Sega Dreamcast™

The Dreamcast Audio64 API

Dreamcast Audio64 API Table of Contents

Preface

The Sega of America audio solutions (Audio64 and MidiDa) offer a different approach to using the audio hardware resources of the Dreamcast console. The main differences fall into the following areas:

- Sound memory usage
- Interrupt notification
- Control of AICA hardware
- Sound asset creation
- Layered API approach

The basic approach to memory usage is to allow a completely dynamic usage of the sound memory resource. This means that you can write any sound assets you require to any available area of the sound RAM and play them. You can also dynamically allocate this memory and dynamically replace it. There are no static "memory maps" to restrict your usage of this memory. Along with the freedom to use this memory as you see fit comes the possibility of unpredictable results if you misuse this resource.

Interrupt notification is implemented for sound (PCM or ADPCM) playback and MIDI playback, and sound driver command processing. This means that you can know immediately when sound resources become available and reuse the resource, thereby increasing the bandwidth of the audio system.

Control of the AICA hardware is intended to be as exposed as possible, implementation time permitting. This means that you can directly allocate all 64 audio channels (in the case of the Audio64 sound driver) and control each channels pan, volume control and sample rate playback directly. You can also gang multiple channels for phase-locked audio playback (up to 64 channels), and can dynamically control DSP effects on each.

All audio assets can be bundled together into banks of multiple or individual type data. The collection into banks is as easy as copying standard assets (.wav or .fpb or .fob or .mid or .mpb files, etc.) into a directory and running a DOS-level utility. Streams can be built in a like manner, and multiple track streams are accommodated by the tools.

Providing a layered API means developers can work at the level they are most comfortable with, and that multiple approaches are accommodated. We believe that full access to the hardware will yield the best results, that developers can adapt their current game development environments more easily and be immediately more productive. The AC (AICA Control) layer allows direct control of the AICA sound driver and hardware, while the AM (AICA Manager) layer provides higher level control with an architecture that provides dynamic resource allocation and stream playback control, among other things.

The AICA hardware is an audio subsystem that supports 2 MB of sound RAM, has 64 audio playback channels which can play 16, 8 or 4 bit data at sample rates from > 11 megaHertz (theoretically) down to 172 hz (approximately). It also contains a built-in DSP unit which can provide high quality reverb and Qsound and a multitude of DSP effects which can be flexibly configured. It has a built-in digital 16 channel mixer, and can route Redbook audio through the DSP. Each voice channel also has a hardware 5 stage resonant Low Pass Filter, wave selectable amplitude and pitch LFOs, and an amplitude ADSR envelope. The sound subsystem has an embedded RISC ARM7 CPU running at 25 MHZ, and the AICA sound registers are controlled by one of the sound drivers (Audio64 or MidiDa) written in ARM7 assembler. SH4 CPU usage is minimized by utilizing ARM7 control.

1. Audio64 Principles of Operation

1.1 General Remarks

This chapter provides information about getting up and running quickly with Audio64 version R10. As mentioned in the introduction, the Audio64 library is provided at two layer levels, the lower level being the “AC” level, or “AICA Control” layer level, and the higher level being the “AM” level, or “AICA Manager” layer level. The first question to ask is: “Which level do I want to implement audio at for our application”?

The AC level provides the most direct and efficient access to the Dreamcast sound hardware and Arm7 sound driver, so you have the most flexibility in loading and controlling sound, managing voices and memory and fielding interrupts back from the AICA hardware. On the other hand, the AM level integrates with the DOS command-line tools which bundle sound assets together, provides mechanisms for loading and accessing the banks files created by the tools, provides a voice manager, memory manager and an integrated asynchronous file streamer.

You may also use CRI’s ADX API to stream music files at either level (AC or AM) so if music streaming is the only additional need you have and you don’t want to write an audio streamer you don’t have to be dissuaded from using the AC level. If you are interested in MIDI file playback you can load the `MidiDa.drv` driver instead of the digital-audio-only `Audio64.drv` driver and additional MIDI commands in the library will become functional. If you can’t decide which layer you want to implement your control at, **you can use AM level calls in general and drop down to the AC level whenever you need additional control just by extracting the voice number from the AM structure.** You can then use AC level calls for additional commands.

The best performance of Audio64 is gained by a liberal use of the interrupt/callback mechanisms. You can always know when any channel has played or command has been processed by using the commands:

```
AcDigiRequestEvent        // return an interrupt when channel reaches a particular playback position
acMidiRequestEvent        // return an interrupt when the MIDI port reaches a particular playback
position
acSystemRequestArmInterrupt// return an interrupt when the ARM7 driver reaches THIS command in the queue
```

The AM layer commands automatically install the playback position request event commands to know when to release (free) one-shot sound effect channels, MIDI ports, or to feed the asynchronous streamer.

1.2 Initialization

The Audio64 Library was written with the idea of providing switchable components for various OS-level subsystems like the file system, memory system, interrupt control, etc. to provide a more OS-independent library where applications could “roll their own” components. In practice this flexibility hasn’t been utilized much, and the increased complication required at initialization time didn’t seem quite worth the effort. Therefore exposed hooks to function pointer variables for these components have been removed at version R10. However, this capability is still possible if you link your own version of these functions when you build your application, in which case the default function pointers will be overwritten (look at the **A64Thunk** example). At any rate, all the samples provided with R10 contain simplified boilerplate initialization code (prefixed with `bp`), and just require the sound driver(s) to be present on the GDROM (audio64.drv and/or midida.drv) for initialization. The initialization procedure for the AC layer is very simple. The call to do this looks like this:

```
if (!bpAcSetup(AC_DRIVER_DA, KTFALSE, KTNULL)) // for MIDI use: AC_DRIVER_MIDI
           ABORT;     //  this will drop into debugger upon initialization failure
```

The called function receives these settings:

```
KTBOOL bpAcSetup(AC_DRIVER_TYPE driverType, KTBOOL usePolling,
           AC_CALLBACK_HANDLER theCallback)
```

The driver type is switchable between DA or MIDI, the `usePolling` flag sets message array polling as opposed to interrupt processing (polling is **NEVER** recommended), and `theCallback` is the application function used to receive interrupt-level messages from the AICA interrupt handler. **In the case of the `AC_DRIVER_DA` sound driver there is a 1-to-1 correspondence between `MessageID` and voice channel number. In the case of the `AC_DRIVER_MIDI` there are 16 digital audio channels and 16 MIDI ports available; in this case `MessageID`s will be 0-15 for digital audio channels and 16-31 for the MIDI port numbers. User-ID numbers are numbered 64-255 for both drivers.**

Typical reasons for initialization failures are: driver file not present on the GDROM or Version number mismatch – the `Audio64.lib` checks the driver’s version number and fails if there is a version number mismatch.

AM level initialization looks much the same:

```
if(!bpAmSetup(AC_DRIVER_DA, KTFALSE, KTNULL))
           ABORT;
```

AM relies on AC, and essentially just installs itself as the AC layer callback and takes it upon itself to dispatch callbacks to the application. It makes use of the file system function pointers for bank loading and audio streaming. Again, these are replaceable as indicated above.

You may also want to issue a few additional commands at this time:

```
acSystemSetStereoOrMono(playbackMode);// mode is 0 for stereo, 1 for mono
acSystemSetMasterVolume(volume);      // volume 0-15, soft to loud. If mono, adjust 1 lower than stereo
```

1.3 Sound Playback at AC level

Sound playback is pretty straightforward. There are just a few requirements: an initialized sound driver (shown above), the `soundfile` in sound ram, and an open channel with the address and characteristics of the `soundfile` indicated. The Dreamcast hardware restricts directly loading `soundfiles` from the GDROM into soundram, because this transfer process can block (and therefore lose) data from the Maple bus. Therefore all sound loading has to be done first to SH4 ram, then copied into soundram. You of course need to know where you can begin to copy data into soundram, because the sound driver and its tables and ring buffers all reside in soundram. The soundram is mapped to address `0xA0800000`, and is currently 2 MB in size, so the highest possible memory address is `0xA09FFFFF`. **All soundram accesses must be `DWORD` (32 bit) aligned. If any access is not aligned the soundram bus controller will malfunction and the driver will most likely hang.** The driver posts the first free memory area you can safely use, so the actual amount of soundram you have available is `FIRSTFREE – 0xA09FFFFF`. Assuming the `soundfile` resides in SH4 memory the following calls accomplish a simple soundram load and playback:

```
nextFree = acSystemGetFirstFreeSoundMemory();  //Get the beginning of free sound (AICA) memory //
(build up your sound memory allocator from here to 0xA09FFFFF)

acG2Write(nextFree, sh4soundfile, lengthInBytes);// do a synchronous 32-bit aligned write to soundram

acDigiOpen(voiceChannel, nextFree, lengthInBytes, AC_16BIT, 44100)// channel is 0-63 or 0-15
                          // address in soundram was returned by driver
                          // length is in bytes, must be < 64k in samples
                          // sample size is 16 bit, type is PCM data
                          // playback sample rate is 44.1khz

acDigiPlay(voiceChannel,offset,loopFlag);// play the sample (default is max volume)
```

Because of multiple G2 bus restrictions and DMA restrictions you should always use `Audio64.lib` calls to copy data to soundram. For DMA data transfers you should set the transfer mode to DMA. This has no effect on foreground transfers (`acG2Read` or `acG2Write`, etc.). **If you never set the `TransferMode` to DMA the `acTransfer` call will use `acG2Write` instead of DMA.**

```
acSetTransferMode(AC_TRANSFER_DMA);
acTransfer(*request);     // use for DMA transfers
```

This method (`acTransfer`) should be used to copy data for large block transfers or continuous (streaming) block transfers. Callbacks are available upon DMA completion.

When you open a port for playback it is important that the length of the sample never exceed 64k samples, so the largest “one-shot” file you can play back is 128k if sample size is 16 bit, 64k if sample size is 8 bit, and 32k if the sample type is ADPCM. The sample playback rate can range from about 2.5 khz to sample rates above human hearing.

`acDigiPlay()` is the simplest of the playback commands. You can loop the sample by setting the `loopFlag` to `KTTRUE`. Some playback variations are:

```
acDigiPlayWithParameters      (voiceChannel, volume, pan,  dspMixerChannel,  dspSendLevel,
                              directLevel, frequencyOrCentsOffset, frequencyOrCentsFlag,
                              callbackOffsetInFrames,  loopStartOffsetInFrames,
                              loopEndOffsetInFrames);
acDigiPlayWithLoopParameters  (voiceChannel, startOffset, aicaLoopFlag, loopStartOffsetInFrames,
                              loopEndOffsetInFrames);
acDigiMultiPlay               (aicaLoopFlag, upperMask, lowerMask);
```

The most powerful play command is the `acDigiPlayWithParameters()` and the one probably used the most. With this command you can set general playback characteristics (volume, pan, etc.) signal routing (mixer channel and level), sample rate, loop parameters and the playback position sample address which will trigger a callback to your application when the audio hardware plays the sample to that point. `AcDigiMultiPlay()` is the command to use to phase-lock multiple channels together for stereo or multiple channels.

Sounds can be stopped with:

```
acDigiStop(voiceChannel); //  generally used to stop looping sounds
```

The sound driver will toggle channels off automatically when played with the loop parameter off.

To clean up and reset the `voiceChannel` to known defaults it was necessary (at R9) to issue the…

```
AcDigiClose(voiceChannel); // close the channel, reset parameters
```

So the cleanest command sequence to trigger sounds for R9 was:

```
AcDigiOpen();
AcDigiPlay();
AcDigiStop();             //optional for looping sound
AcDigiClose();
```

With R10 the `acDigiOpen()` command resets parameters – therefore the most efficient command sequence becomes:

```
AcDigiOpen();
AcDigiPlayWithParameters();
…
sound stops by itself, then:
…
AcDigiOpen();
AcDigiPlayWithParameters();
…etc.
```

Sounds can be opened and played with just 2 calls. **Please note, however, that if you use the latter method you must insure that the sound has completely finished playing before re-playing the port, as the port's new open settings will update the live registers when the voice is played.** If you are depending on a callback to insure the voice has stopped place the callback on the last or second-to-last sample to avoid possible audio glitches, or you can just issue the `acDigiStop` call. When a `voiceChannel` (port) is opened the default pan (center), volume (maximum), and playback rate (what port was opened with) is set. To override these values ***before*** playback, the series of playback parameters like:

```
acDigiSetCurrentPitch(port, pitchOffsetInCents);
acDigiSetSampleRate(port, sampleRate);
acDigiSetVolume           (port, aicaVolume);
acDigiSetPan(port, aicaPan);
acDigiSetFilterEnvelope(port, filterOn, filterQ, filterValue0, filterValue1, filterValue2,
                          filterValue3, filterValue4, filterAttackRate, filterDecayRate1,
                          filterDecayRate2, filterReleaseRate);
acDigiSetAmplitudeEnvelope(port, ampAttackRate, ampDecayRate1, ampDecayRate2, ampReleaseRate,
ampDecayLevel,            ampKeyRateScaling, ampLoopStartLink);
acDigiSetLfoRegisters(port, lfoReset, lfoFrequency, pitchLfoWaveShape, pitchLfoDepth,
                          ampLfoWaveShape, ampLfoDepth);
```

So a simple `acDigiPlay()` command will use these new values. If the sample is playing (looping) when these calls are made, they will be instantly applied to the playing sound with the exception of the "Envelope" commands, which get triggered upon initial playback (or `KEYON`) of the sound. AICA hardware volume settings are actually set in decrements of .375 db, thereby forming a logarithmic power curve, from 0-255. The API inverts this orientation for ease of use. A table has been provided (`AICAVolumeConversionTable.c`) to allow AICA volume increments (decrements) to follow a more linear power curve.

The MIDI system follows the same paradigm, where you do the same Open, Play, Stop, Close sequence of commands, i.e.

```
acMidiOpen(port, gmMode, address, sizeInBytes, pulsesPerQuarterNote);  // port is 0-15 (MIDI interrupt
// message Ids become 16-31)
// gmMode sets the drum channel as 10
// pulsesPerQuarterNote sets the MIDI QN resolution
acMidiPlay(port, startOffset, aicaLoopFlag);// start playback must point at Mtrk data, not Mthd data
acMidiStop(port);         // stop playback, issue key off for all sounding MIDI notes
acMidiClose(port);        // reset port
```

For R10, the…

```
AcMidiOpen();
AcMidiPlay();
AcMidiStop();             // optional if looping, or callback address is not pointing to last note data
…
```

…command sequence should be enough. There is some additional setup you must do in order to "patch" the MIDI instrument bank (`.mpb file`) to the MIDI sequence(s) requiring them. This is done with the following call:

```
acMidiSetTonebank(toneBankslot, bankType, address, sizeInBytes, mttPtr);  // slot is 0-15, corresponds
to
// MIDI Continuous Controller msg 32 bank select in MIDI sequence file
// banktype 0-2.  If 0, then instrument bank, 1 or 2 are drum program banks
// address is location in soundram, size is in bytes
// mttptr is pointer to MIDI program translate table (not yet implemented)
```

There are up to 16 MIDI Program Bank files addressable simultaneously, and they must be resident in soundram at the time of the call. **Only MIDI Format 0 files are supported, and there must be no System Exclusive messages present in the sequences.**

1.4 Sound Playback at AM level

Sound playback at the AM level is meant to integrate with the `MKBANK.EXE` and `MKSTREAM.EXE` DOS tools' output (`.KAT` and `.STR` files), and require very little explicit management of sound resources. Basically the AM playback process requires an `AM_SOUND` structure to be filled in by various AM layer calls for voice and callback management purposes. The AM boilerplate playback code shows the following setup:

```
amSoundFetchSample((KTU32*)theBank, soundNumber, theSound)
          // theBank is a .KAT bank previously
          // loaded into soundram
          // the soundNumber is the asset offset number in the bank (taken from the bankname.h file)
          // theSound is an AM_SOUND struct allocated by your program.

amSoundAllocateVoiceChannel(theSound)// assigns a voiceChannel to the sound (0-63 or 0-15)
          // this is the voiceChannel to use if you want to control sound at the AC level also.

amSoundPlay(theSound)  // this plays the sound with the parameters specified in the .oss file
          // which are built by the MKSCRIPT.EXE program including
          // looping or not, pitch variation, volume, pan, etc.
```

The AM layer will set a callback for these sounds and de-allocate the voice when the callback comes in. You can set the system volume mode at the AM level to use a linear volume conversion table with the `amSystemSetVolumeMode(USELINEAR)` command.

The AM layer streamer is invoked by a series of AM calls, but the boilerplate code shows how to play a stream simply with a few calls:

```
StreamSetup();  // Sets up the transfer and play buffers for the stream


bpAmStreamStart(gStream); //  starts playback based on struct parameters,
                          // including filename, length, nbr of tracks, etc.

ReplayStream();           // invoked when stream finishes for looping playback
```

Also the stream server must be installed in your main loop to keep data transfer happening in the background. For the most efficient playback you should set the `acTransferMode` to `AC_TRANSFER_DMA`. The stream server is called as follows:

```
bpAmStreamServer();   // check for stream transfer status, read from disk and fill buffers if necessary
```

For details on the boilerplate streamer control please check the samples. Multiple files can be streamed at once (see the `AmStream` example). The streamer uses asynchronous file system calls and DMA (if selected) and has low overhead. The streamer sets it's own callback `proc` for stream management. To check the playback characteristics of a bank or stream you may use the `VUBANK.EXE` or `VUSTREAM.EXE` DOS commands.

1.5 Error Handling

Some of the most common errors encountered during normal operations are the 301 (AC-level) and 701 (AM-level) errors. These errors typically occur because the sound driver command queue has been filled up and cannot accept any more commands until the driver processes at least one more command (because of the nature of the circular queue). The driver's command queue is 32 commands deep; there is a great discrepancy between operating speeds of the SH4 CPU and the ARM7 CPU making it quite easy to overflow the command queue. There are basically 2 solutions to this problem: re-send the command or write an Sh4 side queuing mechanism (this will be added for R11). Error checking must be done to insure that important commands like the `acDigiRequestEvent` or `acDigiPlay` commands actually make it into the command queue. The other reason this error can occur is if a non-32bit-aligned write or read or a write or read before or beyond the 2 MB sound ram limit takes place. This can cause the AICA bus controller to malfunction and crash the driver, in which case no further commands will be processed until the driver is re-initialized. As far as other errors are concerned, most errors are illegal parameter range errors; to find out what may have failed you need to get a pointer to the internal error writing `struct` which is filled in by all `Audio64.lib` command functions:

```
bpAcError = acErrorGetLast();// returns pointer to audio64.lib ac error struct – do once at init time
bpAmError = amErrorGetLast();// returns pointer to audio64.lib am error struct – do once at init time
```

Then check the command return boolean, i.e.

```
if(myAcA64Cmd(.…,…,…)==KTFALSE)
{
            bpShowAcError (bpAcError);  // fix or retry, etc.
}

or

if(!myAmA64Cmd(.…,…,…))
{
            bpShowAmError (bpAmError);  // fix or retry, etc.
}
```

The error messages are quite explicit – you should be able to tell what failed immediately. If the failure only seems to occur at runtime on the "release" binary version you can put a hard break into the code and check the return stack trace or AC or AM error pointer(s) to see what may have happened (on Set5 development hardware).

```
acDebugBreak(void);                     // Hard breakpoint -- stack trace enabled
                                        // Runtime check for production hardware (no DA stub)
#define acASEBRK(status)                                  ((status) ? acDebugBreak(), 0 : 0)
```

1.6 Shutdown

The Audio64 driver and library can be gracefully removed by using the…

```
    amShutdown();
```

or

```
    acShutdown();
```

…calls. These calls will unchain their interrupt handlers from the Shinobi OS, clear any running DSP program in the sound driver, free internally allocated memory and stop all sound playback. At this point the sound driver can be switched out.

2. The AICA Control Layer API

acCdInit

Resets CDDA channels to hard pan positions and maximum volume.

FORMAT

```
#include <ac.h>

KTBOOL acCdInit(void)
```

PARAMETERS

void

RETURN VALUE

`KTTRUE`	if successful
`KTFALSE`	if command write fails.

FUNCTION

Sets default pan position and volume for CDDA playback channels.

Note: This must be called prior to playing back redbook audio from the CD.

acCdSetPan

Sets Left & Right Channel pan position.

FORMAT

```
#include <ac.h>
KTBOOL acCdSetPan          (KTU32 leftPan,KTU32 rightPan)
```

PARAMETERS

`leftPan`	Pan Position for left audio channel, 0-127, left to right.
`rightPan`	Volume level for right audio channel, 0-127, left to right.

RETURN VALUE

`KTTRUE`	if successful
`KTFALSE`	if the `leftPanrightPan` is out of range, or command write fails.

FUNCTION

Sets pan position for CD-DA playback channels.

Note: This may affect the behavior of volume command since `left` channel can be set to the right pan position and vice versa.

acCdSetVolume

Sets Left & Right Channels for Redbook Volume Control (dependent on channel pan).

FORMAT

```
#include <ac.h>

KTBOOL acCdSetVolume       (KTU32 leftVolume,KTU32 rightVolume)
```

PARAMETERS

`leftVolume`	Volume level for left audio channel, 0-127.
`rightVolume`	Volume level for right audio channel, 0-127.

RETURN VALUE

`KTTRUE`	if successful
`KTFALSE`	if the `leftVolumerightVolume` is out of range, or command write fails.

FUNCTION

Sets volume level for CD-DA (Redbook) playback channels.

Note: Left and right depend on CD-DA Pan position.

See: `gdfsgdda` documentation for calls to play and stop tracks.

acDigiClose

Closes port previously opened.

FORMAT

```
#include <ac.h>

KTBOOL acDigiClose(KTU32 port)
```

PARAMETERS

`port`	`Voice channel number`,0-63 for `audio64` driver, 0-15 for `MidiDa` driver.

RETURN VALUE

`KTTRUE`	if successful
`KTFALSE`	if port is out of range or command write fails.

FUNCTION

Closes voice channels opened with `acDigiOpen()`. It is important to close a channel and to not iteratively open the results of that type of methodology are undefined.

acDigiGetCurrentReadPosition

Returns the current play position for a given voice channel.

FORMAT

```
#include <ac.h>

KTBOOL acDigiGetCurrentReadPosition(     KTU32 voiceChannel,
KTU32 *currentSampleFrame)
```

Parameters

`KTU32 voiceChannel,`	the voice channel from which to obtain the play position.
`KTU32 *currentSampleFrame,`	the current sample frame is returned via this pointer.

RETURN VALUE

KTTRUE	if successful
KTFALSE	if driver is not installed.
	if voiceChannel is out of range.

FUNCTION

Returns the current play position in sample frames for a given voice channel via the argument `currentSampleFrame`.

