CL-PS7500FE Development Kit

Software User's Guide

Embedded Processors Division

Version 1.0 March 1999

Table of Contents

1. Introduction

The CL-PS7500FE evaluation kit example software is targeted at software developers who plan to port operating systems and applications to the CL-PS7500FE. A library of routines is provided which configure and operate all of the peripherals on the CL-PS7500FE evaluation board. Additionally, there is a set of sample programs that use this library to exercise the peripherals on the board.

To use the CL-PS7500FE evaluation kit example software, the ARM® SDT Version 2.50 containing the project manager, 'C' compiler, assembler, linker, debugger, and ARM instruction set emulator is required. The code in this kit is for a PC running Windows 95® or Windows NT 4.0®. Familiarity with the ARM debugger and project manager is assumed in this document.

2. ARM Software Development Toolkit

Included on the CD-ROM is a 60-day evaluation version of the ARM SDT Version 2.50. This is a fully functional evaluation copy of the SDT that will cease to execute after 60 days. To obtain the full version of the ARM SDT, contact ARM Ltd. (www.arm.com).

To install the ARM SDT, simply run setup.exe from the *sdt250* directory on the CD-ROM and follow the on-screen directions. If you are unfamiliar with the ARM SDT, be sure to select the online manuals as one of the components to install.

The evaluation version of the SDT will create a seemingly useless directory called "c_dilla"; do not remove this directory or its contents else the evaluation version of the SDT will no longer work, uninstall correctly, and reinstalled on your system.

Once the SDT has been installed, there are two applications that will be used to build and debug applications: the ARM Project Manager and the ARM Debugger. These applications are described fully in the online manuals, but brief instructions for basic use of these applications is provided in the following sections.

2.1. ARM Project Manager

The ARM Project Manager is used to develop and build applications for the CL-PS7500FE evaluation board. It is found in the "Start" menu under "Programs" then "ARM SDT v2.50". Once run, it will show an empty workspace. The first thing to do is to open a project file (which has an extension of ".apj"). This is done by selecting "Open" from the "File" menu.

Once a project has been opened, there are several basic operations that you can perform:

1) Edit the source code. To do this, click on the "+" beside the "ARM Executable Image" line in the project window, then the "+" next to the "Debug" line, then the "+" next to the "Sources" line, and then double click on the source file to be edited. The source file will be opened in another window. Standard editing operations can be performed on the source code in this window.

2) Build the executable. To do this, click on the "Build Debug" button at the bottom of the project window. You can build either a debug, debug-release, or release version of the executable; selecting the "Debug", "DebugRel", or "Release" line from the project window will change the button at the bottom to "Build Debug", "Build DebugRel", or "Build Release" respectively. Alternately, you can select "Build" from the "Project" menu or press Shift+F8 to build the executable. The debug version of an executable contains all the symbolic information needed to debug the executable, the debug-release version contains some symbolic debugging information and some optimizations, and the release version contains no symbolic information and is fully optimized.

3) Debug the executable. To do this, select "Debug" from the "Project" menu, or press F5. This will launch the ARM Debugger, load the executable image, and set a breakpoint at the beginning of the "main" function.

4) Run the executable. To do this, select "Execute" from the "Project" menu, or press Ctrl+F5. This will launch the ARM Debugger, load the executable image, and begin execution of the program.

2.2. ARM Debugger

The ARM Debugger is used to debug applications on the CL-PS7500FE evaluation board. It is found in the "Start" menu under "Programs" then "ARM SDT v2.50". Once run, it will show three windows: the "Execution Window", "Command Window", and "Console Window". The execution window shows the current execution context of the target system. The command window allows you to type commands to the debugger, though you will probably rarely use this since the same functionality can be easily accessed via the GUI. The console window shows the output from the target system, including anything printed out from your application.

If the ARM Project Manager launches the ARM Debugger, the image to be debugged will be automatically loaded into the target system's memory. Otherwise, the image to be debugged must be loaded manually by selecting "Load Image" from the "File" menu.

Once an image has been loaded into the target system's memory, there are several basic operations that you can perform:

1) View the source files in the program image. Select "Source Files" from the "View" menu, or press Ctrl+F, and a window will be displayed showing all the source files in the program image. Double clicking on one of the file names in the "Source Files" window will open a new window that displays the selected source file.

