

# Application Note **92**

## LCD and Keyboard ARMulator model

Document number: ARM DAI 0092B

Issued: July 2003

Copyright ARM Limited 2003

The ARM logo is displayed in a bold, black, sans-serif font.

---

**Application Note 92**  
**LCD and Keyboard ARMulator model**

Copyright © 2003 ARM Limited. All rights reserved.

**Release information**

The following changes have been made to this Application Note.

**Change history**

<b>Date</b>	<b>Issue</b>	<b>Change</b>
Sept 2001	A	First release
July 2003	B	Changes to reflect ADS 1.2, RVD 1.6.1, RVARMulator ISS 1.3, 1.3.1, and RVCT2.0

**Proprietary notice**

ARM, the ARM Powered logo, Thumb and StrongARM are registered trademarks of ARM Limited.

The ARM logo, AMBA, Angel, ARMulator, EmbeddedICE, ModelGen, Multi-ICE, ARM7TDMI, ARM9TDMI, TDMI, STRONG, and RealView are trademarks of ARM Limited.

All other products, or services, mentioned herein may be trademarks of their respective owners

**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 support@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 additions and improvements are also welcome.

**ARM web address**

<http://www.arm.com>

---

## Table of Contents

<b>1</b>	<b>Introduction .....</b>	<b>4</b>
<b>2</b>	<b>Memory Map .....</b>	<b>5</b>
<b>3</b>	<b>Using the Model.....</b>	<b>6</b>
<b>4</b>	<b>Example Application .....</b>	<b>8</b>
	4.1 Introduction.....	8
	4.2 Design .....	8
	4.3 Building the Application .....	10
<b>5</b>	<b>The ARMulator Model .....</b>	<b>11</b>
	5.1 Introduction.....	11
	5.2 Design .....	11
	5.3 Building the Model.....	12
	5.4 Modifying the Model for Interactive Behaviour .....	12
<b>6</b>	<b>The Viewer Application .....</b>	<b>14</b>
	6.1 Building the Viewer.....	14
<b>7</b>	<b>Known Issues .....</b>	<b>15</b>
<b>8</b>	<b>Useful References .....</b>	<b>16</b>

# 1 Introduction

Engineers designing systems based around an ARM core often employ an LCD display and keypad 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 in hardware. This document describes 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) or as RealView ARMulator Instruction Set Simulator). 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 (provided in ADS), or RealView Debugger (RVD).

Source code for this Application Note can be downloaded from [http://www.arm.com/arm/Application\\_Notes](http://www.arm.com/arm/Application_Notes).

**Note 1** This demonstration is designed for use on Windows platforms.

**Note 2** If you are using RVD to run the demonstration then you must connect to RVARMulator via the localhost connection.

**Note 3** If you want to rebuild the model or the viewer application, Microsoft Visual C++ version 5 or 6 must be installed first. To rebuild the example application image, you need to have either ADS or RealView Compiler Tools installed first.

**Note 4** The example code provided uses `Install_Handler()` to write to the vector table. This method is valid for uncached processors, or when caches are disabled. However, for cached processors, when caches are enabled (e.g., when the `DEFAULT_PAGETABLES` option is selected) cache coherency problems can occur. Essentially, when the install handler writes the instruction to the vector table location, the instruction is treated as data and hence it can be cached in the data cache. When the same memory location is then read through the instruction path (during an instruction fetch), the real contents of memory are read since the data cache is not visible to the instruction path. The quick workaround is to not use pagetables, although a more complete solution would be to invalidate the vector table addresses in both the I and D cache when program execution begins.

## 2 Memory Map

The examples presented here are based upon an earlier model designed for the SDT development tools. These implemented the Windows CE ODO reference platform. It was decided that a similar memory map would be employed for control and status registers as well as the memory-mapped display region.

