Design Document for:

- MicroWindows Extension: A cross-platform GUI library for Win32 and Linux
- Using MicroWindows as device-independent high-level windows-drawing API
- Porting a device-independent GUI library from Linux edition (MicroWindows) to Win32
- MicroWindows embedded-hardware device simulator: for ease of software development and debugging on embedded systems.

Document No.1

"For the Wearable DVD project at HIT"

CONFIDENTIAL

Written by LiXizhi
Email: LiXizhi@zju.edu.cn

Version # 1.00

Wednesday, August 11, 2004

Version history:

2004-7-5	Author is involved in the project
2004-7-8	Project release 1.0 finished
2004-7-9	Documentation reviewed and finished

Summary

Software debuggers are not only essential for successful Soc architecture exploration and benchmarking, but also the key to late user application development on Soc systems. For the former part of software debugging, both hardware and software are involved to perform a certain debugging technique such as probing brought out by I/O pins and various scan technologies with software support on a host computer. However, when it comes to user application development based on a specific Soc design, there are usually no standard software debugging principles to follow. Usually, no further tools are created for this purpose, and one of the former hardware-software debugging techniques is used. However, user application focuses more on functionality and Graphic User Interface (GUI) development than on probing into the internal states of Soc circuit. Moreover, on-chip debugging is both expensive and time-consuming; a shorter edit-run-debug cycle is preferred. Sometimes, a cross-compiler is developed, so that user application developers can at least compile their code in a standard familiar host environment. Even so, real-time software debugging is not possible if without developing high-level software simulation kits dedicated to a certain Soc design. This article provides the details of a simulation kits design with video output and remote controller (keyboard) input. Any user applications based on this specific Soc can be developed, run and full-debugged all through software environment and in a familiar host IDE such as Visual Studio and Linux KDE. It may provide a useful guide for Soc software developers who are developing GUI applications based on a non-standard Soc system.

Content

1 Introduction

1.1 Background

In July 2004, I took up the project of Wearable DVD during my visit to Shenzhen. The original development kid of Wearable DVD is em85xx [1] , which in turn uses (1) another third party development kits called MicroWindows[2] for its GUI and (2) UcLinux[3] for its kernel protocols. Before I was involved, the whole hardware development system with a flash simulator was established and the Linux software packages can be compiled. However, there was no software support for application development on em85xx, i.e. no software simulator was present. Each time, new source code needs to be compiled on a Linux computer, cross-compiled to em85xx ucLinux core, downloaded to a flash hardware, and tested with the actual hardware. This development cycle was immediately recognized by me as unviable. A new software development environment should be developed on the first place before any custom applications could be built next. It is under this motivation, I embarked on this project.

A second motivation for me to carry on such a project is the observation of its potential to be used in the ParaEngine [4] . MicroWindows is a great tool for writing device independent graphic engine. It is written almost in clean C. I am planning to write my own graphic driver and use it in the 3D ParaEngine. A few months ago, Mitnick and I were designing 2D engine API that is similar to Windows Drawing API. The interface was designed by Mitnick; but due to other tasks, he did not get enough time to implement it. I therefore had to think of other ways to write a simple 2d engine. Currently the 2D engine in the ParaEngine has Bitmap and text only. There is no support for other drawing functions. However, it is critical to allow programmers to draw on a D3Dsurface, such as used in the map display of RPG games. Now that, I met with MicroWindows whose design goals match exactly that of a 2D engine.

1.2 Short Overview

The whole project can be divided into the following intensely intertwined sub-domains.

A. MicroWindows Extension: A cross-platform GUI library for Win32 and Linux
   a) Identifying the em85xx architecture.
   b) Porting a device-independent GUI library from Linux edition (MicroWindows) to Win32.
   c) Developing software interface for user-defined screen device driver.
   d) Developing software interface for user-defined keyboard device driver.
   e) Making the keyboard driver, somewhat compatible with the em85xx FIP driver module
B. MicroWindows embedded-hardware device simulator: integrated with the Microsoft Visual C++.Net IDE and Debugging environment.
   a) User interface design to accommodate a monitor and a key board simulator.
   b) Using another thread to run the simulated application program for em85xx. Multi-threading and data synchronization needs to be considered.
   c) Writing the Screen driver implementation code for the MicroWindows Extension.
   d) Writing the Keyboard driver implementation code for the MicroWindows Extension.
   e) Writing FIP remote control simulation code for em85xx.
   f) Rewriting GsSelect() event polling function for software simulator. It's a combo of the UNIX and DOS version. A simulated timeout is implemented in the polling mechanism.
C. Porting MicroWindows Sample applications and DVD Player program to the Simulator.
   a) Porting sample applications from MicroWindows demos. The following samples are ported.
      i. MyProgram.cpp: A very simple C++ application that shows a gif picture with MicroWindows APIs. IO operation and image display are illustrated.
      ii. NTetris.c: A slightly more complex C application that implements the famous Tetris game. Win GUI and Message handling (for keyboards) are demonstrated in the sample.
   b) DVD player: The original code of this application is provided by em85xx. Its event handling is based on its old polling mechanism with fip module; while its display interface is built upon a

thin-wrapper (OsdWindow) of MicroWindows API. This mixture of coding style makes it extremely inefficient to be ported to the simulator.

