

RZ/A1LU group

R01AN4427EG0100

Rev.1.0

Software Package

Jul 11, 2018

Introduction

This Application Note describes the operation of the software package created for the RZA1LU Stream it! V2.3. 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 RZA1LU and is equipped with:

- RZA1LU Stream it! V2.3 Platform
- Stream it! TFT LCD Screen
- J-Link Debugger
- USB A to Mini-B Cable
- OV7670 Camera
- Okaya LCD PMOD™
- Ethernet Cable and connection to LAN
- USB Keyboard, Mouse and Memory Drive

Optional:

- USB Hub (to allow for concurrent USB connections)

Target Device

RZA1LU

Referenced Documents

Document Type	Document Name	Document No.
Quick Start Guide	RZ/A1 Software Package Quick Start Guide	R01QS0024EJ
User's Manual	RZA1LU Hardware Manual	R01UH0437EJ
Application Note	RZ/A1LU QSPI Flash Boot Loader	R11AN0084EG
Application Note	RZ/A1LU Framework Release Note	R01AN3638EJ
Application Note	RZ/A1LU SDK for Camera Sample Program	R01AN4312EJ
Application Note	RZ/A1LU Video Utility	R01AN4313EJ
Application Note	RZ/A1LU Touch Panel Utility	R01AN4314EJ
Application Note	RZ/A1LU GUI Sample Program	R01AN4413EJ
Hardware Design	Design information V2.3	https://www.renesas.com/en-eu/solutions/key-technology/human-interface/rz-stream-it.html

List of Abbreviations and Acronyms

Abbreviation	Full Form
ANSI	American National Standards Institute
ADC	Analogue to Digital Converter
CDC	Communication Device Class
CMOS	Complementary Metal Oxide Semiconductor
CODEC	Coder-Decoder. A device for encoding/decoding a digital data stream
DHCP	Dynamic Host Configuration Protocol
EEPROM	Electrically Erasable Programmable Read-Only Memory
GPIO	General Purpose Input Output
HID	Human Interface Device
HLD	High Layer / Level Driver
HMI	Human Machine Interface
LAN	Local Area Network
LED	Light Emitting Diode
LLD	Low Layer / Level Driver
MCU	Microcontroller Unit
MSC	Mass Storage Class
OS	Operating system
PMOD	Pmod interface or Peripheral Module interface is an open standard defined by Digilent Inc.
QSPI	Quad Serial Peripheral Interface
QSG	Quick Start Guide
RTOS	Real Time Operating System
STDIO	Standard Input Output
TFT	Thin Film Transistor
USB	Universal Serial Bus

Table 1-1 List of Abbreviations and Acronyms

Contents

1. Outline of the Software Package	6
1.1 Folder Structure	6
2. Description of the System.....	7
2.1 Hardware.....	8
2.1.1 Programming and Serial Console	8
2.1.2 Connectivity	8
2.1.3 HMI.....	8
2.1.4 Memory	8
2.1.5 USB.....	8
2.1.6 Analog.....	8
2.1.7 Audio.....	8
3. System Software.....	9
3.1 Configuration.....	9
3.2 Loading the Software Package.....	10
3.3 Operating System	11
3.4 STDIO	12
3.4.1 Stream configuration example	14
3.4.2 STDIO Files.....	14
3.4.3 STDIO Include files	14
3.4.4 STDIO lowsrc.c file	15
3.4.5 STDIO devlink.c file	15
3.4.6 Device Driver Function Table	16
3.4.7 Dynamic Device List	16
3.4.8 Dynamic Device Link Table	16
3.4.9 Static Device Mount Table	17
3.5 System Commands.....	18
3.6 Doxygen	19
4. Applications.....	20
4.1 ADC.....	20
4.1.1 Software Configuration	20
4.1.2 Commands	20
4.1.3 Application	20
4.1.4 ADC Driver.....	20
4.2 Switch.....	21
4.2.1 Switch Driver	21
4.3 LED	21
4.3.1 Commands	21

4.3.2	LED Driver	21
4.4	PMOD.....	22
4.4.1	Software Configuration	22
4.4.2	Commands	22
4.4.3	Application	22
4.4.4	PMOD Middleware.....	23
4.4.5	RSPI.....	23
4.5	USB Mass Storage	24
4.5.1	Software Configuration	24
4.5.2	Commands	25
4.5.3	FATFS.....	25
4.5.4	USB Host Stack.....	26
4.5.5	Hardware Layer	29
4.6	Website	39
4.6.1	Software Configuration	39
4.6.2	Application	39
4.6.3	Ethernet Stack.....	42
4.6.4	WebIO Server and Files.....	49
4.6.5	WebIF	49
4.6.6	Website Files	49
4.7	Camera	50
4.7.1	Software Configuration	50
4.7.2	Commands	51
4.7.3	Menu.....	52
4.8	USB HID	55
4.8.1	Software Configuration	55
4.8.2	Commands	55
4.8.3	USB HID Mouse.....	55
4.8.4	USB HID Keyboard.....	56
4.9	USB CDC.....	57
4.9.1	Software Configuration	57
4.9.2	Commands	57
4.9.3	CDC Driver.....	57
4.10	Sound Application	58
4.10.1	Software Configuration	58
4.10.2	Console Commands	58
4.10.3	Audio Software Configuration	58
4.10.4	Playback Software Application	59
4.10.5	Record Software Application.....	60
4.11	Touchscreen Application	61

4.11.1 Software Configuration	61
4.11.2 Touchscreen Software	61
4.12 TES GUILIANI	62
Website and Support	63

1. Outline of the Software Package

The RZ/A1LU 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 RZA1LU Stream it! V2.3 development kit.

The main features of the software package include:

- Using a DHCP Server to create a Dynamic Webpage.
- Displays Images from a CMOS camera onto a TFT Touchscreen.
- USB functionality – Mass Storage, CDC, and HID (Keyboard and Mouse).
- Sound Interface provides a record and play functionality.
- TES Guiliani Graphics Display.
- 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.

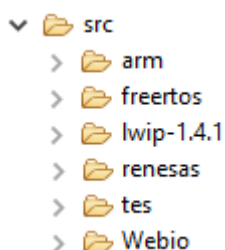


Figure 1-1 'src' Folder

The first level includes third party source arm, FreeRTOS, lwip1.4.1, TES and Webio. All of which are utilized from the 'Renesas' Folder.

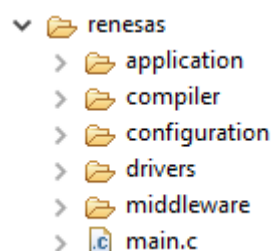


Figure 1-2 'renesas' folder

The Renesas folder is made up of application, compiler, configuration, driver and middleware specific modules. These modules make up the system. Application holds all of the high layer application level, the compiler 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 holds hardware specific code that control the microcontroller peripherals, whilst the middleware holds code that is hardware independent.

2. Description of the System

The following image provides a holistic view of the RZA1LU Stream-it! V2.3.

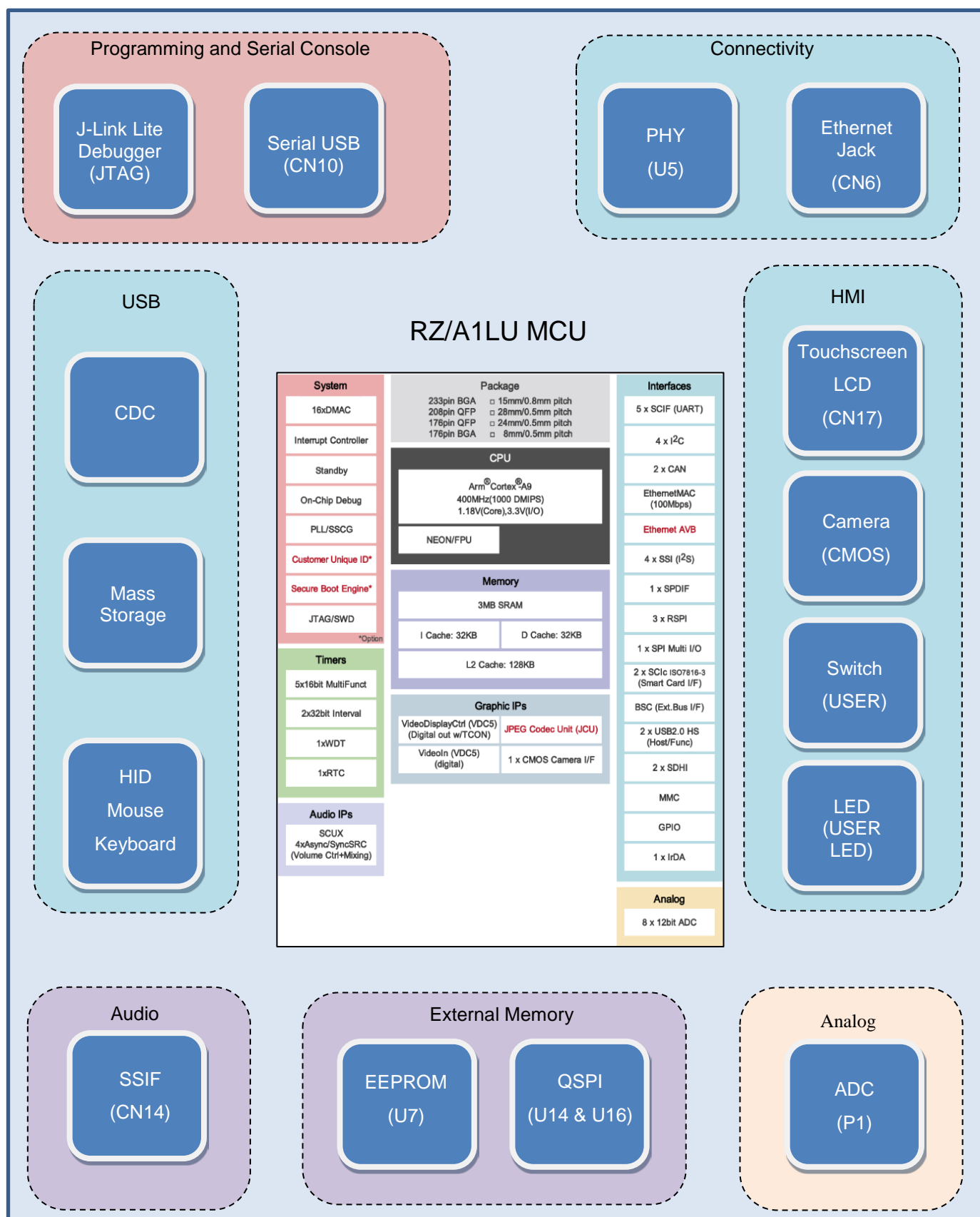


Figure 2-1 Overall System

2.1 Hardware

2.1.1 Programming and Serial Console

The board is powered through the USB function connector, CN10. 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 RZA1LU Stream it! Development Kit has built-in ethernet connectivity. To use this the user will required to connect a RJ45 Cable from CN6 to a Local Area Network.

2.1.3 HMI

The platform includes a TFT LCD Touchscreen which connects to CN7, and a camera TD7470 which connects to CN12. The development board also includes a user-switch and a LED.

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, P1 on the Stream it! board allows a user controllable analog voltage to be presented to the microcontroller's ADC input.

2.1.7 Audio

The RZA1LU Stream it! board uses a CODEC IC, MAX9856 (U13) to manage the acquisition and playback of audio information via the 3.5mm audio jack CN14. 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 IIC connection (on IIC channel 1) to allow the MCU to configure the CODEC appropriately.

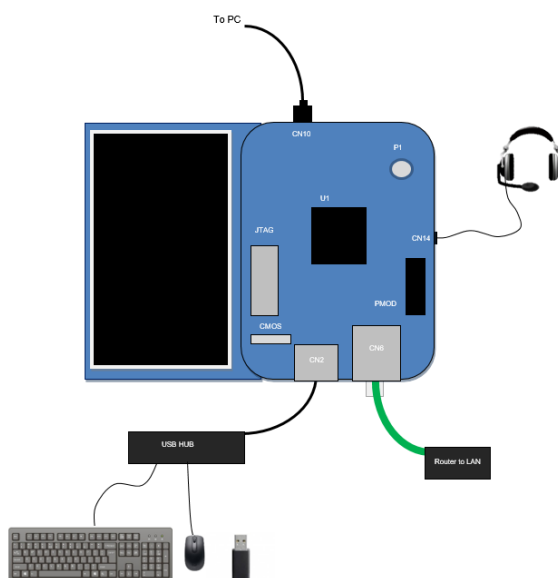


Figure 2-2 System Use

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 configuration_app.h file. In this file the user may 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 configuration_app.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 for GUI sample application */
#define R_SELF_INSERT_APP_GUI (R_OPTION_ENABLE)

/* 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 are not recommended to be enabled concurrently.

