UADE 2.xx design specification

Heikki Orsila <heikki.orsila@iki.fi>

1 Introduction

UADE (*Unix Amiga Delitracker Emulator*) plays various Amiga music formats by hardware and software emulation. Approximately 200 formats are known to work more or less properly.

Despite considerable performance requirements of emulating hardware and software, UADE requires only approximately 5% CPU resources with a current (2006-01-01) computer system (2.0 GHz AMD64).

The full system consists of 4 layers shown in Figure **??**. The frontend layer handles all user-interface and high-level control issues. The frontend relays user-induced commands for the emulator. The emulator is programmed to emulate a reduced Amiga 500 model and load boot software and data for it. The Amiga model is called reduced because it lacks functionality of some peripheral devices and custom graphics chips. It only needs to run MC68000 software to produce sound through Amigas audio chip. The loaded software consists of 2 two components, which are the sound core (*score*) and some eagleplayer plugin. The loaded data is a song to be played by the eagleplayer plugin. The emulated machine is booted after this. Boot procedure starts execution from the score, which initializes emulated hardware and setups an environment that contains some essential features from AmigaOS and Eagleplayer APIs. The score executes the eagleplayer plugin to play the music with the data that was loaded. The eagleplayer plugin can load additional data files from the native host operating system if required.

This document explains internal issues between the frontend and the emulator. Interaction between the emulated and native software is not documented.

2 History

UADE 1.xx (2000-04-09) was written to be a stand-alone program without consideration to implementing many different user interfaces (*frontends*) for it. There was no internal structure to implement different frontends easily, which became rapidly a big problem because users requested many different kinds

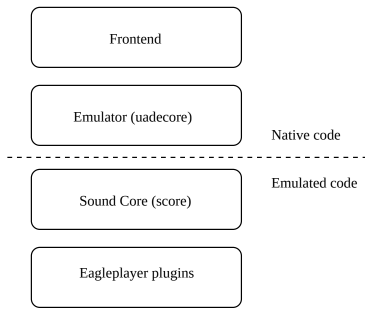

Figure 1: UADE software layers

of interfaces. By much hacking some kind of pseudo-interface was created to facilitate following frontends:

- Beep Media Player
- MorphOS shell without interaction
- Unix shell without interaction
- Unix shell with small interaction
- XMMS plugin

To force the separation of user interface and effective functionality, UADE 2.00 (2006-01-01) removed all the user interface code from the emulator part (*uadecore*).

3 Message-passing protocol

In UADE 2.xx the emulator (uadecore) became an independent process without any user interfaces. Any frontend, or client, that wants to use its services must communicate with the uadecore by using a token-passing based messaging protocol. The protocol is of course implemented by interprocess communication (IPC).

The basic idea of the protocol is that the frontend is the client who issues commands for the server (uadecore). Uadecore may not send any commands at all. Uadecore only sends replies to commands issued by the client. Also, the client never replies anything back to the uadecore.

The communication protocol is based on the concept of *tokens*. Only the party that has a token (there is only one) may send messages. Messages are either commands or replies. Client sends messages and uadecore sends replies. Both of them have to send the token back sometimes. The party that doesn't have the token must reply to all commands sent by the other party.

Initially the client has the token so it may send commands for the uadecore.

3.1 Messaging protocol commands

The messaging protocol has following commands which are only sent by the client. There is an exception, however. The uadecore may send a *token*-command, but it is not really considered a command. All the commands can be found from the file src/include/uadeipc.h.

Config command is used to pass a file name of the emulation configuration file for the uadecore. The file is named *uaerc*.

Score command is used to pass file name of a binary run-time in M68k machine language for the uadecore. The binary run-time is called *score* or *sound core*. The sound core contains implementations of *Eagleplayer* and *AmigaOS* APIs.

Player command is used to pass a file name of a binary player plugin in M68k machine language for the uadecore. This is also called an *Eagleplayer plugin*.

Module command is used to pass a file name of a song to be played for the uadecore.

Read command is used to request more sound data from the uadecore.

Reboot command is used to halt playback synthesis of uadecore.

Set subsong command is used to set the initial subsong for playback.

