RZ/A1, RZ/A2

RZ/A Linux-4.x BSP Porting Guide

EU_00xxx
Rev.1.00
Apr 26, 2019

Introduction

The purpose of this document is to explain how to modify the RZ/A Linux BSP to work with your own board.

Target Device

RZ/A1L, RZ/A1M, RZ/A1H, RZ/A2M

BSP Kernel Version

Linux-4.9, Linux-4.14, Linux-4.19

Contents

1. The RZ Toaster 2
2. u-boot: Converting RZ/A BSP to your own platform 2
3. u-boot: I/O Pin setup 3
4. u-boot: Programming u-boot into QSPI Flash 5
5. u-boot: Customizing Kernel Boot Commands 5
6. u-boot: Initial Debugging using RAM Only (not Flash) 5
7. u-boot: Adding QSPI Support 7
8. u-boot: Programming QSPI Flash 8
9. Kernel: Converting RZ/A BSP to your own platform 11
10. Kernel: Device Tree Configurations 13
11. Misc: Installing J-Link software in Linux 20
12. FAQ: Why the name SH-Mobile? 20

1. The RZ Toaster

1.1 Naming

For all of the examples in this document, we will use a fake board name of "**toaster**" that we are porting to (because we had to pick some type of name). Therefore, anywhere you see "toaster" name, you should replace that with whatever your board name is.

When choosing a name, it is best to use a simple, single word name without hyphens '-' or underscores '_' in the name.

The text in blue will be text you need to add/modify.

The text in red will be the sample 'toaster' name from the company 'my_company' that you should choose for yourself.

Make sure you keep the case (UPPER and lower) the same as in the examples.

1.2 Git Cloning the Public Repositories

This is important.

Driver code is never perfect. As we are constantly finding/fixing bugs, or adding in new functionality and features, it is recommended that you base your project off the public github repositories using **git**. This way it will be much easier to merge in the latest code updates from Renesas.

The Linux-4.x BSPs are currently designed to clone the u-boot and kernel from the github repositories during the first build.

If you plan on using the cloned git repositories in the BSP to keep track of local modifications (ie, "git commit") it is recommended that you create local "branches" for both u-boot and kernel so that you can receive updates to the *master* branch (on github) and then merge in these updates to your custom BSP local branches.

2. u-boot: Converting RZ/A BSP to your own platform

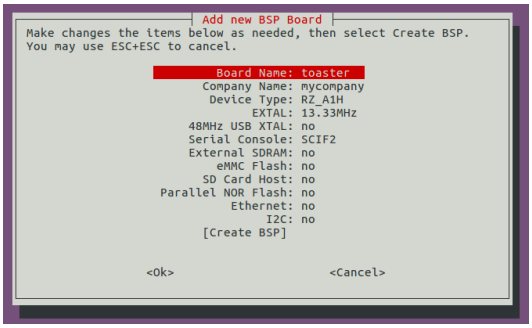
We suggest creating your own files specifically for your board (your own BSP). Do not simply start hacking the existing Renesas board specific files as there will be code and options in there that will probably not apply to your custom board. Also, this will allow you to more easily receive code updates from the Renesas repositories.

2.1 Run the add_new_board.sh script

Located in the base of the u-boot directory is a script that will assist you in creating a BSP for your custom board.

Simply run the script which will display an interface where you can select what components are on your board.

```
$ cd output/u-boot-2017.05
$ ./add_new_board.sh
```



Use the arrow keys and Enter key to make your selections. Once all selections have been chosen, select the last option "[Create BSP]" and press Enter. This will then create new directories and files based on your selections as well as add your board configuration the existing Makefile build system.

2.2 Add your custom board to build.sh

In order to use the "build.sh jlink" command and others, you must define your board.

1. In the BSP directory (rza_linux-4.xx_bsp), edit the file **boards_custom.txt**.
2. Use the **same board name** that you used for add_new_board.sh script. For example, if your board name was "toaster", then you would add that to the line CUSTOM_BOARDS

```
# Enter you board names here:
CUSTOM_BOARDS=" toaster "
```

3. Copy/paste the template (### TEMPLATE TO COPY ###) text and replace the 'xxxx' with your board's name.

   For example:

```
    BRD_DESC_toaster="Toaster board"    # A simple description of your board.
     BRD_SOC_toaster=RZA1H         # The type of RZ/A device: RZA1H, RZA1M, RZA1L, RZA1LU, RZA2M
   BRD_DLRAM_toaster=0x0C000000    # Download RAM address
```

   Please refer to the **boards_renesas.txt** file for examples.

4. Select your board using the "config" command. You can either use the GUI to select your board.

```
./build.sh config
```

Or select it by passing the name of your board on the command line.

```
./build.sh config toaster
```

2.3 Build your BSP

To build a u-boot image for your board, you first need to select the default configuration (defconfig) that was generated by the add_new_boards.sh script. Then you can perform a normal build.

Do this by using the following commands (assumes you are using the Renesas BSP)

```
$ ./build.sh u-boot toaster_defconfig
$ ./build.sh u-boot
```

3. u-boot: I/O Pin setup

While the add_new_board.sh script will help generate and modify the files you need, it will not configure the pins for you.

All the I/O pin mux setup is in board/my_company/toaster/toaster.c.

3.1 Function board_early_init_f( )

Most of the pin setup is done in function board_early_init_f( ). This function is call very early in boot so you can set up all your pins before you try to do things like set up external SDRAM or SPI flash. The "_f" in the function name is there to tell you that this function will run from flash, which makes sense because u-boot can't relocated itself to RAM until the RAM is set up, and the RAM could be external (although by default, we just use internal RAM because its faster and some systems might not even have external SDRAM). With that said, there is also another function board_early_init_r( ) that is still only run once during boot, but later after u-boot has been relocated to RAM.

If you are booting with the MODE pins set to boot from serial flash (QSPI), then some of the QSPI pin will automatically set for you. Basically, only the ones for doing normal single wire SPI. If you want to do the full Quad SPI, or dual Quad SPI, you will need to set those pins up in the board_early_init_f( ) function.

Please make note, the port for the QSPI pins are different between the RZ/A1L and RZ/A1H. So please adjust the code accordingly.