Feature one	Feature two	Reason
R_SELF_BLINK_TASK_CREATION	LED commands	Both features utilize the RZA1LU Stream it!'s 'USER' LED.
R_SELF_BLINK_TASK_CREATION	R_SELF_LOAD_MIDDLEWARE_ETHERNET_MODULES	Both features utilize the RZA1LU Stream it!'s 'USER' LED. The Welcome Page on the Website will not be able to control the LED.
R_SELF_LOAD_MIDDLEWARE_ETHERNET_MODULES	R_SELF_INSERT_APP_GUI	The Ethernet and LCD shares the following pins: P8.14, P8.13, P8.10 down to P8.0.
R_SELF_LOAD_MIDDLEWARE_ETHERNET_MODULES	R_SELF_INSERT_APP_TOUCH_SCREEN	The Ethernet and LCD shares the following pins: P8.14, P8.13, P8.10 down to P8.0.
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/A1LU 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 load's the RZA1LU Software Package to memory at 0x18080000.

For more information regarding the bootloader please refer to the Application Note 'RZ/A1LU QSPI Flash Boot Loader' (R11AN0084EG).

To ensure the QSPI device has the correct QSPI loader simply run the 'Program_RZ_A1LU_boot.bat' found in the util/dos_scripts/Program_RZ_A1LU_boot.bat.

Note that this script must be run from windows explorer.

3.3 Operating System

The RZA1LU Software Package utilizes FreeRTOS V10.0.0 to provide an OS environment. FreeRTOS is a third-party software package which allows for an OS functionality.

The following image provides the software architecture for the OS.

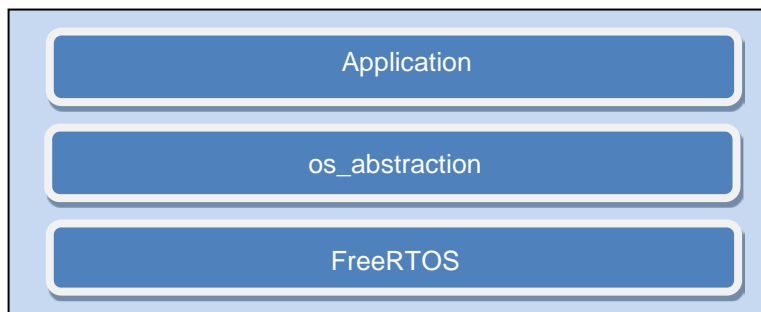


Figure 3-1 OS Hierarchical Software Layers

To allow for easy portability between different OS' 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 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 utilises 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 I2C 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.

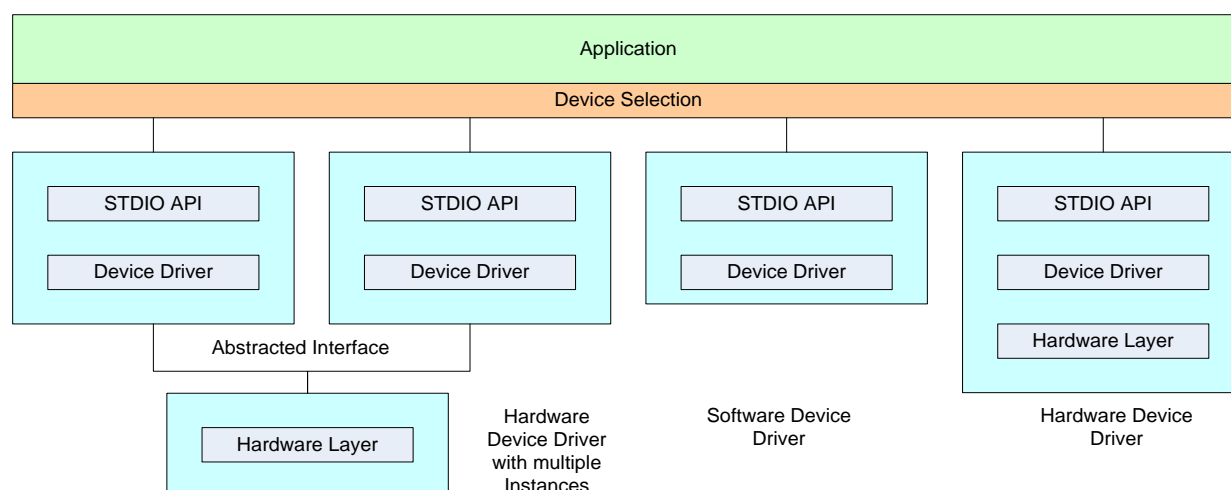


Figure 3-2 Architecture Overview

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 IO library is based on streams. Each of the IO 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 four 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 pre-pended 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 CAN busses or 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:

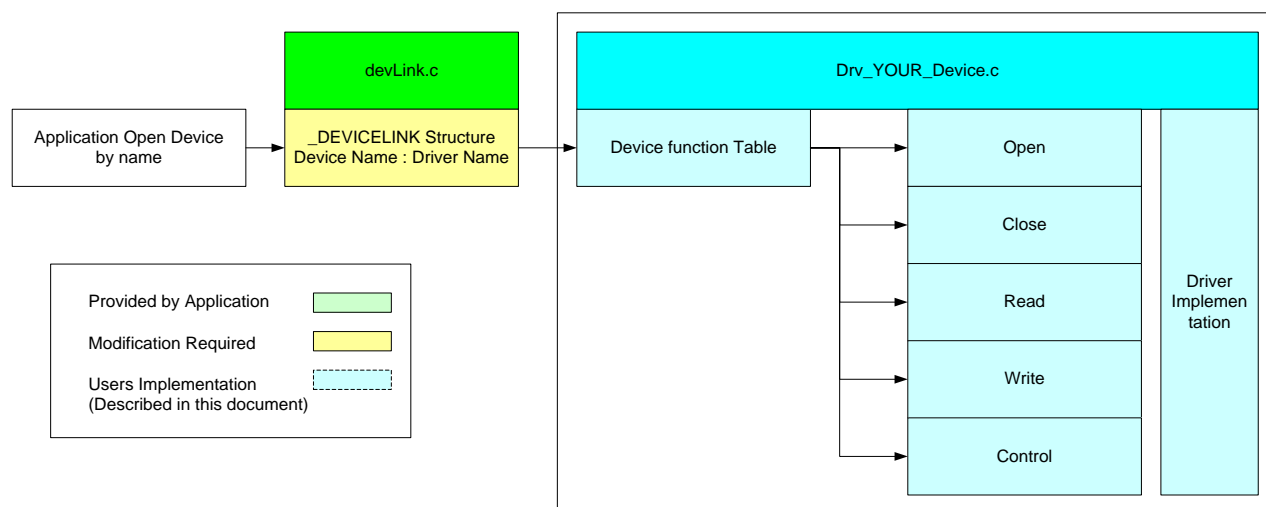


Figure 3-6 Device Linkage

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. IO) 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 3-6 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 dependant 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 specified 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 the via a Communication Port to the PC. The function USB (CN10) allows for a USB to Mini-B wire to be connected to a PC. To access the serial line a serial terminal is required 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 Stream It! Development Kit only has one user LED, so the letter 'a' should be replaced with a '0'. 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 (HEX) 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 RZA1LU 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 RZA1LU, 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

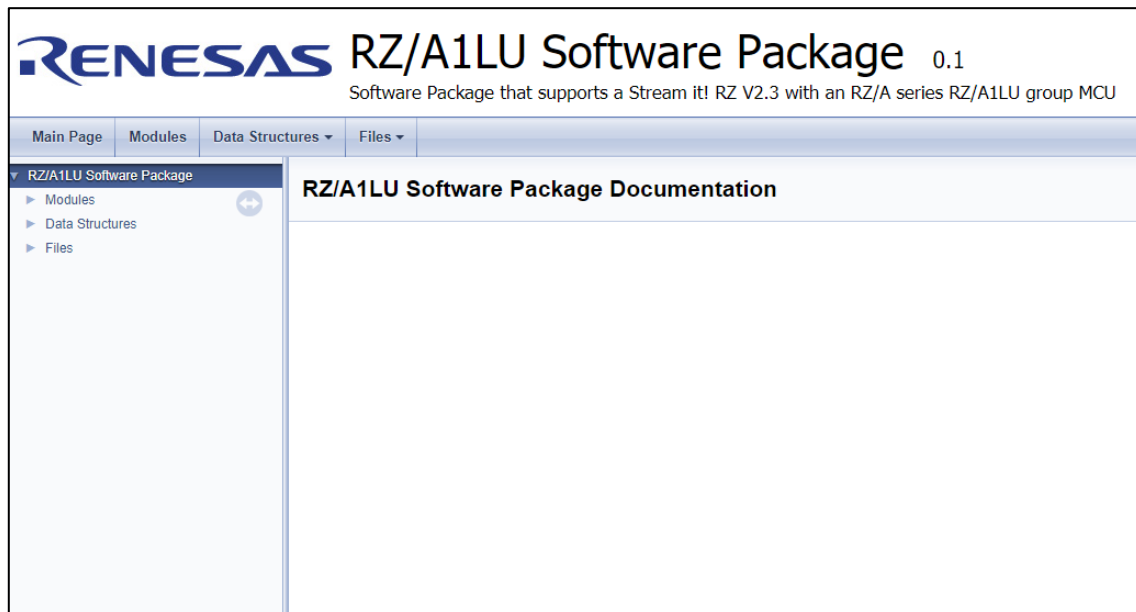


Figure 3-10 Index.html

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 2 and read the pin's associated voltage. The pin that can be altered by varying the potentiometer P1.

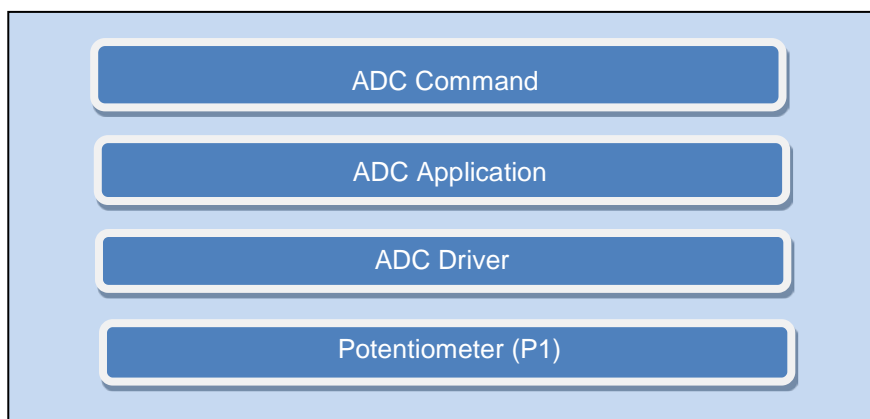


Figure 4-1 ADC Hierarchical Software Layer

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 RZA1LU Stream it! 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.

```

REE> adcdemo
ADC sample program start
Press any key to terminate demo
Rotate Potentiometer to see effect on adc input (AN2)
Potentiometer(AN2) = 3770
  
```

Figure 4-2 ADC Application

4.1.4 ADC Driver

The ADC Driver API can be seen in Doxygen under RZ/A1LU Software Package -> Modules -> Drivers (Not POSIX) -> ADC RZA1LU Driver

4.2 Switch

The switch 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/A1LU Software Package -> Modules -> Drivers (Not POSIX) -> SWITCH RZA1LU Driver

4.3 LED

The 'USER' LED is connected to the Port 7 Pin 8 and is active high. To control the LED the following commands are available.

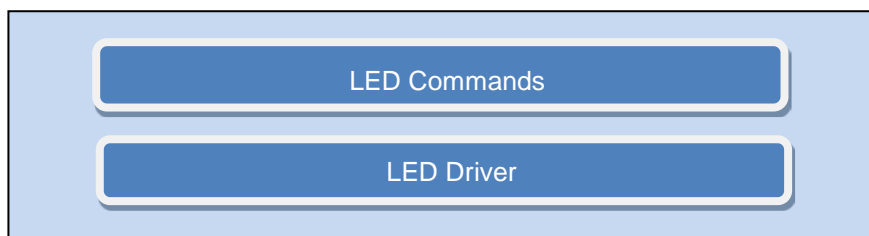


Figure 4-4 LED Hierarchical Software Layer

4.3.1 Commands

Table 4-2 shows the commands available for the LED.

Command	Description
led a s <CR>	Turn LED on (s=1), off (s=0) or toggle (s=^). Note that the Stream It! Development Kit only has one user LED, so the letter 'a' should be replaced with a '0'. For example, to toggle the 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 numerous LEDs although the RZA1LU Stream it! only has one.

The LED Driver API can be seen in Doxygen under RZ/A1LU Software Package -> Modules -> Drivers(POSIX) -> RZA1LU Stream-IT 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 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/A1LU Software Package -> Modules -> Middleware (POSIX) -> PMOD API.

4.4.5 RSPI

The RSPI Driver API can be seen in Doxygen under RZ/A1LU Software Package -> Modules -> Drivers (Not POSIX) -> RSPI RZA1LU 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 RZA1LU to the USB Drive. This can either be done through the command line or through the website as described in section 4.6.

