RZ/A1H Group

Software Package

Introduction

This Application Note describes the operation of the software package created for the Renesas Starter Kit + for RZ/A1H. This document provides a comprehensive overview of the system and its applications. For further details about the software package please refer to the accompanying software.

This document assumes that the user has gone through the Quick Start Guide for the RZ/A1H and is equipped with:

- Renesas Starter Kit + for RZ/A1H Platform
- Renesas Starter Kit + for RZ/A1H TFT LCD Screen
- J-Link Debugger
- USB A to Mini-B Cable
- NTSC Camera
- Okaya LCD PMOD™
- Ethernet Cable and connection to LAN
- USB Keyboard, Mouse and Memory Drive
- USB Function

Target Device

Target Device: RZ/A1H

Target Board: Renesas Starter Kit+ for RZ/A1H (YR0K77210C000BE)

Contents

List of Abbreviations and Acronyms

Abbreviation	Full Form
ADC	Analogue to Digital Converter
AIOCB	Asynchronous I/O Control Block
ANSI	American National Standards Institute
API	Application Programming Interface
ARM	Advanced RISC Machines
ASSP	Application-Specific Standard Product
BSD	Berkeley Software Distribution
CAN	Controller Area Network
CDC	Communications Device Class
CMOS	Complementary Metal-Oxide-Semiconductor
CODEC	COder-DECoder
COM	COMmunications
CPU	Central Processing Unit
CUI	Character User Interface
DHCP	Dynamic Host Configuration Protocol
DMA	Direct Memory Access
DMAC	Direct Memory Access Controller
DOS	Disk Operating System
DTR	Data Terminal Ready
EEPROM	Electrically Erasable Programmable Read-Only Memory
exFAT	Extended File Allocation Table
FAT	File Allocation Table
FIFO	First In First Out
GUI	Graphical User Interface
HID	Human Interface Device
HMI	Human Machine Interface
HTML	Hyper Text Markup Language
Hz	Hertz
IC	Integrated Circuit
ICE	In Circuit Emulator
IIC (or I^2C)	Inter-Integrated Circuit
INTC	INTerrupt Controller
I/O	Input/Output
IP	Internet Protocol
IRQ	Interrupt ReQuest
JTAG	Joint Test Action Group
KHz	KiloHertz
LAN	Local Area Network
LCD	Liquid Crystal Display
LED	Light Emitting Diode
MCU	MicroController Unit
MHz	MegaHertz
NTSC	National Television System Committee
OS	Operating System
OTG	On-The-Go
PC	Personal Computer
PMOD	Peripheral MODule interface is an open standard defined by Digilent Inc™
QE	Quick and Effective tool solutions
QSPI	Quad Serial Peripheral Interface
RAM	Random Access Memory
RIIC	Renesas Inter-Integrated Circuit
ROM	Read Only Memory
RTC	Real Time Clock
RTOS	Real Time Operating System
RTS	Request To Send

SDK	Software Development Kit
SSIF	Serial Sound InterFace
STDIO	Standard Input/Output
TCP	Transmission Control Protocol
TFT	Thin Film Transistor
USB	Universal Serial Bus
VDC5	Video Display Controller 5

Table 1-1 List of Abbreviations and Acronyms

1. Outline of the Software Package

The RZ/A1H Software Package consists of drivers, middleware and applications. The software package is designed so that the user can easily create their own bespoke applications using the Renesas Starter Kit + for RZ/A1H development kit.

The main features of the software package include:

- Using a DHCP Server to create a dynamic webpage.
- Displays images from a NTSC camera onto a TFT Touchscreen.
- USB functionality – Mass Storage, CDC, and HID (keyboard and mouse).
- Sound Interface provides a record and play functionality.
- All the above features are integrated with a FreeRTOS operating system.

1.1 Folder Structure

The software package's folder structure allows for easy navigation of the source code. All source is included in the '`src`' folder.

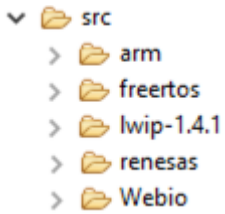

Figure 1-1 'src' Folder

The first level includes third party source `arm`, `FreeRTOS`, `lwip1.4.1`, and `Webio`. All of which are used from the '`renesas`' folder.

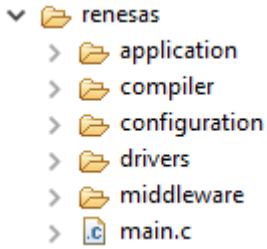

Figure 1-2 'renesas' folder

The `renesas` folder is made up of `application`, `compiler`, `configuration`, `driver` and `middleware` specific modules. These modules make up the system. The `application` folder holds all of the high layer application level. The `compiler` folder holds all of the compiler specifics, allowing for users to easily port the software package between different toolchains. The `configuration` folder holds the configuration for the application, stdio interface, and OS abstraction layer. The `driver` folder holds hardware specific code that control the microcontroller peripherals, whilst `middleware` holds code that is hardware independent.

2. Description of the System

The following image provides an overview of the Renesas Starter Kit + for RZ/A1H

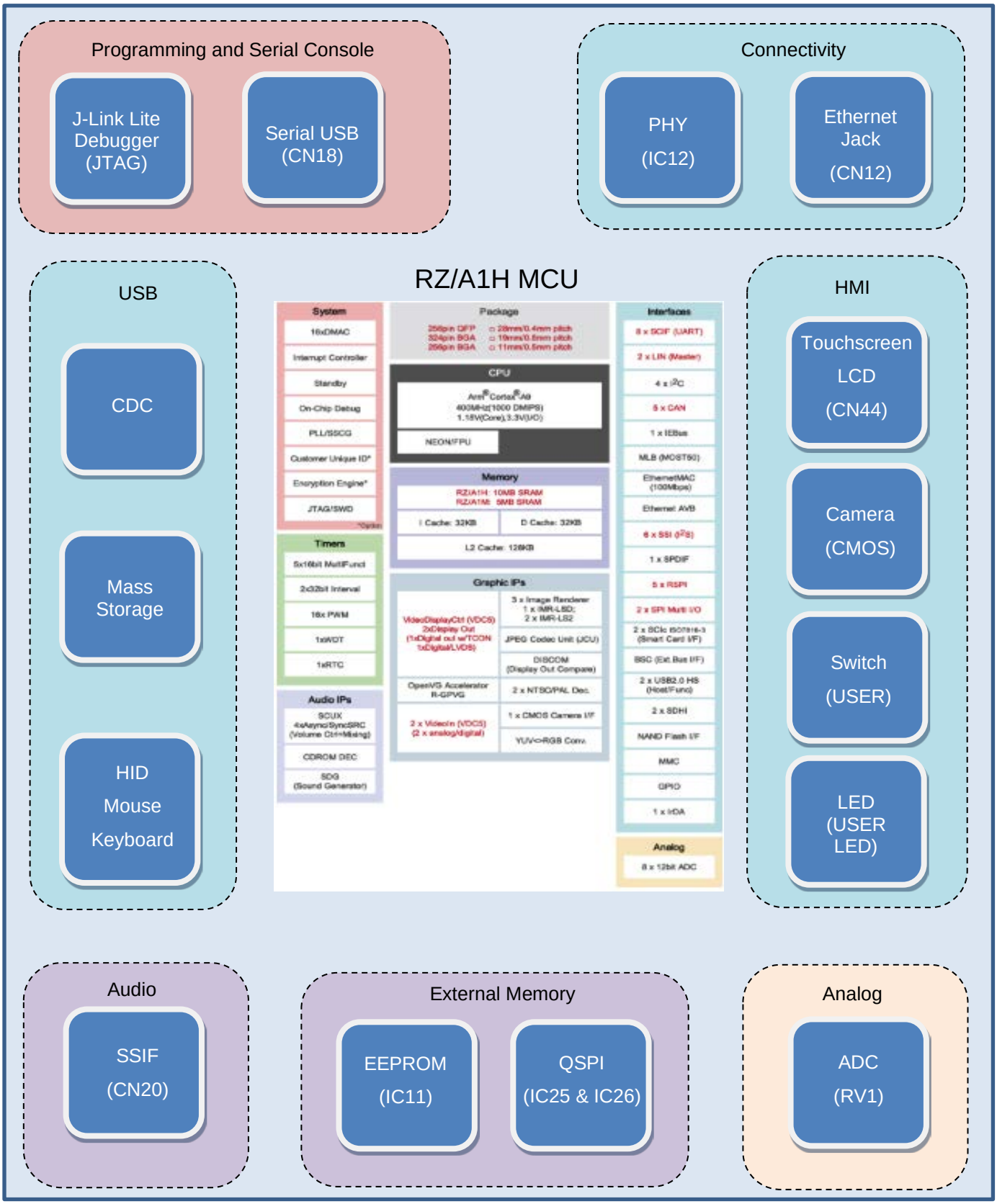

Figure 2-1 Overall System

2.1 Hardware

2.1.1 Programming and Serial Console

The board is powered through the USB function connector, CN18. This also provides a serial communication port to a PC. A USB-A to USB mini-B is required for the for the PC connection and for the board to be powered.

To program the board a J-Link Lite debugger is required to connect to the connector labelled ‘JTAG’.

2.1.2 Connectivity

The Renesas Starter Kit + for RZ/A1H has built-in ethernet connectivity. To use this the user will required to connect a RJ45 Cable from CN12 to a Local Area Network.

2.1.3 HMI

The platform includes a TFT LCD Touchscreen which connects to CN44, and a VIDEO IN which connects to CN38 and CN39. The development board also includes user switches and LEDs.

2.1.4 Memory

On board the platform includes an EEPROM and QSPI.

2.1.5 USB

A USB Hub (not supplied) will allow for multiple devices to be connected to the platform making full use of the USB HID, Mouse and Keyboard and CDC. It is advised that the hub should be self-powered.

2.1.6 Analog

The variable resistor, RV1 on the Renesas Starter Kit + for RZ/A1H board allows a user controllable analogue voltage to be presented to the microcontroller’s ADC input.

2.1.7 Audio

The Renesas Starter Kit + for RZ/A1H board uses a CODEC IC, MAX9856 (IC14) to manage playback of audio information via the 3.5mm audio jack CN20 and to manage the acquisition of audio information via the 3.5mm line-in jack CN19. The analogue sound output is passed to the jack from the CODEC headphone connection and the microphone input on the jack is connected to the MIC input on the CODEC. The audio data connection from the CODEC is connected to the MCU SSIF interface channel 0. There is also a separate I^2C connection (on IIC channel 3) to allow the MCU to configure the CODEC appropriately.

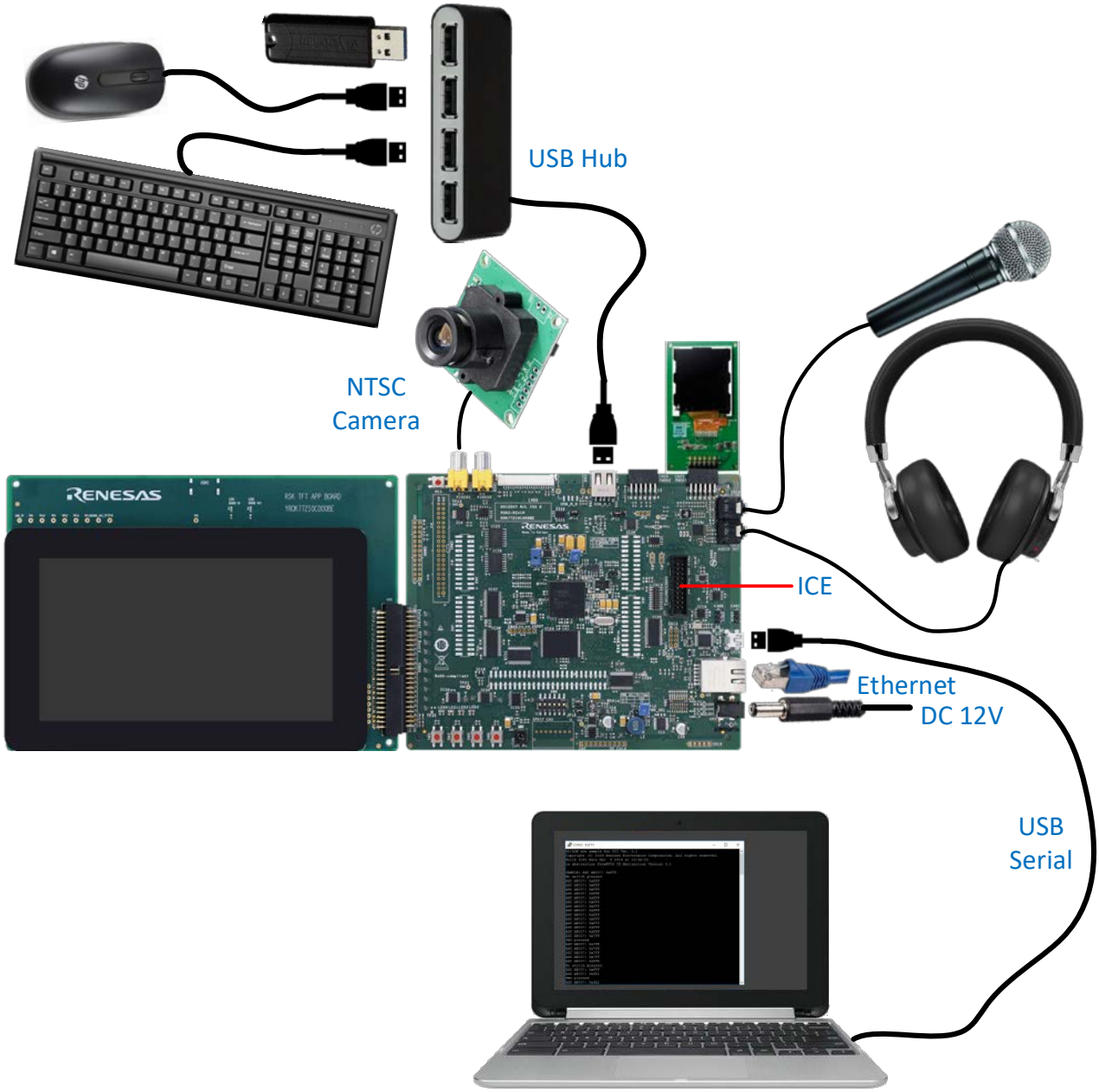

Figure 2-2 System Setup

3. System Software

This section will describe the software that is common throughout the system. This includes information on the operating system, STDIO interface and the system commands.

3.1 Configuration

The software allows for multiple applications to be used through the use of the `application_cfg.h` file. In this file the user can enable or disable some of the features of the application.

The file is located in `src/Renesas/configuration/application_cfg.h` and can be seen below. Note that this may differ from the default state of `application_cfg.h`.

```
/* Enable support for stdio.h in application  */
#define R_USE_ANSI_STDIO_MODE_CFG (R_OPTION_ENABLE)

/* Enable blink LED task in main.c */
#define R_SELF_BLINK_TASK_CREATION (R_OPTION_DISABLE)

/* Enable Ethernet drivers, WebServer Support  */
#define R_SELF_LOAD_MIDDLEWARE_ETHERNET_MODULES (R_OPTION_DISABLE)

/* Enable control for src/application/app_adc sample application */
#define R_SELF_INSERT_APP_ADC (R_OPTION_DISABLE)

/* Enable PMOD RPSI src/application/app_pmod sample application */
#define R_SELF_INSERT_APP_PMOD (R_OPTION_DISABLE)

/* Enable control for src/application/app_sound sample application */
#define R_SELF_INSERT_APP_SOUND (R_OPTION_DISABLE)

/* Enable control for src/application/app_touchscreen sample application */
#define R_SELF_INSERT_APP_TOUCH_SCREEN (R_OPTION_DISABLE)

/* Enable control for src/application/app_sdk_camera sample application */
#define R_SELF_INSERT_APP_SDK_CAMERA (R_OPTION_DISABLE)

/* Enable control to load /src/renesas/middleware/usb_host_controller */
#define R_SELF_LOAD_MIDDLEWARE_USB_HOST_CONTROLLER (R_OPTION_DISABLE)

/** Enable control to load /src/renesas/middleware/usb_host_controller */
#if R_SELF_LOAD_MIDDLEWARE_USB_HOST_CONTROLLER

/* Enable file system support when R_SELF_LOAD_MIDDLEWARE_USB_HOST_CONTROLLER is enabled */
#define INCLUDE_FILE_SYSTEM (R_OPTION_ENABLE)

/** Enable USB CDC Control via console if R_SELF_LOAD_MIDDLEWARE_USB_HOST_CONTROLLER is enabled*/
#define R_SELF_INSERT_APP_HOST_CDC_CONSOLE (R_OPTION_ENABLE)

#endif /* R_SELF_LOAD_MIDDLEWARE_USB_HOST_CONTROLLER */

/* Enable control for src/application/app_hid_mouse application */
#define R_SELF_INSERT_APP_HID_MOUSE (R_OPTION_DISABLE)

/* Enable control for src/application/app_cdc_serial_port application */
#define R_SELF_INSERT_APP_CDC_SERIAL_PORT (R_OPTION_DISABLE)

#if R_SELF_INSERT_APP_CDC_SERIAL_PORT

/* Configure driver mode when R_SELF_INSERT_APP_CDC_SERIAL_PORT is enabled */
/* R_SELF_APP_CDC_ASYNC_MODE (R_OPTION_ENABLE)  = I/O in this mode is non-blocking */
/* R_SELF_APP_CDC_ASYNC_MODE (R_OPTION_DISABLE) = I/O in this mode is blocking */
    #define R_SELF_APP_CDC_ASYNC_ENABLE (R_OPTION_ENABLE)
#endif
```

There are some features which cannot coexist in the same program. Table 3-1 shows the features that can not be enabled concurrently.

Feature one	Feature two	Reason
R_SELF_INSERT_APP_CDC_SERIAL_PORT	R_SELF_APP_CDC_ASYNC_ENABLE	Both features use the USB function

Table 3-1 Feature Conflicts

3.2 Loading the Software Package

QSPI flash is a serial peripheral interface with multiple data lines. A key feature of this interface is the ability to connect two serial flash memories to a channel. The data bus size for each channel can be specified as 1-bit, 2-bits or 4-bits. The RZ/A1H device supports a single channel of QSPI memory.

To successfully run the software package a first program is required to be executed following a system reset. This program is required to configure the device to a known state and then loads the RZ/A1H Software Package to memory at 0x18080000.

For more information regarding the bootloader please refer to the Application Note ‘RZ/A1H QSPI Flash Boot Loader’ (R11AN0084EG).

To ensure the QSPI device has the correct QSPI loader simply run the ‘Program_RZ_A1H_boot.bat’ found in the util/dos_scripts/Program_RZ_A1H_boot.bat.

Note that this script must be run from windows explorer or a DOS prompt.

3.3 Operating System

The RZ/A1H Software Package uses FreeRTOS V10.0.0 to provide an OS environment. FreeRTOS is a third-party software package which provides pre-emptive multi-taking OS functionality.

The following image provides the software architecture for the OS.

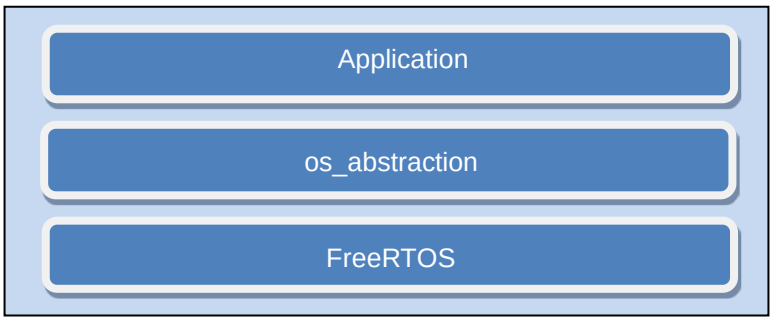

Figure 3-1 OS Hierarchical Software Layers

To allow for easy portability between different operating systems an operating system abstraction layer has been created which encapsulates all of the common features between operating systems. The source code for the OS abstraction can be found in `src/Renesas/configuration/os_abstraction`.

Using the operating system abstraction layer allows for OS objects such as tasks, mutexes, semaphores, events and message queues to be created, with minimal changes to the application code using them if the underlying operating system is changed.

3.4 STDIO

The implementation of this interface is based on the ANSI Standards with some differences made to suit operation in an embedded system where specific parts of the ANSI standard may not be available such as the memory management features and multitasking on an operating system.

The STDIO implementation provides a method for selecting a device driver to support any resource available in any system. The main aim is to ensure that device drivers are portable and the appropriate driver is selected for any device or devices in, or attached to, the system.

There are two types of device which are described in the document:

- Static devices
- Dynamic devices

Any of these devices may be opened when the exact symbolic link name is used. Dynamic devices need to be assessed to find out the class of device.

The selection of the class of device is done externally usually by the application or USB stack that knows how to connect to the device and returns a class name. This name is then compared to a driver name and causes the first capable driver to be opened if it is not already opened.

This structured code implementation provides the developer with the opportunity to write a specific driver for a ‘resource’ and re-use it in any number of other systems. An example of this would be a USB device that could be used on multiple systems using different architectures. It uses the C language standard to ensure compatibility.

For the purposes of this document a resource can be considered as any item that provides a service or that can perform read / write functions. Examples could be a peripheral device, a protocol library, or an entire protocol stack implementation. There are many other examples - please review the supplied sample code. There are three layers to consider.

