EMame

a MAME port to EPOC ER5 and ER6 (Quartz)

Table of Contents

1. Introduction

MAME stands for Multiple Arcade Machine Emulator. When used in conjunction with a game's data files (ROMs), MAME will more or less faithfully reproduce that game on a PC. MAME can currently emulate over 1500 classic arcade video games from the '70s and '80s. The ROM images that MAME utilises are "dumped" from arcade games' original circuit-board ROM chips. MAME becomes the "hardware" for the games, taking the place of their original CPUs and support chips. Therefore, these games are not simulations, but the actual, original games that appeared in arcades. MAME has been ported to numerous platforms including Dos, Windows, UN*X, MacOS and BeOs. For more information visit the official MAME web site at http://www.mame.net

ER5 is the version of EPOC mainly used in Psion devices such as the Psion Revo, Series 5(mx), Series 7 and Netbook.

ER6 is the version of EPOC that will be used in the next generation of wireless devices such as PDAs, communicators and smart phones. ER6 is available in three different flavours called Pearl, Quartz and Crystal. All three versions have the exact same set of core functionality such as the kernel, graphics engine, file server and network software but their graphical user interface (GUI) is designed for different so called device families. Pearl is designed for 1/8 VGA devices, Quartz for 1/4 VGA devices and Crystal for 1/2 VGA or VGA devices.

EMame is my port of MAME to EPOC. You may download EMame free of charge at http://www.yipton.demon.co.uk. You may also download the source code for EMame, which is available under the GNU General Public License (GPL).

This paper describes how MAME was ported first to ER5 and then from ER5 to ER6 (Quartz).

2. MAME architecture

Shown below is the basic MAME architecture, see Figure 1.

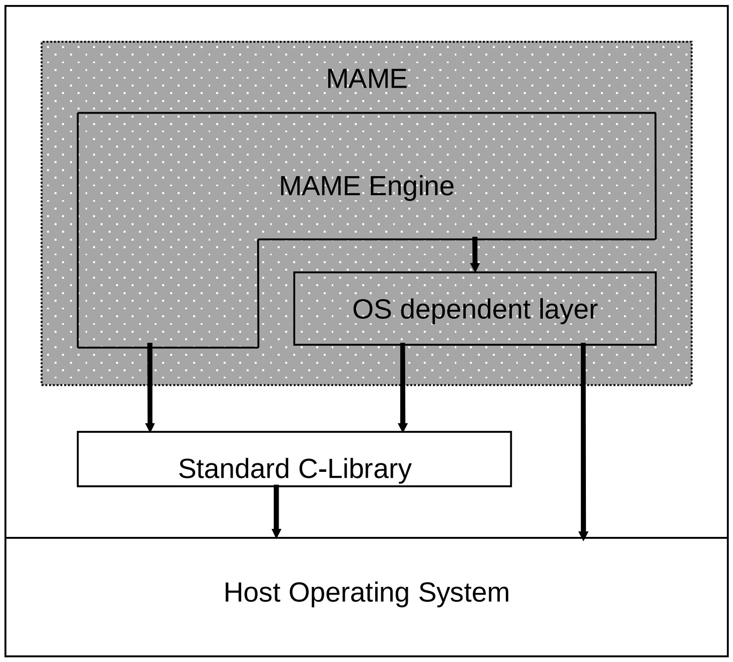

Figure 1: MAME Architecture.

MAME consists of two modules separated by a well defined API: the MAME engine and the Operating System Dependent (OSD) layer. These are discussed below.

MAME engine: The engine takes care of the emulation of all the different hardware sub systems. It emulates various general purpose CPUs (around 30), sound CPUs and video hardware. When running a game, the engine sets up the execution environment by starting the emulation of all the hardware needed by the game. This module supports many more features, however, a more in depth analysis is beyond the scope of this paper.

The MAME engine is designed with portability in mind. All functions that are dependent on the host operating system are encapsulated in the OSD layer and all the modules within the MAME engine access these functions via this layer. When porting MAME to a new platform there is no need to change the MAME engine, implementing a new OSD layer is all it takes. There are always reasons though to make changes to the engine, performance being one of them. This will go however to the expense of not being able to easily upgrade to a new version of MAME.
The MAME engine is written entirely in C and the assumption made in the MAME engine is that the host OS supports the standard C library.

OSD layer: This layer can be functionally split up in:

- **Video functions**: MAME emulates the video hardware by maintaining a bitmap. When the game manipulates the video hardware by setting pixels, drawing lines and filling shapes this result in MAME updating the bitmap. There are three basic functions the OSD layer has to implement:
  1) Create a bitmap;
  2) Map colours;
  3) Draw the bitmap on screen.

  When the game is started, as part of setting up the execution environment, MAME requests the OSD layer to create a bitmap, referred to as the *game bitmap*, and to fill in a data structure that allows MAME to access the memory in the bitmap directly. The size of this bitmap equals to the game's video hardware's screen size. For example, the game PACMAN runs on video hardware which has a screen size of 224 x 288 pixels.
  Each game has its unique set of colours, each represented by a 32-bit RGB value. Dependent on the number of colours, the MAME engine requests the OSD layer to allocate an 8 bits per pixel or a 16 bits per pixel bitmap. Next, the MAME engine requests the OSD layer to map each RGB colour value needed by the game to an OS dependent logical pen. Whenever the MAME engine sets a pixel in the game bitmap, it will do so with these logical pen value. During the game, with the emulated video hardware refresh frequency (typically 50 or 60 Hz), the MAME engine requests the OSD layer to draw the game bitmap on screen.

