F^{2}MC-16FX Family
16-BIT MICROCONTROLLER
MB96F356

bits pot white CAN-LIN board

User's Manual

Revision History

Date	Revision
November 13,2008	Revision 1.0: Initial release
May 13, 2009	Revision 1.1 TSUZUKI DENSAN's Logo mark was changed.
April 23, 2010	Revision 1.2 -Change in installation procedure and execution procedure by Euroscope upgrade. -The description of the PC specifications in Table 1-1 is corrected. -Change in company name of FUJITSU MICROELECTORONICS [New]FUJITSU SEMICONDUCTOR LIMITED
	(left blank)

Note

- The contents of this document are subject to change without notice. Customers are advised to consult with FUJITSU sales representatives before ordering.
- The information, such as descriptions of function and application circuit examples, in this document are presented solely for the purpose of reference to show examples of operations and uses of Fujitsu semiconductor device; Fujitsu does not warrant proper operation of the device with respect to use based on such information. When you develop equipment incorporating the device based on such information, you must assume any responsibility arising out of such use of the information. Fujitsu assumes no liability for any damages whatsoever arising out of the use of the information.
- Any information in this document, including descriptions of function and schematic diagrams, shall not be construed as license of the use or exercise of any intellectual property right, such as patent right or copyright, or any other right of Fujitsu or any third party or does Fujitsu warrant non-infringement of any third-party's intellectual property right or other right by using such information. Fujitsu assumes no liability for any infringement of the intellectual property rights or other rights of third parties which would result from the use of information contained herein.
- The products described in this document are designed, developed and manufactured as contemplated for general use, including without limitation, ordinary industrial use, general office use, personal use, and household use, but are not designed, developed and manufactured as contemplated (1) for use accompanying fatal risks or dangers that, unless extremely high safety is secured, could have a serious effect to the public, and could lead directly to death, personal injury, severe physical damage or other loss (i.e., nuclear reaction control in nuclear facility, aircraft flight control, air traffic control, mass transport control, medical life support system, missile launch control in weapon system), or (2) for use requiring extremely high reliability (i.e., submersible repeater and artificial satellite).
Please note that Fujitsu will not be liable against you and/or any third party for any claims or damages arising in connection with above-mentioned uses of the products.
- Any semiconductor devices have an inherent chance of failure. You must protect against injury, fire, damage or loss from such failures by incorporating safety design measures into your facility and equipment such as redundancy, fire protection, and prevention of over-current levels and other abnormal operating conditions.
- If any products described in this document represent goods or technologies subject to certain restrictions on export under the Foreign Exchange and Foreign Trade Law of Japan, the prior authorization by Japanese government will be required for export of those products from Japan.
- The company names and brand names herein are the trademarks or registered trademarks of their respective owners.

Table of Contents

List of Figures

List of Tables

Introduction

Thank you very much for purchasing the bits pot white (referred to as this starter kit or the starter kit hereafter).

This starter kit is a beginner's kit intended for those who wish to start learning microcontrollers and on-board network processors. The kit is designed so that the beginners who ask "What is a microcontroller?", "How does it work?" and "How does it control a network?" can easily learn what it is.

The kit includes flash microcontroller development tools, so if you have slight understanding about the C language, you can rewrite a program to let the microcontroller perform in various ways. Even if you do not know of programming, you may be able to enjoy learning a microcontroller with a study-aid book about the C language.

This starter kit can also serve as an introductory training tool for electronic circuit practice or future embedded software development in a class of a college or high school of technology or training for freshman engineers of a manufacturer.

Contact

Please ask the following e-mail address for the technical question.

Please confirm HP for the latest information and FAQ of bits pot.

Zip code: 105-8420 2-5-3 Nishi-Shinbashi, Minatoku, Tokyo

E-mail: pd-bitspot@tsuzuki-densan.co.jp

bits pot URL: http://www.tsuzuki-densan.co.jp/bitspot/

Suppliers of the parts/materials

Capacitors	22pF:	GCM1552C1H220JZ02
	1nF:	GCM155R11H102KA01
	1µF:	GCM21BR11E105KA42
	0.1µF:	GCM188R11E104KA42
	4.7µF:	GCM31CR71E475KA40
Ceramic Resonator	4MHz:	CSTCR4M00G15C
Buzzer:		PKLCS1212E40A1

NTC Thermistors:	NTCG164BH103JT1
Ferrite Beads:	MPZ2012S300AT
Common Mode Filters	ZJYS81R5-2P24T-G01

1 Setting up the starter kit

Before using this starter kit, be sure to check the components listed in Table 1-1 are fully supplied.

Before connecting the bits pot white (referred to as the board hereafter), you need to install software in your PC. You can download the software required for the starter kit from our web site.

bits pot URL: http://www.tsuzuki-densan.co.jp/bitspot/

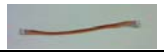
No.	Article	Qty.	Specifications	Remarks
1	Board 	1	Board mounted with a Fujitsu Semiconductor MB96F356 F^2MC-16FX series microcontroller	See Figure 1-1
2	USB cable 	1	USB（A to miniB)	Accessory
3	CAN cable 	1	3-pin cable	Accessory
4	LIN cable 	1	2-pin cable	Accessory
5	PC	1	On which Windows XP normally runs and USB2.0 ports are supported. Approximately 200 MBytes free hard disk space is required.	Prepare the PC by yourself.

