling convention and don't bother about anything. You can skip this section. If you use TASM use ".386", ".model flat,prolog" and switch /ml for case sensitivity. prolog means that public symbols are just the way you write them. (no underscores, etc). flat is something like the tiny model, just that all offsets are 32bit.

memory layout:

Simple: No segments, only 32 bit offsets for everything. Offsets below 1MB refer to real mode memory.

function calling

- cs,ss,ds and es point to flat memory on entry and exit.
- The direction flag is clear on entry any exit

- Arguments are passed in registers, one 32 bit register each. The order is eax,edx,ebx,ecx. Other arguments are pushed on the stack from right to left. They must be removed from the stack on return.
- The return value is passed eax.
- All registers not used as parameter of return value must be preserved.
- Avoid segment registers. You don't need them in flat mode anyway.
- If you need more information read CGUIDE (watcom c++, "whelp cguide")

- ***naming***

- Variables have an underscore before the name (only assembler)
- Functions have an underscore after the name (only assembler)
- There are strange names: "W?...$..". Those are generated by Watcom C++. I hope you can handle those... :) Otherwise you'd have to change the import names or switch to Watcom C++!

- ***fixups***

In flat mode you need fixups, since the program can be loaded to any point in memory using unknown selectors. The DLL loader in CP.EXE can only handle 32 bit relative and absolute offset fixups. It cannot handle 16 bit fixups, selector fixups or byte fixups. Don't use the following code:
- mov ax,segment (no segment fixups)
- mov ax,[sym+si] (offset of sym as a 16 bit value, always use 32bit addr.)

Watcom C++ code is safe.

style

This section helps you understand the basic structure of cubic player and the way arguments are usually passed. PLEASE READ IT. If you want me to maintain your code, please stick to that style. I know that style cannot be inforced, it's something personal, you've worked for, but then again note that it will help me understand your code and make it easier for me to update your code. A uniform style will only help... My style has also changed in the last 2 years of cp development, so there are also some things left from that time, mostly variable types, I wouldn't use today.
And now to the fun part... ;)

symbols

I try to make as only those functions/variables public, that might be used from the outside. In C everything is public by default. The attribute "static" makes functions private, ie. invisible to the outside. That's mainly because I want to avoid problems with double symbol names. Eg. if you link all the drivers to one file, you will get problems if every driver has it's own private "playnote" visible to everyone.

naming

Most global functions in cp have a lowercase prefix, usually about 3 letters. They determine the part of cp they belong to. Then follows the function name with all first letters as capitals and the others in lowercase. NO underscores. Now some examples:
- mcpOpenPlayer, mcp means MultiChannelPlayer
- mcpGetRealMasterVolume
- cfGetProfileString, configuration

This is also to avoid double symbol names. The same goes for global variable names and constants/enums. Structure names are completely in lowercase, and often have the "struct" suffic. Defines are 100% uppercase with underscores dividing sections (I lately tried to use enums instead). I don't like defines like "_CLAX2I" (taken from interwave ddk). They don't help anyone. "MCP_MAXCHAN" is much better, you know what is meant without much writing. :)
All these conventions are only valid for global symbols, ie. symbols that can possibly be seen by other modules or symbols that apper in header files. For local symbols I normally use lowercase or sometimes the conventions stated above. As long as symbols are abolutely 100% invisible to the outside I'm not strict about naming etc.

function arguments, return values, variables

- Arrays are passed as pointers
- Destination arrays first
- Single structures are passed as references (c++: &). A reference is physically equal to a pointer, but it helps the reader understand, that only one thing is meant, and not an array. It also makes code easier: no dereferencing.
- int as a return type often means status. There are two types of status: The one is 0 / not 0, where 0 means not ok, and not 0 means ok. The other is negative / 0, negative means an error (defined in err.h), -1 is a

general undefined error and 0 means ok. Sometimes a positive return value is the return value and negative values mean an error. Which method is used depends... :(

