Application Note 92

LCD and Keyboard ARMulator model for ADS

Document number: ARM DAI 0092A
Issued: September 2001

ARM

Application Note 92
LCD and Keyboard ARMulator model for ADS

Release information

The following changes have been made to this Application Note.

Change history

Date	Issue	Change
Sept 2001	A	First release

Proprietary notice

Confidentiality status

This document is Open Access. This document has no restriction on distribution.

Feedback on this Application Note

If you have any comments on this Application Note, please send email to errata@arm.com giving:

- the document title
- the document number
- the page number(s) to which your comments refer
- an explanation of your comments.

General suggestions for addtions and improvements are also welcome.

ARM web address

http://www.arm.com

Table of Contents

1 Introduction

Engineers designing embedded systems based around an ARM core often employ an LCD display and keyboard as the primary I/O devices. It is of great benefit to be able to prototype these peripherals in a realistic manner prior to implementation on hardware.

This AppsNote describes in full the processes involved in modelling these two devices as memory-mapped peripherals that are accessed by the ARMulator (the processor simulator provided with the ARM Developer Suite (ADS)). A simple demonstration program written in a combination of C and ARM assembler illustrates how to use them effectively. This is executed from a debugger such as AXD, also provided in the ADS.

The provided source code has been tested using ADS 1.1 and ADS 1.2. This is currently a Windows-only implementation and has been tested on:

- Windows NT4 Service Pack 6
- Windows 98
- Windows 2000

For general information on extending ARMulator, please refer to ***Application Note 32 revision E – "The ARMulator"***.

2 Memory Map

The memory map used in this example for the LCD and keyboard is as follows. It can be altered to suit your needs:

Symbol	Address	Purpose
DISP_BASE	0x0C000000	Base address of entire model.
DISP_ISR	(DISP_BASE+0x0000)	Display interrupt Register
DISP_CSR	(DISP_BASE+0x0004)	Display control Register
DISP_XSIZE	(DISP_BASE+0x0008)	Stores the number of horizontal pixels in LCD
DISP_YSIZE	(DISP_BASE+0x000C)	Stores the number of vertical pixels in LCD
DISPLAY_PTR	(DISP_BASE+0x10)	Address of base of LCD region.
REG_BASE CPU_BASE	(DISPLAY_PTR + 1024*768)	Base for Registers, top of display memory.
CPU_ISR	(CPU_BASE+0x0)	interrupt status register Used to enable keyboard interrupts.
CPU_MR	(CPU_BASE+0x4)	Interrupt mask register
KB_BASE	(CPU_MR+0x08)	Keyboard base
KB_CSR	(KB_BASE+0x0000)	Keyboard status register
KB_ISR	(KB_BASE+0x0004)	Keyboard interrupt register

Note that the memory-mapped LCD begins at `DISPLAY_PTR` and ends at `REG_BASE` (equal to `CPU_BASE`). Be careful not to write to the display past this limit, as registers will be corrupted.

To change the memory map, do the following:

- Modify constant definitions in `console.h`. All locations are relative to `DISP_BASE` and therefore the whole model may be moved by changing this constant.
- To increase the memory reserved for the LCD, change the value of `REG_BASE`.
- If you make major changes to the memory map or register architecture then you will also need to alter two functions defined in `console.c`:
  - `BEGIN_INIT()` – Find the function `ARMulif_ReadBusRange` and ensure that parameter 5 is the peripheral base address and parameter 6 is the number of bytes from this base which should be decoded by the peripheral.
  - `MemAccess_Console` – Under the comment `/* Deal with writes to the LCD display frame */` you may wish to alter the range interpreted as writes to display memory.

3 Using the Model

These instructions illustrate how to run the model, Viewer application and example ARM software.

1. Read the file `readme.txt` supplied with the source ZIP file. If you wish to build the models yourself you will need to copy the supplied ARMulate directory into your ADS installation directory.
2. Copy the following files to your `install_path\Bin` directory where `install_path` is the directory in which you installed ADS (e.g. c:\ADSv1_1). This can also be achieved automatically by running the supplied `copy_console.bat` batch file.

Lcd.exe	The LCD Viewer application
Logo_back.bmp	An image used by the Viewer
Palette8.bmp	Windows DIB Bitmap image whose colour palette is used to match values written to the LCD memory with actual display colours. This can be modified by the User
Console.dll	The ARMulator peripheral model for the LCD and Keyboard
Console.dsc	Non-editable settings for the above model

3. Make the following changes to configuration files, also located within the `install_path\Bin` directory.

   Default.ami

   Add the following to the `{ PeripheralSets` section of the file:

```
;; Console model
{Console=Default_Console
}
```

   Peripherals.ami

   Add the following to the `{ Peripherals` section of the file:

```
{ Default_Console=Console
LCD_WIDTH=480
LCD_HEIGHT=240
}
```

   Note: You may alter the dimensions of the display model by changing the above entries. Ensure that enough memory is allocated for the display region (see above).