2) Set a breakpoint. In either the "Execution Window" or a source file window, place the cursor on the line where you want to place a breakpoint. Then select "Toggle Breakpoint" from the "Execute" menu, or press F9. Performing the same action after selecting a line that already has a breakpoint on it will remove the breakpoint.

3) Run the program. Select "Go" from the "Execute" menu, or press F5. The program will begin executing. It will run until a breakpoint is reached, the program terminates, or a fatal error occurs.

4) Single-step the program. The program can be single-stepped in three different ways. You can step to the next point within the program execution, stepping into any procedures called, by selecting "Step In" from the "Execute" menu, or pressing F8. Think of this as stepping into the called procedure. You can step to the next point within the program execution, allowing any called procedure to execute to completion, by selecting "Step" from the "Execute" menu, or pressing F10. Think of this as stepping over any called procedures. Finally, you can step to the first point within the program execution after the current routine has completed executing by selecting "Step Out" from the "Execute" menu, or pressing Shift+F7. Think of this as stepping out of the current procedure.

5) Run the program to a specific line. This is roughly equivalent to setting a breakpoint, running the program, and then removing the breakpoint. Once a line has been selected in the "Execution Window" or a source file window, select "Run to cursor" from the "Execute" menu, or press F7.

6) View the ARM registers. Select "Registers\Current Mode" from the "View" menu. This will display a window with the currently accessible register set. This is the most useful register view for typical use; the other register views would be useful for OS development.

7) View the local variables of the current procedure. Select "Variables\Local" from the "View" menu, or press Ctrl+L. Right clicking on the variables allows you to change the way the variable's value is displayed, or changes the content of the variable. Double clicking on a pointer variable will open a new window showing the contents of the memory pointed to by the variable. Double clicking on other variables will allow you to change the variable's value.

8) View the global variables of the current program. Select "Variables\Global" from the "View" menu, or press Ctrl+G. The global variable window can be manipulated in the exact same way as the local variable window.

9) View the contents of memory. Select "Memory…" from the "View" menu, or press Ctrl+M. A dialog will be displayed allowing you to specify the memory address you wish to view. Once you select "OK", a memory window will be displayed. The memory window will always show 4K worth of data, starting on a 4K boundary, and display the memory contents as hexadecimal 32-bit words.

3. Angel™

The example software provided in the evaluation kit is designed to run under control of the Angel debug monitor version 1.20 from ARM. There are two versions of Angel provided with the evaluation kit: one communicates with the host system debugger via COM1 and the other via COM2. The two versions are identical in every other respect. The two versions exist since the IR port uses COM2, and having Angel communicate with the host via COM2 would prevent the use of the IR port. The version that uses COM1 is called cl7500_1.fl and the version that uses COM2 is called cl7500_2.fl. They reside in the *ps7500fe\angel* directory of the CD-ROM. The evaluation board is shipped with cl7500_1.fl programmed into the boot ROM.

Also contained in the *ps7500fe\angel* directory is the *source* directory, containing the source code for this version of Angel. For this source code to be built, it must be placed in the same directory into which the ARM SDT was installed (i.e., in the same directory that contains the "bin", "cl", "demon", "include", "lib", etc. directories of the ARM SDT).

The ARM Project Manager file cl7500fe.apj under the *source\cl7500fe.b\apm* directory can be used to build Angel. There are two variants of Angel that can be built: Image1 and Image2. The Image1 variant uses COM1 to communicate with the host debugger and the Image2 variant uses COM2. Once built, the file *source\cl7500fe.b\apm\image1\cl7500fe.fl* corresponds to the cl7500_1.fl on the CD-ROM, and the file *source\cl7500fe.b\apm\image2\cl7500fe.fl* corresponds to cl7500_2.fl.