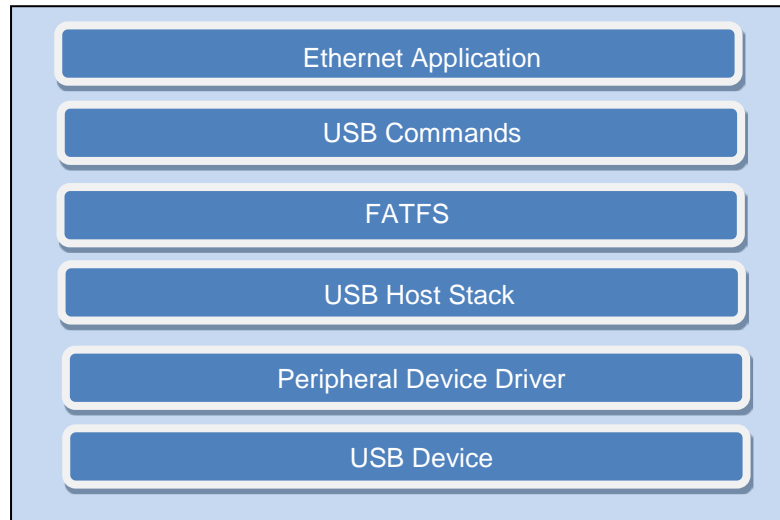


Figure 4-7 USB Hierarchical Software Layer

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 drives are supported in the implementation. Each drive is allocated a letter starting from 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 file 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`.

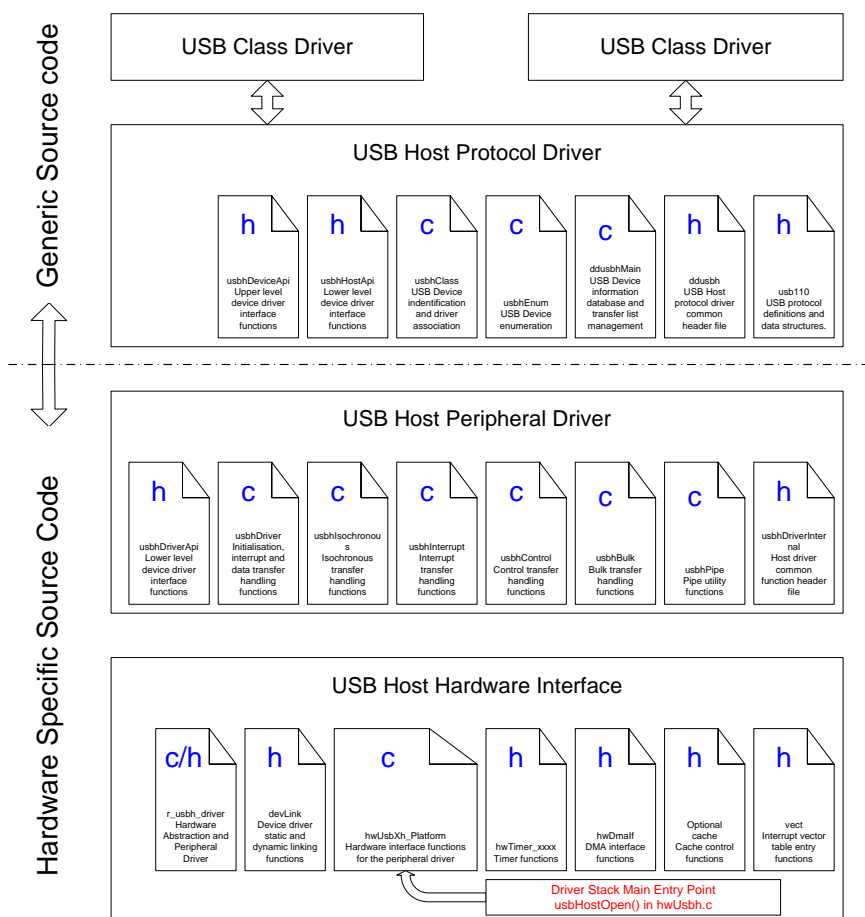


Figure 4-8 USB Host Stack

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.

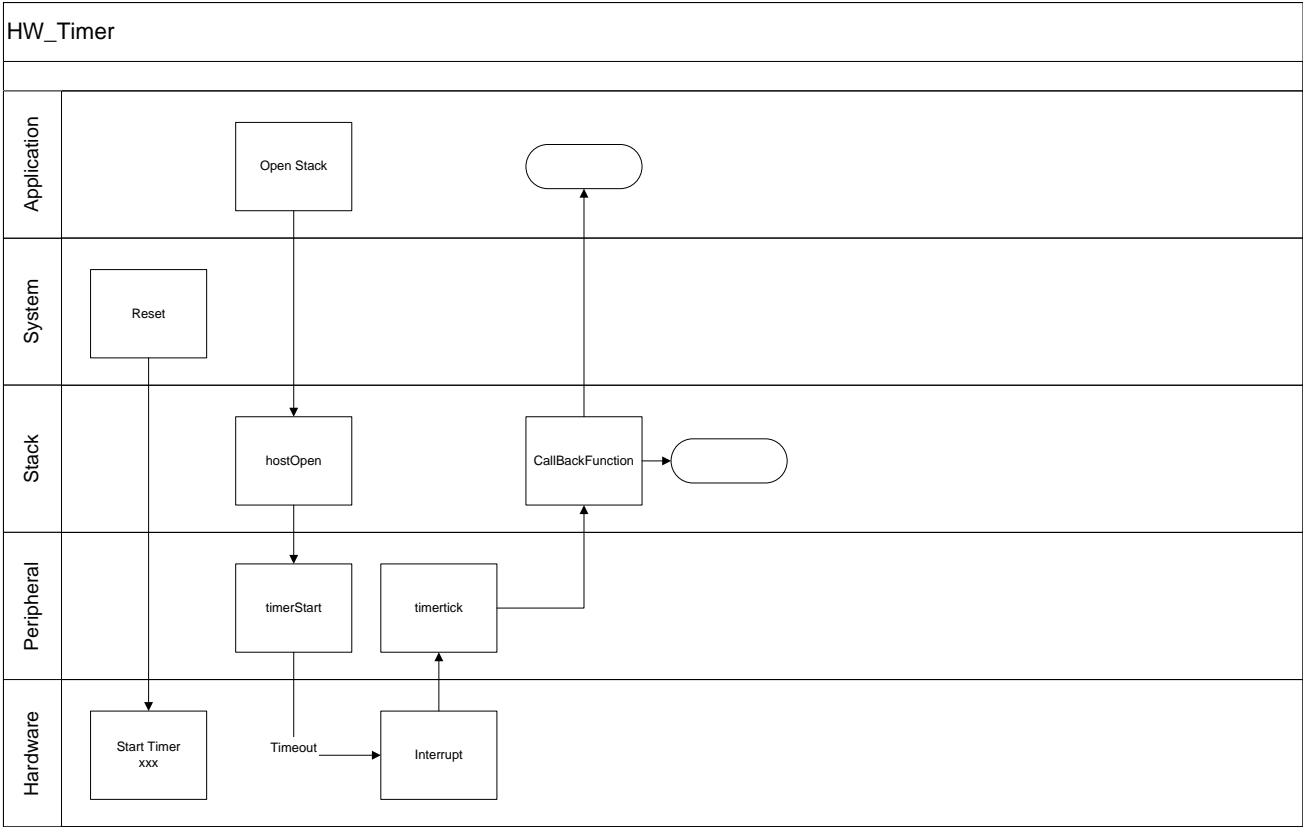


Figure 4-13 Base Hardware Timer Operation

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 DMA module can be used to service many peripherals within the system, the firmware may want to use one channel for more than one application or even disable the Stack from using the DMA functionality. In each case the DMA must be used mutually exclusively which would require run time assignment. Operating systems provide MUTEX events for this purpose. This USB stack provides a bit map (`giChannelMap`) to ensure that the USB Stack only uses channels that are available, see the file `hwDmaIf.c` for implementation. Alternatively, in cases where the DMA is completely disabled the performance of the USB host stack will be severely limited. The DMA functionality can be disabled by setting the `#if` statement in the “`dmaAlloc`” function, found in “`hwDmaIf.c`” to 0. This will prevent a DMA channel being allocated and the stack will default to performing FIFO based transfers.

```
int32_t dmaAlloc (uint32_t ichannel)
{
    uint32_t i_channel_loc;
    int_t imask = R_OS_SysLock(NULL);

    if (ichannel < MAX_DMA_CHANNELS)
    {
        i_channel_loc = (uint32_t)(1 << ichannel);

        /* Check for channel already in use */
        if (gi_channel_map & i_channel_loc)
        {
            R_OS_SysUnlock(NULL, imask);
            TRACE(("dmaAlloc: DMA Channel %d already in use\r\n",
ichannel));
            return -1;
        }

        /* Assign the channel */
        gi_channel_map |= i_channel_loc;

        R_OS_SysUnlock(NULL, imask);

        TRACE(("dmaAlloc: DMA Channel %d allocated\r\n", ichannel));
        return 0;
    }

    return -1;
}
```

Figure 4-14 DMA Allocation Code

If the users application is going to use the DMA channels then before configuring a channel the user should call `dmaAlloc()` to reserve a DMA channel. The return value will confirm whether the DMA channel is free to be used. When finished with the DMA Channel, preferable after every transfer, the user should call `dmaFree()` to release the DMA channel and make it available to other functions. This configuration is left to the user to match the needs of their system.

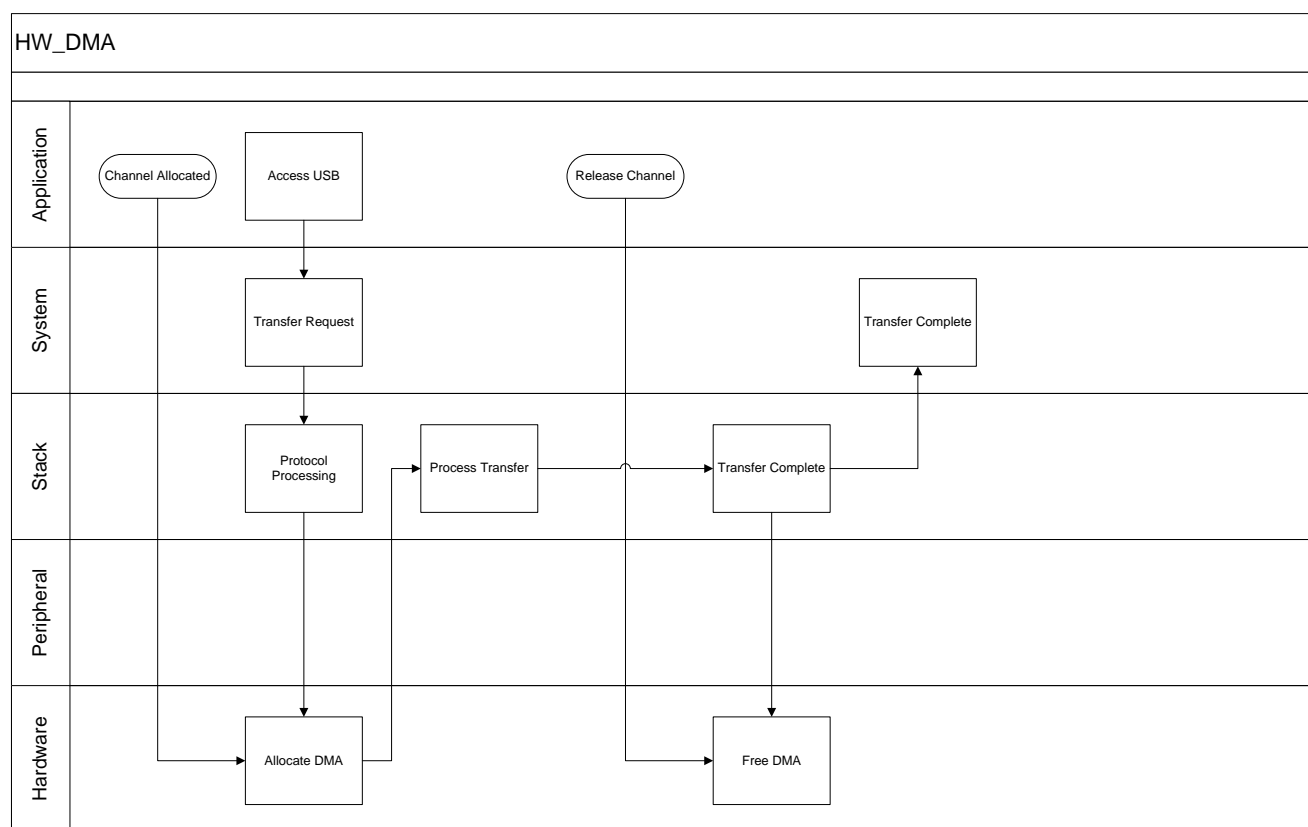


Figure 4-15 DMA Allocation operation

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, close IO 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.