D. Testing and documentation
   a) Software-based tests performed on the software simulator (Windows NT edition)
   b) Documentation edited and reviewed.
   c) TODO: Performing tests with real hardware system: Integration into the hardware development cycle.
   d) TODO: User application development: HTML browser, E-BOOK viewer.

Please see Figure 1. for a comparison of the development cycle before and after the software simulator was introduced.

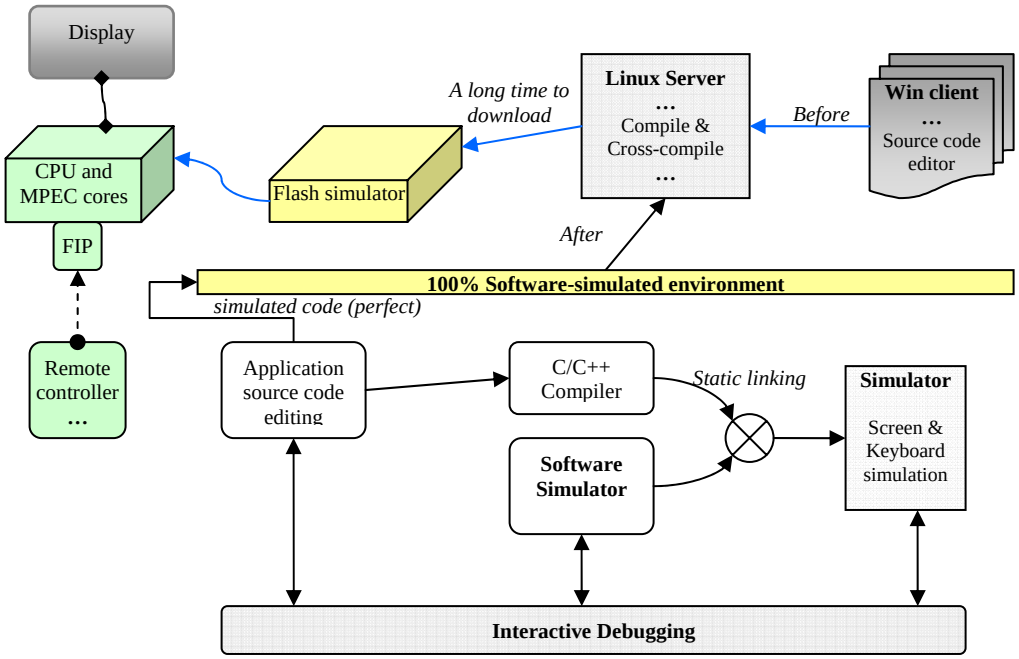

Figure 1. Development path before and after software simulation is introduced

We see that all hardware components are simulated with software debugging enabled (user-friendly and line-by-line tracing); while in the old path no source-code debugging is possible and there is significant delay along the path.

In the following section we will take a deeper look at how each component is implemented and used in the actual application. Section 2, 3, 4 deals with domain A, B, C respectively. To facilitate readers, I have quoted source code where necessary. In section 5, I summarized all three domains and streamlined the process of developing em85xx applications using the software simulator.

2 MicroWindows Extension

This is the most difficult part of the whole system. The ultimate goal is to write a win32 compatible MicroWindows library without changing any of its APIs from its Linux version. By doing so, any Linux source programs that conform to the pure C/C++ standards and that of those MicroWindows APIs would have been ported from embedded Linux system to Win32 without changing one line of code. MicroWindows is a third party operating system written in almost pure C language. I did say ALMOST, because some of its codes do use some UNIX specific system calls and structures, mostly those concerning timing and alarm functions. Fortunately there are few kinds of them, but their total number is large enough to cause a headache.

2.1 Identifying the em85xx Architecture

Before getting into the details of MicroWindows, we will first identify the current software architecture for the embedded application development environment. Figure 2. shows the software architecture of em85xx.