The driver maintains an array of 64 DWORDs that contain the channel offsets. Each element in the array has the offset from the start of the play buffer in sample frames. This function will return the current position of the play cursor as an offset from the start of the buffer in sample frames.

acDigiMultiPlay

Sets the bit masks for acDigiMultiPlay()

FORMAT

```
#include <ac.h>

KTBOOL acDigiMultiPlay(KTS32 aicaLoopFlag, KTU32 upperMask, KTU32 lowerMask)
```

PARAMETERS

`KTS32 aicaLoopFlag,`	Start channels as looping or not, `AC_LOOP_ON` or `AC_LOOP_OFF`
`KTU32 *upperMask,`	A pointer to the upper 32 channel mask, voices 32-63
`KTU32 *lowerMask,`	A pointer to the lower 32 channel mask, voices 0-31

RETURN VALUE

`KTTRUE`	if successful
`KTFALSE`	if `upperMask` is 0 or the top 4 bits of `lowerMask` are set and `MidiDa` driver is in use.
	if upper and lower masks are 0.

FUNCTION

Starts a group of channels as a phase locked gang.

acDigiMultiSetMask

Sets the bit masks for acDigiMultiPlay()

FORMAT

```
#include <ac.h>

KTBOOL acDigiMultiSetMask(KTU32 port,KTU32 *uppermask, KTU32 *lowermask)
```

PARAMETERS

`port`	Voice channel number, 0-63 for `audio64` driver, 0-15 for `MidiDa` driver.
`KTU32 *uppermask,`	A pointer to the upper 32 channel mask, voices 32-63
`KTU32 *lowermask,`	A pointer to the lower 32 channel mask, voices 0-31

RETURN VALUE

`KTTRUE`	if successful
`KTFALSE`	if `upperMask` or `lowerMask` is `NULL` or port is out of range.

FUNCTION

Creates channel masks for use with the `acDigiMultiPlay()` function. This may be called in a loop to set multiple channels in the mask.

acDigiMultiStop

Sets the bit masks for acDigiMultiPlay()

FORMAT

```
#include <ac.h>

KTBOOL acDigiMultiStop(KTU32 upperMask, KTU32 lowerMask)
```

PARAMETERS

`KTU32 *upperMask,`	The upper 32 channel mask, voices 32-63
`KTU32 *lowerMask,`	The lower 32 channel mask, voices 0-31

RETURN VALUE

`KTTRUE`	if successful
`KTFALSE`	if `upperMask` is 0 or the top 4 bits of `lowerMask` are set and `MidiDa` driver is in use. if upper and lower masks are 0.

FUNCTION

Stops a group of channels as a phase locked gang.

acDigiOpen

Open a DA Streaming Port for playback.

FORMAT

```
#include <ac.h>

KTBOOL acDigiOpen(KTU32 port,KTU32 address,KTU32 sizeInBytes,AC_AUDIO_TYPE
aicaAudioType,KTS32 aicaSampleRate)
```

PARAMETERS

`port`	Voice channel number, 0-63 for `audio64` driver, 0-15 for `MidiDa` driver.
`address`	the address in sound memory
`sizeInBytes`	buffer length in bytes, maximum 128k for 16bit data,64k for 8bit data, 32k for 4bit (`ADPCM`) data.
aicaAudioType	format type 4, 8, or 16 bit. See `AC_AUDIO_TYPE` data type enumeration in `ac.h`

```
typedef enum
{
AC_16BIT,
AC_8BIT,
AC_ADPCM
AC_ADPCM_LOOP
} AC_AUDIO_TYPE;
```

`sampleRate`	Base real world sample rate. Further play commands on this open port will start at this rate unless changed by a call to `acSetSampleRate()`.

RETURN VALUE

`KTTRUE`	if successful
`KTFALSE`	if port is out of range, address is 0, sizeInBytes is 0, `audioType` is out of range, `sampleRate` exceeds 1128900 or command write fails.

FUNCTION

Opens a digital voice channel and assigns a buffer, root pitch and loop status to the voice.

acDigiPlay

Starts a buffer playing.

FORMAT

```
#include <ac.h>

KTBOOL acDigiPlay(KTU32 port,KTU32 startOffset, KTS16 aicaLoopFlag)
```

PARAMETERS

`port`	Voice channel number, 0-63 for `audio64` driver, 0-15 for `MidiDa` driver.
`startOffset`	Play from start of buffer assigned to port (only 0 supported for this release).
`aicaLoopFlag`	Play looping buffer, 0 (loop off) or 0xff (loop on). If this is out of range it will be set to `AC_LOOP_OFF`

RETURN VALUE

`KTTRUE`	if successful
`KTFALSE`	if port is out of range or `startOffset != 0` or unable to send command.

FUNCTION

Plays buffer assigned to the voice channel by `acDigiOpen()`. The total length of buffer must be < 64k sample frames.

acDigiPlayWithLoopParameters

Starts a buffer playing Set loop points.

FORMAT

```
#include <ac.h>

KTBOOL acDigiPlay(KTU32 port,
KTU32 startOffset,
KTS16 aicaLoopFlag,
KTU16 loopStartOffsetInFrames,
KTU16 loopEndOffsetInFrames)
```

PARAMETERS

`port`	Voice channel number, 0-63 for `audio64` driver, 0-15 for `MidiDa` driver.
`startOffset`	Play from start of buffer assigned to port. (only 0 supported for this release)
`aicaLoopFlag`	Play looping buffer, `AC_LOOP_ON` or `AC_LOOP_OFF`.

Note: If this is out of range it will be set to `AC_LOOP_OFF`

`loopStartOffsetInFrames`	The 0 based loop start offset in sample frames.
`loopEndOffsetInFrames`	The 0 based loop ending offset in sample frames.

RETURN VALUE

`KTTRUE`	if successful.
`KTFALSE`	if port is > 63, or startOffset != 0, or unable to send command

FUNCTION

Plays buffer assigned to the voice channel by `acDigiOpen()`. The total length of buffer must be < 64k sample frames. The sample loop offsets are 0 based numbers that are expressed in sample frames e.g. 16 bit data is 2 bytes, 8 bit is 1 byte and 4 bit (`ADPCM`) is 1 nibble per frame.

acDigiPlayWithParameters

Starts a buffer playing with all common parameters.

FORMAT

```
#include <ac.h>

KTBOOL acDigiPlayWithParameters(           KTU32 port,
KTU32 volume,
KTU32 pan,
KTU32 dspMixerChannel,
KTU32 dspSendLevel,
KTS32 frequencyOrCentsOffset,
AC_PITCH_SET_TYPE frequencyOrCentsFlag,
KTU32 callbackOffsetInFrames,
KTU16 loopStartOffsetInFrames,
KTU16 loopEndOffsetInFrames)
```

PARAMETERS

`port`	Voice channel number, 0-63 for `audio64` driver, 0-15 for `MidiDa` driver.
`volume`	0-15, soft to loud.
`pan`	0-31, left to right.
`dspMixerChannel`	0-15, needs to match DSP algorithm mapping.
`dspSendLevel`	0-15, min to max.
`frequencyOrCentsOffset`	either the real world sample rate i.e. 44100, 32000, 22050 etc. or the pitch offset in cents.
`frequencyOrCentsFlag`	One of the following values from `ac.h`
`AC_PITCH_NO_CHANGE`	values in `frequencyOrCentsOffset` are ignored.
`AC_PITCH_AS_SAMPLE_RATE`	value in `frequencyOrCentsOffset` will be interpreted as a real world sample rate.
`AC_PITCH_AS_CENT_VALUE`	value in `frequencyOrCentsOffset` will be interpreted as a cents offset from the root pitch at which the port was opened.
`callbackOffsetInFrames`	The 0 based callback offset in sample frames
`loopStartOffsetInFrames`	The 0 based loop start offset in sample frames, ignore == 0.
`loopEndOffsetInFrames`	The 0 based loop ending offset in sample frames, ignore == 0.

RETURN VALUE

`KTTRUE`	if successful.
`KTFALSE`	if port or `dspMixerChannel` is out of range or command write failed,

FUNCTION

Plays buffer assigned to the voice channel by `acDigiOpen()`. The total length of buffer must be < 64k sample frames. The sample loop offsets are 0 based numbers that are expressed in sample frames e.g. 16 bit data is 2 bytes, 8 bit is 1 byte and 4 bit (`ADPCM`) is 1 nibble per frame.

Note: If volume, pan or `dspSendLevel` are out of range they will be set to equal the max value for the range. If the loop offsets are set to `0` they will be ignored by the driver.

acDigiPortsAvailable

Returns count of available ports for DAMIDI.

FORMAT

```
#include <ac.h>

void acDigiPortsAvailable(KTU32 *daCount, KTU32 *midiCount)
```

PARAMETERS

`daCount`	Pointer to a KTU32 where the number of available DA ports is returned. It may be `KTNULL` to ignore these ports.
`midiCount`	Pointer to a KTU32 where the number of available MIDI ports is returned. It may be `KTNULL` to ignore these ports.

RETURN VALUE

void

FUNCTION

Returns the current port availability count for DA and MIDI ports

acDigiRequestEvent

Used to generate an interrupt when a certain buffer position is reached.

FORMAT

```
#include <ac.h>

KTBOOL acDigiRequestEvent(KTU32 port,KTU32 offsetFromBeginningInFrames)
```

PARAMETERS

port	Voice channel number, 0-63 for `audio64` driver, 0-15 for `MidiDa` driver.
`offsetFromBeginningInFrames`	0 based offset from start of buffer in sample frames. (0-65535)

RETURN VALUE

`KTTRUE`	if successful
`KTFALSE`	if the port is out of range or command write fails.

FUNCTION

When channel playback position reaches the indicated offset the driver will raise an ARM external interrupt causing the ARM interrupt handler to be invoked. The channel number of the calling channel will be placed into the drivers interrupt array. If multiple channels are reporting event requests at the same time there will be multiple entries in the drivers interrupt array. The number of channels reporting may be observed by measuring the incrementation of the interrupt array start offset which is available via the `acSystem` call `acSystemGetIntArrayStartOffset()`

Note: The parameter `offsetFromBeginningInFrames` is not error trapped.

acDigiSetAmplitudeEnvelope

Sets the amplitude envelope for a channel.

FORMAT

```
#include <ac.h>

KTBOOL acDigiSetAmplitudeEnvelope(KTU32 port, KTU32 ampAttackRate, KTU32 ampDecayRate1,
KTU32 ampDecayRate2, KTU32 ampReleaseRate, KTU32 ampDecayLevel,
KTU32 ampKeyRateScaling, KTU32 ampLoopStartLink);
```

PARAMETERS

`KTU32 port,`	voice channel, 0-63 for Audio64, 0-15 if `MidiDa` driver
`KTU32 ampAttackRate`	Attack transition time from 8100 to 0.0 msec, 0x00 to 0x1f respectively
`KTU32 ampDecayRate1`	Stage 1 decay transition time from 8100 to 0.0 msec, 0x00 to 0x1f respectively
`KTU32 ampDecayRate2`	Stage 2 decay transition time from 8100 to 0.0 msec, 0x00 to 0x1f respectively
`KTU32 ampReleaseRate`	Release transition time from 8100 to 0.0 msec, 0x00 to 0x1f respectively
`KTU32 ampDecayLevel`	Specifies the EG level for transition from stage 1 to stage 2
`KTU32 ampKeyRateScaling`	Specifies the rate of EG key rate scaling, minimum to maximum, 0x00 to 0x0e, 0x0f scaling is off
`KTU32 ampLoopStartLink`	links playback EG to transition to `DecayRate1` level when playback address exceeds loop start address

RETURN VALUE

`KTTRUE`	if command write successful;
`KTFALSE`	otherwise.

FUNCTION

Sets the Amplitude Envelope parameters for sound playback. Can adjust envelope transition based on frequency (key rate scaling).

```
(where envelope execution rate = (KRS + OCT * 2 + FNS + rate) * 2
```

acDigiSetCurrentPitch

Changes the playback rate of a running channel.

FORMAT

```
#include <ac.h>

KTBOOL acDigiSetCurrentPitch(KTU32 port,KTS32 pitchOffsetInCents)
```

PARAMETERS

`port`	Voice channel number, 0-63 for `audio64` driver, 0-15 for `MidiDa` driver.
`pitchOffsetInCents`	Pitch in cents. (-8400 to 8400)

RETURN VALUE

`KTTRUE`	if successful
`KTFALSE`	if port is out of range or command write fails.

FUNCTION

Changes the pitch of a currently running voice channel in increments of cents.

Cents is a musical measurement of pitch, one octave (frequency double or half) is 1200 cents.

Making this call will not change the default setting of the voice channel but it will change the pitch of a sound that is currently playing on that channel.

If the currently playing sound stops and is retriggered with a call to `acDigiPlay()` it will play at the sample rate that the vice channel was set up for in the `acDigiOpen()` call.

Calling this with an arg of 1200 will make the sound play up one octave, a second call to this with an arg of 0 will make the sound play at the setting to which the voice was initialized in the `acDigiOpen()` call.

acDigiSetFilterEnvelope

Sets the filter envelope for a channel.

FORMAT

```
#include <ac.h>

KTBOOL acDigiSetFilterEnvelope(KTU32 port, KTU32 filterOn, KTU32 filterQ, KTU32
filterValue0, KTU32 filterValue1, KTU32 filterValue2,
KTU32 filterValue3, KTU32 filterValue4, KTU32 filterAttackRate, KTU32 filterDecayRate1,
KTU32 filterDecayRate2, KTU32 filterReleaseRate);
```

PARAMETERS

`KTU32 port,`	voice channel, 0-63 for Audio64, 0-15 if MidiDa driver
`KTU32 filterOn`	1 for filter on, 0 for off
`KTU32 filterQ`	Q = 0.75 x value -3 (0x00 to 0x1f), from -3 to 20.25 db
`KTU32 filterValue0`	starting filter frequency (13 bit) from 1hz to 20khz (0x0000 to 0x1fff) at attack start
`KTU32 filterValue1`	stage 1 filter frequency (13 bit) from 1hz to 20khz (0x0000 to 0x1fff) at decay start time
`KTU32 filterValue2`	stage 2 filter frequency (13 bit) from 1hz to 20khz (0x0000 to 0x1fff) at sustain start time
`KTU32 filterValue3`	stage 3 filter frequency (13 bit) from 1hz to 20khz (0x0000 to 0x1fff) at `KeyOff` time
`KTU32 filterValue4`	release filter frequency (13 bit) from 1hz to 20khz (0x0000 to 0x1fff)
`KTU32 filterAttackRate`	0x1f to 0x00, short to long transition, approximately 3.1 to 118200 msecs
`KTU32 filterDecayRate1`	0x1f to 0x00, short to long transition, approximately 3.1 to 118200 msecs
`KTU32 filterDecayRate2`	0x1f to 0x00, short to long transition, approximately 3.1 to 118200 msecs
`KTU32 filterReleaseRate`	0x1f to 0x00, short to long transition, approximately 3.1 to 118200 msecs

RETURN VALUE

`KTTRUE`	if command write successful;
`KTFALSE`	otherwise.

FUNCTION

Sets the Low Pass Filter Envelope parameters for filter frequencies and transition rates

acDigiSetLfoRegisters N

Sets the pitch and amplitude Low Frequency Oscillators.

FORMAT

```
#include <ac.h>

KTBOOL acDigiSetLfoRegisters(KTU32 port, KTU32 lfoReset, KTU32 lfoFrequency, KTU32
pitchLfoWaveShape, KTU32 pitchLfoDepth, KTU32 ampLfoWaveShape,
KTU32 ampLfoDepth)
```

PARAMETERS

`KTU32 port`,	voice channel, 0-63 for Audio64, 0-15 if MidiDa driver
`KTU32 lfoReset`	Reset LFO at start of playback
`KTU32 lfoFrequency`	Set LFO frequency, 0.17 hz to 172.30 hz, 0x00 to 0x1f
`KTU32 pitchLfoWaveShape`	Set the pitch LFO waveshape to SAW, SQUARE, TRIANGLE or NOISE, 0x00 to 0x03 respectively
`KTU32 pitchLfoDepth`	Set the LFO depth to modulate pitch from (-3 to 2) cent ranges to (-231 to 202) cent ranges, 0x00 to 0x07
`KTU32 ampLfoWaveShape`	Set the Amplitude LFO waveshape to SAW, SQUARE, TRIANGLE or NOISE, 0x00 to 0x03 respectively
`KTU32 ampLfoDepth`	Set the amount of amplitude modulation from -0.4 db displacement to -24.0 db displacement, 0x00 to 0x07

RETURN VALUE

`KTTRUE`	if command write successful;
`KTFALSE`	otherwise.

FUNCTION

Sets the Low Frequency Oscillators for modulating playback characteristics (for pitch and amplitude).

acDigiSetLoopPoints

Sets the playback mode to stereo or mono.

FORMAT

```
#include <ac.h>

KTBOOL acDigiSetLoopPoints(KTU32 port, KTU32 loopStartAddress, KTU32 loopEndAddress)
explicitly set loop points for port
```

PARAMETERS

`KTU32 port`	
`KTU32 loopStartAddress`	Set a port's Loop Start Address(0x0000 to 0xfffd). Recommendation is to use offset value for ADPCM.
`KTU32 loopEndAddress`	Set a port's Loop End Address (0x0001 to 0xffff, Loop End must be > Loop Start Address)

RETURN VALUE

`KTTRUE`	if command write successful;
`KTFALSE`	otherwise.

FUNCTION

Override the default loop points of a port. Especially useful for `acDigiMultiPlay` if there are special loop requirements (like ADPCM)

acDigiSetPan

Adjusts the pan placement of a voice channel.

FORMAT

```
#include <ac.h>

KTBOOL acDigiSetPan(KTU32 port,KTU32 aicaPan)
```

PARAMETERS

`port`	Voice channel number, 0-63 for `audio64` driver, 0-15 for `MidiDa` driver.
	`aicaPan`0-31, right to left.

RETURN VALUE

`KTTRUE`	if successful
`KTFALSE`	if the port is out of range or command write fails.

FUNCTION

Changes the direct output pan of an open voice channel, if a sound is currently playing on the channel the pan of that sound will be changed, if the channel is not playing this will set the pan used when that channel is started with a call to `acDigiPlay()`.

Note: If `aicaPan` is out of range it will be set to `AC_MAX_PAN`.

acDigiSetSampleRate

Set the playback rate (sample rate) of audio stream.

FORMAT

```
#include <ac.h>

KTBOOL acDigiSetSampleRate(KTU32 port,KTS32 sampleRate)
```

PARAMETERS

`port`	Voice channel number, 0-63 for `audio64` driver, 0-15 for `MidiDa` driver.
`sampleRate`	The real world sample rate to set for the indicated the voice channel.

RETURN VALUE

`KTTRUE`	if successful
`KTFALSE`	if port is out of range, `sampleRate` exceeds 1128900, or command write fails.

FUNCTION

This changes sample rate (playback rate) of a currently running voice channel.

The voice will be set to the closest approximation of that sample rate the hardware is capable of reproducing.

acDigiSetVolume

Adjusts the channels volume.

FORMAT

```
#include <ac.h>

KTBOOL acDigiSetVolume(KTU32 port,KTU32 channelVolume)     channelVolume is 0-255
```

PARAMETERS

`port`	voice channel, 0-63 if DA driver, 0-15 if MIDI
`channelVolume`	the volume for the channel, 0-255

RETURN VALUE

`KTTRUE`	if command write successful
`KTFALSE`	otherwise.

FUNCTION

Adjusts the channels volume.

acDigiStop

Stops a voice channel playing.

FORMAT

```
#include <ac.h>

KTBOOL acDigiStop(KTU32 port)
```

PARAMETERS

`port`	`Voice channel number`, 0-63 for `audio64` driver, 0-15 for `MidiDa` driver.

RETURN VALUE

`KTTRUE`	if successful
`KTFALSE`	if the port is out of range or command write fails.

FUNCTION

Stops playback of a previously started voice channel.

acDmaInit

Initializes the ac layer DMA subsystem.

FORMAT

```
#include <ac.h>

KTBOOL acDmaInit(void)
```

PARAMETERS

void

RETURN VALUE

`KTTRUE`	if successful.
`KTFALSE,`	if a transfer proc has not been installed using the `acDmaInstallHandler()` call.

FUNCTION

CALLED BY: `acSystemInit()`

acDmaInstallHandler Ⓝ

Installs a monolithic DMA handler procedure.

FORMAT

```
#include <ac.h>

KTBOOL acDmaInstallHandler(AC_DMA_TRANSFER_PROC theTransferProc)
```

PARAMETERS

`AC_DMA_TRANSFER_PROC`	`theTransferProc`, a pointer to the handler procedure.

RETURN VALUE

`KTTRUE`	if successful.
`KTFALSE,`	if a handler is already installed.

FUNCTION

Installs a DMA handler proc that is used for stream DMA transfers. The transfer proc is prototyped in `ac.h` as follows:

```
typedef KTBOOL(*AC_DMA_TRANSFER_PROC)(             volatile KTU32 *,
volatile KTU32 *,
KTU32,
AC_DMA_TRANSFER_OPERATION,
AC_DMA_CALLBACK);
```

The functional definition is as follows:

```
KTBOOL MyDmaTransferProc(         volatile KTU32 *target,
volatile KTU32 *source,
KTU32 size,
AC_DMA_TRANSFER_OPERATION operation,
AC_DMA_CALLBACK callback)
```

The `AC_DMA_TRANSFER_OPERATION` argument controls which operation is executed using the AICA DMA. The modes of operation are enumerated in `ac.h` as follows:

```
typedef enum
{
AC_DMA_SUSPEND_ALL              =     7,            suspend all channels
AC_DMA_RESUME_ALL               =     8             resume all channels
}AC_DMA_TRANSFER_OPERATION;

typedef AC_DMA_TRANSFER_OPERATION * AC_DMA_TRANSFER_OPERATION_PTR;
```

An example of the DMA handler using the Shinobi library can be found in `MyDma.c.h`.

Note: If this proc is not installed `acSystemInit()` will fail when it calls `acDmaInit()`.

ICON LEGEND:	Ⓝ NEW function for R10	⦸ OBSOLETE function for R10	Ⓡ REVISED function for R10

acDmaResumeAll

Resumes all DMA activity.

FORMAT

```
#include <ac.h>

KTBOOL acDmaResumeAll(void)
```

PARAMETERS

void

RETURN VALUE

`KTTRUE`	if successful.
`KTFALSE`	if a transfer proc has not been installed using the `acDmaInstallHandler()` call.
	if the OS level DMA system failed to resume all.

FUNCTION

Resumes all DMA activity.

It is necessary when transferring data across the G2 bus to suspendresume all DMA activity, this call allows that operation to be performed. Prior to resumption of DMA the FIFO's are checked to assure that they are empty.

CALLED BY: `acG2Write()`, `acG2WriteLong()`

acDmaShutdown

Deinitializes the ac layer DMA subsystem.

FORMAT

```
#include <ac.h>

KTBOOL acDmaShutdown(void)
```

PARAMETERS

void

RETURN VALUE

`KTTRUE`	if successful.
`KTFALSE`	if a transfer proc has not been installed using the `acDmaInstallHandler()` call.

FUNCTION

CALLED BY: `acSystemShutdown()`

acDmaSuspendAll

Suspends all DMA activity.