- **User input functions**: MAME supports both keyboards and joysticks. At any given point in time when a game is running, a particular subset of keys and joystick movements are valid. MAME will frequently poll the OSD layer on whether any of these key events or joystick movements occurred. It's up to the OSD to retrieve key events and joystick events from the host OS.

- **Audio functions**: For each game, the MAME engine converts the sound format supported by the game's sound hardware into 16-bit linear PCM frames. The OSD is responsible for playing these sound frames when appropriate during the game. If the host OS is not capable of playing 16-bit linear PCM frames, the OSD layer needs to convert them into a sound format that is supported by the host OS. It's up to the OSD layer to keep the audio synchronised with the video.

- **File I/O functions**: The OSD layer is responsible for opening, closing, reading from and writing to all sorts of files needed by the MAME engine (ROM image files, configuration files, high score files etc.). The API for file I/O is quite similar to the standard C-library's `fopen()`, `fread()`, `fwrite()` and family of calls.
  The OSD layer may optionally implement support for transparent zip files. Each game has typically

more than one ROM image file and they are conveniently stored together in a zip file. For example, all the ROM images for the game PACMAN are stored in PACMAN.zip. If the OSD layer doesn't implement support for transparent zip files, the user needs to extract all the individual ROM image files.

Apart from the above functions, the OSD layer is also responsible for controlling the emulation speed. The best measure is the video hardware's refresh frequency. For example, if the refresh rate is 50 Hz, the OSD should draw the game bitmap precisely 50 times per second. Depending on how powerful the hardware is (that is, the hardware the host OS is running on), the OSD layer can or cannot cope with this speed. If it's faster it needs to slow the game down by introducing small periods where it sleeps. If it's slower it needs to occasionally skip drawing the game bitmap.

3. EMame: porting MAME to ER5

The first section describes the EMame architecture. In section 3.2 I will discuss an alternative solution for handling global modifiable data in EPOC. Section 3.3 describes the implementation of the OSD layer and, finally, section 3.4 deals with other implementation challenges.

3.1 Architecture

EMame's architecture diagram is show in Figure 2. Apart from depicting EMame's architecture, this diagram also shows how EMame software components are physically partitioned in terms of binary image files (executable files, dlls etc.) and EPOC processes. The reason for the latter is that EMame implements a different solution for EPOC's notorious problem of not allowing global modifiable data in GUI applications. Because of that restriction, porting source code to EPOC usually requires heavy source code changes.

I will present EMame's solution with respect to global modifiable data first. After that I will discuss the four OSD modules audio, video, key input and file I/O in more detail.