Angel enables the MMU on the CL-PS7500FE and uses it to rearrange the memory map of the processor. The main reason this is done is to place the DRAM of the system at location 0. This has the advantage of placing the exception vectors in DRAM such that they can be replaced (or chained) by an application that needs access to one of the exception vectors. The memory map under Angel appears as follows:

0x1800:0000	ISA Bus Space
0x1400:0000	Reserved
0x1300:0000	I/O Space / CL-PS7500FE Registers
0x1200:0000	Reserved
0x1100:0000	*Flash SIMM
0x1000:0000	Flash SIMM / Boot ROM
0x0C00:0000	DRAM Bank 3
0x0800:0000	DRAM Bank 2
0x0400:0000	DRAM Bank 1
0x0000:0000	DRAM Bank 0

* only when Boot ROM is enabled

The basic change is that physical address 0x0yyy:yyyy appear as logical address 0x1yyy:yyyy and physical address 0x1yyy:yyyy appears as logical address 0x0yyy:yyyy.

4. Board Setup

Follow these steps to setup and communicate with the evaluation board.

1) Connect the supplied NULL modem cable between the "COM 1" connector on the evaluation board and any available COM port on the host system.

2) Apply power to the evaluation board.

3) Start the ARM Debugger on the host system.

4) Select "Configure Debugger…" from the "Options" menu.

5) Select "remote_a" as the target environment.

6) Click on the "Configure…" button to select the host COM port to be used. To speed the debugging process, select a baud rate of 115,200. Click on "OK" when done.

7) Click on "OK". The ARM Debugger should connect to Angel on the evaluation board and then print out a message similar to the following in the "Console Window":

```
Angel Debug Monitor V1.20 (ARM Ltd. 1.20/Cirrus Logic 1.00) for the CL-PS7500FE
Built for ARM7 Serial, IRQ, cache on
Build number 1
Serial Rate: 115200
```

   If there is a problem (such as a bad serial cable, attempting to use the wrong serial port, etc.), the following message will be displayed:

```
Cannot open target: the target is not responding.
```

The ARM Debugger can be changed between debugging programs on the CL-PS7500FE evaluation board (via Angel) and the ARMulator software ARM emulator at any time. Simply select "Configure Debugger…" from the "Options" menu, select "remote_a" (for the CL-PS7500FE evaluation board) or "ARMulate" (for the ARMulator) as the target environment, and then click on "OK".

If there are any communication problems between the host and the CL-PS7500FE evaluation board, the ARM tools might revert to using the ARMulator as a result of an obscurely worded dialog box. If this happens, you will have to use the "Configure Debugger…" option to switch back to debugging on the CL-PS7500FE evaluation board.

5. Lib7500

Lib7500 is a library of routines for accessing the various peripherals on the CL-PS7500FE evaluation board. These routines are not necessarily the most efficient in terms of execution time or code space; they simply demonstrate the proper operation of the peripherals. The library is divided into separate source files for each peripheral. The lib7500.h file contains prototypes for all the functions in the library. The lib7500.apj file is the ARM project manager project file used to build the debug, debug-release, and release variants of the library.

The source code for this library is contained in the *ps7500fe\lib7500* directory of the CD-ROM.

5.1. audio.c

This file contains routines for using the codec interface of the CL-PS7500FE. The codec interface is an output only 44.1-kHz, 16-bit stereo serial codec interface; on the evaluation board it is connected to a Crystal CS4333 DAC.

5.1.1. AudioBreakLoop

```
void AudioBreakLoop(void)
```

This routine stops the repeated playback of an audio buffer (started by calling AudioPlayBg with `iRepeat` non-zero).

5.1.2. AudioDisable

```
void AudioDisable(void)
```

This routine powers off the internal codec interface. The codec interrupt is masked and the interrupt handler is removed.

5.1.3. AudioEnable

```
void AudioEnable(void)
```

This routine configures the internal codec interface. An interrupt handler is installed to handle the codec interrupts. The codec interrupt handler will write pointers to the next block of data to be played into the hardware DMA registers when there is data to be played.

5.1.4. AudioPlay

```
void AudioPlay(char *pcBuffer, long lLength)
```

This routine plays the given buffer of data via the codec interface. This routine will not return until the entire buffer has been played out to the codec interface.

5.1.5. AudioPlayBg

```
void AudioPlayBg(char *pcBuffer, long lLength, iRepeat)
```

This routine plays the given buffer of data via the codec interface. If `iRepeat` is non-zero, the buffer will be played repeatedly until AudioBreakLoop is called. This routine will return immediately, playing the buffer in the background while other processing continues.