- Application

The application is written by the developer and uses the resources controlled by this implementation. Note that the resources can also be used by other device drivers for example a EEPROM driver would implement the I^2C protocol.

- Device Drivers

Device drivers provide all the functionality required to use the device without needing to know how that support is implemented. This is a form of abstraction.

- Hardware Layer

Device drivers that access hardware or peripheral interfaces can (and should) be abstracted from the physical nature of the device. This allows the device driver to remain constant handling the flow of information or controls, while the hardware layer accesses the peripherals.

The hardware layer is still part of the device driver as the interface is defined by the device driver implementation. The hardware layer needs to be changed when porting the driver to another device.

The driver layer provides a common interface to the application and specific functionality to the device being controlled. The hardware layer provides the specific configuration and control codes required for the physical hardware to be used by the driver. This layer is intended to change between systems and architectures.

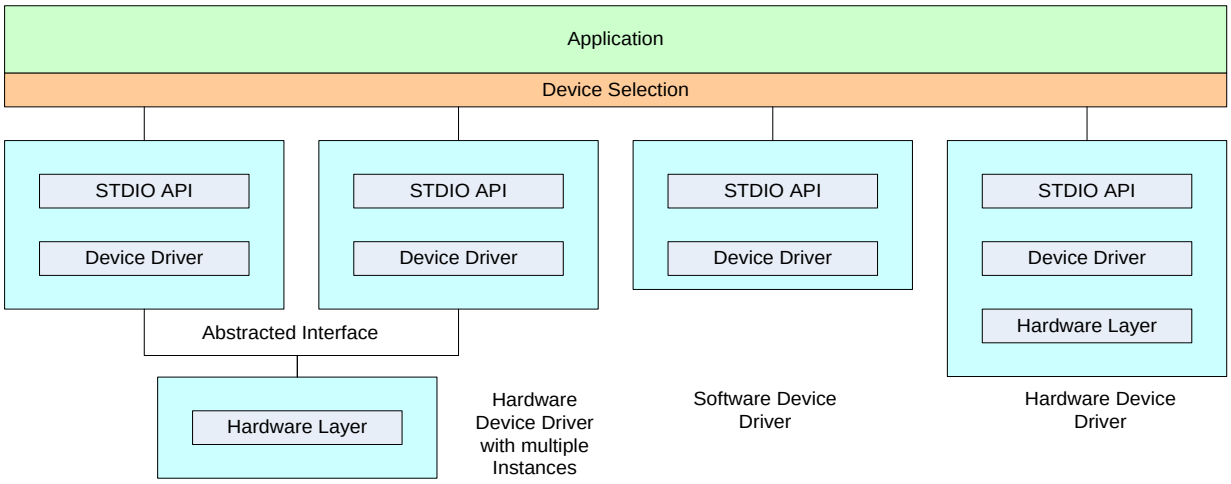

Figure 3-2 Architecture Overview

The terms ‘abstraction’ and ‘Object Orientation’ apply to this application note in that the device driver is defined and contains the principles of an object. The use of a hardware layer implements abstraction in that the device driver is abstracted from the specifics of the hardware being used by this layer. This means that the device driver becomes portable between systems and architectures allowing maximum re-use of code that is available for a minimal amount of effort to port the code between systems. Object orientation is not directly part of the C language, but the concept is the same in this implementation.

Figure 3-2 shows the architecture of a system where there are different implementations of device drivers, all of these have the common STDIO interface.

The ANSI I/O library is based on streams. Each of the I/O functions we will describe later contains a reference to the stream in use. A stream can be considered as a river of information flowing from one point to the next. Each stream has a source and a termination (or terminal). The number of streams that can be supported by the ANSI library will control the maximum number of open files and devices at any one time.

There are three standard streams in Unix and ‘C’ language implementations. These are commonly known as **stdin**, **stdout** and **stderr**. Each of these relate to the base terminal interface that may or may not be available in any system.

Each stream that is supported requires some system resources. The number of streams that are supported in the demonstration is defined in `lowsrc.c`.

```
/* Define the number of ANSI IO Streams */
#define IOSTREAM                    32
```

Figure 3-2 Number of Streams

The value of this definition should be increased to allow the opening of all devices in the system and an additional number relating to the number of open files required in the system.

3.4.1 Stream Configuration Example

The demonstration application may use:

Device	Streams In Use
The standard terminal (in, out and error)	3
Host Stack	1
Mass Storage device driver	1
Audio Device Driver	1
HID Mouse Device Driver	1
HID Keyboard Device Driver	1
Number of Files opened	F
Total Streams	8 + F

Table 3-2 Configuration Example

The default value of IOSTREAM is 32 to provide scope to add more devices or open more files. If memory optimisation is required then reduce this number to the limits needed in your application.

3.4.2 STDIO Files

To implement the STDIO functionality in your system there are some additional files that need to be included in the application. For normal STDIO operation these would be the `stdio.h` header file and `lowsrc.c` code file. When generating a new application project that includes STDIO support the Renesas project generator will create an example `lowsrc.c` file in the project for you. This file provides an example of the minimum support for STDIO. The details of the file and its contents are described later. For this enhanced implementation a further file is added to the standard files this is `devlink.c` and is described later.

3.4.3 STDIO Include Files

Each C compiler will provide a standard library header to support the STDIO functionality. For Renesas this file is called `stdio.h`. This header file is included by the toolchain but can be opened and viewed by browsing to the toolchain include file location.

To use the STDIO library the developer must include the `stdio.h` file in the application code. Other files may also need to be included. For our implementation the files below are required.

```
/* Standard IO library include file for STDIO support */
#include <stdio.h>
/* Standard library helper functions providing memory managaemnt functions. */
#include <stdlib.h>
/* Standard library string handling functions for selecting devices by name. */
#include <string.h>
```

Figure 3-3 System Include Files for a Device Driver

As part of the STDIO implementation in our example there are three other files that assist the implementation.

```
/* Helper utility to allow opening of device drviers by name */
#include "r_devlink_wrapper.h"
/* Header file for devie driver configuration */
#include "r_devlink_wrapper_cfg.h"
/* Endian swapping and support utilities if required */
#include "endian.h"
```

Figure 3-4 Application Include Helper Files

Device linkage is covered later in this chapter, the other header files do not relate to this document but are worth mentioning in context.

3.4.4 STDIO lowsrc.c File

In the Renesas implementation of STDIO there are two files. The standard `lowsrc.c` file is required in all STDIO implementations. This implementation adds another file `r_devlink_wrapper.c` which provides the additional linkage to specify devices as well as files by using the same underlying structure. For more information, please refer to the source files in `src/renesas/application/system`.

3.4.5 STDIO devlink.c File

The enhancement to the ANSI STDIO implementation to support device drivers requires that the application has a way of opening a device driver by name. To achieve this a distinction is required between files and devices. To achieve this, the file name is prepended with a DEVICE_INDENTIFIER string as shown in Figure 3-5 below.

```
/* This is so the low level open function can tell the difference between a
   file and a device:
   Device name \\.\myDevice
   file name a:\myFile.txt
*/
#define DEVICE_INDENTIFIER          "\\\\.\\"
```

Figure 3-5 Device Identifier String Definition

There are two types of devices that are supported in the system, static and dynamic devices. Static devices are devices that physically exist either as a software driver or a hardware interface in the system and that can be ‘mounted’ when it is running. These devices are always available. Dynamic devices are those that can be added and removed from the system. The easiest description of this is a USB device connected to a USB host system. Everyone can see how a user may add a mass storage device and use it and then remove it from the system. This can also be applied to other devices, such as plug in cards, devices connected to Ethernet terminals.

Device drivers are opened by name by the application, this results in a function pointer being made available to allow access to the device driver function table. The architecture of this is shown in the diagram below:

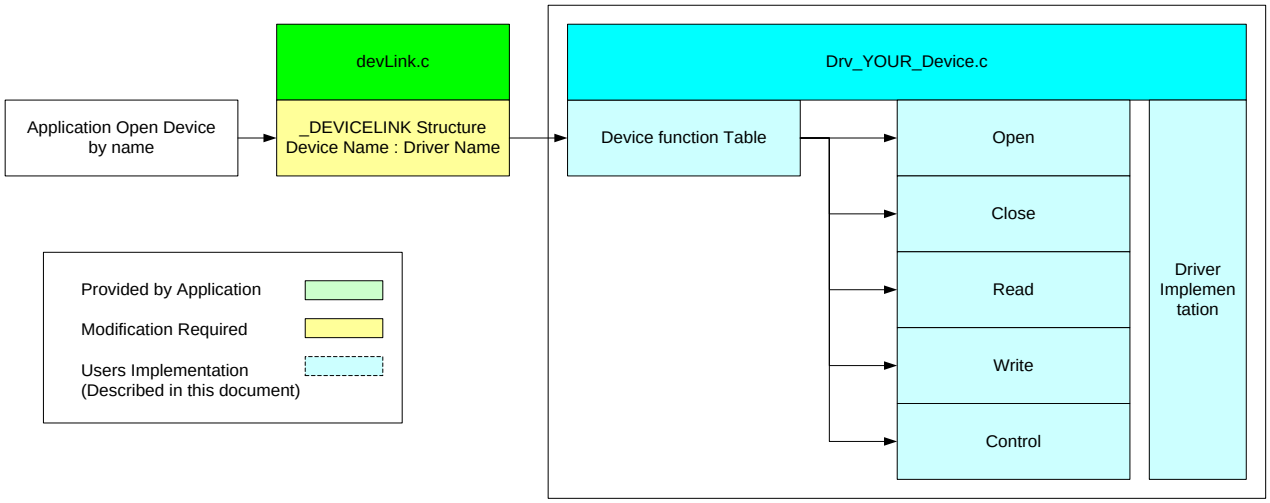

Figure 3-6 Device Linkage

To support this functionality the device link module maintains a list of static devices with a link to the Device Function Table for each, and supported dynamic devices using a common link name that can be used to select the device driver.

3.4.6 Device Driver Function Table

All device drivers must contain a device driver function Table that contains the device driver name and pointers to the standard `open`, `close`, `read`, `write` and `control` functions. This list is a constant data structure that is defined at compilation. The devlink.c file supports stub functions for all of these standard functions to allow a device driver that does not have one of the functions (e.g. I/O) to specify the stub function to ensure that there is known functionality in case of an erroneous call by the application to a uninitialized pointer.

An example of how this may be implemented is below. Note that the keyboard driver implementation does not have a 'write' function so the default stub has been specified.

```
/* Define the driver function table for this device */
 const DEVICE gHidKeyboardDriver =
{
    "HID Boot Mode Keyboard Device Driver",
    kbdOpen,
    kbdClose,
    kbdRead,
    nodevIO,
    kbdControl
};
```

Figure 3-7 Sample Hid Keyboard Device Driver Function Table

The Device Driver Name as shown in Figure 3-7 is used by the Dynamic Device Link Table to match a Device Driver to a Device Driver Link Name that is passed to the 'open' function by the application.

3.4.7 Dynamic Device List

The Dynamic Device List provides the system with a database of all dynamically connected devices. The method for detecting and adding / removing devices is dependent upon the application and cannot be described here.

It is very important to realise that this Dynamic Device List must be protected from concurrent access throughout the system and must therefore only be accessed through this API. The API must also utilise appropriate system locking implementation to enforce this restriction.

3.4.8 Dynamic Device Link Table

The Dynamic Device Link Table is written by the developer for the application and provides connectivity between the Device Driver Name an application (or another driver) passes to the open function and the Device Driver Link Name.

```
static const struct _DEVICELINK
{
    /* The link name passed to the open function */
    const int8_t *const pszClassLinkName;
    /* The pszDeviceName member in the DEVICE structure */
    const int8_t *const pszDriverName;
} gDeviceLinkTable[] =
{
    {   /* USB Mass Storage Class devices */
        "Mass Storage",
        "MS BULK Only Device Driver",
    },
    {   /* USB HID Class keyboard devices */
        "HID Keyboard",
        "HID Boot Mode Keyboard Device Driver",
    }
    /* TODO: Add in other supported class drivers here */
};
```

Figure 3-8 Sample Dynamic Device Link Table

This table is a constant structure that is resident and created on compilation. All dynamic devices that the system needs to support must be added to this list by the developer at the same time as adding the device driver code into the link map.

3.4.9 Static Device Mount Table

The static device mount table does not need to support the complexity of a managed list as all the devices and associated drivers are known to be present in the system. The table therefore provides a direct correlation between the symbolic link name passed by the open function and the device driver function table pointer.

A sample mount table is shown below.

```
static const struct _MOUNTTABLE
{
    int8_t  *pszStreamName;
    DEVICE  *pDeviceDriver;

} gMountTable[] =
{
    /* ANSI Standard IO Devices */
    "stdin",    &gScif3Driver,
    "stdout",   &gScif3Driver,
    "stderr",   &gScif3Driver,
    /* System specific devices */
    "swtimer",  &gTimerDriver,
    "usbh",     &gUsbHostDriver,
    "rtc",      &gRtcDriver,
    "kbd",      &gStdInDriver,
    "lcd",      &gLcdDriver,
    "eeprom",   &gAT24C16Driver,
    "ether0",   &gEtherCDriver,
    "socket",   &gSocketDriver,
    "fsocket",  &gFileSocketDriver,
    "udpecho",  &gUdpSocketDriver
    /* TODO: Add other devices in here */
};
```

Figure 3-9 Sample Static Device Linkage List

This table includes software devices such as the Ethernet Socket Driver along with hardware drivers for example the LCD driver. All of these devices are present in the system at compilation time. The DEVICE pointers specifed are added to the table by the compiler.

3.5 System Commands

All of the systems applications and features can be accessed via the command line interface. This interface operates via a communications port to the PC. The function USB (CN18) allows for a USB to Mini-B wire to be connected to a PC. To access the serial line a serial terminal is requires such as PuTTY or Tera Term. The serial command works on the following set up:

- Baud Rate: 115200
- Data Bits: 8
- Parity: None
- Stop Bits: 1
- Flow Control: None
- COM Port: As shown in Windows™ Device Manager.

Once accessed the user should see the following message:

```
RZ/ALU RZ/A Software Package Ver X.XX.XXX
Copyright (C) Renesas Electronics Europe.

REE>
```

Where X.XX.XXX is the release version of the software.

Following this the user may enter any system commands. Following this the user may then explore the application commands. The system commands can be seen in the below table:

Command	Description
? F1 help	Displays the complete supported command list to the terminal window All supported commands and associated help text will be output
F2	Retypes the last command line so it can be changed or repeated
F3	Repeats the last command exactly as typed
sys	Shows the system resource usage information
task	List tasks from the task list
mperf	Executes a memory performance test Generates a buffer in the available RAM and executes the highest performance copy available to the system. If a DMA channel is available and supported then this will be used
led a s	Turn LED on (s = 1), off (s = 0) or toggle (s = ^). Note that the Renesas Starter Kit + for RZ/A1H has four user LED, so the letter ‘a’ should be replaced with a ‘0’, ‘1’, ‘2’, or ‘3’. For example, to toggle the user LED, type ‘led 0 ^’
restart	Invokes a restart of the MCU. This is usually achieved by allowing the watchdog timer to time out, creating an MCU reset Note: This is only applicable to code run from ROM. Restart will not operate correctly if using an E1 debugger
mem a l	Read any memory locations in the system memory map – intended as a debugging tool No address checking is performed – ensure address entered is a valid memory location a = address (Hexadecimal) l = length in lines of 16 bytes
logout	Exit the current login shell Note that on exiting the serial console a new console will immediately be launched
date dd/mm/yyyy	Set the date
time hh:mm:ss	Set the time
ver	Shows the version of the software package
handles	Lists the opened driver information
drivers	Lists the driver table

Table 3-3 System Commands

3.6 Doxygen

Doxygen is a third-party tool used for generating documentation from the source code. The RZ/A1H software supports this tool, allowing users to generate document to further their understanding of the source code.

For further information regarding Doxygen please refer to their website. The Doxygen output for the RZ/A1H, in HTML form, can be found in the doc folder in the root folder of the software package source, compressed into file html.zip. The following sections will refer to Doxygen for more information regarding drivers, middleware and applications.

When unzipped, the homepage for Doxygen can be found relative to the root of the zip file at: html/index.html

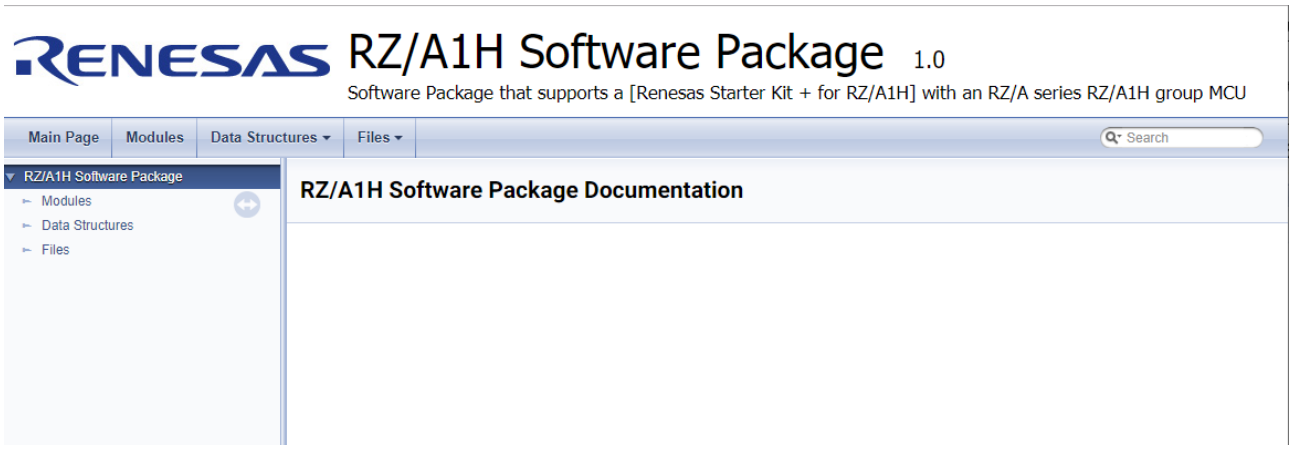

Figure 3-10 Index.html

4. Applications

The following sections will provide information on each application that is included in the software package. Each subsection will contain a block diagram containing hierarchical software layers.

4.1 ADC

The ADC demo allows for ADC Port 1 Pin 10 to be used as ADC Channel 7 and read the pin's associated voltage. The voltage on that pin can be altered by varying the position of potentiometer RV1.

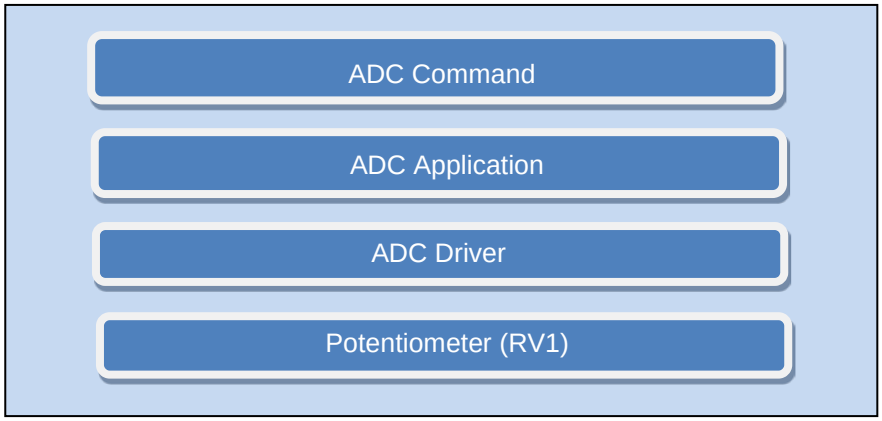

Figure 4-1 ADC Hierarchical Software Layer

4.1.1 Software Configuration

To enable, set `R_SELF_INSERT_APP_ADC` to `R_OPTION_ENABLE`. To disable, set it to `R_OPTION_DISABLE`. See 3.1 Configuration for further details.

```
/** Enable control for src/application/app_adc sample application */
#define R_SELF_INSERT_APP_ADC (R_OPTION_ENABLE)
```

4.1.2 Commands

The Renesas Starter Kit + for RZ/A1H Board ADC application has the following ADC commands:

Command	Description
adcdemo	Executes the ADC Application

Table 4-1 ADC Commands

4.1.3 Application

The application displays the real time value of the 12-bit ADC. To terminate the demo press any key.

Figure 4-2 ADC Application

4.1.4 ADC Driver

The ADC Driver API can be seen in Doxygen under RZ/A1H Software Package → Modules → Drivers (Not POSIX) → ADC RZ/A1H Driver

4.2 Switch

Switch SW1 is connected Port 7 pin 9 can be used at any time. Pressing the USER switch results in the following message.

```
REE> USER switch was pushed!!
```

Figure 4-3 Switch Application

The switch pin is initialized as an interrupt on start up.

4.2.1 Switch Driver

The switch Driver API can be seen in Doxygen under RZ/A1H Software Package → Modules → Drivers (Not POSIX) → SWITCH RZ/A1H Driver

4.3 LEDs

The green LED (LED0) is connected to Port 7 Pin 8 and is active low. The yellow LED (LED1) and red LEDs (LED2 and LED3) are driven by IC34 (CAT9554) pins 4, 5 and 6 and are also active low. To control the LEDs the following commands are available.