Table 1-1 Component list

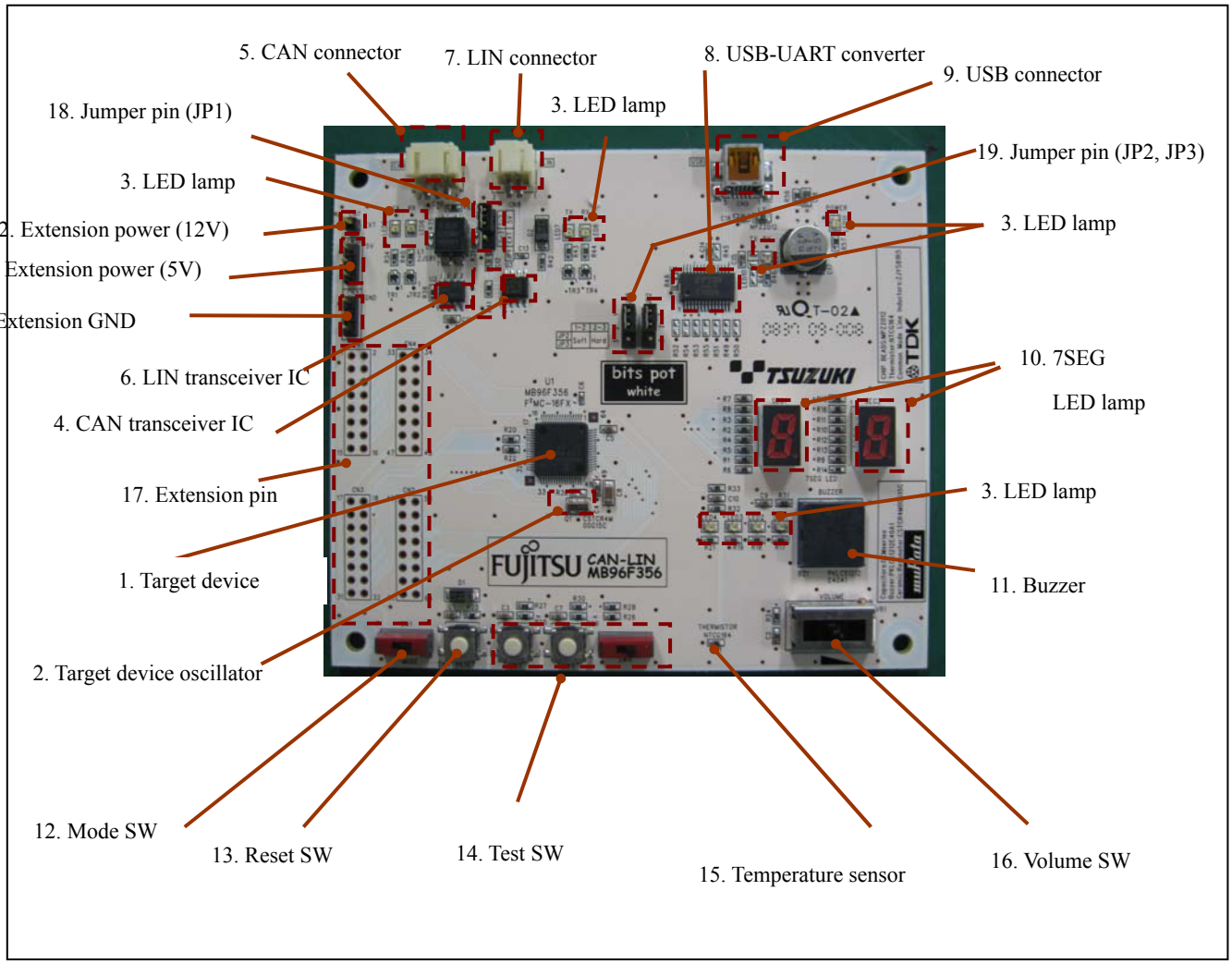


Figure 1-1 External board view

"Table 1-2 List of board parts" shows the list of parts that make up this board.

No.	Name	Function	Description
1	Target device	MB96F356	Main microcontroller (MB96F356).
2	Target device oscillator	CSTCR4M00G15C (4MHz)	Ceralock made by Murata Manufacturing Oscillator for the main microcontroller.
3	LED lamps	LED (red) x 10	LED lamps connected to the general-purpose I/O pins.

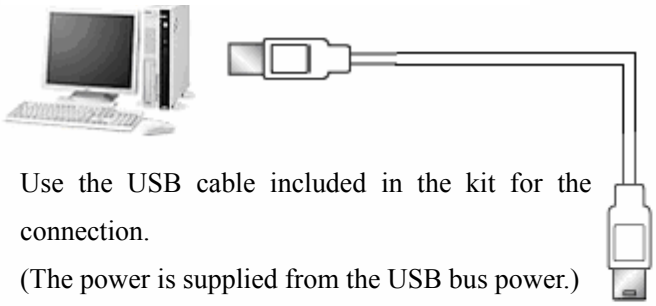
4	CAN transceiver IC	MAX3058ASA+	Transceiver IC for CAN communication.
5	CAN connector	3-pin connector	Connector for CAN communication. Connect this connector to the CAN connector on the bits pot red.
6	LIN transceiver IC	TJA1020T	Transceiver IC for LIN communication.
7	LIN connector	2-pin connector	Connector for LIN communication. Connect this connector to the LIN connector on the bits pot yellow .
8	USB to UART converter	FT232RL	IC for conversion between UART and USB.
9	USB connector	miniB	USB connector for connection with the PC to write/debug a program.
10	7SEG LED	7SEG LED x 2	7SEG LEDs connected to general-purpose I/O pins.
11	Buzzer	PKLCS1212E40A1	External-drive electric sounder made by Murata Manufacturing. Connected to the PPG timer output pin.
12	Mode SW	Slide switch	Switches the operation mode of the main microcontroller (MB96F356).
13	Reset SW	Push switch	Switch to reset the starter kit.
14	Test SW	Push switch × 2 Slide switch × 1	Push switches and slide switches connected to general-purpose I/O pins for testing purpose.
15	Temperature sensor	NTCG164BH103	NTC thermistor made by TDK Temperature sensor connected to the A/D converter.
16	Volume SW	Volume SW	Volume SW connected to the A/D converter input.
17	Extension pins	—	Extension pins of the main microcontroller. For details, see the circuit diagram.
18	Jumper pin (JP1)	—	Jumper pins for switching the LIN transceiver IC power supply. 1-2: Supplied by USB bus power (5V) 2-3: Supplied from external power supply (CN7) (12V) The default is 1-2.
19	Jumper pins (JP2, JP3)	—	Jumper pins for USB-UART conversion setting. UART communication handshake setting. 1-2: Handshake by software. 2-3: Handshake by hardware.

