Renesas RA Family

CAN FD Example Using RA4E2 and RA6E2

How to Use This Document

This document is intended for audiences with various levels of beginning knowledge, experience, and confidence. The document has been broken up into four main sections:

1. **Introduction** – This is comprised of the introductory pages (before the Table of Contents) informing the reader of what is required to work through the example project and providing audiences with an introduction to CAN FD technology and its intended use.
2. **Background** – This is comprised of section 1, which talks extensively about the reasons for use and the functionality of CAN FD. It is intended for those deciding if CAN/CAN FD is right for their system, or those curious about the technology.
3. **Example** – Section 2 walks the reader through an example use of CAN FD by first creating a working application on an EK-RA6E2 board and then repeating that process for an EK-RA4E2 board and connecting them together to communicate. Section 2 includes both these base projects and details on how to enhance them to create a final example application.

   Accompanying this application note are four attached project files. Two of the projects, labeled "_base" in the name, are the starting point for the APN. There is one for the RA4 and one for the RA6. The other two projects labeled "_final" are the pre-built final projects for quick reference and validation.

   The "_base" projects actually contain the completed application projects as well, in a git repository. They are checked out at the commit for the base project. If you are comfortable with git, you can simply work with these two projects. If you are not, or do not have Git on your machine, the two "_final" projects contain the final, completed version, for user reference.

   Note: If you are not comfortable with project creation and just want to run the completed project, load the two final versions and skip sections 2-4.
4. **Reference** – For those readers who already have a working CAN FD application and only want the technical information to execute it, the API Section plus Sections 3-6 address that need.

Introduction [Why CAN FD, why even CAN?]

It is often desirable to have multiple processors working on an application in tandem. If the problem is complex, multiple functions need to happen asynchronously and sometimes the individual processing parts are too far apart to wire together at a central point. These individual processing engines must communicate to coordinate data or actions. There are many protocols to support communications but many of them are not real-time-aware. The original CAN protocol was designed with specific real-time coordinated functionality in mind (see background in section 1).

CAN FD is an extension that allows a larger amount of data to be exchanged in a real-time manner to ensure proper functionality.

While originally developed for a more costly application, CAN and CAN FD, as on-chip MCU peripherals, have reduced the cost and design complexity that it is now even appropriate for some consumer electronic devices, as well as industrial, HVAC, building, and other systems that can be cost/time sensitive.

CAN FD solves many of the problems designers face when rolling their own protocols such as SAE J1939, CANopen, and DeviceNet – even on top of stable transport layers.

Target Device

RA4E2 and RA6E2 (to follow this application note exactly)

[Note: Because of the way the FSP is designed, the description in this application note can be easily applied to other RA devices with CAN FD peripherals.]

Required Resources

To build and run the associated example project, you will need the following:

- EK-RA4E2 (https://www.renesas.com/us/en/products/microcontrollers-microprocessors/ra-cortex-m-mcus/ek-ra4e2-evaluation-kit-ra4e2-mcu-group
- EK-RA6E2 (https://www.renesas.com/us/en/products/microcontrollers-microprocessors/ra-cortex-m-mcus/ek-ra6e2-evaluation-kit-ra6e2-mcu-group
- Three jumper wires
- e^2 studio ISDE, version 2023-10 or later
- RA-Family Flexible Software Package (FSP), v5.1.0

The FSP and e^2 studio are bundled in a downloadable platform installer available on the Renesas website at: renesas.com/ra/fsp

Prerequisites and Intended Audience

This application note assumes you have some experience with the Renesas e^2 studio ISDE and RA Family Flexible Software Package (FSP). Before you perform the procedures in this application note, familiarize yourself with e^2 studio and the FSP and validate the debug connection to your boards.

The intended audience are users who have determined that CAN FD is a valid solution to their communication needs in a multi-processor system.

Note: If you are using this Application Note as a reference for implementing on your own proprietary system and you understand the background, you can skim Section 1 and skip Section 2 entirely, as stated above.

Note: This document is not intended to provide a full, deep-dive, explanation of CAN, but only a basic understanding for both an effective understanding of the lab work and the use of the Renesas FSP and hardware resources.

Contents

1. Background on CAN FD

CAN FD, which stands for Controller Area Network with Flexible Data-Rate, is an extension of the original CAN (Controller Area Network) protocol. CAN FD was developed to address the increasing demand for higher data transfer rates and larger payload sizes in automotive and industrial applications. CAN FD is a protocol based on the CAN definition that allows larger data block to be transmitted efficiently by changing transmission speeds for the bulk data exchange. The protocol provides a flexible and scalable solution that maintains compatibility with the original CAN protocol while offering higher data rates and larger payload sizes.

1.1 Why Do We Need CAN FD?

As previously stated, processing engines working together need a data exchange and coordination process, and CAN FD provides that base layer of interaction. CAN FD is needed to address the evolving requirements of modern embedded systems, particularly in automotive and industrial domains. It provides the necessary enhancements in terms of data rate, payload size, flexibility, and reliability, enabling it to meet the challenges posed by increasingly sophisticated and data-intensive applications.

1.2 How CAN and CAN FD Came About

There has always been a need for MCUs and other subsystems to work together, share data, and synchronize events. Over the years, many standard and proprietary communication and network standards have emerged. For non-time-critical data exchange, most standards will suffice and are geared towards bulk data transfer with large messages (like USB, Ethernet, Modbus, BacNet, and so forth.). But for real-time data exchange, these bus types do not perform well in harsh environments such as industrial and automotive domains. The CAN protocol was designed to transfer data with large message sizes in real time.

1.3 CAN Bus Overview

CAN bus was originally designed by Bosch corporation for use in communicating between small devices in an automobile to replace the large wiring harness with a power/bus system (what we now call the “Controller Area Network” was originally called “Car Area Network” when first conceived).

Because of its originally intended use and environment, CAN has some very interesting inherent features some of which are: High-noise environment; short, real-time messages; robust collision detection and recovery, and differential signaling for high SNR without high voltage swings.

The Controller Area Network (CAN) follows the OSI (Open Systems Interconnection) model, which is a conceptual framework used to understand and describe network protocols. CAN operates mainly at the physical and data link layers of the OSI model, and higher-layer functions are typically handled by additional protocols or applications built on top of the CAN bus. The simplicity, reliability, and deterministic nature of CAN make it well-suited for real-time communication in embedded systems. The transmission speed is 1 MHz. Many MCU devices have hardware modules on them, which are dedicated not just to the toggling of the lines (like a UART) but a CAN/CAN FD peripheral has hardware to decode part of the messaging to filter out messages not intended for the particular node. This is done appropriately so a very simple MCU can avoid handling messages not intended for it.