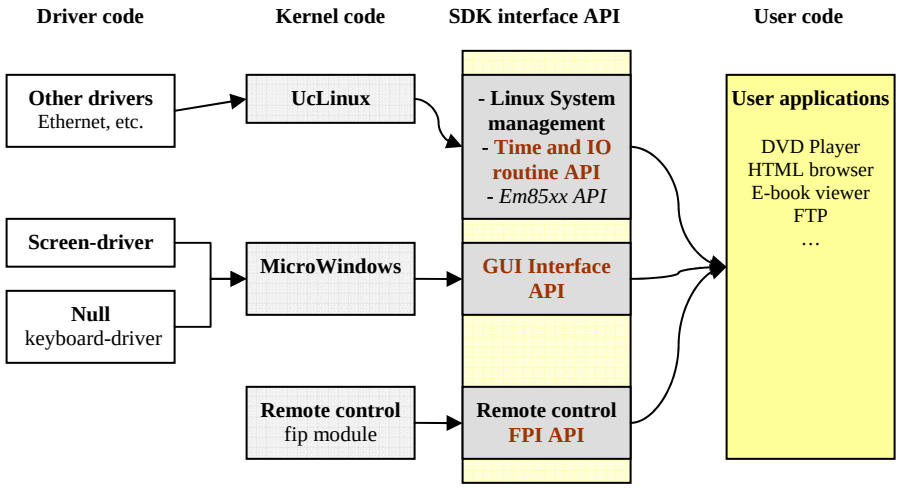

Figure 2. Software architecture of em85xx

We see that, as exposed to User application developers, there are three sets of APIs: Linux System functions provided by UcLinux, GUI API by MicroWindows and remote control API by the FIP module. In addition, there is em85xx DVD core API (such as mpec-4 codec), which I have classified into Linux System set.

The ultimate task of MicroWindows Extension is therefore to port all these APIs from its Linux edition to Win32 edition. Most operating system APIs are compatible and have the same names, such as IO functions. However, there are some discrepancies of the API in Time and alarm routines between GNU C and Microsoft C. Fortunately, most programs (including those with WIN GUI interfaces) can be programmed using clean C/C++ code with limited support from System Time/IO API, GUI Interface API, and FPI API. FPI API is not standard windows API and can be discarded by using the keyboard event interface provided by the standard GUI interface API. I will explain this in section 2.4 and 2.5.

For now, the major task is to ensure that those APIs described above (also shown in brown letters in Figure 2.) are all ported to Win32 and functional. Application developers are required to conform to this code standard, i.e. using only standard c/c++ and the supported APIs. In this way, their program can be compiled with both Visual C++ (a standard C/C++ compiler) and GCC (a Linux compiler) using respective versions of the MicroWindows Extension (Win32) and MicroWindows (Linux). Please refer to Figure 1.

2.2 Porting MicroWindows: a device-independent GUI library

This section I will describe the process of porting the mayor part of MicroWindows to Win32. I say MAJOR part, because the original package downloaded from the website [2] includes custom builds for Linux and MSDOS. Many different kinds of screen, keyboard and mouse drivers are included. Three sets of equivalent GUI API sets are also provided. Please refer to ".\configure" file from the package. However, I only need to port so-called NanoX GUI API and screen and keyboard drivers that are suitable to be implemented by a software simulator. In this section I will describe only issues on porting NanoX GUI API. In the following sections I will talk about driver development.

Compile MicroWindows with VC++7.1:

Although, MicroWindows is largely written for compact Linux kernel, it is clean enough to be compiled with Microsoft c++ compiler, on condition that one do not choose to build any network capabilities. Functionalities in MicroWindows are grouped with files, i.e. the more files you compile and link, the more functionalities you get. In my current solution, I have included only files for a simulated graphic and keyboard server. Please refer to the VC++ solution for a list of files I have included to compile.

All errors during direct compilation with vc7.1 compiler can be eliminated in the following ways:

(1) Add MACRO to the project: NONETWORK; MSDOS; HAVE_FILEIO; COUNTER_EVENTSELECT.

NONETWORK: turns off all network codes in the srvmain.c, and turn on
MSDOS: tells the compiler not to compile Linux-specific code.
HAVE_FILEIO: Include IO functions, such as loading photo/font files.
COUNTER_EVENTSELECT: This macro is not included by the original build. I have added it to substitute timer and polling mechanism for keyboard and mouse used by the Library, so that user-defined software drivers can be loaded. Please see (13) for more information.

(2) Remove any include files that is only UNIX specific. There is only one or two such header files to remove
(3) Remove <endian.h> lines in swap.h. We will assume -little endian- for Intel CPU;
(4) In the device.h, the timeval struct should be replaced by an arbitray one

```
*       //#if UNIX || DOS_DJGPP
        #include <time.h>
        ...
        // added by LiXizhi
*       struct timeval {
*               time_t          tv_sec;             /* seconds */
*               time_t          tv_usec;    /* microseconds */
*       };
O       typedef struct mw_timer MWTIMER;
        struct mw_timer {
O               struct timeval      timeout;
        ...

*       //#endif
```

Asterisked lines need to be changed as above. See (9) for more information.