5.2. draw.c

This file contains generic drawing primitives. None of these routines are aware of the dimensions of the screen or its color depth (with the exception of DrawSetPixel and DrawGetPixel, which perform the actual pixel manipulations for all the other routines).

5.2.1. DrawChar

```
void DrawChar(char cChar, long lX, long lY, char cColor)
```

This routine draws an ASCII character.

5.2.2. DrawCharX2

```
void DrawCharX2(char cChar, long lX, long lY, char cColor)
```

This routine draws an ASCII character at twice its normal width and height (i.e., 16x16).

5.2.3. DrawCircle

```
void DrawCircle(long lX, long lY, long lRadius, char cColor)
```

This routine draws a circle.

5.2.4. DrawCls

```
void DrawCls(void)
```

This routine erases the frame buffer.

5.2.5. DrawFillCircle

```
void DrawFillCircle(long lX, long lY, long lRadius, char cColor)
```

This routine draws a filled circle.

5.2.6. DrawGetPixel

```
char DrawGetPixel(long lX, long lY)
```

This routine returns the current value of the specified pixel.

5.2.7. DrawLine

```
void DrawLine(long lX1, long lY1, long lX2, long lY2, char cColor)
```

This routine draws a line.

5.2.8. DrawSetPixel

```
void DrawSetPixel(long lX, long lY, char cColor)
```

This routine fills the specified pixel with the given color.

5.2.9. DrawString

```
void DrawString(char *pcBuffer, long lX, long lY, char cColor)
```

This routine draws a string of ASCII characters.

5.2.10. DrawStringX2

```
void DrawString(char *pcBuffer, long lX, long lY, char cColor)
```

This routine draws a string of ASCII characters at twice their normal width and height (i.e., 16x16).

5.3. flash.c

This file contains routines for programming data into the Sharp FLASH memory in the FLASH SIMM socket of the CL-PS7500FE evaluation board. The implementation of these routines is specific to the Sharp FLASH memory specified in the CL-PS7500FE Hardware User's Guide; if another FLASH memory is purchased, these routines will need to be tailored to the programming model of the specific FLASH memory. A sector is an individually erasable block of memory within the FLASH memory.

5.3.1. FlashEraseChip

```
void FlashEraseChip(unsigned long ulFlashBase)
```

This routine erases the FLASH memory located at the given base address. When completed, the entire contents of the memory will be 0xFF.

5.3.2. FlashEraseSector

```
void FlashEraseSector(unsigned long ulFlashBase, long lSector)
```

This routine erases a sector of the FLASH memory located at the given base address. The sector that contains the offset lSector is erased. When completed, the entire contents of that sector of memory will be 0xFF.

5.3.3. FlashNumSectors

```
long FlashNumSectors(void)
```

Returns the number of sectors in the FLASH memory.

5.3.4. FlashProgramBlock

```
void FlashProgramBlock(unsigned long ulFlashBase, long lOffset,
                       unsigned char *pucData, long lNumBytes)
```

This routine programs data into the FLASH memory located at the given base address. The data is programmed at the specified offset into the FLASH memory. The FLASH memory (and therefore this routine) will hang if an attempt is made to program a 0 bit to a 1…this can only be accomplished by an erase operation.

5.3.5. FlashSectorInfo

```
long FlashSectorInfo(long lSector, long *plSectorOffset,
                     long *plSectorLength)
```

Returns information about the specified sector of the FLASH memory. The offset to the beginning of the sector and the length of the sector are returned.

5.4. ir.c

This file contains routines for using the IrDA compatible port on the CL-PS7500FE evaluation board. The IrDA port is a bi-directional infrared communication port that is capable of data transfers at up to 115200 baud. The IrDA port is implemented as a post-processing of the COM2 data stream; therefore the IrDA port can not be used in conjunction with COM2. For the IrDA routines to function properly, jumper JP13 must not be connected (this disables the IrDA encoder).

5.4.1. IRCharReady

```
long IRCharReady(void)
```

This routine determines if there is a character ready to be read from the IrDA port. The return value will be non-zero if there is a character waiting to be read.

5.4.2. IRDisable

```
void IRDisable(void)
```

This routine disables the IrDA port.

5.4.3. IREnable

```
long IREnable(long lDataRate)
```

This routine will configure the IrDA port to the specified data rate. The supported data rates are 115200, 57600, 38400, 19200, and 9600 baud. The return value will be zero if the IrDA port has already been configured or the data rate is invalid, and one otherwise.