As an aside, Renesas has been including CAN hardware in selective MCUs from as far back as the old Hitachi H8 devices before the year 2000, – and they have been refining the hardware and firmware support since that time. Because of its long history, on-chip hardware support, and inherent environmental advantages, CAN bus has long-since moved beyond automotive applications to industrial and medical devices. In fact, the industry is seeing it be employed in higher-end consumer devices, where multiple processors are used in concert to address system design needs in an effective way.

1.4 Base functionality - CAN

At the hardware level, a CAN message, called a “Frame” in CAN bus terminology, is defined as follows:

SOF	11-bit Identifier	RTR	IDE	r0	DLC	0…8 Bytes Data	CRC	ACK	EOF	IFS

Figure 1. CAN Message Frame

SOF–The single dominant start of frame (SOF) bit marks the start of a message, and is used to synchronize the nodes on a bus after being idle.

Identifier-The standard CAN 11-bit identifier establishes the priority of the message. The lower the binary value, the higher its priority.

RTR–The single remote transmission request (RTR) bit is dominant when information is required from another node. All nodes receive the request, but the identifier determines the specified node. The responding data is also received by all nodes and used by any node interested. In this way, all data being used in a system is uniform.

IDE–A dominant single identifier extension (IDE) bit means that a standard CAN identifier with no extension is being transmitted.

r0–Reserved bit (for possible use by future standard amendment).

DLC–The 4-bit data length code (DLC) contains the number of bytes of data being transmitted.

Data–Up to 64 bits of application data may be transmitted.

CRC–The 16-bit (15 bits plus delimiter) cyclic redundancy check (CRC) contains the checksum (number of bits transmitted) of the preceding application data for error detection.

ACK–Every node receiving an accurate message overwrites this recessive bit in the original message with a dominate bit, indicating that an error-free message has been sent. Should a receiving node detect an error and leave this bit recessive, it discards the message and the sending node repeats the message after arbitration. In this way, each node acknowledges (ACK) the integrity of its data. ACK is 2 bits, where the first bit is an acknowledgment bit and the second is a delimiter.

EOF–This end-of-frame (EOF), 7-bit field marks the end of a CAN frame (message) and disables bit-stuffing, indicating a stuffing error when dominant. When 5 bits of the same logic level occur in succession during normal operation, a bit of the opposite logic level is stuffed into the data.

IFS–This 7-bit interframe space (IFS) contains the time required by the controller to move a correctly received frame to its proper position in a message buffer area.

There is a second frame definition, called “Extended CAN” which uses a 29-bit identifier, but this is rarely used and is beyond the scope of this document.

At the beginning of a transmission there is an arbitration phase to determine which device controls the bus, if two or more devices request to communicate on the bus at the same time. Arbitration is done through comparing the 11-bit identifier at the beginning of the frame, which has the message/node priority embedded in it. This would allow a node with an emergency indication to come through with a higher priority with, say temperature or fan speed control messages.

Because of the arbitration and collision detection/resolution, the CAN bus is considered a multi-commander bus. This is different than other buses where there is a master controlling traffic on the bus by initiating all messages and passing on responses to the appropriate endpoint.

In CAN and CAN FD, messages received by a node that are waiting to be processed, are stored in “Mailboxes” which can be prioritized for processing or processed in a FIFO manner, depending on the application requirements.

1.5 CAN FD

CAN FD is a new, higher speed standard that can co-exist with a CAN network. It obviously has a faster throughput because of its speed (8 MHz vs. 1 MHz), but it can also carry a larger payload per message – up to 64 bytes.

On the surface, as a simple comparison, this is the simple difference. To ensure these CAN FD nodes can coexist with regular CAN nodes on the same bus, the arbitration phase is done at the slower, 1 MHz rate. After the CAN FD device “owns the bus” it can transmit to other CAN FD devices at the higher, 8 MHz-speed and larger payload.

An MCU with a CAN FD peripheral can manage both CAN and CAN FD messaging. The CAN FD frame looks more like this:

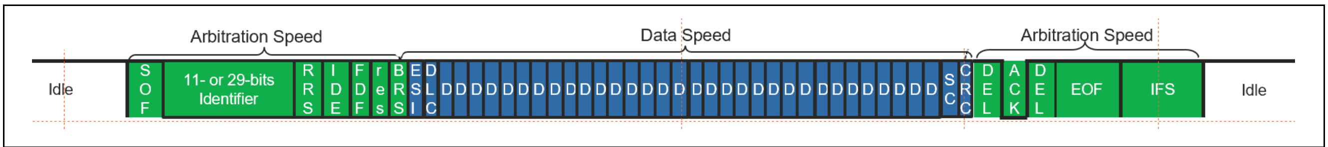


Figure 2. CAN FD Message Frame

Some of the bits in the CAN FD frame map to bits in the classical CAN frame arbitration phases for capturing and releasing the bus.

Note that, for a classical CAN node to exist on a bus with CAN FD, it must either send an error frame when it received a CAN FD message, or the peripheral must filter out CAN FD messages. This is clearly reserved for new devices that have implemented classical CAN for cost reasons but are "CAN FD aware".

1.6 CAN Messaging Support

Because of the standard definition for messages, priorities, destinations, and data, filters can be set up in digital hardware or software to allow (accept) and "stack up" messages for a particular node to process.

Various market MCUs do this differently, with varying mixes of hardware (HW), firmware (FW), and software (SW) support. In the latest Renesas RA devices, the CAN support includes CAN and CAN FD, hardware filtering for messages, and hardware management of mailboxes for both standard (prioritized) and FIFO mode. This allows for the system software to only be aware of messages for its specific node, and increases flexibility by taking time to process messages in a timely but convenient manner (obviously to the limit of the number of mailboxes, which is the same limitation of any buffering strategy).

With a rich interrupt fabric, the designer can select how aware the system software needs to be for bus activity. The peripheral also can monitor bus traffic, and work with or without TrustZone™ security support.

Hardware support alone is not enough to efficiently manage CAN bus messaging. Left alone, it would take weeks to build a driver that has all corner cases and different traffic patterns handled.