3.2 RZ/A1 Users

Please refer to tables 54.11 to 54.33 in the RZ/A1H Hardware Manual, and tables 41.11 to 41.29 in the RZ/A1L Hardware manual to determine the correct setting for “Alternative Mode”.

For example, in order to assign pin P3_1 to TxD2, you would need to set that pin to Alternative Mode 4.

Table 54.17 Pin Function (P3)

Port Mode		Alternative Mode															
		1st Alternative		2nd Alternative		3rd Alternative		4th Alternative		5th Alternative		6th Alternative		7th Alternative		8th Alternative	
Input	Output	Input	Output	Input	Output	Input	Output	Input	Output	Input	Output	Input	Output	Input	Output	Input	Output
P3_0	P3_0		LCD0_CLK	ET_TXCLK		IRQ2		SCK2		SCI_SCK1			TxD2		PWM2A	RSPCK3	
P3_1	P3_1		LCD0_TCON0		ET_TXER	IRQ6			TxD2		SCI_TXD1	AUDIO_CLK			PWM2B	SSL30	
P3_2	P3_2		LCD0_TCON1		ET_TXEN			RxD2		SCI_RXD1			TEND0		PWM2C	MOSI3	
P3_3	P3_3		LCD0_TCON2	ET_MDIO		IRQ4			BS	SCI_CTS1/RTS1			DACK0		PWM2D	MISO3	

The code would then look like this:

```
pfc_set_pin_function(3, 1, ALT4, 0, 0);    /* P3_1 = TxD2 */
```

3.3 RZ/A2 Users

Please refer to the tables in section 51.3.7 through 51.3.28 in the RZ/A2M Hardware Manual to determine the correct setting for “Alternative Mode”.

For example, in order to assign pin PE_2 to TxD2, you would need to set that pin to Alternative Mode 3.

Table 51.23 Register Settings for Input/Output Pin Function

PSEL[2:0] Register Setting Value	Pin						
	PE_0	PE_1	PE_2	PE_3	PE_4	PE_5	PE_6
000b (Value after reset)	Hi-Z						
001b	ET0_RXCLK	ET0_RXD0	ET0_RXD1	ET0_RXER	ET0_CRS	ET0_MDC	ET0_MDIO
010b	VIO_FLD	VIO_D7	VIO_D6	VIO_D5	VIO_D4	VIO_D3	VIO_D2
011b	SCK2	RxD2	TxD2	SSIBCK0	SSILRCK0	SSITxD0	SSIRxD0
100b	POE4	POE8	POE10	MTIOC0A	MTIOC0B	MTIOC0C	MTIOC0D
101b	—	VBUSIN1	—	—	—	CC1_Rd1	CC2_Rd1
110b	—	IRQ1	—	—	—	—	—
111b	REF50CK0	RMII0_RXD0	RMII0_RXD1	RMII0_RXER	RMII0_CRS_DV	—	—

The code would then look like this:

```
pfc_set_pin_function(PE, 2, 3);      /* PE_2 = TxD2 */
```

4. u-boot: Programming u-boot into QSPI Flash

Segger has added support for many SPI flash devices. If you are using the latest version of Segger J-Link software, then the following command will program the SPI flash without an issue.

For RZ/A1

```
$ ./build.sh jlink u-boot 0x18000000
```

For RZ/A2

```
$ ./build.sh jlink u-boot 0x20000000
```

5. u-boot: Customizing Kernel Boot Commands

In order to boot a kernel, you must first copy the Device Tree binary image to RAM, then pass that location to the kernel as you start it.

Additionally, when using a traditional (non-XIP) kernel, the compressed kernel image (uImage) must also be copied from Flash to RAM and decompressed before kernel booting can begin.

Please review the file board/my_company/toaster/toaster.c that was created for by the add_new_board.sh script.

The function **board_late_init**( ) manually creates the booting macros (xa_boot, xsa_boot, etc…) in source code using the setenv( ) function.

Please review the #define settings found in the **board_late_init**( ) function to match the design of your board.

6. u-boot: Initial Debugging using RAM Only (not Flash)

If you have attempted to program the SPI Flash (or NOR Flash) with the Segger J-Link and it did not work (unsupported SPI flash), then you should consider starting out with just trying to load u-boot into RAM with the JTAG. Then running it from there and getting to a serial prompt (no Flash involved). Then, you could start to confirm pin setup and figure out the SPI flash programming and such.

These instructions walk you through using a Segger J-Link.

6.1 Change Base Address

You will need to change the base address of where your code will run from (from the default address to the RAM address). You would do that in your **include/configs/toaster.h** file (remember, your filename will be whatever you choose from the first step). You will want to change the CONFIG_SYS_TEXT_BASE setting to 0x20020000 (for RZ/A1) and rebuild.

In the header file for your board, simply change the “BOOT_MODE” setting to 3 as seen below.

```
 #define BOOT_MODE 3     /* << MAKE YOUR SELECTION HERE >> */
```

Side Note If you are wondering why the RAM address 0x20020000 instead of 0x20000000, look at System Control Register 3 (SYSCR3) which says that the Data Retention RAM areas are read-only after reset until you enable them in code (which we do in the file lowlevel_init.S immediately after RESET). However, when you are trying to download your code using the J-Link, that code has not run yet so the RAM at address 0x20000000 is Read-Only, meaning you can’t download to it. Therefore, we’ll use address 0x20020000.

Note, the same thing exists for RZ/A2. The internal RAM address 0x80020000 should be used.

6.2 Downloading (to RAM) and Running using J-LinkExe

With your board powered and your J-Link plugged in, start up JLink Commander ('JLinkExe' in Linux) in the directory that has your u-boot.bin binary.

6.2.1 RZ/A1 Users:

```
$ cd output/u-boot-2017.05
$ JLinkExe -speed 12000 -jtagconf -1,-1 -if JTAG -device R7S721001
```

Next, from within the JLink window, issue commands to reset the device (rx 100), download your code (loadbin), set the PC to your code (setpc), then start it running (g), then exit Jlink (exit).

```
J-Link> rx 100
J-Link> loadbin u-boot.bin,0x20020000
J-Link> setpc 0x20020000
J-Link> g
J-Link> exit
```

Hopefully you'll get a serial prompt at that point.