- Returned zero pointers usually mean an error.
- I hate "NULL"!! 0 is better!
- I also hate "BOOL", "FALSE" and "TRUE". use int, 0 and !0...
- I don't use types like char or short (I've seen they make WC generate stupid code). I use int as long as I don't really mean to use something else. Only if I want to use the limits of a type, I use it. Structures are different, of course.
- If you want to typedef unsigned char for example. use "uchar", not "BYTE" or stuff... I think I really have an uppercasephobia. :)
- Don't load your code with unnecessary variables or calculation steps: numsqr=num*num; numsqr2=numsqr/2; res=numsqr2+1; return res;
- Don't put all the variables at the top of a big function.

- **spacing**
- Look at the example code... Imitate it!
- { on same column as }, 2 columns left of code in the block, and on the same column as the code above. There are only very few exeptions to this rule (structure, array initialization, class inline functions), otherwise I'm very very strict with this rule!
- All code after if,else,while,for... new row, two columns to the right.
- No block with only one instruction, unless required.
- NO TABs!!!! I HATE TABs!!! They only mess my editing up!! Don't let your editor save tabs! Use spaces.
- No space before a comma, usually one space after a comma.

- **closing words on style**

After all, the most important thing is the code, so don't let me bother you with style questions. :)

non sound programming

binfile (binfile.h)

binfile is a basic class for binary files. The main reason why I made binfile is, that there is nothing as easy to use, as small or as useful as binfile in the default c++ libraries. If you want to access files in cubic player, use binfile. If you want to use binfile for other non-commercial projects, just take it, the source is included.

binfile

members of binfile:

mode	protected: file mode
filepos	protected: current position
filelen	protected: length of file
canread	const: file can be read
canwrite	const: file can be written to
canseek	const: file position can be changed
canchsize	const: file size can be changed
getmode()	gets the file mode
binfile	constructor: initializes a file that can do nothing
~binfile	destructor: calls close
close()	closes the file
read(b,n)	reads n bytes to buffer b and returns the number of bytes read
write(b,n)	writes n bytes from buffer b and returns the number of bytes written
seek(p)	sets the current file position to p
chsize(l)	changes the file size to l
seekcur(p)	seek(filepos+p)
seekend(p)	seek(filelen+p)
operator [p]	seek(p)
length()	gets the size of the file
tell()	gets the position
eof()	at end of file?
get...()	read and return a number. postfix: [u:unsigned|s:signed] (c:char|s:short|l:long)

put...(v)	write a number. postfix see get...()
eread(b,n)	like read, but returns 1 if buffer is read correctly, otherwise 0
ewrite(b,n)	like write, but returns 1 if buffer is written correctly, otherwise 0
eget...(s)	like get..., but sets s to 1 if number is read correctly, otherwise 0
eput...()	like put, but returns 1 if number is written correctly, otherwise 0

You might have noticed, that there is no open function, but binfile is only the basic class and does nothing. To use binfile overload close, read, write, seek and chsize and write an open method, all other functions use these. The open method must initialize mode, filepos and filelen, so that the predefined functions can work. close should close the file and call binfile::close. read, write, seek and chsize need not be overloaded, if the file cannot do that action.

examples

- f[100].getc(): get a char from file position 100
- f.putc('M').putc('Z'): write "MZ" to a file
- f.seekend(0): place file pointer to the end of the file

sbinfile

sbinfile is the binfile for normal files. It is unbuffered you should only use it if you read long sections at a time. If you read one byte after another it will get slow. open a file with open(name, mode) where mode is:

openro	open existing file for reading only
openrw	open existing file for reading and writing
opencr	create a file (delete old) for reading and writing
opencrn	create a new file (fail if exists) for reading and writing

I did not need more modes so far...