			The default setting is 1-2 (common to JP2/JP3).
20	Extension power (5V)	—	Extension 5V power terminal.
21	Extension GND	—	Extension GND terminal.
22	LIN transceiver IC extension power (12V)	—	Extension power pin for the LIN transceiver IC. This is used to supply power (12V) from an external source. When used, jumper pin (JP1) is required to be set to 2-3.

Table 1-2 List of board parts

"Figure 1-2 System connection diagram" shows the connection of the system for single-unit operation.

* Prepare the PC by yourself.



Use the USB cable included in the kit for the connection.
(The power is supplied from the USB bus power.)

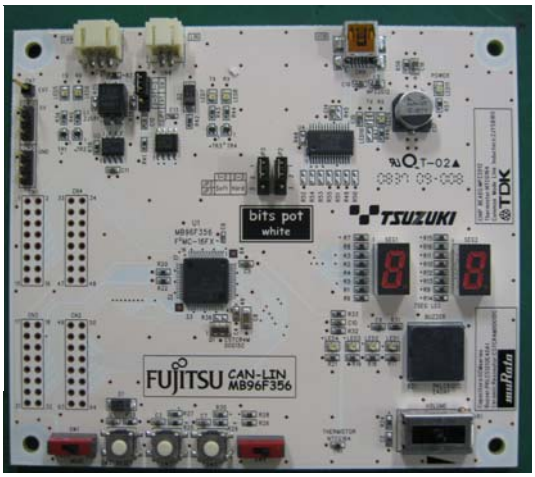


Figure 1-2 System connection diagram

Connect the PC with the board by using the USB cable included in the kit.
The power supply for the board is supplied from USB. (USB bus power)

[Note]
Connect the USB directly to the PC. Do not connect the USB via an extension unit such as a docking station, or via a USB hub.

"Figure 1-3 System connection diagram (when performing CAN communication or LIN communication)" shows the connection of the system for CAN communication and LIN communication. (Note: When performing CAN communication or LIN communication, the bits pot red or bits pot yellow, respectively, need to be purchased separately. Refer to each of the manuals for the settings of the bits pot red or bits pot yellow when performing communication.)

* Prepare the PC by yourself.

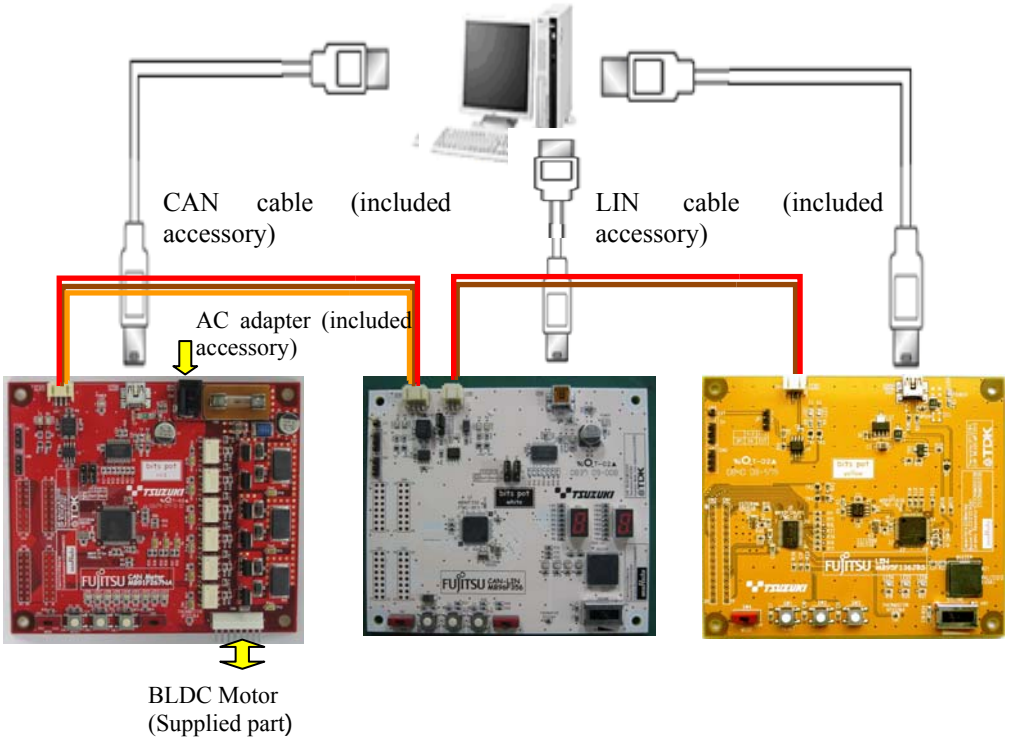


Figure 1-3 System connection diagram (when performing CAN communication or LIN communication)

Connect the board to the PC using the supplied USB cable, and connect the bits pot red (CAN communication) or bits pot yellow (LIN communication) to the board using the dedicated cables. The power for the bits pot red and bits pot yellow is also supplied from USB, the same as this board. (USB bus power)

"Table 1-3 MB96F356 pin assignment" shows the pin assignment of the main microcontroller MB96F356.