6.2.2 RZ/A2 Users:

```
$ cd output/u-boot-2017.05
$ JLinkExe -speed 12000 -jtagconf -1,-1 -if JTAG -device R7S921053VCBG
```

Next, from within the JLink window, issue commands to reset the device (rx 100), download your code (loadbin), set the PC to your code (setpc), then start it running (g), then exit Jlink (exit).

```
J-Link> rx 100
J-Link> loadbin u-boot.bin,0x80020000
J-Link> setpc 0x80020000
J-Link> g
J-Link> exit
```

Hopefully you'll get a serial prompt at that point.

6.3 Downloading (to RAM) and Running using J-Link and GDB

NOTE: The addresses in these instructions are specific to RZ/A1 only

These instructions are for the RAM-based u-boot.

If you need to debug u-boot at this stage, you can use GDB. This sections assumes you have already read the application note "GDB Debugging for RZA1 Linux with J-Link v1.00".

Use the following commands to start debugging with the RAM based u-boot.

The 'monitor' command in GDB basically lets you send commands directly to the J-Link, so we can use the loadbin J-Link custom command from within GDB. Note that you have to use the full path to your u-boot.bin.

```
(gdb) target remote localhost:2331
(gdb) monitor reset
(gdb) file u-boot
(gdb) monitor loadbin ~/rza_linux-4.9_bsp/output/u-boot-2017.05/u-
boot.bin,0x20020000
(gdb) set $pc = 0x20020000
(gdb) si
(gdb) si

    etc...
```

If you are doing a lot of rebuilding of u-boot and running, you could make a macro that you put inside your local .gdbinit file and will take care of loading u-boot and setting up all the addresses and such. For example, you could put the following in your .gdbinit file, and then if you build a new u-boot, you simply just type “ramload” in GDB and it will reload it for you.

```
define ramload
        #Get rid of any existing breakpoints
        delete

        #Load in our file/binary
        #NOTE: that you need the full path of your file!!
        monitor reset
        file u-boot
        monitor loadbin /home/renesas/u-boot-2017.05/u-boot.bin,0x20020000

        #Run the code till right before the relocation
        set $pc = 0x20020000
        tb relocate_code
        c
        shell sleep 1

        #Figure out where it will be copying to, then reload symbol file
        set $i = gd->relocaddr
        print $i
        symbol-file
        add-symbol-file u-boot $i

        #Set a 1-time breakpoint at board_init_r()
        tb board_init_r
        c

        #At the end, you should be in the board_init_r function.
        #Now you can set breakpoint or whatever you want, or continue to run
end
```

7. u-boot: Adding QSPI Support

7.1 Add Support for your SPI Flash

When you first try to connect to your SPI flash, you might get a message like this:

```
=> sf probe 0
SF: Unsupported flash IDs: manuf 20, jedec ba20, ext_jedec 1000
Failed to initialize SPI flash at 0:0
```

This means you need to add the manufacture of your SPI flash device.

You can look up what the SPI flash devices are supported and match up yours by looking at file:

`u-boot-2017.05/drivers/mtd/spi/spi_flash_ids.c`

For the example above you will see that line:

```
{"n25q512",         INFO(0x20ba20, 0x0,  64 * 1024,  1024, RD_FULL | WR_QPP | E_FSR | SECT_4K) },
```

is only included if CONFIG_SPI_FLASH_STMICRO is defined.

Therefore, you simply add that Flash manufacture to your **include/configs/toaster.h** file

You can check file

u-boot-2017.05/drivers/mtd/spi/spi_flash_ids.c

For example:

```
/* SPI Flash Device Selection */
/* Enabled using menuconfig:
 * Device Drivers > SPI Flash Support > Legacy SPI Flash Interface support
 */
#define CONFIG_SPI_FLASH_SPANSION
#define CONFIG_SPI_FLASH_STMICRO
#define CONFIG_SPI_FLASH_BAR        /* For SPI Flash bigger than 16MB */
```

Now, after you rebuild and download again, you should see something like this:

```
=> sf probe 0
SF: Detected N25Q512 with page size 256 Bytes, erase size 4 KiB, total 64 MiB
```

7.2 u-boot: QSPI XIP Support

If plan on using the QSPI in XIP mode for Linux, then you will need to make sure the commands and device settings are correct. In file **cmd/qspi.c**, you will see a function **do_qspi**( ). Please review that function and confirm the setup will work for the SPI flash device you have selected.

In the BSP, a Spansion, Micron and Macronix device have been tested. If you are not using one of those device, hopefully they are similar enough where you only need to make minor modifications. Many device manufactures follow the same conventions as the Spansion devices on the RSK board. Micron devices tend to be different, so they have been included in the BSP as an example.

8. u-boot: Programming QSPI Flash

This section only applies if you were unable to program the SPI flash directly with the Segger JLink because you are using a SPI flash not yet supported by Segger.

8.1 Using ‘RAM-based’ u-boot to program QSPI flash

NOTE: The addresses in these instructions are specific to RZ/A1 only

If you have successfully confirmed that your RAM-based u-boot can erase and program SPI flash memory, then you can use it to program the real Flash-based u-boot into SPI flash.

First, save your working RAM based u-boot.bin

```
$ cp u-boot.bin u-boot-ram.bin
```

Now edit your configuration file and put your CONFIG_SYS_TEXT_BASE setting back to 0x18000000 by changing BOOT_MODE back to ‘1’ and re-build.

Once again, connect with J-Link and download your RAM-based u-boot and get it up and running. However, this time, do not exit the JLinkExe program, but instead download the Flash based u-boot.bin to RAM as well. It’s OK to use the low RAM address (0x20000000) because by this point, u-boot has already relocated itself into the upper part of your memory so there will not be any conflict. Also, the u-boot will have already enabled the RAM below address 0x20020000.

Then you can use the serial flash commands to erase the write the Flash-based u-boot into SPI flash.