abinfile

abinfile is a binfile in another binfile (archive). Open a file with open(arc, ofs, len). You can open many files in one archive. The archive must have mode canseek and the abinfile will not have canchsize mode as the length is fixed. abinfile is used to access the files in cp.pak

mbinfile

binfile in memory. open(buf, len, mode) will use the buffer as a virtual file. mode can be openro (read only) or openrw (read and write), and openfree can be added to free the buffer when the file is closed. opencs(buf, len, inc) can be used if the file size will change. buf is a buffer reference, len the length reference and inc the temporary file size increment when the size is increased in small amounts (to avoid many memory allocations)

binfilecache

Buffer for binfiles. Use this if you have many small reads from a file, it will speed it up. (not a mbinfile of course) open(f, len) open a buffer on an open file f with the length len. len should be some k.

pakbinfile

File inside cp.pak or in cp.exe directory. open(name) opens a file

delbinfile

Like sbinfile, but it deletes the file when it is closed (temporary file)

CP.EXE and initialization of components (pmain.h)

CP.EXE provides some basic functions like the configuration and the linking system, those features are described below. There is also the initclose system which initializes the components that can be linked.
The initcloseregstruct has 2 fields: Init and Close. (TADAA! :)) Those are function(pointer)s, which can also be 0 if not used. When an instance of initcloseregstruct is processed, Init is called. If Init worked (returns nonnegative value), the instance will be registered and Close will be called on exit (in opposite order of course: lifo). If Init failed (negative value) cp will call all registered Close functions and exit.

When cp.exe starts, it initializes all native subsystems and then:

- link all DLLs from [general] link=...
- link all DLLs from [<current config>] link=...
- link all DLLs from [<current config>] prelink=...

- register all from [general] initclose=...
- register all from [<current config>] initclose=...
- register all from [general] initcloseafter=...
- call [<current config>] main=...

the main function receives no parameters and returns an int: int fn(); usually this main is provided by the fileselector which will then enter the main loop. This is a point where you can add new features, I don't recommend it... you'd have to rewrite everything at this point. :(It's rather for completenesses sake. If you want to play around or test/try something, you can do it here... maybe just a hello world in cubic player... :)

There are some basic functions:

plDosShell	shell to dos, haha!
plSystem	like c function system, but use this one
conRestore	set 80x25 mode and restore console. do this before a dos shell
conSave	save console. do this after a dos shell

Use cputs to write on the console. that's tested and safe. For fileio use binfile (see above) or the open,close,read,write,lseek,tell,eof,chsize functions.
This is all you need to know about cp.exe and besides there is nothing more in there. :)

*Linker: *.DLL loader (plinkman.h)*

The Link-Manager has 3 functions: lnkLink links a spacelist of DLLs and returns a handle to free them later on with lnkFree. Negative return values indicate an error. lnkLink("dos4gfix poutput inflate pfilesel") will link 4 DLLs for example. To link a DLL all the imported modules and symbols have to be present, otherwise the linking fails, so make sure you load all DLLs in the right order. If there is an entry point, it will be called. DOSDLL.LIB provides a correct entry point and calls __dll_initialize (extern "C" unsigned __dll_initialize()). When the DLL is closed __dll_terminate (same decl.) will be called. You can define those functions to do some initialization, but I recommend you do it with the initcloseregstruct structure (see above). A DLL will never be loaded twice, only a reference counter will be increased.
lnkGetSymbol gives you the address of a symbol or 0 if the symbol is not found. There is not much to say besides this: Only exported symbols can be found, of course!