4. Load the AXD debugger and open the file `console_demo.axf` which is located in `ARMulate\demo`. You should see an LCD viewer window if debugger loaded the model successfully.
5. Run the demo by pressing F5 or choosing Execute->Go from the menu.

4 Example Application

4.1 Introduction

By default, the example application draws a background image, asks for your age then either draws an animated bitmap or echoes all keystrokes. It illustrates the following aspects of programming the console model:

- Handling interrupts generated by the keyboard.
- Retargetting standard I/O functions (normally Semihosted) to interface to the model.
- Accessing individual pixels in different display bit depths
- Drawing bitmaps
- Optimising data transfers using inline assembler and Load/Store multiple ARM instructions

4.2 Design

The example application consists of the following ARM source files:

- demodata.s Includes Bitmap image data using the INCBIN directive
- console_demo.c The demo application

The following additional files are required for the demo:

- Logo_back.bmp Background image used in the demo
- Armlogo.bmp Bitmap used in animation sequence
- Chars.bmp Bitmap containing characters
- makedemo.bat Batch file to build the demonstration using Visual C++

The application makes use of image data from standard Windows DIB (Device Independent Bitmap) files. The data is loaded into memory with the program image. These structures are exported from the file `demodata.s` (below) and imported into `console_demo.c`.

```
       ;; Define the images to be exported
       EXPORT armlogo
       EXPORT chars
       EXPORT backdrop

       AREA ARMex, DATA, READONLY  ; name this block of code

armlogo
       INCBIN armlogo.bmp
chars
       INCBIN chars.bmp
backdrop
       INCBIN logo_back.bmp

    END                          ; Mark end of file
```

The basic structure of the DIB files is explained below. An in-depth explanation is beyond the scope of this document. Note that with an appropriate decoding function, image data may be loaded from a file format of your choice.

The data begins with a BITMAPFILEHEADER structure which identifies whether or not the image is a bitmap and the number of bytes to offset from this structure where the image data lies. This structure is immediately followed by a BITMAPINFOHEADER giving image width, height, bits per pixel any compression used and some colour information. Next there may be some palette entries followed by the image data. It is assumed that the palette used is the same as that defined in `palette.bmp`. No programmable palette support is provided in this model. However, you may change the `palette.bmp` file to adjust the palette. Note that this only applies when using 8 bit per pixel images. The demonstration ARM program has been designed to operate in two different display modes; 2 and 8 bits-per-pixel. To change from one to the other, do the following:

- Change the constant definition `BITS_PER_PIXEL` in `console.h` to 2 or 8 (the default)
- Rebuild the peripheral model (see later)
- Rebuild the demo program using `makedemo.bat`

The application defines the bitmap structures taken from the windows header files. These are packed by default but have to be explicitly packed when using the armcc compiler. The RECT structure is also defined but because the display is 'bottom-up' the top-left corner is actually the bottom-right. FAST_DRAW determines whether or not to use the optimized inline assembler code to perform some graphics operations.

The way in which Windows DIBs work means that all rows must be word-aligned i.e. take up a multiple of four bytes per row. This is accounted for by the `getAlignmentTweak()` function. The main program performs the following operations:

- Call `initLCD()`. This clears the display by filling its address range with the current background colour.
- An IRQ handler routine is installed at address 0x18 in the ARM vector table.
- IRQs are enabled by modifying the CPSR (current program status register).
- A demo is chosen based upon the pixel depth.

When in 2 bit-per-pixel (bpp) mode the program draws a black pixel at each corner of the display. It then enters an infinite loop which draws randomly coloured dots in random positions. The 8bpp demo uses several more features. A background is drawn using the `drawSpriteXY` function. This is used in several contexts as the most basic bitmap drawing function. It takes both a source and destination rectangle so that a partial image can be drawn anywhere on the screen. Note that this will only work in an 8bpp mode.

The example proceeds to display some text by using the retargeted `fputc` function defined in the same file. This means that the semihosted version of the function does not get linked with the other object files. All other functions which use this low-level command will output to the LCD display rather than the debugger console. Equally, this applies to the `fgetc` function used for keyboard input. Depending on the option that you choose, a moving bitmap image will be displayed or characters that you enter will be echoed to the LCD.

The LCD model sends keystrokes from the Viewer window to the ARMulator peripheral model. The character code is stored in the low byte of the `KB_CSR` register and an IRQ signal is asserted. The IRQ may be cleared by writing to the `KB_ISR` register. Key Up events have a character code 255.

See the source code for further implementation details. You may also wish to refer to the ***Developer Guide (ARM DUI 0056C), Chapter 6: Writing Code for ROM*** for more details on retargeting.