For example:

```
$ cd u-boot-2017.05
$ JLinkExe -speed 12000 -if JTAG -jtagconf -1,-1 -device R7S721001
```

Next, from within JLinkEXE window:

```
J-Link> rx 100
J-Link> loadbin u-boot-ram.bin,0x20020000
J-Link> setpc 0x20020000
J-Link> g
```

Confirm that u-boot comes up in your serial terminal. Then, go back to your JLinkExe session and download you Flash-based u-boot to RAM. You will need to GO ('g') command because JLink automatically halts the CPU when a loadbin command is used.

```
J-Link> loadbin u-boot.bin,0x20000000
J-Link> g
J-Link> exit
```

Now, in your u-boot console, program in your u-boot.

```
=> sf probe 0
SF: Detected N25Q512 with page size 256 Bytes, erase size 4 KiB, total 64 MiB
=> sf erase 0 80000
SF: 393216 bytes @ 0x0 Erased: OK
=> sf write 20000000 0 80000
SF: 393216 bytes @ 0x0 Written: OK
=> md 18000000
18000000: ea0000be e59ff014 e59ff014 e59ff014    ................
18000010: e59ff014 e59ff014 e59ff014 e59ff014    ................
18000020: 18000060 180000c0 18000120 18000180    `....... .......
18000030: 180001e0 18000240 180002a0 deadbeef    ....@...........
18000040: 0badc0de e320f000 e320f000 e320f000    ...... ... ... .
18000050: e320f000 e320f000 e320f000 e320f000    .. ... ... ... .
18000060: e51fd028 e58de000 e14fe000 e58de004    (.........O.....
18000070: e3a0d013 e169f00d e1a0e00f e1b0f00e    ......i.........
18000080: e24dd048 e88d1fff e51f2050 e892000c    H.M.....P ......
18000090: e28d0048 e28d5034 e1a0100e e885000f    H...4P..........
180000a0: e1a0000d eb0002c7 e320f000 e320f000    .......... ... .
180000b0: e320f000 e320f000 e320f000 e320f000    .. ... ... ... .
180000c0: e51fd088 e58de000 e14fe000 e58de004    ..........O.....
180000d0: e3a0d013 e169f00d e1a0e00f e1b0f00e    ......i.........
180000e0: e24dd048 e88d1fff e51f20b0 e892000c    H.M..... ......
180000f0: e28d0048 e28d5034 e1a0100e e885000f    H...4P..........
=>
```

HINT: If you are using external SDRAM, then after the RAM-u-boot runs, your SDRAM has been setup. Therefore, you could use that address to download your binary file to. This is helpful when you start to program in bigger binaries like the Linux kernel and root file system.

8.2 Automating Programing using a JLink Script

NOTE: The addresses in these instructions are specific to RZ/A1 only

You can automate the process of programming the SPI flash by creating a script file that you pass to JLinkEXE.

First create a script file called **load_spi_uboot.txt** in the directory where your u-boot.bin file is located.

```
$ cd u-boot-2017.05
$ gedit load_spi_uboot.txt
```

Contents of load_spi_uboot.txt

```
rx 100
loadbin u-boot-ram.bin,0x20020000
setpc 0x20020000
g
Sleep 2000
loadbin u-boot.bin,0x20000000
g
exit
```

Now run your script:

```
$ JLinkExe -speed 12000 -if JTAG -jtagconf -1,-1 -device R7S721001 -CommanderScript
load_spi_uboot.txt
```

After your script is complete, your RAM-u-boot should be running and your new image to be programmed should be at the beginning of RAM. Now, you just need to program it in:

```
=> sf probe 0 ; sf erase 0 80000 ; sf write 20000000 0 80000
SF: Detected N25Q512 with page size 256 Bytes, erase size 4 KiB, total 64 MiB
SF: 524288 bytes @ 0x0 Erased: OK
SF: 524288 bytes @ 0x0 Written: OK
=>
```

9. Kernel: Converting RZ/A BSP to your own platform

We suggest creating your own files specifically for your board (your own BSP). Do not simply start hacking the Renesas board files as there will be code and options in there that will probably not apply to your board. Also, this will allow you to more easily receive code updates from the Renesas repositories.

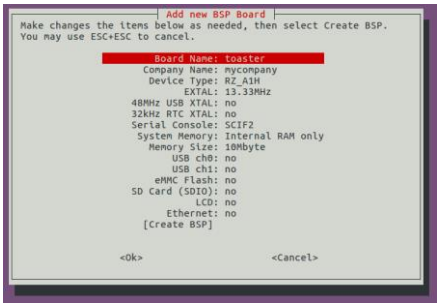
For all of these examples we'll assume a board name of "**toaster**". Therefore, anywhere you see that name, you can replace it with whatever your real board name is.

9.1 Run the add_new_board.sh script

Located in the base of the linux-4.xx directory is a script that will assist you in creating a BSP for your custom board.

Simply run the script which will display an interface where you can select what components are on your board.

```
$ cd output/linux-4.xx
$ ./add_new_board.sh
```



Use the arrow keys and Enter key to make your selections. Once all selections have been chosen, select the last option "[Create BSP]" and press Enter. This will then create new directories and files based on your selections as well as add your board configuration the existing Makefile build system.

Please use the same "Board Name" and "Company Name" that you used for u-boot

9.2 Device Tree for your board

In the kernel, device configurations are moving away from defining your devices in code (usually in your *board* file) and instead defining what devices are in your system in what is called a Device Tree file that is passed the kernel at boot time.

The add_new_board.sh script will have created a file for your board and already made some modifications based on your selections.

Please review it further. You will find it as:

RZ/A1: arch/arm/boot/dts/r7s72100-toaster.dts

RZ/A2: arch/arm/boot/dts/r7s9210-toaster.dts

This new Device Tree will have also been added to the Makefile build system and will be compiled automatically with the other Renesas Device Trees files

```
$ ./build.sh kernel dtbs
```

9.3 I/O Pin Configurations

While some pins may have been setup in u-boot, those were probably only the minimum functionality in order to allow you to boot the kernel. The rest of the pin setup is done in the Device Tree for your board.

NOTE: The Device Tree file will not compile until you have edited the pin configurations because the character '?' has been used where you need to enter real values. Please replace the '?' characters with real values.

The reason is that question mark characters ("?") were used for the pin configurations instead of real values in order to force you to review your schematic and hardware manual in order to select the correct pin combination. Not doing this step first (and correctly) is the most common problem customers run into. Therefore, to save yourself hours of debugging issues that could have been easily avoided, we are forcing you to do it first and get it out of the way. Yes, it is a boring job….but it needs to get done.

See section 10.1.2 Configuring I/O Pins for more information about Device Tree syntax.

9.4 The "Board" file

Even though device configurations are now in the device tree file, a "board" file was still created by the add_new_board.sh script. This was because it still serves as a good place to put customer code for your board for things like a LED heartbeat or custom startup code.