Renesas' FSP now contains stacks to support the new CAN bus on-chip peripherals, making it easier to incorporate in a system design. Renesas FSP has support for Classical CAN, CAN FD, and CAN FD Lite. That may sound complex, but the FSP helps in this case. All Renesas CAN peripherals can use the standard CAN stack in CAN mode. The advanced CAN FD peripherals differ between devices. The FSP is aware of which device supports CAN FD and CAN FD Lite – the major differences are the size and number of buffers and mailboxes.

2. Example Project

2.1 Connecting the Devices

The two boards being used for this example: EK-RA6E2 and EK-RA4E2 function as devices on a CAN/CAN FD network. Both boards have an exposed CAN-Transceiver pair available on J32 (left side of the boards just under the RA logo) and pins can be connected 1-1, 2-2, 3-3. The signals are A, B, and Ground. Connecting the CAN transceiver pins of an EK-RA6E2 and EK-RA4E2 establishes a CAN network of two nodes.

2.2 Example Overview

You may notice right away, as you work through the code in this application note in `ek_ra6e2_canfd_example_base.zip` and `ek_ra4e2_canfd_example_base.zip` that the exact same code, except for an ID or two, is used for both boards. This is because a CAN bus communication can start from any node. The resulting functionality uses the two user switches: S1 and S2, to control LEDs on both boards. Switch S2 initiates the CAN message from Transmitter Board to Receiver Boards. The LEDs shows the reception of the CAN message on received board.

From the base projects (one for each board) you will add some prepared files as a new base and add code to those files to build up the working application. A zip file called "`Example_Files.zip`" is included with the application note and the two files contained inside it will be used in one of the steps.

With each addition, a `#define` must be changed to expose some pre-written code to the project. This is done so that the code can be part of the base project and not cause compile errors before it is used. This `#define` is located in the "`user_cfg.h`" file and is called LAB_SECTION. Each step requires a change in that value – it must be changed in both projects' "`user_cfg.h`" file.

Optional: (If you are using the Git repository, the changes described in this document are already done for each commit – but it is valuable to try it yourself by branching from each commit – you can always switch over to the next commit.)

2.3 First Steps

Import the zipped base projects to your workspace (`ek_ra6e2_canfd_example_base.zip` and `ek_ra4e2_canfd_example_base.zip`). After the projects are imported to the e² studio workspace, ensure the following provided files are in the src folder for both the base projects.

- common_utils.h,
- hal_entry.c
- user_cfg.h
- user_interface.c
- user_interface.h

Generate project content, build, download and run the code on the respective boards (EK-RA6E2 and EK-RA4E2). The following behavior will be observed:

- RUN: All the user LEDs will flash and then turn off
- PRESS S1: Pressing the S1 switch on either board will make the green LED light up on that board.

Note: Based on the version of the board, if you are noticing that the green LED is always ON and does not respond to the switch, make a change to the macro "WS_VERSION" from 1 to 0 then build and test the project; the Green LED lights up and responds to the switch.

Note: It is difficult to debug two systems at the same time. The Renesas EK boards provide a method of disabling the on-board debugger so you can keep the boards connected but only have one active debugger. This is done with the J9 jumper located just below the debug USB connection on each board. If the jumper is in place (shorting both pins) the debugger is disabled. Each board ships with a shorting jumper in a "resting position" that is not shorting the pins.

2.4 Adding the CAN FD Support

The two base projects will be modified. You can go through all the steps for each project at one time, or alternate between the projects at each step and keep them both in sync. However, you will not be able to test the CAN FD code until both projects are modified, compiled, and downloaded to the respective MCU.

For each project, open the configurator and make the changes described in the following sections.

2.4.1 Adding the CAN FD Lite Stack

The light version of the stack simply supports smaller hardware buffers than the regular stack – the communications functionality is the same.

Change the value of LAB_SECTION to 3 in the "`user_cfg.h`".

On the "**Stacks**" tab, add the CAN FD Lite stack to the project as shown in the following graphic.

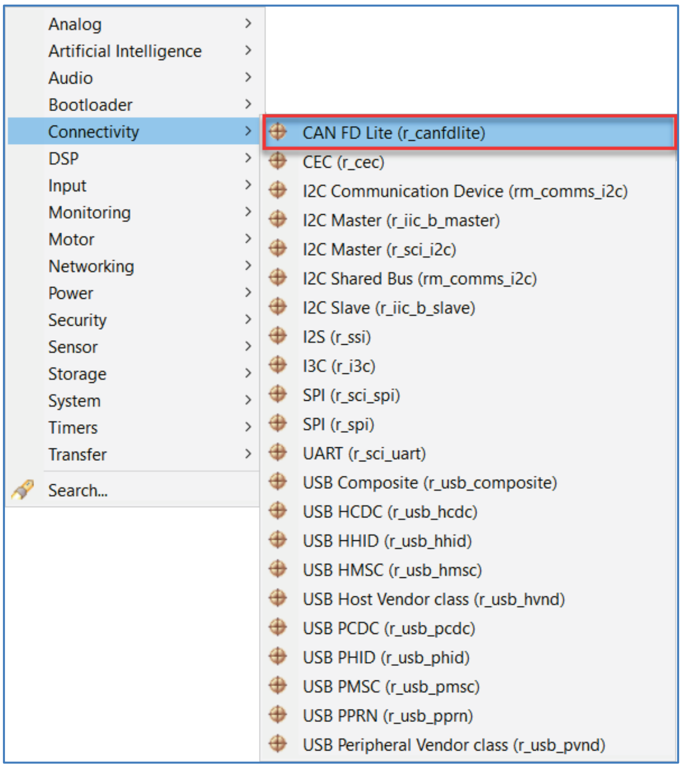


Figure 3. Adding the CAN FD Lite Stack

The CAN FD Lite stack block indicates that it needs more support – in this case, the system knows that it does not have the correct clock enabled. To do this, navigate to the **Clocks** tab and make the following changes:

- Change the PLL (Phase Locked Loops) divider to 2.
- Change the PLL multiplier to x16.
- Enable the CANFDCLK.
- Change the CANFDCLK divider to 4.