(5) Choose any single screen driver in the driver directory to compile

In the example, I have chosen to use scr_svga.c. We need to replace the body of those functions in the screen driver with our own ones. Any screen driver has a global struct called scrdev, which contains function pointers to current device drivers. You can replace them with your own driver implementation. The most important ones are _drawpixel, _readpixel, _drawhline, _drawvline.

```
SCREENDEVICE    scrdev = {
                    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, NULL,
                    SVGA_open,
                    SVGA_close,
                    SVGA_getscreeninfo,
                    SVGA_setpalette,
                *   SVGA_drawpixel,
                *   SVGA_readpixel,
                *   SVGA_drawhline,
                *   SVGA_drawvline,
                    SVGA_fillrect,
                    gen_fonts,
                    SVGA_blit,
                    NULL,                   /* PreSelect*/
                    NULL,                   /* DrawArea subdriver*/
                    NULL,                   /* SetIOPermissions*/
                    gen_allocatememgc,
                    NULL,                   /* MapMemGC*/
                    NULL,                   /* FreeMemGC*/
                    NULL,                   /* StretchBlit subdriver*/
                    NULL                    /* SetPortrait*/
                };

            // Here we provides any implementations
            static PSD  SVGA_open(PSD psd);
            static void SVGA_close(PSD psd);
            static void SVGA_getscreeninfo(PSD psd,PMWSCREENINFO psi);
            static void SVGA_setpalette(PSD psd,int first,int count,MWPALENTRY *pal);
            static void SVGA_drawpixel(PSD psd,MWCOORD x, MWCOORD y, MWPIXELVAL c);
            static MWPIXELVAL SVGA_readpixel(PSD psd,MWCOORD x, MWCOORD y);
            static void SVGA_drawhline(PSD psd,MWCOORD x1, MWCOORD x2, MWCOORD y,
MWPIXELVAL c);
            static void SVGA_drawvline(PSD psd,MWCOORD x, MWCOORD y1, MWCOORD y2,
MWPIXELVAL c);
            static void SVGA_fillrect(PSD psd,MWCOORD x1, MWCOORD y1, MWCOORD x2,
                    MWCOORD y2, MWPIXELVAL c);
            static void SVGA_blit(PSD dstpsd, MWCOORD dstx, MWCOORD dsty, MWCOORD w,
                    MWCOORD h, PSD srcpsd, MWCOORD srcx, MWCOORD srcy,
long op);
```

We are certainly not using SVGA implementation. So comment the line as below
///#include <vga.h>
And clear the body of all the driver functions in scr_svga.c. I have renamed this file to scr_nul_sim.c; because it is actually a NULL driver. The actual driver is loaded when the simulator is running. So far, we have created a blank screen driver, we can later write our own driver. We do so, only to make the MicroWindows library compliable.

(6) Add the following Include dir to project:
microwindows\include
microwindows\driver
microwindows\engine

(7) set compiler warning level to 2
(8) remove all as //#include <unistd.h>
(9) use win32 GetTimeofday in .\engine\devtimer.c

```
#include <time.h>
//--------------------------------------------------------------------
// added by LiXizhi for porting code to win32
// gettimeofday is the win32 counterpart of the linux function gettimeofday
//--------------------------------------------------------------------
/* Functions to capsule or serve linux style function
* for Windows Visual C++
*/
int gettimeofday(struct timeval *time_Info, struct timezone *timezone_Info)
{
```

```
                /* Get the time, if they want it */
                if (time_Info != NULL) {

                        time_Info->tv_sec = time(NULL)%100000;
                        time_Info->tv_usec = (time_Info->tv_sec*1000000);
                }
                /* Get the timezone, if they want it */
                if (timezone_Info != NULL) {
                }
                /* And return */
                return 0;
        }
```

(10) in device.h
change #define ALLOCA(size) alloca(size) into
#define ALLOCA(size) _alloca(size)
This will force to use the C version.

(11) in srvmain.c
change as below
//return (DWORD)(clock() * 1000 / CLOCKS_PER_SEC);
return (GR_TIMEOUT)(clock() * 1000 / CLOCKS_PER_SEC);

(12) in Engine\devimage.c add image file support
//#define HAVE_MMAP 1 /* =1 to use mmap if available */
#define HAVE_GIF_SUPPORT
//#define HAVE_PNG_SUPPORT
#define HAVE_BMP_SUPPORT
Here above, we will turn on GIF and BMP file format support.

in GdLoadImageFromFile() add **O_BINARY** flag. See below. This is because O_BINARY is not default with win32 IO; otherwise the size of bits read will not match the size of the file. Return character will be regarded as EOF in text file mode.
fd = open(path, O_RDONLY|**O_BINARY**);

(13) COUNTER_EVENTSELECT Macro controlled code in srvmain.c

This macro is not included by the original build. I have added it to substitute timer and polling mechanism for keyboard and mouse used by the Library, so that user-defined software drivers can be loaded. If COUNTER_EVENTSELECT is defined, then the event handling mechanism is given in the code block below.

GsSelect() is the function that actually generate primitive keyboard and mouse event and queue them into the system event queue. Difference OS system has different ways to read those events from corresponding drivers. In Linux, all such IO drivers are interfaced through FILE-like interface and it supports timeout; the MSDOS version just polls the drivers for any incoming events and does not support any timeout mechanism. In my own implementation, I combined the timeout functionality of the Linux version and the ease of the MSDOS version. This is for two reasons:

(a) FILE interface between Linux and Windows differs.
(b) Software simulated drivers suit the polling mechanism best.

If you pay attention to the code, I have avoided using any System time functions (since they are different in different OS as I have described). Instead a simulated timeout mechanism is given. It just polls timeout*nUnitIterTimes times before exits . Most functions only uses GsSelect(0), which behaves very much like a simple polling for all devices connected to the OS.