Configuration: cp.ini and commandline (psetting.h)

.ini files are quite well known, I won't explain them in detail: The structure is:

```
[sec1]
  key1=str1
  key2=str2

[sec2]
  key1=   ; an empty entry does not mean the entry was not defined.
...
```

To make the configuration easier cp reads the commandline into cp.ini.
consider the following commandline:

```
  cp -cCONFIG -s8,m-
```

In the cp.ini you'll then find the following (only virtually...)

```
[commandline]
  c=CONFIG
  s=8,m-

[commandline_c]
  c=onfig ;this is rather stupid, but the parser doesn't know.

[commandline_s]
  8=
  m=-
```

There are currently 6 functions to read cp.ini:

You can not yet modify cp.ini with them... :(

- cfGetProfileString: windows programmers know what it does. Read a string from cp.ini, section (1st par.), and key (2nd). If this string couldn't be found return the 3rd parameter instread. Quite a powerful function! Never modify the return value.
- cfGetProfileInt: Reads an integer from CP.INI, like cfGetProfileString. The 4th parameter is the radix of the number, 10 for dec, 16 for hex.
- cfGetProfileBool: Reads a bool from CP.INI, like cfGetProfileString. The 4th parameter is the return value if the string is empty. Used for commandline processing: If you set -sm the user wants mono to be enabled, and not stereo as by default. To read the switch use mono=cfGetProfileBool("commandline_s", "m", 0, 1); strings like on,off,yes,no,+,-,true,false,1,0 are recognized.
- cfGetProfileString2(a,b,c,d) is like cfGetProfileString(a,c,cfGetProfileString(b,c,d)); This is used for overriding a default configuration. In section b you have a complete configuration and in section a you have only the changes to that configuration.
- cfGetProfileInt2 is like cfGetProfileString2
- cfGetProfileBool2 is like cfGetProfileString2

Example:
cp.ini: [sec] key=str
cfGetProfileString("sec", "key", "xxx") returns "str"
cfGetProfileString("sec", "key2", "xxx") returns "xxx"

The following 2 funtions can help you read lists from cp.ini:

- cfCountSpaceList counts the entries in a spacelist (a list divided by spaces) that are shorter than the given value
- cfGetSpaceListEntry copies the next string from the spacelist that is shorter than the given value to the buffer (1st) and advances the spacelist pointer (2nd) to the next entry.

There are some global variables:

- cfConfigSec: the section given by commandline parameter -cXXX use this with cfGetProfileString2(cfConfigSec, "defaultconfig", ...);
- cfSoundSec, cfScreenSec: sections stated in section cfConfigSec.
- cfCommandLine: the commandline
- cfDataDir: directory of cp.exe or [general] datadir=
- cfTempDir: getenv("TEMP") or [general] tempdir=
- cfConfigDir: directory of cp.ini. you can write to this path.

- ***Fileselector general: (pfilesel.h)***

Filenames are 12 byte arrays.
- fsConvFileName12 and fsConv12FileName convert normal filenames (name.ext) to and from these 12 byte fields.

Directories are stored as word handles, with
- dmGetPathReference and
- dmGetPath you can convert paths to and from these references.

Module information is stored in the 256 byte moduleinfostruct structure. The following functions work with this structure:
- mdbGetModuleReference allocates a reference for such a structure. 0xFFFF is a bad reference
- mdbGetModuleInfo read the structure from a reference.
- mdbWriteModuleInfo writes that structure into the database.
- mdbInfoRead tells you if the info for a reference has been read.
- mdbGetModuleType gets the moduletype of a reference.
- mdbReadMemInfo reads the module information from a memory block (at least 1084 bytes)

Module list entries are stored in 16 bytes modlistentry. modlistentry has 3 fields: filename, directoryreference and modinforeference (see above)
- mdbAppend adds an entry to a module list.
- mdbAppendNew adds only if not present.

- ***Archive Reading: (pfilesel.h)***
- see example
- multivolume archives are not yet supported.
- register your archive reader with an exported instance of the adbregstruct structure in the [fileselector] arcs= list in cp.ini
- adbregstruct::ext points to a string containing the uppercase archive extension. eg. ".ARJ" all the files that have this extension will be scanned by Scan.
- adbregstruct::Scan point to a function that will be called if an archive is to be read. This is the main part.
- adbregstruct::Get and Delete extract and delete files from the archive. They usually call adbCallArc to call the archiver.