5.4.4. IRReceiveChar

```
char IRReceiveChar(void)
```

This routine reads a character from the IrDA port and returns it to the caller.

5.4.5. IRSendChar

```
void IRSendChar(char cChar)
```

This routine sends a character to the IrDA port.

5.5. isr.c

This file contains routines for handling the IRQ interrupt on the ARM processor in the CL-PS7500FE. The InterruptHandler routine is the actual interrupt handler; InterruptShell (contained in isrshell.s) calls it. It is responsible for determining the cause of the interrupt, calling the appropriate handler routine (if one has been registered), and performing the necessary steps to clear the interrupt.

These routines are not called directly by an application. They are used by the other peripheral support routines in lib7500.

5.5.1. InterruptInstallISR

```
void InterruptInstallISR(void)
```

This routine registers the IRQ interrupt handler with Angel so that it can handle any IRQ interrupts to the ARM processor. Calls to this routine are reference counted; if the ISR is already installed, the reference count is simply incremented.

5.5.2. InterruptRemoveISR

```
void InterruptRemoveISR(void)
```

This routine un-registers the IRQ interrupt handler with Angel so that it is no longer called to handle IRQ interrupts to the ARM processor. Calls to this routine are reference counted; the ISR is only un-registered when the reference count reaches zero.

5.5.3. InterruptSetDACHandler

```
void InterruptSetDACHandler(PFNISR pfnDAC)
```

This routine is used to register the address of the routine that will be called when the codec interface interrupt is the cause of the IRQ interrupt to the ARM processor.

5.5.4. InterruptSetKeyboardHandler

```
void InterruptSetKeyboardHandler(PFNISR pfnKeyboard)
```

This routine is used to register the address of the routine that will be called when the keyboard data receive interrupt is the cause of the IRQ interrupt to the ARM processor.

5.5.5. InterruptSetMouseHandler

```
void InterruptSetMouseHandler(PFNISR pfnMouse)
```

This routine is used to register the address of the routine that will be called when the mouse data receive interrupt is the cause of the IRQ interrupt to the ARM processor.

5.6. isrshell.s

This file contains a shell routine used in handling the IRQ interrupt on the ARM processor of the CL-PS7500FE. Since the ARM Procedure Call Standard allows only some of the ARM registers to be destroyed by the called routine, and the actual interrupt handler is in C and therefore will destroy some of those registers, the registers must be saved to prevent the interrupted program from being corrupted. The InterruptShell routine in this file takes care of saving and restoring those registers so that the C language interrupt handler does not trash the system. This routine is also tailored to the interrupt sharing mechanism used by the Angel debug monitor and as such would not work in a fully embedded system (i.e., no Angel).

There are also routines (GetIRQ and SetIRQ) that get and set the IRQ interrupt handler address. They are used by InterruptInstallISR and InterruptRemoveISR and should not be directly called.

5.7. kbd.c

This file contains routines for reading data from the keyboard controller. These routines will perform a simple conversion of the keyboard scan code to the basic ASCII representation of the pressed key; therefore, special keys such as the function keys and the arrow keys are ignored.

5.7.1. KbdDisable

```
void KbdDisable(void)
```

This routine disables the keyboard controller and removes the interrupt handler for the keyboard receives data interrupt.

5.7.2. KbdEnable

```
int KbdEnable(void)
```

This routine configures the keyboard controller and installs the interrupt handler for the keyboard receives data interrupt. The return code will be one if the keyboard was successfully configured and zero otherwise. The keyboard must be plugged into the keyboard port for this routine to succeed.

5.7.3. KbdRead

```
char KbdRead(void)
```

This routine will return the next key read from the keyboard. If there is no key waiting to be read, it will wait until a key has been pressed and return that key.

5.7.4. KbdReady

```
int KbdReady(void)
```

This routine will return non-zero if there is a key waiting to be read from the keyboard and zero if there is no key waiting to be read.

5.8. led.c

This file contains routines for manipulating the LEDs in the LED bar.

5.8.1. LEDOff

```
void LEDOff(int iLED)
```

This routine will turn off the specified LED.