If no message is read during timeout, a GR_EVENT_TYPE_TIMEOUT is injected to the message queue.

```
Void GsSelect(GR_TIMEOUT timeout)
{
	/*================================================================
	* modifed by LiXizhi
	* handle timeout in an ad hoc way.
	* let the polling runs timeout*nUnitIterTimes times before exits
	================================================================*/
	const int nUnitIterTimes = 1;
	int nTimes = timeout*nUnitIterTimes;
	int i = 0;
	int bReceived = 0;

	do
	{
		/* If mouse data present, service it*/
		if(mousedev.Poll())
		{
			while(GsCheckMouseEvent())
				continue;
			bReceived = 1;
		}

		/* If keyboard data present, service it*/
		if(kbddev.Poll())
		{
			while(GsCheckKeyboardEvent())
				continue;
			bReceived = 1;
		}
	}while( (++i<nTimes) && (bReceived==0) );

	if(bReceived==0)
	{
		/*
		* Timeout has occured.  Currently return
		* a timeout event regardless of whether
		* client has selected for it.
		*/
		GR_EVENT_GENERAL *	gp;
		gp = (GR_EVENT_GENERAL *)GsAllocEvent(curclient);
		if(gp)
			gp->type = GR_EVENT_TYPE_TIMEOUT;
	}
}
```

(14) For screen driver. Change as below to set pixel format in mwtypes.h

```
#ifndef MWPIXEL_FORMAT
//#define MWPIXEL_FORMAT    MWPF_TRUECOLOR0888
#define MWPIXEL_FORMAT      MWPF_TRUECOLOR332
#endif
```

For simplicity, the software driver for screen device will not use a palette. So MWPIXEL_FORMAT is set to MWPF_TRUECOLOR332 in the MicroWindows type header file. MWPF_TRUECOLOR332 uses a simple bit mapping between 32 bits true color and 8(3/3/2) bits output color. You can refer to the screen driver implementation from the simulator in Section 3.

2.3 User-defined screen device driver interface

In MicroWindows, all device drivers are linked to the system by setting function-pointers in a global stance of the responding device structure. For Screen driver, this instance is called kdbsrc. See below. I have developed this driver from the original Src_SVGA driver, now named kdb_null_sim.c. The em85xx also developed its own OSD screen driver, which can be found in scr_quasarosd.c.

```
/* specific driver entry points*/
```

```
static PSD  SVGA_open(PSD psd);
static void SVGA_close(PSD psd);
static void SVGA_getscreeninfo(PSD psd,PMWSCREENINFO psi);
static void SVGA_setpalette(PSD psd,int first,int count,MWPALENTRY *pal);
static void SVGA_drawpixel(PSD psd,MWCOORD x, MWCOORD y, MWPIXELVAL c);
static MWPIXELVAL SVGA_readpixel(PSD psd,MWCOORD x, MWCOORD y);
static void SVGA_drawhline(PSD psd,MWCOORD x1, MWCOORD x2, MWCOORD y, MWPIXELVAL c);
static void SVGA_drawvline(PSD psd,MWCOORD x, MWCOORD y1, MWCOORD y2, MWPIXELVAL c);
static void SVGA_fillrect(PSD psd,MWCOORD x1, MWCOORD y1, MWCOORD x2,
                        MWCOORD y2, MWPIXELVAL c);
static void SVGA_blit(PSD dstpsd, MWCOORD dstx, MWCOORD dsty, MWCOORD w,
                        MWCOORD h, PSD srcpsd, MWCOORD srcx, MWCOORD srcy, long op);

SCREENDEVICE      scrdev = {
            0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, NULL,
            SVGA_open,
            SVGA_close,
            SVGA_getscreeninfo,
            SVGA_setpalette,
            SVGA_drawpixel,
            SVGA_readpixel,
            SVGA_drawhline,
            SVGA_drawvline,
            SVGA_fillrect,
            gen_fonts,
            SVGA_blit,
            NULL,                           /* PreSelect*/
            NULL,                           /* DrawArea subdriver*/
            NULL,                           /* SetIOPermissions*/
            gen_allocatememgc,
            NULL,                           /* MapMemGC*/
            NULL,                           /* FreeMemGC*/
            NULL,                           /* StretchBlit subdriver*/
            NULL                            /* SetPortrait*/
};

void ResetDevice(SCREENDEVICE* pScrDev)
{
            scrdev.Open = pScrDev->Open;
            scrdev.Close = pScrDev->Close;
            scrdev.GetScreenInfo = pScrDev->GetScreenInfo;
            scrdev.SetPalette = pScrDev->SetPalette;
            scrdev.DrawPixel = pScrDev->DrawPixel;
            scrdev.ReadPixel = pScrDev->ReadPixel;
            scrdev.DrawHorzLine = pScrDev->DrawHorzLine;
            scrdev.DrawVertLine = pScrDev->DrawVertLine;
            scrdev.FillRect = pScrDev->FillRect;
}
```

By calling ResetDevice from the simulator, any user-defined screen device can replace the original DO_NOTHING driver. You can refer to the driver implementation from the simulator in Section 3.