You can find your board file here:

$ arch/arm/mach-shmobile/board-toaster.c

Note, you can also do some pin configurations manually in your board file in case you have to do something too tricky for the Device Tree.

9.5 Default Configuration Files

As part of the add_new_board.sh script, 2 default kernel configuration files were created based on your selections.

They were:

```
arch/arm/configs/toaster_defconfig
arch/arm/configs/toaster_xip_defconfig
```

If you make changes to your kernel configuration and wish to save them, you can use the following command in the BSP

```
$ ./build.sh kernel savedefconfig
```

This will create a file output/linux-4.xx/**defconfig** which you can then rename and place under arch/arm/configs/

9.6 Building your BSP

To build the kernel for your board, use the following commands:

For a XIP kernel:

```
$ ./build.sh kernel toaster_xip_defconfig
$ ./build.sh kernel xipImage
$ ./build.sh kernel dtbs
```

For a non-XIP kernel:

```
$ ./build.sh kernel toaster_defconfig
$ ./build.sh kernel uImage
$ ./build.sh kernel dtbs
```

NOTE: The Device Tree file will not compile until you have edited the pin configurations.

10. Kernel: Device Tree Configurations

In the kernel, device configurations are moving away from defining your devices in code (usually in your *board* file) and instead defining what devices are in your system in what is called a Device Tree file that is passed the kernel at boot time.

10.1 General Overview of the RZ/A1 Device Tree

The Linux-4.x BSP contains a file that describes almost all the available drivers for the RZ/A1. That file is

```
linux-4.x/arch/arm/boot/dts/r7s72100.dtsi
```

That file is #included in your boards individual Device Tree file (.dts).

10.1.1 Enabling Drivers and Devices

In the r7s72100.dtsi and r7s9210.dtsi files, you will notice that almost all the drivers are set to “disabled”.

```
        i2c3: i2c@fcfeec00 {
                #address-cells = <1>;
                #size-cells = <0>;
                compatible = "renesas,riic-r7s72100", "renesas,riic-rz";
                reg = <0xfcfeec00 0x44>;
                interrupts = <GIC_SPI 181 IRQ_TYPE_LEVEL_HIGH>,
                             <GIC_SPI 182 IRQ_TYPE_EDGE_RISING>,
                             <GIC_SPI 183 IRQ_TYPE_EDGE_RISING>,
                             <GIC_SPI 184 IRQ_TYPE_LEVEL_HIGH>,
                             <GIC_SPI 185 IRQ_TYPE_LEVEL_HIGH>,
                             <GIC_SPI 186 IRQ_TYPE_LEVEL_HIGH>,
                             <GIC_SPI 187 IRQ_TYPE_LEVEL_HIGH>,
                             <GIC_SPI 188 IRQ_TYPE_LEVEL_HIGH>;
                clocks = <&mstp9_clks R7S72100_CLK_I2C3>;
                clock-frequency = <100000>;
                power-domains = <&cpg_clocks>;
                status = "disabled";
        };
```

The intention is that in order to select a driver, you simple override the status value and change it to “okay” in your board specific .dts file (arch/arm/boot/dts/r7s72100-toaster.dts or arch/arm/boot/dts/r7s9210-toaster.dts).

```
&i2c3 {
        pinctrl-names = "default";
        pinctrl-0 = <&i2c3_pins>;
        status = "okay";
        clock-frequency = <100000>;
};
```

10.1.2 Configuring I/O Pins

Note that you also need to set up your I/O pins correctly for I2C functionality in the “pinctrl” section.

The RZA1_PINMUX and RZA2_PINMUX macro takes 3 parameters: The port number, The pin number, and the “Alternative Mode” number. Please refer to section “Ports” in the hardware manual and use the “Pin Function” tables in order to determine the correct Alternative Mode number for each pin.

Table 41.13 Pin Function (P1)

Port Mode		Alternative Mode							
		1st Alternative		2nd Alternative		3rd Alternative		4th Alternative	
Input	Output	Input	Output	Input	Output	Input	Output	Input	Output
P1_0	P1_0	RIIC0SCL		IRQ4		ET_RXD0		DV0_DATA0	
P1_1	P1_1	RIIC0SDA		IRQ5		ET_RXD1		DV0_DATA1	
P1_2	P1_2	RIIC1SCL		IRQ6		ET_RXD2		DV0_DATA2	
P1_3	P1_3	RIIC1SDA		IRQ7		ET_RXD3		DV0_DATA3	
P1_4	P1_4	RIIC2SCL		IRQ0		DREQ0		VIO_D0	
P1_5	P1_5	RIIC2SDA		IRQ1				VIO_D1	
P1_6	P1_6	RIIC3SCL		IRQ2		SSIRxD0		VIO_D2	
P1_7	P1_7	RIIC3SDA		IRQ3		RxD2		VIO_D3	

```
&pinctrl {

        /* RIIC ch3 */
        i2c3_pins: i2c3 {
                pinmux = <RZA1_PINMUX(1, 6, 1)>,       /* P1_6 (ALT1) = RIIC3SCL */
                         <RZA1_PINMUX(1, 7, 1)>;       /* P1_7 (ALT1) = RIIC3SDA */
        };
};
```

NOTE: For **RZ/A2**, you must use the PORTx macros for defining the poart (ie, “PORT3” instead of just “3”. For example:

```
&pinctrl {

        /* RIIC ch2 */
        i2c2_pins: i2c2 {
                pinmux = <RZA2_PINMUX(PORTD, 4, 1)>, /* PD_4 (ALT1) = RIIC2SCL */
                         <RZA2_PINMUX(PORTD, 5, 1)>; /* P1_7 (ALT1) = RIIC2SDA */
        };
};
```

10.2 Sample Device Tree Configurations

When testing the BSP for the RZ/A1H RSK board, various device tree configurations were used. For example, using an RSPI channel connected to a serial SPI flash to be mounted as a JFFS2 file system.