Ignore check command is not necessary (will be documented later, if ever).

Song end not possible command is not necessary.

Set ntsc command is not necessary.

Filter command is used to select between A500, A1200 or no filter emulation.

Set frequency command is used to set sample rate for output.

Set resampling mode command is not necessary.

Speed hack command is not necessary.

Change subsong command is not necessary.

Activate debugger command is not necessary.

Token command is used to pass back the token for the other party.

3.2 Messaging protocol replies for commands

Messages are answered by following replies. All the replies can be found from the file src/include/uadeipc.h.

MSG reply is just any random text string message that the client should know. Could be spam, blackmailing, empty threats or last words of a dying process.

Can't play reply is issued by the uadecore if it is not able to play a given triplet of score, player and module.

Can play reply is issued by the uadecore if it can play a given triplet of score, player and module.

Song end reply is issued by the uadecore to indicate that playback has ended.

Subsong info is issued by the uadecore when it knows the amount of subsongs contained in the song. This happens a short while after playback has started, or usually during the first fraction of a second since playback has been started.

Player name is a reply containing the human (geek?) readable player name that is decoded by the eagleplayer plugin,

Module name is a reply containing the human readable form of the module name decoded by the eagleplayer plugin.

Format name is a reply containing the human readable form of the song format that is being played.

Data reply is issued by the uadecore to pass synthesized sample data back to the client. This is a reply for the *read*-command.

.

3.3 Message format

All messages are sent as finite sized byte sequences. Each message begins with a header shown in Table **??**. **msgtype** field is simply an unsigned 32-bit integer in a packed binary structure. The integer is sent in network byte-order aka big-endian format. All integers are sent in big-endian format. The **size** field is 32-bit length of the bytes coming after the header (which is in big-endian format). Notice the use of C99 empty record in a structure (excuse the annoying GCC feature there that forces the structure to be tightly packed) called **data**. **data** has zero size and thus the whole message size in memory is $8 + size$ bytes. The full size must not be over 4096 bytes, and thus **size** can be at most 4088.

Table 1: Messages header

uint32_t	msgtype (big-endian)
uint32_t	size (big-endian)
uint8_t[]	data

Table 2: Short message

uint32_t	msgtype
uint32_t	0

There are 3 types of messages: short messages, string messages and custom messages. Each follow the given low-level format but the contents differs.

Messages C language definition is:

```
struct uade_msg {
  uint32_t msgtype;
  uint32_t size;
  uint8_t data[];
} __attribute__((packed));
```

3.3.1 Short message

Short message has the value 0 in **size** field implying that there is no special payload with the message. Sending a token, for example, is such a message. Short messages are sent with **uade_send_short_message** and received with **uade_receive_short_message**. The structure of short message is shown in Table **??**. Table **??** shows all short messages.

Table 3: Short messages

Command	Activate debugger
Command	Ignore check
Command	Reboot
Command	Song end not possible
Command	Speed hack
Command	Token
Reply	Can play
Reply	Can't play

Table 4: String message

uint32_t	msgtype
uint32_t	$x + 1$, where **x** is the number of letters.
uint8_t[]	$x + 1$ bytes.

Table 5: String messages

Command	Config
Command	Module
Command	Player
Command	Score
Command	Set resampling mode
Reply	Format name
Reply	Module name
Reply	MSG
Reply	Player name

3.3.2 String message

String message is a message containing one zero-terminated text string. A string message is sent by **uade_send_string** and received by **uade_receive_string**. Table **??** shows the structure of a string message. Table **??** shows all string messages.

3.3.3 Custom message

Custom message is any kind of message which is not short or string message. We quickly present all different custom messages in following tables.

Table **??** shows format of sound data reply. Table **??** shows format of subsong info reply. Table **??** shows format of song end reply. Table **??** shows format of read command. Table **??** shows format of filter command. Table **??** shows format of set frequency command. Table **??** shows format of subsong commands which are set subsong and change subsong.

Table 6: Data reply message

uint32_t	Reply: Data
uint32_t	x, where **x** is the number of sample data bytes.
uint8_t[]	x sample data bytes. The format is 16-bit interleaved PCM stereo (big-endian).