The memory map is as follows and 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+0x0010)	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+0x0000)	interrupt status register Used to enable keyboard interrupts.
CPU_MR	(CPU_BASE+0x0004)	Interrupt mask register
KB_BASE	(CPU_MR+0x0008)	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 - 1` (equal to `CPU_BASE`). Be careful not to write to the display past this limit, otherwise other 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 `1024*768`.
- If you make major changes to the memory map or register architecture then you will 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 (e.g., `C:\Program Files\ARM\ADSV1_2`). If you are using RealView ARMulator ISS, follow the `ExtensionKit` directory structure down until you reach `win_32-pentium`. Copy the contents of the console model's `ARMulate` directory to this location. The Extension kit is present if you have done a full installation, or a custom installation including the Extension Kit.
- 2 If you are using ADS, copy the following files to your `install_path\Bin` directory where `install_path` is the directory in which you installed ADS (e.g. `C:\Program Files\ARM\ADSV1_2`). This can also be achieved automatically by running the supplied `copy_console.bat` batch file provided you have installed ADS in the default location. If you are using RV ARMulator ISS, you will need to move these files to the RVARMulator executables directory. This is normally a location similar to: `C:\Program Files\ARM\RVARMulator\ARMulator\1.3.1\1310.20030312\win_32-pentium`
  - `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 or equivalent for RV ARMulator ISS

#### Default.ami

Add the following to the `{ PeripheralSets` section of the `default.ami` file within the

`{ Default_Common_Peripherals=Default_Processors_Common`  
definition:

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

#### Peripherals.ami

Add the following to the `{ Peripherals` section of the `peripherals.ami` 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 For RVD only, create a shortcut icon so that RVD will start in the ARMulator `bin` directory. To do this, find the RVD executable (`rvdebug.exe`), normally in a location similar to `C:\Program Files\ARM\RVD\Core\1.6.1\159\win_32-pentium\bin\`. Right click on `rvdebug.exe` and select "Create shortcut" from the resulting context menu. Drag the shortcut to the desktop. Right click on the icon and select "Properties", then choose the "Shortcut" tab in the resulting dialog. In the "Start in" field, enter the path of the ARMulator `bin` directory. For RV ARMulator ISS, this is similar to  
`C:\ProgramFiles\ARM\RVARMulator\ARMulator\1.3.1\1310.20030312\win_32-pentium`.
- 5 Start the debugger and load the image `console_demo.axf` which is located in `ARMulate\demo`. You will see an ARM virtual LCD viewer window if the debugger has successfully loaded the model.
- 6 Run the demo by pressing F5.
- 7 To end the demo, stop execution in the debugger. When AXD is closed, the LCD viewer is automatically closed as well. You need to manually terminate the LCD viewer after closing RVD.

## 4 Example Application

### 4.1 Introduction

The example application illustrates the following aspects of programming the console model:

- Handling interrupts generated by the keyboard.
- Rewriting 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 armasm's INCBIN directive
- Console\_demo.c        The demo application

The following additional files are required for the demo:

- Logo\_back.bmp        Image data used by the demo
- Armlogo.bmp
- Chars.bmp
- makedemo.bat        Batch file to compile and link the demo using ARM compilers

The application makes use of image data from standard Windows DIB (Device Independent Bitmap) files. The data are 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 begin 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
- Rebuild the peripheral model
- 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 new `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.

See the source code for further implementation details.

### 4.3 Building the Application

The demo is compiled and linked by running the supplied batch file `makedemo.bat`. Ensure that the ADS or RealView Compilation Tools 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, queuing and dequeuing 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<sup>1</sup>.
- `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

---

<sup>1</sup> 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.

The value of `HOURGLASS_COUNT` determines how often keyboard events are processed. After this interval, 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. 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. If you are using RV ARMulator ISS, you will need to copy the DLL file to the RVARMulator executables directory. This is normally a location similar to:

```
C:\Program Files\ARM\RVARMulator\ARMulator\1.3.1\1310.20030312\win_32-pentium
```

### 5.4 Modifying the Model for Interactive Behaviour

In the supplied model, ARMulator causes the application interrupt handler to trigger when a key is pressed. In `console_demo.c` the key press is retrieved by calling `getchar()` and displayed on the LCD viewer by calling `putchar()`. Since `getchar()` is a blocking routine it does not allow any other applications to run.