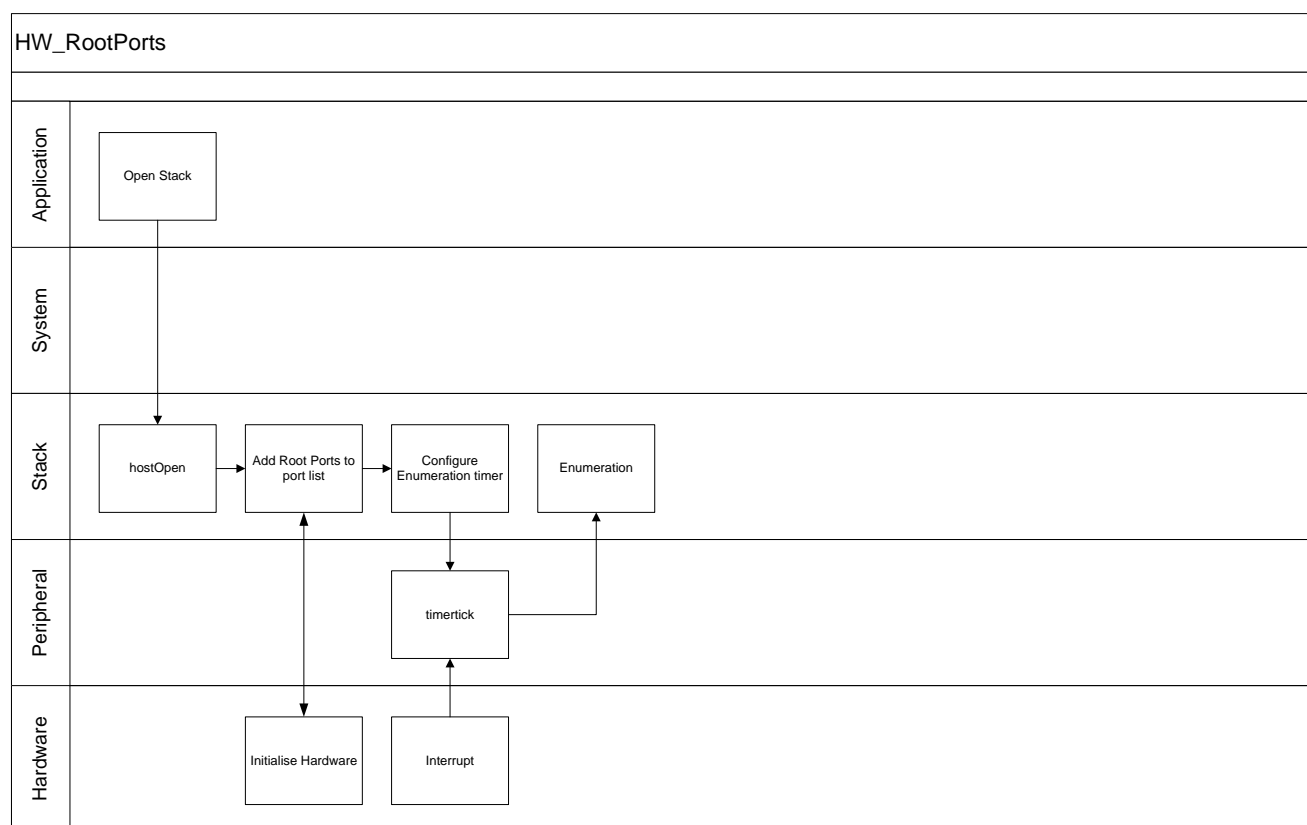


Figure 4-16 Host Port

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 4-17 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 4-18 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 4-19 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 form 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.COIRQRR.BIT.IRQaF = 0;
}
/*****
End of function INT_IRQa
*****/

```

Figure 4-20 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 End-Point. 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.

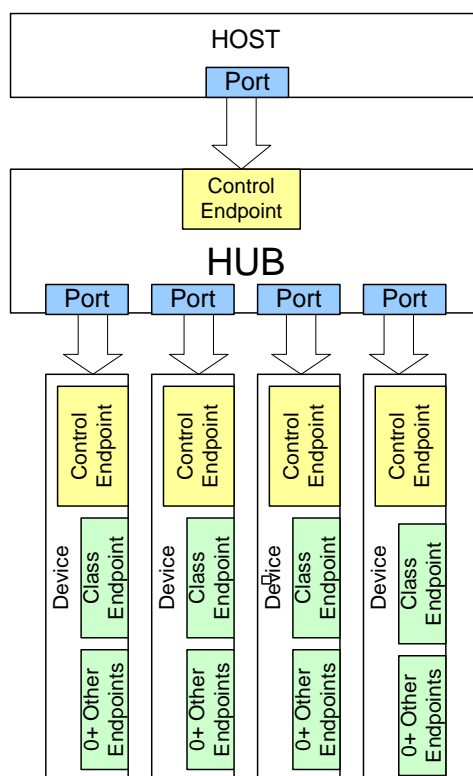


Figure 4-21 HUB Topology

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 re-tried. 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.

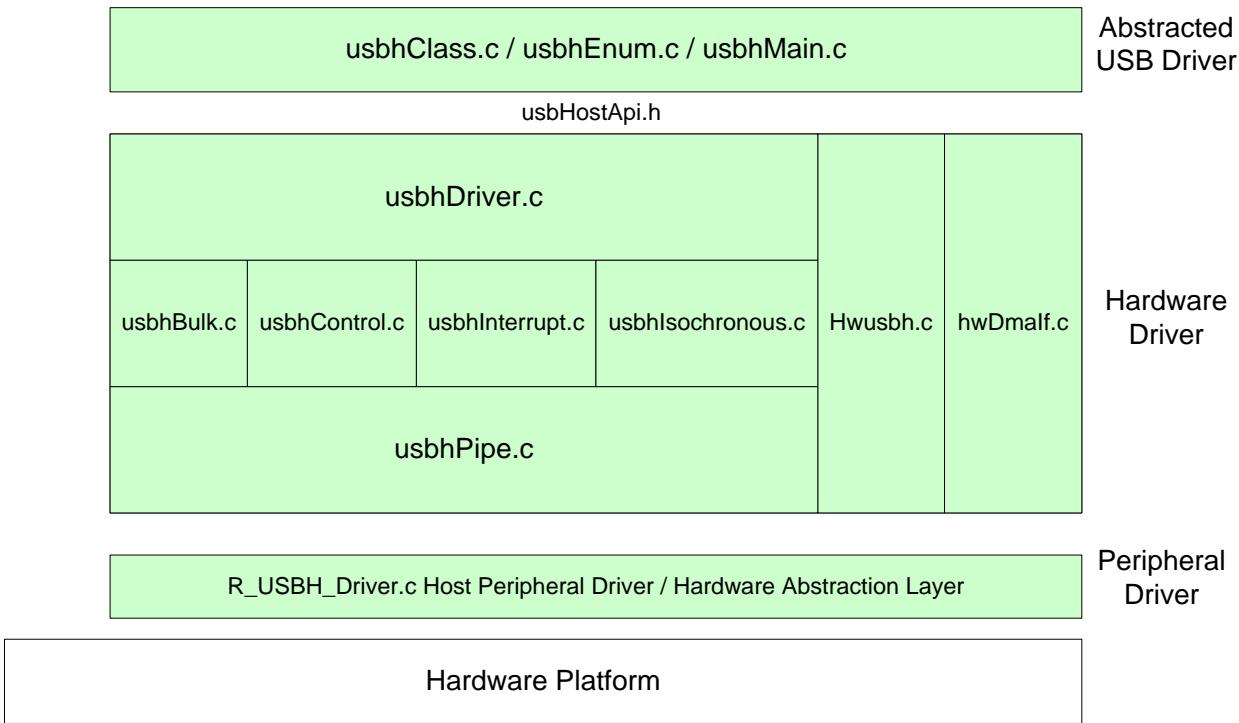


Figure 4-22 Hardware Driver Overview

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 end-point is configured from a class driver and provides routing information for the data to be sent or received through a pipe. The end-point 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
USBT	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 a 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 RZA1LU Software package allows for a website shows 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.

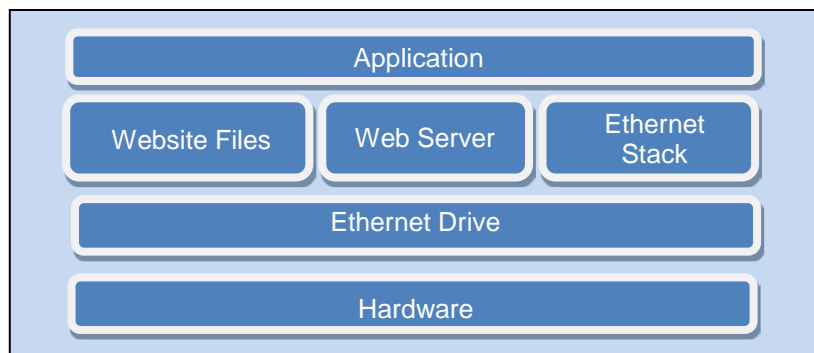


Figure 4-23 Website Hierarchical Software Layer

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 Application

Ensure that the Stream it! 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.

```

RZ/A1LU RZ/A Software Package Ver.3.01.0542
Copyright (C) Renesas Electronics Europe.

REE> Ethernet Controller "\\.\ether0"
Link Up
MAC      = 68:65:73:65:20:61
DHCP     = Enabled
Address  = 192.168.  2.102
Mask     = 255.255.255.  0
Gateway  = 192.168.  2.  1

```

Figure 4-24 Website Application

Once the Address is retrieved, '192.168.2.102' 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 RZA1LU device. Furthermore, the user may toggle the 'USER' LED through enabling and disabling tick box.

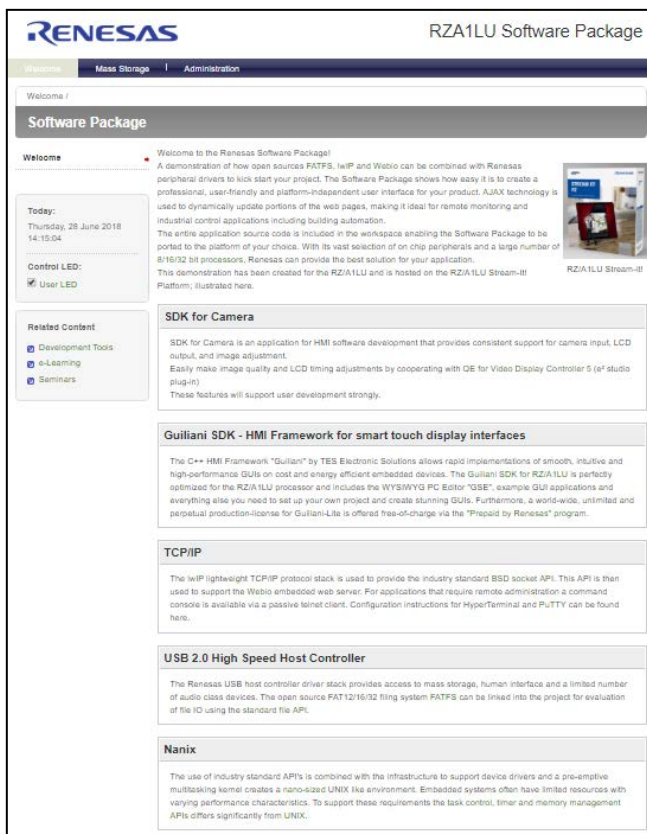


Figure 4-25 Welcome page

The 'Mass Storage' page allows for the user to interact with the connected mass storage device. In the image below, the Folders 1 to 5 and file 1 to 5 may be manipulated.

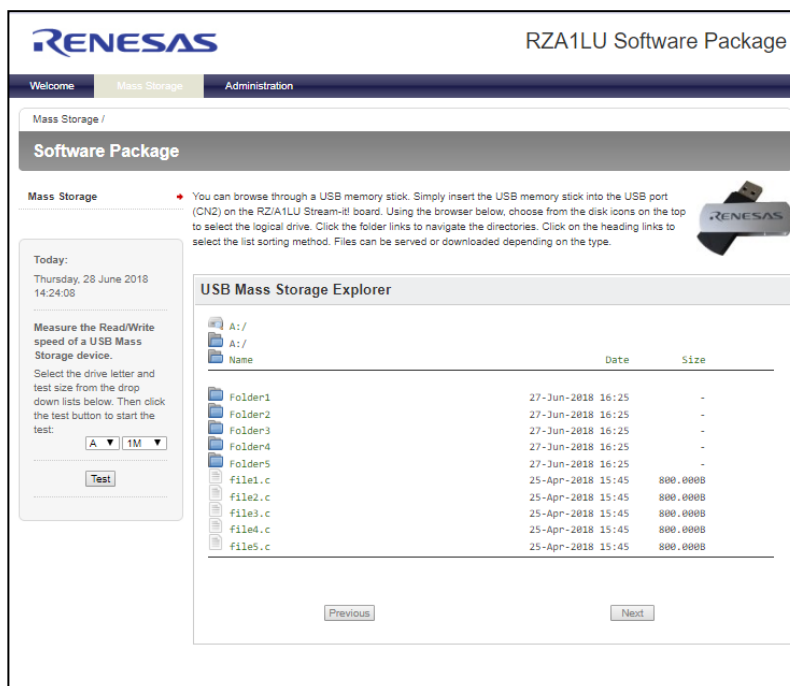


Figure 4-26 Mass Storage Page

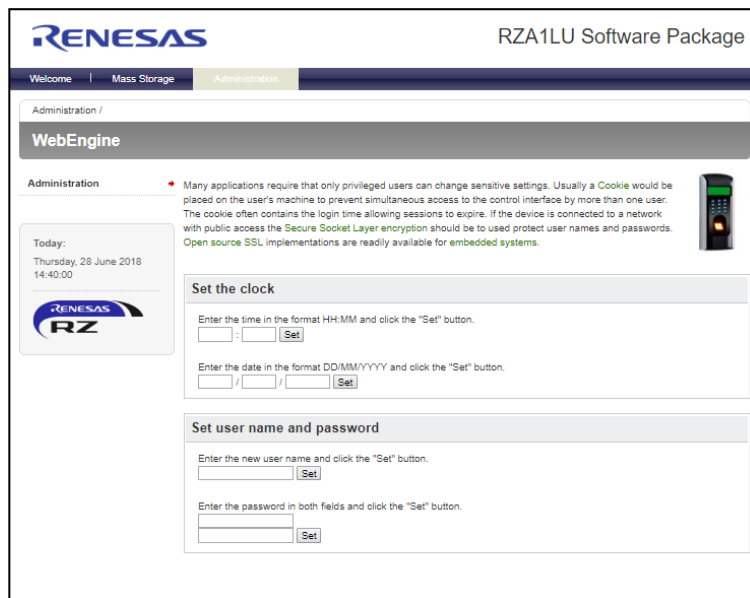
The user may also go into the Administration Page. This page requires for the user to have a user name and password. The username and password is stored in the on-board EEPROM.

The default details are:

Username: Admin

Password: Password

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



RENESAS RZA1LU Software Package

Welcome | Mass Storage | **Administration**

Administration /

WebEngine

Administration

Many applications require that only privileged users can change sensitive settings. Usually a Cookie would be placed on the user's machine to prevent simultaneous access to the control interface by more than one user. The cookie often contains the login time allowing sessions to expire. If the device is connected to a network with public access the Secure Socket Layer encryption should be to used protect user names and passwords. Open source SSL implementations are readily available for embedded systems.

Today:
Thursday, 28 June 2018
14:40:00

RENESAS RZ

Set the clock

Enter the time in the format HH:MM and click the "Set" button.
[] : [] [Set]

Enter the date in the format DD/MM/YYYY and click the "Set" button.
[] / [] / [] [Set]

Set user name and password

Enter the new user name and click the "Set" button.
[] [Set]

Enter the password in both fields and click the "Set" button.
[] [Set]

Figure 4-27 Administration Page

4.6.3 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
- byEnabledDHCP: override local settings with configuration from DHCP server flag.

When byEnabledDHCP 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 NVDHCP_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      byEnabledDHCP;        // Use DHCP assigned settings
} NVDHCP,
*PNVDHCP;
```