2.4 User-defined keyboard device driver interface

In MicroWindows, all device drivers are linked to the system by setting function-pointers in a global stance of the responding device structure. For Keyboard driver, this instance is called kdbdev. See below. I have developed this driver from the original NULL driver (kdb_null), now named kdb_null_sim.c.

```
KBDDEVICE kbddev = {
            NUL_Open,
            NUL_Close,
            NUL_GetModifierInfo,
            NUL_Read,
            NUL_Poll
};

/*============================================================
```

```
* ResetKdbDev(); ResetNullKdbDev(); will be called by user application
=============================================================*/
//-----------------------------------------------------------
// desc: added by LiXizhi
// Install user-defined simulated Keyboard device.
//-----------------------------------------------------------
void ResetKdbDev(KBDDEVICE * pKdbDriver)
{
	kbddev = *pKdbDriver;
}

//-----------------------------------------------------------
// desc: added by LiXizhi
// Install user-defined simulated Keyboard device.
//-----------------------------------------------------------
void ResetNullKdbDev()
{
	KBDDEVICE  Nulkbddev = {
		NUL_Open,
		NUL_Close,
		NUL_GetModifierInfo,
		NUL_Read,
		NUL_Poll
	};
	kbddev = Nulkbddev;
}
```

By calling ResetKdbDev from the simulator, any user-defined keyboard device can replace the original DO_NOTHING driver. You can refer to the driver implementation from the simulator in Section 3.

2.5 FIP driver module

Although I have provided simulation for FIP driver module of the em85xx in the MWinSimulator, I highly recommend using the build-in keyboard driver and standard message routing mechanism for any keyboard handling in user application development. The DVD player application provided in the original ship uses FIP driver module and installs a NULL keyboard driver for the MicroWindows. This is very inefficient to deal with keyboard message.

I therefore suggest that a keyboard driver be developed that calls FIP functions in its implementation and compile it with the MicroWindows Lib, so that User application can use keyboard in a standard consistent way. This is actually exactly what I did in the software simulator. In the software simulator, I only simulated FIP APIs, and in my simulation for the keyboard driver, I called the corresponding simulated FIP functions.

3 MicroWindows embedded-hardware device simulator

3.1 Introduction

MWinSimulator is a software simulator for programs developed using MicroWindows Extension. The main task of the simulator is to allow embedded software developers edit, run and debug their code all in a single place with their most familiar IDE such as Visual C++.NET 2003(the current version). Please see Figure 3.