Please refer to these files for examples of common device configurations.

```
linux-4.x/arch/arm/boot/dts/r7s72100-rskrza1.dts
linux-4.x/arch/arm/boot/dts/r7s72100-rskrza1_testing.dts

linux-4.x/arch/arm/boot/dts/r7s9210-rza2mevb.dts
```

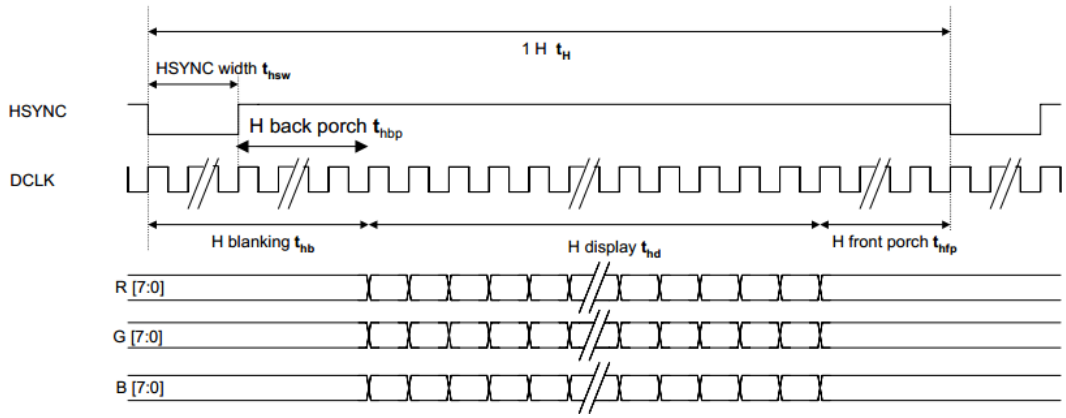
10.3 LCD Signal Timing

The signal timing can be a little tricky for someone new to LCDs, so we'll explain how to enter your LCD information here.

The Device Tree is where you will specify all the timings for your specific LCD.

10.3.1 General LCD Timing

As a general review of LCDs:



HSYNC is a signal pulse (LOW pulse) that starts each line.

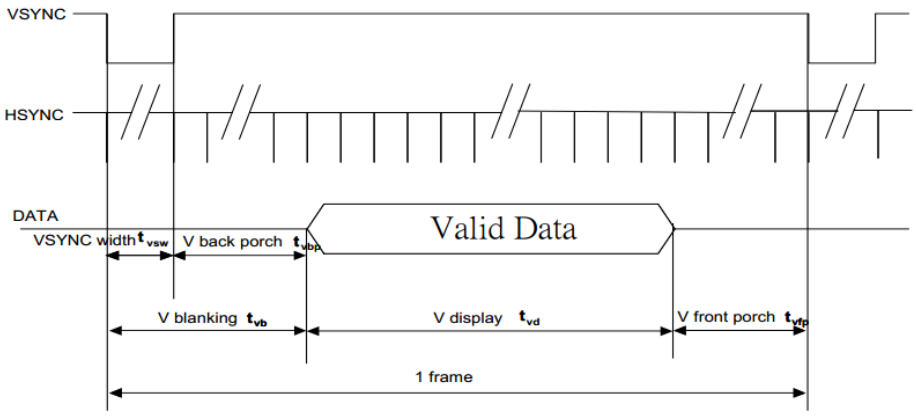
HSYNC width (length): The time (in dot clocks) when HSYNC goes LOW, to when it goes back HIGH.

Horizontal Back Porch: The time (in dot clocks) from when HSYNC goes HIGH to the beginning of valid pixels being transmitted.

Horizontal Front Porch: The time (in dot clocks) from the end of valid pixels being transmitted to HSYNC going LOW (start of a new HSYNC pulse).

Horizontal Blanking: The total time (in dot clocks) for each line from the start of HSYNC until valid pixels start transmitting (HSYNC width + H Back Porch)

Horizontal Line Period: The total number of dot clocks for 1 line. 'HSYNC width' + 'H back porch' + pixel data + 'H Front Porch'. Or, 'Horizontal Blanking' + pixel data + 'H Front Porch'.



VSYNC is a signal pulse (LOW pulse) that starts each frame.

VSYNC width (length): The number of horizontal lines from when VSYNC goes LOW, to when it goes back HIGH.

Vertical Back Porch: The number of horizontal lines from when VSYNC goes HIGH to the beginning of the first line of the LCD is transmitted.

Vertical Front Porch: The number of horizontal lines from the end of the last line of the image is transmitted to

VSYNC going LOW (start of a new VSYNC pulse).

Vertical Blanking: The number of horizontal lines for each frame from the start of VSYNC until valid pixels start transmitting (VSYNC width + V Back Porch)

Vertical Line Period: The number of horizontal lines for full image. 'VSYNC width' + 'V back porch' + pixel data + 'V Front Porch'. Or, 'Vertical Blanking' + pixel data + V Front Porch'.

You must enter the correct number for **Back Porch** (Horizontal and Vertical). However, many LCD specifications only show Horizontal and Vertical **Blanking**. Therefore, if a value for Back Porch is not shown, simple subtract the 'SYNC width' from the total 'Blanking'.

For example:

Item	Symbol	Value			Unit	Note
		Min.	Typ.	Max.		
Horizontal Display Area	thd	-	800	-	DCLK	
DCLK Frequency	fclk	26.4	33.3	46.8	MHz	
One Horizontal Line	th	862	1056	1200	DCLK	
HS pulse width	thpw	1	-	40	DCLK	
HS Blanking	thb	46	46	46	DCLK	
HS Front Porch	thfp	16	210	354	DCLK	

Item	Symbol	Value			Unit	Note
		Min.	Typ.	Max.		
Vertical Display Area	tvd	-	480	-	TH	
VS period time	tv	510	525	650	TH	
VS pulse width	tvpw	1	-	20	TH	
VS Blanking	tvb	23	23	23	TH	
VS Front Porch	tvfp	7	22	147	TH	

This LCD does not specify a Horizontal or Vertical Back Porch. But, it does specify a fixed HS and VS Blanking. Also, it gives you a range for the HSYNC pulse with and VSYNC pulse with. So what we do is pick a number for HSYNC and VSYNC width, then subtract that from Blanking to get our Back Porch.