Table 1-3 MB96F356 pin assignment

Pin-No.	Description	Connected to	Remarks
1	AVss	GND	—
2	AVRH	GND	—
3	P06_2/AN2/PPG2	TH1	—
4	P06_3/AN3/PPG3		—
5	P06_4/AN4/PPG4		—
6	P06_5/AN5/PPG5		—
7	P06_6/AN6/PPG6	JP2	—
8	P06_7/AN7/PPG7	JP3	—
9	P05_0/AN8/SIN2/INT3_R1	FT232RL(TXD)	—
10	P05_1/AN9/SOT2	FT232RL(RXD)	—
11	P05_2/AN10/SCK2		—
12	P05_3/AN11/TIN3		—
13	P05_4/AN12/TOT3/INT2_R	SW5	SW pressed=L
14	P05_5/AN13/INT0_R/NMI_R	SW3	SW pressed=L
15	P05_6/AN14/INT4_R	SW4	—
16	P04_2/IN6/RX1/INT9_R/TTG6/TTG14	MAX3058(RXD)	—
17	P04_3/IN7/TX1/TTG7/TTG15	MAX3058(TXD)	—
18	Vss	GND	—
19	X0A/P04_0		—
20	X1A/P04_1		—
21	MD2	GND	—
22	MD1	PULL-UP	—
23	MD0	SW1	—
24	P00_0/AD00/INT8	SEG1	H output = On
25	P00_1/AD01/INT9	SEG1	H output = On
26	P00_2/AD02/INT10	SEG1	H output = On
27	P00_3/AD03/INT11	SEG1	H output = On
28	P00_4/AD04/INT12	SEG1	H output = On
29	P00_5/AD05/INT13	SEG1	H output = On
30	P00_6/AD06/INT14	SEG1	H output = On

31	P00_7/AD07/INT15	SEG1	H output = On
32	P01_0/AD08/CKOT1/TIN1		—
33	P01_1/AD09/CKOTX1/TOT1		—
34	P01_2/AD10/INT11_R/SIN3	TJA1020T(RXD)	—
35	P01_3/AD11/SOT3	TJA1020T(TXD)	—
36	P01_4/AD12/SCK3	TJA1020T(NSLP)	—
37	P01_5/AD13/SIN2_R/INT7_R		—
38	P01_6/AD14/SOT2_R		—
39	P01_7/AD15/SCK2_R		—
40	P02_0/A16/PPG12	LED4	L output = On
41	P02_1/A17/PPG13	LED3	L output = On
42	P02_2/A18/PPG14	LED2	L output = On
43	P02_3/A19/PPG15	LED1	L output = On
44	P02_4/A20/TTG8/TTG0/IN0		—
45	RSTX	RESET	L input = Reset
46	X1	Q1	4 MHz oscillator
47	X0	Q1	4 MHz oscillator
48	Vss	GND	—
49	Vcc	5V	—
50	C	GND	—
51	P02_5/A21/TTG9/TTG1/IN1/ADTG_R		—
52	P04_4/SDA0/FRCK0		—
53	P04_5/SCL0/FRCK1		—
54	P03_0/ALE/IN4/TTG4/TTG12	SEG2	H output = On
55	P03_1/RDX/IN5/TTG5/TTG13	SEG2	H output = On
56	P03_2/WRLX/WRX/RX2/INT10_R	SEG2	H output = On
57	P03_3/TX2/WRHX	SEG2	H output = On
58	P03_4/HRQ/OUT4	SEG2	H output = On
59	P03_5/HAKX/OUT5	SEG2	H output = On
60	P03_6/RDY/OUT6	SEG2	H output = On
61	P03_7/ECLK/OUT7	SEG2	H output = On
62	P06_0/AN0/PPG0	VR1	Power supply voltage division 0 to 100%
63	P06_1/AN1/PPG1	BZ1	—
64	AVcc	5V	—

1.1 Setting up the PC

Install the software required to operate this starter kit into the PC.

To set up the PC, take the following procedures.

① Downloading the software (refer to Section 1.1.1)
② Installing the USB driver (refer to Section 1.1.2)
③ Installing the integrated development environment SOFTUNE (bits pot white dedicated version) (refer to 1.1.3)
④ Installing the PC Writer FUJITSU FLASH MCU Programmer (bits pot white dedicated version) (refer to 1.1.4)
⑤ Installing EUROScope (refer to 1.1.5)
⑥ Configuring the board and connecting it to the PC (refer to 1.1.6)

1.1.1 Downloading the software

Download the file from the following website, and decompress it.

bits pot URL：http://www.tsuzuki-densan.co.jp/bitspot/

1.1.2 Installing the USB driver

① Install the USB driver.

Download the driver that matches your OS from the following FTDI website.