Figure 4-28: 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 PBufs) 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 PBufs when the network driver has transmitted the data. The lwIP task is responsible for creating and destroying the PBufs. The ipOutput function (called by lwIP) is responsible for writing the PBufs to the network driver (see Figure 4-29 Relationship of tasks).

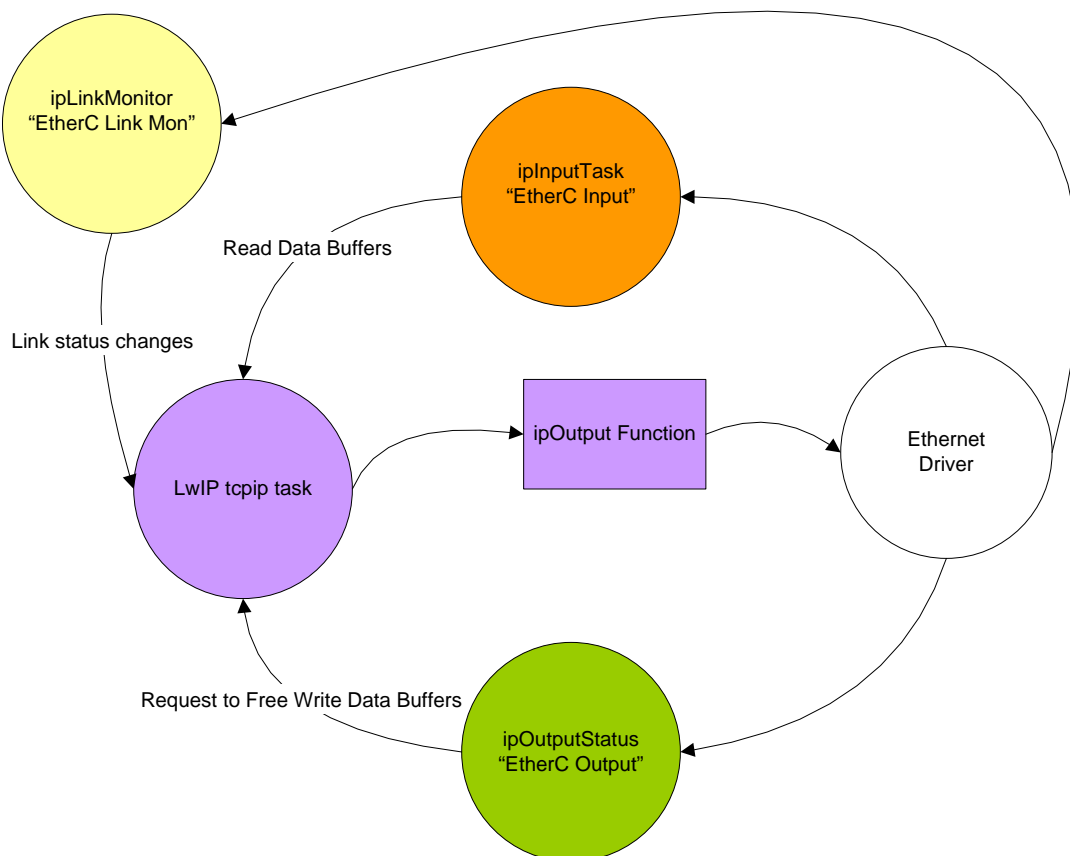


Figure 4-29 Relationship of tasks

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 4-30 IpLinkMonitor task Overview, note that messages sent to initiate action within the lwIP task have a 'MSG lwip:' pre-fix).

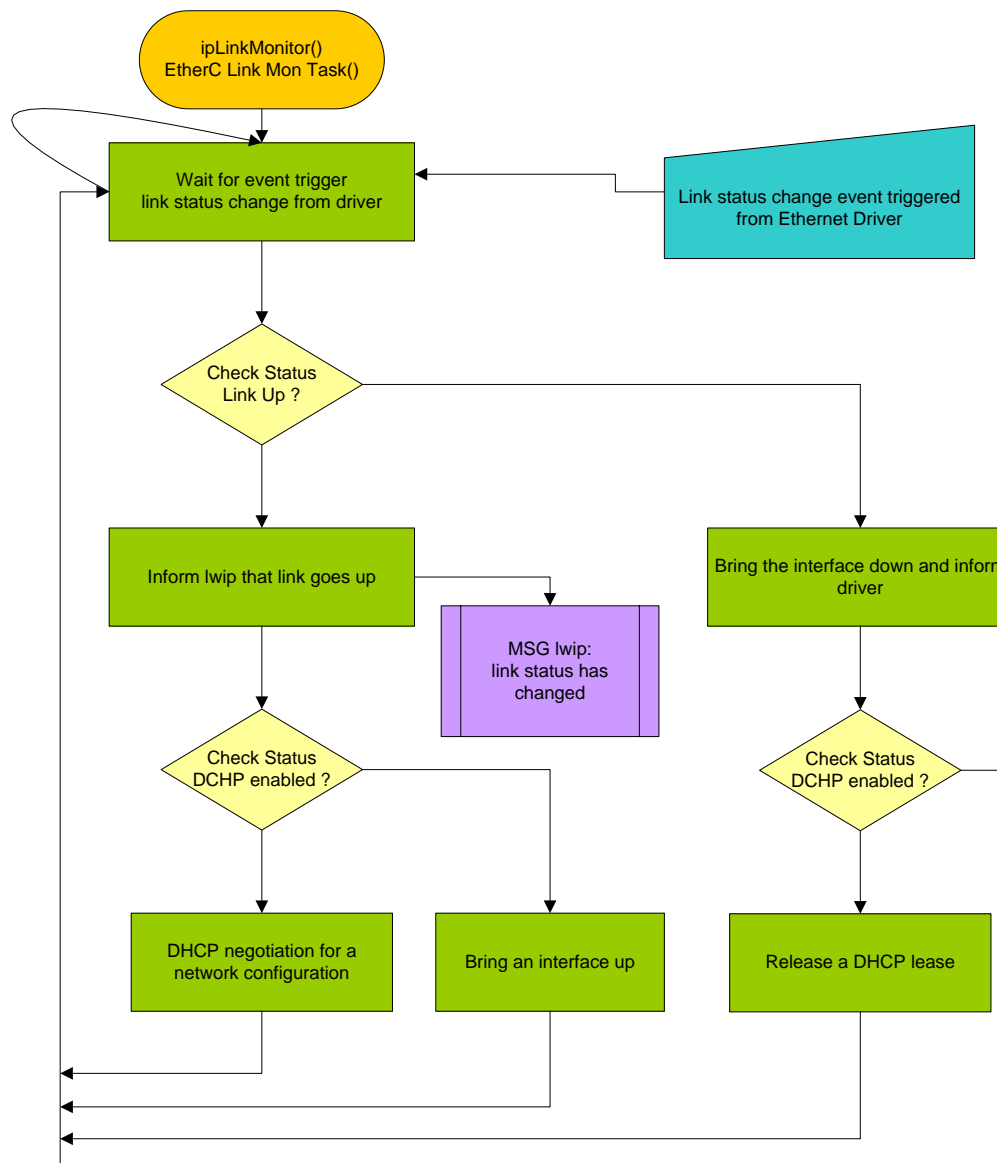


Figure 4-30 IpLinkMonitor task Overview

4.6.3.3 IpInputTask Details

The ipInputTask task allocates and submits the buffers (PBufs) 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 4-31 IpInputtask Overview note that messages sent to initiate action within the lwIP task have a 'MSG lwip:' pre-fix).