If we pick '20' for our 'HS pulse width', then our Horizontal back porch will be 46 - 20 = 26. And if we pick 10 for our 'VS pulse width', then our Vertical back porch will be 23 – 10 = 13; Then in our Device Tree we would enter:

```
hback-porch = <26>;          /* back porch = 'HS Blanking'(46) - hsync-len(20)  */
hfront-porch = <210>;        /* 'HS Front Porch (210)' */
vback-porch = <13>;          /* back porch = 'VS Blanking'(23) - vsync-len(10)  */
vfront-porch = <22>;         /* 'VS Front Porch(22)'
hsync-len = <20>;            /* pulse width of HSYNC (min=1, max=40) */
vsync-len = <10>;            /* pulse width of VSYNC (min=1, max=20) */
```

Below are the descriptions for what each of the entries mean in the Device Tree.

10.3.2 RAM Buffer Settings

LCD Frame Buffer RAM

fb_phys_addr : The hard coded physical address (not virtual) of the LCD Frame buffer. If you set this to '0', memory will be allocated on boot from general system memory.

> **NOTE:** This address must always be in internal RAM on the RZ/A. If using external SDRAM, do not set this setting to '0' because you cannot have an LCD frame buffer in external SDRAM.
>
> **NOTE:** (RZ/A1 only) When selecting the physical address, please use the 'mirror' address range starting at address **0x60000000** as this is a non-cached access area to RAM. If you use the normal 0x20000000 address, this is a cached access area to RAM and you might need to flush your data after any modifications to your video frame buffer. Therefore, using the non-cached area is easier for your application to use.

fb_phys_size : The size of the buffer in bytes.

10.3.3 LCD Panel Resolution and Timing

Native Panel Resolutions

These are the actually pixel resolutions of the LCD. The reason these are separate setting from the ‘dispaly0’ settings below is so that you can set a display resolution setting smaller than the actual physical LCD. For example, you only want a QVGA display on a VGA sized LCD.

panel_pixel_xres : The panel’s native x (width) resolution.

panel_pixel_yres : The panel’s native y (height) resolution.

display0: display0

bits-per-pixel : The bits-per-pixel that you want for your LCD frame buffer (as seen by the application)

bus-width : This is the output width of the LCD. Usually “24”

display-timings

hactive : The horizontal (x) width of the screen. You can set this to less than the LCD panel if you want.

vactive : The vertical (y) height of the screen. You can set this to less than the LCD panel if you want.

hback-porch : This is the **Horizontal Back Porch** value from the LCD specification. This value is in dot clocks. If the LCD spec only provides a timing for **Horizontal Blanking**, then please break the number up between ‘hback-porch’ and ‘hsync-len’.

hfront-porch : This is the **Horizontal Front Porch** value from the LCD specification. This value is in dot clocks.

vback-porch : This is the **Vertical Back Porch** value from the LCD specification. This value is in lines.

vfront-porch : This is the **Vertical Front Porch** value from the LCD specification. This value is in lines.

hsync-len : This is the **HSYNC pulse width** (length) of the HSYNC signal before the start of each line. This value is in dot clocks. This value is used to generate the HSYNC signal on a TCON pin. Do not set this to 0, otherwise a HSYNC signal will not be output.

vsync-len : This is the **VSYNC pulse width** (length) of the VSYNC signal before the start of each frame. This value is in lines. This value is used to generate the VSYNC signal on a TCON pin. Do not set this to 0, otherwise a VSYNC signal will not be output.

hsync-active : HSYNC pulse is: 0=Active LOW, 1=Active HIGH

vsync-active : VSYNC pulse is: 0=Active LOW, 1=Active HIGH

de-active : Data Enable signal is: 0=Active LOW , 1= Active HIGH

pixelclk-active : Pixel Data clock polarity: 0=drive pixel data on falling edge and sample data on rising edge, 1=drive pixel data on rising edge and sample data on falling edge

NOTE: For more information on Device Tree settings related to video, see here in the kernel source code: Documentation/devicetree/bindings/video/display-timing.txt

10.3.4 Clock Source, Format and LVDS Settings

Please set the following properties accordingly.

```
/* See 'drivers/video/fbdev/renesas/vdc5fb.h' for valid choices
* for panel_icksel, panel_ocksel, and out_format */
panel_icksel = <3>;     /* 3=ICKSEL_P1CLK (Peripheral clock 1) */
panel_ocksel = <0>;     /* (don't care when lvds=0 ) */
out_format = <0>;       /* 0=OUT_FORMAT_RGB888 */
use_lvds = <0>;         /* Set to 0 or 1 */
```

panel_icksel = <?>;

When .use_lvds=0:

This driver only supports ICKSEL_P1CLK because the math to calculate PANEL_DCDR[5:0] assumes a P1 clock rate.

When .use_lvds=1

This value is not used (PANEL_ICKSEL[1:0] is don't care when using LVDS).

See "Panel Clock Control Register (SYSCNT_PANEL_CLK)" for more details.

Valid values for panel_icksel:

0 (ICKSEL_INPSEL)	Video image clock (VIDEO_X1 or DV_CLK based on INP_SEL=0,1)
1 (ICKSEL_EXTCLK0)	External clock (LCD0_EXTCLK)
2 (ICKSEL_EXTCLK1)	External clock (LCD1_EXTCLK)
3 (ICKSEL_P1CLK)	Peripheral clock 1

panel_ocksel = <?>;

When .use_lvds=0, only ‘0’ (OCKSEL_ICK) should be selected

When .use_lvds=1, ‘1’ (OCKSEL_PLL) or ‘2’ (OCKSEL_PLL_DIV7) should be selected

See "Panel Clock Control Register (SYSCNT_PANEL_CLK)" for more details */

Valid values for panel_ocksel:

0 (OCKSEL_ICK)	Selected by PANEL_CLK[1:0]
1 (OCKSEL_PLL)	LVDS PLL clock
2 (OCKSEL_PLL_DIV7)	LVDS PLL clock divided by 7

out_format = <?>;

This property sets the output format of the LCD controller as it is connected directly to the LCD panel. Note that this is physical interface to the LCD panel and not the video frame buffer. Meaning, your LCD panel might have a physical RGB888 signal connection, but you can select a RGB565 or YCbCr video frame buffer for application to use (the VDC5 hardware will do the format conversion automatically in real time while displaying the image)

Valid Values for out_format:

0	RGB888
1	RGB666
2	RGB565

use_lvds = <?>;

Specifies if you would like to use the LVDS output instead of Parallel output.

Note that the TCONs are internally multiplexed when using LVDS, so please remember to configure the "tcons_sel" section as well.