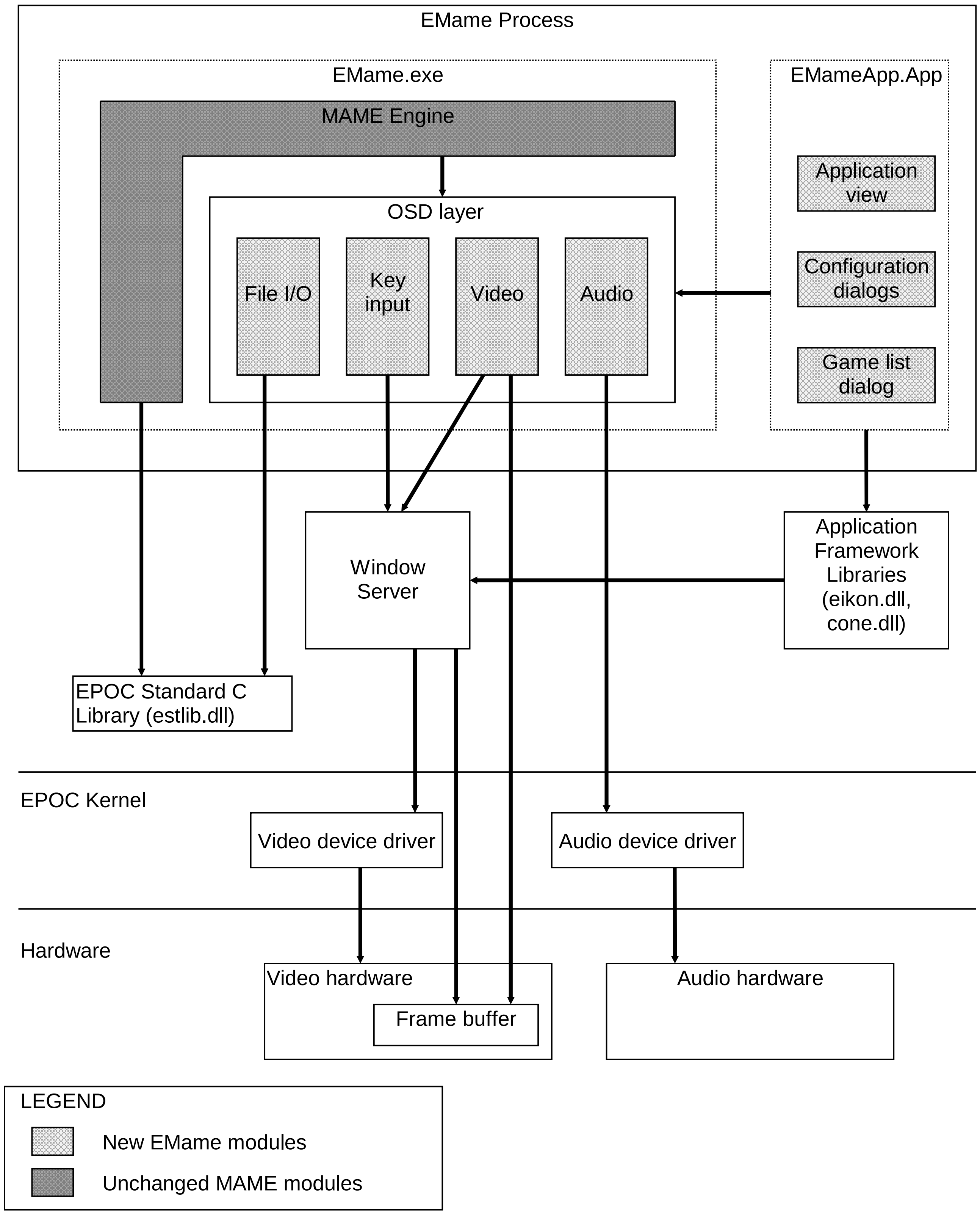

Figure 2: EMame Architecture

3.2 Solution for global modifiable data

When looking at Figure 2 again, it's worth observing the following:

- The ported MAME engine plus the EPOC OSD layer are implemented in the EPOC executable EMame.exe;
- EMame's front end (configuration dialogs etc.) is implemented in the EPOC Application EMameApp.app;
- Both EMame.exe and EMameApp.app are mapped in the same EPOC process.

To understand why EMame is designed like this, I'll summarise the issues with respect to global modifiable data in EPOC first.

Global modifiable data in EPOC revisited

A common misunderstanding is to think that global modifiable data in EPOC is not supported at all. To see what is and isn't feasible, it is best to look at the target types that ER5 supports, these are: App, dll, Exe, exedll, fsy, ldd and pdd The latter three are used for respectively file system dlls, logical and physical device driver dlls and are irrelevant for the discussion in this paper. The remaining types are discussed below.

Dll: A dynamic link library in the most broad sense of the word. In EPOC, a dll has the restriction that is cannot have global modifiable data. This restriction is regardless of whether the dll is loaded from ROM or RAM. It's the EPOC kernel who enforces this restriction by marking the data section of the dll as read-only. Any attempt to write to this data section results in an access violation and the kernel will terminate the owning process.

App: Used for GUI applications using EIKON, the EPOC GUI application framework. An App is a dll that exports a single method called: `CApaApplication* NewApplication()`.On target, an App runs as a separate process, in WINS it runs as a thread in the single process emulator. The following method can be used to start an App:

```
        TThreadId EikDll::StartAppL(const CApaCommandLine& aCommandLine)
```

Since an App is still a special type of dll, it is subject to the same restrictions as a normal dll, so an App cannot have global modifiable data either. If you want to port source code to EPOC and implement it as an App you must get rid of all non-const global data.

Exe: An Exe is an executable image with all external linkage resolved. On target, an Exe is started in a new process. Because the WINS emulator is a single process solution, running an Exe in WINS results in a different emulator environment; instead of the normal graphical Window Server, a special text-based console Server is started. Because of this WINS behaviour, targettype Exe is used for simple text console applications and is typically used for testing engine libraries. If the intention is to have an Exe for target, then the preferred targettype is Exedll. An Exe does not have any restriction on global data.

Exedll: This target is defined for convenience. When specified in a mmp file, the generated executable file for WINS builds is a dll, Exe otherwise. Exedlls are typically used by Servers.

To get rid of global modifiable data in a dll there are 2 basic approaches:

1. Move all global data into classes. In the API exported by the dll you enforce that clients create at least one object (for example, an *engine* object or *model* object). All the global data in the dll should now refer to data in such an engine or model object (or in objects owned by the engine or model object).
2. Use thread local storage (TLS). Move all global data into one class, instantiate a single object of that class and modify all source code referring to global data to refer to data in that object. This reduces the problem of having global data to having a single global pointer: the pointer to the object holding all "global" data. TLS allows you to do precisely that. The kernel associates with each dll a 32-bit

value and this value is stored per thread. `TAny* Dll::Tls()` and `void Dll::SetTls(TAny*)` are used to get and set this pointer respectively.

There are a few variations possible on the above two solutions, but they all have in common that you have to make changes to the source code.