5.8.2. LEDOn

```
void LEDOn(int iLED)
```

This routine will turn on the specified LED.

5.8.3. LEDSetState

```
void LEDSetState(int iState)
```

This routine will set the state of all 8 LEDs. Setting a bit in `iState` to one will turn on the corresponding LED.

5.9. lpt.c

This file contains routines for sending and receiving data via the parallel port. The parallel port is contained in the SMSC 37C665 SuperIO chip and is capable of SPP, EPP, and ECP operation. These routines use simple SPP mode to perform bi-directional data transfer.

5.9.1. LptEnable

```
void LptEnable(void)
```

This routine configures the parallel port for operation as a bi-directional SPP parallel port.

5.9.2. LptReceiveChar

```
char LptReceiveChar(void)
```

This routine reads a character from the parallel port.

5.9.3. LptSendChar

```
void LptSendChar(char cChar)
```

This routine sends a character to the parallel port.

5.10. mouse.c

This file contains routines for reading data from the mouse controller. The mouse movement commands are interpreted and returned as simple changes to the pointer in the X and Y axis, as well as the current state of the mouse buttons.

5.10.1. MouseDisable

```
void MouseDisable(void)
```

This routine disables the mouse controller and removes the interrupt handler for the mouse receives data interrupt.

5.10.2. MouseEnable

```
int MouseEnable(void)
```

This routine configures the mouse controller and installs the interrupt handler for the mouse receive data interrupt. The return code will be one if the mouse was successfully configured and zero otherwise. The mouse must be plugged into the mouse port for this routine to succeed.

5.10.3. MouseRead

```
void MouseRead(int *iDx, int *iDy, int *iButtons)
```

This routine will return the next position update from the mouse. If there is no update presently waiting to be read, it will wait until there is an update to be read.

5.10.4. MouseReady

```
int MouseReady(void)
```

This routine will return non-zero if there is a mouse position update waiting to be read and zero if there is not.

5.11. uart.c

This file contains routines for using the UARTs on the CL-PS7500FE evaluation board. Each UART is a 16C550 compatible device capable of bi-directional asynchronous communication at up to 115200 baud. The two UARTs are contained within the SMSC 37C665 SuperIO chip.

Two global variables are contained in this file: lPort1Enabled and lPort2Enabled. These are used to determine which ports are currently in use. Initially, it is assumed that port 1 is in use (since that is the port being used by Angel for communication with the host debugger, and reusing that port would interfere with Angel). If the hardware configuration varies, the initial values of these two variables can be changed to accurately reflect the hardware.

5.11.1. UARTCharReady

```
long UARTCharReady(long lPort)
```

This routine determines if there is a character ready to be read from the specified UART port. The return value will be non-zero if there is a character waiting to be read.

5.11.2. UARTDisable

```
void UARTDisable(long lPort)
```

This routine will unconfigure the specified UART.

5.11.3. UARTEnable

```
long UARTEnable(long lPort, long lDataRate, long lDataBits,
                long lStopBits, long lParity, long lEvenParity)
```

This routine configures the UART port to the specified data rate and format. The supported data rates are 115200, 57600, 38400, 28800, 19200, 14400, and 9600 baud. The supported data bits are 5, 6, 7, and 8. The supported stop bits are 1 and 2. If lParity is zero, there is no parity bit. If it is non-zero and lEvenParity is zero, then there is an odd parity bit, otherwise there is an even parity bit. The return value will be zero if the UART has already been configured (or the IrDA port if UART2 is requested) or if the data rate and/or format in invalid, and one otherwise.

5.11.4. UARTReceiveChar

```
char UARTReceiveChar(long lPort)
```

This routine will read a character from the specified UART and return it to the caller.

5.11.5. UARTSendChar

```
void UARTSendChar(long lPort, char cChar)
```

This routine will send a character to the specified UART.

5.12. vga.c

This file contains routines for using the VGA controller on the CL-PS7500FE. The VGA controller is configured for operation in 640x480 mode with 8 bits per pixel. The refresh rate is 31.47-kHz horizontal and 60-Hz vertical (i.e., standard VGA refresh rate). This configuration depends upon the Chrontel CH9294 providing a 25.18-MHz pixel clock to the CL-PS7500FE VGA controller; the jumpers at JP2 are configured to provide this clock rate by default.