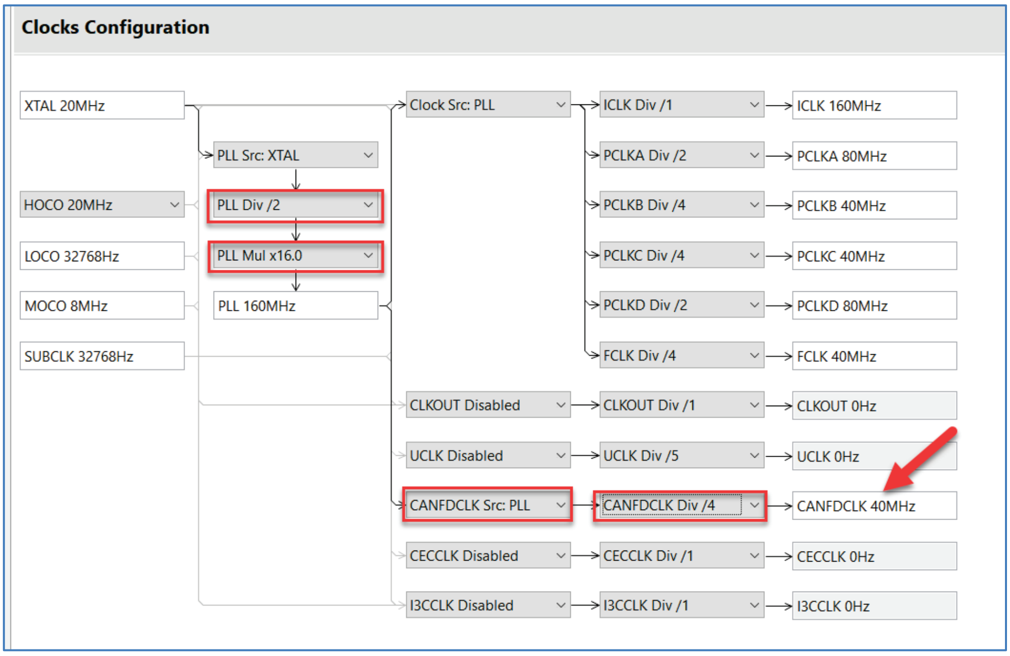


Figure 4. Clock Changes for CAN FD

Switch back to the **Stacks** tab and select **Properties** to make some changes. Change the following parameters:

- Common → Reception → Acceptance Filtering → Channel 1 Rule Count: **0**
- Module CAN FD Lite → Transmit Interrupts: **Enable All**
- Module CAN FD Lite → Reception → Message Buffer → Number of Buffer: **1**

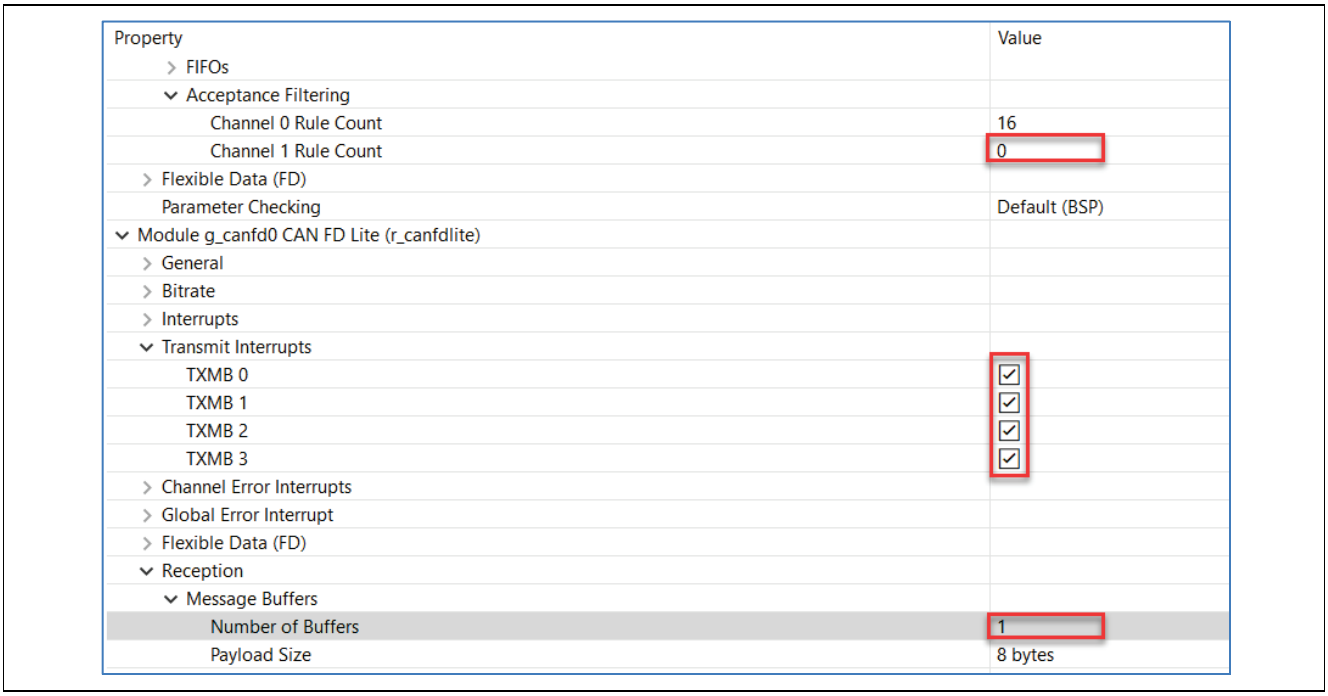


Figure 5. Changing CAN FD Lite Stack Properties

You may now press the **Generate Project Content** button to create all the required configuration files.