EMame's solution for global modifiable data

One thing was clear to me from day one: neither of the two approaches in removing global data in MAME are acceptable. The MAME source code consists of over 1000 C files most of which declare global or refer to global data in other source files. Making changes to 1000 source files to remove global data is not only a massive undertaking, it is also extremely error prone and prevents me from upgrading to newer versions of MAME.

The obvious solution was to see if I could implement EMame as an Exe because that wouldn't impose any restrictions on global data. The biggest issue with such an approach is that it is unclear if EIKON can be used in an Exe. In EMame's case the questions were:
1. Is EIKON needed in the Exe to implement video and key input in the OSD layer?;
2. Even if EIKON isn't needed for the above, is it possible to use EIKON to implement the front end (menus, dialogs etc.)?

The answer to 1 is "no", EIKON is not needed to implement the OSD layer (see next section). The answer to 2 is, "yes", you can use EIKON in an Exe.

EIKON and application dlls (Apps) seem almost inseparable. To use EIKON there seems no way getting round implementing an App because at some point in starting the EIKON framework it insists on loading an application dll[1]. As mentioned before, to start an EIKON App you could call:

```
        TThreadId EikDll::StartAppL(const CApaCommandLine& aCommandLine)
```

and the command line object you pass in should refer to an application dll.
The downside of using `EikDll::StartAppL` is that it runs the application in a separate process. This means that data nor functions can be shared between the calling process and the process in which the application is run.

In EMame's case, the front end needs data from the MAME engine such as the list of all supported games. Also, at some point the front end must instruct the MAME engine to start a particular game. Running the front end in a separate process is not quite helpful.

It turns out the class `EikDll` has a few different ways to start an application, see Figure 3.

```
class EikDll
        {
public:
        IMPORT_C static void StartDocL(const TDesC& aDocName);
        IMPORT_C static TThreadId StartExeL(const TDesC& aName);
        IMPORT_C static TThreadId StartAppL(const CApaCommandLine& aCommandLine);
        IMPORT_C static void RunAppInsideThread(CApaCommandLine* aCommandLine);
        };
```

Figure 3: Different ways of starting Apps and Exes.

`EikDll::RunAppInsideThread` is precisely what EMame needs. Just as the name suggests, this method will start the application specified in the command line object inside the current thread. The method only returns after the application has been terminated, for example by selecting close from the menu.

Sharing data and functions between the Exe and the App

[1] There are ways of using EIKON without an App dll, however, these require some work arounds.

Because the Exe and the App can now run in the same process, the App can share data and functions defined in the Exe. However, since there is no link dependency between the App and the Exe, the App doesn't know how to address them. One way or the other the App and the Exe need to exchange information in order to share data and functions. There is more than one way of doing this, the solution I choose was to use shared memory to exchange a single object pointer between the Exe and the App. The Exe defines an object holding pointers to data structures and function that the App can use. It then creates a piece of shared memory by means of a named `RChunk`. The size of this RChunk is 4 bytes, enough to store the object's pointer. When the App is started, it opens the chunk specifying the name used by the Exe when creating it. The App then reads the object's pointer and from that moment on it can access the data structures and functions defined by that object.

Multi-threaded solution

Initially I thought that I could get away with a single threaded solution. The user starts EMame.exe which in its turn starts the EIKON application EMameApp.app within its thread context. Whenever the user selects a game in the application, EMameApp.app calls `run_game`. `run_game` in EMame.exe via the export mechanism as described in the previous section. Although not the most elegant solution (`run_game` will not return until the game finishes therefor blocking EMameApp.app's UI) I at least thought this would work. Unfortunately, this doesn't work out of the box.

The reason this does not work as is, is due to the active objects (AOs) and active scheduler (AS) frame work. EMameApp.app being a normal EIKON application uses this framework as well. In a nutshell, this framework sits on top of EPOC's lower level client-server framework. At any given point in time, the current thread has multiple asynchronous request outstanding at one or more servers. The AS is sleeping until at least one of these requests has been completed. When that happens, the AS wakes up, searches for the highest priority AO which indicates its request has been completed and calls `RunL()` on it.

The problem in EMame is that the OSD layer in EMame.exe is managing its own client-server requests with the Window Server (see section 3.3.1). These requests are associated with the same thread for which the AS in EMameApp.app is handling the requests. When a game finishes (remember that the game is run by calling `run_game` which does not return until the games finishes), the AS in the App will detect there are many requests that have been completed but it can't find AOs which indicate their request has been completed. The AS therefor concludes this is a *stray event* and *panics* the current thread.

Although there is a way the OSD can co-operate with the AS in handling the completion of requests, thereby allowing for a single threaded solution, I decided to implement a 2 threaded solution. Each time the App starts a game, it will create a new thread and have `run_game` called in the context of that thread. This way, all of the client-server interaction in the OSD layer won't interfere with the AS in the App as they occur in a different thread.