Valid values for use_lvds:

0	Use standard parallel output
1	Use LVDS output

10.3.5 TCON Pins

There are 7 "TCON" pins. These pins are uses as the control signals (HSYNC, VSYNC, etc...). You can choose any functionality for any of the TCON pins.

The numbers are assigned as follows:

0 = VSYNC

1 = VE (Vertical Enable)

2 = HSYNC

3 = HE (Horizontal Enable)

4 = CPV/CGK (Gate clock signal)

5 = POLA (VCOM voltage polarity control signal)

6 = POLB (VCOM voltage polarity control signal)

7 = DE (Data Enable)

0xFF = pin not used

Here is an example of how the pins are used on the RZ/A1H RSK board.

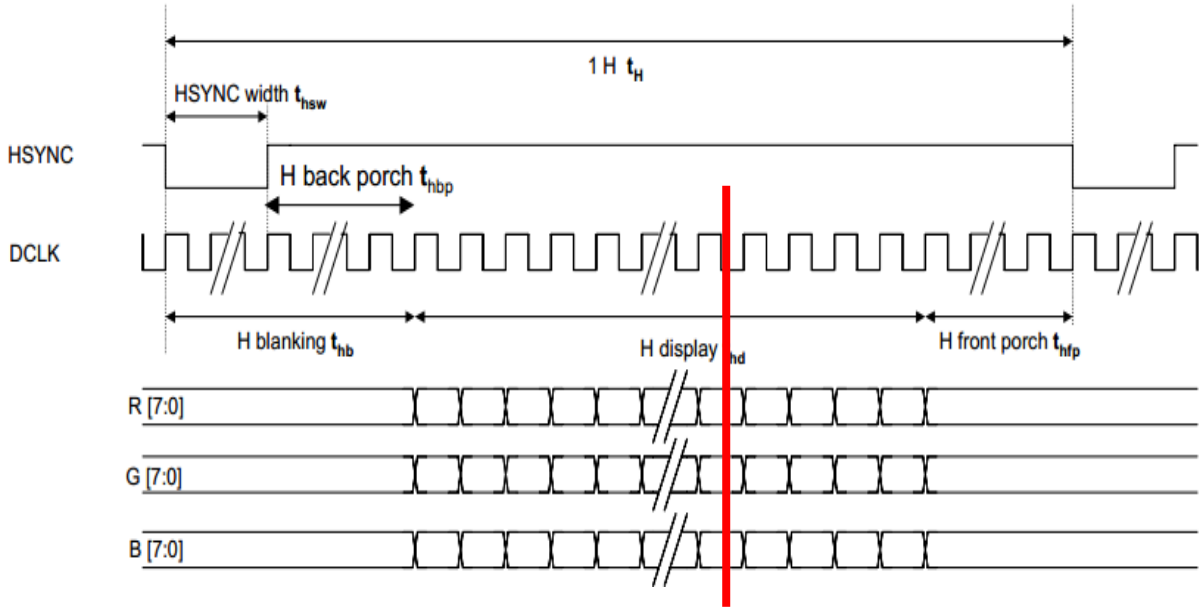
```
/* How are the TCON pins going to be used */
tcon_sel = <
        0xff    /* TCON0: 0xff=TCON_SEL_UNUSED */
        0xff    /* TCON1: 0xff=TCON_SEL_UNUSED */
        7       /* TCON2:    7=TCON_SEL_DE (DATA ENABLE) */
        2       /* TCON3:    2=TCON_SEL_STH (HSYNC) */
        0       /* TCON4:    0=TCON_SEL_STVA (VSYNC) */
        0xff    /* TCON5: 0xff=TCON_SEL_UNUSED */
        0xff    /* TCON6: 0xff=TCON_SEL_UNUSED */
        >;
```

10.3.6 Output Phase Control for LCD_DATAx Pin

Output Phase Control for LCD_DATA23 to LCD_DATA0 is specified by OUTCNT_LCD_EDGE bit of OUT_CLK_PHASE register.

OUT_CLK_PHASE	OUTCNT_LCD_EDGE	0	Output Phase Control of LCD_DATA23 to LCD_DATA0 Pin 0: Output at the rising edge of LCD_CLK pin 1: Output at the falling edge of LCD_CLK pin

The setting of this bit should depend on the specification of LCD to be used. For example, when the spec of LCD is as follows, OUTCNT_EDGE should be 0. This is because LCD latches data at the falling edge of DCLK (please see the red colored line in the following figure) and therefore, VDC5 should be configured so that it outputs the data at rising edge conversely.



Please set the **pixelclk-active** property correctly in the device tree.

pixelclk-active : Pixel Data clock polarity: 0=drive pixel data on falling edge and sample data on rising edge, 1=drive pixel data on rising edge and sample data on falling edge

11. Misc: Installing J-Link software in Linux

To install the Segger Jlink drivers and utilities for Linux do the following.

For example, for a 64-bit Ubuntu install, you would download "J-Link Software and Documentation pack for Linux, DEB Installer, 64-bit" from

https://www.segger.com/downloads/jlink/#J-LinkSoftwareAndDocumentationPack

In a command window, install the downloaded file:

```
$ sudo dpkg -i JLink_Linux_V622_x86_64.deb
```

After the install, on a command line you can type "JLink" then hit the 'tab' key on your keyboard to see all the utilities it installed.

12. FAQ: Why the name SH-Mobile?

- Why the name SH-Mobile?

You may have noticed that the directory that this device is under the ARM directory is called SH-Mobile. This is because over the years, Renesas original supported SH4A and SH-Mobile devices in the kernel under the arch/sh directory. However, Renesas started making devices which had both SH4A and ARM cores in them, so support for those boards needed to be under the ARM tree when you wanted to build a kernel for the ARM core. Eventually, the SH4A cores were removed and only the ARM core remained, but almost all the peripheral drivers were still the same as the SH4A and SH-Mobile devices, hence the Ethernet driver is called sh-eth. Basically, you were building an ARM device with SH4A family peripherals. Renesas does have a plan to eventually rename the mach-shmobile directory under the ARM tree to something more generic like “mach-renesas”, but that will take some time to happen.

Revision Record

		Description	
Rev.	**Date**	**Page**	**Summary**
1.00	Apr 26, 2019	—	First edition issued