At first you have to register your archive with an arcentry with adbAdd. Then get a reference to the archive with adbFind. Scan the archive and add every module (check with fsIsModule) with adbAdd.

simple, eh? :)

If you know how to read the module information directly from the archive, you can do so... (do only if fsScanInArc is true).

- Get a reference to the module info: mdbGetModuleReference
- Check if the information has already been read: mdbInfoRead
- Get the module info: mdbGetModuleInfo
- Extract at least 1084 bytes from the archive and call mdbReadMemInfo
- Write the info: mdbWriteModuleInfo

- ***Interfaces: (pfilesel.h)***

You CAN add a completely new interface... This is quite difficult, since nearly everything is based on the standard cubic player interface. You should only think about this point if you want to "play" something different than a module... This makes cubic player open for enhancements like an MPEG players. Currently this feature is only used for selecting devices (@:\DEVICES\).

You can set up a different interface for all module types (cp.ini: [filetype ...] interface=...). An interface is a structure (interfacestruct) of 3 function pointers. Init will be called with the path, a moduleinfostruct and an open binfile for the module. If Init returns zero the next module will be played. Afterwards Run will be called. Run will set up the screen, process events and afterwards clean up everything it did. The return value determines what to do next:

- 1: play next module
- 2: exit cp
- 3: play next module if files left, otherwise select a module
- 4: select a module
- 5: dos shell

After the dos shell or if esc is pressed in the fileselector Run will again be called. When the next module is to be played, Close will be called

Cubic Player Interface (cpiface.h)

This is the standard interface. It provides basics for players, display modes and predefined modes etc.

- player system
- general display mode system (scopes, note dots)
- automatic textmode window system (extended mode)
- keyhandler system with standard keys (f,enter,esc,...)
- keyhandler to select and mute channels (if there are channels)
- display the top of the screen in text & graphics modes scopes, text and graphic graphic analyser, instruments

players

A player provides 2 functions in the structure cpifaceplayerstruct: OpenFile(path, info, file) and CloseFile. OpenFile gets an open binfile (file) or 0 if the path should be used instead, info is a moduleinfostruct which gives you the information from the fileselector. OpenFile should load the file and start playing it in the background. How it does that does not matter at all. CloseFile should undo all the things OpenFile did. OpenFile can

furthermore set some variables for the defaultmodes (eg. give a function that returns the output sample for the analysers) or define new modes for that player only.

display modes

If you want to use the predefined modes, you either have to set up the mode explicitly by calling functions (for instruments, message, ...) or you define functions to get channel/master samples (for analysers, ...). If you want to make new modes (Most probably in connection with new players, because you have to get information what to play, and the standard player does not provide too much information.. :(I simply don't know what information to give.) you must register them when you start playing. The standard modes register automatically if certain functions are defined. If you want to make a graphics mode or a fullscreen-textmode use cpimoderegstruct:

handle	string handle for the mode (for use in cp.ini)
SetMode	will be called to set up the screen
Draw	will be called to draw the screen
IProcessKey	will be called if the mode is not active and a key is pressed. The parameter is the keycode for the key (bios extended keycode). If the key matches the key to enable the mode call cpiSetMode and return 1, the key will not be processed by other handlers. Otherwise return 0 and let other handlers try to recognize the key.
AProcessKey	will be called if the mode is active. check for and process mode setting keys and return 1 if processed, otherwise 0 to let other handler process the key.
Event	will be called at special points. return 1 if ok or 0 if failed or not ok to use that mode. the parameter tells the program which point. (see below: cpiev...)