FORMAT

```
#include <ac.h>

KTBOOL acDmaSuspendAll(void)
```

PARAMETERS

void

RETURN VALUE

`KTTRUE`	if successful.
`KTFALSE`	if a transfer proc has not been installed using the `acDmaInstallHandler()` call.
	if the OS level DMA system failed to suspend all.

FUNCTION

Suspends activity on the all DMA channels. This call checks the FIFO's after suspension to assure that the current frame of DMA activity has completed prior to its return.

It is necessary when transferring data across the G2 bus to suspendresume all DMA activity, this call allows that operation to be performed.

CALLED BY: `acG2Write()`, `acG2WriteLong()`

acDspInstallOutputMixer

Registers an output mixer patch with the driver.

FORMAT

```
#include <ac.h>

KTBOOL acDspInstallOutputMixer(KTU32 address,KTU32 sizeInBytes)
```

PARAMETERS

`KTU32 address,`	The address of the output mixer bank in sound memory.
`KTU32 sizeInBytes,`	The size in bytes of the output mixer bank.

RETURN VALUE

`KTTRUE`	if successful
`KTFALSE`	if address is `NULL`, size in bytes is 0 or command write failed...

FUNCTION

Sets an output mixer bank as the current output routing. This output mixer bank is currently produced using the Mac DSP editor tool.

acDspInstallProgram

Registers a dsp program bank with the driver.

FORMAT

```
#include <ac.h>

KTBOOL acDspInstallProgram(KTU32 address,KTU32 sizeInBytes)
```

PARAMETERS

`KTU32 address,`	The address of the program bank in sound memory.
`KTU32 sizeInBytes,`	The size in bytes of the program bank.

RETURN VALUE

`KTTRUE`	if successful
`KTFALSE`	if address is `NULL`, size in bytes is 0 or command write failed.

FUNCTION

Sets a DSP program bank as the current DSP program. This program bank is currently produced using the Mac DSP editor tool.

acDspSetMixerChannel ●

Sets DSP Mixer level and channel for that port.

FORMAT

```
#include <ac.h>

KTBOOL acDspSetMixerChannel(KTU32 port,KTU32 mixer,KTU32 level)
```

PARAMETERS

`port`	DA stream prot number, 0-63.
`mixer`	DSP mixer channel number, 0-15.
`level`	Audio signal level, 0-15.

RETURN VALUE

`KTTRUE`	if command write successful
`KTFALSE`	otherwise.

FUNCTION

Set port's audio signal to DSP mixer channel to enable DSP effects for stream. This allows stream to be altered by reverb, etc.

Note: The parameters' mixer and level are not error trapped

acDspSetQSoundAngle

Sets Q-Sound position.

FORMAT

```
#include <ac.h>

KTBOOL acDspSetQSoundAngle(KTU32 qSoundChannel,KTU32 angle)
```

PARAMETERS

`qSoundChannel`	The 0 based Q-Sound channel number, if the effect patch has 4 channels of qsound and they are mixer channels 12-16 then for the sound on mixer channel 12 the Q-Sound channel is 0, 13 = 1 etc... range: 0-7
`angle`	0-127, left to right.

RETURN VALUE

`KTTRUE`	if successful
`KTFALSE`	if command write fails

FUNCTION

Allows the setting of the Q-Sound angle parameter in real time.

Note: The parameter angle is trapped so that if it is out of range it will be set to 127 and the function will continue to execute.

acErrorClear

Clears the AC error structure.

FORMAT

```
#include <ac.h>

void acErrorClear(void)
```

PARAMETERS

```
void
```

RETURN VALUE

`KTTRUE`	if successful
`KTFALSE`	if unable to send command or interruptId is out of range (0-255).

FUNCTION

Clears the AC Error structure.

acErrorExists

Checks to see if an error condition exists.

FORMAT

```
#include <ac.h>

KTBOOL acErrorExists(void)
```

PARAMETERS

`void`

RETURN VALUE

`KTTRUE`	if a error exists
`KTFALSE`	if no error exists

FUNCTION

Allows checking of the error state for the AC layer in a single call returning a bool.

acErrorGetLast

Gets a pointer to the error structure.

FORMAT

```
#include <ac.h>

AC_ERROR_PTR acErrorGetLast(void)
```

PARAMETERS

`void`

RETURN VALUE

`AC_ERROR_STRUCT`	a pointer to the AC error structure.

FUNCTION

Gets a pointer to the AC error structure. This contains an error number enumerated as an `AC_ERROR_TYPE` in `ac.h` and a more informative error message that tells the name of the function that failed as well as some descriptive text regarding the cause of the failure.

acG2FifoCheck N

Checks the AICA and Holly FIFO bits.

FORMAT

```
#include <ac.h>

void acG2FifoCheck(void)
```

PARAMETERS

void

RETURN VALUE

void

FUNCTION

Used to check the AICA and Holly FIFO bits when doing writes on the G2 bus.

G2 Bus restriction Compliancy

To ensure no loss or corruption of data on the G2 bus certain countermeasures must be taken when reading and writing sound memory. These measures include checking the Holly and AICA FIFO's, disabling and enabling interrupts and stopping and resuming DMA transfers. All of the acG2 routines are compliant with the necessary countermeasures for reading and writing and all of the reading and writing of AICA memory done by the sound library is done using this interface.

acG2Fill

Used to fill sound memory.

FORMAT

```
#include <ac.h>

KTBOOL acG2Fill(KTU32 *target,KTU8 value,KTU32 sizeInBytes)
```

PARAMETERS

`KTU32 *target`,	The target address
`KTU8 value`,	The byte value with which to fill. (0xab is promoted to dword 0xabababab)
`KTU32 sizeInBytes`,	The size of the block to fill. (MUST be divisable by 4)

RETURN VALUE

`KTTRUE`	If the operation was successful
`KTFALSE`	If target is `NULL`
	If `sizeInBytes` < 4
	If `sizeInBytes` is not evenly divisible by 4
	If target is not divisible by 4

FUNCTION

Fills sound memory with dwods filled with the given byte value, a fill value of 0xab will cause the memory to be filled with dwords containing 0xabababab.

G2 Bus restriction Compliancy

To ensure no loss or corruption of data on the G2 bus certain countermeasures must be taken when reading and writing sound memory. These measures include checking the Holly and AICA FIFO's, disabling and enabling interrupts and stopping and resuming DMA transfers. All of the acG2 routines are compliant with the necessary countermeasures for reading and writing and all of the reading and writing of AICA memory done by the sound library is done using this interface.

acG2Read

Reads from sound memory.

FORMAT

```
#include <ac.h>

KTBOOL acG2Read(KTU32 *target,KTU32 *source,KTU32 sizeInBytes)
```

PARAMETERS

`KTU32 *  target`,	a pointer to the target buffer in SH4 memory
`KTU32 *  source`,	a pointer to the source buffer in sound (AICA) memory
`KTU32 sizeInBytes`,	the size of the block to transfer in bytes

RETURN VALUE

`KTTRUE`	If the operation was successful
`KTFALSE`,	If target or source is `NULL`
	If `sizeInBytes` < 4
	If `sizeInBytes` is not divisible by 4
	If `source` is not divisible by 4

FUNCTION

Reads a block of memory from a 4 byte aligned buffer to a 4 byte aligned target buffer the size of the data must also be a multiple of 4 bytes.

G2 Bus restriction Compliancy

To ensure no loss or corruption of data on the G2 bus certain countermeasures must be taken when reading and writing sound memory. These measures include checking the Holly and AICA FIFO's, disabling and enabling interrupts and stopping and resuming DMA transfers. All of the acG2 routines are compliant with the necessary countermeasures for reading and writing and all of the reading and writing of AICA memory done by the sound library is done using this interface.

acG2ReadLong

Reads a DWORD from AICA memory.

FORMAT

```
#include <ac.h>

KTBOOL acG2ReadLong(KTU32 *address,KTU32 *value)
```

PARAMETERS

`KTU32 *address`	The address to read.
`KTU32 *value`	The value read is returned via this pointer.

RETURN VALUE

`KTTRUE`	If the operation was successful
`KTFALSE`	If address is `NULL`
	If value is `NULL`
	If address is not divisible by 4

FUNCTION

Reads a single `DWORD` from AICA memory and returns the value via a pointer. The read is a double read to ensure that a simultanious write was not occuring wile the read was being performed.

G2 Bus restriction Compliancy

To ensure no loss or corruption of data on the G2 bus certain countermeasures must be taken when reading and writing sound memory. These measures include checking the Holly and AICA FIFO's, disabling and enabling interrupts and stopping and resuming DMA transfers. All of the `acG2` routines are compliant with the necessary countermeasures for reading and writing and all of the reading and writing of AICA memory done by the sound library is done using this interface.

acG2Write

Writes dword aligned data to sound memory.

FORMAT

```
#include <am.h>

void acG2Write(KTU32 *  target,KTU32 *  source,KTU32 sizeInBytes)
```

PARAMETERS

`KTU32 *  target`	a pointer to the target buffer
`KTU32 *  source`	a pointer to the source buffer
`KTU32 sizeInBytes`	the size of the block to transfer in bytes

RETURN VALUE

`KTTRUE`	If the operation was successful
`KTFALSE`	If target is `NULL`
	If source is `NULL`
	If target is not divisable by 4
	If source is not divisable by 4
	If sizeInBytes is not divisible by 4
	If sizeInBytes is less then 4

FUNCTION

Copys a block of memory from a 4 byte aligned buffer to a 4 byte aligned target buffer the size of the data must also be a multiple of 4 bytes.

G2 Bus restriction Compliancy

To ensure no loss or corruption of data on the G2 bus certain countermeasures must be taken when reading and writing sound memory. These measures include checking the Holly and AICA FIFO's, disabling and enabling interrupts and stopping and resuming DMA transfers. All of the acG2 routines are compliant with the necessary countermeasures for reading and writing and all of the reading and writing of AICA memory done by the sound library is done using this interface.

acG2WriteLong N

Writes a DWORD to AICA memory.

FORMAT

```
#include <ac.h>

KTBOOL acG2WriteLong(KTU32 *address,KTU32 value)
```

PARAMETERS

`KTU32 *address`	The address to write.
`KTU32 value`	The value to write.

RETURN VALUE

`KTTRUE`	If the operation was successful
`KTFALSE`	If address is `NULL`
	If address is not divisible by 4

FUNCTION

Writes a single DWORD to AICA memory.

G2 Bus restriction Compliancy

To ensure no loss or corruption of data on the G2 bus certain countermeasures must be taken when reading and writing sound memory. These measures include checking the Holly and AICA FIFO's, disabling and enabling interrupts and stopping and resuming DMA transfers. All of the acG2 routines are compliant with the necessary countermeasures for reading and writing and all of the reading and writing of AICA memory done by the sound library is done using this interface.

acGetSystemFlag

True if the system has been initialized.

FORMAT

```
#include <ac.h>

KTBOOL acGetSystemFlag(void)
```

PARAMETERS

void

RETURN VALUE

`KTTRUE`	If the driver has been installed and the system initialized.
Or...	
`KTFALSE`	If not.

FUNCTION

Returns `KTTRUE` if the function `acInstallDriver` has been run successfully.

acGetTransferMode N

Retrieves current transfer mode setting.

FORMAT

```
#include <ac.h>

KTBOOL acGetTransferMode(AC_TRANSFER_MODE *mode)
```

PARAMETERS

`mode`	Returns either `AC_TRANSFER_CPU` or `AC_TRANSFER_DMA`.

RETURN VALUE

`KTTRUE`	if mode change was successful
`KTFALSE`	otherwise.

FUNCTION

Retrieves current transfer mode setting. See `acSetTransferMode()`.

acIntInit

Initializes the ac interrupt system.

FORMAT

```
#include <am.h>

void acIntInit(void)
```

PARAMETERS

`void`

RETURN VALUE

`KTTRUE`	if the interrupt system was successfully initialized
`KTFALSE`	if OS based chain add or delete managers not installed, see the following functions to install these OS based services.
`KTBOOL`	`acIntInstallOsChainAddManager AC_INT_CHAIN_ADD_MANAGER theChainAddManager);`
`KTBOOL`	`acIntInstallOsChainDeleteManager (AC_INT_CHAIN_DELETE_MANAGER theChainDeleteManager);`

FUNCTION

Initializes the am interrupt system by installing the ARM interrupt handler to the OS's ARM interrupt chain.

If user interrupt handler and or callback handlers have been installed these will not be overwritten by this function.

See Also: `MyInt.c` (a part of the samples)

acIntInstallArmInterruptHandler

Installs an ARM interrupt handler.

FORMAT

```
#include <am.h>

KTBOOL acIntInstallArmInterruptHandler(AC_ARM_INTERRUPT_HANDLER theInterruptHandler)
```

PARAMETERS

`AC_ARM_INTERRUPT_HANDLER theInterruptHandler`a pointer to an interrupt handler function.

The prototype of the callback handler function is as follows:

```
void feeb(void *);
```

The value `AC_ARM_INTERRUPT_HANDLER_ID` will be incoming as the argument to this function if it is a legitimate interrupt message.

RETURN VALUE

`KTTRUE`	if the proc was installed.
`KTFALSE`	if the proc was not installed due to a prior initialization of the vector.

FUNCTION

Initializes the ARM interrupt handler vector with your function. A default function will be installed if this vector has not been initialized at startup time. This default handler is illustrated in `MyInt.c` and is described below.

The default ARM interrupt handler is installed into the OS's ARM external interrupt chain.

This is done at start up if a user handler has not been supplied via this routine.

It is invoked when ever an ARM interrupt is raised. The interrupt handler parses the drivers interrupt array to determine which channels are reporting on this interrupt cycle, it then calls the callback handler once for each message it finds in the drivers interrupt array.

See Also: `KTBOOL acSystemRequestArmInterrupt(KTU32 interruptId)`

See Also: `MyInt.c` (a part of the samples)

acIntInstallCallbackHandler

Installs a callback handler into the ARM interrupt handler.

FORMAT

```
#include <am.h>

KTBOOL acIntInstallCallbackHandler(AC_CALLBACK_HANDLER theCallbackHandler)
```

PARAMETERS

`AC_CALLBACK_HANDLER theCallbackHandler`, a pointer to a callback handler function.

The prototype of the callback handler function is as follows:

```
void     fuu(volatile KTU32);
```

RETURN VALUE

`KTTRUE`	if the proc was installed.
`KTFALSE`	if the proc was not installed due to a prior initialization of the vector.

FUNCTION

Installs a callback handler into the ARM interrupt handler. This allows developers wanting to work at the AC level to get callbacks from both the voice `channelsmidi` ports and from the interruptId arg to `acSystemRequestArmInterrupt()`.

Please note that the `audio64` driver claims the first 64 ID’s (0-63) while the `MidiDa` driver claims the first 32 (0-31) ID's. MIDI ports report the (port + 16) so in using the `MidiDa` driver 0-15 are the 16 available digital voice channels and 16-31 are the 16 available MIDI ports.

The ARM interrupt handler is installed into the OS’s ARM external interrupt chain. It is invoked when ever an ARM interrupt is raised. The interrupt handler parses the drivers interrupt array to determine which channels are reporting on this interrupt cycle, it then calls the callback handler once for each message it finds in the drivers interrupt array.

See Also: `KTBOOL acSystemRequestArmInterrupt(KTU32 interruptId)`

See Also: `MyInt.c` (a part of the samples)

acIntInstallOsChainAddManager

Installs proc pointer to interrupt chain add routine.

FORMAT

```
#include <am.h>

KTBOOL acIntInstallOsChainAddManager(AC_INT_CHAIN_ADD_MANAGER theChainAddManager)
```

PARAMETERS

`AC_INT_CHAIN_ADD_MANAGER theChainAddManager` a pointer to the wrapped routine.

The wrapper prototype is defined as follows:

```
KTU32 foo(KTS16,AC_ARM_INTERRUPT_HANDLER,KTU32,void *);
```

RETURN VALUE

`KTTRUE`	if the proc was installed.
`KTFALSE`	if the proc was not installed due to a prior initialization of the vector.

FUNCTION

This allows the app programmer to wrap a given OS's interrupt chain add routine and send it to the audio system. This provides for OS neutrality.

See Also: `MyInt.c` (a part of the samples)

acIntInstallOsChainDeleteManager

Installs pointer to interrupt chain delete routine.

FORMAT

```
#include <am.h>

KTBOOL acIntInstallOsChainDeleteManager(AC_INT_CHAIN_DELETE_MANAGER
theChainDeleteManager)
```

PARAMETERS

`AC_INT_CHAIN_DELETE_MANAGER` `theChainDeleteManager` a pointer to the wrapped routine.

The wrapper prototype is defined as follows:

```
void     feiux(KTU32);
```

RETURN VALUE

`KTTRUE`	if the proc was installed.
`KTFALSE`	if the proc was not installed due to a prior initialization of the vector.

FUNCTION

Allows installation of OS specific interrupt chain removal `procPointer`. The procedure's wrapper must have the following prototype: `void foo(KTU32);` the argument being the chain ID to be removed.

Note: This **MUST** be done prior to calling `acInit()` or `amInit()` or init failure will result.

See Also: `MyInt.c` (a part of the samples)

acIntSetAicaChainId

Sets the AICA interrupt chain ID.

FORMAT

```
#include <am.h>

void acIntSetAicaChainId(KTU32 chainId)
```

PARAMETERS

`KTU32 chainId`	The OS specific ID for the AICA EXTERNAL interrupt in the case of Shinobi it is 0xb20

RETURN VALUE

void

FUNCTION

Sets the interrupt ID for the AICA external interrupt, this is used when installing interrupt handlers.

Note: This defaults to 0xb20

See Also: `MyInt.c` (a part of the samples)

acIntShutdown

Shuts down the ac interrupt system.

FORMAT

```
#include <am.h>

void acIntShutdown(void)
```

PARAMETERS

`void`

RETURN VALUE

`void`

FUNCTION

Shuts down the am interrupt system by removing the ARM interrupt callback from the OS using the chain delete function and clearing the OS service vectors.

See Also: `MyInt.c` (a part of the samples)

acMidiClose

Close a MIDI port.

FORMAT

```
#include <ac.h>

KTBOOL acMidiClose(KTU32 port)
```

PARAMETERS

`port`MIDI port number, 0-15.

RETURN VALUE

`KTTRUE`	if successful
`KTFALSE`	if the port is out of range, or command write fails.

FUNCTION

Closes the indicated MIDI port and sends an `All Notes Off` message to the drivers midi parser.

acMidiOpen

Open a MIDI Port buffer for SMF format 0 playback.

FORMAT

```
#include <ac.h>

KTBOOL acMidiOpen(KTU32 port,
KTU8 gmMode,
KTU32 address,
KTU32 sizeInBytes,
KTU32 pulsesPerQuarterNote)
```

PARAMETERS

`port`	MIDI port number, 0-15.
`gmMode`	`AC_GM_ON` or `AC_GM_OFF`, selects General MIDI mode on or off. Allows use of a Bank 0 General MIDI instrument and drumset.
`address`	address in sound ram of the start of buffer.
`MidiBufferSize`	buffer length in bytes.
`TicksPerQNote`	time base in ticks per quarter note (ppqn).

RETURN VALUE

`KTTRUE`	if successful
`KTFALSE`	if the port is out of range, address is 0, sizeInBytes is 0 or command write fails.

FUNCTION

Opens a MIDI port. Midi ports are fully polyphonic 16 channel ports for MIDI streams. This call sets the default set of parameters for a MIDI port, i.e. GM mode, the address of the Standard MIDI Type 0 asset in sound memory and the PPQN (pulses per quarter note) for that asset.

Note: If `gmMode` is out of range it will be set to `AC_GM_OFF`

acMidiPause

Pauses an active MIDI port.

FORMAT

```
#include <ac.h>

KTBOOL acMidiPause(KTU32 port)
```

PARAMETERS

`port`MIDI port number, 0-15.

RETURN VALUE

`KTTRUE`	if successful
`KTFALSE`	if the port is out of range, or command write fails.

FUNCTION

Pauses playback on an activly playing MIDI port. Sends an `All Notes Off` message to the drivers midi parser.

acMidiPlay

Starts playback on opened MIDI port.

FORMAT

```
#include <ac.h>

KTBOOL acMidiPlay(KTU32 port,KTU32 startOffset, KTS16 aicaLoopFlag)
```

PARAMETERS

`port`	MIDI port number, 0-15.
`startOffset`	Start playback layback from buffer start position + offset.
`loopFlag`	Loop MIDI playback buffer, `AC_LOOP_ON` or `AC_LOOP_OFF`.

RETURN VALUE

`KTTRUE`	if successful
`KTFALSE`	if the port is out of range, or command write fails.

FUNCTION

Starts Standard MIDI File Type 0 playback on the given port from the start of the port's buffer plus startOffset. The default tempo is 120 BPM, until a MIDI tempo message is parsed.

Note: If `aicaLoopFlag` is out of range it will be set to `AC_LOOP_OFF`

acMidiRequestEvent

Generates interrupt to host upon MIDI port reaching specified address.

FORMAT

```
#include <ac.h>

KTBOOL acMidiRequestEvent(KTU32 port,KTU32 offsetFromBeginningInBytes)
```

PARAMETERS

`port`	MIDI port number, 0-15.
`portEventAddress`	Sound memory event address.

RETURN VALUE

`KTTRUE`	if successful
`KTFALSE`	if the port is out of range, or command write fails.

FUNCTION

When MIDI playback position reaches the indicated offset the driver will raise an ARM external interrupt causing the ARM interrupt handler to be invoked. The channel (port + 16) number of the caller will be placed into the drivers interrupt array. If multiple channelsports are reporting event requests at the same time there will be multiple entries in the drivers interrupt array. The number of channelsports reporting may be observed by measuring the incrementation of the interrupt array start offset which is available via the `acSystem` call `acSystemGetIntArrayStartOffset().`

Note: The parameter `offsetFromBeginningInBytes` is not error trapped.

acMidiReset

Resets MIDI controllers on port to default values.

FORMAT

```
#include <ac.h>

KTBOOL acMidiReset(KTU32 port)
```

PARAMETERS

`port`	MIDI port number, 0-15.

RETURN VALUE

`KTTRUE`	if successful
`KTFALSE`	if the port is out of range, or command write fails.

FUNCTION

Resets controller values to standard defaults. The bank select per channel is set to 0.

The MIDI continuous controller settings affected are as follows:

```
CC7                       = 100
CC11                      = 127
CC10, CC71, CC74          = 64
CC20-CC28, CC88           = 32
CC0, CC52-CC56, CC70      = 0
Pitch Bend                = 0 (center).
```

acMidiResume

Resumes playback on active MIDI port.

FORMAT

```
#include <ac.h>

KTBOOL acMidiResume(KTU32 port)
```

PARAMETERS

`port`	MIDI port number, 0-15.

RETURN VALUE

`KTTRUE`	if successful
`KTFALSE`	if the port is out of range, or command write fails.

FUNCTION

Resumes playback on the given MIDI port. When playback is resumed running status mode is retained from the point at which the sequence was paused.

acMidiSendMessage

Sends raw MIDI messages to ports.