2.4.2 Adding code/configuration support for the stack

In the zip file “Example_Files.zip” there are two support files for this project. Copy them to the src directory:

- `Canfd.c`
- `Canfd.h`

Since the CAN protocol works off of message IDs, each button message must have its own ID. To do this, open the `user_cfg.h` file and make the following changes:

```
#define EK_RA6E2_BTN1_ID    (0x60)
```

```
#define EK_RA4E2_BTN1_ID    (0x61)
```

The data length code (DLC) must match the payload size we set in the **Properties** tab. Change the value in the same file as follows:

```
#define CAN_DATA_LENGTH 8
```

CAN controllers often provide a set of acceptance filters (AFL) that can be programmed to allow specific CAN messages with certain IDs to be accepted while rejecting others. This is useful when a node is only interested in a subset of messages on the bus. AFL block’s the ability to filter out only those messages meant for its node on the bus. This is done with the AFL properties. While these are fully documented in the FSP manual, the parameters are shown here for reference:

AFL Field	Property	Description
id	id	AFL message ID (Expected income ID)
	frame_type	• **CAN_FRAME_TYPE_DATA**: Data Frame • **CAN_FRAME_TYPE_REMOTE**: Remote Frame (Classical CAN only)
	Id_mode	• **CAN_ID_MODE_STANDARD**: Standard IDs of 11 bits used. • **CAN_ID_MODE_EXTENDED:** Extended IDs of 29 bits used.
mask	mask_id	AFL mask ID
	mask_frame_type	If this bit is 0 both data and remote frame will be accepted.
	mask_id_mode	If this bit is 0 both Standard and Extended IDs will be accepted.
destination	minimum_dlc	Minimum DLC value to accept (valid when DLC Check is enabled)
	rx_buffer	RX Message Buffer to receive messages accepted by this rule
	fifo_select_flags	RX FIFO(s) to receive messages accepted by this rule

We will set these parameters for our application, which will start as classic CAN and then be switched to FD. Open the `canfd.c` file you copied to the src folder and add the following code before the `canfd_init` function (around line 47):

```
const canfd_afl_entry_t p_canfd0_afl[CANFD_CFG_AFL_CH0_RULE_NUM] =
{
 {
     .id =
     {
         .id         = 0x60,
         .frame_type = CAN_FRAME_TYPE_DATA,
         .id_mode    = CAN_ID_MODE_STANDARD,
     },
     .mask =
     {
         .mask_id         = 0x7FE,
         .mask_frame_type = 0,
         .mask_id_mode    = 1,
     },
     .destination =
     {
         .minimum_dlc = CANFD_MINIMUM_DLC_0,
         .rx_buffer   = CANFD_RX_MB_0,
     }
 }
};
```

The `#define` definitions are located in the CAN FD Lite fsp stack and are documented in the FSP documentation – but you can right-click on the macro name and select “**Open Declaration**” to go there automatically if you are interested.

You can see the ID value is 0x60, which is the value you set in `user_cfg.h` for the S1 on the RA6 board. You will note that the mask for the ID has the last bit as 0, which indicates that the system will respond to a 0x60 or 0x61 (last bit is “don’t care”).

Now that we know what messages we want coming across the bus, we need to configure our transmission handling to send messages. We do that using the CAN frame information contained in a `can_frame_t` structure. This is also well detailed in the FSP documentation, however a quick overview is provided as follows.

can_frame_t field	Description
id	AFL message ID (Expected income ID)
id_mode	• **CAN_ID_MODE_STANDARD**: Standard IDs of 11 bits used. • **CAN_ID_MODE_EXTENDED:** Extended IDs of 29 bits used
type	• **CAN_FRAME_TYPE_DATA**: Data Frame • **CAN_FRAME_TYPE_REMOTE**: Remote Frame (Classical CAN only)
data_length_code	CAN Data Length Code (DLC)
options	This is used for CANFD frames only: • **CANFD_FRAME_OPTION_ERROR**: Error state set (ESI). • **CANFD_FRAME_OPTION_BRS**: Bit Rate Switching (BRS) enabled. • **CANFD_FRAME_OPTION_FD**: Flexible Data frame (FDF).
data	CAN frame payload

Since we are starting by transmitting a classical can message, set the parameters properly by copying the following code and placing it in the `canfd_write()` function where indicated:

```
    g_can_tx_frame.id                = ID;
    g_can_tx_frame.id_mode           = CAN_ID_MODE_STANDARD;
    g_can_tx_frame.type              = CAN_FRAME_TYPE_DATA;
    g_can_tx_frame.data_length_code  = CAN_DATA_LENGTH;
    g_can_tx_frame.options            = 0;
```

Now we can send a frame, and we can receive into the node only the frame messages we have filtered. It's time to set up the reading function to read the messages that get through the filter criteria.

Go to the `canfd_read()` function and add the following code where indicated:

```
    /* Get the status information for CAN transmission */
    err = R_CANFD_InfoGet(&g_canfd0_ctrl, &can_rx_info);
    if (err != FSP_SUCCESS)
        APP_ERR_TRAP();
    /* Check if the data is received in FIFO */
    if((can_rx_info.rx_mb_status & (1<<CANFD_MB_0)) == (1<<CANFD_MB_0))
    {
        /* Read the input frame received */
        err = R_CANFD_Read(&g_canfd0_ctrl, CANFD_MB_0, &g_can_rx_frame);
        if (err != FSP_SUCCESS)
        {
            ledState(LED_RED, ON);
            APP_ERR_TRAP();
        }


        if (msgId == g_can_rx_frame.id)
        {
            memcpy(p_data, g_can_rx_frame.data, CAN_DATA_LENGTH);
            err = FSP_SUCCESS;
        }
    }
    else
    {
        err = FSP_ERR_CAN_DATA_UNAVAILABLE;
    }
```

At this point you are ready to compile and download the code. This code runs independently once programmed into the board. For each project, ensure the debugger is enabled, build, download, and execute the code. Stop execution and repeat for the other board.

Note: It is okay to see 2 unused parameter warnings in the `canfd.c` file , for the `canfd_read_fifo()` function, after building the project at this step. This function is not yet used, and will be edited in section 4.

3. Testing the Code

Verify that on both boards CAN-Transceiver pins (J-32) are connected 1←→1, 2←→2, 3←→3.

To test the code is simple. The functionality that we introduced with the changes in section 2 creates the following functionality:

STARTUP: All LEDs flash and then go out.

PRESS S1: This time, when you press S1 switch on either board, the green light on both boards comes on.

When you press the button, it sends a message on the CAN bus to which both boards respond.

(Note: If one or both red LEDs come on, check the wiring, and press the rest buttons – this represents a condition that only happens because of startup from debug or a missed wire.)

In section 4, the modifications include an expansion in logic. To test the new code the addition of S2 switch functionality is introduced. Press either of the S2 switches, and the blue LED on the opposite board toggles on/off.

4. Modifying Standard Implementation

4.1 About the Modifications

We will modify the code in two ways. The first is to add FIFO support and the second is to add additional functionality. We will add S2 switch support which will toggle the blue LED on the *other* board: press once and it goes on, press again and it goes off.

Again, we will make the following changes to both projects.

Before we go any further, change the LAB_SECTION macro in `user_cfg.h` to 4.