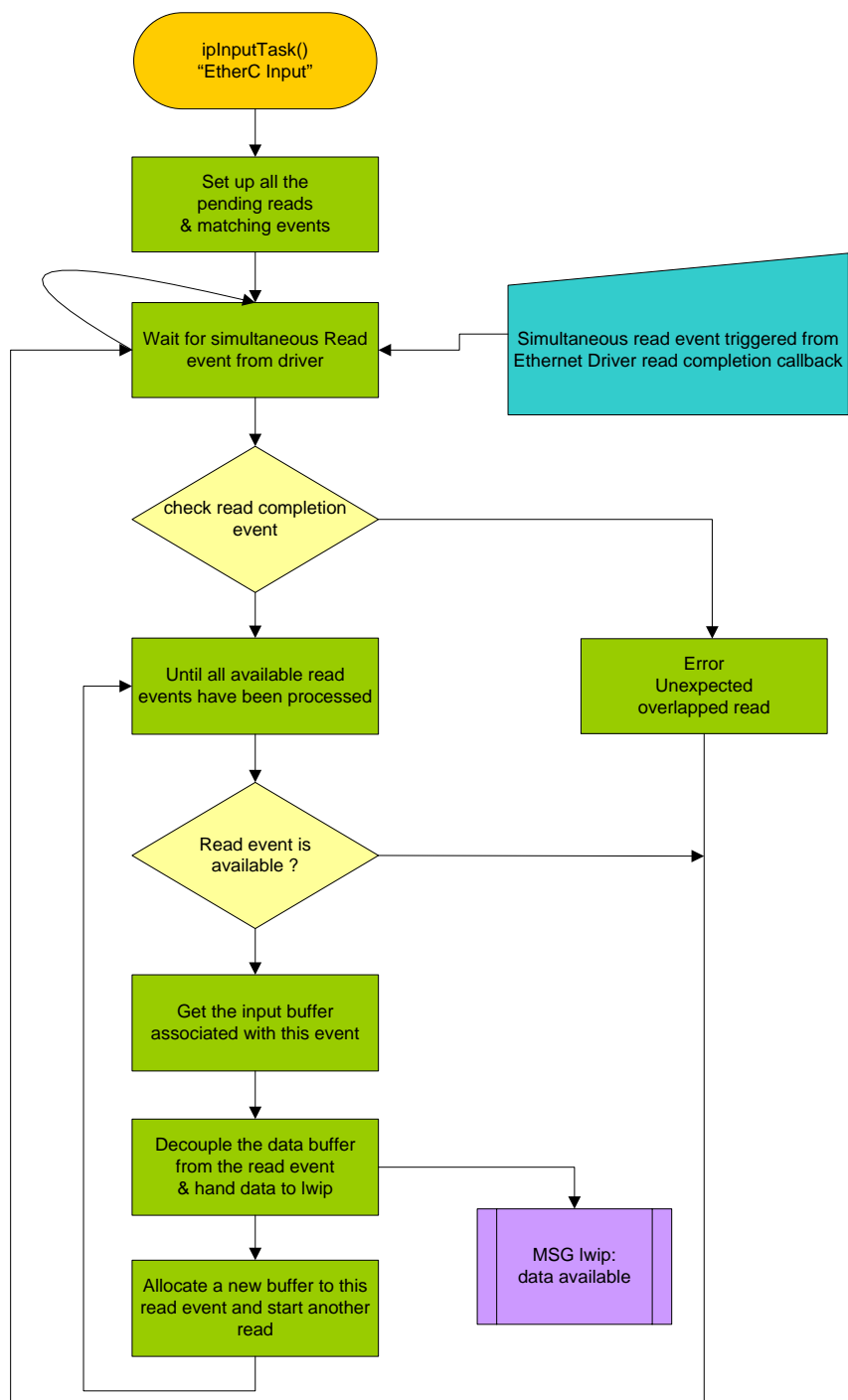


Figure 4-31 IpInputtask Overview

4.6.3.4 IpOutputStatus Details

The IpOutputStatus function frees the buffers (PBufs) 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 (PBuf) 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 4-32 IpOutputStatus Overview, note that messages sent to initiate action within the lwIP task have a 'MSG lwip:' pre-fix).

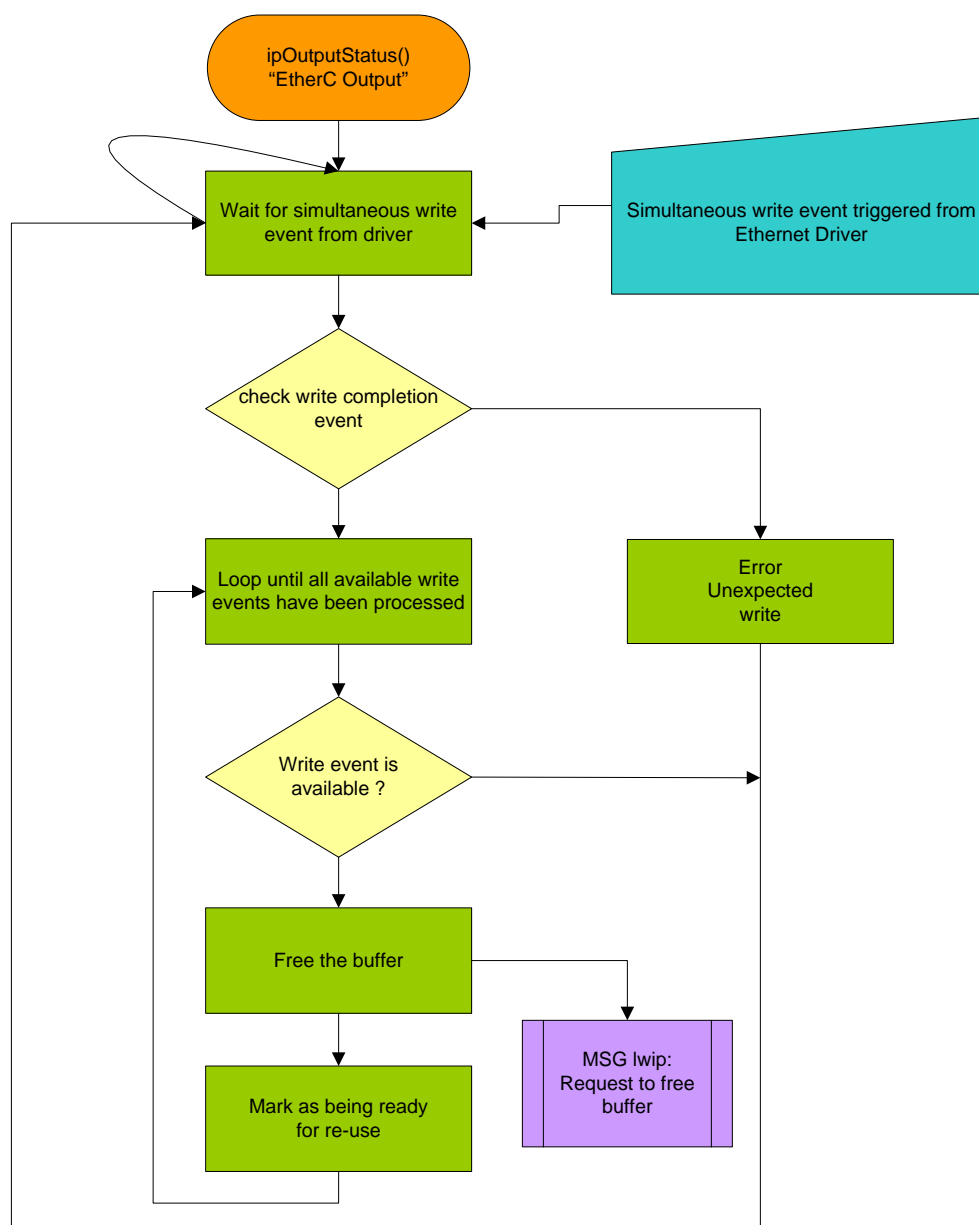


Figure 4-32 IpOutputStatus Overview

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.

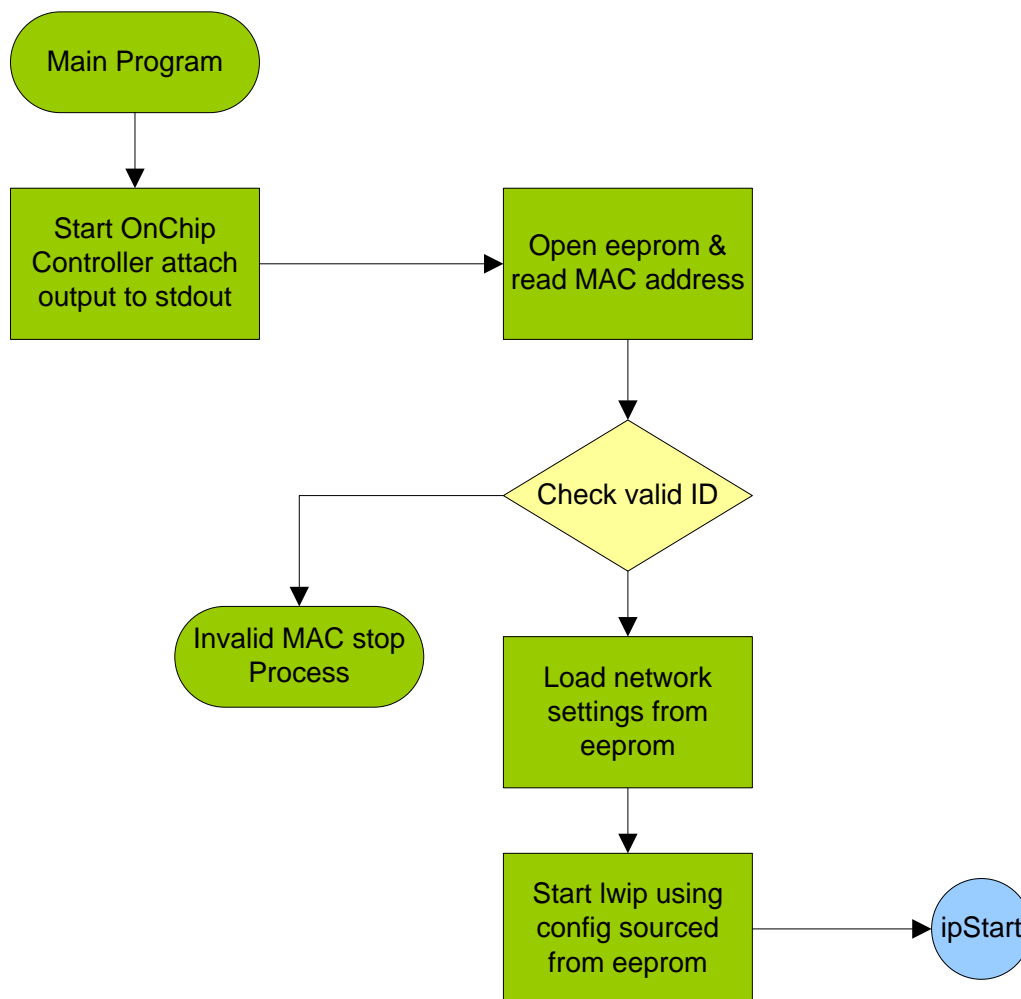


Figure 4-33 Initialisation Sequence

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 IO 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.

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

4.6.4 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:

4.6.5 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.

4.6.6 Website Files

For static web development the html, css and javascript for the website is included in the folder src\renesas\Application\WebSite.

4.7 Camera

The RZ/A1LU which supports camera input has peripheral devices VDC5, and CEU. The VDC5 and CEU are for digital 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.