FORMAT

```
#include <ac.h>

KTBOOL acMidiSendMessage(KTU32 port,
KTU32 channel,
KTU32 midiMessage,
KTU32 value1,
KTU32 value2)
```

PARAMETERS

`port`	MIDI port number, 0-15.
`channel`	MIDI channel number, 0-15.
`midiMessage`	MIDI command number (channel nibble ignored), Channel voice messages 0x80-0xe0.
`midiValue1`	First MIDI voice message data byte.
`midiValue2`	Second MIDI voice message data byte.

RETURN VALUE

`KTTRUE`	if successful
`KTFALSE`	if the port, channel, value1 or value2 is out of range, or command write fails.

FUNCTION

Immediately sends raw MIDI message to port by channel number. Allows real-time dynamic control of note-on and controller messages, etc.

Note: The parameter `midiMessage` is not error trapped.

acMidiSetTempo

Set playback tempo of MIDI port.

FORMAT

```
#include <ac.h>

KTBOOL acMidiSetTempo(KTU32 port,KTU32 microSecondsPerQuarterNote)
```

PARAMETERS

`port`	MIDI port number, 0-15.
`microSecondsPerQuarterNote`	Sets the Microseconds per Quarter Note for MIDI port.

RETURN VALUE

`KTTRUE`	if successful
`KTFALSE`	if the port is out of range, or command write fails.

FUNCTION

Allows real-time control of tempo for port.

Note: The parameter `microSecondsPerQuarterNote` is not error trapped.

acMidiSetTonebank

Assign a MIDI Program Bank (tonebank) to an active bank slot.

FORMAT

```
#include <ac.h>

KTBOOL acMidiSetTonebank(KTU8 toneBank,
AC_BANK_TYPE bankType,
KTU32 address,
KTU32 sizeInBytes,
KTU32 mttPtr)
```

PARAMETERS

`toneBank`	tone bank slot number (0-15)
`bankType`	`AC_MELODIC_BANK` for melodic banks or `AC_DRUM_BANK` for drum banks.
`address`	offset in sound memory of start of tone bank.
`offset=`	(`addressInSoundMemory` & 0x003fffff)
`sizeInBytes`	size of tone bank in bytes
`mttPtr`	MIDI translate table pointer (not implemented yet)

RETURN VALUE

`KTTRUE`	if successful
`KTFALSE`	if the `toneBank` or `bankType` is out of range, address is 0, sizeInBytes is 0 or command write fails.

FUNCTION

Sets a tonebank for active playback. Assigns a bank number to a tonebank slot that will be accessable via MIDI Bank Select messages in the sequence data.

acMidiSetVolume

Sets scaled volume setting for MIDI port.

FORMAT

```
#include <ac.h>

KTBOOL acMidiSetVolume(KTU32 port,KTU32 portMasterVolume)
```

PARAMETERS

`port`	MIDI port number, 0-15.
`portMasterVolume`	Global volume setting for port, (0-127).

RETURN VALUE

`KTTRUE`	if successful
`KTFALSE`	if the port is out of range, or command write fails.

FUNCTION

Sets global volume setting for the given MIDI port. This will cause the driver to scale all MIDI CC7 (volume) messages accordingly. This will affect ALL channels in the sequence running on the port.

Note: If `portMasterVolume` is out of range it will be set to `AC_MAX_MIDI_VOLUME`.

acMidiStop

Stops standard MIDI file playback on port.

FORMAT

```
#include <ac.h>

KTBOOL acMidiStop(KTU32 port)
```

PARAMETERS

`port`	MIDI port number, 0-15.

RETURN VALUE

`KTTRUE`	if successful
`KTFALSE`	if the port is out of range, or command write fails.

FUNCTION

Stops MIDI playback on the indicated port and sends an “All Notes Off” message to the drivers midi parser.

acSetTransferMode

Indicates use of CPUDMA for memory transfers.

FORMAT

```
#include <ac.h>

KTBOOL acSetTransferMode(AC_TRANSFER_MODE mode)
```

PARAMETERS

`mode`	Either `AC_TRANSFER_CPU` (default) or `AC_TRANSFER_DMA`.

RETURN VALUE

`KTTRUE`	if mode change was successful
`KTFALSE`	otherwise.

FUNCTION

Indicates use of SH4-CPU or AICA-DMA for memory transfers between SH4 memory and AICA memory.

acSystemCheckDriverRevision

Tests the driver version against the supplied version.

FORMAT

```
#include <ac.h>

KTBOOL acSystemCheckDriverRevision(KTU8 *driver,KTU8 major,KTU8 minor,KTCHAR local)
```

PARAMETERS

`KTU8 *driver`	An image in memory of the driver
`KTU8 major`	The major revision desired
`KTU8 minor`	The minor revision desired
`KTCHAR local`	The local version desired

RETURN VALUE

`KTTRUE`	if it is the same version
`KTFALSE`	if it is not the same version

FUNCTION

Used in `acSystemInstallDriver()` to test the driver revision.

See: The top of `ac.h` for the constants that it uses to test the driver.

acSystemDelay

Use to delay for short periods of time.

FORMAT

```
#include <ac.h>

void acSystemDelay(KTU32 delay)
```

PARAMETERS

`KTU32 delay`	the number of NOP's of delay.

RETURN VALUE

void

FUNCTION

Uses a loop with a no-op in it to delay for short periods of time, used to allow memory to "settle" or for ARM writes to take place fully when critical values are read from sound memory.

acSystemDisableArmInterrupts

Use to disable the ARM interrupt.

FORMAT

```
#include <ac.h>

void acSystemDisableArmInterrupts(void)
```

PARAMETERS

void

RETURN VALUE

void

FUNCTION

Disables the ARM external interrupt. The driver does not enable or disable the interrupt this allows the interrupt to be enabled\disabled in critical sections.

acSystemEnableArmInterrupts

Use to enable the ARM interrupt.

FORMAT

```
#include <ac.h>

void acSystemEnableArmInterrupts(void)
```

PARAMETERS

void

RETURN VALUE

void

FUNCTION

Enables the ARM external interrupt. The driver does not enable or disable the interrupt this allows the interrupt to be enabled\disabled in critical sections.

acSystemGetBaseOfSoundMemory

Gets the starting address for sound memory.

FORMAT

```
#include <ac.h>

KTBOOL acSystemGetBaseOfSoundMemory(KTU32 *baseOfSoundMemory)
```

PARAMETERS

`KTU32 *baseOfSoundMemory` the address of the base of sound memory represented as a `KTU32`

RETURN VALUE

`KTTRUE`	on success.
`KTFALSE`	if the `baseOfSoundMemory` is `NULL` or driver is not installed.

FUNCTION

Gets the address of the base of sound memory represented as a KTU32.

acSystemGetCommandFlag

Gets address of driver command flag register.

FORMAT

```
#include <ac.h>

volatile KTU32 * acSystemGetCommandFlag(void)
```

PARAMETERS

void

RETURN VALUE

a pointer to the command flag register

FUNCTION

Gets the command flag register address for system usage. The command flag is written after commands are placed into the drivers command queue to indicate to the driver that there are commands to be processed.

When setting the flag the value should be 0xffffffff, the driver will then start processing the command queue from its last queue position. When writing commands it is necessary to observe the state of the G2 buss FIFO to ensure that the command write has completed prior to setting the command flag.

acSystemGetDriverRevision

Tests the driver version against the supplied version.

FORMAT

```
#include <ac.h>

KTBOOL acSystemGetDriverRevision(KTU8 *driver,KTU8 *major,KTU8 *minor,KTCHAR *local)
```

PARAMETERS

`KTU8 *driver`	An image in memory of the driver
`KTU8 *major`	The major revision is returned via this pointer
`KTU8 *minor`	The minor revision is returned via this pointer
`KTCHAR *local`	The local version is returned via this pointer

RETURN VALUE

`KTTRUE`	the version was returned intact
`KTFALSE`	if driver was `NULL`

FUNCTION

Called by `acSystemCheckDriverRevision` to obtain the driver revision.

acSystemGetFirstFreeSoundMemory Gets the address of first free sound memory.

FORMAT

```
#include <ac.h>

volatile KTU32 * acSystemGetFirstFreeSoundMemory(void)
```

PARAMETERS

void

RETURN VALUE

a pointer to the first free memory in the sound heap as obtained from the driver.

FUNCTION

Gets the address of the first free memory in the sound memory area as specified by the driver.

acSystemGetIntArray

Gets the address of the SH4 side interrupt message array.

FORMAT

```
#include <ac.h>

KTBOOL acSystemGetIntArray(char **interruptArray)
```

PARAMETERS

none

RETURN VALUE

`KTTRUE`	on success
`KTFALSE`	`*interruptArray` is `NULL` or driver is not installed.

FUNCTION

Gets the address of the SH4 side interrupt message array buffer that is contained in the `acSystem` structure.

Note: This is broken in R8 as it gets the SH4 side array but does not fill it from the driver.

acSystemGetIntArrayLength

Gets the length of the drivers interrupt message array.

FORMAT

```
#include <ac.h>

KTBOOL acSystemGetIntArrayLength(KTU32 *interruptArrayLength)
```

PARAMETERS

`KTU32 *interruptArrayLength`, the length of the interrupt message array is returned via this pointer.

RETURN VALUE

`KTTRUE`	on success
`KTFALSE`	if `interruptArrayLength` is `NULL` or the driver is not installed.

FUNCTION

Gets the length of the drivers interrupt message array.

Note: This is a vestigal function from when the Midi driver had a shorter message array then the DA driver. Now they both use 64 byte arrays.

acSystemGetIntArrayStartOffset

Gets the interrupt array write cursor offset.

FORMAT

```
#include <ac.h>

KTBOOL acSystemGetIntArrayStartOffset(KTU32 *interruptArrayStartOffset)
```

PARAMETERS

`KTU32 *interruptArrayStartOffset,` a byte pointer expressed as a KTU32 indicating the current write position within the drivers interrupt message array.

RETURN VALUE

`KTTRUE`	if successful
`KTFALSE`	if `interruptArrayStartOffset` is `NULL` or driver is not installed.

FUNCTION

Gets the interrupt array start offset from the driver. By comparing the movement of this number from interrupt to interrupt it can be determined how many messages are being returned and where they are located in the drivers interrupt message array.

The `audio64` driver claims the first 64 ID's (0-63) while the `MidiDa` driver claims the first 32 (0-31) ID's. MIDI ports report the (port + 16) so in using the `MidiDa` driver 0-15 are the 16 available digital voice channels and 16-31 are the 16 available MIDI ports.

See Also: `MyInt.c` (a part of the samples)

acSystemGetIntArrayStartPtr

Gets a pointer to the start of the drivers interrupt message array.

FORMAT

```
#include <ac.h>

KTBOOL acSystemGetIntArrayStartPtr(char **intArrayStartPointer)
```

PARAMETERS

`char **intArrayStartPointer` a pointer to the start of the interrupt message array.

RETURN VALUE

`KTTRUE`	on success
`KTFALSE`	if drive is not installed or `*intArrayStartPointer` is `NULL`

FUNCTION

Gets a pointer to the start of the 64 byte interrupt message array in the driver.

This address is in `SOUND` memory so `NO BYTE READS` move it into SH4 memory before you start to dissect it in a byte wise fashion or sound memory will be turned into putty.

The `audio64` driver claims the first 64 ID's (0-63) while the `MidiDa` driver claims the first 32 (0-31) ID's. MIDI ports report the (port + 16) so in using the `MidiDa` driver messages 0-15 are the 16 available digital voice channels and 16-31 are the 16 available MIDI ports.

acSystemInit

Makes the ac system ready to use.

FORMAT

```
#include <ac.h>

KTBOOL acSystemInit(void)
```

PARAMETERS

void

RETURN VALUE

`KTBOOL`

FUNCTION

Makes the ac system ready to use, must be called prior to any `AC lib` calls.

acSystemInstallDriver

Installs the sound driver.

FORMAT

```
#include <ac.h>

KTBOOL acSystemInstallDriver(void)
```

PARAMETERS

void

RETURN VALUE

`KTBOOL`, `KTTRUE`	if the driver was successfully installed and started

FUNCTION

Installs the AICA driver image and sets the system data structure.

acSystemRequestArmInterrupt

Causes the driver to raise an ARM external interrupt.

FORMAT

```
#include <ac.h>

KTBOOL acSystemRequestArmInterrupt(KTU32 interruptId)
```

PARAMETERS

`KTU32 interruptId,` This value will be reported into the callback handler as its arg (0-255).

RETURN VALUE

`KTTRUE`	if successful
`KTFALSE`	if unable to send command or interruptId is out of range (0-255).

FUNCTION

Will raise an ARM external interrupt that will be fielded by the ARM interrupt handler on the SH4 side.

Note: Under the `audio64` DA driver the first 64 ID's (0-63) are taken for use by the 64 voice channels; the `MidiDa` driver will use the first 32 (0-31) ID’s to report the voice’sports.

The remaining ID’s are available for USER application purposes.

acSystemResetArmInterrupt

Resets the ARM interrupt flag.

FORMAT

```
#include <ac.h>

void acSystemResetArmInterrupt(void)
```

PARAMETERS

void

RETURN VALUE

void

FUNCTION

Resets the ARM interrupt status flag

acSystemSetMasterVolume N

Sets the master volume for all digital sound.

FORMAT

```
#include <ac.h>

KTBOOL acDigiSetMasterVolume(KTU32 masterVolume)    0-15
```

PARAMETERS

`KTU32 masterVolume`	The global volume level for the sound system

RETURN VALUE

`KTTRUE`	if command write successful
`KTFALSE`	otherwise.

FUNCTION

Sets the master volume for all sound (digital, MIDI, CD).

acSystemSetStereoOrMono

Sets the playback mode to stereo or mono.

FORMAT

```
#include <ac.h>

KTBOOL acSystemSetStereoOrMono(KTU32 playbackMode) mode is 0 for stereo, 1 for mono
```

PARAMETERS

`KTU32 playbackMode`	Set to 0 if Stereo playback, 1 if Mono playback

RETURN VALUE

`KTTRUE`	if command write successful
`KTFALSE`	otherwise.

FUNCTION

Set the audio hardware to playback Stereo or Mono.

acSystemShutdown

Shuts down the AC layer.

FORMAT

```
#include <ac.h>

void acSystemShutdown(void)
```

PARAMETERS

void

RETURN VALUE

void

FUNCTION

Shuts down the AC layer by removing the interrupt handler using the delete chain vector and clearing all of the OS service vectors.

acSystemWaitUntilG2FifoIsEmpty

Waits until the G2 FIFO is clear.

FORMAT

```
#include <ac.h>

void acSystemWaitUntilG2FifoIsEmpty(void)
```

PARAMETERS

void

RETURN VALUE

void

FUNCTION

The G2 FIFO is 32 bytes deep, when writing critical messages to sound RAM the FIFO status must be checked to determine when the write is complete. For each check that it makes of the FFST bits it increments a counter to allow real time observation of the amount of waiting required.

acTransfer

Requests a transfer using the current transfer mode.

FORMAT

```
#include <ac.h>

KTBOOL acTransfer(AC_TRANSFER_REQUEST *request)
```

PARAMETERS

RequestPointer to a caller allocated `AC_TRANSFER_REQUEST` block.

See below.

RETURN VALUE

`KTTRUE`	if mode change was successful
`KTFALSE`	otherwise.

FUNCTION

Requests a transfer using the arguments contained in the request block.

```
typedef struct             caller allocated
{
KTU32 *aicaMem;          AICA memory address (32-byte aligned)
KTU32 *sh4Mem;           SH4 memory address (32-byte aligned)
KTU32 size;              size (multiple of 32-bytes)
KTU32 direction;         direction (0: SH4->AICA, 1: AICA->SH4)
KTU32 status;            AC_TRANSFER_*
AC_TRANSFER_CALLBACK callback; user callback or NULL
} AC_TRANSFER_REQUEST;
```

Ownership of this request block switches to the internal transfer manager until the status is changed to either `AC_TRANSFER_COMPLETE` or `AC_TRANSFER_FAILED`. The caller must not alter the request block until this occurs.

If the transfer mode is `AC_TRANSFER_DMA`, the caller may optionally supply a pointer to a callback handler for the transfer. When the transfer mode is `AC_TRANSFER_CPU`, any supplied callback handler is ignored.

```
typedef void (*AC_TRANSFER_CALLBACK)(AC_TRANSFER_REQUEST *);
```

The installed handler is called passing a pointer to original request block. If additional data is required, the caller must setup this information before calling `acTransfer()`.

```
Example:              typedef struct
{
AC_TRANSFER_REQUEST request;
KTU32 idObject;
...
} CALLER_TRANSFER_BLOCK;
CALLER_TRANSFER_BLOCK xfer;
...
if (acTransfer((AC_TRANSFER_REQUEST*)&xfer)) ...

void caller_handler(AC_TRANSFER_REQUEST *xfer)
{
if ((CALLER_TRANSFER_BLOCK*)xfer)->idObject == idFoo) ...
...
}
```

3. The AICA Manager API

amBankFetchAsset

Fetches an asset from a `katbank`.

FORMAT

```
#include <am.h>
KTBOOL amBankFetchAsset(                                AM_BANK_PTR theBank,
AM_BANK_FILE_UNION_PTR parameters,
KTU32 assetNumber,
KTU32 **theAsset,
KTU32 *assetSize
)
```

PARAMETERS

`AM_BANK_PTR theBank,`	A pointer to a `.kat` bank.
`AM_BANK_FILE_UNION_PTR parameters,`	The parameter block is returned via this pointer.
`KTU32 assetNumber,`	The number of the asset.
`KTU32 **theAsset,`	A pointer to the asset is returned via this handle.
`KTU32 *assetSize`	The assets size is returned via this pointer.

RETURN VALUE

`zumber` is not in this bank,

FUNCTION

Fetches an asset from a `katbank` aggregation. Returns the size, parameters and a pointer to data via the arguments.

amBankFetchAssetParameters

Fetches parameters from any `katbank` asset.

FORMAT

```
#include <am.h>
KTBOOL amBankFetchAssetParameters          (AM_BANK_PTR theBank,
KTU32 assetNumber,
AM_BANK_FILE_UNION_PTR parameters
)
```

PARAMETERS

`AM_BANK_PTR theBank`	A pointer to a `.kat` bank.
`KTU32 assetNumber`	The number of the asset.
`AM_BANK_FILE_UNION_PTR parameters`	The parameter block is returned via this pointer.

RETURN VALUE

`KTTRUE`	on success
`KTFALSE`	`theBank` is `NULL`, parameters is `NULL`, `assetNumber` is not in this bank

FUNCTION

This will fetch the parameter block from any type of `katbank` asset.

amBankFetchMidiGmModeFlag

Fetches GM mode flag from a MIDI type `katbank` asset.

FORMAT

```
#include <am.h>
KTBOOL    amBankFetchMidiGmModeFlag(AM_BANK_PTR theBank,KTU32 assetNumber,KTU32
*gmModeFlag)
```

PARAMETERS

`AM_BANK_PTR theBank`	A pointer to a `.kat` bank.
`KTU32 assetNumber`	The number of the asset.
`KTU32 *gmModeFlag`	The GM mode of the asset is returned via this pointer.

RETURN VALUE

`KTTRUE`	on success
`KTFALSE`	`theBank` is `NULL`, `gmModeFlag` is `NULL`, `assetNumber` is not in this bank `assetNumber` is not a `MIDI` asset

FUNCTION

This fetches the value set via the `GmMode` tag in the `katbank` build script file. This should be set to 1 if it is a GM sequence or 0 if it is not.

amBankFetchMidiLoop

Fetches the loop flag from a MIDI type asset.

FORMAT

```
#include <am.h>
KTBOOL amBankFetchMidiLoop(AM_BANK_PTR theBank,KTU32 assetNumber,KTU32 *loop)
```

PARAMETERS

`AM_BANK_PTR theBank`	A pointer to a `.kat` bank.
`KTU32 assetNumber`	The number of the asset.
`KTU32 *loop`	The loop flag is returned via this pointer.

RETURN VALUE

`KTTRUE`	on success
`KTFALSE`	`theBank` is `NULL` loop is `NULL` `assetNumber` is not in this bank `assetNumber` is not a `MIDI` asset

FUNCTION

Fetches the loop flag of a midi asset in a bank file.

amBankFetchMidiPpqn

Fetches ppqn from a MIDI type `katbank` asset.

FORMAT

```
#include <am.h>
KTBOOL    amBankFetchMidiPpqn(AM_BANK_PTR theBank,KTU32 assetNumber,KTU32 *ppqn)
```

PARAMETERS

`AM_BANK_PTR theBank`	A pointer to a `.kat` bank.
`KTU32 assetNumber`	The number of the asset.
`KTU32 *ppqn`	The ppqn is returned via this pointer.

RETURN VALUE

`KTTRUE`	on success
`KTFALSE`	`theBank` is `NULL` ppqn is `NULL` `assetNumber` is not in this bank `assetNumber` is not a `MIDI` asset

FUNCTION

Gets the ppqn(pulses per quarter note) from a SMF Type 0 `MIDI` file asset in a bank.

amBankFetchMidiUspqn

Fetches uspqn from a MIDI type asset.

FORMAT

```
#include <am.h>

KTBOOL    amBankFetchMidiUspqn(AM_BANK_PTR theBank,KTU32 assetNumber,KTU32 *uspqn)
```

PARAMETERS

`AM_BANK_PTR theBank`	A pointer to a `.kat` bank
`KTU32 assetNumber`	The number of the asset
`KTU32 *uspqn`	The the microseconds pqn is returned via this pointer

RETURN VALUE

`KTTRUE`	on success

`KTFALSE`	`theBank` is `NULL`
	`uspqn` is `NULL`
	`assetNumber` is not in this bank
	`assetNumber` is not a `MIDI` asset

FUNCTION

Fetches the microseconds per quarter note (uspqn) of a midi asset in a bank file.

amBankFetchMidiVolume

Fetches master volume from a MIDI type `katbank` asset.

FORMAT

```
#include <am.h>
KTBOOL    amBankFetchMidiVolume(AM_BANK_PTR theBank,KTU32 assetNumber,KTU32
*masterVolume)
```

PARAMETERS

`AM_BANK_PTR theBank`	A pointer to a `.kat` bank
`KTU32 assetNumber`	The number of the asset
`KTU32 *masterVolume`	The master volume of the asset is returned via this pointer

RETURN VALUE

`KTTRUE`	on success

`KTFALSE`	`theBank` is `NULL`
	`masterVolume` is `NULL`
	`assetNumber` is not in this bank
	`assetNumber` is not a `MIDI` asset

FUNCTION

Fetches the master volume setting from a `MIDI` type `katbank` asset. This setting is set in the `katbank` build script file (`.oss`) via the"Volume" tag and is used to set the overall starting volume of a MIDI sequence. This allows the volumes of the sequences used in a game to be balanced against each other.

amBankFetchUnknownParameters

Fetches one of the 7 user parameters from a `katbank` "unknown" type asset.