4.2 Add FIFO Support to CANFD Lite

First, we will add support for FIFO. Open the configurator and disable pin 304:

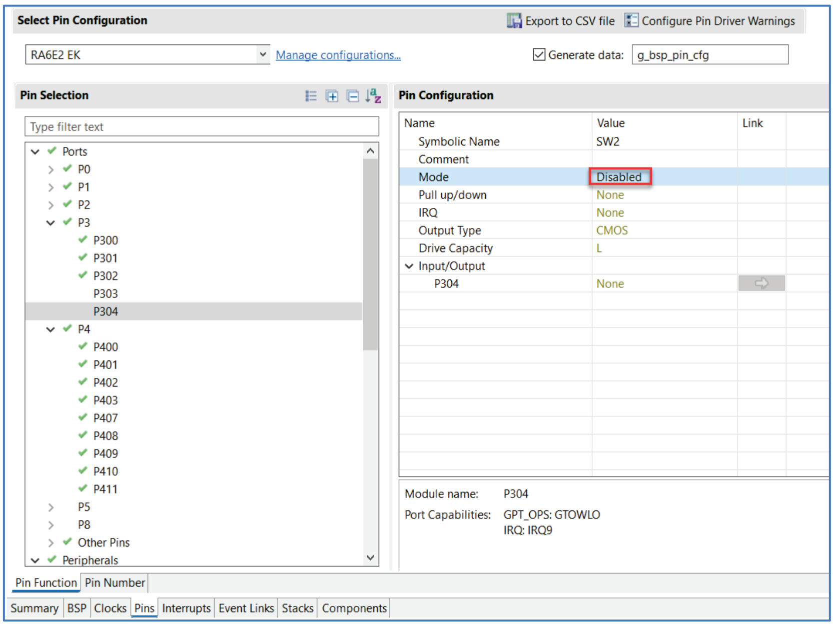


Figure 6. Pin Selection for IRQ

Go to **Peripherals → Interrupt → IRQ**, then set the Operation Mode to **Custom**, set IRQ9 to **P304.**

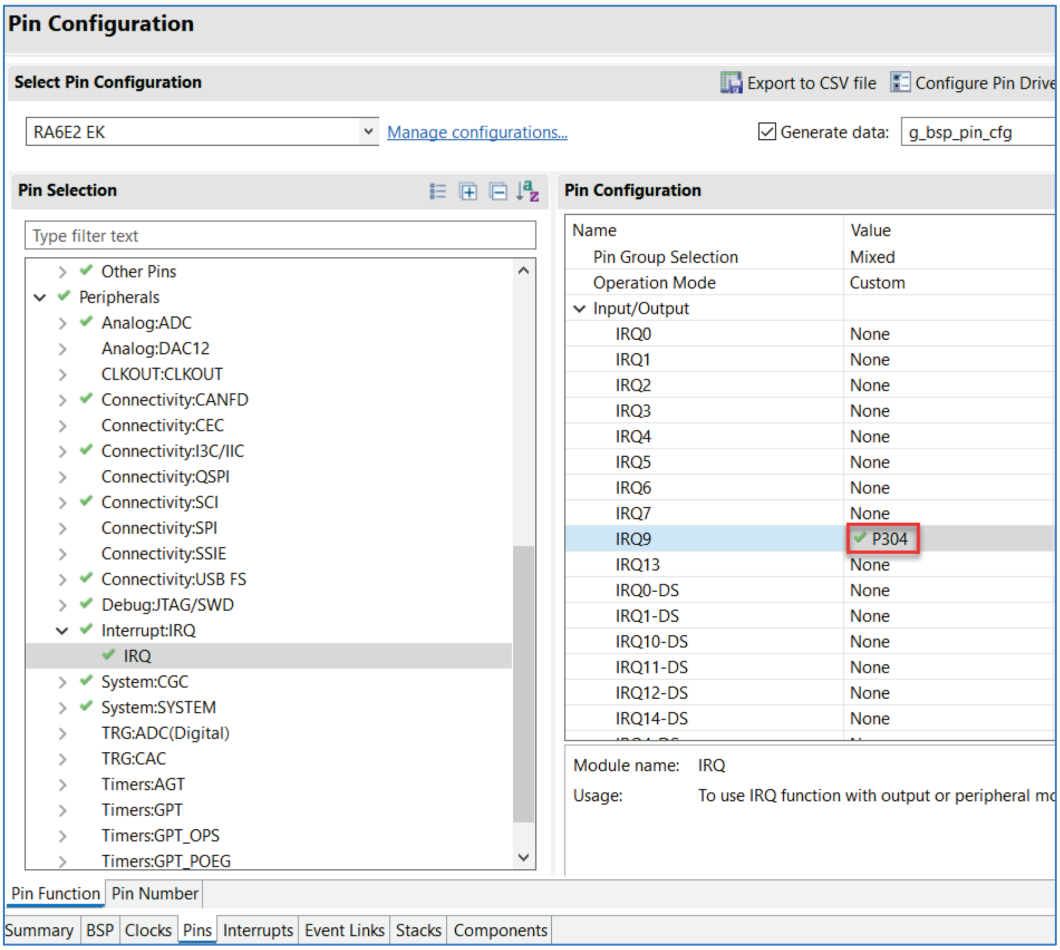


Figure 7. Configuring Port Pin to IRQ

Switch to the **Stacks** tab and add the Input → External IRQ stack:

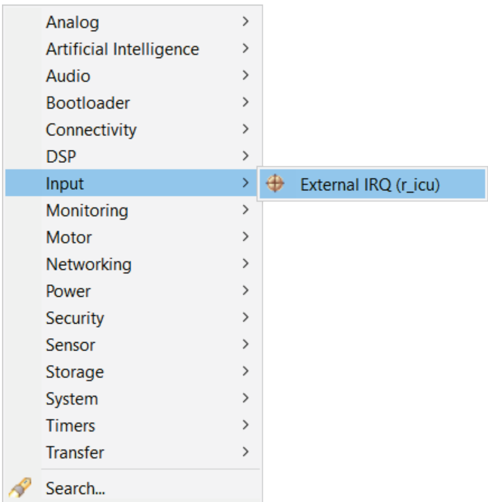
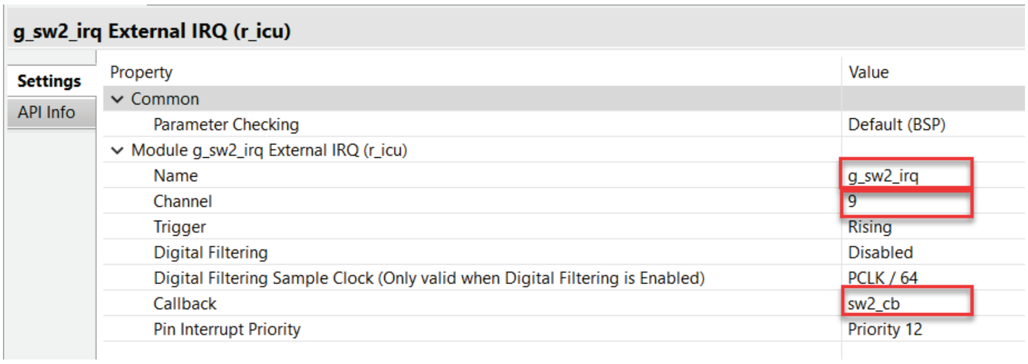