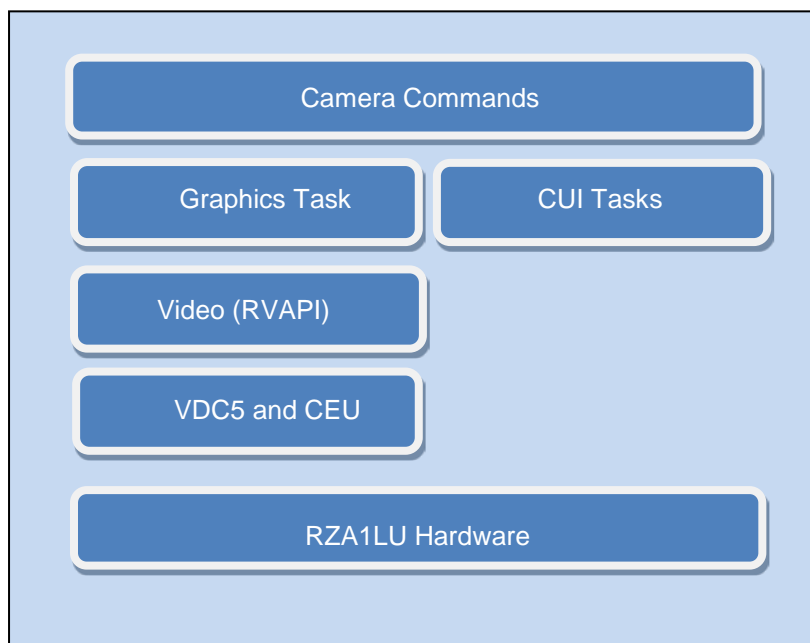


Figure 4-34 Camera Hierarchical Software Layer

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_DISABLE)

```

4.7.2 Commands

The commands for the Camera Applications are described in Table 4-7.

Command	Operation
Numerical value	Operations in each menu - Selection of image adjustment content - Selection of image adjustment position (selection of H/W 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 H/W manual)
B, b	Return to the previous menu
R, r	Output current image adjustment value
T, t	Return to the Top menu
Enter	Determine contents inputted
Delete / Backspace	Delete one character from the input character

Table 4-7 Camera Commands

This sample program implements CUI task which can perform image adjustment in real time with Terminal Application on PC. The CUI task operates the register value of VDC5 by command input from PC's Terminal Application to realize image adjustment.

This chapter describes the screen operation method and commands from Terminal Application.

Note, however, that in case of adjusting images with QE for Display, it is not allowed to combine use of CUI and QE for Display.

4.7.3 Menu

Display menu of Terminal Application and operation overview shown in Figure 4-35.

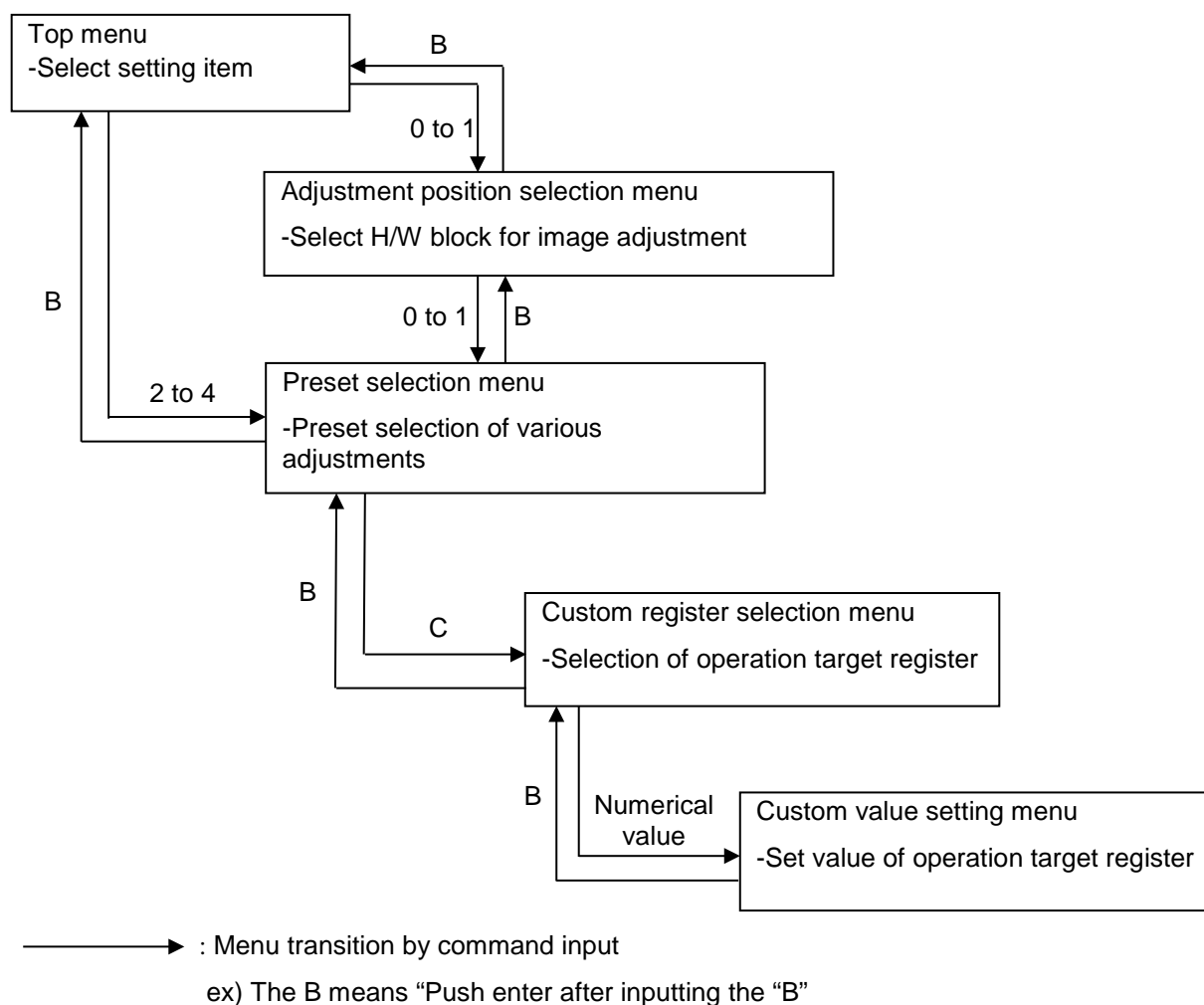


Figure 4-35 Display menu and operation overview

4.7.3.1 Acquisition of image adjustment value

The user can acquire various image adjustment values set by preset selection or custom by executing the R command. When the R command is executed, the current values of the various adjustment parameters are output on the Terminal Application in a format that can be directly applied to the C language source code. (header format) To apply the output setting value when initializing this sample, overwrite the contents outputted on the Terminal Application to the following file the CUI operation commands on the Terminal Application.

4.7.3.2 Process Sequence

Figure 4-36 shows the process sequence of this sample program.

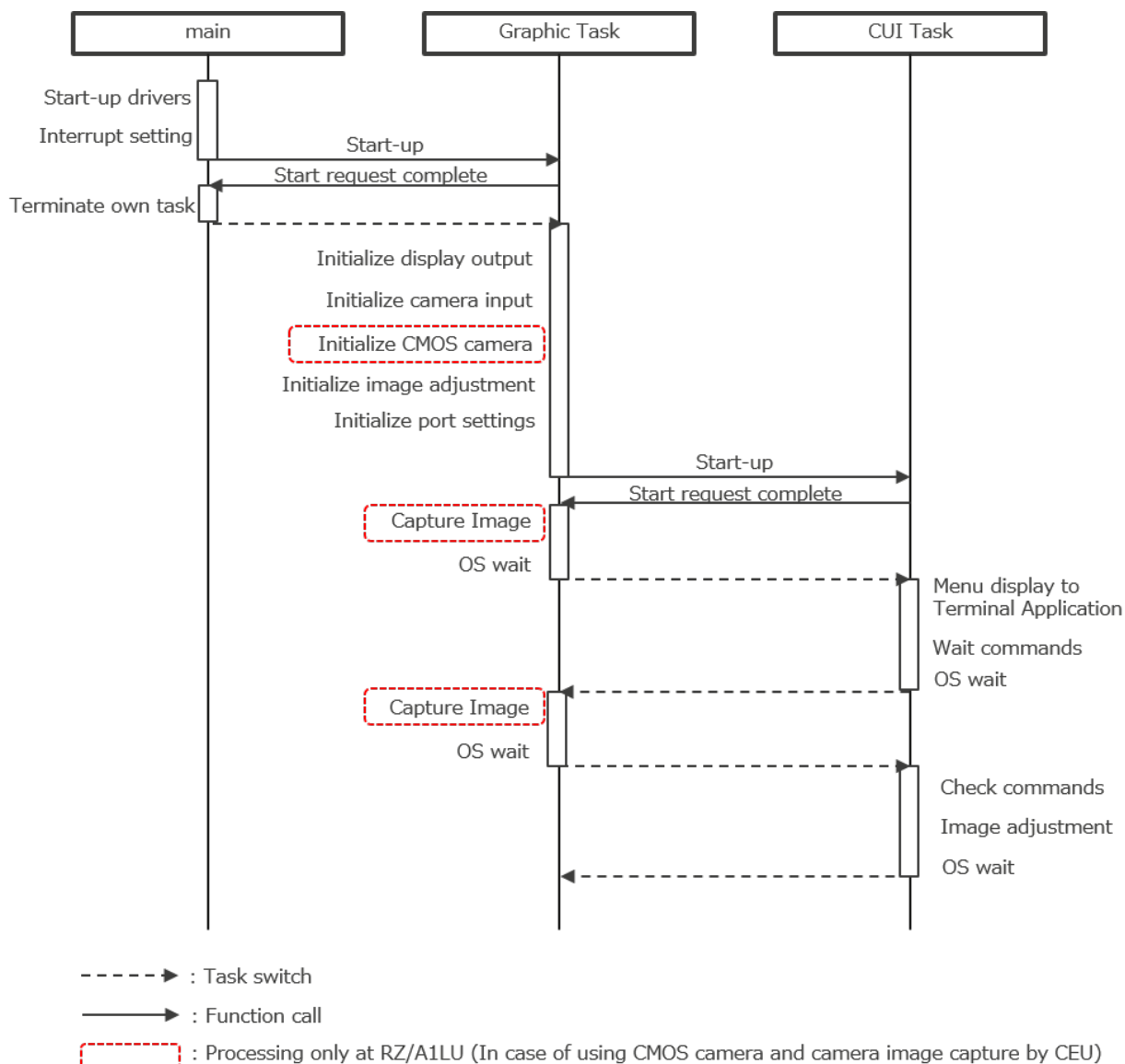


Figure 4-36 Process Sequence of SDK for Camera

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 4-37 shows the H/W configuration for RZ/A1 image input/output.

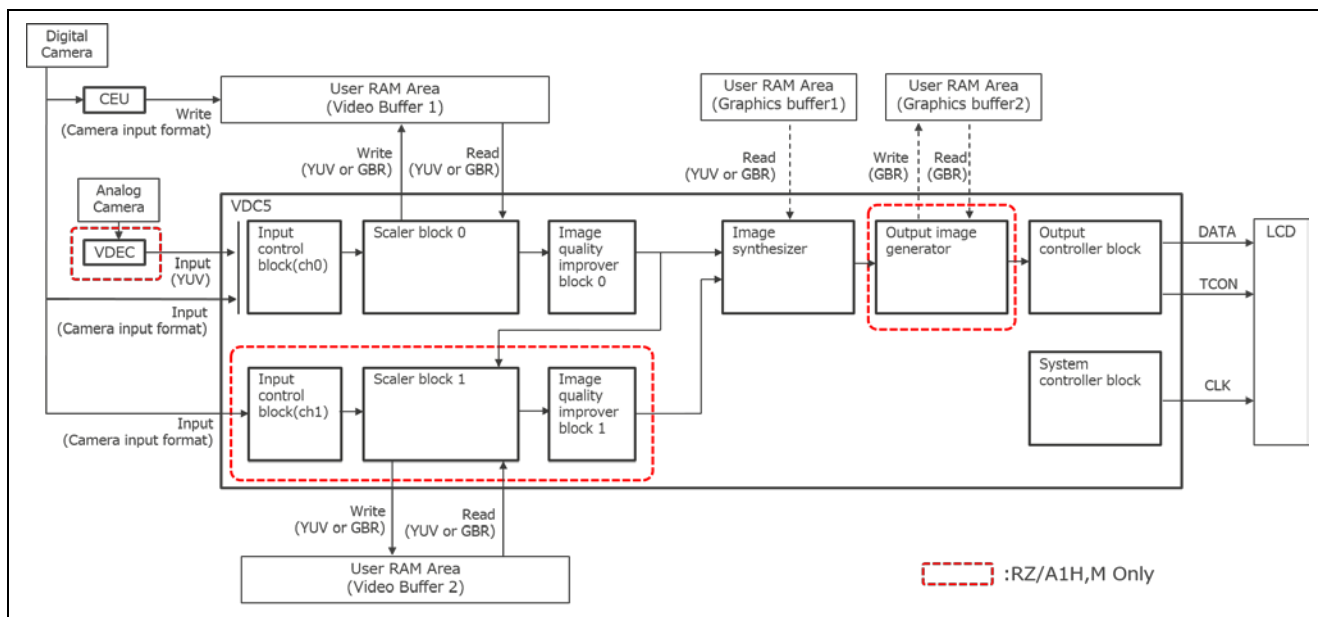


Figure 4-37 Block Diagram of the H/W Configuration for RZ/A1 Image Input/Output

For more information regarding the camera please refer to the Application Note: SDK for Camera Sample Program (R01AN4312EJ)

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.

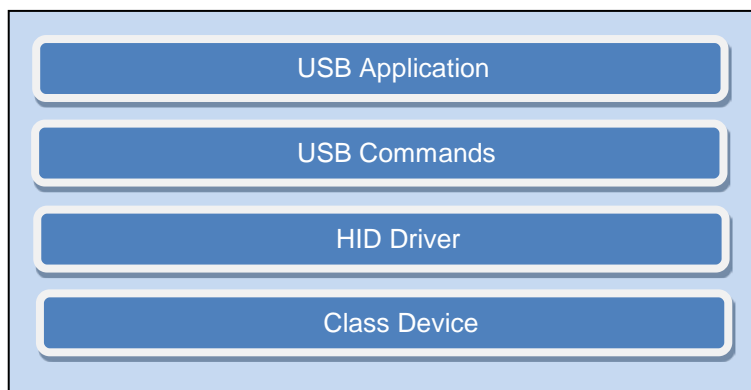


Figure 4-38 USB Function Hierarchical Software Layer

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 4-8 shows the USB Functions commands.

Command	Description
usbm	Provides access to the USB mouse application
usbk	Provides access to the USB keyboard application

Table 4-8 USB Function Commands

4.8.3 USB HID Mouse

The USB HID Mouse application allows for the user to visualize the coordinates and scroll of the mouse as well as highlighting which mouse button is pressed. Figure 4-39 shows the application of the USB HID Mouse.

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

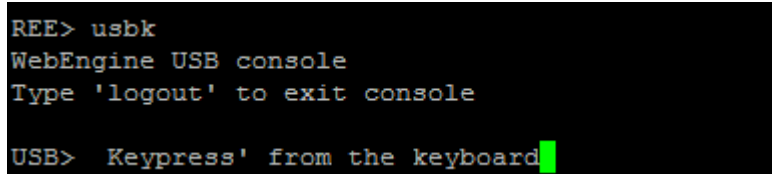
Figure 4-39 USB HID Mouse Application

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 4-40 shows the application of the USB HID Mouse.



```
REE> usbk
WebEngine USB console
Type 'logout' to exit console

USB> Keypress' from the keyboard
```

Figure 4-40 USB HID Keyboard Application

The USB HID keyboard application is included as part of the `cmd_usb.c` found at `src/renesas/application/console/cmd`. Similarly 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/A1LU 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.

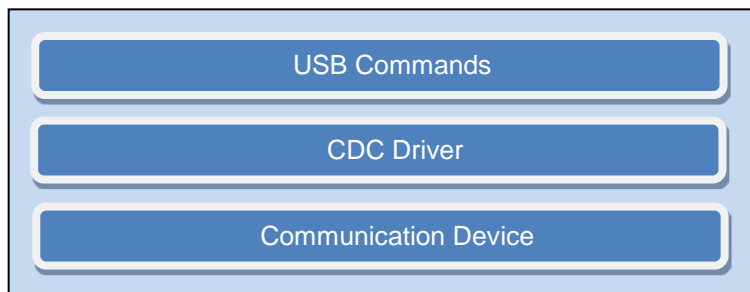


Figure 4-41 USB Function Hierarchical Software Layer

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)
```

4.9.2 Commands

Table 4-8 shows the USB CDC commands.