cpievOpen	when the mode is opened
cpievClose	when the mode is closed
cpievInit	once per module at module initialization
cpievDone	once per module at module closedown
cpievInitAll	once per session at interface initialization (defaultmodes only)
cpievDoneAll	once per session at interface closedown
cpievGetFocus	(textwin): when the mode receives the input focus
cpievLoseFocus	(textwin): when the mode loses the input focus
cpievSetMode	(textwin): when the textmode is set

textmode windows

The textmode windows work similar to the display modes, only that multiple modes share the screen. The working space for every window must be assigned. The width of the window can be either 52, 80 or 132. 52 wide windows start at x-coordinate 80, the others at 0. y-position is quite free.

handle	string handle for the mode (for use in cp.ini)
GetWin	will be called to get the desired window position/size (fill querystruct)
SetWin	will be called to set the window position/size
Draw	will be called to draw the window
IProcessKey	will be called when a key is pressed and mode is inactive
AProcessKey	will be called when a key is pressed and mode is active
Event	events... see above

top	window at bottom (0) else top of screen
xmode	2 bits: 1: occupy first 80 columns, 2: occupy last 52 columns
killprio	priority when windows have to be killed when lacking space
viewprio	determines how close to the top/bottom border the window will be
size	relative size. instruments==1, track==2. (stupid me, forgot to increase those values...)
hgtmin	minimum height
hgtmax	maximum height

sound programming

There are currently 3 types of devices: wavetable, player and sampler devices.
The sampler devices are not yet very useful, they can only be used to display the sample input from a source, but they cannot really be used to do sampling. (and what should you want to sample from a player? :))
With the player devices you can play a constant stream of sample data.

The wavetable devices are the devices, you'll most probably use. They play multiple samples at different volumes and frequencies at the same time.

Sampler Devices (sampler.h)

Player Devices (player.h)

Player devices play sample data from one looped buffer. To play a long sample you must copy parts of this sample into the buffer before a part of the buffer is played more than once.

creating new player devices

Wavetable Devices (mcp.h)

playing sounds, sending commands, getting information

To communicate with the soundsystem only two functions are used: mcpSet(ch, opt, val) and val=mcpGet(ch, opt). mcpGet usually returns what you've set with mcpSet before. ch is the channel you want to send a command, ch=-1 means no particular channel for global commands. opt is the command value, a description of all the values is listed below. val is the data for the command.

	global settings
mcpGSpeed	tick callback frequency in 1/256 Hz. 256 means 1 Hz, 50*256 is 50 Hz
mcpMasterVolume	master volume. range: 0..64
mcpMasterPanning	master panning: range -64..0..64, invert..mono..normal
mcpMasterBalance	master balance: range -64..0..64, left..mono..right
mcpMasterAmplify	master amplification in 1/65536 full channel volume: 65536 one channel has no amplification. (.MOD: 16384)
mcpMasterPause	master pause: 0: no pause, 1: pause
mcpMasterSpeed	
mcpMasterPitch	
mcpMasterBass	
mcpMasterTreble	
mcpMasterSurround	
mcpMasterReverb	
mcpMasterChorus	
mcpMasterFilter	
	channel settings
mcpCVolume	sets the channel volume. range: 0..256, linear scale
mcpCPanning	sets the channel panning. range -128..128, left to right
mcpCPosition	channel sample position: samples, not bytes. currently this starts playing the sample. this will change!
mcpCPitch	channel pitch in 1/256 halftones: 0 unpitched, linear tone scale
mcpCPitchFix	channel pitch in 1/65536 base frequency: 65536: unpitched, linear frequency scale
mcpCPitch6848	channel pitch in 1/6848 inverse scale: 6848: unpitched, inverse linear frequency scale
mcpCReset	reset channel, does not change muting
mcpCPanY	
mcpCPanZ	
mcpCSurround	
mcpCBass	
mcpCTreble	
mcpCReverb	
mcpCChorus	
mcpCMute	channel muting: 0: off, 1: on
mcpCStatus	channel status. 0: stopped, 1:playing. currently you cannot start a sample with this command. mcpCPosition does this.
mcpCInstrument	channel instrument
mcpCSetLoop	sample loop: 0: no loop, 1: sustain loop, 2: loop

mcpCSetDir	playing direction: 0: forward, 1:backward

Note:

Getting information is currently very limited. you can only use mcpGet with mcpCStatus reliably.
setting mcpCPosition will currently start the sample, while mcpCStatus with 1 as parameter will not. This will change, so if you want to start playing always use both commands together.