FORMAT

```
#include <am.h>
KTBOOL amBankFetchUnknownParameters       (   AM_BANK_PTR theBank,
KTU32 assetNumber,
KTU32 parameterNumber,
KTS32 *parameterValue
)
```

PARAMETERS

`AM_BANK_PTR theBank`	A pointer to a `.kat` bank
`KTU32 assetNumber`	The number of the asset
`KTU32 parameterNumber`	The parameter to fetch (0-7)
`KTBOOL *parameterValue`	The parameter value is returned via this pointer

RETURN VALUE

`KTTRUE`	on success

`KTFALSE`	`theBank` is `NULL`
	`parameterValue` is `NULL`
	`assetNumber` is not in this bank
	`parameterNumber` is out of range
	`assetNumber` is not a `UNKNOWN` asset

FUNCTION

Fetches one of the seven user parameters from a `katbank` asset. These parameters are defined in the `katbank` build script using the `Parameter0` to `Parameter7` tags.

amBankFetchWaveBitDepth

Fetches the bit depth of a WAVE type asset in a `katbank`.

FORMAT

```
#include <am.h>
KTBOOL amBankFetchWaveBitDepth(AM_BANK_PTR theBank,KTU32 assetNumber,KTU32 *bitDepth)
```

PARAMETERS

`AM_BANK_PTR theBank`	A pointer to a `.kat` bank
`KTU32 assetNumber`	The number of the asset
`KTBOOL *bitDepth`	The bit depth is returned via this pointer

RETURN VALUE

`KTTRUE`	on success

`KTFALSE`	`theBank` is `NULL`
	`bitDepth` is `NULL`
	`assetNumber` is not in this bank
	`assetNumber` is not a MIDI asset

FUNCTION

Fetches the bit depth of a WAVE type asset in a `katbank`.

amBankFetchWaveLoopFlag

Fetches the loop flag from a `katbank` asset.

FORMAT

```
#include <am.h>
KTBOOL    amBankFetchWaveLoopFlag(AM_BANK_PTR theBank,KTU32 assetNumber,KTBOOL
*loopFlag)
```

PARAMETERS

`AM_BANK_PTR theBank`	A pointer to a `.kat` bank
`KTU32 assetNumber`	The number of the asset
`KTBOOL *loopFlag`	The loop flag value is returned via this pointer

RETURN VALUE

`KTTRUE`	on success

`KTFALSE`	`theBank` is `NULL`
	`loopFlag` is `NULL`
	`assetNumber` is not in this bank
	`assetNumber` is not a MIDI asset

FUNCTION

Fetches the loop flag from a WAVE type `katbank` asset. The loop flag is set in the `katbank` build script via the “Loop” tag. If the wave is to loop the value is set to 1 if not it is set to 0.

amBankFetchWaveRandomPitch

Fetches random pitch amount from a `katbank` asset.

FORMAT

```
#include <am.h>
KTBOOL    amBankFetchWaveRandomPitch(AM_BANK_PTR theBank,KTU32 assetNumber,KTU32
*randomPitchAmount)
```

PARAMETERS

`AM_BANK_PTR theBank`	A pointer to a `.kat` bank
`KTU32 assetNumber`	The number of the asset
`KTBOOL *randomPitchAmount`	The random pitch amount is returned via this pointer

RETURN VALUE

`KTTRUE`	on success
`KTFALSE`	`theBank` is `NULL` randomPitchAmount is `NULL` `assetNumber` is not in this bank `assetNumber` is not a MIDI asset

FUNCTION

Fetches the random pitch amount from a WAVE type `katbank` asset. This amount will be applied as a random percentage of change from the root pitch of the sound when it is played using the `amSound`... interface.

amBankFetchWaveSampleRate

Fetches the sample rate from a `katbank` WAVE asset.

FORMAT

```
#include <am.h>
KTBOOL amBankFetchWaveSampleRate(AM_BANK_PTR theBank,KTU32 assetNumber,KTU32
*sampleRate)
```

PARAMETERS

`AM_BANK_PTR theBank`	A pointer to a `.kat` bank
`KTU32 assetNumber`	The number of the asset
`KTBOOL *sampleRate`	The real world sample rate is returned via this pointer

RETURN VALUE

`KTTRUE`	on success

`KTFALSE`	`theBank` is `NULL`
	`sampleRate` is `NULL`
	`assetNumber` is not in this bank
	`assetNumber` is not a MIDI asset

FUNCTION

Fetches the sample rate from a `katbank` WAVE asset. This is the real world sample rate number that is set in the `katbank` build script (.oss) using the “SampleRate” tag.

amBankGetAssetSize

Gets the size of an asset from a `katbank`.

FORMAT

```
#include <am.h>
KTBOOL amBankGetAssetSize(AM_BANK_PTR theBank,KTU32 assetNumber,KTU32 *assetSize)
```

PARAMETERS

`AM_BANK_PTR theBank`	A pointer to a `.kat` bank
`KTU32 assetNumber`	The number of the asset
`KTU32 *assetSize`	The size of the asset is returned via this pointer

RETURN VALUE

`KTTRUE`	on success

`KTFALSE`	`theBank` is `NULL`
	`assetSize` is `NULL`
	`assetNumber` is not in this bank

FUNCTION

Fetches the size of an asset from a `katbank`.

amBankGetHeaderSize

Gets the size of the header portion of a `katbank`.

FORMAT

```
#include <am.h>
KTBOOL amBankGetHeaderSize(AM_BANK_PTR theBank,KTU32 *headerSize)
```

PARAMETERS

`KTU8 *theBank`	A pointer to either the header from a bank file or an entire bank file
`KTU32 *headerSize`	The size of the `katbank` header is returned via this pointer

RETURN VALUE

`KTTRUE`	on success

`KTFALSE`	`theBank` is `NULL`
	`headerSize` is `NULL`
	`assetNumber` is not in this bank

FUNCTION

Gets the size of the header portion of a `katbank`.

amBankGetNumberOfAssets

Gets the number of assets in a `katbank`.

FORMAT

```
#include <am.h>
KTBOOL amBankGetNumberOfAssets(AM_BANK_PTR theBank,KTU32 *numberOfAssets)
```

PARAMETERS

`KTU8 *theBank`	A pointer to either the header from a bank file or an entire bank file
`KTU32 *numberOfAssets`	The number of assets in the `katbank` is returned via this pointer

RETURN VALUE

`KTTRUE`	on success

`KTFALSE`	`theBank` is `NULL`
	`assetSize` is `NULL`
	`assetNumber` is not in this bank

FUNCTION

Gets the number of assets in a `katbank` file.

amBankLoad

Loads a `katbank` asset from disk into sound memory.

FORMAT

```
#include <am.h>
KTBOOL amBankLoad(KTSTRING fileName,AM_BANK_PTR buffer)
```

PARAMETERS

`KTSTRING fileName`	The filename and path of the bank to load
`AM_BANK_PTR buffer`	A 32 byte aligned buffer in sound memory big enough to hold the asset

RETURN VALUE

`KTTRUE`	on success
`KTFALSE`	`fileName` is `NULL` File not found buffer is `NULL` buffer is not 32 byte aligned

FUNCTION

Loads a `katbank` asset from disk into sound memory. This calls the redirectable file system (`amFile`...) to do the loading operation.

amDmaMemCpy

Performs DMA copys to sound memory.

FORMAT

```
#include <am.h>
KTBOOL amDmaMemCpy(KTU32 *target, KTU32 *source, KTU32 size,KTU32 bytesPerTransfer,KTU32
dmaChannel)
```

PARAMETERS

`KTU32 *target`	The target buffer, must be large enough to hold size bytes
`KTU32 *source`	The source buffer
`KTU32 size`	The number of bytes to transfer
`KTU32 bytesPerTransfer,`	The number of bytes to transfer in one DMA frame
`KTU32 dmaChannel`	`AM_DMA_CHANNEL` only for now

RETURN VALUE

`KTTRUE`	On success
`KTFALSE`	target or source is `NULL`, `size` is 0 `bytesPerTransfer` is not 1,2,4,8 or 32 `dmaChannel` is not `AM_DMA_CHANNEL`

FUNCTION

Note: This is not implemented in R8, the function will simply return false with an AC error condition.

Copies memory from one place to the other starting at the bottom of the block. The source target and size must be multiples of `bytesPerTransfer` or failure will result. The transfer is made in burst mode rather then cycle steal mode as timelyness is important to streaming audio processes.

amDspFetchOutputBank

Fetches and installs a DSP output bank from a `KatBank` asset.

FORMAT

```
#include <am.h>
KTBOOL amDspFetchOutputBank(AM_BANK_PTR theBank,KTU32 assetNumber)
```

PARAMETERS

`AM_BANK_PTR theBank`	A pointer to a `.kat` bank
`KTU32 assetNumber`	The number of the asset

RETURN VALUE

`KTTRUE`	On success
`KTFALSE`	`theBank` is `NULL` `assetNumber` is not in this bank unable to send driver command

FUNCTION

Fetches and installs a DSP output bank from a `KatBank` asset.

amDspFetchProgramBank

Fetches and installs a DSP program bank from a KatBank asset.

FORMAT

```
#include <am.h>
KTBOOL amDspFetchProgramBank(AM_BANK_PTR theBank,KTU32 assetNumber)
```

PARAMETERS

`AM_BANK_PTR theBank,`	A pointer to a `.kat` bank.
`KTU32 assetNumber,`	The number of the asset.

RETURN VALUE

`KTTRUE`	On success
`KTFALSE`	`theBank` is `NULL` `assetNumber` is not in this bank unable to send driver command

FUNCTION

Fetches and installs a DSP program bank from a `KatBank` asset.

amErrorClear

Clears the AM error structure.

FORMAT

```
#include <ac.h>
void amErrorClear(void)
```

PARAMETERS

void

RETURN VALUE

`KTTRUE`	if successful
`KTFALSE`	if unable to send command or interruptId is out of range (0-255)

FUNCTION

Clears the AM Error structure.

amErrorExists

Checks to see if an error condition exists.

FORMAT

```
#include <ac.h>
KTBOOL amErrorExists(void)
```

PARAMETERS

void

RETURN VALUE

`KTTRUE`	if a error exists
`KTFALSE`	if no error exists

FUNCTION

Allows checking of the error state for the AM layer in a single call returning a bool.

amErrorGetLast

Gets a pointer to the error structure.

FORMAT

```
#include <ac.h>
AC_ERROR_PTR amErrorGetLast(void)
```

PARAMETERS

`void`

RETURN VALUE

`AC_ERROR_STRUCT`	a pointer to the AM error structure

FUNCTION

Gets a pointer to the AM error structure. This contains an error number enumerated as an `AC_ERROR_TYPE` in `ac.h` and a more informative error message that tells the name of the function that failed as well as some descriptive text regarding the cause of the failure.

amFileClose

Closes a file.

FORMAT

```
#include <am.h>
KTBOOL amFileClose(ACFILE fd)
```

PARAMETERS

`ACFILE fd`	A GD system file descriptor

RETURN VALUE

`KTBOOL`, `KTTRUE`	on success
`KTFALSE`	on fail

FUNCTION

Closes a file. This operates through the am lib IO shell and is redirectable to the applications file system.

See: `amFileInstallAlternateIoManager()`

An example of this redirection is available in `MyFile.c` as well as a boilerplate copy of the IO proc for modification.

amFileGetSize

Gets the size of a file.

FORMAT

```
#include <am.h>
KTBOOL amFileGetSize(KTSTRING fileName, KTU32 * size)
```

PARAMETERS

`KTSTRING fileName`	The name of the file to load
`KTU32` * size	The size of the asset is returned via this pointer

RETURN VALUE

`KTBOOL`, `KTTRUE`	on success
`KTFALSE`	on fail

FUNCTION

Gets the size of a file. This operates through the am lib IO shell and is redirectable to the applications file system.

See: `amFileInstallAlternateIoManager()`

An example of this redirection is available in `MyFile.c` as well as a boilerplate copy of the IO proc for modification.

amFileInstallAlternateIoManager

Installs a custom Io proc.

FORMAT

```
#include <am.h>
void amFileInstallAlternateIoManager(AM_IO_PROC ioProc)
```

PARAMETERS

`AM_IO_PROC ioProc`	A pointer to a custom Io proc, see the example in `MyFile.c`

RETURN VALUE

void

FUNCTION

Installs a custom Io proc into the Io shell, this allows all file system calls to be intercepted by the applications file system.

The prototype for the IO proc is as follows:

```
KTBOOL MyCustomIoProc(            KTSTRING fileName,
ACFILE * fd,
KTU8 * buffer,
KTU32 * size,
AM_FILE_OPERATION_MODE mode
)
```

An example of this redirection is available in `MyFile.c` as well as a boilerplate copy of the IO proc for modification.

amFileLoad

Loads specified file into the buffer.

FORMAT

```
#include <am.h>
KTBOOL amFileLoad(KTSTRING fileName,KTU8 * buffer)
```

PARAMETERS

`KTSTRING fileName`	The name of the file to load
`KTU8 * buffer`	A buffer large enough to hold the file

RETURN VALUE

`KTBOOL`, `KTTRUE`	on success,
`KTFALSE`	on fail

FUNCTION

Loads a file given the file name and a buffer to load it into. This operates through the am lib IO shell and is redirectable to the applications file system.

See: `amFileInstallAlternateIoManager()`

An example of this redirection is available in `MyFile.c` as well as a boilerplate copy of the IO proc for modification.

amFileOpen

Opens a file for reading.

FORMAT

```
#include <am.h>
KTBOOL amFileOpen(KTSTRING fileName,ACFILE *fd)
```

PARAMETERS

`KTSTRING fileName`	The name of the file to load
`ACFILE fd`	A GD system file descriptor

RETURN VALUE

`KTBOOL`, `KTTRUE`	on success
`KTFALSE`	on fail

FUNCTION

Loads a file given the file name and a buffer to load it into. This operates through the am lib IO shell and is redirectable to the applications file system.

See: `amFileInstallAlternateIoManager()`

An example of this redirection is available in `MyFile.c` as well as a boilerplate copy of the IO proc for modification.

amFileRead

Reads from a file that is already open.

FORMAT

```
#include <am.h>
KTBOOL amFileRead(ACFILE fd,KTU8 * buffer,KTU32 size)
```

PARAMETERS

`ACFILE fd`	A GD system file descriptor
`KTU8 * buffer`	A pointer to a buffer into which to read
`KTU32 size`	The size of the data to be read

RETURN VALUE

`KTBOOL`, `KTTRUE`	on success
`KTFALSE`	on fail

FUNCTION

Reads from an open file. This operates through the am lib IO shell and is redirectable to the applications file system.

See: `amFileInstallAlternateIoManager()`

An example of this redirection is available in `MyFile.c` as well as a boilerplate copy of the IO proc for modification.

amFileRewind

Seeks to the start of a file.

FORMAT

```
#include <am.h>
KTBOOL amFileRewind(ACFILE fd)
```

PARAMETERS

`ACFILE fd`	A GD system file descriptor

RETURN VALUE

`KTBOOL,` KTTRUE	on success
KTFALSE	on fail

FUNCTION

Seeks to the head (byte 0) of the file. This operates through the am lib IO shell and is redirectable to the applications file system.

See: `amFileInstallAlternateIoManager()`

An example of this redirection is available in `MyFile.c` as well as a boilerplate copy of the IO proc for modification.

amHeapAlloc

Allocates aligned memory from the audio heap.

FORMAT

```
#include <am.h>
KTBOOL amHeapAlloc(
volatile KTU32 **buffer,
KTU32 size,KTU32 alignment,
AM_HEAP_MEMORY_TYPE memoryType,
AM_MEMORY_CALLBACK callback
)
```

PARAMETERS

`volatile KTU32 **buffer`	A pointer to the block of memory is returned via this handle
`KTU32 size,KTU32 alignment`	The desired alignment for the block (4 or 32)
`AM_HEAP_MEMORY_TYPE memoryType`	The type of memory desired `AM_FIXED_MEMORY` or `AM_PURGABLE_MEMORY`
`AM_MEMORY_CALLBACK callback`	A pointer to a callback function for the memory

RETURN VALUE

`KTTRUE`	if the operation was successful
`KTFALSE`	buffer is `NULL` size is 0 size exceeds available free memory alignment is not 4 or 32 `memoryType` is not `AM_FIXED_MEMORY` or `AM_PURGABLE_MEMORY`

FUNCTION

Allocates aligned memory from the audio heap zone. The memory can be allocated in alignments of either 4 or 32 bytes. Non DWORD aligned writes to the audio memory area are illegal and will corrupt the audio memory area severly. If the type is `AM_FIXED_MEMORY` the blocks will be allocated from the top of the heap progressing downwards, if the type is `AM_PURGABLE_MEMORY` the blocks are allocated from the bottom of the heap progressing upwards.

There is a variable amount of block overhead, this is applied as a fixed amount of ((alignment-1) * 2) + 4 when the parameters are tested so it is not possible to call for the amount of free memory remaining and allocate all of it. Depending on the alignment value the maximum allocation would be: alignment=4, maxMem - 10; or alignment=32, maxMem-66; The callback function will be invoked when the block is either purged or freed. The argument of the function is the address of the block that owned the callback.

Prototype for callback: `void MyCallback(KTU32 blockAddress)`

Note: All GD file system calls currently require that the buffer be aligned on a 32 byte boundry. This may only be called post a successful call to `amHeapInit()`

amHeapCheck

Checks the MCB fingerprints for overwrites.

FORMAT

```
#include <am.h>
KTBOOL amHeapCheck(void)
```

PARAMETERS

void

RETURN VALUE

`KTTRUE`	If the heap fingerprints are intact
`KTFALSE`	If the fingerprints are corrupted or the heap is not open

FUNCTION

Checks the MCB fingerprints in the heap to detect overwrites in that memory zone. Use this liberally to detect corruption or its possibility it will disappear in non-DEBUG versions.

Note: This is a `MACRO` that is expanded to the heap check function if `DEBUG` is defined.

If `DEBUG` is not defined it will become `((void)0); a null statement`.

amHeapFree

Frees purgable memory allocated using amHeapAlloc()

FORMAT

```
#include <am.h>
KTBOOL amHeapFree(volatile KTU32 *buffer)
```

PARAMETERS

`volatile KTU32 *buffer`	A pointer to the buffer to be freed

RETURN VALUE

`KTTRUE`	On success
`KTFALSE`	If buffer is `NULL`, buffer does not point to an allocated block buffer is not the higest address allocated in purgable memory

FUNCTION

This will free purgable memory from the top down by block address. If there is a block allocated with a higher address the call will fail, this prevents fragmentation. On freeing a block, if the block has a callback, it will be executed.

amHeapGetFree

Gets the amount of free memory.

FORMAT

```
#include <am.h>
KTBOOL    amHeapGetFree(KTU32 *freeMemory)
```

PARAMETERS

`KTU32 *freeMemory`	The amount of free memory is returned via this pointer

RETURN VALUE

`KTTRUE`	On success
`KTFALSE`	If the heap has not been initialized, `freeMemory` is `NULL`

FUNCTION

Gets the amount of free memory remaining in the heap.

amHeapGetInfo

Gets info necessary to start an audio heap.

FORMAT

```
#include <am.h>
KTBOOL amHeapGetInfo(volatile KTU32 **freeSoundMemory,KTU32 *size)
```

PARAMETERS

`volatile KTU32 **freeSoundMemory`	The pointer to the first free sound memory is returned via this handle
`KTU32 *size`	The size of the free portion of sound memory

RETURN VALUE

`KT`	as not been successfully installed

FUNCTION

Gets the necessary information for the `amHeapInit()` call from the sound driver.

Note: The driver must have been successfully installed prior to this call.

amHeapGetMaxPurgable

Gets amount of memory available from a full purge.

FORMAT

```
#include <am.h>
KTBOOL amHeapGetMaxPurgable(KTU32 *maxPurgable)
```

PARAMETERS

`KTU32 *maxPurgable`	The free memory size is returned via this pointer

RETURN VALUE

`KTTRUE`	on success
`KTFALSE`	heap is not initialized `maxPurgable` is `NULL`

FUNCTION

Gets the amount of memory available from the free memory pool + all AM_PURGABLE_MEMORY type blocks. This amount of memory is only available if a call is made to the function `amHeapClear(AM_PURGABLE_MEMORY)` or a call to `amHeapPurge(sizeNeeded)`.

amHeapInit

Initializes the audio heap.

FORMAT

```
#include <am.h>
KTBOOL amHeapInit(volatile KTU32 *memoryPool,KTU32 size)
```

PARAMETERS

`volatile KTU32 *memoryPool`	The start of the audio heap zone
`KTU32 size`	The size of the heap

RETURN VALUE

`KTTRUE`	If the operation was successful
`KTFALSE`	If `memoryPool` is `NULL` size is 0 heap is already open

FUNCTION

Initializes the heaps data structures

Note: A warning will be issued if size is not a multiple of 4, in this case size will be rounded down to the next multiple of 4.

amHeapPurge

Purges memory marked as purgable.

FORMAT

```
#include <am.h>
KTBOOL amHeapPurge(KTU32 sizeNeeded)
```

PARAMETERS

`KTU32 sizeNeeded`	The size of the block of memory needed

RETURN VALUE

`KTTRUE`	If the memory is now available
`KTFALSE`	If the heap has not been initialized, `sizeNeeded` is 0, `sizeNeeded` exceeds free + purgable,

FUNCTION

Will purge (if necessary) blocks of purgable memory in a top down fashion until sufficient memory is available to fill the requested size. If there is sufficient free memory to fill the request the function returns `KTTRUE` and does nothing. When a block is purged its callback (if installed) is invoked. This returns the address of the block to the application.

This function will not alter blocks of memory allocated as `AM_FIXED_MEMORY`.

amHeapShutdown

Shuts down the AM heap management system.

FORMAT

```
#include <am.h>
void amHeapShutdown(void)
```

PARAMETERS

void

RETURN VALUE

void

FUNCTION

Shuts down the AM heap management system.

amInit

Starts up the AM audio subsystem.

FORMAT

```
#include <am.h>
KTBOOL amInit(void)
```

PARAMETERS

void

RETURN VALUE

`KTTRUE`	on success
`KTFALSE`	on fail Driver file not found Driver startup fail

FUNCTION

Starts up the AM audio subsystem. This will load the driver into the middle of the audio heap then install that image using `acInstallDriver`. It then starts up the am interrupt and heap management systems. This also calls `acCdInit()` to initialize the redbook playback mechanism.

amInitSelectDriver

Selects driver to be installed by amInit().

FORMAT

```
#include <am.h>
KTBOOL amInitSelectDriver(AM_DRIVER_TYPE driverType)
```

PARAMETERS

`AM_DRIVER_TYPE driverType`Either `AM_DA_DRIVER` or `AM_MIDI_DRIVER`

RETURN VALUE

`KTTRUE`	on success
`KTFALSE`	`on fail` Driver is already installed or bad arg for `driverType`

FUNCTION

Allows selection of the type of driver to be loaded by the `amInit()` call. The default driver is the audio64 driver so if this call is not made the system will be set up as `audio64`.

Note: This must be called **PRIOR** to the call to `amInit()`.

amMemInit ⃠

Initializes the Sh4 memory shell system.