Figure 8. Adding External IRQ

And set the following properties:

- Name: **g_sw2_irq**
- Channel: **9**
- Callback: **sw2_cb**



g_sw2_irq External IRQ (r_icu)

Settings
API Info

Property	Value
∨ Common	
Parameter Checking	Default (BSP)
∨ Module g_sw2_irq External IRQ (r_icu)	
Name	g_sw2_irq
Channel	9
Trigger	Rising
Digital Filtering	Disabled
Digital Filtering Sample Clock (Only valid when Digital Filtering is Enabled)	PCLK / 64
Callback	sw2_cb
Pin Interrupt Priority	Priority 12

Figure 9. Setting up External IRQ for SW2

You can now generate project content. Repeat for the second board's project.

We need to define message codes for the S2 switches (one on each board) as we did for S1. Open the `user_cfg.h` file in both projects and modify the following lines as shown:

```
#define EK_RA6E2_BTN2_ID    (0x40)
#define EK_RA4E2_BTN2_ID    (0x41)
```

Now we will add some code support for the S2 switch. We need to add to our filter criteria, so open the `canfd.c` file and add a second entry to the AFL structure you created in section 2. Paste the following lines of code inside the outer curly braces, and under the first record: (Reference `p_canfd0_afl around line 67`).

```
,
{
     .id =
     {
         .id         = 0x40,
         .frame_type = CAN_FRAME_TYPE_DATA,
         .id_mode    = CAN_ID_MODE_STANDARD,
     },
     .mask =
     {
         .mask_id         = 0x7F0,
         .mask_frame_type = 0,
         .mask_id_mode    = 1,
     },
     .destination =
     {
         .minimum_dlc = CANFD_MINIMUM_DLC_0,
         .fifo_select_flags = CANFD_RX_FIFO_0
     }
 }
```

(Note: you may see some context highlighting indicating an error. That means you did not put an element separator between the records. Add a comma "," between the records (after the close curly brace above the newly pasted code).

Once the message gets through the filter and into the FIFO, we must retrieve it. This is done in a callback function. Add the following code to the `canfd0_callback()` function:

```
            /* Read received frame */
             memcpy(&g_can_rx_frame_fifo, &p_args->frame, sizeof(can_frame_t));
             g_can_rxfifo_flag = 1;
```

and the following code to the `canfd_read_fifo()` function:

```
    if (msgId == g_can_rx_frame_fifo.id)
    {
        memcpy(p_data, g_can_rx_frame_fifo.data,CAN_DATA_LENGTH);
        err = FSP_SUCCESS;
    }
```

Now compile the code for both boards, download, run, end execution. Make sure to test the code for the modified functionality similar to as described in section 3 above.

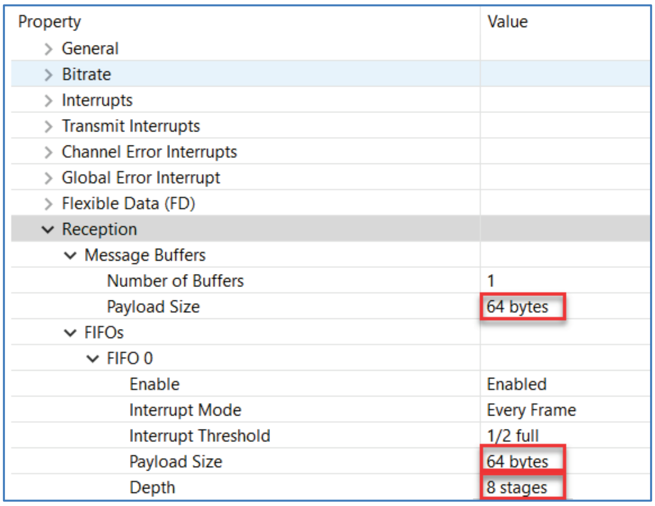
4.3 Switch CAN to CAN FD Transmission

In this section we will make some very small changes to settings, verify one piece of auto-generated code, and the system will easily switch from CAN to CAN FD transmission. No processing code changes, no additional stack inclusion.

As always, first we change our LAB_SECTION #define. In this case, change the value to 5. And, as always, we will change both projects in the same way.

Let's change the appropriate parameters in our existing stack. Open the configurator, select the CAN FD Lite stack, and change the following parameters:

- Module CAN FD Lite → Reception → Message Buffer → Payload Size: 64 Bytes
- Module CAN FD Lite → Reception → FIFO → FIFO 0 → Payload Size: 64 Bytes
- Module CAN FD Lite → Reception → FIFO → FIFO 0 → Depth: 8 Stages



Property	Value
General	
Bitrate	
Interrupts	
Transmit Interrupts	
Channel Error Interrupts	
Global Error Interrupt	
Flexible Data (FD)	
Reception	
Message Buffers	
Number of Buffers	1
Payload Size	64 bytes
FIFOs	
FIFO 0	
Enable	Enabled
Interrupt Mode	Every Frame
Interrupt Threshold	1/2 full
Payload Size	64 bytes
Depth	8 stages

Figure 10. Changing from CAN to CAN FD

Generate New Project Content.

To verify the configuration changes, open `ra_gen/hal_data.c` and check to see if the CAN bit timing variables match what we have selected in the configurator. There is a variable declared at the top of the file of type `can_bit_timing_cfg_t` and the FD and non-FD timing parameters should match what we have just changed.