Summary

It is easy to get lost in the discussion about Exes, Apps, processes and threads, so I'll give a summary below:

- The ported MAME source code, together with the OSD layer are implemented in the EPOC Exe: EMame.exe. No changes are needed for global modifiable data in the MAME source code because an Exe is allowed to have global modifiable data;
- EMame's front end is implemented as a normal EIKON App called EMameApp.app. It has menus, configuration dialogs and a dialog displaying a list of supported games;
- There are no link dependencies between EMame.exe and EMameApp.app;
- The user starts EMame by starting EMame.exe from the EPOC shell;
- EPOC creates a new process and starts EMame.exe in the context of that process;

- EMame.exe starts EMameApp.app within its own thread context;
- Prior to doing that, EMame.exe creates a `RChunk` and stores a pointer to a global object in it;
- When EMameApp.app starts, it opens the `RChunk` and retrieves the pointer to the global object. Using that object it can access relevant data defined in EMame.exe, such as the list of supported games;
- Each time the user starts a game, EMameApp.app creates a new thread and `run_game()` is called in the context of that thread.

3.3 Implementation of the OSD layer

As described in the MAME section, the OSD layer is responsible for implementing support for video, user input, audio and file I/O. These are discussed in the following subsections.

3.3.1 Video

For both video and user input, EMame needs the functionality provided by the Window Server (WServ). In EIKON applications, most Window Server interaction is hidden in lower layer classes. That is not to say that you cannot access WServ functions directly; as a matter of fact it's quite easy to do this. Figure 4 shows the method that sets up the Window Server session; this method also creates a window group (`RWindowGroup`) and a full screen window (`RWindow`).

```
void CMameWindow::ConstructL()
{
	User::LeaveIfError(iWsSession.Connect());
	iWsScreen=new(ELeave) CWsScreenDevice(iWsSession);
	User::LeaveIfError(iWsScreen->Construct());
	User::LeaveIfError(iWsScreen->CreateContext(iWindowGc));

	iWsWindowGroup=RWindowGroup(iWsSession);
	User::LeaveIfError(iWsWindowGroup.Construct((TUint32)this));
	iWsWindowGroup.SetOrdinalPosition(0);

	iWsWindow=RWindow(iWsSession);
	User::LeaveIfError(iWsWindow.Construct(iWsWindowGroup, (TUint32)this));
	iWsWindow.Activate();
	iWsWindow.SetSize(iWsScreen->SizeInPixels());
	iWsWindow.SetVisible(ETrue);

	StartWServEvents();
}
```

Figure 4: Setting up a Window Server session.

Using the `RWindow` object, the OSD layer can now draw bitmaps directly to the screen using the code shown in Figure 5

```


/*
 * CMameBitmap encapsulates an EPOC CFbsBitmap and provides the information
 * needed by the MAME engine to directly access the memory in the bitmap.
 */
void CMameWindow::WsBlitBitmap(CMameBitmap* aMameBitmap)
{
	iWindowGc->Activate(iWsWindow);

	TRect rect = TRect(iWsWindow.Size());
	iWsWindow.Invalidate(rect);
	iWsWindow.BeginRedraw(rect);

	if (aMameBitmap)
		iWindowGc->BitBlt(iMameScreen->Position(), aMameBitmap->FbsBitmap());

	iWsWindow.EndRedraw();
	iWindowGc->Deactivate();
	iWsSession.Flush();
}
```

Figure 5: Drawing a bitmap.

As is shown in Figure 2, the OSD layer directly accesses the video hardware's frame buffer. It can only do so, if the video hardware actually supports a framer buffer. Failing that, EMame will use the above CMameWindow::WsBlitBitmap() to draw the image to screen. For a discussion on frame buffer access see section 3.4.

In a normal EIKON application, you would implement both client-initiated redrawing (like the method in Figure 5) and server-initiated redrawing. Since the MAME engine is instructing the OSD layer to draw the game bitmap to screen with a typical frequency of 50 or 60 Hz, server-initiated redrawing is not really needed. The only exception is when the user pauses the game by pressing 'p'. At that point the OSD layer will handle server-initiated redrawing by using the method in Figure 5. Also, when the OSD layer detects that its window group looses the focus it will simulate a game pause because loosing the focus implies that another application or dialog is being put in front of the game. If EMame would not pause the game it will still obscure the application or dialog because the OSD layer is updating the frame buffer directly.

3.3.2 User input

The MAME engine supports various types of user input such as keyboards and joysticks. EMame's OSD layer only supports the keyboard.

To receive key events from the Window Server, the OSD layer needs to issue the request `EventReady()` to the Window Server to receive Window Server events, see Figure 6.

```
void CMameWindow::StartWServEvents()
{
	iWsEventStatus = KRequestPending;
	iWsSession.EventReady(&iWsEventStatus);		// get Window Server events

	iRedrawEventStatus = KRequestPending;
	iWsSession.RedrawReady(&iRedrawEventStatus);	// get server-initiated redraw events
}
```

Figure 6: Requesting Window Server events.

The Window Server completes this request each time it wants to pass an event to a client. The Window Server maintains an event queue for each client to ensure that events don't get lost if the client cannot process them fast enough.

The MAME engine's design is such that the OSD layer is being polled for key events during a game. The OSD layer simply checks whether the request it had issued for events has been completed, if so it will handle the event to see if it's a key event and will reissue the request, see Figure 7.

```
void CMameWindow::PollWServEvents()
{
	if (iWsEventStatus != KRequestPending)
	{
		iWsSession.GetEvent(iWsEvent);
		HandleWsEvent(iWsEvent);
		iWsEventStatus = KRequestPending;
		iWsSession.EventReady(&iWsEventStatus);
	}
}
```

Figure 7: Receiving Window Server events.