Examples:

To play a sound you should do at least the following steps:

- mcpSet(ch, mcpCInstrument, ins);
- mcpSet(ch, mcpCVolume, vol);
- mcpSet(ch, mcpCPosition, 0);
- mcpSet(ch, mcpCStatus, 1);

- **struct sampleinfo:**

All samples must fit into the following structure. I think it is sufficient for every purpose. I hope two loops are enough for everyone.

int	type	format, loops, etc: • 0 sample is signed, 8 bit, pcm • mcpSampUnsigned sample is unsigned • mcpSampDelta sample is stored as delta • mcpSamp16Bit sample is 16 bits • mcpSampBigEndian 16 bit sample is big endian • mcpSampLoop sample has loop • mcpSampBiDi loop is bidirectional • mcpSampSLoop sample has sustain loop • mcpSampSBiDi sustain loop is bidirectional
void *	ptr	pointer to the sample
long	length	length (samples, not bytes)
long	samprate	samplerate (Hz)
long	loopstart	loop start position (samples)
long	loopend	loop end position
long	sloopstart	sustain loop start position
long	sloopend	sustain loop end position

using wavetable devices

- check if a wavetable device is available: If mcpOpenPlayer is zero you may not use the wavetable device
- upload the samples with mcpLoadSamples
- open the device with mcpOpenPlayer
- if anything fails beyond this point close the player!! (but only if it opened correctly)
- set the tick frequency (optional): mcpSet(-1, mcpGSpeed, ...)
- do whatever you want, eg. display something, let the user do something
- call mcpIdle (if not zero) when idling.
- call mcpProcessKey when a key is pressed and the standard interface is used. (this is not part of the wavetable device system)
- close the device with mcpClosePlayer

mcpLoadSamples(sampleinfo* si, int n)
upload samples to soundcard memory. must be called before mcpOpenPlayer

mcpOpenPlayer(int n, void (*p)())
open the player, n channels, tick callback is p. must upload the samples with mcpLoadSamples before.
mcpNChan: number of open channels

mcpClosePlayer()
close the player

int mcpGetFreq6848(int note);
int mcpGetFreq8363(int note);

int mcpGetNote6848(int freq);
int mcpGetNote8363(int freq);

mcpIdle:
call this function when idling. don't call if 0. mixer devices: mix when called

tick callback function, synchronous playing

This function is called at a constant rate, which you can set with mcpSet(-1, mcpGSpeed, ...). This is the function which actually plays the module. It can and should only use the mcpGet and mcpSet commands to play. Note that mcpSet has no immediate effect and therefore the callback function is called before you might expect it. If the callback function is called, it only means that the player is to advance one tick in time and do everything that has to be done in that interval. In the end everything is timed correctly and all the mcpSet commands issued in the callback function are processed simultaneously.

asynchronous playing

Asynchronous playing (ie playing outside the tick callback function) was originally not planned for cubic player. What for anyway? Asynchronous playing should work, but I never tested it. Another problem is, that you can only play the samples, that were uploaded at the beginning. There can also be a long lag when you start a sample. I plan to make better support for asynchronous playing in the future.

getting information

mcpGetRealVolume(int ch, int &l, int &r)
gets the effective volume of a channel
mcpGetRealMasterVolume(int &l, int &r)
gets the effective volume of all channels
mcpGetMasterSample(short *s, int len, int rate, int mono)
gets the total output sample that is to be played. format is 16 bit, signed, stereo (mono)
mcpGetChanSample(int ch, short *s, int len, int rate)
gets the output sample of a channel. format is 16 bit, signed, mono
mcpAddChanSample(int ch, short *s, int len, int rate)
adds the output sample of a channel to the buffer.