http://www.ftdichip.com/Drivers/D2XX.htm

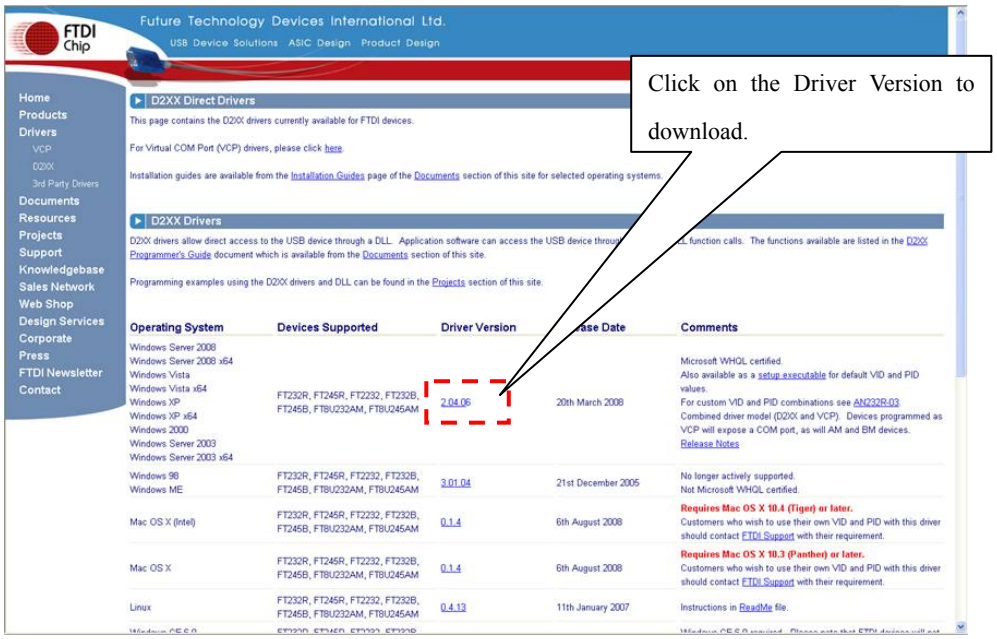


Figure 1-4 Downloading the USB driver

②After downloading the driver, decompress it, and then connect the board to the PC by using the USB cable included in the kit. As shown in "Figure 1-5 Installing FT232R USB UART" the dialog for “FT232R USB UART” installation is displayed; select “Install from a list or specific location”, and then click the “Next” button.

Figure 1-5 Installing FT232R USB UART

③ As shown in “Figure 1-6 Selecting the search locations”, to search for the installation file, check “Search for the best driver in these locations” and “Include this location in the search” only, select the location at which the driver was decompressed, and then click the “Next” button; installation of the driver starts.

Figure 1-6 Selecting the search locations

④When the driver installation ends, the dialog shown in “Figure 1-7 Completing the USB Serial Converter”, is displayed; click the “Finish” button.

Figure 1-7 Completing the USB Serial Converter

⑤After that, as shown in “Figure 1-8 Installing USB Serial Port”, installation of “USB Serial Port” is indicated; select “Install from a list or specific location” and then click the “Next” button.

Figure 1-8 Installing USB Serial Port

⑥As shown in “Figure 1-9 Selecting the search locations”, to search for the installation file, check “Search for the best driver in these locations” and “Include this location in the search” only, select the location at which the driver was decompressed, and then click the “Next” button; installation of the driver starts.

Figure 1-9 Selecting the search locations

⑦When the driver installation ends, the dialog shown in “Figure 1-10 Installing USB Serial Port” is displayed; Click the “Finish” button.

Figure 1-10 Installing USB Serial Port

1.1.3 Installing the integrated development environment SOFTUNE (bits pot white dedicated version)

Note

If SOFTUNE V3 of the product version has been installed, first uninstall it, and then install the bits pot white dedicated version.

Start installing the integrated development environment SOFTUNE. Decompress the following file in the folder you decompressed in Section 1.1.1, “Downloading the software”:

¥softwares¥softune¥ REV300021-BV.zip

① Double-click “setup.exe” from among the decompressed files to begin the installation.

Figure 1-11 Installer

② Perform the installation by following the on-screen directions. Click the “OK” button.

Figure 1-12 SOFTUNE setup confirmation

③ Click the “Next” button.

Figure 1-13 Starting SOFTUNE setup

⑤ Click the “Next” button.

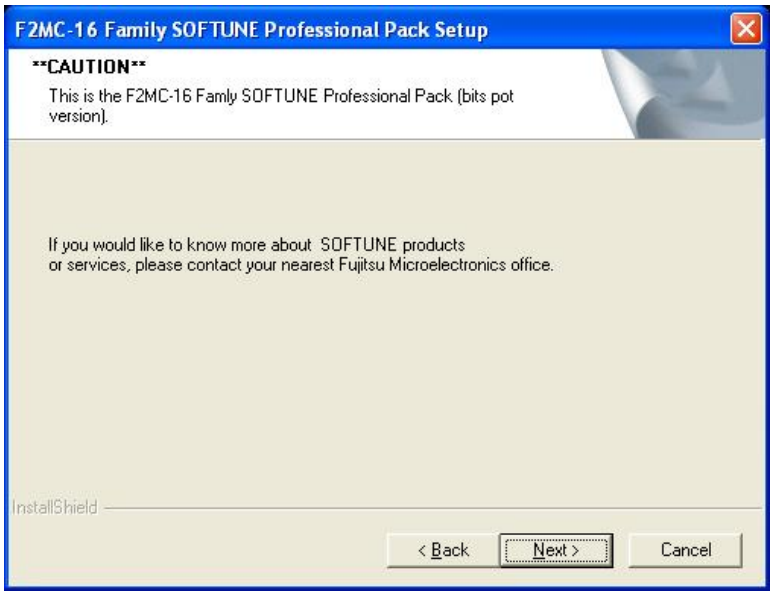


Figure 1-14 Caution on SOFTUNE setup

⑥ If you agree, click the “Yes” button.

(If you do not agree, you cannot use the software.)