3.3.3 Audio

The easiest (and probably the fastest) way to handle audio in ER5 is to interface directly with the sound driver. Figure 8 shows how to open the sound driver and configure it for playing audio frames.

```
void CMameAudio::ConstructL()
{
      User::LeaveIfError(iDevSound.Open());          // iDevSound is an RDevSound object

      TSoundCapsV01       soundCaps;
      TPckg<TSoundCapsV01>       caps(soundCaps);

      iDevSound.Caps(caps);

      TSoundConfigV01            soundConfig;
      TPckg<TSoundConfigV01>     sc(soundConfig);

      iDevSound.Config(sc);
      soundConfig.iAlawBufferSize = KAudioBufferSize;
      soundConfig.iVolume = EVolumeMedium;
      User::LeaveIfError(iDevSound.SetConfig(sc));
      User::LeaveIfError(iDevSound.PreparePlayAlawBuffer());
}
```

Figure 8: Opening and configuring the sound driver.

As mentioned before, the MAME engine passes 16 bit linear PCM frames to the OSD layer. The audio format in ER5 is *Alaw* and therefor the OSD layer needs to converts the audio samples into this format. I re-used the actual audio conversion routines from a software package called *spAudio - Audio I/O Library*.

After the conversion is done, the audio samples can be played as is shown is Figure 9.

```
void CMameAudio::SoundUpdate()
{
      if (iFrameCount > 10)
      {
            TInt   noOfSamples = iSamplesPerFrame * iFrameCount;
            const TPtrC8 ptr((const TUint8*)iAlawSoundBuffer, noOfSamples);
            if (iStatus == KRequestPending)
            {
                  iDevSound.PlayCancel();
            }
            iStatus = KRequestPending;
            iDevSound.PlayAlawData(iStatus, ptr);
            iFrameCount = 1;
      }
}
```

Figure 9: Playing audio samples.

The OSD layer is requested to play a few samples (`iSamplesPerFrame` to be precise) with each video frame. Since games typically run at 50 or 60 frames per seconds, it would mean 50 or 60 calls to the sound driver per second. Apart from poor performance, the sound driver doesn't seem to be capable of handling so few samples because it doesn't produce any sound at all. A common solution is to buffer up samples for a couple of video frames (10 in the above) and have them played all at once.

3.3.4 File I/O

Implementing the file I/O functions is strait forward as almost all of them map directly onto file I/O functions (`fopen()`, `fread()`, `fwrite()` etc) from the standard C library.
EMame supports transparent zip files as well. The MAME team recommends using the popular compression library *zlib* for zip file support. In EMame, the zlib library is ported over as part of the MAME engine to avoid the usual problem with global data.

3.4 Other implementation challenges

Apart from the biggest challenge on how to port MAME without making changes to 1000 C files because of global modifiable data restrictions, there were a few other challenges; these are described below.

Floating point arithmetic

Although ER5 fully supports floating point arithmetic, it only does so in its own EUSER C++ class library. The EPOC tool chain does not support implicit casting operations for floating point arithmetic in C source code. For example, the code snippet below will not work in EPOC.

```
int    pi = (int) 3.1415927f;
```

It will compile just fine, but you'll get an error during linking saying: unresolved symbol `fixunsdfsi`. In order to cast a float to an integer the compiler generates a call to the helper function `fixunsdfsi`, which implements the actual conversion from floating point to integer. In one of the articles in the Symbian ER5 Knowledge database the suggested solution is to replace the above code snippet with relevant conversion methods from class `Math`.

Unfortunately, the MAME engine uses floating point quite a lot, so changing them means a lot of work. The solution suggested by Symbian is by no means the only solution. For WINS builds you could link to `msvcrt.lib` to get these helper functions. For target builds you need to link against a library that comes as part of the GNU tool chain. The problem is that the latter library is not shipped with the ER5 SDK.

I've solved this issue in EMame without any changes to the MAME engine source code. For WINS I have included the `msvcrt.lib` in the mmp file and for target I have included the helper functions of the GCC compiler directly into my EMame source code. The latter requires downloading the GCC compiler source code and configuring it for an ARM platform.

In ER6 the above mentioned gcc library is included in the SDK.

EMame tool chain

Setting up a decent tool chain for EMame wasn't entirely trivial. MAME uses traditional makefiles, whereas EPOC's starting point is mmp files (from which makefiles are generated). Compiling MAME on say Windows or UN*X results in an executable file which is many megabytes in size because it includes support for almost all built-in games. Supporting all games in EMame is not realistic because most EPOC devices are not powerful enough to emulate the hardware needed by most games. Since EMame will only support a (small) subset of these games, I wanted to have the smallest possible executable size by only including the individual modules in the MAME engine that are needed to run the supported games.

What I ended up doing was compiling all of MAME source code in one huge static library called mameengine.lib. This library is around 30MB for WINS builds. The OSD layer implemented in EMame.exe holds a table of the supported games and supported hardware emulation cores. By linking EMame.exe against mameengine.lib I could rely on the linker to get the smallest sized executable possible.