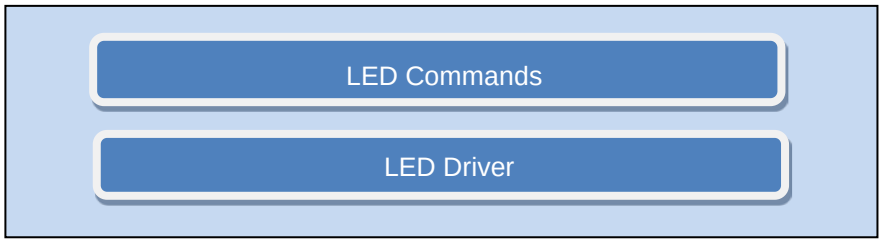

Figure 4-4 LED Hierarchical Software Layer

4.3.1 Commands

Table 4-2 shows the commands available for the four user LEDs.

Command	Description
led a s <CR>	Turn LED on (s = 1), off (s = 0) or toggle (s = ^). Note that the Renesas Starter Kit + for RZ/A1H has four user LEDs, so the letter ‘a’ should be replaced with a ‘0’, ‘1’, ‘2’, or ‘3’. For example, to toggle the green user LED, type “led 0 ^”

Table 4-2 LED Commands

4.3.2 LED Driver

The LED driver uses the interface as discussed in section 3.4. The LED driver allows for the control of four LEDs of the Renesas Starter Kit + for RZ/A1H.

The LED Driver API can be seen in Doxygen under RZ/A1H Software Package → Modules → Drivers (POSIX) → Renesas Starter Kit + for RZ/A1H LED driver.

4.4 PMOD

The PMOD™ application allows for three different images to be displayed on the Okaya PMOD LCD.

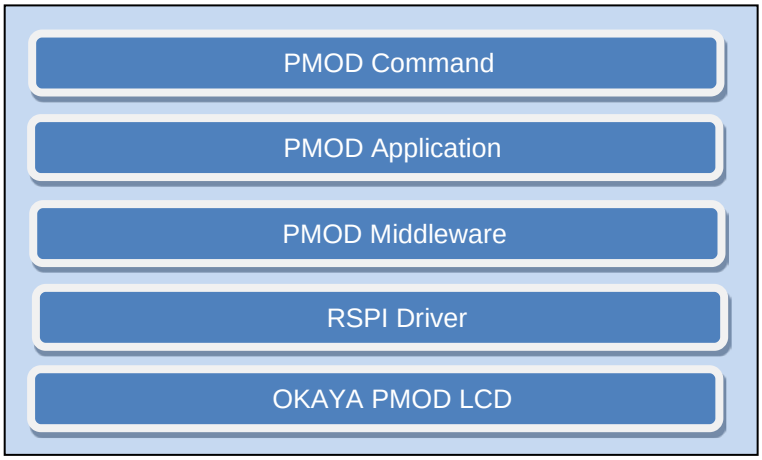

Figure 4-5 PMOD Hierarchical Software Layer

4.4.1 Software Configuration

To enable, set `R_SELF_INSERT_APP_PMOD` to `R_OPTION_ENABLE`. To disable, set it to `R_OPTION_DISABLE`. See 3.1 Configuration for further details.

```
/** Enable PMOD RPSI src/application/app_pmod sample application */
#define R_SELF_INSERT_APP_PMOD (R_OPTION_ENABLE)
```

4.4.2 Commands

The PMOD application has the following commands:

Command	Description
pmoddemo	Executes the PMOD Application

Table 4-3 PMOD Commands

4.4.3 Application

The USER may attach any PMOD compatible device to the connector, this will require for the user to create their own bespoke application. This software package is geared to display three images: a desert, hydrangeas and penguins on the Okaya Pmod LCD.

Figure 4-6 ADC Application

The application can be found in `src/renesas /application/app_pmod`. The images are stored in the bitmap image files in the '`graphics`' folder. The BMP file format is capable of storing two-dimensional digital photo images both in monochrome and color in various depths.

The bitmap image file consists of fixed-size structures (headers) as well as variable-size structures appearing in a predetermined sequence. Many different versions of some of these structures can appear in the file, due to the long evolution of this file format.

The format used in this application can be seen in Table 4-4. These bitmaps are then passed to a PMOD driver.

Structure name	Optional	Size	Purpose	Comments
Bitmap file header	No	14 bytes	To store general information about the bitmap image file	Not needed after the file is loaded in memory
DIB header	No	Fixed-size (7 different versions exist)	To store detailed information about the bitmap image and define the pixel format	Immediately follows the Bitmap file header
Extra bit masks	Yes	3 or 4 DWORDs (12 or 16 bytes)	To define the pixel format	Present only in case the DIB header is the BITMAPINFOHEADER and the Compression Method member is set to either BI_BITFIELDS or BI_ALPHABITFIELDS
Color table	Semi-optional	Variable-size	To define colors used by the bitmap image data (Pixel array)	Mandatory for color depths ≤ 8 bits
Gap1	Yes	Variable-size	Structure alignment	An artifact of the File offset to Pixel array in the Bitmap file header
Pixel array	No	Variable-size	To define the actual values of the pixels	The pixel format is defined by the DIB header or Extra bit masks. Each row in the Pixel array is padded to a multiple of 4 bytes in size
Gap2	Yes	Variable-size	Structure alignment	An artifact of the ICC profile data offset field in the DIB header
ICC color profile	Yes	Variable-size	To define the color profile for color management	Can also contain a path to an external file containing the color profile. When loaded in memory as "non-packed DIB", it is located between the color table and Gap1.

Table 4-4 Bitmap Structure

4.4.4 PMOD Middleware

The pmod_lcd driver supports the OKAYA LCD. The middleware can be found at: `src/renesas/middleware`. The middleware allows for the control of the PMOD Driver and allows for STDIO interface as described in section 3.4.

The PMOD Middleware API can be seen in Doxygen under RZ/A1H Software Package → Modules → Middleware (POSIX) → PMOD API.

4.4.5 RSPI

The RSPI driver API can be seen in Doxygen under RZ/A1H Software Package → Modules → Drivers (Not POSIX) → RSPI RZ/A1H Driver.

4.5 USB Mass Storage

USB Mass Storage device Class (MSC) is a set of computing communication protocols that allows for the USB device to be accessible from a host PC. This allows the host to transfer data to the USB device.

The software package makes use of the FAT file system, to allow files to be transferred from the RZ/A1H to the USB drive. This can either be done through the command line or through the website as described in section 4.6.

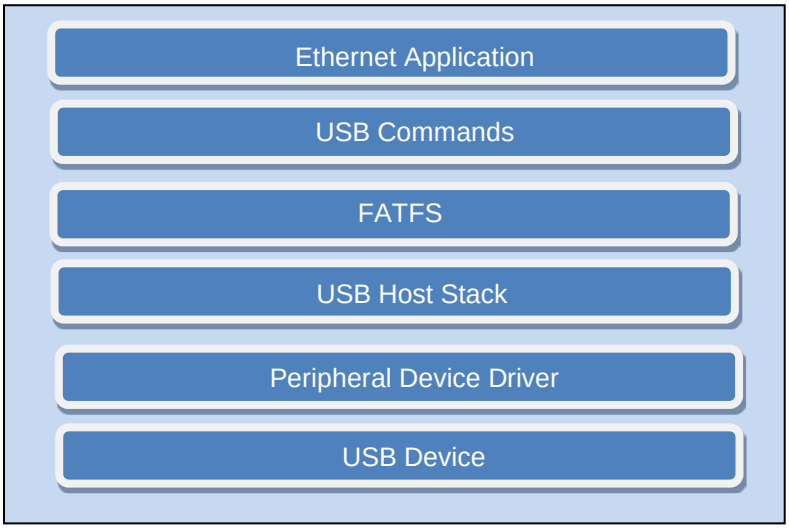

Figure 4-7 USB Hierarchical Software Layer

4.5.1 Software Configuration

To enable, set `R_SELF_LOAD_MIDDLEWARE_USB_HOST_CONTROLLER` to `R_OPTION_ENABLE`. To disable, set it to `R_OPTION_DISABLE`. See 3.1 Configuration for further details.

```
/** Enable control to load /src/renesas/middleware/usb_host_controller */
#define R_SELF_LOAD_MIDDLEWARE_USB_HOST_CONTROLLER (R_OPTION_ENABLE)
```

4.5.2 Commands

These commands can be called from the serial terminal and allow for file and USB MSC manipulation.

Command	Description
n:	Multiple derives are supported in the implementation. Each drive is allocated a letter starting form a. This command allows the user to select a working disk drive to "n"
vol	Interrogate the working drive and provide details of the volume information
type f	Reads a text based file and output the characters to the current console. Do not use this command on files that include non-alphanumeric characters
copy s d	Copies file "s" to destination "d". The full path can be specified. If no path is specified the current working drive directory will be used
view f	Similar to the type command; however, this will output any type of file and display the contents in mixed hex and character format
dir	List the working directory contents to the current terminal
pwd	Print the working directory to the current terminal
cd d	Change working directory to "d" where “d” is the full path or a subdirectory or ‘..’ to return to the parent directory
del f	Delete file "f" from the specified path. If no path is specified then the current directory is used. There is no confirmation of this command
md n	Make a new directory "n" in the working directory. Full paths cannot be used
rd d	Delete directory "d" from the specified path. If no path is specified then the current directory is used. There is no confirmation of this command
ren s d	Rename / move file "s" to "d"
disk	List the available disk drives attached to the system
eject d	Eject disk "d". Ensures all files are closed on the disk
dismount	Dismount all mounted drives. This ensures all files are closed on all drives and their allocated resources are released
mount	Mount all Mass Storage devices that are detected in the system. This is normally done when the device is detected. This would be needed if the drives have been “dismounted”

Table 4-5 USB MSC Commands

4.5.3 FATFS

FatFs is a free source third party file system. The files system is a generic FAT/exFAT filesystem module for embedded systems. The FatFs module is written in compliance with ANSI C (C89) and completely separated from the disk I/O layer. Therefore, it is device independent.

As a result, FATFS may be used on various memory devices such as QSPI, NAND, NOR, etc. The software package implements this on USB Mass storage.

The FATFS source can be found at; `src/renesas/middleware/fatfs`. In this folder an abstraction layer is implemented in the `r_fatfs_abstraction.c` to allow for users to easily port another file system (such as VFAT and FullFAT). Functions from this file are called in the application.

The FATFS interfaces with the USB manipulation through the `diskio.c` file.

4.5.4 USB Host Stack

The USB Host Stack is comprised of a Protocol Driver, Peripheral Driver, and a Hardware Interface. The diagram below shows a simplified block diagram of the USB stack detailing the source files used. The USB Host Protocol Driver is abstracted from the hardware driver by the interface functions defined in usbHostApi.h. Class or application specific device drivers interface with the USB Host Protocol Driver through the functions defined in `usbhDeviceApi.h`.

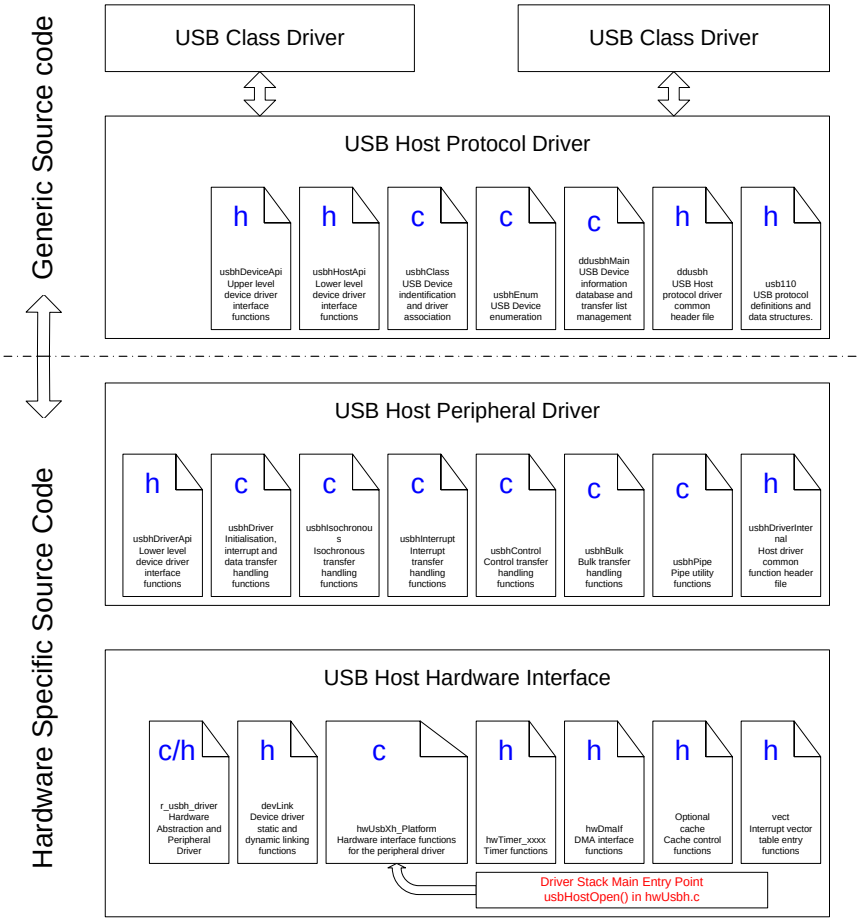

Figure 4-8 USB Host Stack

The Host Stack has some key configuration definitions that need to be set to suit the target hardware and the peripheral in the device. These definitions can be found in the `usbConfig.h` header file. Care must be taken to ensure that the configuration selected matches the capability of the peripheral. For the R8A66597 device the following configuration is valid for a single port implementation:

```
/* Define the number of root ports 1 or 2 */
#define USBH_NUM_ROOT_PORTS         1
/* Define high speed support (1 or 0)*/
#define USBH_HIGH_SPEED_SUPPORT     1
/* Define the FIFO access size only 32 and 16 are valid */
#define USBH_FIFO_BIT_WIDTH         32
/* The maximum number of host controllers supported */
#define USBH_MAX_CONTROLLERS        1
/* The number of tiers of hubs supported */
#define USBH_MAX_TIER               1
/* The maximum number of hubs that can be connected */
#define USBH_MAX_HUBS               1
/* The maximum number of ports */
#define USBH_MAX_PORTS              5
/* The maximum number of devices */
#define USBH_MAX_DEVICES            5
/* The maximum number of endpoints */
#define USBH_MAX_ENDPOINTS          32
```

Figure 4-9 Single Port Example Configuration Settings

The following configuration is valid for a dual root port implementation of the peripheral:

```
/* Define the number of root ports 1 or 2 */
#define USBH_NUM_ROOT_PORTS         2
/* Define high speed support (1 or 0)*/
#define USBH_HIGH_SPEED_SUPPORT     1
/* Define the FIFO access size only 32 and 16 are valid */
#define USBH_FIFO_BIT_WIDTH         32
/* The maximum number of host controllers supported */
#define USBH_MAX_CONTROLLERS        1
/* The number of tiers of hubs supported */
#define USBH_MAX_TIER               1
/* The maximum number of hubs that can be connected */
#define USBH_MAX_HUBS               2
/* The maximum number of ports */
#define USBH_MAX_PORTS              10
/* The maximum number of devices */
#define USBH_MAX_DEVICES            10
/* The maximum number of endpoints */
#define USBH_MAX_ENDPOINTS          64
```

Figure 4-10 Dual Port Example Configuration Settings

The following configuration is valid for a multiple device implementation of the R8A66597 peripheral:

```
/* Define the number of root ports 1 or 2 */
#define USBH_NUM_ROOT_PORTS         2
/* Define high speed support (1 or 0)*/
#define USBH_HIGH_SPEED_SUPPORT     1
/* Define the FIFO access size only 32 and 16 are valid */
#define USBH_FIFO_BIT_WIDTH         32
/* The maximum number of host controllers supported */
#define USBH_MAX_CONTROLLERS        2
/* The number of tiers of hubs supported */
#define USBH_MAX_TIER               1
/* The maximum number of hubs that can be connected */
#define USBH_MAX_HUBS               4
/* The maximum number of ports */
#define USBH_MAX_PORTS              20
/* The maximum number of devices */
#define USBH_MAX_DEVICES            20
/* The maximum number of endpoints */
#define USBH_MAX_ENDPOINTS          128
```

Figure 4-11 R8A66597 Peripheral Example Configuration Settings

The USBH_MAX_ENDPOINTS is an educated guess at the total number of endpoints of all attached devices. Under some circumstances this value may need to be increased as some devices may have a number of endpoints with only one physical connection.

To estimate the required number of endpoints the following information is required:

By design in the USB Stack, every device will require three control endpoints. Added to this is one endpoint for each endpoint in the device.

An example of the calculation for common devices is below:

A single Mass Storage Device will require:

3 (OSFUSB control pipe endpoints) + 2 (bulk IN endpoint + bulk OUT endpoint) = 5 MSD endpoints

A single hub will require:

3 (OSFUSB control pipe endpoints) + 1 (hub information endpoint) = 4 hub endpoints

Therefore a single four port hub with four mass storage devices will require:

4 (ports) x 5 (MSD endpoints) + 4 (hub endpoints) = 24 endpoints

For two root ports configured as above the number is doubled to 48 endpoints.

4.5.5 Hardware Layer

The hardware interface controls peripheral functions that should be supported on the target platform or device. The key features to be included are the Timer, DMA and Host Port controls.

4.5.5.1 Timer

The Host Stack does not require an operating system or scheduler. Many of the functions of USB are time critical and therefore one system timer must be allocated to provide a timing reference. The USB peripheral is responsible for the critical USB timing and requires no user configuration. However some higher level functions of the stack need to be completed within a specified time. To enforce this timing, many of the stack core functions will run in interrupt space. To ensure compatibility with your application, please review the `interruptPriority.h` file.

The key timing definitions are in the file `usbhEnum.c`. All definitions in this file assume a timer count of 1ms. If the base hardware timer rate is changed these definitions must be updated to preserve the timing.

```
/* Times are related to enumRun call frequency. This should be called at 1kHz so
   delay times will be in ms */
#define ENUM_PORT_POLL_DELAY_COUNT  100UL
#define ENUM_PORT_POWER_DELAY_COUNT 100UL
#define ENUM_PORT_ENABLE_DELAY_COUNT 50UL
#define ENUM_ROOT_PORT_RESET_COUNT  50UL
#define ENUM_HUB_PORT_RESET_COUNT   10UL
#define ENUM_DEV_REQUEST_COUNT_OUT  500UL
```

Figure 4-12 Coded Timer Settings

The hardware level physical timer functions provide Start, Stop and Interrupt functions that link directly to the hardware timer. These functions are called during the initialisation of the stack and can be found in `hwtimer_xxx.c`.

The stack also includes a timer module that allows multiple virtual timers to be registered in a linked list. These functions can be found in `timer.c`. These virtual timers can be used in the users end application by calling the **timerStart()** and **timerStop()** functions and providing a pointer to locally defined timer structure storage.

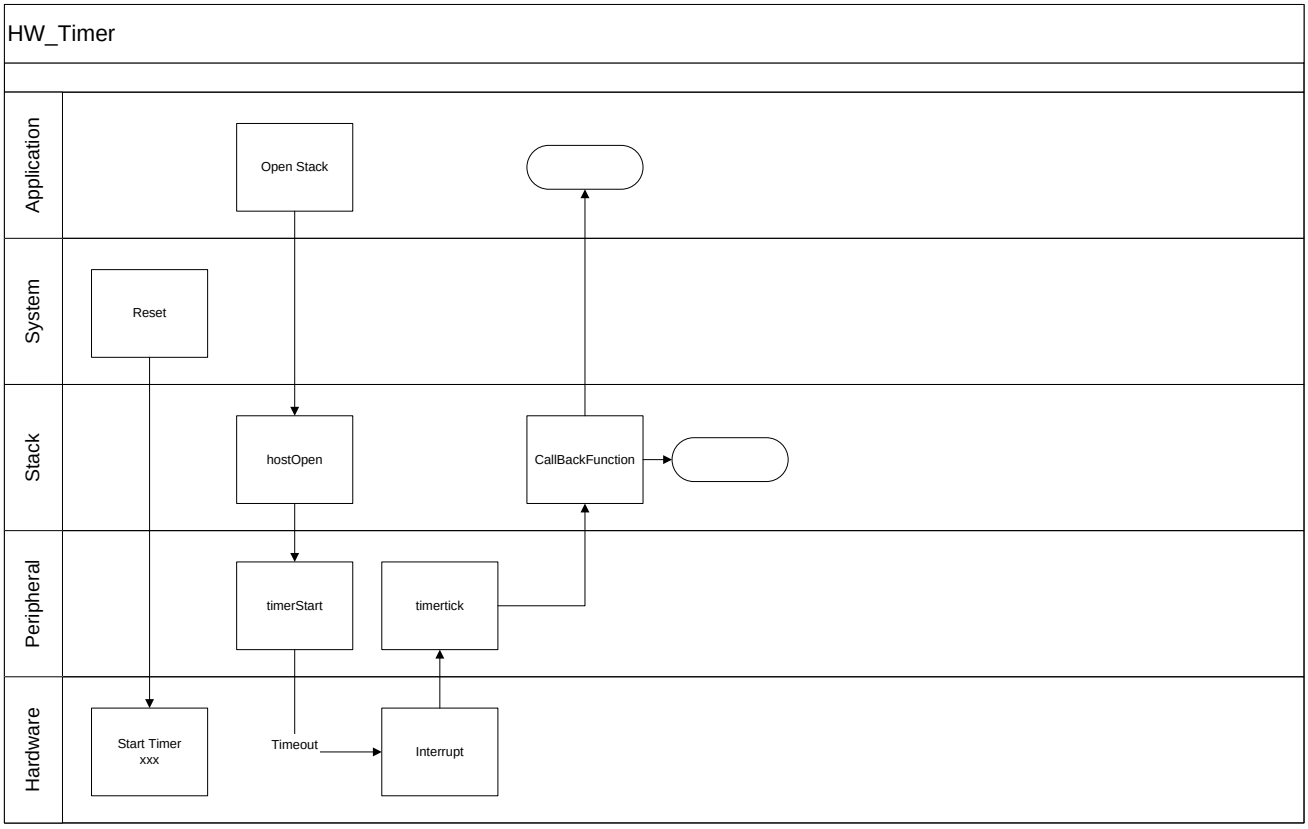

Figure 4-13 Base Hardware Timer Operation

The hardware timer must be configured to provide a 1ms interrupt for correct operation of the stack. The timer uses a DWORD value to count milliseconds giving a maximum time of over 49 days.