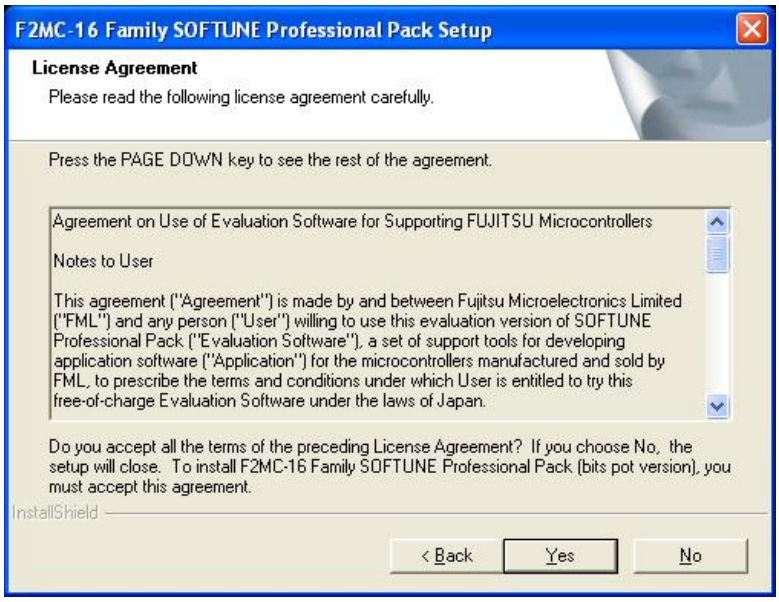


Figure 1-15 SOFTUNE setup/License agreement

⑦ Click the “Next” button.

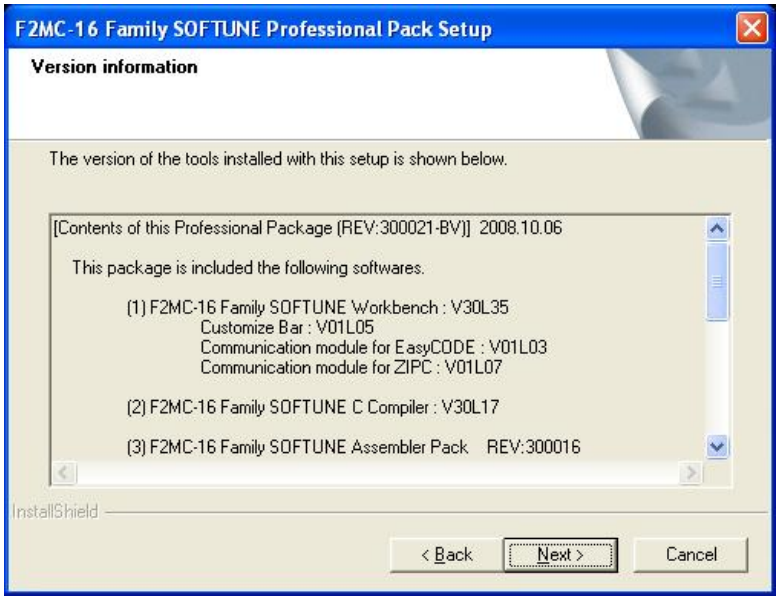


Figure 1-16 SOFTUNE setup/Version information

⑧ The dialog about the destination of installation appears; select the default folder or desired folder and then click the “Next” button.

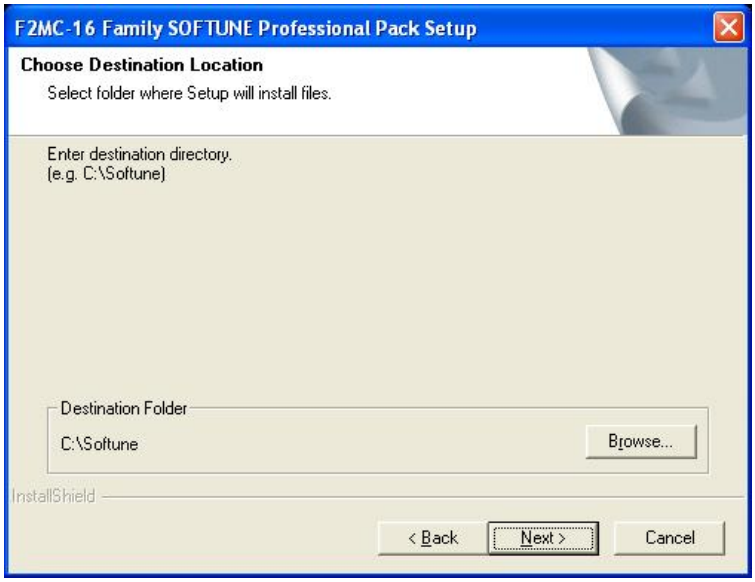


Figure 1-17 SOFTUNE setup/Selecting the destination of installation

⑨ Keep the default settings and then click the “Next” button.

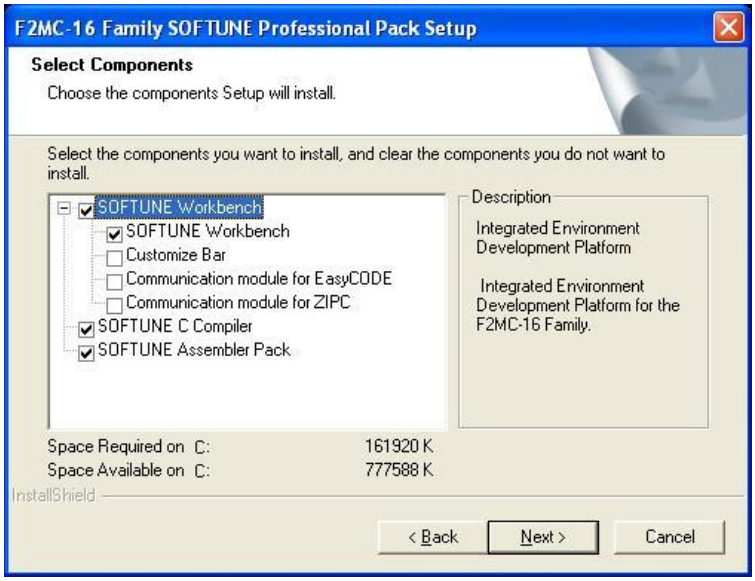


Figure 1-18 SOFTUNE setup/Selecting the components

⑩ Click the “Next” button.