5.12.1. VGAEnable

```
void VGAEnable(void)
```

This routine configures the VGA controller for a 640x480, 256 color, 60-Hz refresh VGA display. The color palette is configured as 'bbgggrrr', where bb is two bits of blue, ggg is three bits of green, and rrr is three bits of red. Therefore, 0x07 is pure red, 0x38 is pure green, 0xC0 is pure blue, 0x00 is black, and 0xFF is white.

5.12.2. VGAOff

```
void VGAOff(void)
```

This routine turns off the VGA controller.

5.12.3. VGAOn

```
void VGAOn(void)
```

This routine turns on the VGA controller.

6. Samples

There are several sample programs that use the routines in lib7500 to perform some simple demonstrations of the peripherals on the CL-PS7500FE evaluation board. The source code for these routines is contained in the *ps7500fe\samples* directory of the CD-ROM.

6.1. audio

'audio' is a program that plays a 689-Hz sine wave. When run, it will play the sine wave until a key is pressed on the keyboard. The audio.apj project file will build this program.

6.2. flashit

'flashit' is a program that writes data into the FLASH SIMM. When run, it will ask for the name of the file to be programmed into the FLASH. It will then read that file into memory, erase the FLASH SIMM, and program the contents of the file into the FLASH SIMM. All user interactions with this program occurs through the ARM Debugger. The flashit.apj project file will build this program.

Since this program uses the flash routines in lib7500, it is specific to the Sharp FLASH ROM specified in the CL-PS7500FE Hardware User's Guide. If another FLASH ROM is purchased and the routines in lib7500 are changed to match the programming model of the FLASH ROM, then this program will work as is when linked against the modified lib7500.

6.3. irrecv

'irrecv' is a program that receives data from the IrDA port and displays it on the VGA display. When run, the IrDA port is configured at 9600 baud. It will then print all characters received from the IrDA port onto the display. When the "-" character is received, the program will exit. The irrecv.apj project file will build this program.

6.4. irxmit

'irxmit' is a program that sends data out to the IrDA port. When run, the IrDA port is configured at 9600 baud. Data is then read from the host debugger using gets() (which uses the Angel multi-hosting library). The data is then sent out to the IrDA port. This will continue until a "-" character is sent, which will cause the program to exit. The irxmit.apj project file will build this program.

6.5. keyboard

'keyboard' is a program that displays keyboard key presses on the VGA display. Once run, all ASCII keys pressed on the keyboard will be shown on the display. When the "ESC" key is pressed, the program will exit. The keyboard.apj project file will build this program.

6.6. led

led is a program that demonstrates the use of the LEDs in the LED bar. When run, it will light a single LED in the LED bar, and move the lit LED from left to right. It therefore appears that there is a "light ball" which bounces from one end of the LED bar to the other. This will continue until a key is pressed on the keyboard. The led.apj project file will build this program.

6.7. lptrecv

'lptrecv' is a program that demonstrates the use of the parallel port. When run, it will display on the VGA screen each character that is read from the parallel port. When a "-" character is read, the program will exit. The lptrecv.apj project file will build this program.

6.8. mouse

'mouse' is a program that tracks the movements of the mouse with an 'X' on the VGA screen. When run, it will draw a large box in the middle of the screen and two small boxes beneath the large box. An 'X' will appear in the large box and will move around the box when the mouse is moved. An 'X' will appear in either of the two small boxes when the corresponding mouse button is pressed. Pressing both mouse buttons simultaneously will exit the program. The mouse.apj project file will build this program.

6.9. screen

'screen' is a program that demonstrates the drawing capabilities of the VGA controller. When run, it will draw a series of images on the display. Once each image is drawn, it will wait until a keyboard button is pressed, causing it to proceed to the next image. After the last image is drawn and a keyboard button pressed, the program will exit. The screen.apj project file will build this program.

6.10. uartecho

'uartecho' is a program that retransmits all data received on a COM port (i.e., echoes the data back to the sender). When run, it will configure the COM port (which is configurable by the PORT #define in the source code) for 9600 baud, 8-N-1 data format. It will then read characters from the COM port, sending each character read back to the same COM port. This will continue until a "-" character is received, which will cause the program to exit. The uartecho.apj project file will build this program.