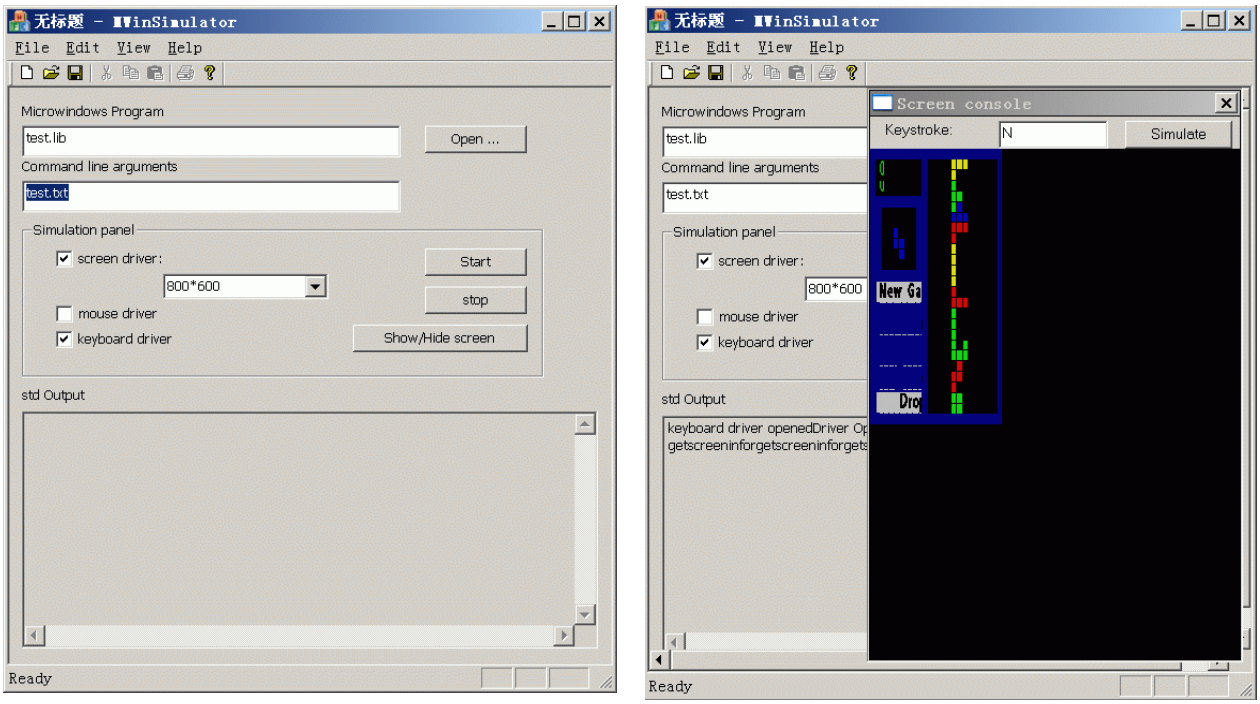

Figure 3. MWinSimulator screen shot

MWinSimulator can be used to simulate any programs (including those with WIN GUI interfaces) that use only clean C/C++ code with limited support from System Time/IO API, MicroWindows GUI Interface API, and FPI API. FPI API is a remote controller module provided by the em85xx. FPI API is not standard API and can be discarded by using the keyboard event interface provided by the MicroWindows GUI interface API.
MWinSimulator simulates both the keyboard input and screen output. In the right figure above, a sample Tetris game is being simulated.

3.2 MWinSimulator Architecture

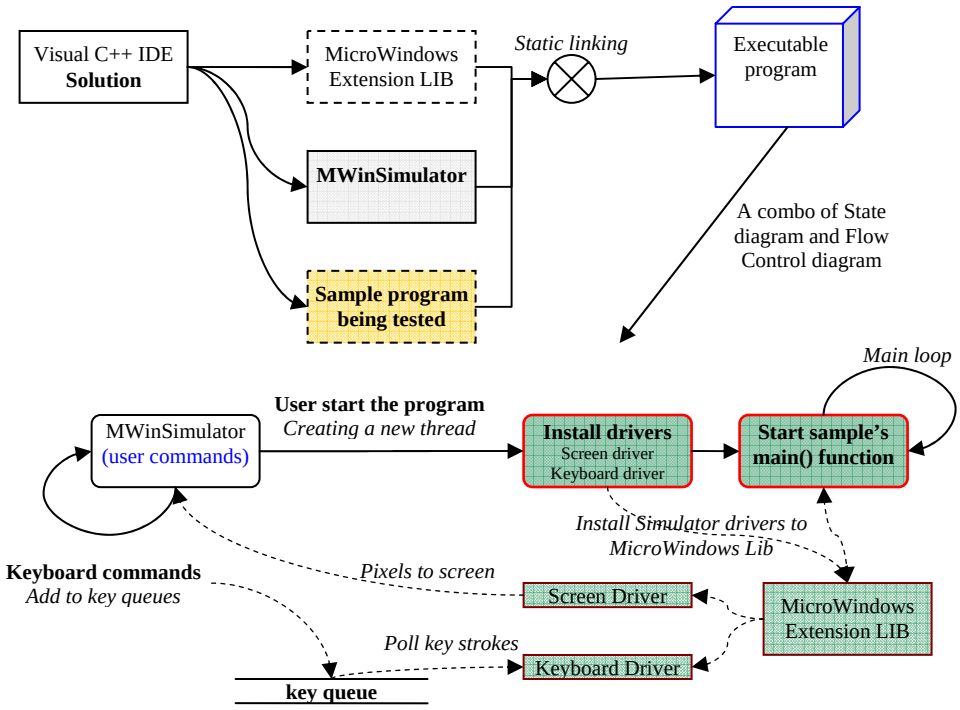

Figure 4. A combo of State diagram and Flow Control diagram

Both the sample program and the MicroWindows are static libraries as indicated by the dotted lines.
A separate thread is created to run the sample program as indicated in red border of the state block in the above figure.

3.3 User interface design

The user interface must accommodate a monitor and a key board simulator. In Figure 3(on the right), a screen console window simulate the video output from the system. Key input is simulated by entering the key in the hotkey box and press Simulate button next to it; the effect is similar like pressing a key with the real remote controller or keyboard.

3.4 Multi-threading

Since we are simulating the program, we need to run the code in another thread for em85xx.
Multi-threading and data synchronization needs to be considered, when passing event from the simulator interface to the tested program. The Screen driver, keyboard driver and FIP remote control driver need to be thread-safe. Critical section is used in the situation.

3.5 Screen driver

The main task of the simulated driver is to output everything that the screen driver received to the screen (draw using GDI functions to a region on the screen).

3.6 Keyboard driver

The main task of the simulated driver is to read from the key queue prepared by the software simulator. Whenever the user has sent a keyboard stroke to the simulator, it is first inserted to the key queue as shown in Figure 4. The keyboard driver then read from this FIFO queue to decide whether and what key has been pressed.

3.7 FIP remote control simulation for em85xx

Although I have provided simulation for FIP driver module of the em85xx in the MWinSimulator, I highly recommend using the build-in keyboard driver and standard message routing mechanism for any keyboard handling in user application development. The DVD player application provided in the original ship uses FIP driver module and installs a NULL keyboard driver for the MicroWindows. This is very inefficient to deal with keyboard message.

I therefore suggest that a keyboard driver be developed that calls FIP functions in its implementation and compile it with the MicroWindows Lib, so that User application can use keyboard in a standard consistent way. This is actually exactly what I did in the software simulator. In the software simulator, I only simulated FIP APIs, and in my simulation for the keyboard driver, I called the corresponding simulated FIP functions.