4.5.5.2 DMA

The DMA configuration in the stack uses two DMA channels. One DMA channel is designated for incoming data (device to host) and another channel is configured for outgoing data (host to device).

The DMA functionality is used to transfer data to and from the stack in large quantities, quickly. This type of transfer is used by the bulk transport in USB. As part of the bulk transport the amount of data to be transferred is checked. The DMA transfer is started so that it is configured to send or receive full packets of data. Any data left over from the transaction, or transactions smaller than the endpoint packet size, are completed using the FIFO. All transfers performed by the DMA are aligned to two or four byte boundaries (dependant upon the processor implementation) and at least one packet size (8, 32, 64, 512) to make the data handling simple in the stack code.

The DMAC module can be used to service many peripherals within the system and there are 16 independent DMA channels for that purpose. Two of these channels have been dedicated to support the USB stack.

These are set up in the driver mounting table in the file `r_devlink_wrapper_cfg.h`:

```
/** USB DMA driver added by USER */
{"dma_usb_in", (st_r_driver_t *) &g_dmac_driver, R_SC2},
{"dma_usb_out", (st_r_driver_t *) &g_dmac_driver, R_SC3},
```

The two channels are allocated and the initial configuration is defined in the file `r_dmac_drv_sc_cfg.h` which is located in the r_dmac driver folder:

```
static const st_r_drv_dmac_sc_config_t DMAC_SC_TABLE[] =
{
    { 0,  /* USB DMA read */
        {
            DMA_RS_USB0_DMA0_RX,
            DMA_DATA_SIZE_4,
            DMA_DATA_SIZE_4,
            DMA_ADDRESS_INCREMENT,
            DMA_ADDRESS_FIX,
            DMA_REQUEST_SOURCE,
            NULL,
            NULL,
            0x00000000,
            0x00000000,
            0,
        }
    },
    { 1,  /* USB DMA write */
        {
            DMA_RS_USB0_DMA0_TX,
            DMA_DATA_SIZE_4,
            DMA_DATA_SIZE_4,
            DMA_ADDRESS_INCREMENT,
            DMA_ADDRESS_FIX,
            DMA_REQUEST_SOURCE,
            NULL,
            NULL,
            0x00000000,
            0x00000000,
            0,
        }
    },
    …
};
```

Here we can see that DMA channel 0 is assigned for USB read operations, and channel 1 is assigned for write operations. None of the configuration is important here as it is overridden in the USB DMA code before a transfer is invoked. See functions `dmaStartUsbOutCh0()` and `dmaStartUsbInCh1()` in `hwDmaIf.c`.

4.5.5.3 Host Ports

The IP supports two root USB ports, however in some microcontroller devices where this IP has been used only one USB root port is available, also some devices have dual instances of the IP but with only one port available on each (to support two OTG channels).

To support the case when two or more instances of the IP are used, including when the external ASSP device is used in a discrete system design the USB Host stack has been designed to be Object Orientated. This means that to use more than one Instance of the R8A66597 IP the only changes required are to create a new host controller structure and a root port structure for each root port available.

The core IP has two limitations:

- Each individual root port on this IP can support one hub only.
- A maximum of 10 devices can be addressed (two hubs and 8 devices).

The Host Driver interface file '`hwusbh.c`' includes the lowest level hardware configuration and interface functions. The ANSI open and close I/O and control functions are provided in this file. Also included are the low level interrupt functions for the USB module and the over-current detection.

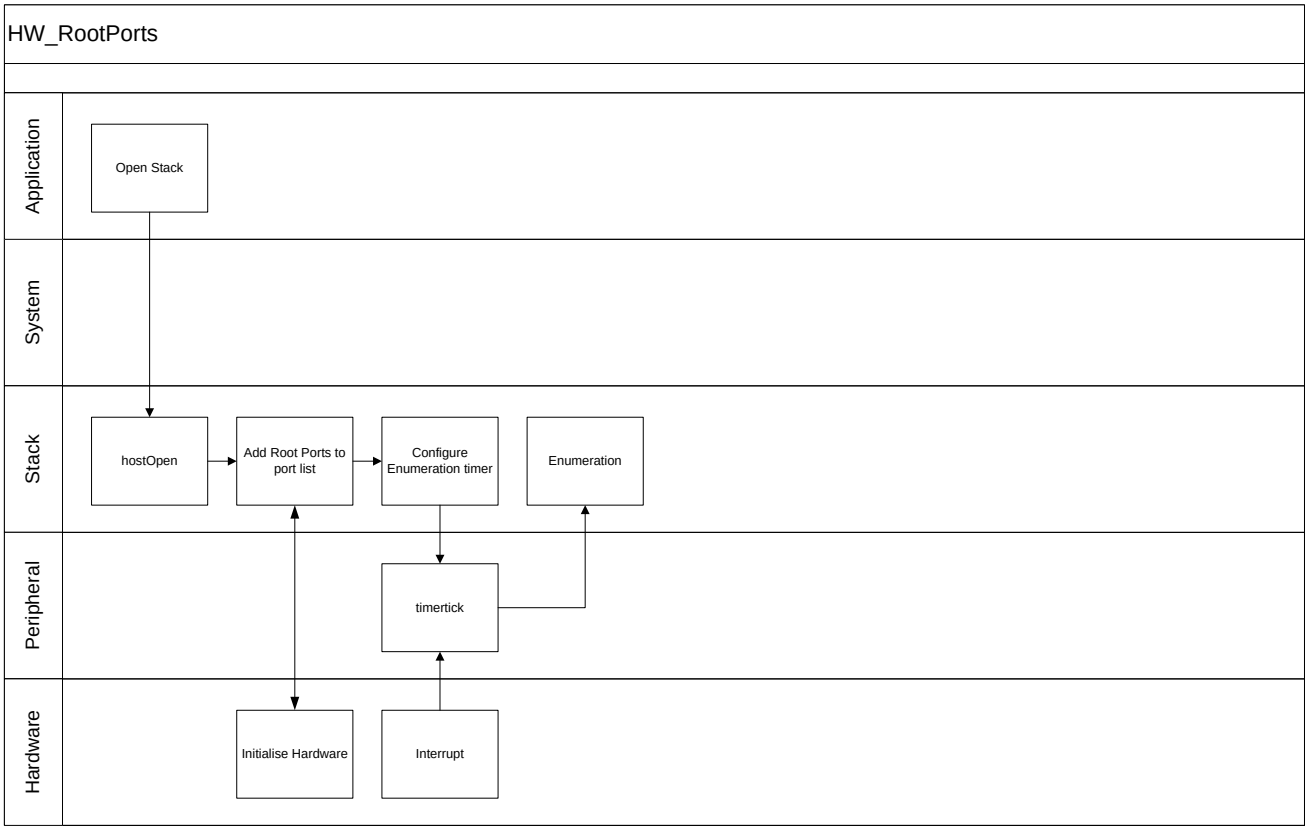

Figure 4-14 Host Port

The Universal Serial Bus forms a tree network of devices, each device has a physical port and each port supports multiple communication endpoints. The trunk of the tree is formed by the root port(s) special devices called hubs provide additional ports to form the branches of the tree. This port structure is maintained in the USB Host Stack by defining a tree of Port Information Structures (USBPI). The definition of this is below:

```
typedef struct _USBPI
{
    PUSBPI   pNext;                 /* Pointer to the next port */
    PUSBPC   pRoot;                 /* Pointer to root-port control functions */
    PUSBHI   pHub;                  /* Pointer to the hub that this port is on */
    PUSBDI   pDevice;               /* Pointer to a device attached to the port
                                       NULL if no device attached */
    uint32_t uiPortIndex;           /* The index of the port */
    uint32_t dwPortStatus;          /* The current status of the port */
    PUSB     pUSB;                  /* Pointer to the Host Controller to which
                                       the port is attached */
    PUSBHC   pUsbHc;                /* Pointer to the host controller data */
    _Bool    bfAllocated;           /* true if allocated */
} USBPI;
```

Figure 4-15 Port Information Structure

The root port(s) only, will have the pRoot member with a non-zero value as described. The USB Host Protocol driver maintains a linked list of all of the ports attached to the controller. Root ports are controlled through the functions defined in the `_USBPC` structure and ports on hubs are controlled through the protocol defined in chapter 11 of the USB 2.0 specifications.

This means that the port status request to a normal port will return a status word as defined in the USB specifications. When requesting the port status of the root port(s) the stack will call the functions in the `_USBPC` structure. These functions will return the same format status word as a standard port status request. Therefore the calling function will not see any difference in the return value. A root port will look like any other port.

Root ports are added to the Stack by calling the `AddRootPort()` function call passing in the table of function pointers.

4.5.5.4 Host Port Power Control

Depending upon the design of the target the support for root port power control may or may not be available. It is not necessary to be able to control the power output of the root port, however this should be provided to fully meet the USB specification.

The Port control functions are provided in the `hwusbh_platform.c` file in a function table:

```
const USBPC gcRootPortx =
{
    hwResetPortx,
    hwEnablePortx,
    hwSuspendPortx,
    hwStatusPortx,
    hwPowerPortx,
    x,
    &USB
};
```

Figure 4-16 Port Control Functions Structure

The power control function provides the ability to control the power state of a Root Host port. In this example the power of root port ‘z’ is controlled by a port pin on port ‘x’ pin ‘y’.

```
/******************************************************************************
Function Name: hwPowerPortz
Description:   Function to control the power to port z
Parameters:    IN  bfState - TRUE to switch the power ON
Return value:  none
******************************************************************************/
void hwPowerPortz(BOOL bfState)
{
    gbfOverCurrentPortz = FALSE;
    PORT.PxDRL.BIT.PxyDR = bfState;
}
/******************************************************************************
End of function  hwPowerPortz
******************************************************************************/
```

Figure 4-17 Port Power Control Sample

When power control is not available then the over current variable is still set in the interrupt routine, however there is no way to control the supply of power so operation is not guaranteed as it is dependent upon the physical power control device connected to the processor. When power control is implemented, the code should ensure the power is turned off in this error state. An example from the code is given below.

```
/******************************************************************************
Function Name: INT_IRQa
Description:   Over current interrupt for root port x
Parameters:    none
Return value:  none
******************************************************************************/
void INT_IRQa(void)
{
    /* Check the state of the OC detect line */
    if (PORT.PxPRL.BIT.PxyPR == 0)
    {
        /* Switch the power to the port off */
        PORT.PuDRL.BIT.PuvDR = 0;
        gbfOverCurrentPortx = TRUE;
    }
    /* Clear the interrupt flag */
    INTC.C0IRQRR.BIT.IRQaF = 0;
}
/******************************************************************************
End of function  INT_IRQa
******************************************************************************/
```

Figure 4-18 Port Over-current Sample

When there is no monitoring of the status of the USB power supply from the external devices, this interrupt is not required. The code can be configured to assume that power is always applied by setting the **gbfOverCurrentPortx** variable to FALSE. The interrupt code can be removed in this case.

4.5.5.5 Host Database

The host driver contains a structure that provides a database of the current USB connection topology. This database is updated whenever a relevant event occurs including device attach and removal events and at the end of each transfer for

the data PID. The database is global to the stack but only accessed via the pointers contained in the root Host Controller structure. This provides all layers with access to the information needed during operation.

The code structure of the database root can be found in `ddusb.h`.

This database is not visible outside of the core stack code. No user code may attempt to modify this database in any way.

As shown in the diagram above the Host Stack Database is effectively in three sections. Each of these is described below:

4.5.5.6 Device Management

The USB specification defines Ports, Hubs, Devices and Endpoints. To describe this, we can consider a hub.

A hub can have four ports. Common configurations are for four or seven ports. A seven-port hub is formed by internally connecting two hubs together therefore resulting in seven and not eight ports. The IP only supports one tier of hubs so seven port hubs will not be supported. The host will have one or more root ports. Each root port can support either a normal device or a special device called a hub. The hub will be connected to the root port and will provide additional port resources.

A device connected to any port will contain at least one endpoint. The control endpoint must exist and is used to enumerate the device on the USB. Other endpoints may be provided in the device to support one or more destinations for USB data class transfers.

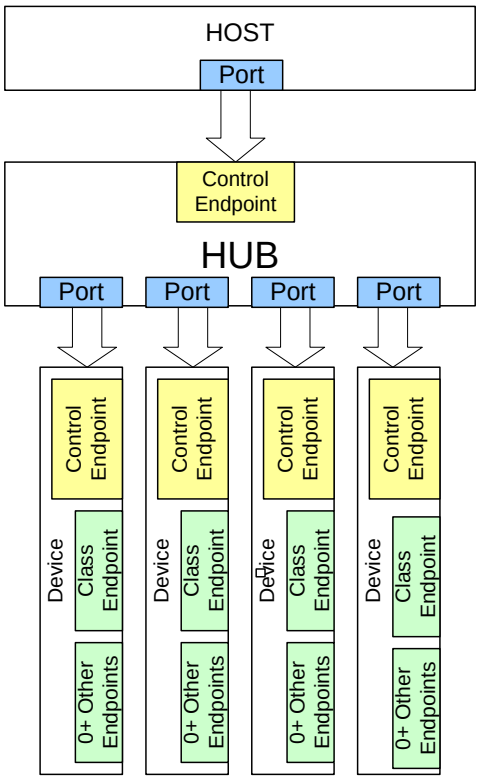

Figure 4-19 HUB Topology

The database maintains various lists. Devices and hubs are referenced to a specific port. Ports and endpoints are added and configured to a list dynamically depending upon the device attached.

4.5.5.7 Transfer Management

The transfer of data to a device is achieved using the endpoints discovered in the device during enumeration. There are four classes of transfer in the USB specification. Control transfers are used to configure a device. The other three classes are used to transfer data that has different Quality of Service requirements.

Bulk transfers are designed to transfer large quantities of data reliably between two endpoints.

Interrupt transfers are designed to transfer very small packets of data with guaranteed latency. This is commonly used in user interface transfers.

Isochronous transfers are designed to transfer data that is time critical. Transfers that fail for any reason are never retried. This is suitable for streaming applications such as audio playback. Missing data results in a “glitch” in the playback but is not critical to the application.

For each of these four transfer types the Stack maintains a list of transfers waiting to happen. The stack will send the next data block as soon as possible considering the priority of the transfer type.

4.5.5.8 Scheduling Transfers

Transfers are generated in a transfer request structure. This data structure is added to the specific transfer type list using a “Start Transfer” function call in the device API. When a transfer request is made the Stack can be forced to schedule the new transfer by calling a call back function from the appropriate device driver. This will happen anyway during the 1ms Start of Frame interrupt service.

4.5.5.9 Hardware Driver

This layer of the stack provides interface form the processing level of the stack to the hardware peripheral interface. It submits the transfer requests provided by upper level to the host/function peripheral. The API is defined in the file “`usbhHostApi.h`”.

The hardware driver is provided with pointers to the head of four linked lists of transfer requests, one for each of the different transfer types. The driver processes the requests, updating the member variables `stIdx`, `uiTransferLength`, `errorCode` and `bfComplete`. When the transfer has completed or an error occurs the request is removed from the list by calling the function `usbhComplete`. It is the responsibility of the hardware driver to schedule the transfers on the USB bus in accordance with the USB specifications.

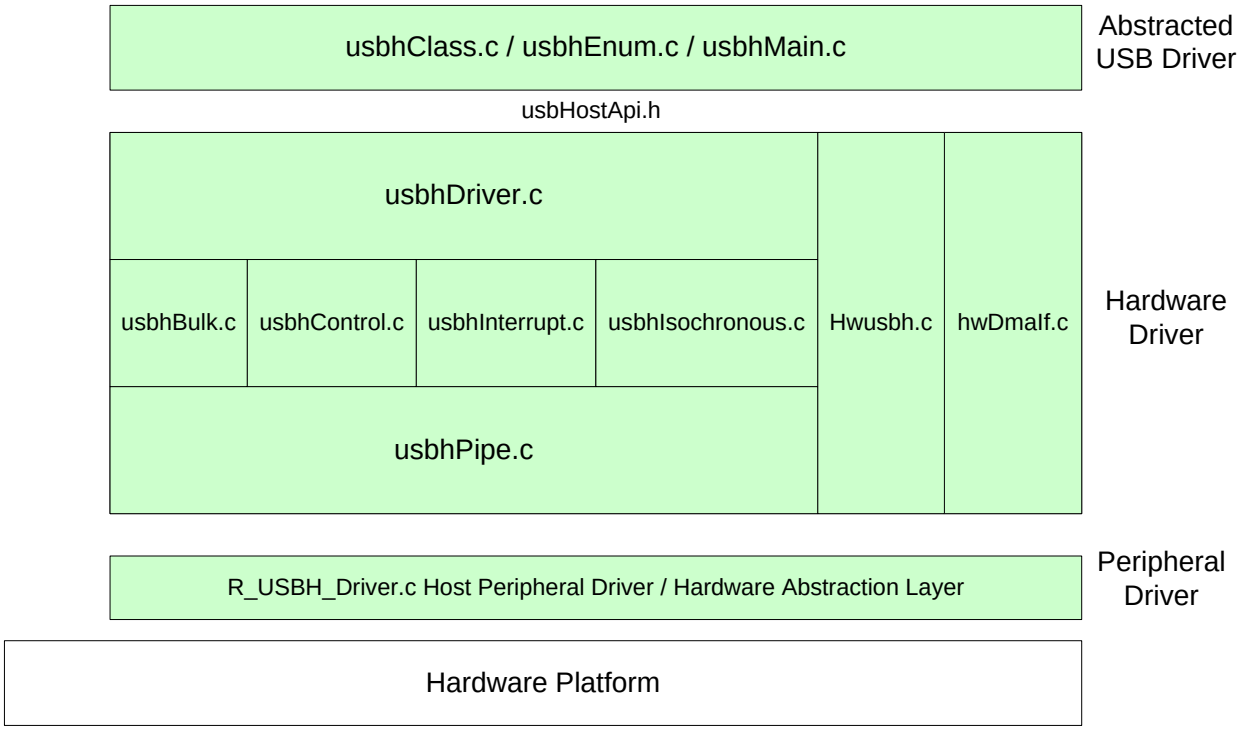

Figure 4-20 Hardware Driver Overview

4.5.5.10 USB Pipes

The following description comes from the USB Specification Version 2.0:

"A USB pipe is an association between an endpoint on a device and software on the host. Pipes represent the ability to move data between software on the host via a memory buffer and an endpoint on a device."

It is worth noting in addition that each pipe is unidirectional and can either be used to send or receive data to / from a device depending upon its configuration. The Host Controller driver is provided with controlled access to USB pipes by the APIs provided in `usbhPipe.c`.

The hardware driver provides a method to assign any available pipes to the stack for any particular transfer. In this specific case nine pipes can be allocated a transfer type. Pipe zero is fixed function and cannot be allocated. Options for pipes are defined in the USB specification as Interrupt, Isochronous and Bulk. Control transfers are always completed on the reserved pipe zero. Some pipes have limitations to the transfer types supported on each. The limitations specific to this device are:

PIPE0: Control transfer (default control pipe: DCP), 64-byte fixed single buffer

PIPE1 and PIPE2: Bulk transfers/isochronous transfer, continuous transfer mode, programmable buffer size. (up to 2-kbytes: double buffer can be specified)

PIPE3 to PIPE5: Bulk transfer, continuous transfer mode, programmable buffer size. (up to 2-kbytes: double buffer can be specified)

PIPE6 to PIPE9: Interrupt transfer, 64-byte fixed single buffer

The selection of the pipe to use for a transfer is abstracted in this module so that the user has no need to be aware of these limitations. The function `usbhAllocPipeNumber` will attempt to allocate an appropriate pipe on demand according to the transfer type stored in the transfer request. It will then return a reference to the allocated pipe if one is available. Note that control transfers are not included. In this way multiple devices can share the limited pipe resources.