The bad news is that static libraries are not supported by the ER5 tool chain; the good news is that you can make it support static libraries. Discussing how this is done is beyond the scope of this paper. Interested readers are referred to the building instructions of the EMame source code.

ER6 has full support for static libraries, so the work around I used for ER5 is not needed in ER6.

Video performance: using frame buffer access

Initially EMame used the EPOC Window Server to draw the game bitmap on screen. Early tests showed that playing PACMAN resulted in around 10-12 frames per second on a Series 5mx. Given that PACMAN needs to run at 60 frames per second this performance was not satisfactory.

I dramatically improved performance (by a factor 2 or 3) when bypassing the Window Server and directly accessing the video hardware's framer buffer. In my opinion, this performance advantage heavily outweighs the disadvantage of having to implement things like scaling and converting the game bitmap from 8 bits-per-pixel (bpp) to say 4 bpp. Using direct frame buffer access, PACMAN now runs at 25-30 frames per second on a Series 5mx, which makes it even enjoyable to play.

Getting the frame buffer's address is trivial. The method below fills in a `TScreenInfoV01`, see Figure 10.

```
class TScreenInfoV01
       {
public:
       TBool iWindowHandleValid;
       TAny *iWindowHandle;
       TBool iScreenAddressValid;
       TAny *iScreenAddress;
       TSize iScreenSize;
       };


       …
       UserSvr::ScreenInfo(…);
       …
```

Figure 10: Getting the frame buffer address.

For target builds, the `iScreenAddress` points to the frame buffer and `iWindowHandle` has no meaning. This is the opposite for WINS builds in which the `iWindowHandle` refers to the WIN32 window handle for the emulator window and `iScreenAddress` has no meaning.

Shown below is the routine that copies an 8bpp game bitmap into a 4 bpp frame buffer, see Figure 11.

```
void CMameScreen::DirectDrawBitmap8bppTo4bpp(CMameBitmap* aMameBitmap)
{
       const struct osd_bitmap*   osdBitmap = aMameBitmap->OsdVisibleBitmap();

       TUint8* frameBufferLine = iBitmapStartAddress;
       TUint8* bitmapLine;

       TInt   height = (osdBitmap->height > iScreenSize.iHeight) ? iScreenSize.iHeight :
osdBitmap->height;
       TInt   width = (osdBitmap->width > iScreenSize.iWidth) ? iScreenSize.iWidth : osdBitmap-
>width;
       for(TInt h=0 ; h<height ; h++)
       {
              TUint8* frameBuf = frameBufferLine;
              bitmapLine = osdBitmap->line[h];
              for (TInt w=0 ; w<width ; w+=2)
              {
                     *frameBuf++ = (bitmapLine[1] << 4) | bitmapLine[0];
                     bitmapLine+=2;
              }
              frameBufferLine += iBytesPerScanLine;
       }
}
```

Figure 11: Drawing an 8 bpp bitmap into a 4bpp frame buffer.

To test routines like the one above EMame emulates the frame buffer for WINS builds. In WINS it creates a bitmap with the same dimension and colour depth as reported by `UserSvr::ScreenInfo` and sets the frame buffer address to the bitmap's memory. When drawing the game bitmap the EMame routines that copy the bitmap into the frame buffer are actually copying it into the frame buffer bitmap. Afterwards, the frame buffer bitmap is drawn to screen using normal WServ functionality. To test different hardware configurations, you should configure the EPOC emulator with a screen size and colour depth you want to test. See the SDK documentation on how to do that. Alternatively, there are a handful of scripts in the EMame source distribution (`\mame\src\epoc\group\wins`) to set the configuration to a target platform.

4. Porting EMame from ER5 to ER6 (Quartz)

Porting EMame from ER5 to ER6 was almost trivial. I had EMame running in the ER6 WINS emulator within half a day. Note that this only includes the time it took to get it to run under the emulator, there are more changes needed to run it on ER6 target hardware. Examples of the latter are:

- Video: It can be expected that upcoming Quartz devices have better video hardware than the current devices that run ER5. Since EMame accesses the video hardware's frame buffer directly it needs updating to support 12, 16, 24 and 32 bpp.
- Key handling: Quartz devices don't have a keyboard but just a handful of navigation keys. EMame needs to be updated to just use those navigation keys.

The ER5 version of EMame fully supports audio. Unfortunately, at the time doing the port from ER5 to ER6 port it did not. Therefor, EMame support for audio in ER6 is not covered in this chapter.

Listed below are the changes I had to make for ER6.

Mmp files

The syntax in mmp files has slightly changed. If you have source code in multiple directories, then in the ER5 mmp file you usually specify a single `PROJECT` statement and for each sub directory a `SUBPROJECT` statement. In ER6 the `PROJECT` and `SUBPROJECT` statements are replaced by a single `SOURCEPATH` for each sub directory.

Another change in the mmp file is that `UNICODEUID` is no longer needed in ER6. Also, the UID for an non-Unicode App is different from an Unicode App. The first is 0x1000006c (which should therefor be used in ER5 mmp files), the latter is 0x100039CE (which should therefor be used in ER6 mmp files).