Table 7: Subsong info reply message

uint32_t	Reply: Subsong info
uint32_t	12
uint32_t	Minimum subsong (big-endian)
uint32_t	Maximum subsong (big-endian)
uint32_t	Current subsong (big-endian)

Table 8: Song end reply message

uint32_t	Reply: Song end
uint32_t	$8 + x + 1$, where **x** is length of an explanation string.
uint32_t	Number of bytes of sample data valid in the next data reply (big-endian)
uint32_t	Unintentional end: 0 or 1. 0 means unintentional. 1 means an error resulted into the song end. (big-endian)
uint8_t[]	$x + 1$ bytes containing a textual reason for song end.

Table 9: Read command

uint32_t	Command: Read
uint32_t	4
uint32_t	Number of bytes of sample data to be read (big-endian)

Table 10: Filter command

uint32_t	Command: Filter
uint32_t	8
uint32_t	Filter type. See src/include/amigafilter.h for values (big-endian).
uint32_t	$2x + y$, where **x** means whether or not filter state should be forced and **y** is filter state to be set. Both x and y are either 0 or 1. (big-endian)

Table 11: Frequency command

uint32_t	Command: Set frequency
uint32_t	4
uint32_t	Sample rate (big-endian)

Table 12: Subsong command

uint32_t	Command: change subsong or set subsong
uint32_t	4
uint32_t	New subsong number (big-endian).

4 Interaction example

To illustrate the dialog between client and server, look at the following log between uade123 and uadecore:

```
<This is done once during client startup>
uade123: sending string 1 (command: config)

<Here begins play back of new song. this is done for each song.>
uade123: sending string 2 (command: score)
uade123: sending string 3 (command: player)
uade123: sending string 4 (command: module)
uade123: sending message 16 (token pass)

<Now the uadecore is pondering whether or not the thing can be played>
uadecore: sending message 19 (reply: can play)
uadecore: sending message 16 (token pass back)

<It's okay to play, so send some additional commands for uadecore>
uade123: sending message 11 (command: filter. not needed.)
uade123: sending string 12 (command: interpolation. not needed.)

<Then start audio synthesis by issuing read command>
uade123: sending message 5 (command: read)
uade123: sending message 16 (token pass)

<uadecore starts to synthesize and floods back messages>
uadecore: sending string 17 (reply: message. don't care.)
uadecore: sending string 22 (reply: player name back. don't care.)
uadecore: sending string 23 (reply: module name back. don't care.)
uadecore: sending message 21 (reply: subsong info)
uadecore: sending string 17 (reply: birthday congratulation message)
uadecore: sending message 25 (reply: data)
uadecore: sending message 25 (reply: data)
uadecore: sending message 16 (token back as all the requested data has been sent)

<Okay, the first round of messages was good. Next read then..>
uade123: sending message 5 (command: read. 2nd one actually.)
uade123: sending message 16 (token pass)
uadecore: sending message 25 (reply: data)
uadecore: sending message 25 (replay: data)
...
```

Based on that log and previous explanation of messages, you should be able to write an independent frontend by doing little reverse-engineering into uade headers. More specifically, you need following information:

- Message type numbers (see src/include/uadeipc.h)
- Filter setting codes (see src/include/amigafilter.h)
- something else you will find out..

Figure **??** shows play loop interaction from client perspective. That is, song initialization has already happened which means that uadecore configuration, score, player and module names and other options have been sent already.

5 Portability considerations

Language requirements:

- Limited C99 compiler that has:
  - Anonymous initializers for structures: struct foo bar = (struct foo) {.x = y};
  - Portable integer types from stdint.h.

Architecture specific parts that need to be implemented:

- User-interface (frontend)
- IPC between a frontend and the emulator may have to implemented. The unixipc.c module implements the IPC for UNIX systems. The interface for IPC is generic and an implementation must follow ipcsupport.h headers. However, the interface can be changed if it causes too much problems for some system.

Other issues:

- File modes of C fopen() calls. Windows systems have binary and non-binary modes which is different to UNIX systems. UNIX systems only have the binary mode. Some fopen() calls may open files without the “b” flag which means non-binary mode for Windows. As a result there may be data corrupt when reading binary files such as eagleplayer plugins. Please check that all fopen calls have the “b” flag set.