A pipe is configured by allocating it to an end-point. An endpoint is configured from a class driver and provides routing information for the data to be sent or received through a pipe. The endpoint information is passed to the pipe configuration functions in a structure called `_USBEI` defined in `ddusbh.h`. The structure is given below:

Size	Name	Description
`PUSBEI`	`pNext`	Forward reference to the next endpoint entry in a list
`PUSBDI`	`pDevice`	Pointer to the device the endpoint belongs to.
`WORD`	`wPacketSize`	The endpoint packet size
`BYTE`	`byEndpointNumber`	The device endpoint number
`BYTE`	`byInterval`	The polling interval for interrupt transfers
`USBTT`	`transferType`	The type of transfer
`USBDIR`	`transferDirection;`	The direction of transfer
`USBDP`	`dataPID`	The DATA0/1 PID
`BOOL`	`bfAllocated`	TRUE if allocated

Table 4-6 USB Endpoint Structure

4.5.5.11 USB Bulk Transfers

The bulk data transfer method provides the ability to transfer large quantities of data to and from an endpoint. These transfers are detailed in a transfer request data structure. The transfer is routed to a specific pipe to an endpoint. When the transfer is configured a pipe will be allocated. This pipe reference is then provided to the bulk data handling functions.

To provide efficiency in data transfers the bulk driver will attempt to start a DMA transfer for any bulk data transaction. As long as there is an available DMA channel, the DMA will be used for the majority of the transfer. DMA transfers will only be used when the quantity of data is greater than four times the USB endpoint packet size. These full packets will be sent using the DMA controller. Data that is smaller than two times the USB endpoint packet size, such as at the end of a bulk transfer or if the bulk transfer is too small then the driver will process the data by directly loading the FIFO (FIFO configuration is completed in the `usbhPipe` module).

4.5.5.12 USB Control Transfers

The following description comes from the USB Specification Version 2.0:

"Control transfers allow access to different parts of a device. Control transfers are intended to support configuration/command/status type communication flows between client software and its function. A control transfer is composed of a Setup bus transaction moving request information from host to function, zero or more Data transactions sending data in the direction indicated by the Setup transaction, and a Status transaction returning status information from function to host. The Status transaction returns "success" when the endpoint has successfully completed processing the requested operation."

Control functions are scheduled along with all other transfers by the USB Host Driver function. Critically only one control transaction is permitted at any time on the USB bus by the USB IP. This exclusion is enforced in the USB Host Driver.

The USB Control module (`usbhControl.c`) provides the control pipe transactions. The functions are directly related to request, setup, data in, data out and completion with error processing for failed transactions. Each function requires the request data structure to be provided.

4.5.5.13 USB Interrupt Transfers

The following description comes from the USB 2.0 Specification:

"The interrupt transfer type is designed to support those devices that need to send or receive data infrequently but with bounded service periods. Requesting a pipe with an interrupt transfer type provides the requester with the following:

- *Guaranteed maximum service period for the pipe*
- *Retry of transfer attempts at the next period, in the case of occasional delivery failure due to error on the bus*

Interrupt transfers will occur on the USB when the transfer request is made with the appropriate transfer type selected. The process for starting and stopping an interrupt transfer is the same as for the Bulk transport."

4.5.5.14 USB Isochronous Transfers

The following description comes from the USB specification version 2.0:

"*In non-USB environments, isochronous transfers have the general implication of constant-rate, error tolerant transfers. In the USB environment, requesting an isochronous transfer type provides the requester with the following:*

- *Guaranteed access to USB bandwidth with bounded latency*
- *Guaranteed constant data rate through the pipe as long as data is provided to the pipe*

In the case of a delivery failure due to error, no retrying of the attempt to deliver the data

The sample application provided with the stack uses the isochronous transfer type to stream audio data to a basic USB Audio compliant endpoint. The process for starting and stopping a interrupt transfer is the same as for the Bulk transport. As per the specification any errors in the transfer will not be retried."

An additional driver API function is also provided for this transfer type. This API provides the ability to specify an average transfer rate for cases where the number of bytes transferred on each successive.

4.6 Website

The RZ/A1H Software package implements a website using an implementation of the open source lwIP TCP/IP Ethernet stack. Alongside the Ethernet Stack is the implementation of the open source web server. The below image shows the software architecture for the website.

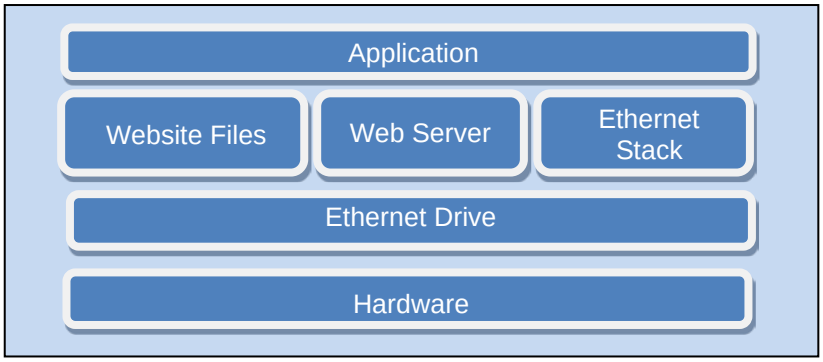

Figure 4-21 Website Hierarchical Software Layer

4.6.1 Software Configuration

To enable, set `R_SELF_LOAD_MIDDLEWARE_ETHERNET_MODULES` to `R_OPTION_ENABLE`. To disable, set it to `R_OPTION_DISABLE`. See 3.1 Configuration for further details.

```
/** Enable Ethernet drivers, WebServer Support  */
#define R_SELF_LOAD_MIDDLEWARE_ETHERNET_MODULES (R_OPTION_ENABLE)
```

4.6.2 Commands

The Ethernet commands are split into two sections. This first section provides the normal configuration required to suit the infrastructure available at the point of installation. This provides the ability force the IP addressing settings.

Command	Description

ipconfig –o
 	
ifconfig –o
	Where o is one of the or more following options
	`-r = Reset to default settings`
	`-i:xxx:xxx:xxx:xxx = Set IP address`
	`-m:xxx:xxx:xxx:xxx = Set IP address mask`
	`-g:xxx:xxx:xxx:xxx = Set DHCP server gateway address`
	`-on = Enable DCHP`
	`-off = Disable DCHP`
	`-s = Save current settings (Must follow the settings to be saved)`
	`-all = Display current settings`
	`(no option) = Command list display`

setmacaddress –o	Caution - Changing the MAC address should not be completed unless you are restoring the allocated number or have your own address range allocated from the relevant authority.
	Where o is one or more of the following options
	`-a:xx.xx.xx.xx.xx.xx = Set MAC address`
	`-s = Save current settings (Must follow the settings to be saved)`
	`(no option) = Command list display`

readrom	Read's the on-board EEPROM
rstrom	Reset's all the user data in the EEPROM.

Table 4-7 Ethernet Commands

4.6.3 Application

Ensure that the Renesas Starter Kit + for RZ/A1H Platform is connected to a Local Area Network and that the software package is configured.

The Application uses a DHCP protocol to retrieve the IP Address. If configured correctly the details of the connection will be displayed on the terminal on the start-up of the device.

Figure 4-22 Website Application

Once the Address is retrieved, '192.0.2.9' in the above example the user may access the address on their local network.

The first page of the website is the welcome page. This page provides the user with the information about this product and allows navigation to other pages. The left-hand side widget gives the user the time and date retrieved from the RZ/A1H device. Furthermore, the user may toggle the 'USER' LED0 through enabling and disabling the tick box.

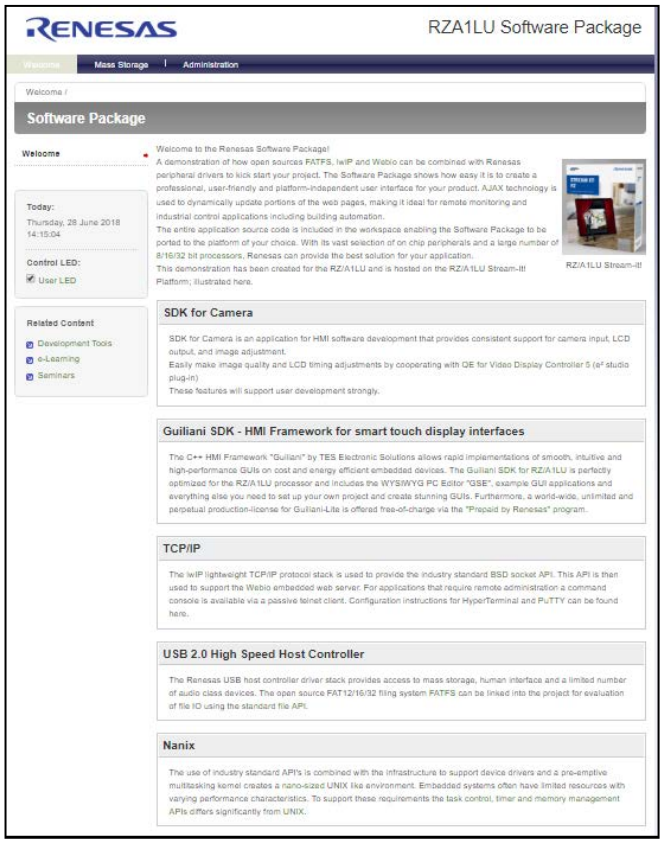

Figure 4-23 Welcome page

The 'Mass Storage' page allows for the user to interact with the connected mass storage device. In the image below, the two files may be manipulated.

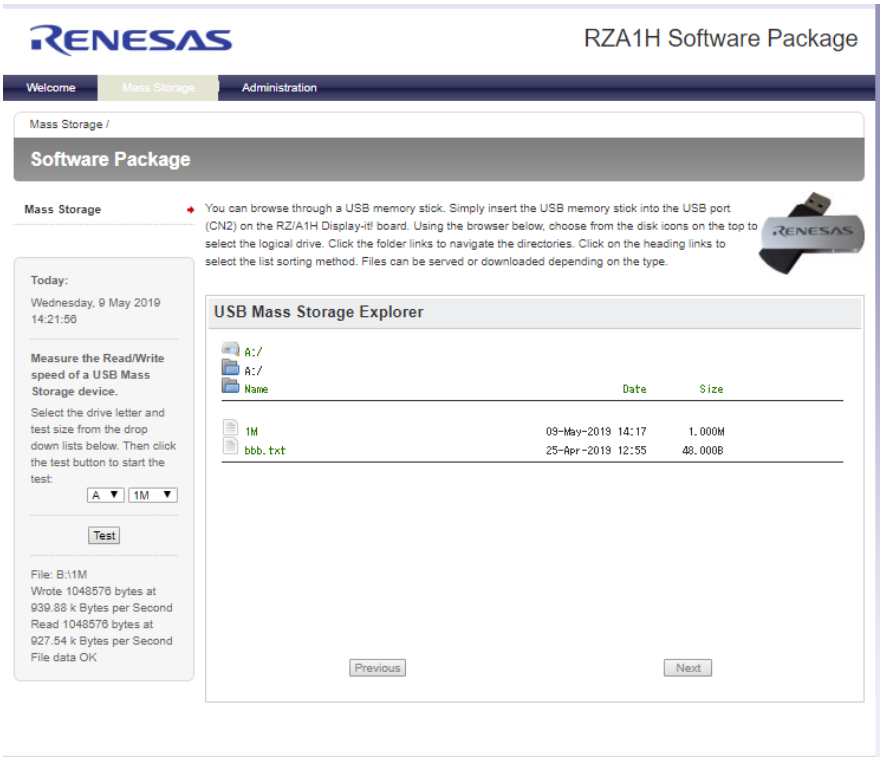

Figure 4-24 Mass Storage Page

The user may also go into the ‘Administration’ page. This page requires for the user to have a user name and password. The username and password is stored in the on-board EEPROM.

The default details are:

Username: Admin

Password: Password

Once successfully logged in the user may then configure the clock, username and password.

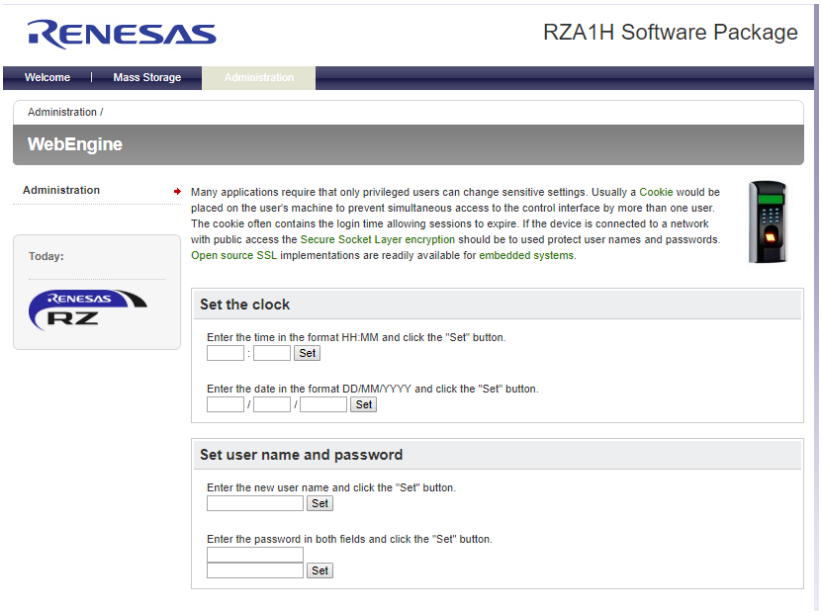

Figure 4-25 Administration Page

4.6.4 Ethernet Stack

The Ethernet Stack provided by the lwIP TCP/IP is accessed through Berkeley sockets Application Programming Interface (BSD socket API). This BSD-API comprises a library for developing applications in the C programming language that perform communications across a computer network.

Note: When lwIP is configured to provide the BSD socket API it is not as efficient with memory and it requires a multi-tasking system.

The software application uses lwIP with the BSD socket API enabled. The demonstration uses the Renesas abstraction layer to provide the multi-tasking function required by lwIP.

The file "`lwIP_interface.c`" provides the interface between the open source lwIP stack, the network driver(s) and the application. The application interface provides functions to control and reconfigure the IP connection (MAC address, IP Address, Gateway Addresses and Address Mask). This interface is object oriented to support more than one network driver.

The Ethernet adapter has two configuration objects to describe the fundamental configuration of the Ethernet controller which are the MAC address and the IP configuration. The MAC address should be unique to each Ethernet controller and a unique valid MAC address is supplied with each hardware kit.

The IP configuration data type NVDHCP is used by the lwIP interface, among others, to transport the current IP configuration from function to function within the application. The data members for this object are as follows:

- pbyIpAddress: local device IP Address
- pbyAddressMask: local device Subnet Mask
- pbyGatewayAddress: local device Gateway Address
- byEnableDHCP: overide local settings with configuration from DHCP server flag.

When byEnableDHCP is non-zero, the local configuration as described by pbyIpAddress, pbyAddressMask and pbyGatewayAddress are ignored and instead lwIP will request an assignment of IP address from the local DHCP server. If there is no DHCP configured to assign IP addresses, the Ethernet control will fail to initialise.

```
/* Define the data structure for the NVDT_DHCP_SETTINGS_V1 data type */
typedef struct _NVDHCP
{
    uint8_t         pbyIpAddress[4];        // IP Address e.g. 192.168.1.1
    uint8_t         pbyAddressMask[4];      // Subnet Mask e.g. 255.255.255.0
    uint8_t         pbyGatewayAddress[4];   // Gateway Address e.g. 192.168.1.10
    uint8_t         byEnableDHCP;           // Use DHCP assigned settings

} NVDHCP,
*PNVDHCP;
```

Figure 4-26: Configuration Object

During the initialisation of the lwIP stack, the scheduler creates four specific tasks to interface with the network driver.

LwIP tcpip_task – lwIP main task, handles all lwIP activity (see lwIP open source documentation for operation of this task)

- EtherC Link Mon: Link monitor task, handles changes in state of the network connection
- EtherC Input: Read task, handles reading data from the network driver
- EtherC Output: Write task, handles writing data to the network driver

4.6.3.1 Interaction Between Read/Writing Tasks and the lwIP Task

All data is passed between the controller and the lwIP task in the form of buffers (known as PBuffs) created and managed by lwIP. Task ‘ipInputTask’ is responsible for reading data from the controller and passing it to the lwIP stack. Task ‘ipOutputStatus’ is responsible for requesting lwip to free the PBuffs when the network driver has transmitted the data. The lwIP task is responsible for creating and destroying the PBuffs. The ipOutput function (called by lwIP) is responsible for writing the PBuffs to the network driver (see Figure 4-27 Relationship of tasks).

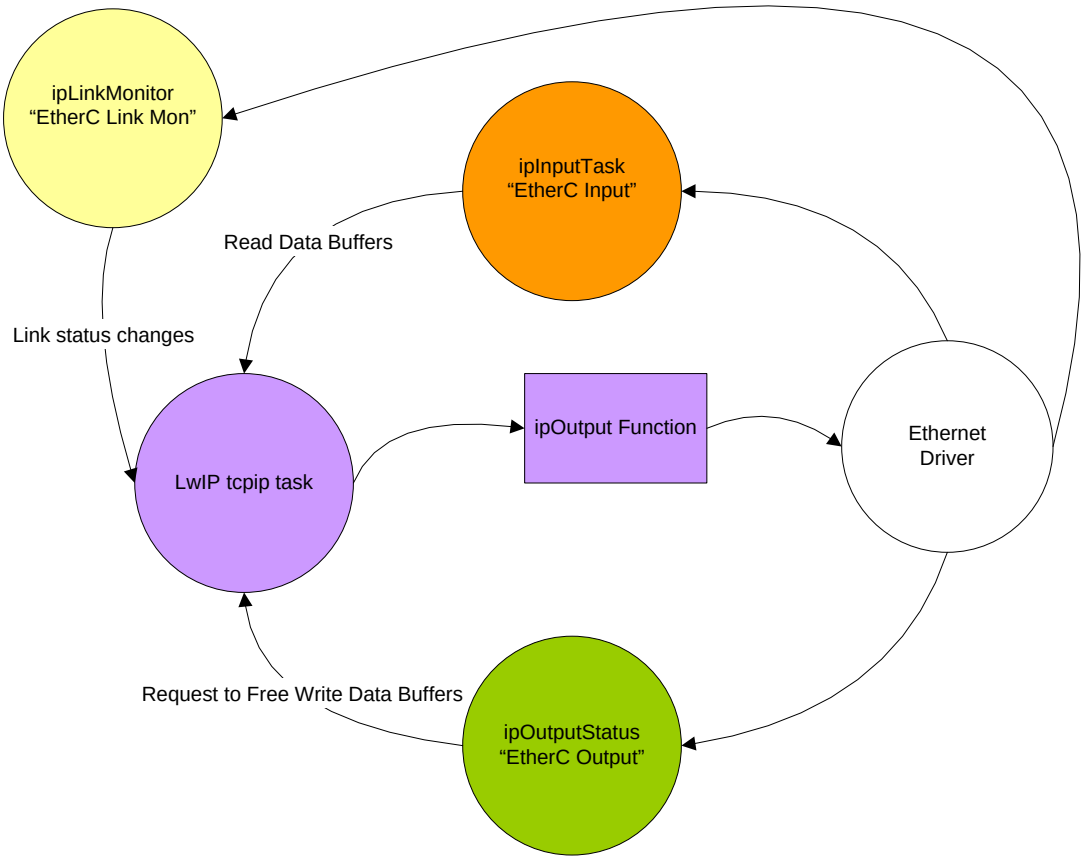

Figure 4-27 Relationship of tasks

4.6.3.2 IpLinkMonitorTask Details

The IpLinkMonitor task waits for the driver layer to set an event which signifies that the status of the network connection has changed. In response to this event the task decides what actions need to be taken with the lwIP interface (see Figure 4-28 IpLinkMonitor task Overview, note that messages sent to initiate action within the lwIP task have a 'MSG lwip:' pre-fix).