Unicode

Both ER5 and ER6 can handle Unicode and non-Unicode strings without any problem, so manipulations on `TDesC8` and `TDesC16` work fine in both versions. The subtle difference is that a `TDesC` on ER5 equals to a `TDesC8`, on ER6 it's a `TDesC16`. Since almost all EPOC API calls use the neutral `TDesC` rather than an explicit `TDesC8` or `TDesC16` you have to make some changes when manipulating normal C `char*` strings.

Since EMame's UI is rather small I prefer to build both the ER5 and ER6 builds from one source set. The approach I've taken for Unicode is shown below.

Original ER5 implementation to set a label in a dialog:

```
	TBuf<128>	text;

	CEikLabel*	label =(CEikLabel*)Control(ECtlGameInfoGameManufacturer);
	text.Format(_L("%s, %s"), game->year, game->manufacturer);  // 2 char* strings
	label->SetTextL(text);
```

New implementation to facilitate both ER5 and ER6 from the same source code:

```
	Tbuf8<128>	text8;		// 8 bit character string
```

```
        Tbuf<128>     text;                // TBuf8 for ER5, TBuf16 for ER6

        CEikLabel*    label =(CEikLabel*)Control(ECtlGameInfoGameManufacturer);
        text8.Format(_L8("%s, %s"), game->year, game->manufacturer); // 2 char* strings
        text.Copy(text8);
        label->SetTextL(text);
```

Unfortunately, `TDes16::Format()` only allows for string formatting via `TText16*` (%s) or `TDesC16` (%S) and, in similar vein, `TDes8::Format()` only allows for `TText8*` or `TDesC8`.
To mix `char*` strings defined in C source code in ER6 you'll have to copy the 8-bit string into a 16-bit string. Although this copy is not needed for ER5, I favour the single source set approach, and accept the small performance penalty. Note that these string conversions in EMame are only needed for non-time critical things such as constructing a simple dialog.

API name changes

In ER6 some method names have changed; some of them now include a trailing L and others have their trailing L removed. For example, the ER5 method `CEikDialog::SetCornerAndSizeL()` as been renamed to `CEikDialog::SetCornerAndSize()`. To use both in one source set you could replace the call to `SetCornerAndSize()` by the macro `SET_CORNER_AND_SIZE`. The macro is defined to expand to the appropriate method for ER5 and ER6 builds.

EIKON changes

The following changes were needed for EIKON:
- In ER5 you had to include all sorts of different .rh files in your application's resource file. In ER6 you only need to include a single eikon.rh.
- In ER5 you link against a single eikon.lib, in ER6 you link against a couple of smaller sized EIKON libraries. For example, in ER6 EMameApp.app links against: eikcore.lib, eikdlg.lib, eikcoctl.lib and eikctl.lib.

Since EMame's UI needs are very modest I want to build everything from one common resource file. I have split up the resource file in individual platform dependent files and a single platform independent file. The subtle changes that are needed in the platform independent resource file to facilitate different screen size can easily be solved with some strait forward pre-processor macros. Remember that the EPOC tool chain uses the pre-processor in resource compilation, so all the nifty tricks that you can do with pre-processor macro's can be applied to resource files.

5. Future work

The current version of EMame is far from complete. Listed below are things on my todo list::

- Improve the mechanism to run a game at the correct speed, by either slowing it down or skipping frames.
- Audio support in ER5 needs to be improved so that it is synchronised with video;
- Add support for audio in the ER6 version;
- Run EMame on actual ER6 hardware;
- Port EMame to the other ER6 flavours: Pearl and Crystal;
- An ongoing activity will be to support more games ☺.

6. Conclusions

EMame successfully demonstrates that porting legacy C source code to EPOC does not necessarily require blood, sweat and tears. The Exe-and-App-in-a-single-process solution shows that ported C code can happily coexist with EPOC C++ EIKON source code without the need of modifying any of the C source code. Of course, the fact that MAME was designed with portability in mind made things considerably more easy.

For performance reasons, EMame directly accesses the video hardware's framer buffer. This goes to the expense of having to implement support for frame buffers of different colour depth and other functionality that was otherwise provided by the Window Server, such as scaling.

Although ER6 supports many more features than ER5, the changes between ER5 and ER6 were minimal on the source code level. EMame's front end is quite small and with a few pre-processor tricks both the ER5 and ER6 version could be built from one source set.

Acknowledgements and credits
I would like to thank:

Birgit Arkesteijn and John McAleely (Symbian) for reviewing this paper;
Joe Branton for reviewing the paper and the many interesting discussions we had on EPOC;
Lars Persson for helping out with audio support and sharing my interest in porting games to EPOC.

Full credit to Nicola Salmoria and the MAME team for developing MAME.

About the author
Peter van Sebille is a software engineer working for Ericsson on EPOC based smart phones. In his free time he likes to indulge himself with hobby programming projects, of which EMame is one. He has a passion for all things low-level and one of these days he will just take that Linux kernel and port it to a new platform. He can be contacted at peter@yipton.demon.co.uk.