FORMAT

```
#include <am.h>
void amMemInit(void)
```

PARAMETERS

void

RETURN VALUE

void

FUNCTION

Initializes the memory manager shell proc pointers with the default routines if they have not been previously initialized. Called by `amInit()`.

Note: Neither the AM nor AC layers allocate or free SH4 memory.

amMemInstallAlternateMemoryManager

Allows redirection of sh4 memory requests.

FORMAT

```
#include <am.h>
void amMemInstallAlternateMemoryManager(AM_SH4_ALLOC_PROC allocProc,AM_SH4_FREE_PROC
freeProc)
```

PARAMETERS

`AM_SH4_ALLOC_PROC allocProc`	a pointer to an correctly prototyped malloc proc
`AM_SH4_FREE_PROC freeProc`	a pointer to an correctly prototyped free proc

RETURN VALUE

void

FUNCTION

Initializes the malloc and free proc pointers in the audio engines memory allocation shell.

Note: This **MUST** be called prior to the call to `amInit().`
Neither the AM nor AC layers allocate or free SH4 memory.

amMemSh4Alloc ⃠

Sh4 memory allocation shell.

FORMAT

```
#include <am.h>
KTBOOL amMemSh4Alloc(         volatile KTU32 ** base,
volatile KTU32 ** aligned,
KTU32 size,
KTU32 alignment)
```

PARAMETERS

`volatile KTU32 ** base`	an unaligned pointer to the block allocated
`volatile KTU32 ** aligned`	a pointer to the first aligned address after the base address
`KTU32 size,KTU32 alignment`	the alignment desired

RETURN VALUE

`KTTRUE`	if a block was successfully allocated
`KTFALSE`	if insufficient memory available

FUNCTION

This is a shell that verifies a proc pointer then calls it to invoke whatever malloc proc is currently installed.

Note: Neither the AM nor AC layers allocate or free SH4 memory.

amMemSh4Free

Sh4 memory free shell.

FORMAT

```
#include <am.h>
void amMemSh4Free(volatile KTU32 * block)
```

PARAMETERS

`volatile KTU32 * block`	a pointer to the unaligned base address of the block to be freed

RETURN VALUE

void

FUNCTION

This is a shell that verifys a proc pointer then calls it to invoke whatever free proc is currently installed.

Note: Neither the AM nor AC layers allocate or free SH4 memory.

amMidiAllocateSequencePort

Allocates a MIDI port for the sequence.

FORMAT

```
#include <am.h>
KTBOOL amMidiAllocateSequencePort(AM_SEQUENCE_PTR theSequence)
```

PARAMETERS

`AM_SEQUENCE_PTR theSequence`	A properly initialized sequence object

RETURN VALUE

`KTTRUE`	on success

`KTFALSE`	`on fail`
	unable to send command to driver
	`theSequence` is `NULL`
	port allocation failed (all voices busy)

FUNCTION

Allocates a `MIDI` port for the sequence. This calls `amVoiceAllocate()` and allocates a `AM_MIDI_VOICE` type channel. This voice channel number is the `midiPort` number + 16.

Note: This sets the user callback in the voice management system so the callback proc must be installed prior to making this call. This group of functions will only work with the `MidiDa` driver, they will not work with the Audio64 driver.

amMidiFetchSequence

Fetches a sequence asset from a `katBank`.

FORMAT

```
#include <am.h>
KTBOOL amMidiFetchSequence(AM_SEQUENCE_PTR theSequence,KTU8 *theBank,KTU32
sequenceNumber)
```

PARAMETERS

`AM_SEQUENCE_PTR theSequence`	A properly initialized sequence object
`AM_BANK_PTR theBank`	A pointer to a `katBank` in sound memory
`KTU32 sequenceNumber`	The bank asset number to fetch, see the banks `.h` file for bank and asset `inf.`

RETURN VALUE

`KTTRUE`	on success

`KTFALSE`	on fail
	unable to send command to driver
	`theSequence` is `NULL`
	`theBank` is `NULL`
	the asset fetch failed (asset not present in bank)
	the requested asset was not a MIDI asset
	the bank header is corrupt

FUNCTION

Fetches a standard MIDI type 0 sequence asset from a kat type bank using the amBank...() API. This type of bank is manufactured with the `mkscript` and `mkbank` utilities.

Note: This group of functions will only work with the `MidiDa` driver, they will not work with the Audio64 driver.

amMidiFetchToneBank

Installs an MTB asset from a bank file aggregate.

FORMAT

```
#include <am.h>
KTBOOL amMidiFetchToneBank(AM_BANK_PTR theBank,KTU32 assetNumber,KTU8 toneBankSlot)
```

PARAMETERS

`AM_BANK_PTR theBank`	A pointer to a `.kat` bank type asset aggregation
`KTU32 assetNumber`	The number of the tone bank asset in the bank
`KTU8 toneBankSlot`	The slot number of the bank 0-15

RETURN VALUE

`KTTRUE`	on success
`KTFALSE`	on fail asset is wrong type asset number not in bank unable to post command to driver

FUNCTION

Installs an MTB asset that is contained in a `.kat` bank aggregate file.

Note: This group of functions will only work with the `MidiDa` driver, they will not work with the Audio64 driver.

amMidiInstallCallback

Sets the callback proc for a sequence.

FORMAT

```
#include <am.h>
KTBOOL amMidiInstallCallback(AM_SEQUENCE_PTR theSequence,AC_MIDI_CALLBACK theCallback)
```

PARAMETERS

`AM_SEQUENCE_PTR theSequence`	A properly initialized sequence object
`AC_MIDI_CALLBACK theCallback`	The callback proc

RETURN VALUE

`KTTRUE`	on success
`KTFALSE`	on fail unable to send command to driver `theSequence` is `NULL`

FUNCTION

Sets the callback proc for a sequence.

The voice channel number is returned to the callback, however, please note that this is not the same as the `midiPort` number. The `midiPort` number is 16 less then the voice channel number.

The format of the callback is:

```
void MyCallbackProc(KTU32 voiceChannelNumber)
```

Note: This must be called prior to the `amMidiAllocateSequencePort()` and `amMidiPlay()` calls. This group of functions will only work with the `MidiDa` driver, they will not work with the Audio64 driver.

amMidiLoadToneBank

Loads a Sega tone bank asset

FORMAT

```
#include <am.h>
KTBOOL amMidiLoadToneBank(KTSTRING fileName,KTU8 gmMode,volatile KTU32 * buffer,KTU32
bankSize,KTU8 toneBankSlot)
```

PARAMETERS

`KTSTRING fileName`	The name of the bank file to be loaded from the GD system
`KTU8 gmMode`	`AC_GM_ON` or `AC_GM_OFF`, enables or disables general midi mode
`volatile KTU32 * buffer`	A 32 byte aligned buffer in sound memory
`KTU32 bankSize`	The size of the bank to be loaded
`KTU8 toneBankSlot`	The slot number of the bank 0-15

RETURN VALUE

`KTTRUE`	on success
`KTFALSE`	on fail buffer not 32 byte aligned file not found or unable to send driver command

FUNCTION

This loads a midi tonebank made by the SOJ mac tool from the GD-ROM using the redirectable file system.

Note: If `gmMode` is out of range it will be set to `AC_GM_ON.` This group of functions will only work with the `MidiDa` driver, they will not work with the Audio64 driver.

amMidiNoteOff

Stops a MIDI triggered sound effect.

FORMAT

```
#include <am.h>
KTBOOL amMidiNoteOff(KTU32 midiPort,KTU32 midiChannel,KTU8 midiNoteNumber)
```

PARAMETERS

`KTU32 midiPort`	The MIDI port number 0-15
`KTU32 midiChannel`	The MIDI channel number 1-16
`KTU8 midiNoteNumber`	The MIDI note number of the sound to be played

RETURN VALUE

`KTTRUE`	on success
`KTFALSE`	on fail unable to send command to driver `midiPort` out of range (0-15) `midiChannel` out of range (1-16) midiNoteNumber out of range (0-127)

FUNCTION

This will stop a MIDI triggered sound effect if it is currently playing.

Note: This group of functions will only work with the `MidiDa` driver, they will not work with the Audio64 driver.

amMidiNoteOn

Plays a MIDI triggered sound effect.

FORMAT

```
KTBOOL amMidiNoteOn(KTU32 midiPort,KTU32 midiChannel,KTU8 midiNoteNumber,KTU32
midiNoteOnVelocity)
#include <am.h>
```

PARAMETERS

`KTU32 midiPort`	The MIDI port number 0-15
`KTU32 midiChannel`	The MIDI channel number 1-16
`KTU8 midiNoteNumber`	The MIDI note number of the sound to be played. 0-127
`KTU32 midiNoteOnVelocity`	The MIDI note on velocity 0-127

RETURN VALUE

`KTTRUE`	on success
`KTFALSE`	on fail unable to send command to driver `midiPort` out of range (0-15) `midiChannel` out of range (1-16) `midiNoteNumber` out of range (0-127)

FUNCTION

Plays a MIDI triggered sound effect from a Sega Tonebank type asset loaded with the `amMidiLoadBank()` call.

Note: If `midiNoteOnVelocity` is out of range it will be set to `AC_MAX_MIDI_VELOCITY` (127). This group of functions will only work with the `MidiDa` driver, they will not work with the Audio64 driver.

amMidiPause

Pauses a currently playing MIDI sequence.

FORMAT

```
#include <am.h>
KTBOOL amMidiPause(AM_SEQUENCE_PTR theSequence)
```

PARAMETERS

`AM_SEQUENCE_PTR theSequence`	A properly initialized sequence object

RETURN VALUE

`KTTRUE`	on success
`KTFALSE`	on fail unable to send command to driver `theSequence` is `NULL`

FUNCTION

Pauses a currently playing MIDI sequence. This will silence all currently sounding notes.

Note: This group of functions will only work with the `MidiDa` driver, they will not work with the Audio64 driver.

amMidiPlay

Plays a MIDI sequence.

FORMAT

```
#include <am.h>
KTBOOL amMidiPlay(AM_SEQUENCE_PTR theSequence)
```

PARAMETERS

`AM_SEQUENCE_PTR theSequence`	A properly initialized sequence object

RETURN VALUE

`KTTRUE`	on success
`KTFALSE`	on fail unable to send command to driver `theSequence` is `NULL`

FUNCTION

Plays a standard `MIDI` type 0 asset obtained from a kat bank using `amMidiPlayRaw()`. This type of bank is manufactured with the `mkscript` and `mkbank` utilities.

Note: This group of functions will only work with the `MidiDa` driver, they will not work with the Audio64 driver.

amMidiPlayRaw

Plays a MIDI sequence given the basic parameters.

FORMAT

```
#include <am.h>
KTBOOL amMidiPlayRaw(KTU32 midiPort,KTU8 gmMode,KTU32 ticksPQN,KTU32 sequenceSize,
KTU32 *sequenceAddress,KTU32 midiVolume,AC_MIDI_CALLBACK callback)
```

PARAMETERS

`KTU32 midiPort`	The `MIDI` port number (0-15)
`KTU8 gmMode`	`AC_GM_ON` or `AC_GM_OFF`, enables or disables general midi mode
`KTU32 ticksPQN`	The number of ticks per quarter note. (often 480)
`KTU32 sequenceSize`	The size in bytes of the `MIDI` sequence data
`KTU32 *sequenceAddress`	The address of a `MIDI` type 0 asset in sound memory
`KTU32 midiVolume`	The `MIDI` volume at which to start the sequence (0-127)
`AC_MIDI_CALLBACK callback`	The address of a callback proc or `KTNULL` for no callback

RETURN VALUE

`KTTRUE`	on success
`KTFALSE`	on fail unable to send command to driver `sequenceAddress` is `NULL` `sequenceSize` is 0

FUNCTION

Plays a MIDI type 0 asset in sound memory at the given volume with an optional callback that will be raised at the end of the sequences play. The voice channel number is returned to the callback however please note that this is not the same as the `midiPort` number. The `midiPort` number is 16 less then the voice channel number.

Note: If midiVolume is out of range it will be set to 127

If `gmMode` is out of range it will be set to `AC_GM_ON`

The format of the callback is:

```
void MyCallbackProc(KTU32 voiceChannelNumber)
```

Note: This group of functions will only work with the `MidiDa` driver, they will not work with the Audio64 driver.

amMidiResume

Resumes playback of a paused MIDI sequence.

FORMAT

```
#include <am.h>
KTBOOL amMidiResume(AM_SEQUENCE_PTR theSequence)
```

PARAMETERS

`AM_SEQUENCE_PTR theSequence`	A properly initialized sequence object

RETURN VALUE

`KTTRUE`	on success
`KTFALSE`	on fail unable to send command to driver `theSequence` is `NULL`

FUNCTION

Resumes playback of a previously paused `MIDI` sequence.

Note: This group of functions will only work with the `MidiDa` driver, they will not work with the Audio64 driver.

amMidiSetChannelPan

Sets the pan of a MIDI sound.

FORMAT

```
#include <am.h>
KTBOOL amMidiSetChannelPan(KTU32 midiPort,KTU32 midiChannel,KTU32 midiPan)
```

PARAMETERS

`KTU32 midiPort`	The MIDI port number 0-15
`KTU32 midiChannel`	The MIDI channel number 1-16
`KTU32 midiPan`	The MIDI pan to set 0-127

RETURN VALUE

`KTTRUE`	on success
`KTFALSE`	on fail unable to send command to driver `midiPort` out of range (0-15) `midiChannel` out of range (1-16)

FUNCTION

Sets pan (position) of a currently iterating MIDI triggered sound. This sends a MIDI Control Change 10 value ? to the driver.

Note: If midi pan is out of range it will be set to `AC_MAX_MIDI_PAN` (127)

Note: This group of functions will only work with the `MidiDa` driver, they will not work with the Audio64 driver.

amMidiSetChannelProgram

Sets the current bank slot.

FORMAT

```
#include <am.h>
KTBOOL amMidiSetChannelProgram(KTU32 midiPort,KTU32 midiChannel,KTU32
midiProgramNumber)
```

PARAMETERS

`KTU32 midiPort`	The MIDI port number 0-15
`KTU32 midiChannel`	The MIDI channel number 1-16
`KTU8 midiProgramNumber`	The slot number of the program to be played for the midi channel

RETURN VALUE

`KTTRUE`	on success
`KTFALSE`	on fail unable to send command to driver

FUNCTION

Prior to playing a sound effect from a midi bank the bank slot must be made the current bank slot this allows the setting of a current bank for a given portchannel configuration.

Note: This group of functions will only work with the `MidiDa` driver, they will not work with the Audio64 driver.

amMidiSetChannelVolume

Sets volume of a midi sound.

FORMAT

```
#include <am.h>
KTBOOL amMidiSetChannelVolume(KTU32 midiPort,KTU32 midiChannel,KTU32 midiVolume)
```

PARAMETERS

KTU32 `midiPort`	The MIDI port number 0-15
KTU32 `midiChannel`	The MIDI channel number 1-16
KTU32 midiVolume	The MIDI volume to set 0-127

RETURN VALUE

`KTTRUE`	on success
`KTFALSE`	on fail unable to send command to driver `midiPort` out of range (0-15) `midiChannel` out of range (1-16)

FUNCTION

Sets `CHANNEL` volume of a currently playing MIDI triggered sound. This sends a MIDI Control Change 7 value ? to the driver.

Note: If midi volume is out of range it will be set to `AC_MAX_MIDI_VOLUME` (127). This group of functions will only work with the `MidiDa` driver, they will not work with the Audio64 driver.

amMidiSetLoopFlag

Sets the loop flag on a MIDI sequence.

FORMAT

```
#include <am.h>
KTBOOL amMidiSetLoopFlag(AM_SEQUENCE_PTR theSequence,KTBOOL onOrOff)
```

PARAMETERS

`AM_SEQUENCE_PTR theSequence`	a pointer to an `AM_SEQUENCE` object
`KTBOOL`	`onOrOff`
`KTTRUE`	to loop
`KTFALSE`	to not.

RETURN VALUE

`KTTRUE`	on success
`KTFALSE`	on fail bad arguments `theSequence` is `NULL`

FUNCTION

Sets the loop flag in an `AM_SEQUENCE` object.

Note: if onOrOff is out of range it will be set to `KTTRUE`. This group of functions will only work with the `MidiDa` driver, they will not work with the Audio64 driver.

amMidiSetTempo

Sets the tempo of a MIDI sequence.

FORMAT

```
#include <am.h>
KTBOOL amMidiSetTempo(AM_SEQUENCE_PTR theSequence,KTS32 percentOfChange)
```

PARAMETERS

`AM_SEQUENCE_PTR theSequence`	A pointer to an `AM_SEQUENCE` object
`KTS32 percentOfChange`	The percent of change over or under the root tempo

RETURN VALUE

`KTTRUE`	on success
`KTFALSE`	on fail `theSequence` is `NULL`

FUNCTION

Changes the tempo of a currently playing midi sequence to the new tempo. This is expressed as a percentage of change from the root (original) tempo. i.e. the tempo of the file is 120, a +10% change is applied, the sequence is now playing at tempo 132. If a change of 0 is specified the sequence will play at its root tempo.

Note: This group of functions will only work with the `MidiDa` driver, they will not work with the Audio64 driver.

amMidiSetVolume

Sets the master volume of a MIDI sequence.

FORMAT

```
#include <am.h>
KTBOOL amMidiSetVolume(AM_SEQUENCE_PTR theSequence,KTU32 newAicaVolume)
```

PARAMETERS

`AM_SEQUENCE_PTR theSequence`	A properly initialized sequence object
`KTU32 newMidiVolume`	the MIDI volume for the port master (0-127)

RETURN VALUE

`KTTRUE`	on success
`KTFALSE`	on fail unable to send command to driver `theSequence` is `NULL`

FUNCTION

This call sets the `MASTER` volume of a `MIDI` sequence. The `MASTER` volume is the overall volume of the sequence as opposed to the `CHANNEL` volume which would affect only one of the 16 possible MIDI channels in the sequence.

If the `newMidiVolume` value is out of range it will be set to 127

Note: This group of functions will only work with the `MidiDa` driver, they will not work with the Audio64 driver.

amMidiTransferToneBank

Transfers a Sega tone bank to sound memory and sets it as the current bank.

FORMAT

```
#include <am.h>
KTBOOL amMidiTransferToneBank(volatile KTU32 *destination,KTU32 *source,KTU8 gmMode,KTU32
bankSize,KTU8 toneBankSlot)
```

PARAMETERS

`volatile KTU32 * destination`	A dword aligned buffer in sound memory
`KTU32 *source`	A buffer that contains the bank to be transferred
`KTU8 gmMode`	`AC_GM_ON` or `AC_GM_OFF`, enables or disables general midi mode
`KTU32 bankSize`	The size of the bank
`KTU8 toneBankSlot`	The slot number of the bank 0-15

RETURN VALUE

`KTTRUE`	on success
`KTFALSE`	on fail destination is not 32 byte aligned, unable to send driver command

FUNCTION

This transfers a midi tonebank made by the SOJ mac tool from any memory to sound memory and sets it as the current bank.

Note: If `gmMode` is out of range it will be set to `AC_GM_ON.` This group of functions will only work with the `MidiDa` driver, they will not work with the Audio64 driver.

amShutdown

Shuts down the AM audio subsystem.

FORMAT

```
#include <am.h>
void amShutdown(void)
```

PARAMETERS

void

RETURN VALUE

void

FUNCTION

Shuts down the AM audio subsystem by stopping all sounds and closing their voice channels, releasing all OS service vectors and closing the `amHeap` subsystem.

amSoundAllocateVoiceChannel

Allocates a hardware voice channel.

FORMAT

```
#include <am.h>
KTBOOL amSoundAllocateVoiceChannel(AM_SOUND_PTR theSound)
```

PARAMETERS

`AM_SOUND_PTR theSound`	A pointer to a properly initialized sound object

RETURN VALUE

`KTBOOL, KTTRUE`	on success
`KTFALSE`	can’t allocate voice (all channels busy) `theSound` is `NULL`

FUNCTION

This allocates a hardware voice channel (an ac lib “port”) for playback by the `amSound` subsystem. The channel is freed via the system callback mechanism when the sound has been stopped prior to the end or has finished playing.

Note: The sound structure must have been initialized with the `amBankFetchSound` function for it to contain valid data.

amSoundFetchSample

Fetches a sound and its parameters from a Katana format bank.

FORMAT

```
#include <am.h>
KTBOOL amSoundFetchSample(AM_BANK_PTR theBank,KTU32 soundNumber,AM_SOUND_PTR theSound)
```

PARAMETERS

`AM_BANK_PTR theBank`	A pointer to a `katbank` containing the sound to be fetched
`KTU32 soundNumber`	The sound number to be fetched
`AM_SOUND_PTR sound`	A pointer to an `AM_SOUND` structure, this will contain all needed information on the sound on successful return from this function. On fail this structure will be filled with 0x00

RETURN VALUE

`KTTRUE`	on success

`KTFALSE`	`theSound` is `NULL`
	`theBank` is `NULL`
	`soundNumber` is out of range
	the bank asset is not of the right type

FUNCTION

Fetches a digital sound from a given `Katbank`.

Calls: `amBankFetchAsset()`

amSoundGetCallback

Gets the address of the user callback.

FORMAT

```
#include <am.h>
KTBOOL amSoundGetCallback(AM_SOUND_PTR theSound,AM_USER_CALLBACK *theCallback)
```

PARAMETERS

`AM_SOUND_PTR theSound`	A pointer to a properly initialized sound object
`AM_USER_CALLBACK *theCallback`	A pointer to the callback is returned via this handle

RETURN VALUE

`KTTRUE`	on success

`KTFALSE`	`theSound` is `NULL`
	`theCallback` is `NULL`

FUNCTION

This gets the address of the user callback proc assigned to a sound, if no callback has been assigned `KTNULL` will be returned.

Note: The sound structure must have been initialized with the `amBankFetchSound` function for it to contain valid data.

amSoundGetPan

Gets the current pan position.

FORMAT

```
#include <am.h>
KTBOOL amSoundGetPan(AM_SOUND_PTR theSound,KTU32 *aicaPan)
```

PARAMETERS

`AM_SOUND_PTR theSound`	A pointer to a properly initialized sound object
`KTU32 *pan`	The pan (0-127) is returned via this pointer

RETURN VALUE

`KTBOOL, KTTRUE`	on success
`KTFALSE`	`theSound` is `NULL` pan is `NULL`

FUNCTION

This returns the current pan of the sound in normal pan units (0-127).

Note: The sound structure must have been initialized with the `amBankFetchSound` function for it to contain valid data.

amSoundGetSampleRate

Gets the real world sample rate.

FORMAT

```
#include <am.h>
KTBOOL amSoundGetSampleRate(AM_SOUND_PTR theSound,KTU32 *realWorldSampleRate)
```

PARAMETERS

`AM_SOUND_PTR theSound`	A pointer to a properly initialized sound object
`KTU32 *realWorldSampleRate`	The real world sample rate is returned via this pointer

RETURN VALUE

`KTBOOL, KTTRUE`	on success

`KTFALSE`	can't allocate voice (all channels busy)
	`theSound` is `NULL`
	`realWorldSampleRate` is `NULL`

FUNCTION

This will return the real world sample rate of the given sound. Real world rates are 44100, 22050 etc.