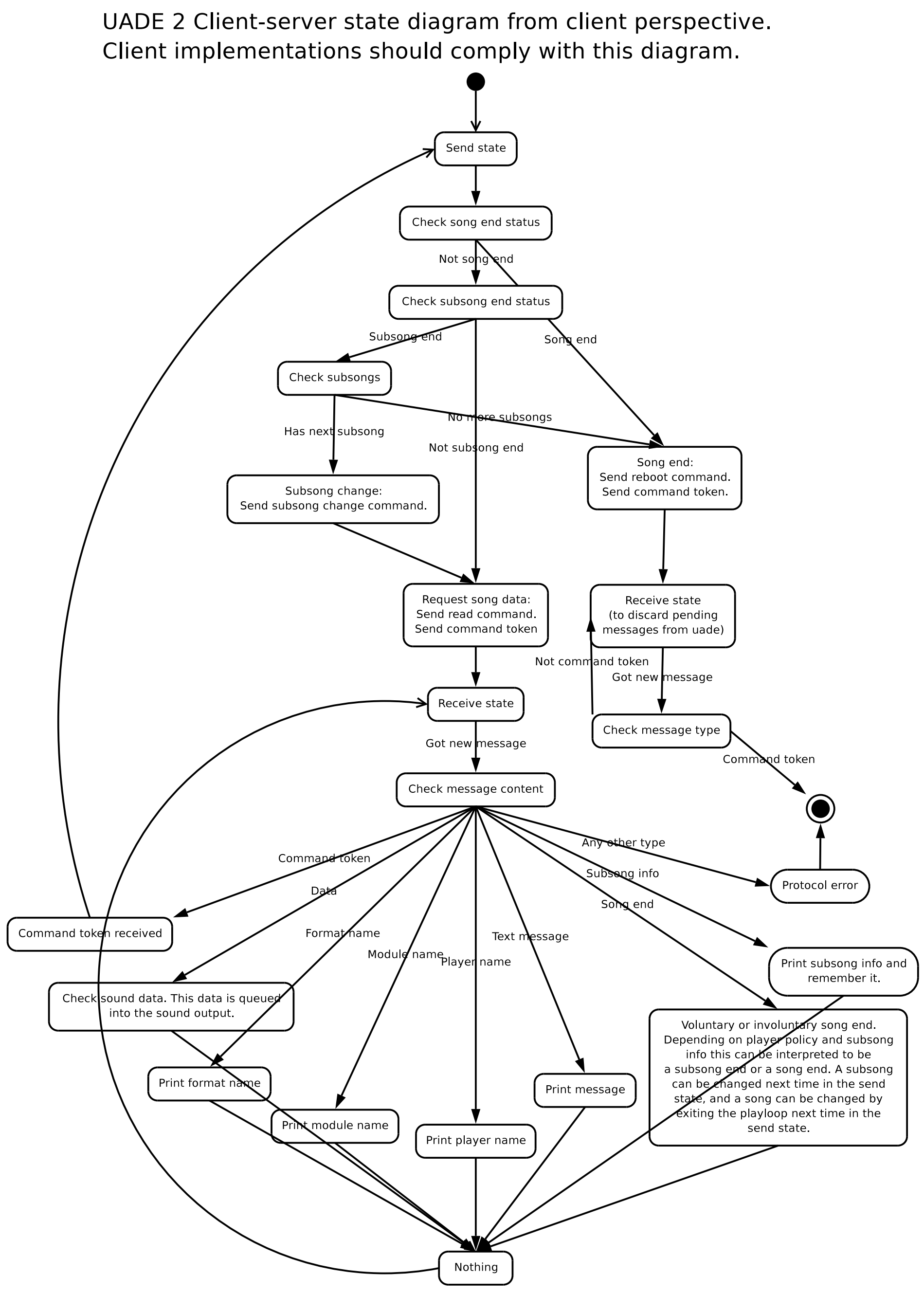

Figure 2: Play loop interaction from client (frontend) perspective