The Simulator currently supports either programming model for the keyboard.

4 Sample Applications

User Manuals

Here are the basic steps to begin your first user application with the new environment.

- Copy the all files in the SDK to your working directory.
- Open the WinEDVD solution: there should be three projects in it: MicroWindows, MWinSimulator and SampleMVsimLib
- Create your new program file under the sample SampleMVsimLib project. For instance MyProgram.cpp
- Place the basic include and main() function code using the template given below

```
#include <stdio.h>
#include <stdlib.h>
#include "nano-X.h"

#ifdef TESTLIB
#define exit(x)                         return (x) // just a ad hoc manner
int testlib_main (int argc, char *argv[])
#else
int main(int argc, char *argv[])
#endif
{
        return 0;
}
```

  The macro TESTLIB ensures that in the Simulator project, the main function is compiled as a library function called testlib_main() which will be imported and called by the Simulator. You can not change the name of this function. You are definitely free to change the declaration in the #else part. You can declare the function as a main() or as any names a library is what you need.

- Change the MyProgram.h header file
  You are not allowed to replace this header file, because this file is required by the simulator. It contains the definition of the testlib_main() function. Undo the extern "C" linkage in the header file, if you are simulating C++ programs. Pay attention to the file extensions, since *.cpp will be compiled as c++; while *.c will be compiled as c. So if you are naming your file as MyProgram.cpp, you must remove extern “C” linkage. And you must include it if your file is Myprogram.c

Note: Pay attention to the code, do not use exit(0) function. Make sure that your program can always quit gracefully either by responding to a key stroke or the completion of a lengthy task.

Now you are ready to compile, run and debug your code with the Visual C++.NET IDE. A sample file called MyProgram.Cpp is included. You can always start your own application by modifying it.

The following sample applications are given as tutorials of using the new platform. It may contain solutions to some unexpected errors your might met.

4.1 Sample applications from MicroWindows demos

I have translated some sample applications from the MicroWindows demo to the new platform. Basically code can be directly compiled using the new environment. However, we must sidestep some Unix Specific system calls if you have to use this functionality in the final application such as timing and file IO.

4.1.1 MyProgram.cpp

This is a very simple C++ application that shows a gif picture with MicroWindows APIs. IO operation and image display are illustrated. It uses a gif resource file called tux.gif. If you use relative path to specify the file, it must be located in the WMSimulator directory.

4.1.2 NTetris.c

This is a slightly more complex C application that implements the famous Tetris game. Nano-X Win GUI and Message handling (for keyboards) are demonstrated in the sample. In order for your GUI window to receive keyboard messages, you must manually set focus to the receiving window after you created it. The following is the sample code. You can find it in the project files.

```
// added by LiXizhi, since we donot have the mouse driver, we must manual set the focus
// on this window, otherwise no keyboard message will be received
GrSetFocus(state->main_window);
```

Please pay attention to use any time and file IO specific functions, such as Sleep(), Gettimeofday(), open(), close(), etc. You need to write your own implementation of these functions that are compatible with both versions of the OS.

4.2 DVD player

The original code of this application is provided by em85xx. Its event handling is based on its old polling mechanism with fip module; while its display interface is built upon a thin-wrapper (OsdWindow) of MicroWindows API. This mixture of coding style makes it extremely inefficient to be ported to the simulator.

5 Summarizing and streamlining

5.1 Conclusion

Software debuggers are not only essential for successful Soc architecture exploration and benchmarking, but also the key to late user application development on Soc systems. For the former part of software debugging, both hardware and software are involved to perform a certain debugging technique such as probing brought out by I/O pins and various scan technologies with software support on a host computer. However, when it comes to user application development based on a specific Soc design, there are usually no standard software debugging principles to follow. Usually, no further tools are created for this purpose, and one of the former hardware-software debugging techniques is used. However, user application focuses more on functionality and Graphic User Interface (GUI) development than on probing into the internal states of Soc circuit. Moreover, on-chip debugging is both expensive and time-consuming; a shorter edit-run-debug cycle is preferred. Sometimes, a cross-compiler is developed, so that user application developers can at least compile their code in a standard familiar host environment. Even so, real-time

software debugging is not possible if without developing high-level software simulation kits dedicated to a certain Soc design. This article provides the details of a simulation kits design with video output and remote controller (keyboard) input. Any user applications based on this specific Soc can be developed, run and full-debugged all through software environment and in a familiar host IDE such as Visual Studio and Linux KDE. It may provide a useful guide for Soc software developers who are developing GUI applications based on a non-standard Soc system.

5.2 Streamlining Software-based Simulations

TODO: Performing tests with real hardware system: Integration into the hardware development cycle.
TODO: User application development: HTML browser, E-BOOK viewer.

Appendix A: Sample code

References (incomplete)
Refer to the main design doc
[1] EM85xx_usergui.pdf
[2] www.microwindows.org
[3] www.UcLinux.com
[4] ParaEngine