The Data Length Code (DLC) must match what was set for the data buffer payload size. In `user_cfg.h` change the CAN_DATA_LENGTH #define from 8 to 64.

Now we need to tell the code to use CAN FD instead of classical CAN. There are three bits that control this, and they are defined in the stack header files:

- **CANFD_FRAME_OPTION_BRS** = Bit Rate Switching (BRS) enabled.
- **CANFD_FRAME_OPTION_FD** = Flexible Data frame (FDF).

Open the `canfd.c` and change the TX frame parameters in the `canfd_write` function (reference line 108):

```
g_can_tx_frame.id               = ID;
g_can_tx_frame.id_mode          = CAN_ID_MODE_STANDARD;
g_can_tx_frame.type             = CAN_FRAME_TYPE_DATA;
g_can_tx_frame.data_length_code = CAN_DATA_LENGTH;
g_can_tx_frame.options    = CANFD_FRAME_OPTION_BRS | CANFD_FRAME_OPTION_FD;
```

Build, download, run, and execute on both boards. Then test the functional changes made in section 4.2 above.

On the scope, the difference between CAN and CAN FD frames should look like those shown in the figures that follow.

Users can see the difference between the CAN and CAN FD frame in terms of data payloads 8 bytes vs 64 bytes.

Figure 11. CAN Frame

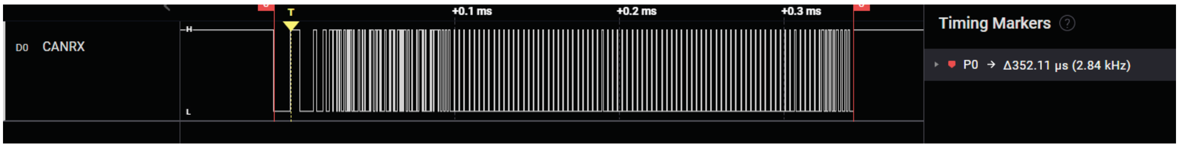


Figure 12. CAN FD Frame

CAN FD provides an increase in data capacity, allowing for larger data payloads, which is particularly beneficial in applications which requires higher data rates. CAN FD also maintains backward compatibility with classical CAN, allowing for a smooth transition in networks where both frame types coexist.

5. API Details

There are four common APIs used to control operation of the CANFD Lite module:

- **Open** - used to “open” the CAN_FD interface (initialize the stack).
- **Read** - used to read data sent from another CAN FD device.
- **Write** - used to write to a CAN FD device.
- **Mode Transition** - used to switch CAN FD channels, or to enter test mode.

These APIs are outlined in the FSP documentation found here: [RA Flexible Software Package Documentation: Introduction (renesas.github.io)]() but we highlight them in this document for your convenience.

Each of the APIs has a return value. The value of FSP_SUCCESS for any of them means the function call ended successfully. Each of the calls has their own set of return error values.

Here is the outline of each of the four APIs:

5.1 Open

```
fsp_err_t R_CANFD_Open (can_ctrl_t *const p_api_ctrl,
                              can_cfg_t const *const p_cfg)
```

For inputs this API takes two structures that define the interface.

Example:

```
/* Initialize the CAN module */
err = R_CANFD_Open(&g_canfd0_ctrl, &g_canfd0_cfg);
```

5.2 Read

```
fsp_err_t R_CANFD_Read (can_ctrl_t *const p_api_ctrl,
                        uint32_t buffer,
                        can_frame_t *const p_frame)
```

For inputs this API takes two structures that define the interface and the frame, and a buffer.

Example:

```
/* Read the input frame received */
err = R_CANFD_Read(&g_canfd0_ctrl, CANFD_MB_0, &g_can_rx_frame);
```

5.3 Write

```
fsp_err_t R_CANFD_Write (can_ctrl_t *const p_api_ctrl,
                        uint32_t buffer,
                        can_frame_t *const p_frame)
```

For inputs this API takes two structures that define the interface and the frame, and a buffer.

Example:

```
/* Send data on the bus */
err = R_CANFD_Write(&g_canfd0_ctrl, CANFD_TX_MB_0, &g_can_tx_frame);

assert(FSP_SUCCESS == err);
```

5.4 Mode Transition

```
fsp_err_t R_CANFD_ModeTransition (can_ctrl_t *const p_api_ctrl,
                                  can_operation_mode_t operation_mode,
                                  can_test_mode_t test_mode)
```

For inputs this API takes a control structure and two enumerations: one for operating mode, and to indicate if it's in a test mode, and if so which test.

Example:

```
/* Switch to external loopback mode */
err = R_CANFD_ModeTransition(&g_canfd0_ctrl, CAN_OPERATION_MODE_NORMAL,
CAN_TEST_MODE_LOOPBACK_EXTERNAL);

assert(FSP_SUCCESS == err);
```

6. Equations

While there are no theoretical equations for calculation per-se, the classical CAN frame time with 8 data bytes at 500 kB is 87 μs and CAN FD transmitting 64 bytes with 500 kB/2 MB takes 352.11 μs, which is 8x the data with approximately 4x the time.

We can use this information, since the data is being transmitted at a faster speed, to estimate the time it would take to transmit different size blocks of data.

In the classical CAN packet, it takes about 2/3 of the frame to hold the data, so 2/3 of the time is overhead that will be at that speed even in CAN FD. Then, the data time difference is basically the data size difference divided by the clock difference between the slow and fast clocks (which as we have seen are both settable in the FSP).

7. References

- FSP CAN Documentation: RA Flexible Software Package Documentation: CAN Interface (renesas.github.io)
- EK-RA6E2 Getting Started Guide: https://www.renesas.com/us/en/document/qsg/ek-ra6e2-quick-start-guide
- EK-RA4E2 Getting Started Guide: https://www.renesas.com/us/en/document/qsg/ek-ra4e2-quick-start-guide

8. Website and Support

Visit the following URLs to learn about key elements of the RA family, download components and related documentation, and get support:

RA Product Information	renesas.com/ra
RA Product Support Forum	renesas.com/ra/forum
RA Flexible Software Package	renesas.com/FSP
Renesas Support	renesas.com/support

Revision History

Rev.	Date	Description	
		Page	Summary
1.00	Mar.11.24	—	Initial release

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

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

1. Precaution against Electrostatic Discharge (ESD)

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

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

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

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

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

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

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

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

Corporate Headquarters

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

Trademarks

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

Contact information

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