Note: The sound structure must have been initialized with the `amBankFetchSound` function for it to contain valid data.

amSoundGetVoiceChannel

Gets the current voice channel assignment.

FORMAT

```
#include <am.h>
KTBOOL amSoundGetVoiceChannel(AM_SOUND_PTR theSound,KTU32 *voiceChannel)
```

PARAMETERS

`AM_SOUND_PTR theSound`	A pointer to a properly initialized sound object
`KTU32 *voiceChannel`	The voice channel is returned via this pointer

RETURN VALUE

`KTTRUE`	on success

`KTFALSE`	`theSound` is `NULL`
	`voiceChannel` is `NULL`

FUNCTION

This gets the current voice channel assignment of a sound. If the sound has not yet been initialized with a voice channel assignment the value `AM_UNINITIALIZED_VOICE_CHANNEL` will be returned.

Note: The sound structure must have been initialized with the `amBankFetchSound` function for it to contain valid data.

amSoundGetVolume

Gets the current volume setting.

FORMAT

```
#include <am.h>
KTBOOL amSoundGetVolume(AM_SOUND_PTR theSound,KTU32 *volume)
```

PARAMETERS

`AM_SOUND_PTR theSound`	A pointer to a properly initialized sound object
`KTU32 *volume`	The volume (0-127) is returned via this pointer

RETURN VALUE

`KTBOOL, KTTRUE`	on success
`KTFALSE`	`theSound` is `NULL` volume is `NULL`

FUNCTION

This returns the current volume of the sound in normal volume units (0-127).

Note: The sound structure must have been initialized with the `amBankFetchSound` function for it to contain valid data.

amSoundIsLooping

Tells if the given sound has a loop.

FORMAT

```
#include <am.h>
KTBOOL amSoundIsLooping(AM_SOUND_PTR theSound,KTBOOL *loopFlag)
```

PARAMETERS

`AM_SOUND_PTR theSound`	A pointer to a properly initialized sound object
`KTBOOL *loopFlag`	The loop flag is returned via this pointer

RETURN VALUE

`KTTRUE`	on success

`KTFALSE`	`theSound` is `NULL`
	`loopFlag` is `NULL`

FUNCTION

Queries weather a given sound has a loop or not.

Note: The sound structure must have been initialized with the `amBankFetchSound` function for it to contain valid data.

amSoundIsPlaying

Tells if a sound is currently playing.

FORMAT

```
#include <am.h>
KTBOOL amSoundIsPlaying(AM_SOUND_PTR theSound)
```

PARAMETERS

`AM_SOUND_PTR theSound`	A pointer to a properly initialized sound object

RETURN VALUE

`KTTRUE`	if the sound is playing.
`KTFALSE`	if `theSound` is `NULL` the sound is not playing.

FUNCTION

Note: The sound structure must have been initialized with the `amBankFetchSound` function for it to contain valid data.

amSoundPlay

Plays a sound.

FORMAT

```
#include <am.h>
KTBOOL amSoundPlay(AM_SOUND_PTR theSound)
```

PARAMETERS

`AM_SOUND_PTR theSound`	A pointer to a properly initialized sound object

RETURN VALUE

`KTTRUE`	if the sound was played
`KTFALSE`	can't send a command to the driver `theSound` is `NULL` a voice channel had not been allocated

FUNCTION

This will start a properly initialized sound object playing.

Note: The sound structure must have been initialized with the `amBankFetchSound` function for it to contain valid data.

On Failure, due to failing `amSoundPlayRaw()` or internal error, this will release the voice channel that was allocated for the sound to prevent a failed call from leaking a voice channel.

The member the`Sound->voiceChannel` will be set to `AM_UNINITIALIZED_VOICE_CHANNEL` if it has been released.

amSoundPlayRaw

Plays a sound given all of the required parameters.

FORMAT

```
#include <am.h>
KTBOOL amSoundPlayRaw(         KTS32 voiceChannel,
KTU32 sizeInBytes,
KTU32 address,
KTU32 sampleRate,
AC_AUDIO_TYPE aicaAudioType,
KTU32 pitchOffsetInCents,
KTS32 aicaLoopFlag,
AM_USER_CALLBACK userCallbackProc,
KTU32 dryVolume,
KTU32 wetVolume,
KTU32 pan,
KTU32 mixerChannel,
KTBOOL effectsOnOrOff
)
```

PARAMETERS

`KTS32 voiceChannel`	The DA port number to use for the sound playback
`KTU32 sizeInBytes`	The size of the sound in bytes
`KTU32 address`	The address of the sound in sound memory
`KTU32 sampleRate`	The real world sample rate, i.e. 44100, 22050, 16000 etc
`AC_AUDIO_TYPE aicaAudioType`	The audio type, i.e. `AC_16BIT, AC_8BIT, AC_ADPCM_LOOP, see ac.h`
`KTU32 pitchOffsetInCents`	The amount to offset the pitch (positive offset only)
`KTS32 aicaLoopFlag`	The aica loop flag, either `AC_LOOP_ON` or `AC_LOOP_OFF`, see `ac.h`
`AM_USER_CALLBACK userCallbackProc`	A pointer to a user callback proc or `KTNULL`
`KTU32 dryVolume`	The normal volume (0-127)
`KTU32 wetVolume`	The effects volume (0-127)
`KTU32 pan`	The normal pan (0-127)
`KTU32 mixerChannel`	The effects bank mixer channel to use
`KTBOOL effectsOnOrOff`	True if mixer channel supplied is valid turns effects on and off

RETURN VALUE

void

FUNCTION

Plays a raw PCM intel byte order sound from sound memory. If a user callback proc is supplied the callback will be invoked when the sound is finished with its play.

The proc will need to have the following prototype:

```
void MyCallbackProc(KTU32 voiceChannel);
```

The voice channel (DA port #) that raised the interrupt will be passed up in the arg `voiceChannel`.

amSoundSetCallback

Sets the user callback.

FORMAT

```
#include <am.h>
KTBOOL amSoundSetCallback(AM_SOUND_PTR theSound,AM_USER_CALLBACK callback)
```

PARAMETERS

`AM_SOUND_PTR theSound`	A pointer to a properly initialized sound object
`KTU32 callback`	The address of a user callback function

RETURN VALUE

`KTTRUE`	on success
`KTFALSE`	`theSound` is `NULL` `theSound` is playing

FUNCTION

Sets the user callback for a sound. This function will be called when a sound has finished playing. The callback function will need to be protyped as void foo(`KTU32 voiceChannel`).

Note: The sound structure must have been initialized with the `amBankFetchSound` function for it to contain valid data.

amSoundSetCurrentPlaybackRate

Sets the playback rate.

FORMAT

```
#include <am.h>
KTBOOL amSoundSetCurrentPlaybackRate(AM_SOUND_PTR theSound,KTU32 sampleRate)
```

PARAMETERS

`AM_SOUND_PTR theSound`	A pointer to a properly initialized sound object

RETURN VALUE

`KTTRUE`	on success
`KTFALSE`	`theSound` is `NULL` sampleRate is > 1128900 can’t send a command to the driver

FUNCTION

If called prior to playing a sound this will set the sounds initial playback rate. If called while the sound is playing the current playback rate will be set.

Note: The sound structure must have been initialized with the `amBankFetchSound` function for it to contain valid data.

amSoundSetEffectsBuss

Sets the effects buss send and source mix for a sound object.

FORMAT

```
#include <am.h>
KTBOOL amSoundSetEffectsBuss(AM_SOUND_PTR theSound,KTU32 dspMixerChannel,KTU32 sourceMix)
```

PARAMETERS

`AM_SOUND_PTR theSound`	A pointer to a properly initialized sound object
`KTU32 dspMixerChannel`	The DSP mixer channel to route the dry send into
`KTU32 sourceMix`	The percentage of the dry volume to route to wet volume (1-100)

RETURN VALUE

`KTTRUE`	On success

`KTFALSE`	if theSound is `NULL`
	`if dspMixerChannel > AM_MAX_DSP_MIXER_CHANNEL`
	`if sourceMix > AM_MAX_DSP_SOURCE_MIX`

FUNCTION

This will set the effects send and source mixof the given sound. The argument `sourceMix` is how much of a DSP program is added to the dry send. If a source mix of 100% is selected and the sound has a volume of 90 then the wet level will be 90 and the dry level will be 90, if a sourceMix of 50% is selected the the wet level will be 45 and the dry level 90.

amSoundSetPan

Sets a sounds pan.

FORMAT

```
#include <am.h>
KTBOOL amSoundSetPan(AM_SOUND_PTR theSound,KTU32 newPan)
```

PARAMETERS

`AM_SOUND_PTR theSound`	A pointer to a properly initialized sound object
`KTU32 newPan`	The pan to set (0-127), right to left

RETURN VALUE

`KTTRUE`	on success
`KTFALSE`	can't send a command to the driver `theSound` is `NULL`

FUNCTION

If called prior to playing a sound this will set the sounds initial playback pan position.

If called while a sound is playing it will set the current playback pan position.

Note: The sound structure must have been initialized with the `amBankFetchSound` function for it to contain valid data.

If pan > `AM_MAX_PAN` (127) pan will be set to `AM_MAX_PAN`.

Because the AICA pan scale is 0-31 the normal pan numbers of 0-127 are quantitized to 31 steps.

amSoundSetQSoundChannels

Used to identify which channels in an output bank are Q-Sound channels.

FORMAT

```
#include <am.h>
KTBOOL amSoundSetQSoundChannels(KTU32 firstQChannel,KTU32 numberOfQChannels)
```

PARAMETERS

`KTU32 firstQChannel`	The first Q-Sound channel in the output bank (`.fob`) asset
`KTU32 numberOfQChannels`	The number of Q-Sound channels in the output bank (`.fob`) asset

RETURN VALUE

`KTTRUE`	On success
`KTFALSE`	`firstQChannel` is out of range `numberOfQChannels > AM_MAX_Q_CHANNELS`

FUNCTION

Used to identify which channels in an output bank are Q-Sound channels. If this is called with `numberOfQChannels==0` then the Q channel identification system is cleared.

amSoundSetVolume

Sets a sounds volume.

FORMAT

```
#include <am.h>
KTBOOL amSoundSetVolume(AM_SOUND_PTR theSound,KTU32 newVolume)
```

PARAMETERS

`AM_SOUND_PTR theSound`	A pointer to a properly initialized sound object
`KTU32 newVolume`	The volume to set (0-127)

RETURN VALUE

`KTTRUE`	on success
`KTFALSE`	can't send a command to the driver `theSound` is `NULL`

FUNCTION

If called prior to playing a sound this will set the sounds initial playback volume. If called while a sound is playing it will set the current playback volume.

Note: The sound structure must have been initialized with the `amBankFetchSound` function for it to contain valid data.

If new`Volume > AM_MAX_VOLUME` (127) newVolume will be set to `AM_MAX_VOLUME`

Further the aica volume range is 0-15 so the 0-127 range is quantitized into 15 steps.

amSoundStop

Stops a currently playing sound.

FORMAT

```
#include <am.h>
KTBOOL amSoundStop(AM_SOUND_PTR theSound)
```

PARAMETERS

`AM_SOUND_PTR theSound`	A pointer to a properly initialized sound object

RETURN VALUE

`KTTRUE`	if the sound was stopped
`KTFALSE`	if the sound was not playing can't send a command to the driver `theSound` is `NULL`

FUNCTION

This stops a currently playing sound and releases its voice channel.

Note: The sound structure must have been initialized with the `amBankFetchSound` function for it to contain valid data.

amStreamAllocateVoiceChannels

Allocates voice channels.

FORMAT

```
#include <am.h>
KTBOOL amStreamAllocateVoiceChannels(AM_STREAM_PTR theStream)
```

PARAMETERS

`AM_STREAM_PTR theStream`	The stream object to get voices

RETURN VALUE

`KTBOOL, KTTRUE`	If the voice(s) were successfully allocated
`KTFALSE`	If the allocation failed. If the stream was not open.

FUNCTION

This calls `amVoiceAllocate()` to allocate voices for the given stream, playback requires one voice per channel of program. A mono stream is one channel, a single track of stereo is two channels.

Note: This must be called post to call to `amStreamOpen()`.

amStreamClose

Closes a stream object.

FORMAT

```
#include <am.h>
KTBOOL amStreamClose(AM_STREAM_PTR theStream)
```

PARAMETERS

`AM_STREAM_PTR theStream`	the stream object to be closed

RETURN VALUE

`KTBOOL, KTTRUE`	if the stream object was (or is) closed

FUNCTION

Used to close a stream file. This closes the file but does not release the resources.

amStreamGetIsrCount 🚫

Gets the Interrupt Service Routine count.

FORMAT

```
#include <am.h>
KTU32 amStreamGetIsrCount(AM_STREAM_PTR theStream)
```

PARAMETERS

`AM_STREAM_PTR theStream`	the stream object to get the ISR count from

RETURN VALUE

`KTU32`	The number of times the ISR has been invoked for this stream

FUNCTION

Gets the number of times that the ISR has been invoked in this play cycle.

amStreamGetMemoryRequirement

Gets memory sizes necessary to play the stream.

FORMAT

```
#include <am.h>
KTBOOL amStreamGetMemoryRequirement(        AM_STREAM_PTR theStream,
KTU32 *transferBufferSize,
KTU32 *playBufferSize
)
```

PARAMETERS

`AM_STREAM_PTR theStream`	The stream object to get the requirement from
`KTU32 *transferBufferSize`	The size of the transfer buffer is returned via this pointer
`KTU32 *playBufferSize`	The size of the play buffer(s) are returned via this pointer

RETURN VALUE

`KTBOOL, KTTRUE`	If the memory requirements were returned
`KTFALSE`	If the stream was not open or is corrupt

FUNCTION

Gets the minimum amount of memory that will need to be passed into the `amStreamSetBuffers()` call.

Note: This must be called post the calls to `amStreamOpen()` and `amSetBufferSizes()`

amStreamGetMsPerIrq 🚫

Gets the number of milliseconds per callback.

FORMAT

```
#include <am.h>
KTBOOL    amStreamGetMsPerIrq(AM_STREAM_PTR theStream,KTU32 *millisecondsPerIrq)
```

PARAMETERS

`AM_STREAM_PTR theStream`	The stream object
`KTU32 *millisecondsPerIrq`	The number of milliseconds per callback

RETURN VALUE

`KTBOOL`

FUNCTION

Gets the number of milliseconds per callback. There are two callbacks per iteration of the play buffer which is playing at sample rate in frames, this resolves all of the variables to produce the number of milliseconds per callback.

Note: This must be called post the call to `amStreamOpen()` or it will return 0

amStreamGetNibblesPerFrame

Gets the number of nibbles in a frame.

FORMAT

```
#include <am.h>
KTBOOL    amStreamGetNibblesPerFrame(AM_STREAM_PTR theStream,KTU32 *nibblesPerFrame)
```

PARAMETERS

`AM_STREAM_PTR theStream`	The stream object
`KTU32 *nibblesPerFrame`	The number of nibbles in a frame of data

RETURN VALUE

`KTBOOL`

FUNCTION

Gets the number of nibbles in a frame for the sample format of the given stream.

Note: This must be called post the call to `amStreamOpen()` or it will return `0`

amStreamGetPan ⃠

Gets the streams current pan

FORMAT

```
#include <am.h>
KTU32 amStreamGetVolume(AM_STREAM_PTR theStream)
```

PARAMETERS

`AM_STREAM_PTR theStream`	The stream object to get the volume from
`KTU32 *pan`	The pan is returned via this pointer

RETURN VALUE

`KTBOOL`	`KTTRUE` on success
`KTFALSE`	on `NULL` parameter or track number out of range

FUNCTION

Gets the current pan of a stream.

amStreamGetSampleRate

Gets the real world sample rate of a stream.

FORMAT

```
#include <am.h>
KTBOOL    amStreamGetSampleRate(AM_STREAM_PTR theStream,KTU32 *sampleRate)
```

PARAMETERS

`AM_STREAM_PTR theStream`	The stream object
`KTU32` *sampleRate	The real world sample rate of a stream

RETURN VALUE

`KTBOOL`

FUNCTION

Returns the real world (44100, 22050, 11025, ...) sample rate of a stream. In the stream object the sample rate is AICA encoded and bears no resemblance to the real world rate. This allows access to a meaningful value for sample rate.

Note: This must be called post the call to `amStreamOpen()` or it will return `0`

amStreamGetTrackLengthInFrames

Gets the length of a stream in frames.

FORMAT

```
#include <am.h>
KTBOOL    amStreamGetTrackLengthInFrames(AM_STREAM_PTR theStream,KTU32 trackNumber,KTU32
*trackLengthInFrames)
```

PARAMETERS

`AM_STREAM_PTR theStream`	The stream object to get the length from
`KTU32 trackNumber`	The number of the track
`KTU32 *trackLengthInFrames`	The length of a stream in frames is returned via this pointer

RETURN VALUE

`KTBOOL`

FUNCTION

Gets the length of a stream in frames.

Note: This must be called post the call to `amStreamOpen()` or it will return `0` `amUtilGetNibblesPerFrame`()

amStreamGetVolume

Gets the streams current volume

FORMAT

```
#include <am.h>
KTU32 amStreamGetVolume(AM_STREAM_PTR theStream)
```

PARAMETERS

`AM_STREAM_PTR theStream`	The stream object to get the volume from
`KTU32 *volume`	The volume is returned via this pointer

RETURN VALUE

`KTBOOL, KTTRUE`	on success
`KTFALSE`	on `NULL` parameter or track number out of range

FUNCTION

Gets the current volume of a stream.

amStreamInitBuffer 🚫

Initializes a stream object to play a mono stream from a buffer.

FORMAT

```
#include <am.h>
KTBOOL amStreamInitBuffer(        AM_STREAM_PTR theStream,
volatile KTU32 *buffer,
KTU32 size,
KTU32 sampleRate,
KTU32 bitDepth)
```

PARAMETERS

`AM_STREAM_PTR theStream`	The stream object to be initialized
`volatile KTU32 *buffer`	A buffer in either sh4 memory or sound memory
`KTU32 size`	The size of the buffer

Note: The buffer **MUST** be a multiple of the play buffer size.

`KTU32 sampleRate`	The real world integral sample rate of the file, 44100, 22050, or 11025
`KTU32 bitDepth`	4,8 or 16

RETURN VALUE

`KTBOOL`, `KTTRUE`	if the stream object was successfully initialized

FUNCTION

Sets the members of the stream object necessary to the preparation for a call to am`StreamOpen()`, this allows the playback of a chunk of headerless raw sound data. This is the way to play a stream that is to be constructed at play time. When preparing the buffer it should be sized to be an even multiple of the playbuffer size, allocate the buffer wherever you want it, sound or sh4 memory, then fill it with silence.

For ADPCM data silence is 0x80, for 8 and 16 bit data silence is 0x00.

For a 16 bit44.1k memory stream a 4096 byte play buffer is sufficient.

Note: This does not use the `streamIO` subsystem, it is also possible to play multitrack buffers using that subsystem.

ICON LEGEND:	N	NEW function for R10	🚫	OBSOLETE function for R10	R	REVISED function for R10

amStreamInitFile

Initializes a stream object to play a file.

FORMAT

```
#include <am.h>
KTBOOL amStreamInitFile(       AM_STREAM_PTR theStream,KTSTRING fileName)
```

PARAMETERS

`AM_STREAM_PTR theStream`	the stream object to be initialized
`KTSTRING fileName`	the file name of the .str file to stream

RETURN VALUE

`KTBOOL`, `KTTRUE`	if the stream object was successfully initialized

FUNCTION

Sets the members of the stream object necessary to the preparation for a call to `amStreamOpen()`

Note: if the length of the filename is in excess of `AM_STREAM_FILENAME_LEN` the call will fail.

amStreamInstallUserCallback ⃠

Installs a user callback for a stream.

FORMAT

```
#include <am.h>
KTBOOL amStreamInstallUserCallback(AM_STREAM_PTR theStream,AM_USER_CALLBACK userCallback)
```

PARAMETERS

`AM_STREAM_PTR theStream`	The stream object
`AM_USER_CALLBACK userCallback`	The address of the callback function, see `am.h`

RETURN VALUE

`KTEAM_PTEAM_PTR theStream`	The stream object
`AM_USER_CALLBACK`	Object has not been opened or is corrupt

FUNCTION

This function will `call _amVoiceInstallUserCallback` to install a user callback into the interrupt handling system.

The callback will be issued when the stream is stopped via `amStreamStop` or the stream reaches the end.

Note: This must be called post the call to `amStreamAllocateVoiceChannels()` and the call to `amStreamOpen()`.

amStreamIoInstallAlternateIoManager

Installs a custom Io proc.

FORMAT

```
#include <am.h>
void amStreamIoInstallAlternateIoManager(AM_STREAM_IO_PROC ioProc)
```

PARAMETERS

`AM_STREAM_IO_PROC ioProc`	A pointer to a custom Io proc, see the example in `MyFile.c`

RETURN VALUE

void

FUNCTION

Installs a custom Io proc into the Io shell, this allows all file system calls to be intercepted by the applications file system.

The prototype for the IO proc is as follows:

```
KTBOOL MyCustomIoProc(            KTSTRING fileName,
KTU32 * sd,
KTU8 * buffer,
KTU32 * size,
AM_FILE_OPERATION_MODE mode
)
```

An example of this redirection is available in `MyFile.c` as well as a boilerplate copy of the IO proc for modification.

amStreamIsMono ⃠

Tells if a stream is mono.

FORMAT

```
#include <am.h>
KTBOOL amStreamIsMono(AM_STREAM_PTR theStream)
```

PARAMETERS

`AM_STREAM_PTR theStream`	the stream object to be queried

RETURN VALUE

`KTBOOL, KTTRUE`	if the stream is mono

FUNCTION

If the given stream is mono this will return `KTTRUE`.

amStreamIsr0 - 4

Interrupt Service Routine for the amStream subsystem.

FORMAT

```
#include <am.h>
void _amStreamIsr0(KTU32 streamPtr)
void _amStreamIsr1(KTU32 streamPtr)
void _amStreamIsr2(KTU32 streamPtr)
void _amStreamIsr3(KTU32 streamPtr)
void _amStreamIsr4(KTU32 streamPtr)
```

PARAMETERS

RETURN VALUE

`KTTRUE`	On success
`KTFALSE`	

FUNCTION

ISR routine for the `amStream` subsystem. These routines are used as the `theIsr` argument to the `amStreamSetIsr()` call.

See Also: `KTBOOL amStreamSetIsr(AM_STREAM_PTR theStream,AM_STREAM_ISR theIsr)`

amStreamIsStereo ⊘

Tells if a stream is stereo.