Command	Description
sopen	Opens a CDC device
sclose	Closes a CDC device
sctlst	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 4-9 USB Function Commands

4.9.3 CDC Driver

The CDC Middleware API can be seen in Doxygen under RZ/A1LU 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 microphone. To achieve this a single jack headset (headphones and a mic) will be needed to connect to CN14.

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 RZA1LU Stream it! Board, allowing control of the headphone and microphone volume, sampling frequency etc. Audio data to and from the CODEC is sent via the SSIF driver, using DMA channels to minimize CPU overheads.

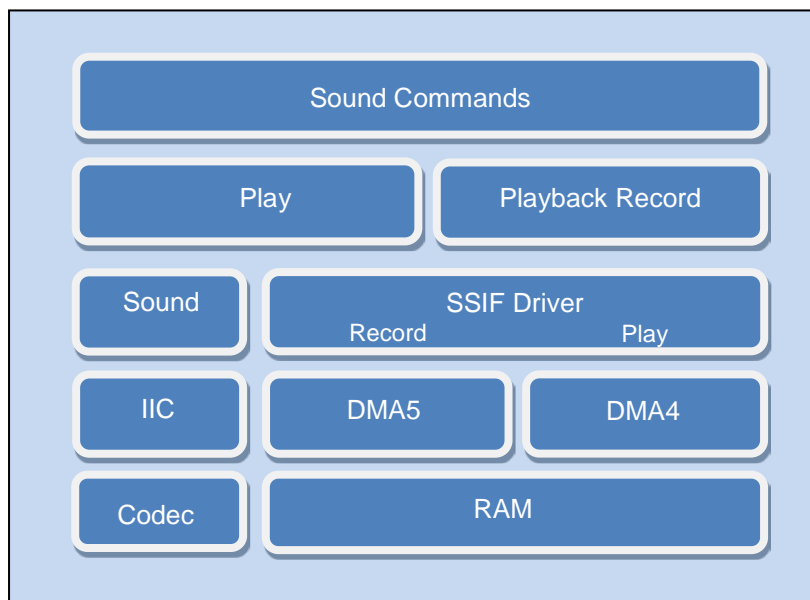


Figure 4-42 Sound Function Hierarchical Software Layer

4.10.2 Console Commands

The RZA1LU Stream it! 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' which mean "left" and "right" in English.
record	Allows for the sound picked up by the mic to be played through the headphones.

Table 4-10 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 16bit, 44.1kHz format.

The playback application starts in the function R_SOUND_PlaySample, which is run when the command “play” is entered in to 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.

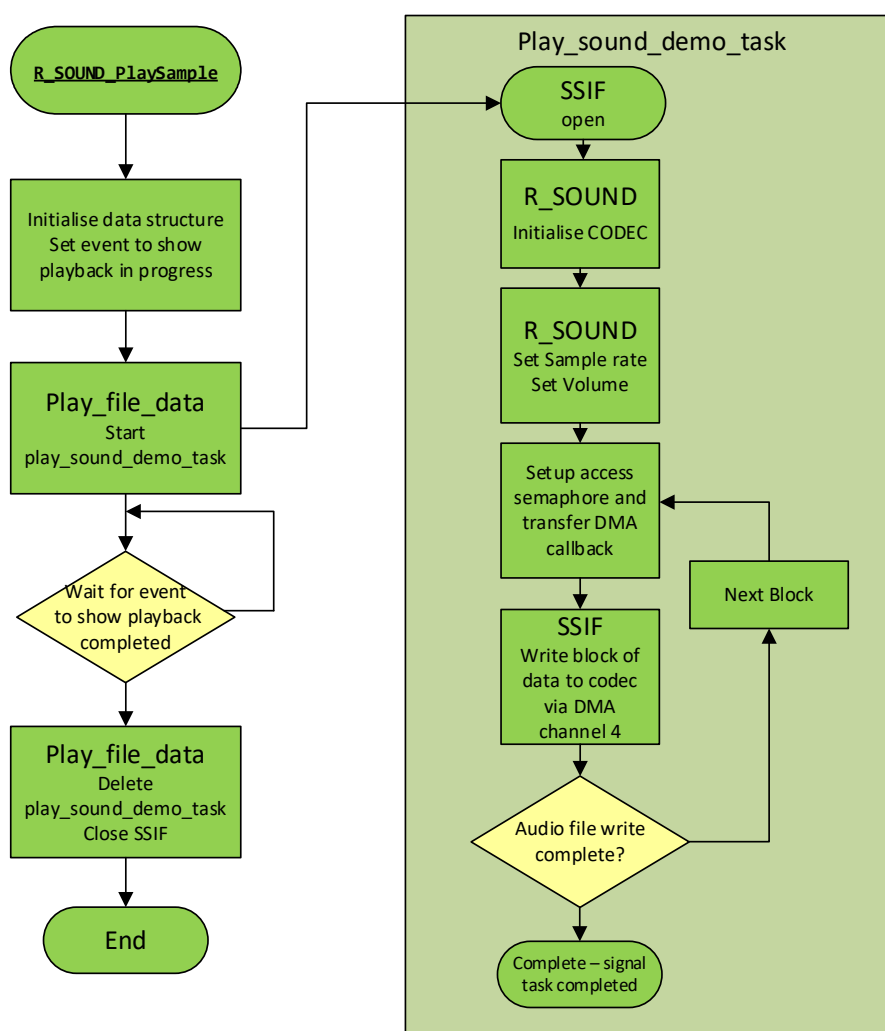


Figure 4-43 Play Application Flowchart

4.10.5 Record Software Application

The record software application reads the input from a microphone 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 a function `play_recorded`. The function `play_recorded` creates a buffer area, initializes access control semaphores, and configures the SOUND driver, before creating a task; `task_record_sound_demo`, then 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.

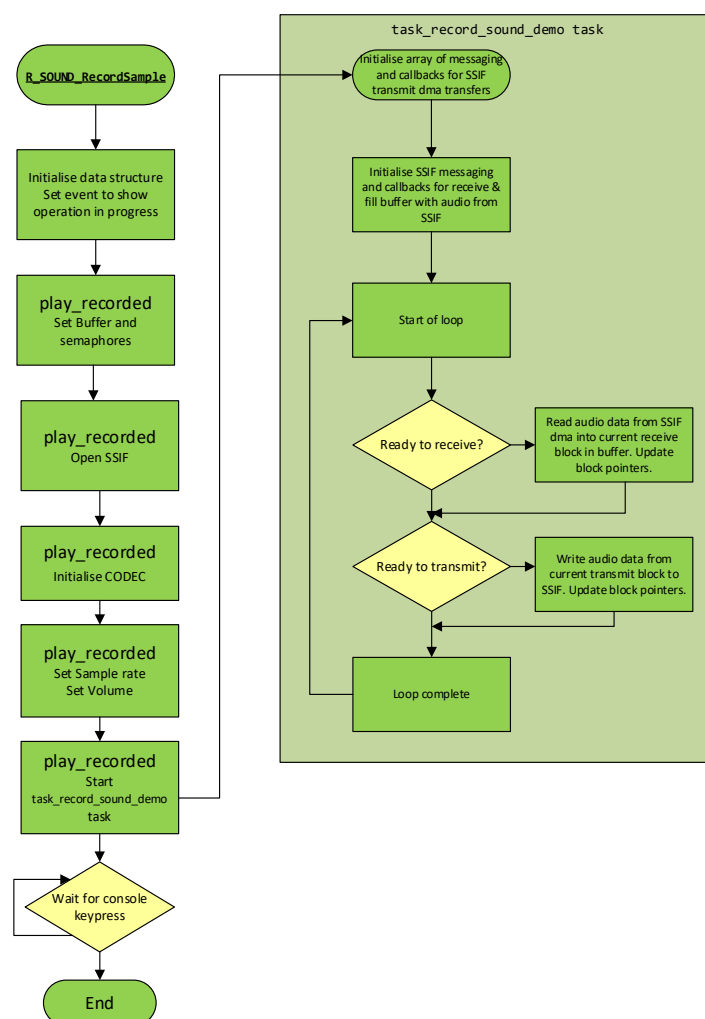


Figure 4-44 Record Application Flowchart

4.11 Touchscreen Application

The touchscreen application allows for the touch capability of the RZ/A1LU Stream IT! Development Kit LCD to be used. This is achieved through using the RIIC peripheral (channel 1) on the RZ/A1LU.

The software architecture is seen below.

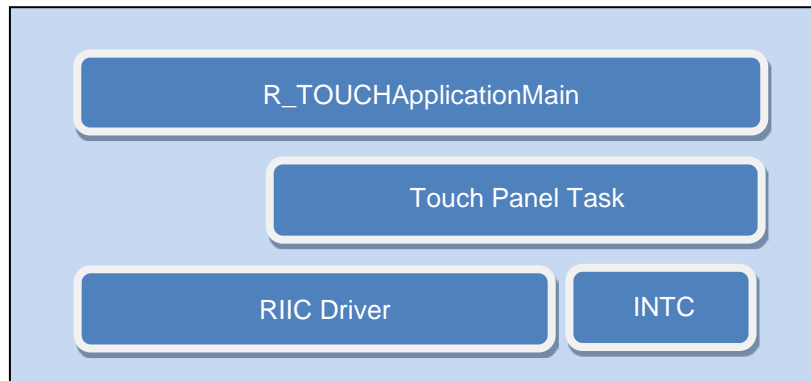


Figure 4-45 Sound Function Hierarchical Software Layer

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 Touchscreen Software

For more information regarding the Touchscreen Application please refer to the Application Note RZ/A1LU Touch Panel Utility (R01AN4314EJ).

4.12 TES GUILIANI

Guiliani is a third party easy-to-use framework for the quick and uncomplicated creation of visually appealing graphical user interfaces on embedded systems.

It combines the comfort of a PC based development toolchain with the benefits of a highly-optimized software framework, specifically designed for the use on resource-limited embedded hardware.

The GUI Sample on the RZA1LU Stream it! V2.3 board shows a sample application that indicates how to connect the Stream-it! Platform to the Guiliani library. This sample provides shows how to control and LED and show the value of the RTC.

Following figure shows the image of the screen displayed on Stream it! board.

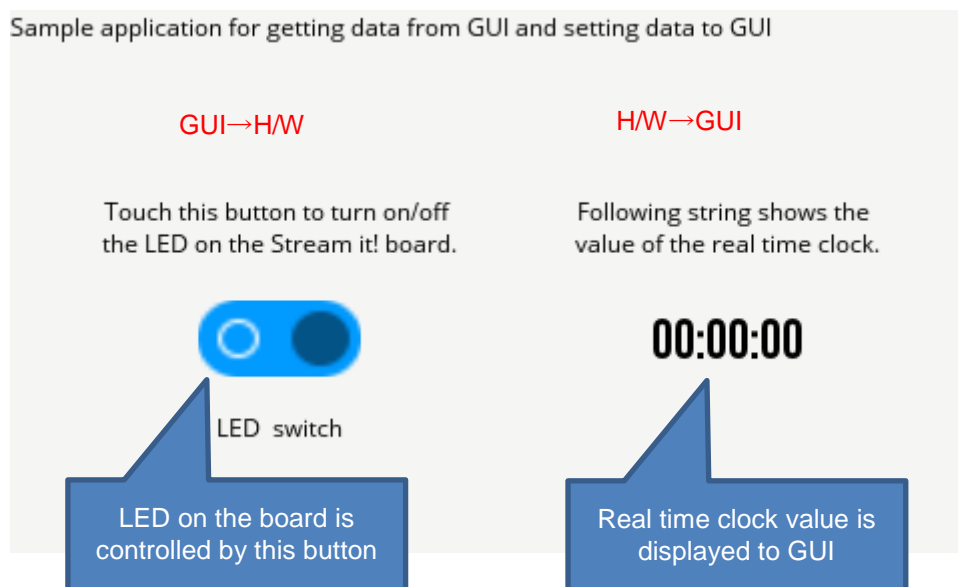


Figure 4-46 Sample Guiliani Application

For more information regarding the Guiliani Development please refer to the GUI Sample Program Application Note (R01AN4413EJ).

Website and Support

Renesas Electronics website

<http://www.renesas.com/>

Inquiries

<http://www.renesas.com/contact/>

All trademarks and registered trademarks are the property of their respective owners.

Revision History

Rev.	Date	Description	
		Page	Summary
1.00	Jul 11, 2018	All	Created document.