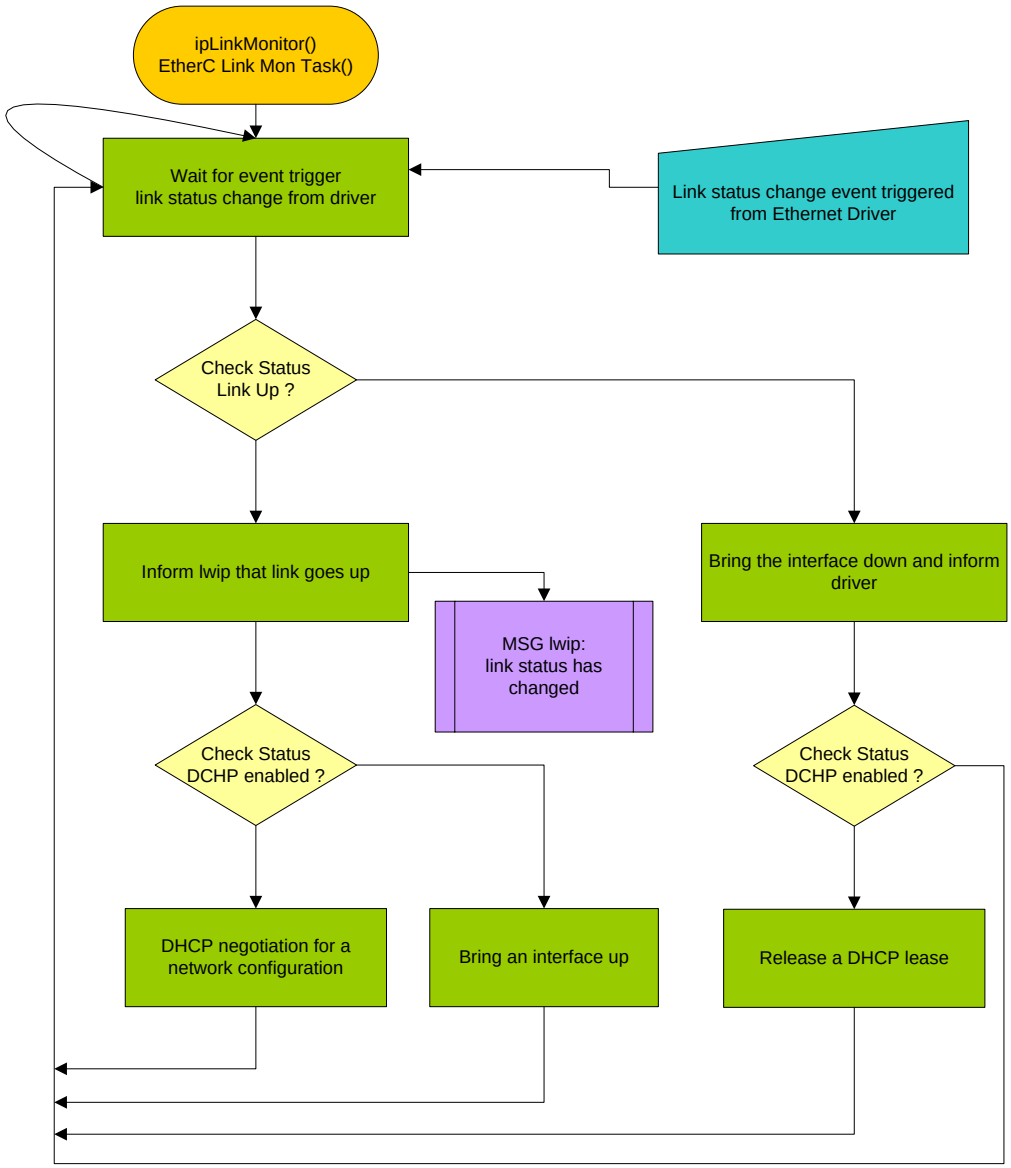

Figure 4-28 IpLinkMonitor task Overview

4.6.3.3 IpInputTask Details

The ipInputTask task allocates and submits the buffers (PBuffs) to the Ethernet driver. The task waits for the ETHERNET_SIMULTANEOUS_READS event, which signifies that at least one read has completed. Once the event has been set the task will check that the read completed without error and then pass it to lwIP for processing.

The Ethernet controller supports a number of simultaneous read events, definition is called _NET_MAX_RX_ defined in file inc\hwethernet.h, to increase throughput (see, Figure 4-29 ipInputTask Overview note that messages sent to initiate action within the lwIP task have a 'MSG lwip:' pre-fix).

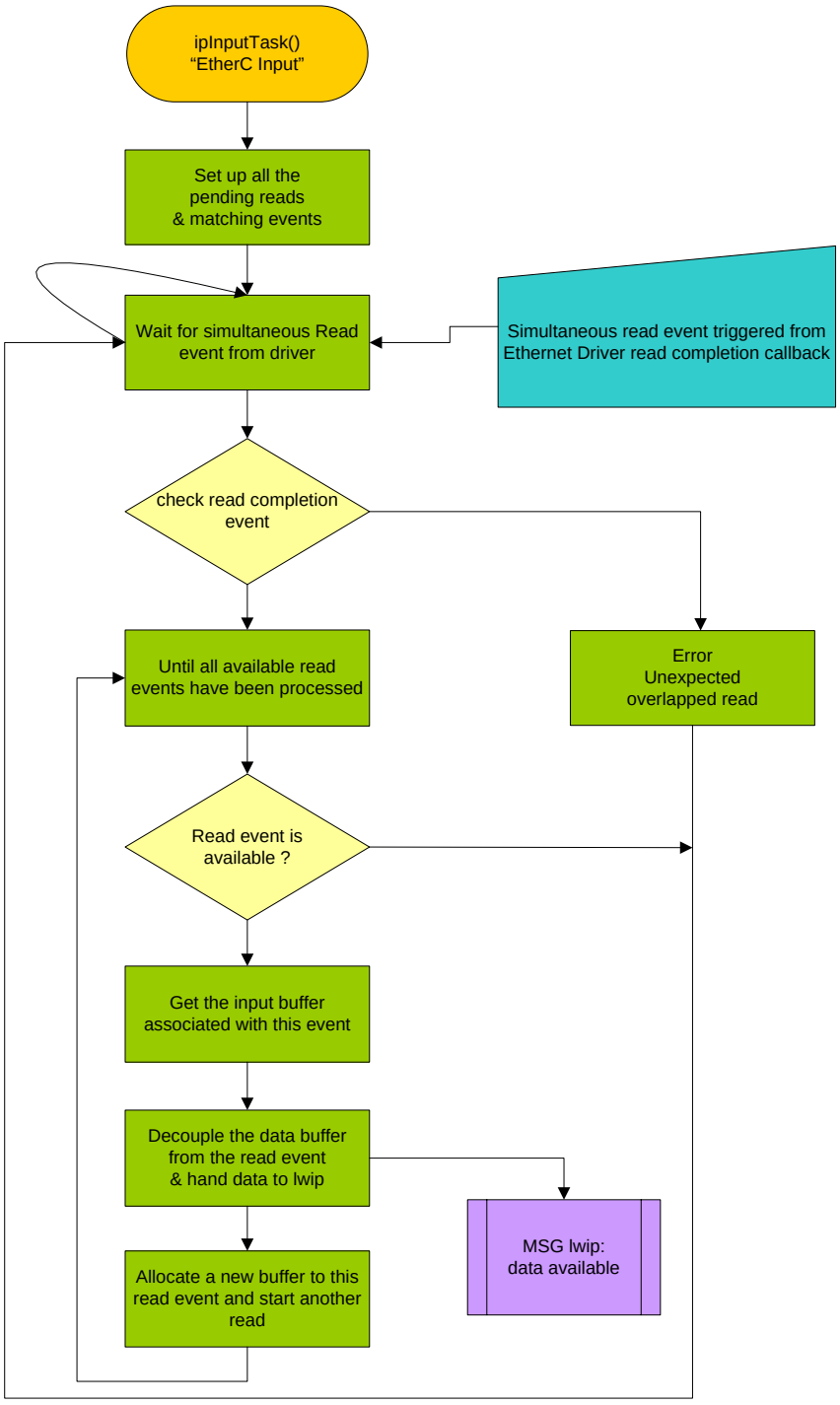

Figure 4-29 ipInputTask Overview

4.6.3.4 IpOutputStatus Details

The IpOutputStatus function frees the buffers (PBuffs) which have been allocated by lwIP after the Ethernet driver has completed sending the data. The task waits for the network interface driver to signal an event which signifies that at least one buffer has been written. Once the event occurs the task will delete the buffer (PBuff) associated with the data. It should be noted that every buffer has a reference count and the memory is freed when the reference count reaches zero. Although the output status task has freed the buffer it is not until an ACK is received that the reference count will reach zero and the memory will be freed.

The Ethernet controller supports a number of simultaneous write events, definition is called _NET_MAX_TX_ defined in file "`inc\hwethernet.h`", to increase throughput (see Figure 4-30 IpOutputStatus Overview, note that messages sent to initiate action within the lwIP task have a 'MSG lwip:' pre-fix).

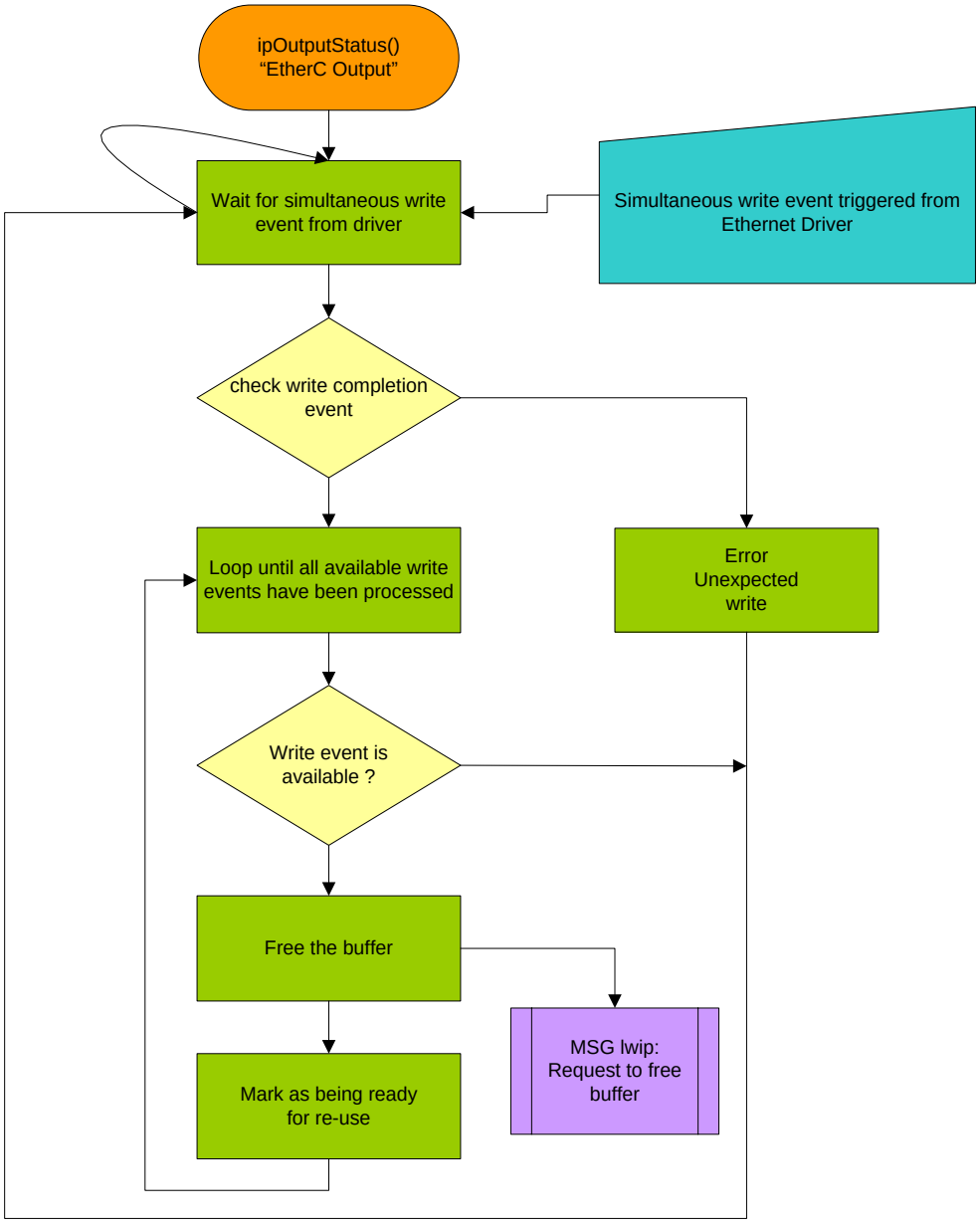

Figure 4-30 IpOutputStatus Overview

4.6.3.5 Stack Initialization Sequence

The Ethernet stack is initialised by opening the interface using the STDIO open function and the device name '`Ether0`'. This is performed by the function StartOnChipController(). The function reads the existing configuration, stored in non-volatile storage, performs some basic sanity checking on the supplied MAC address and if the MAC address is acceptable the function proceeds to load the network configuration from non-volatile storage and uses the stored configuration to start the lwIP task.

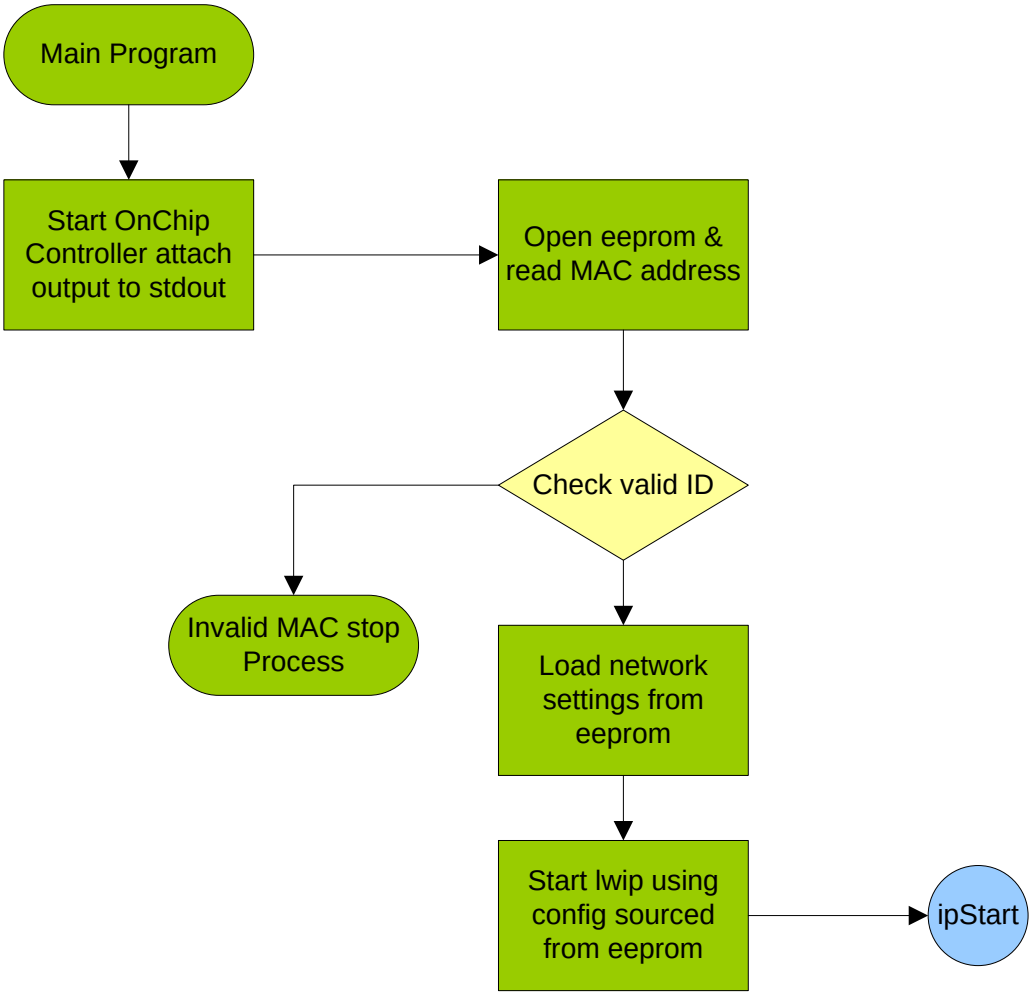

Figure 4-31 Initialisation Sequence

4.6.3.6 Callback Function ipInitialise

One of the parameters passed to the netif_add() function that is used by ipStart to initialise the lwIP driver interface is the function ipInitialise(). The purpose of this function is to initialise the lwIP device interface. The function ipInitialise() is responsible for creating the interface tasks (‘ipInputTask’, ‘ipOutputStatus’ and ‘ipLinkMonitor’). It also tells lwIP the host name, the link capabilities and provides a pointer to a function that lwIP can use to write data to the network driver.

4.6.3.7 Purpose of the Interface

This interface is to allow the BSD sockets API to be used with the file descriptors to allow the use of the ANSI I/O library. The effect of this change is to enable an internet socket to be treated as a file stream like “stdin” and “stdout”. The main application of this is so the command line console code can be re-use for the passive telnet server. The file `drvSocket.c` is a driver that wraps the lwIP BSD socket API functions *lwip_close()*, *lwip_read()* and *lwip_write()*. This allows the ANSI C standard IO library functions to read and write to a socket. A special function called faccept() wraps the function lwip_accept() and returns a file pointer instead of a socket. This allows the standard C input and output file formatting function to be used directly on the socket.

4.6.3.8 Berkeley Socket Interface

lwIP uses a subset of the Berkeley socket interface is an application programming interface (API) to code applications performing communication between hosts or between processes on one computer, using the concept of an Internet socket. It can work with many different Input/output devices, although support for these depends on the operating-system implementation. This interface implementation is the original API of the Internet Protocol Suite (TCP/IP). The wrapper interface is defined in the following files:

File name: `src\socket.c`	Wrapper source code
File name: `src\drvSocket.h`	Wrapper driver
File name: `inc\socket.h`	Wrapper interface

4.3.6.9 Socket API Wrapper Functions

These functions are used by the user process to send or receive packets and to do other socket operations.

socket():	Creates a new socket of specified type and allocates system resources for it.
bind():	Associates a socket with a specified socket address structure, i.e. specified local port number and creates a new socket of specified type and allocates system resources for it.
listen():	Causes a bound TCP socket to enter a listening state.
connect():	Assigns a free local port number to a socket. With TCP sockets, attempts to establish new connection.
accept():	Accepts a received incoming attempt to create a new TCP connection from the remote client, and creates a new socket associated with the socket address pair of this connection.
faccept():	Identical to accept() function but returns a file pointer instead of a socket ID.
setsocketopt():	Sets a particular socket option for the specified socket.
getsocketopt():	Retrieves the current value for a particular socket option for the specified socket.
getpeername():	Retrieves the name of the connected peer socket.
getsockname():	Retrieves the name of the socket.
ioctl():	Control of additional commands not supported by socketopt() (only FIONREAD & FIONWRITE) are implemented in this stack.
select():	Classify the list of sockets into three lists (ready to read, ready to write, sockets with exceptions).
send():	Send data to a remote known socket, can only be used when socket is in connected state.
sendto():	Send data to a remote specified socket.
recv():	Receives data from a remote known socket, can only be used when socket is in connected state.
recvfrom():	Receives data to a remote specified socket.
shutdown():	Closes connection, causing system to release resources. Terminates a TCP connection.

4.6.5 WebIO Server and Files

The WebIO server is a third party small portable web server designed as a library for inclusion in embedded systems or as a Browser-based GUI in applications.

For more details about the WebIO please refer to the program.html found at: `src\webio`:

4.6.6 WebIF

The software package includes an interface with WebIO. This middleware can be found at: `src\renesas\middleware\Webif`. The interface allows for the dynamic content to be created by CGI and SSI. An overview of the files.

`Webif.c` – This file initiates the server and the webif task.

`webSSI.c` – Allows for time and date to be shown on the website.

`cgiMsExplore.c, efsWebSites.c efsFile.c` – Implements the browsing of the files and folders found on the USB mass storage.

4.6.7 Website Files

For static web development the HTML, CSS and javascript source files for the website are included in the folder `src/renesas/application/WebSite`.

4.7 Camera

The RZ/A1H which supports camera input has peripheral devices VDC5, and NTSC. The VDC5 and NTSC are for analogue camera input. The VDC5 also supports image output to display devices. This sample program offers sample applications for camera input which involves using those peripheral devices. For details about these peripheral devices, refer to the User's Manual (Hardware).

This sample program has two tasks, graphics processing task that performs initial setting of camera input, display output, and image adjustment, and CUI (Character User Interface) task for performing image adjustment.

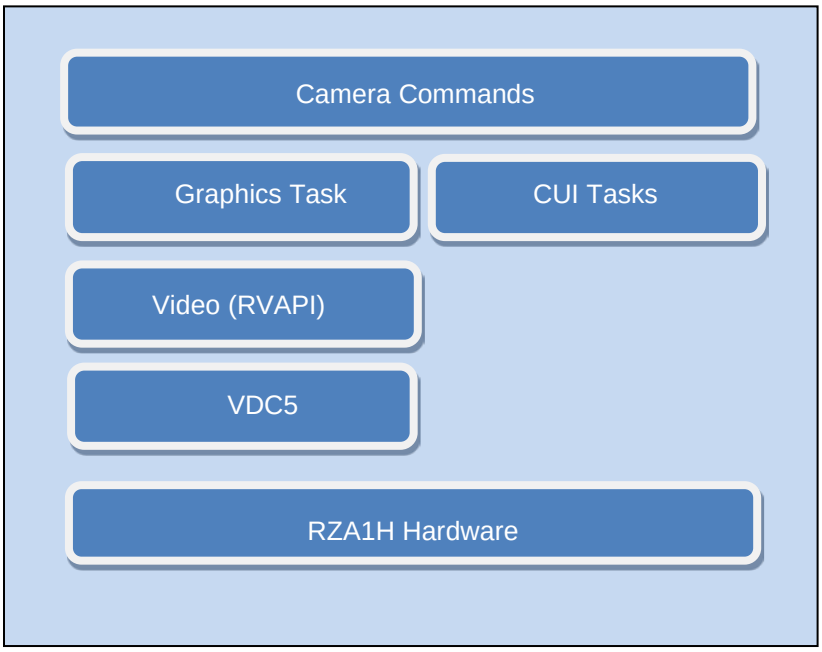

Figure 4-32 Camera Hierarchical Software Layer

4.7.1 Software Configuration

To enable, set `R_SELF_INSERT_APP_SDK_CAMERA` to `R_OPTION_ENABLE`. To disable, set it to `R_OPTION_DISABLE`. See 3.1 Configuration for further details.

```
/** Enable control for src/application/app_sdk_camera sample application */
#define R_SELF_INSERT_APP_SDK_CAMERA (R_OPTION_ENABLE)
```

4.7.2 Commands

The camera application is launched with the command ‘sdk’. The commands then available are described in Table 4-8.