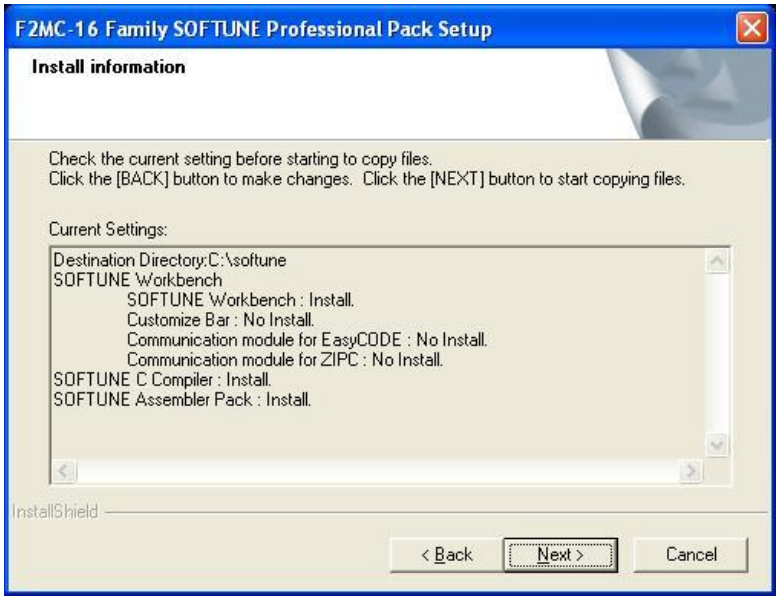


Figure 1-19 SOFTUNE setup/Confirming the installation settings

⑪ The installation is performed.

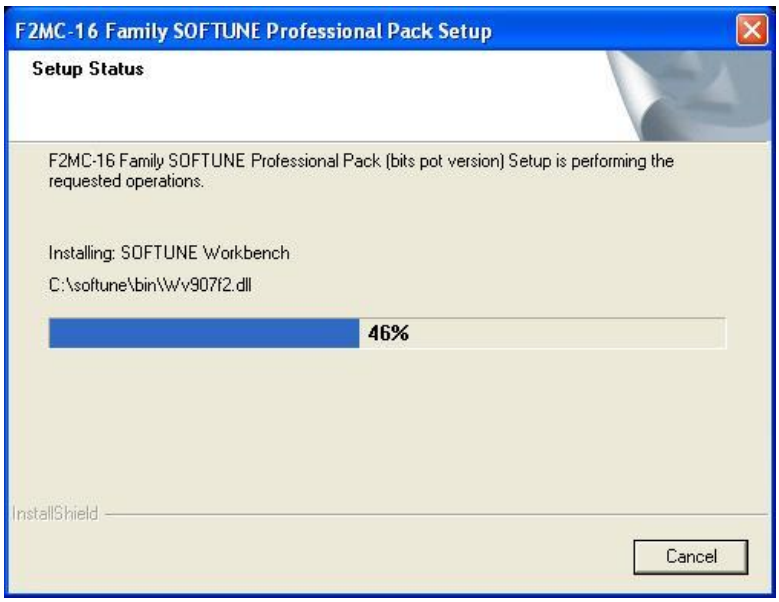


Figure 1-20 SOFTUNE setup/Status

⑫ Click the “Finish” button.

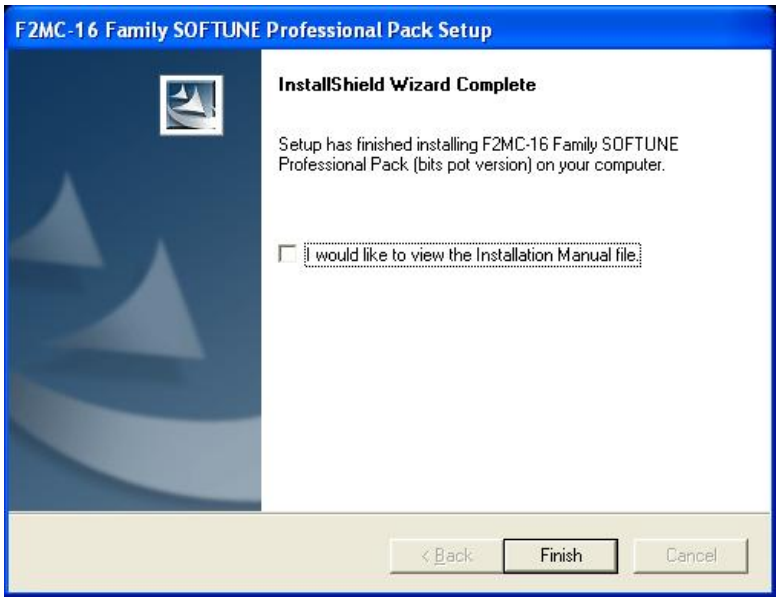


Figure 1-21 SOFTUNE setup/Completion

This completes the installation of SOFTUNE.

1.1.4 Installing PC Writer (bits pot white dedicated version)

Start installing PC Writer. Look for the following file in the folder where you decompressed in "Section 1.1.1 Downloading the software":

¥softwares¥pc writer¥MB96F356_setup.exe

① Double-click the downloaded "MB96F356_setup.exe"; the dialog shown in "Figure 1-22 PC Writer/Installation dialog" appears and installation starts; click the "Next" button.