FORMAT

```
#include <am.h>
KTBOOL amStreamIsStereo(AM_STREAM_PTR theStream)
```

PARAMETERS

`AM_STREAM_PTR theStream`	the stream object to be queried

RETURN VALUE

`KTBOOL`, `KTTRUE`	if the stream is stereo

FUNCTION

If the given stream is stereo this will return `KTTRUE`.

amStreamOpen

Opens a stream object.

FORMAT

```
#include <am.h>
KTBOOL amStreamOpen(AM_STREAM_PTR theStream)
```

PARAMETERS

`AM_STREAM_PTR theStream`	the stream object to be opened

RETURN VALUE

`KTBOOL, KTTRUE`	if the stream was successfully started

FUNCTION

Opens the stream file named in the `InitStream()` call, aquires buffers and "primes" the play buffer(s) with data from the transfer buffer.

amStreamPlaying

Monitors if a stream is currently playing.

FORMAT

```
#include <am.h>
KTBOOL _amStreamPlaying(AM_STREAM_PTR theStream)
```

PARAMETERS

`AM_STREAM_PTR theStream`	the stream object to be monitored for play activity

RETURN VALUE

`KTBOOL, KTTRUE`	if the stream is currently playing

FUNCTION

Used to monitor the play status of a stream.

amStreamPrimeBuffers

Primes the play buffer.

FORMAT

```
#include <am.h>
KTBOOL amStreamPrimeBuffers(AM_STREAM_PTR theStream)
```

PARAMETERS

`AM_STREAM_PTR theStream`	The stream object to be primed

RETURN VALUE

`KTBOOL`, `KTTRUE`	If the stream was successfully primed
`KTFALSE`	If the stream was not open If the read failed If the stream is corrupt

FUNCTION

Moves the first load of data into the transfer and play buffer(s) for the given stream.

Note: This must be called post the call to `amStreamOpen()`.

amStreamReleaseVoiceChannels

Releases voice channels allocated for a stream.

FORMAT

```
#include <am.h>
KTBOOL amStreamReleaseVoiceChannels(AM_STREAM *theStream)
```

PARAMETERS

`AM_STREAM *theStream`	The stream object to be set

RETURN VALUE

`KTBOOL, KTTRUE`	if voice(s) were released.
`KTFALSE`	if `theStream` was `NULL` or had not been opened with `amStreamOpen()`

FUNCTION

Used when terminating the building of a stream object for which voice channels had already been allocated.

amStreamRewind

Rewinds an open stream to its start.

FORMAT

```
#include <am.h>
KTBOOL amStreamRewind(AM_STREAM_PTR theStream)
```

PARAMETERS

`AM_STREAM_PTR theStream`	The stream object to rewind

RETURN VALUE

`KTBOOL, KTTRUE`	If the stream was successfully rewound
`KTFALSE`	If the seekrewind call failed or the stream is not open

FUNCTION

Allows an open stream to be rewound to its start with out closing and reopening the file to get to its beginning.

amStreamServer

Serves data to a currently playing stream.

FORMAT

```
#include <am.h>
KTBOOL amStreamServer(AM_STREAM_PTR theStream)
```

PARAMETERS

`AM_STREAM_PTR theStream`	the stream object to be served

RETURN VALUE

`KTBOOL, KTTRUE`	if the stream is currently playing
`KTFALSE`	if it is finished

FUNCTION

Fills the transfer buffer of a given stream when its contents have been completely transferred by the ISR.

amStreamSetBuffers

Sets buffer memory pointers in a stream.

FORMAT

```
#include <am.h>

KTBOOL amStreamSetBuffers(        AM_STREAM *theStream,
KTU32 *transferBuffer, KTU32 transferBufferSize,
KTU32 *playBuffer, KTU32 playBufferSize)
```

PARAMETERS

`AM_STREAM *theStream`	The stream object to be set
`KTU32 *transferBuffer`	The transfer buffer memory
`KTU32 transferBufferSize`	
`KTU32 *playBuffer`	The play buffer memory
`KTU32 playBufferSize`	

RETURN VALUE

`KTBOOL, KTTRUE`	If the buffer pointers were set
`KTFALSE`	If the `transferBuffer` is not in SH4 memory If the `playBuffer` is not in AICA memory

FUNCTION

Sets the buffers necessary to run a stream, stereo and multi track streams require a play buffer per channel. This routine will subdivide the buffer passed in for the play buffer as necessary for the given stream.

Note: This must be called post the call to `amStreamSetBufferSizes()` and the call to `amStreamOpen()`

amStreamSetBufferSizes ⃠

Sets the sizes for the play and transfer buffers.

FORMAT

```
#include <am.h>
void amStreamSetBufferSizes(          AM_STREAM_PTR theStream,
KTU32 transferBufferSize,
KTU32 playBufferSize)
```

PARAMETERS

`AM_STREAM_PTR theStream`	The stream object to be set
`KTU32 transferBufferSize`	The size of the transfer buffer
`KTU32 playBufferSize`	The size of a play buffer

RETURN VALUE

`KTBOOL, KTTRUE`	If the sizes were set
`KTFALSE`	If the stream is already open

FUNCTION

Sets the buffer sizes in a stream that is not currently open.

Currently the basic recommendations for buffer sizes are as follows:

1) Play buffer size must be a multiple of 2048

2) Mono streams play well with a 2048 byte play buffer, stereo with a 4096 byte play buffer.

3) Transfer buffer size should be as follows: `transferBufferSize  =   (playBufferSize * 2)`

Note: This must be called **PRIOR** to the call to `amStreamOpen()`

amStreamSetHeaderBuffer

Sets the buffer for loading the stream header sector.

FORMAT

```
#include <am.h>

KTBOOL amStreamSetHeaderBuffer(AM_STREAM *theStream,KTU8 *headerBuffer)
```

PARAMETERS

`AM_STREAM *theStream`	The stream object to be set.
`KTU8 *headerBuffer`	A pointer to a 32 byte aligned 2048 byte buffer.

RETURN VALUE

`KTBOOL, KTTRUE`	If successful
`KTFALSE`	If `theStream` is `NULL` If `headerBuffer` is `NULL` If `headerBuffer` is not 32 byte aligned

FUNCTION

Sets the headerBuffer member of the `AM_ASYNC_STREAM` structure. The header is loaded at the start of the stream and dereferenced for its information in the streams warm up phase.

amStreamSetIsr

Sets the streams data transfer ISR.

FORMAT

```
#include <am.h>
KTBOOL amStreamSetIsr(AM_STREAM_PTR theStream,AM_STREAM_ISR theIsr)
```

PARAMETERS

`AM_STREAM_PTR theStream`	The stream object to be set.
`AM_STREAM_ISR theIsr`	A pointer to a data transfer ISR

Library ISR identifiers:

```
void _amStreamIsr0         (KTU32 streamPtr)
void _amStreamIsr1         (KTU32 streamPtr)
void _amStreamIsr2         (KTU32 streamPtr)
void _amStreamIsr3         (KTU32 streamPtr)
void _amStreamIsr4         (KTU32 streamPtr)
```

RETURN VALUE

`KTBOOL, KTTRUE`	If the ISR was installed successfully
`KTFALSE`	If the stream was not open

FUNCTION

Sets the streams ISR that pumps data from the transfer buffer to the `play buffer(s)`. To determine if a stream is mono or stereo, post `amStreamOpen()`, use the calls:

`amStreamIsMono() or amStreamIsStereo().`

Note: This must be called post the calls to `amStreamOpen()` and `amStreamAllocateVoiceChannels()`

amStreamSetMix

Sets volume and pan for all tracks in a stream.

FORMAT

```
#include <am.h>
KTBOOL amStreamSetMix(AM_STREAM_PTR theStream,AM_STREAM_MIX_PTR theMix)
```

PARAMETERS

`AM_STREAM_PTR theStream`	The stream object to be set
`AM_STREAM_MIX_PTR theMix`	The new scene mix to be set

RETURN VALUE

`KTBOOL`, `KTTRUE`	if the mix was successfully set

FUNCTION

Sets the volume and pan of all tracks in a stream to new values. If a value in the new scene is the same as the current value the command is not sent.

amStreamSetPan

Sets the pan on a mono stream.

FORMAT

```
#include <am.h>
KTBOOL amStreamSetPan(AM_STREAM_PTR theStream,KTU8 newPan)
```

PARAMETERS

`AM_STREAM_PTR theStream`	The stream object of the stream to be panned
`KTU8 newPan`	The new setting for the pan (0-127), right to left

RETURN VALUE

`KTBOOL, KTTRUE`	if the pan was successfully set
`KTFALSE`	if the stream is not playing or the stream is stereo

FUNCTION

Sets the pan of a currently playing MONO stream, if a stereo sream is submitted the call will return `KTFALSE` as stereo streams can not be panned.

amStreamSetTransferMethod

Selects DMA or memcpy as the data transfer method.

FORMAT

```
#include <am.h>
KTBOOL amStreamSetTransferMethod(       AM_STREAM_PTR theStream,
AM_STREAM_TRANSFER_METHOD transferMethod,
KTU32 dmaChannel,
KTU32 dmaFrameSize)
```

PARAMETERS

`AM_STREAM_PTR theStream`	the stream object
`AM_STREAM_TRANSFER_METHOD transferMethod`	either `AM_STREAM_DMA` or `AM_STREAM_NON_DMA`
`KTU32 dmaChannel`	`AM_DMA_CHANNEL`
`KTU32 dmaFrameSize`	4,8 or 32 bytes per frame

RETURN VALUE

`KTTRUE`	on success

`KTFALSE`	if `theStream` is open
	if `theStream` is `NULL`
	if `transferMethod` is not `AM_STREAM_DMA` or `AM_STREAM_NON_DMA`
	if `dmaChannel` is not `AM_DMA_CHANNEL`
	if `dmaFrameSize` is not 4,8 or 32

FUNCTION

Causes the stream to be transfered via DMA rather then using the foreground memcpy process. `dmaFrameSize` controls how many bytes are transferred in a single dma transfer.

Note: This must be called prior to `StreamOpen()`

amStreamSetVolume

Sets the volume on a stream.

FORMAT

```
#include <am.h>
KTBOOL amStreamSetVolume(AM_STREAM_PTR theStream,KTU8 newVolume)
```

PARAMETERS

`AM_STREAM_PTR theStream`	The stream object of the stream to have its volume set
`KTU8 newVolume`	The new volume to set (0-127)

RETURN VALUE

`KTBOOL`, `KTTRUE`	if the pan was successfully set
`KTFALSE`	if the stream is not playing

FUNCTION

Sets the volume of the currently playing mono or stereo stream.

amStreamStart

Starts a stream object playing.

FORMAT

```
#include <am.h>
KTBOOL amStreamStart(AM_STREAM_PTR theStream)
```

PARAMETERS

`AM_STREAM_PTR theStream`	the stream object to be started

RETURN VALUE

`KTBOOL`, `KTTRUE`	if the stream was successfully started

FUNCTION

Starts the given stream object playing, aquires callback procs and aquires and configures the ports based on the type of stream contained in the `.str` file.

amStreamStop

Stops a currently playing stream.

FORMAT

```
#include <am.h>
KTBOOL amStreamStop(AM_STREAM_PTR theStream)
```

PARAMETERS

`AM_STREAM_PTR theStream`	the stream object of the stream to be stopped

RETURN VALUE

`KTBOOL`, `KTTRUE`	if the stream was successfully stopped

FUNCTION

Used to stop a currently playing stream, this routine is called by `amStreamServer()` at the end of a stream. This closes and frees the port, removes and releases the callback and releases the port and the buffers from the stream.

AmSystemSetVolumeMode N

Switches AM layer volume mapping to linear.

FORMAT

```
#include <am.h>

void amSystemSetVolumeMode(KTBOOL mode)
```

PARAMETERS

`Mode`	if `USELINEAR` will convert volume parameter to “linear” volume mapping, if `USELOG` will default to AICA hardware values

RETURN VALUE

void

FUNCTION

Switches volume mapping between LOG based power curve and “LINEAR” based power curve.

amUtilAlignNumber

Performs numerical boundry alignment.

FORMAT

```
#include <am.h>
KTBOOL amUtilAlignNumber(KTU32 theNumber,KTU32 theAlignment,KTU32 *theResult)
```

PARAMETERS

`KTU32 theNumber`	the number to be aligned
`KTU32 theAlignment`	the desired boundry
`KTU32 *theResult`	the aligned number is returned via this pointer

RETURN VALUE

`KTTRUE`	on success

`KTFALSE`	if alignment is 0
	if `theResult` is `NULL`

FUNCTION

Rounds a number up to the next multiple of alignment. If the number is evenly divisible by alignment it will be returned untouched.

amUtilGetAicaSampleRate

Makes a real world sample rate into an AICA sample rate.

FORMAT

```
#include <am.h>
KTBOOL amUtilGetAicaSampleRate(KTU32 realWorldSampleRate,KTS32 *aicaSampleRate)
```

PARAMETERS

`KTU32 realWorldSampleRate`	The real world sample rate, i.e. 44100, 22050, 11025, 5012
`KTS32 *aicaSampleRate`	The returned AICA sample rate

RETURN VALUE

`KTTRUE`	on success
`KTFALSE`	on fail

FUNCTION

Extrapolates from real world sample rates to AICA sample rates, the only allowed AICA rates are 44100, 22050, 11025 and 5012. Using any other rate will cause this function to fail.

amUtilGetAicaSampleType

Extrapolates sample bit depth to AICA sample type.

FORMAT

```
#include <am.h>
KTBOOL amUtilGetAicaSampleType(KTU32 bitDepth,AC_AUDIO_TYPE_PTR aicaSampleType)
```

PARAMETERS

`KTU32 bitDepth`	The bit depth of the sample.
`AC_AUDIO_TYPE_PTR aicaSampleType`	The returned AICA sample type.

RETURN VALUE

`KTTRUE`	on success
`KTFALSE`	on fail

FUNCTION

Extrapolates sample bit depth to AICA sample type. See the `AC_AUDIO_TYPE` enum in `ac.h` for the types returned by this function.

Note: This will always identify 4 bit data as the type `AC_ADPCM_LOOP`

amUtilGetAicaVolume

Converts midi volume units to AICA units

FORMAT

```
#include <am.h>
KTBOOL amUtilGetAicaVolume(KTU32 midiVolume,KTU32 *aicaVolume)
```

PARAMETERS

`KTU32 midiVolume`	the midi volume to be converted (0-127)
`KTU32 *aicaVolume`	the AICA volume (0-15) is returned via this pointer

RETURN VALUE

`KTTRUE`	on success
`KTFALSE`	if aicaVolume is `NULL`

FUNCTION

Converts from midi volume units (0-127) to AICA (0-15) volume units.

Note: If midiVolume is > 127 it will be set to 127 and a debug warning issued.

amUtilGetEndOfBufferInFrames

Calculates the end of the buffer in frames.

FORMAT

```
#include <am.h>
KTBOOL amUtilGetEndOfBufferInFrames(KTU32 bitDepthOfSample,KTU32
sizeOfBufferInBytes,KTU32 * middleInFrames)
```

PARAMETERS

`KTU32 bitDepthOfSample`	the bit depth of the sample data
`KTU32 sizeOfBufferInBytes`	the size of the buffer in bytes
`KTU32 * endInFrames`	the offset of the end in frames is returned via this value

RETURN VALUE

`KTBOOL, KTTRUE`	if the calculation was successful
`KTFALSE`	if the bit depth is unsupported

FUNCTION

Calculates the end of the buffer in frames.

amUtilGetLengthInFrames

Gets the length of a stream in frames.

FORMAT

```
#include <am.h>
KTU32 amUtilGetLengthInFrames(AC_AUDIO_TYPE type,KTU32 channels,KTU32 size,KTU32
*lengthInFrames)
```

PARAMETERS

`AC_AUDIO_TYPE type`	the AICA type of the sound, see `ac.h`
`KTU32 channels`	the number of channels in the sound, 1=mono 2=stereo
`KTU32 size`	the size of the sound in bytes
`KTU32 *lengthInFrames`	the length is reurned via this pointer.

RETURN VALUE

`KTU32`	The length of a stream in frames

FUNCTION

Gets the length of a stream in frames.

Note: This must be called post the call to `amUtilOpen()` or it will return `0`

amUtilGetLengthInMs

Gets the length of a stream in milliseconds.

FORMAT

```
#include <am.h>
KTBOOL    amUtilGetLengthInMs(AC_AUDIO_TYPE type,KTU32 channels,KTU32 size,KTU32
aicaSampleRate,KTU32 *lengthInMs)
```

PARAMETERS

`AC_AUDIO_TYPE type`	the AICA type of the sound, see `ac.h`
`KTU32 channels`	the number of channels in the sound, 1=mono 2=stereo
`KTU32 size`	the size of the sound in bytes
`KTU32 aicaSampleRate`	the AICA sample rate of the sound, see `ac.h`
`KTU32 *lengthInMs`	the length of a stream in milliseconds.

RETURN VALUE

`KTBOOL`

FUNCTION

Gets the length of a stream in milliseconds.

Note: This must be called post the call to `amUtilOpen()` or it will return `0`

amUtilGetMiddleOfBufferInFrames

Calculates the middle of the buffer in frames.

FORMAT

```
#include <am.h>
KTBOOL amUtilGetMiddleOfBufferInFrames(KTU32 bitDepthOfSample,KTU32
sizeOfBufferInBytes,KTU32 * middleInFrames)
```

PARAMETERS

`KTU32 bitDepthOfSample`	the bit depth of the sample data
`KTU32 sizeOfBufferInBytes`	the size of the buffer in bytes
`KTU32 * middleInFrames`	the offset of the middle in frames is returned via this value

RETURN VALUE

`KTBOOL`, `KTTRUE`	if the calculation was successful, `KTFALSE` if the bit depth is unsupported

FUNCTION

Calculates the middle of the buffer in frames.

amUtilGetMsPerIrq

Gets the number of milliseconds per callback.

FORMAT

```
#include <am.h>
KTBOOL    amUtilGetMsPerIrq(AC_AUDIO_TYPE type,KTU32 aicaSampleRate,KTU32
playBufferSizeInBytes,KTU32 *msPerIrq)
```

PARAMETERS

`AC_AUDIO_TYPE type`	the AICA type of the sound, see `ac.h`
`KTU32 aicaSampleRate`	the AICA sample rate of the sound, see `ac.h`
`KTU32 playBufferSizeInBytes`	the size of the playback buffer in bytes
`KTU32 *msPerIrq`	the number of milliseconds per callback

RETURN VALUE

void

FUNCTION

Gets the number of milliseconds per callback. There are two callbacks per iteration of the play buffer which is playing at sample rate in frames, this resolves all of the variables to produce the number of milliseconds per callback.

Note: This must be called post the call to `amUtilOpen()` or it will return `0`

amUtilGetNibblesPerFrame

Gets the number of nibbles in a frame.

FORMAT

```
#include <am.h>
KTBOOL    amUtilGetNibblesPerFrame(AC_AUDIO_TYPE type,KTU32 *nibblesPerFrame)
```

PARAMETERS

`AC_AUDIO_TYPE type`	the AICA type of the sound, see `ac.h`
`KTU32 *nibblesPerFrame`	the number of nibbles in a frame is returned via this pointer

RETURN VALUE

`KTBOOL`

FUNCTION

Gets the number of nibbles in a frame for the sample format of the given stream.

amUtilGetSampleRate

Gets the real world sample rate of a stream.

FORMAT

```
#include <am.h>
KTBOOL    amUtilGetSampleRate(KTU32 aicaSampleRate,KTU32 *sampleRate)
```

PARAMETERS

`KTU32 aicaSampleRate`	the AICA sample rate as defined in `ac.h`
`KTU32 *sampleRate`	The real world sample rate of a stream is returned via this pointer.

RETURN VALUE

`KTBOOL`	fails if `sampleRate` is `NULL` or `aicaSampleRate` is not a correct value.

FUNCTION

Returns the real world (44100, 22050, 11025, ...) sample rate of a stream. In the stream object the sample rate is AICA encoded and bears no resembalance to the real world rate. This allows access to a meaningful value for sample rate.

Note: This must be called post the call to `amUtilOpen()` or it will return `0`

amVoiceAllocate

Allocates a voice channel.

FORMAT

```
#include <am.h>

KTBOOL amVoiceAllocate(KTU32 * voiceChannel,AM_VOICE_TYPE voiceType,void * owner)
```

PARAMETERS

`KTU32 * voiceChannel`	The voice channel allocated is returned via this pointer
`AM_VOICE_TYPE voiceType`	The type, `AM_ONESHOT_VOICE` or `AM_STREAM_VOICE`, see `am.h`
`void * owner`	The address of the `AM_SOUND` or `AM_STREAM` object that holds the channel

RETURN VALUE

`KTTRUE`	If a channel was available
`KTFALSE`	If no channels are available

FUNCTION

Allocates voice channels (DA port #'s) needed for playing sounds with the ac layer. If the type is `AM_ONESHOT_VOICE` the owner arg is the address of the `AM_SOUND` structure that holds the sound to be played on that channel.

If the type is `AM_STREAM_VOICE` the owner arg is the address of the `AM_STREAM` structure. If the type is `AM_MIDI_VOICE` the midi port number that must be used is the following:

```
(voiceChannel - AM_FIRST_MIDI_VOICE)
```

The midi ports are 0-15 but the driver reports them as event numbers 16-31 in the interrupt message array.

amVoiceFree Ⓝ

Frees a voice channel.

FORMAT

```
#include <am.h>

KTBOOL amVoiceFree(KTU32 voiceChannel)
```

PARAMETERS

`KTU32 voiceChannel`	The voice channel to free.

RETURN VALUE

`KTBOOL, KTTRUE`	If voice channel is successfully freed.

FUNCTION

Sets the internal voice structure to a clean state.

amVoiceInit

Initializes the voice pool.

FORMAT

```
#include <am.h>
void amVoiceInit(void)
```

PARAMETERS

void

RETURN VALUE

void

FUNCTION

Initializes the voice pool and voice (port) allocation functionality.

amVoiceInstallUserCallback

Installs a user callback.

FORMAT

```
/#include <am.h>

KTBOOL amVoiceInstallUserCallback(KTU32 voiceChannel, AM_USER_CALLBACK userCallbackProc)
```

PARAMETERS

`KTU32 voiceChannel`	The voice channel for the callback.
`AM_USER_CALLBACK userCallbackProc`	The callback procedure to be installed.

RETURN VALUE

`KTBOOL, KTTRUE`	If the user callback is successfully installed.

FUNCTION

Installs a user callback procedure into the indicated channels data. When the end of the buffer is reached this procedure will be invoked during the ARM interrupt.

The voice channel that triggers the callback is passed to the user callback.

amVoiceSetStreamIsr

Installs an ISR for a voice channel.

FORMAT

```
#include <am.h>

KTBOOL amVoiceSetStreamIsr(KTU32 voiceChannel, AM_STREAM_ISR streamIsr)
```

PARAMETERS

`KTU32 voiceChannel`	The voice channel for the ISR.
`AM_STREAM_ISR streamIsr`	The ISR to be installed.

RETURN VALUE

`KTBOOL, KTTRUE`	If the ISR is successfully installed.

FUNCTION