Command	Operation
Numerical value	Operations in each menu - Selection of image adjustment content - Selection of image adjustment position (selection of hardware block for image adjustment) - Selection of presets - Input custom value
C, c	Custom setting selection (Selected when user want to set a preset other than the various adjustment items)
D, d	Set image adjustment to default (Default value of each register described in hardware manual)
B, b	Return to the previous menu
R, r	Output current image adjustment value
T, t	Return to the top menu
Enter	Terminate input command
Delete / Backspace	Delete one character from the input string

Table 4-8 Camera Commands

This sample program implements a Character User Interface (CUI) task which provides image adjustment in real time via a terminal console application running on a PC. It updates the register values of the VDC5 controller following command input to adjust image settings.

This chapter describes the screen operation method and commands from terminal application.

Note, however, that in case of adjusting images with e^2 Studio’s QE for Display, it is not permissible to combine the use of this CUI and QE for Display.

4.7.3 Menu

The display menu of the terminal application and operation overview is shown in Figure 4-33.

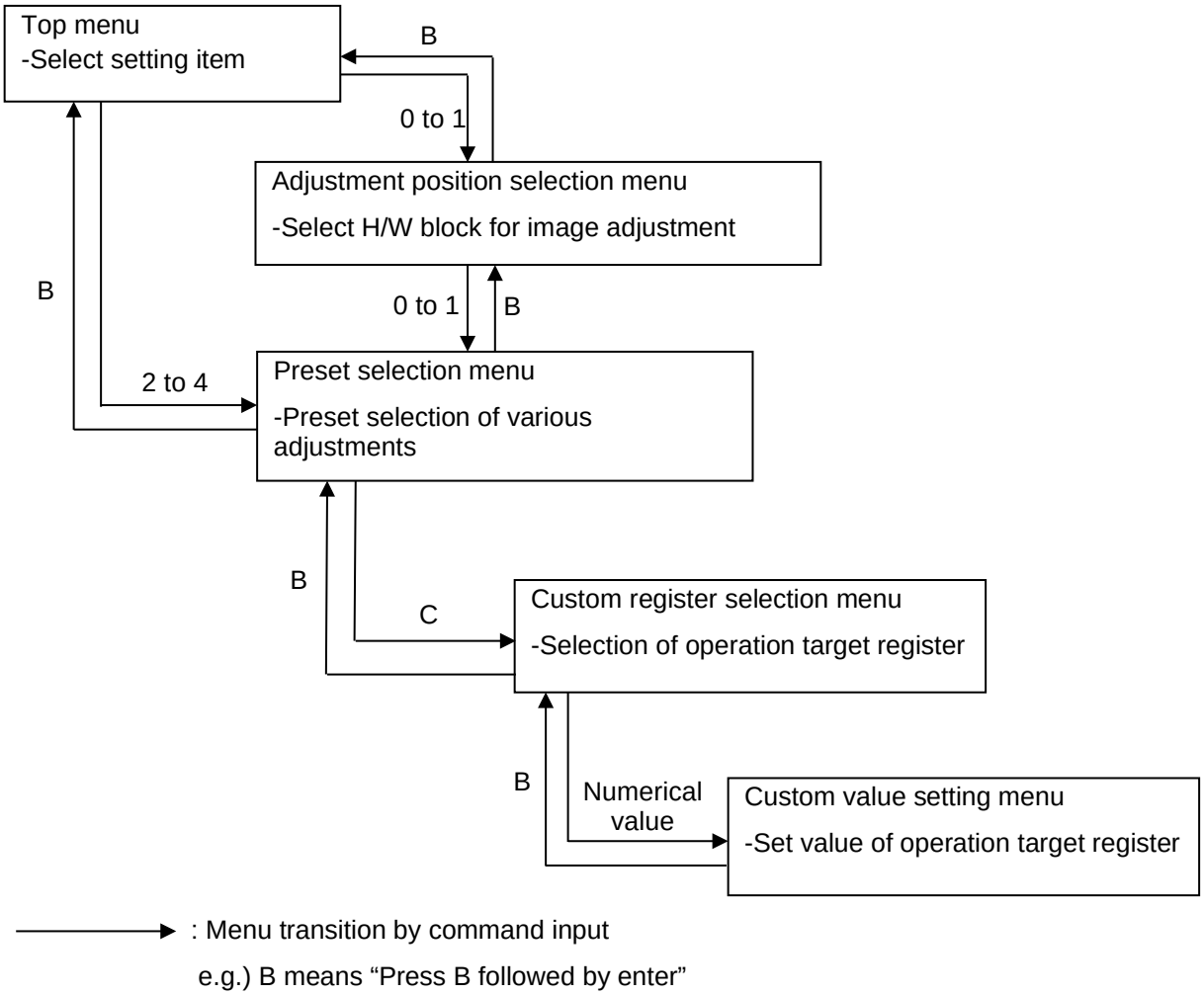

Figure 4-33 Display Menu and Operation Overview

4.7.3.1 Acquisition of Image Adjustment Value

The user can adjust the camera image settings using the menu options provided by the application. They can choose between one of several presets supplied, or enter a custom value. The current state of all of the settings can be output at any time by entering the ‘**R**’ command. When this is done, the current values of the customisable registers are output on the terminal console in a format that can be directly copied and pasted over the source code (the C header file `r_image_config.h`). These settings will become the new defaults after the application is rebuilt and run again.

4.7.3.2 Processing Sequence

Figure 4-34 shows the processing sequence of this sample program.

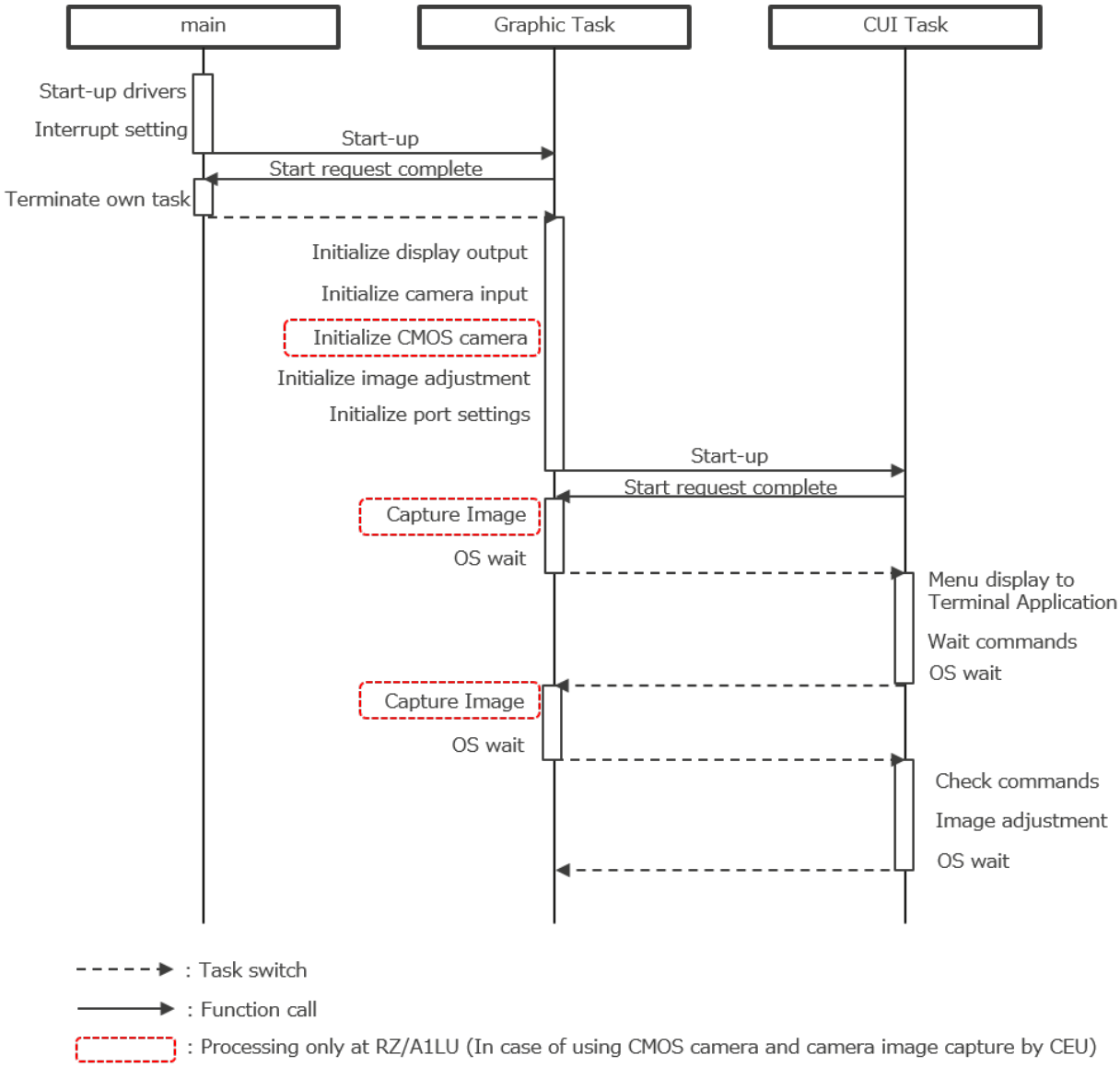

Figure 4-34 Camera SDK Processing Sequence

4.7.3.3 Image Adjustment Effects and Adjustment Methods

This sample program provides preset values for various possible image adjustments. This section describes the adjustment effects and preset values. It also shows which blocks in the H/W configuration for RZ/A1 image input/output are responsible for adjustments.

4.7.3.4 Overall Configuration

Figure 4-35 shows the hardware configuration for RZ/A1 image input/output.

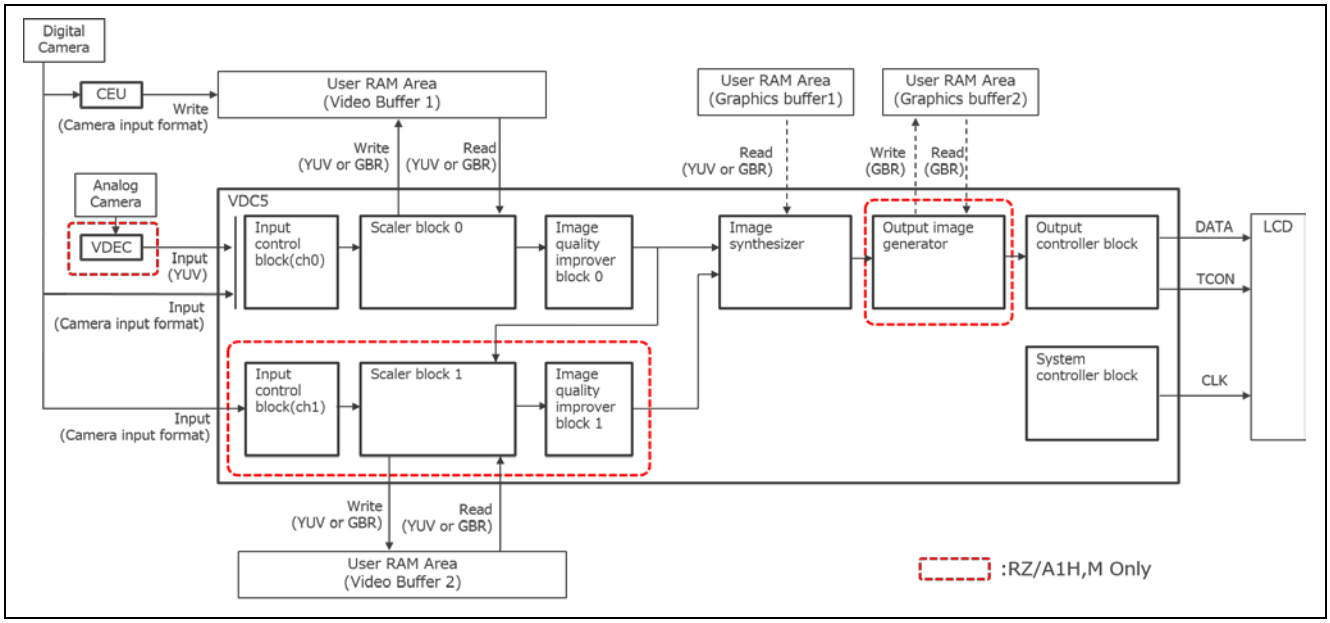

Figure 4-35 Block Diagram of the Hardware Configuration for RZ/A1 Image Input/Output

For more information regarding the camera please refer to the Application Note: SDK for Camera Sample Program (R01AN5094EJ)

4.8 USB HID

The USB function allows for HID.

USB Human Interface Device class specifies a class for devices such as keyboards, mice, game controllers and display devices. It allows for manufactures to design a product to USB HID specification and expect it to work with software of the same specifications.

The two most popular HID devices are keyboards and mice. The software package allows for processing both of these devices.

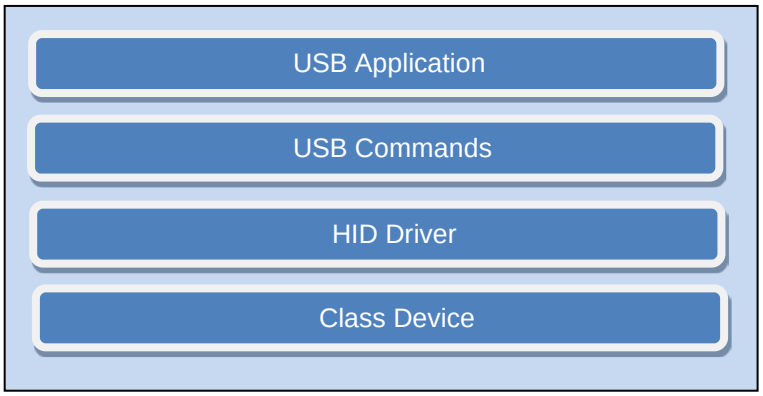

Figure 4-36 USB Function Hierarchical Software Layer

4.8.1 Software Configuration

To enable, set `R_SELF_INSERT_APP_HID_MOUSE` to `R_OPTION_ENABLE`. To disable, set it to `R_OPTION_DISABLE`. See 3.1 Configuration for further details.

```
/** Enable control for src/application/app_hid_mouse application */
#define R_SELF_INSERT_APP_HID_MOUSE (R_OPTION_ENABLE)
```

4.8.2 Commands

Table 4-9 shows the USB Functions commands.

Command	Description
usbm	Invokes the USB mouse application
usbk	Invokes the USB keyboard application

Table 4-9 USB HID Commands

4.8.3 USB HID Mouse

The USB HID mouse application allows for the user to see the coordinates and scroll wheel position of the mouse as well as highlighting which mouse button is pressed. Figure 4-37 shows the output of the USB HID mouse application.

```
REE> usbm
Mouse monitor, press any key to stop.
Left Middle Right        X        Y        S
1    0      0     +00001028 -00000193 -00009959
```

Figure 4-37 USB HID Mouse Application

The USB HID mouse application is included as part of the `cmd_usb.c` found at `src/renesas/application/console/cmd`.

Here the function cmdMonitorMouse() makes use of the HID mouse driver to access the mouse data. The HID mouse driver uses the STDIO interface as discussed in section 3.4. The driver utilises a small state machine to poll mouse data and then uses the circular buffer to store the data. The API for the driver is shown below.

4.8.4 USB HID Keyboard

The USB HID keyboard application displays key pressed on a connected keyboard. Figure 4-38 shows the output from the USB HID keyboard application.

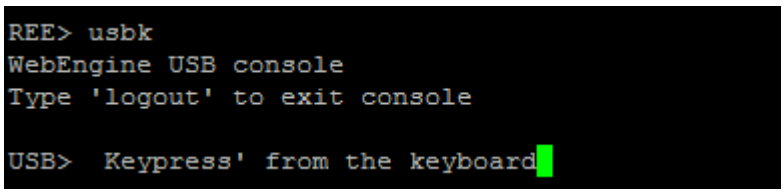

Figure 4-38 USB HID Keyboard Application

The USB HID keyboard application is included as part of the file `cmd_usb.c` which can be found at `src/renesas/application/console/cmd`. Similar to the USB HID mouse, the function cmdConsoledUSB() makes use of the HID keyboard driver to access the data gathered from the keyboard.

The HID Middleware API can be seen in Doxygen under RZ/A1H Software Package → Modules → Middleware (POSIX) → USB HID.

4.9 USB CDC

USB Communication device class may include more than one interface. Such as a custom control interface, data interface, audio, mass storage related interfaces.

The software package allows for communication with any CDC device, through a range of commands.

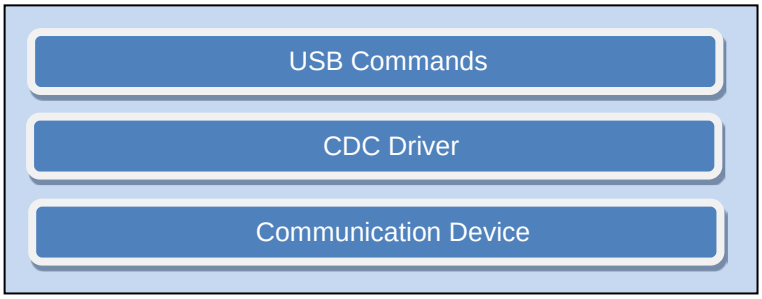

Figure 4-39 USB Function Hierarchical Software Layer

4.9.1 Software Configuration

To enable, set `R_SELF_INSERT_APP_CDC_SERIAL_PORT` to `R_OPTION_ENABLE`. To disable, set it to `R_OPTION_DISABLE`. See 3.1 Configuration for further details.

```
/** Enable control for src/application/app_cdc_serial_port application */
#define R_SELF_INSERT_APP_CDC_SERIAL_PORT (R_OPTION_ENABLE)
```

Commands Table 4-9 shows the USB CDC commands.

Command	Description
sopen	Opens a CDC device
sclose	Closes a CDC device
sctltst	Performs control API test for the connected CDC device
sttx n	Transmit n, k bytes of data through the CDC device
sloop	Loops-back received characters through the CDC device
sbaud n	Sets the baudrate to n
Scontrol n	Controls the RTS and DTR signals
sparity p	Sets the parity, p to N = none, E = Even, O = odd.
sstop s	Sets the number of stop bits to 1, 1.5 or 2
sline	Returns the line status
sbreak n	Sets / clears the break signal
stest	Tests all CDC driver function with a loop-back connector
sloopall	Performs a loop-back test on a maximum of 4 CDC devices (at a constant baudrate)

Table 4-10 USB CDC Commands

4.9.2 CDC Driver

The CDC Middleware API can be seen in Doxygen under RZ/A1H Software Package → Modules → Middleware (POSIX) → USB CDC.

4.10 Sound Application

The sound application allows the user to play sound and to playback sound through a line-in. To achieve this a twin jack headset (headphones and a line-in jack) will be needed to connect to CN20 and CN19.

4.10.1 Software Configuration

The software architecture for this application can be seen below. The R_SOUND driver manages the configuration of the audio on the Renesas Starter Kit + for RZ/A1H Board, allowing control of the headphone and line-in volume, sampling frequency etc. Audio data to and from the CODEC is sent via the SSIF driver, using DMA channels to minimize CPU overheads.

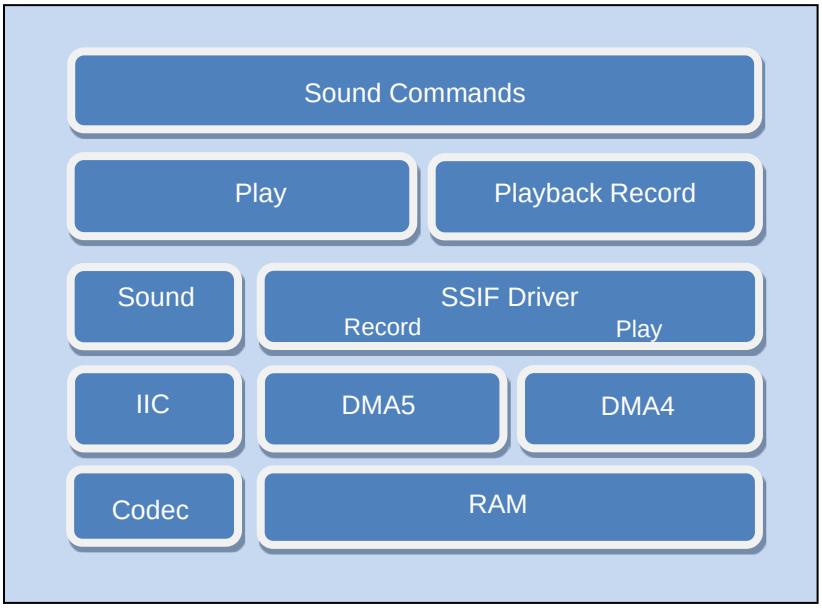

Figure 4-40 Sound Function Hierarchical Software Layer

4.10.2 Commands

The Renesas Starter Kit + for RZ/A1H Board sound demo has the following sound demonstrations available on the console:

Command	Description
play	This command allows for pre-recorded sound to come out of the audio jack. The prerecorded sound provides sound from both the left and right channels. The left headphone is a male saying '左- Hidari' and a right headphone is a female saying'右-Migi' are the Japanese words for "left" and "right
record	Allows for the sound picked up by the line-in to be played through the headphones

Table 4-11 Sound Commands

4.10.3 Audio Software Configuration

To enable, set `R_SELF_INSERT_APP_SOUND` to `R_OPTION_ENABLE`. To disable, set it to `R_OPTION_DISABLE`. See 3.1 Configuration for further details.

```
/** Enable control for src/application/app_sound sample application */
#define R_SELF_INSERT_APP_SOUND (R_OPTION_ENABLE)
```

4.10.4 Playback Software Application

The playback software application plays a pre-recorded audio file through the headphone connection. The audio file is directly included in the source code, in file LR_44_1K16B_S.dat. This file is encoded in stereo 16-bit, 44.1kHz format.