4.3 Building the Application

The demo is compiled and linked by running the supplied batch file `makedemo.bat`. Ensure that the ADS has been installed and the relevant environment variables have been set.

5 The ARMulator Model

5.1 Introduction

The ARMulator peripheral model is responsible for the following:

- Registering the peripherals address range with the address decoder contained in the ARMulator.
- Setting up a shared memory file containing the LCD data and launching the viewer application. Also set up a Remote Procedure Call (RPC) server which is the mechanism used for communicating keystrokes and screen dimensions from/to the viewer.
- Handling all memory accesses and either storing or providing bytes.
- Controlling keyboard interrupts, queueing and dequeueing keyboard events.
- Shutting down the viewer and freeing up all allocated resources.

5.2 Design

The model resides within console.dll which is dependent upon the following files:

- Console.h - constant definitions used by the model, viewer and demo applications.
- Console.c - Main peripheral model.
- Xlcd.c - Routines which handle the platform-specific LCD interface[1].
- Xlcd.h - Header file for the above.
- Console_rpc_s.c - RPC interface code generated by Microsoft MIDL utility.
- Console_rpc.h - Header file for the above.

The macros `BEGIN_STATE_DECL` and `END_STATE_DECL` create a new structure called `Console_State` that is passed to callback functions and stores the current state of the registers.

`BEGIN_INIT()` is called whenever the module is loaded. It retrieves screen dimensions from the Toolconf database settings in `peripherals.ami`. The LCD model and RPC server are initialised then an Hourglass callback is registered (invoked by the ARMulator every time an instruction is executed. This deals with keyboard interrupts. Finally the memory access function `MemAccess_Console` is registered. Whenever an address is accessed that falls within the registered range, this function will be called.

Similarly, `BEGIN_EXIT` and `END_EXIT` perform the appropriate clean-up operations to free resources allocated in `BEGIN_INIT`.

`MemAccess_Console` works as follows:

- For writes to an LCD memory address, the address itself and the appropriate number of data bytes (depending on access width) are passed to `LcdModelWrite` which updates the shared memory region with the given data.
- Writes to registers are stored within the Console_State data structure. If the register involved is the Mask Register (CPU_MR) this may change the state of the interrupt line. The `IrqUpdate` function is called to handle any interrupt state change. When writing to the keyboard status register (KB_CSR), if the KB_RDRF bit is set then any outstanding key press in the buffer will be dequeued. This will

[1] Currently only implemented for Windows

cause the low byte of KB_CSR to be set to the key code or the interrupt signal is cleared if no key events are still buffered. Write operations to the LCD always return a result `PERIP_NODECODE` which allow the data to 'fall through' into the default underlying memory model. This removes the need to manually handle read operations from the LCD memory.

- When reading from memory only register accesses are trapped by the peripheral. The read-only registers `DISP_XSIZE` and `DISP_YSIZE` return the display dimensions and may be used by application programs.

Every `HOURGLASS_COUNT` instructions the function `MemHourglass` will check for queued keyboard events. Both 'key up' and 'key down' events are queued. An interrupt is generated for each key event that is queued. Events arrive via the function `QueueKey` implemented in xlcd.c. This is called by the viewer application via the RPC interface.

The `ConfigChange` handler stores the endian configuration of the current processor.

5.3 Building the Model

The model can be built by using the Visual C++ 6 project file supplied with the source code. Alternatively, open a command window and change to the directory which contains the file `makefile.mak`. Typing nmake will build the model. Ensure that the resulting DLL (Dynamic Link Library) file (`console.dll`) is copied to the `install_path\Bin` directory, where `install_path` is the directory in which you installed the ADS.

Note: Microsoft Visual C++ version 5 or 6 must be installed before you can rebuild the model.

6 The Viewer Application

This section summarises the operation of the LCD viewer application that is launched by the peripheral model when the debugger is started. This program makes use of the MFC (Microsoft Foundation Classes). Each of the main classes will be discussed.

`CApp`

This object is created at program startup. The `InitInstance` function is called to initialise the application. After checking the command line to determine whether or not it was launched from the model it will either display a warning message or initialise RPC before displaying the main window.

`CWindow`

The main window class. On initialisation the desired display dimensions are retrieved from the model. Next, a CDib object is instantiated to display the image data held in the shared memory file (or sample image if the viewer was not launched from the peripheral model). After loading the colour palette from the file `palette.bmp` the DIB is passed to a `CPixelDepthChanger` (see below).

`CDib`

This class provides functions to manipulate DIB data including drawing to a device context, palette and file handling.

`CPixelDepthChanger`

This helper class performs a conversion between the source colour depth and the destination (8-bit) colour depth by using a lookup table. This allows the use of different source colour depths.

`CIntervalTimer`

`CIntervalTimer` provides accurately timed callbacks which are used to refresh the LCD display. The higher the rate, the greater the overhead on the host PCs CPU.