dynamic channel allocation

When you open wavetable devices, you specify a maximum number of channels to use. The driver will try to open that amount of channels, but many wavetable devices do not support more than 32 channels, the SoundBlaster AWE only 30. Some modules have more channels, than could be opened. The player could then simply fail, but that's not quite what the user likes. The player could ignore excessive channels. better, but... The player should do dynamic channel allocation, ie. if a channel idles it can be used by other channels. There are also music formats which rely on dynamic channel allocation, like MIDI and Impulse Tracker, which can play multiple samples in one channel, but of course the wavetable device can only play one sample per channel at a time. Now the devices could be made to have the desired amout of virtual channels, but I think the player knows more about the music played and which channels to discard, if there are more playing samples than available channels. For example a loud note should not be discarded if possible, while a soft note can be killed without the audience noticing. A sustained note should be kept, because it is meant to continue playing, if the key is released a channel can be used for a new note, for the note would stop playing soon anyway. If you allocate a channel you should first reset the channel and set the desired muting of the corresponding logical channel. Afterwards you can play the sample. If you use multiple physical channels per logical channel, and want to return information on a logical channel add up the values of the physical channels.

creating new wavetable devices

I won't talk too much on this topic... If you think you need to create a new wavetable device, try to understand the example source.

int mcpReduceSamples(sampleinfo *smps, int nsmps, long mem, int opt);
This function is used internally by mcpLoadSamples to convert the samples to a unique format and to reduce the samples to a maximum amout of memory, so that they will fit into soundcard memory
smps: samples (sampleinfo array)
nsmps: number of samples

mem: maximum memory in bytes
opt: options
mcpRedAlways16Bit: only 16 bit samples, mem passed in number of words
mcpRedNoPingPong: unroll ping pong loops
mcpRedGUS: no 16 bit samples > 256k

mcpMixMaxRate:
mixer devices: maximum number of output samples processed per second
mcpMixProcRate
mixer devices: maximum number of input samples processed per second (#chan*rate)
mcpMixOpt
mixer devices: desired output format

postprocessing the quality mixer output (devwmixq.h)

I believe this is quite a powerful feature. You can freely edit the output sample before it is played (or written to disk with the disk writer device). You can test your effects/filters directly without coding soundcard drivers or temporarily saving the output to disk and play it with some tool afterwards... :)
There are 3 functions you have to write for a postprocessor:

Process(buf, len, rate, stereo)	will be called to process a buffer.
Init(rate, stereo)	will be called to allocate internal buffers (if needed)
Close()	will be called to free the buffers

The buffer is a signed long int buffer and the values will typically fit into far less than 24 bits, so there is much space for you. The sample will later on be clipped and converted to the desired output format, so don't worry about ranges... Len is the buffer length in samples, if stereo is nonzero you must therefore process 2*len longints. You must (of course) export a mixqpostprocregstruct filled with the functionpointers, export it and make changes to cp.ini to use the postprocessor. Add the dll module name to the "link" list in the quality mixer (devwmixq) section in cp.ini and add the export name of that structure to "postprocs". That's it.
If you need user interaction to set up parameters you can use the mixqpostprocaddregstruct: it contains a keyhandler function (as used everywhere in cp). That keyhandler receives a bios extended keycode and should return zero if the key was not recognized or 1 if the key was recognized and should not be processed further. Return 2 when you modified the screen (mode). ie. when you make a setup screen for the effect. CP will then reset the screen mode and redraw the screen. When you return 2 you can also spend a long time in your routines without returning (eg. for the setup screen).

contact information

Send me a mail ONLY if you have a question concerning CPDK or technical question concerning CP.
If you send me a stupid question about CP be prepared for much downloading or stuff.

nbeisert@physik.tu-muenchen.de

closing words

I wish you all a happy new year

pascal