The playback application starts in the function R_SOUND_PlaySample, which is run when the command **'play'** is entered into the console. This function initializes a control structure for the sound playback and calls a function play_file_data to manage the audio playback.

The function play_file_data creates a task for the playback, task_play_sound_demo, then waits for completion or a user keypress before finishing. The playback task, task_play_sound_demo, opens the SSIF and sound drivers. It then loops through the audio file, sending blocks of audio data via DMA to the SSIF peripheral to be sent to the CODEC. When playback is complete, the audio is closed and an event is set to signal for the calling function, play_file_data, to close the task and finish.

The streaming of the audio data to the SSIF is organized using a number of buffers (set by `#define NUM_AUDIO_BUFFER_BLOCKS_PRV_`, set to 3). DMA transfers from the audio file to the SSIF peripheral are managed by the SSIF driver using an array of AIOCB messaging blocks, which define the access control semaphore and callback function for each block of data to be transferred. The appropriate AIOCB messaging block is registered with the SSIF driver and a write initiates the transfer of data to the SSIF peripheral from the relevant point of the audio file. Once the transfer is complete, the access semaphore is released and the next block is set to transfer, until the whole file has been transferred. When complete, the SOUND driver and SSIF drivers are closed and the calling function notified via the event to close the task.

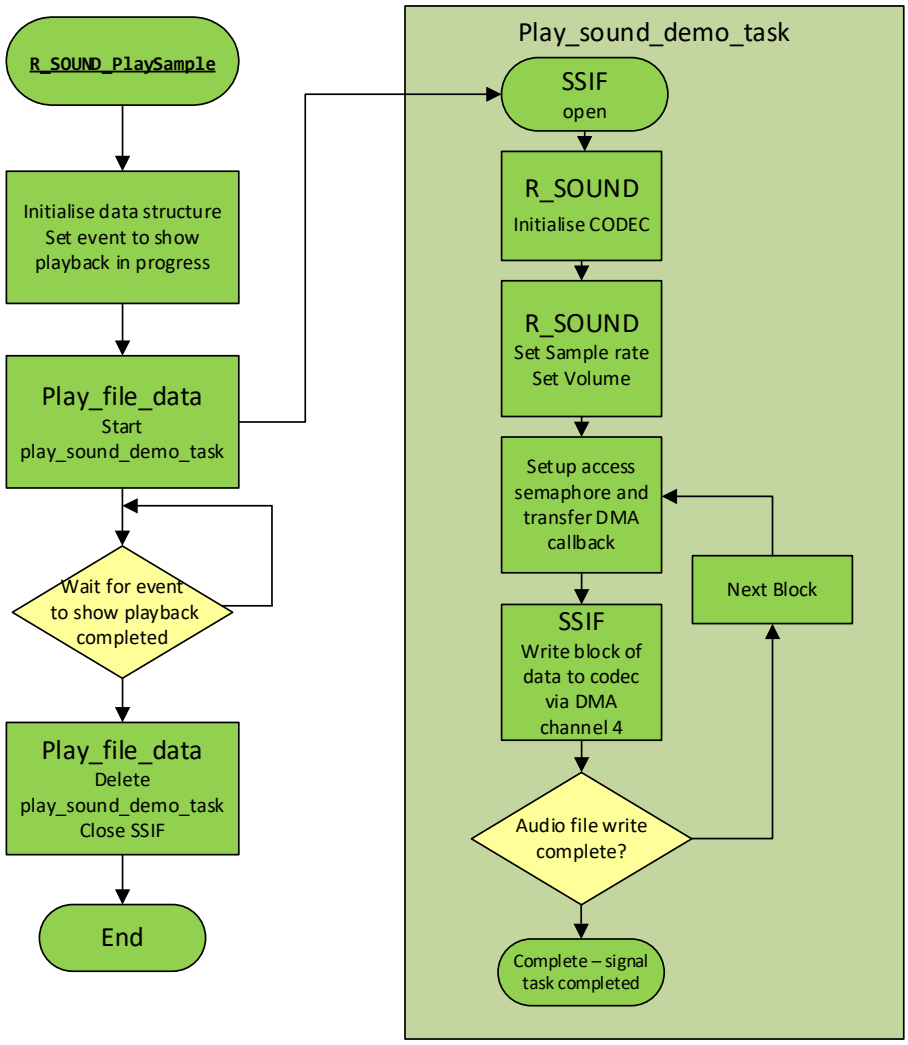

Figure 4-41 Play Sound Application Flowchart

4.10.5 Record Software Application

The record software application reads the input from a line-in connected to the audio jack and outputs it to the headphone connection in a software 'loopback' mode.

The record application starts in the function R_SOUND_RecordSample(), which is run when the command **'record'** is entered in to the console. This function initializes a control structure for the audio operation and calls the function play_recorded(). The function play_recorded() creates a buffer area, initializes access control semaphores, and configures the SOUND driver, before creating a task. Then task_record_sound_demo() waits for a user keypress before closing the task and finishing.

The task, task_record_sound_demo(), initializes the SSIF messaging structures for the transmit and receive operations, before entering a loop, receiving and transmitting audio information when transfers have completed, when indicated by flagging from end-of-transfer callback functions. This continues until a keypress is detected in the parent function, which then closes the task and completes.

The streaming of the audio data to/from the SSIF peripheral is organized using a number of buffers (set by `#define NUM_AUDIO_BUFFER_BLOCKS_PRV_`, set to 3). DMA transfers to and from the SSIF peripheral are managed by the SSIF driver using an array of AIOCB messaging blocks, for each receive and transmit channel. These define the access control semaphores and callback functions for each block of data to be transferred. When a SSIF read or write is ready to be setup, the SSIF driver is configured with the relevant AIOCB messaging block and the SSIF driver read or write command initiates the transfer of data. Once the transfer is complete, the access semaphore is released and the next block is set to transfer. As the read and write operations are working on the same data area, they are sharing the same semaphore accessing.

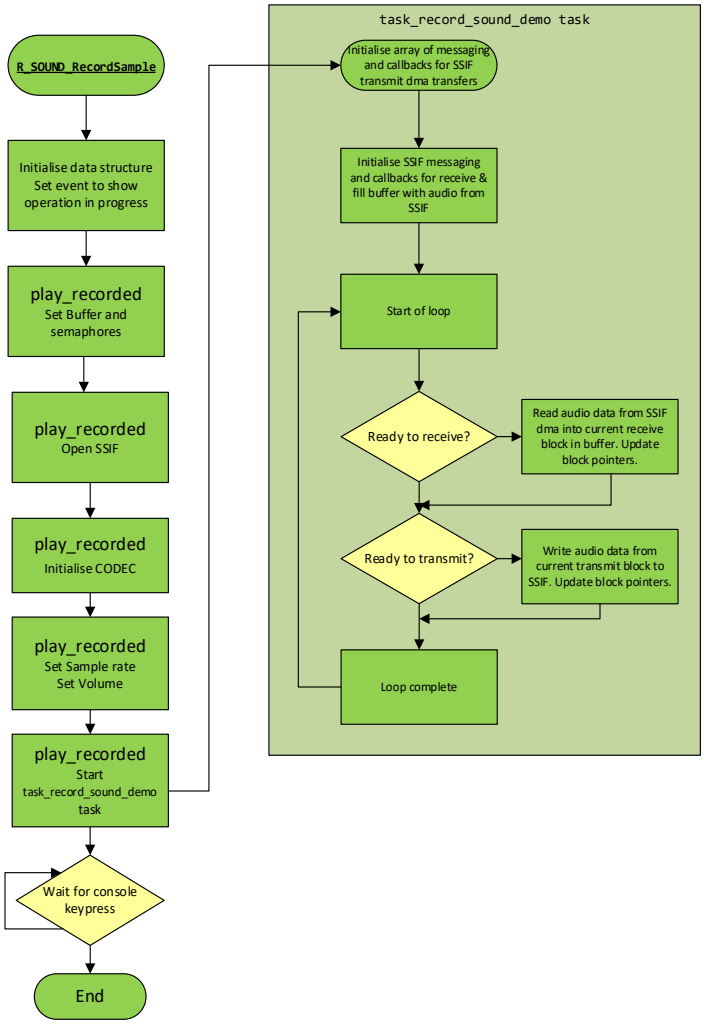

Figure 4-42 Record Sound Application Flowchart

4.11 Touchscreen Application

The touchscreen application allows for the touch capability of the Renesas Starter Kit + for RZ/A1H LCD to be used. This is achieved through using the RIIC peripheral (channel 3) on the RZ/A1H.

The software architecture is seen below.

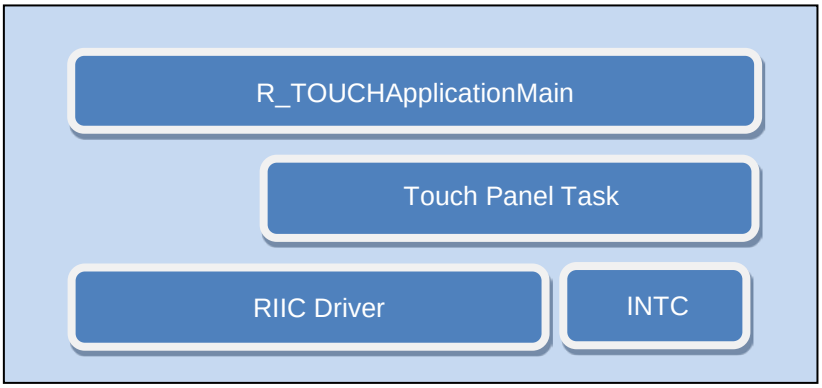

Figure 4-43 Sound Function Hierarchical Software Layer

4.11.1 Software Configuration

To enable, set `R_SELF_INSERT_APP_TOUCH_SCREEN` to `R_OPTION_ENABLE`. To disable, set it to `R_OPTION_DISABLE`. See 3.1 Configuration for further details.

```
/** Enable control for src/application/app_touchscreen sample application */
#define R_SELF_INSERT_APP_TOUCH_SCREEN (R_OPTION_ENABLE)
```

4.11.2 Commands

Table 4-12 shows the USB Functions commands.

Command	Description
tsdemo	Enables the touch screen application

Table 4-12 Touch Screen Command

4.11.3 Touchscreen Software

For more information regarding the touchscreen application please refer to the Application Note RZ/A1H Touch Panel Utility (R01AN5096EJ).

4.12 Light and Versatile Graphics Library

A port of the LVGL library has been provided for this device. The Benchmark demo (LV_USE_DEMO_BENCHMARK).

4.12.1 Commands

Table 4-13 shows the lvgl command.

Command	Description
gui	Invokes the LVGL demonstration application

Table 4-13 LVGL Command

4.12.2 GUI Software

For more information regarding the LVGL Development please refer to Introduction — LVGL documentation.

Revision History

Rev.	Date	Description	
		Page	Summary
1.00	Nov. 29, 2019	-	First Edition issued
1.01	Feb. 27, 2026	-	Removed Guiliani from document Update handling of INTC.IRQRR
1.02	Aug. 31, 2026	64	Added section 4.12 Light and Versatile Graphics Library

General Precautions in the Handling of Microprocessing Unit and Microcontroller Unit Products

The following usage notes are applicable to all Microprocessing unit and Microcontroller unit products from Renesas. For detailed usage notes on the products covered by this document, refer to the relevant sections of the document as well as any technical updates that have been issued for the products.

1. Precaution against Electrostatic Discharge (ESD)

   A strong electrical field, when exposed to a CMOS device, can cause destruction of the gate oxide and ultimately degrade the device operation. Steps must be taken to stop the generation of static electricity as much as possible, and quickly dissipate it when it occurs. Environmental control must be adequate. When it is dry, a humidifier should be used. This is recommended to avoid using insulators that can easily build up static electricity. Semiconductor devices must be stored and transported in an anti-static container, static shielding bag or conductive material. All test and measurement tools including work benches and floors must be grounded. The operator must also be grounded using a wrist strap. Semiconductor devices must not be touched with bare hands. Similar precautions must be taken for printed circuit boards with mounted semiconductor devices.
2. Processing at power-on

   The state of the product is undefined at the time when power is supplied. The states of internal circuits in the LSI are indeterminate and the states of register settings and pins are undefined at the time when power is supplied. In a finished product where the reset signal is applied to the external reset pin, the states of pins are not guaranteed from the time when power is supplied until the reset process is completed. In a similar way, the states of pins in a product that is reset by an on-chip power-on reset function are not guaranteed from the time when power is supplied until the power reaches the level at which resetting is specified.
3. Input of signal during power-off state

   Do not input signals or an I/O pull-up power supply while the device is powered off. The current injection that results from input of such a signal or I/O pull-up power supply may cause malfunction and the abnormal current that passes in the device at this time may cause degradation of internal elements. Follow the guideline for input signal during power-off state as described in your product documentation.
4. Handling of unused pins

   Handle unused pins in accordance with the directions given under handling of unused pins in the manual. The input pins of CMOS products are generally in the high-impedance state. In operation with an unused pin in the open-circuit state, extra electromagnetic noise is induced in the vicinity of the LSI, an associated shoot-through current flows internally, and malfunctions occur due to the false recognition of the pin state as an input signal become possible.
5. Clock signals

   After applying a reset, only release the reset line after the operating clock signal becomes stable. When switching the clock signal during program execution, wait until the target clock signal is stabilized. When the clock signal is generated with an external resonator or from an external oscillator during a reset, ensure that the reset line is only released after full stabilization of the clock signal. Additionally, when switching to a clock signal produced with an external resonator or by an external oscillator while program execution is in progress, wait until the target clock signal is stable.
6. Voltage application waveform at input pin

   Waveform distortion due to input noise or a reflected wave may cause malfunction. If the input of the CMOS device stays in the area between V_{IL} (Max.) and V_{IH} (Min.) due to noise, for example, the device may malfunction. Take care to prevent chattering noise from entering the device when the input level is fixed, and also in the transition period when the input level passes through the area between V_{IL} (Max.) and V_{IH} (Min.).
7. Prohibition of access to reserved addresses

   Access to reserved addresses is prohibited. The reserved addresses are provided for possible future expansion of functions. Do not access these addresses as the correct operation of the LSI is not guaranteed.
8. Differences between products

   Before changing from one product to another, for example to a product with a different part number, confirm that the change will not lead to problems. The characteristics of a microprocessing unit or microcontroller unit products in the same group but having a different part number might differ in terms of internal memory capacity, layout pattern, and other factors, which can affect the ranges of electrical characteristics, such as characteristic values, operating margins, immunity to noise, and amount of radiated noise. When changing to a product with a different part number, implement a system-evaluation test for the given product.

Notice

1. Descriptions of circuits, software and other related information in this document are provided only to illustrate the operation of semiconductor products and application examples. You are fully responsible for the incorporation or any other use of the circuits, software, and information in the design of your product or system. Renesas Electronics disclaims any and all liability for any losses and damages incurred by you or third parties arising from the use of these circuits, software, or information.
2. Renesas Electronics hereby expressly disclaims any warranties against and liability for infringement or any other claims involving patents, copyrights, or other intellectual property rights of third parties, by or arising from the use of Renesas Electronics products or technical information described in this document, including but not limited to, the product data, drawings, charts, programs, algorithms, and application examples.
3. No license, express, implied or otherwise, is granted hereby under any patents, copyrights or other intellectual property rights of Renesas Electronics or others.
4. You shall be responsible for determining what licenses are required from any third parties, and obtaining such licenses for the lawful import, export, manufacture, sales, utilization, distribution or other disposal of any products incorporating Renesas Electronics products, if required.
5. You shall not alter, modify, copy, or reverse engineer any Renesas Electronics product, whether in whole or in part. Renesas Electronics disclaims any and all liability for any losses or damages incurred by you or third parties arising from such alteration, modification, copying or reverse engineering.
6. Renesas Electronics products are classified according to the following two quality grades: “Standard” and “High Quality”. The intended applications for each Renesas Electronics product depends on the product’s quality grade, as indicated below.
   "Standard": Computers; office equipment; communications equipment; test and measurement equipment; audio and visual equipment; home electronic appliances; machine tools; personal electronic equipment; industrial robots; etc.
   "High Quality": Transportation equipment (automobiles, trains, ships, etc.); traffic control (traffic lights); large-scale communication equipment; key financial terminal systems; safety control equipment; etc.

   Unless expressly designated as a high reliability product or a product for harsh environments in a Renesas Electronics data sheet or other Renesas Electronics document, Renesas Electronics products are not intended or authorized for use in products or systems that may pose a direct threat to human life or bodily injury (artificial life support devices or systems; surgical implantations; etc.), or may cause serious property damage (space system; undersea repeaters; nuclear power control systems; aircraft control systems; key plant systems; military equipment; etc.). Renesas Electronics disclaims any and all liability for any damages or losses incurred by you or any third parties arising from the use of any Renesas Electronics product that is inconsistent with any Renesas Electronics data sheet, user’s manual or other Renesas Electronics document.
7. No semiconductor product is absolutely secure. Notwithstanding any security measures or features that may be implemented in Renesas Electronics hardware or software products, Renesas Electronics shall have absolutely no liability arising out of any vulnerability or security breach, including but not limited to any unauthorized access to or use of a Renesas Electronics product or a system that uses a Renesas Electronics product. RENESAS ELECTRONICS DOES NOT WARRANT OR GUARANTEE THAT RENESAS ELECTRONICS PRODUCTS, OR ANY SYSTEMS CREATED USING RENESAS ELECTRONICS PRODUCTS WILL BE INVULNERABLE OR FREE FROM CORRUPTION, ATTACK, VIRUSES, INTERFERENCE, HACKING, DATA LOSS OR THEFT, OR OTHER SECURITY INTRUSION (“Vulnerability Issues”). RENESAS ELECTRONICS DISCLAIMS ANY AND ALL RESPONSIBILITY OR LIABILITY ARISING FROM OR RELATED TO ANY VULNERABILITY ISSUES. FURTHERMORE, TO THE EXTENT PERMITTED BY APPLICABLE LAW, RENESAS ELECTRONICS DISCLAIMS ANY AND ALL WARRANTIES, EXPRESS OR IMPLIED, WITH RESPECT TO THIS DOCUMENT AND ANY RELATED OR ACCOMPANYING SOFTWARE OR HARDWARE, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE.
8. When using Renesas Electronics products, refer to the latest product information (data sheets, user’s manuals, application notes, “General Notes for Handling and Using Semiconductor Devices” in the reliability handbook, etc.), and ensure that usage conditions are within the ranges specified by Renesas Electronics with respect to maximum ratings, operating power supply voltage range, heat dissipation characteristics, installation, etc. Renesas Electronics disclaims any and all liability for any malfunctions, failure or accident arising out of the use of Renesas Electronics products outside of such specified ranges.
9. Although Renesas Electronics endeavors to improve the quality and reliability of Renesas Electronics products, semiconductor products have specific characteristics, such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Unless designated as a high reliability product or a product for harsh environments in a Renesas Electronics data sheet or other Renesas Electronics document, Renesas Electronics products are not subject to radiation resistance design. You are responsible for implementing safety measures to guard against the possibility of bodily injury, injury or damage caused by fire, and/or danger to the public in the event of a failure or malfunction of Renesas Electronics products, such as safety design for hardware and software, including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other appropriate measures. Because the evaluation of microcomputer software alone is very difficult and impractical, you are responsible for evaluating the safety of the final products or systems manufactured by you.
10. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. You are responsible for carefully and sufficiently investigating applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive, and using Renesas Electronics products in compliance with all these applicable laws and regulations. Renesas Electronics disclaims any and all liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.
11. Renesas Electronics products and technologies shall not be used for or incorporated into any products or systems whose manufacture, use, or sale is prohibited under any applicable domestic or foreign laws or regulations. You shall comply with any applicable export control laws and regulations promulgated and administered by the governments of any countries asserting jurisdiction over the parties or transactions.
12. It is the responsibility of the buyer or distributor of Renesas Electronics products, or any other party who distributes, disposes of, or otherwise sells or transfers the product to a third party, to notify such third party in advance of the contents and conditions set forth in this document.
13. This document shall not be reprinted, reproduced or duplicated in any form, in whole or in part, without prior written consent of Renesas Electronics.
14. Please contact a Renesas Electronics sales office if you have any questions regarding the information contained in this document or Renesas Electronics products.

(Note1) “Renesas Electronics” as used in this document means Renesas Electronics Corporation and also includes its directly or indirectly controlled subsidiaries.
(Note2) “Renesas Electronics product(s)” means any product developed or manufactured by or for Renesas Electronics.

(Rev.5.0-1 October 2020)

Corporate Headquarters

TOYOSU FORESIA, 3-2-24 Toyosu,
Koto-ku, Tokyo 135-0061, Japan
www.renesas.com

Trademarks

Renesas and the Renesas logo are trademarks of Renesas Electronics Corporation. All trademarks and registered trademarks are the property of their respective owners.

Contact information

For further information on a product, technology, the most up-to-date version of a document, or your nearest sales office, please visit:
www.renesas.com/contact/.