You may wish to change the model so that it responds to key presses on an interrupt basis while other tasks in the application continue to run. In other words, the model needs to respond to key presses without having to stop other applications. To do this, you need to implement a basic type of state machine

Modify the existing `main()` routine in the `console_demo.c` source file to contain the following:

```
int main()
{
    /* Set up keyboard input */
    unsigned *irqvec = (unsigned *)0x18;
    int LCD();

    /* The next line will only work on uncached targets */
    /* or cached targets that have their caches disabled */
    Install_Handler( (unsigned)myIRQhandler, irqvec );

    /* DR added - ENABLE IRQs */
    enable_IRQ();

    /* enable keyboard interrupts through the keyb setup reg. */
    *((unsigned*)CPU_MR) |= KEYB_INTR;
    *((unsigned*)KB_CSR) |= KB_CLK_EN;

    while( 1 ) {
```

```

switch (keyflag) {
    case 0x74:
        /* if t on keyboard presses */
        printf("KEY74\n");
        keyflag = 0;
        break;
    case 0x76:
        /* if v on keyboard pressed */
        printf("KEY76\n");
        keyflag = 0;
        break;
    case 0x32:
        /* if 2 on keyboard pressed */
        printf("KEY32\n");
        keyflag = 0;
        break;
    case 0x73:
        /* if s on keyboard pressed */
        printf("KEY73\n");
        keyflag = 0;
        break;
    default:
        /* When keypressed that we have no state for
*/
        break;
}; /*end switch*/

/* place other application tasks here */

}; /*end while*/

return 0;
}

```

This code will respond when the 't', '2', 's' and 'v' keys on the keyboard are pressed. Other cases can be added to the `switch()` statement using the same format.

The code works as follows. In `__irq myIRQhandler` the code of the key pressed is retrieved from the keyboard interrupt and it is assigned to the global variable `keyflag`. The state machine in the `main()` routine is then set up to respond to the value of `keyflag`. When some action is taken for that value of `keyflag` (in this case a `printf()` statement), `keyflag` is then set to have a value of 0 to ensure that the action is not taken twice. Outside the `switch` statement other tasks can be done until/or if another key is pressed.

The `printf()` statement used here prints its output to the LCD viewer. If you would rather have the output go to the console window on AXD, then all you need to do is to remove the retargeted `fputc()` from the file `console_demo.c`.

## 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 `CPixelFormatChanger` (see below).

### CDib

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

### CPixelFormatChanger

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 load on the CPU.

### 6.1 Building the Viewer

The viewer can be built by using the Visual C++ 6 project file supplied with the source code.

Ensure that the viewer executable (`lcd.exe`) is copied to the `install_path\Bin` directory, where `install_path` is the directory in which you installed ADS. If you are using RV ARMulator ISS, you will need to move this file to the RVARMulator executables directory. This is normally a location similar to:

```
C:\Program Files\ARM\RVARMulator\ARMulator\1.3.1\1310.20030312\win_32-pentium
```

## 7 Known Issues

The LCD model may cause an error if you change the processor core model being used whilst the debugger is loaded. Restarting the debugger will solve this problem.

RVD may not start the LCD viewer application, even though all required files have been copied to the correct locations, the environment variables are correctly set, and you are using a correctly configured shortcut. Exit the debugger, ensure that RVBroker has been shut down, then restart RVD.

If on starting AXD you are presented with “RDI Warning 00129: Can’t initialize” then it is likely that you have an incorrect ARMulator configuration file. Check your model .ami and .dsc files and restart the debugger. See section 3 above for the required changes to peripherals.ami and default.ami, and use the supplied console.dsc file for reference.

If on connecting to ARMulator via localhost in RVD, you get an error which says, “Error V20013 (Vehicle): Unable to connect to emulator/simulator. See Output log for details” followed by, “Failed on remote SIM/EMU/EVM: OpenAgent failed (UnableToInitialise) with a non-null agent handle!”, there is probably an error in your model .ami and .dsc files. If after dismissing the error dialogs you find an error on the Cmd tab which states, “A child of Flatmem failed to initialize, error 129.” then there is likely a problem in the console.dsc file. If instead the Cmd tab only reports a very short ARMulator header, then the fault is probably in the peripherals.ami or default.ami files. In any case, close RVD, ensure that RVBroker has been shut down, correct the error, then restart RVD.

## 8 Useful References

Application Note 32 – The ARMulator