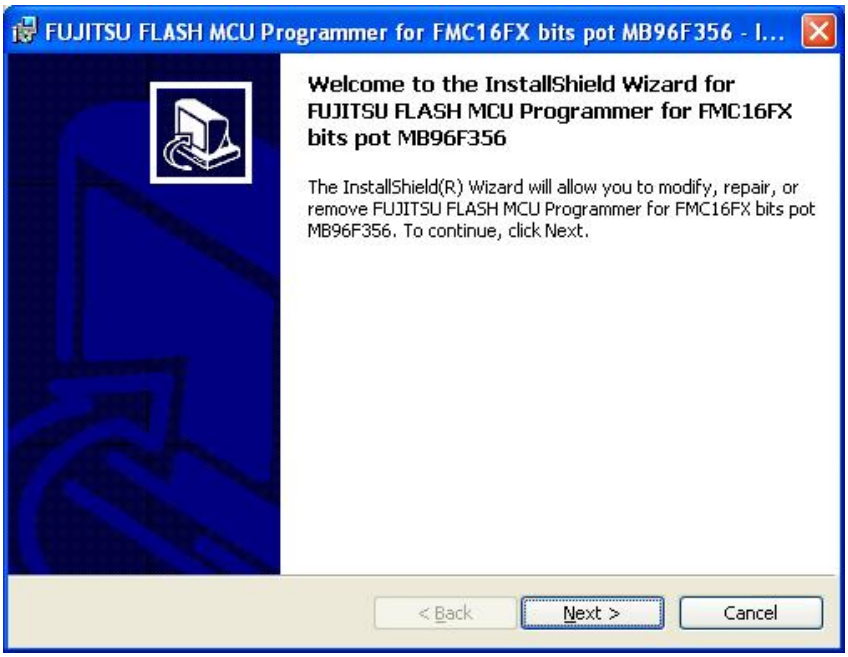


Figure 1-22 PC Writer/Installation dialog

② The dialog shown in “Figure 1-23 PC Writer/Setup type” appears; select “All”, and then click the “Next” button.

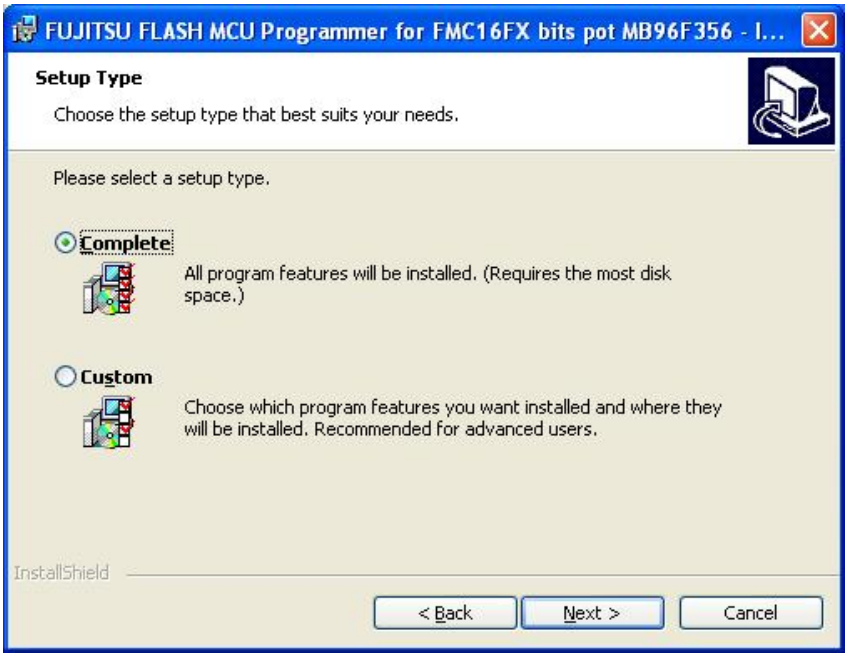


Figure 1-23 PC Writer/Setup type

③ The dialog shown in “Figure 1-24 PC Writer/Ready to install” appears to tell that the setup is ready to install PC Writer; click “Install”.

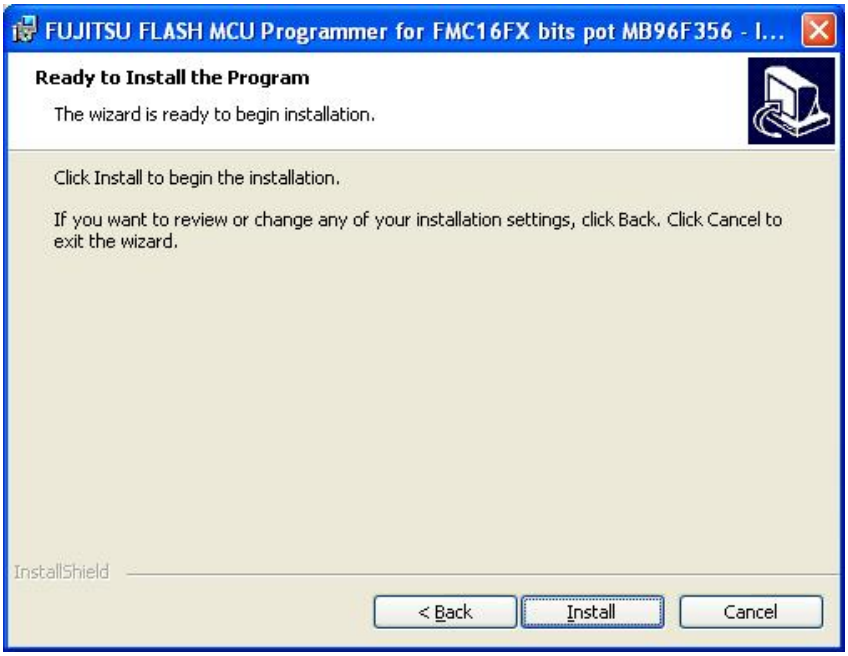


Figure 1-24 PC Writer/Ready to install

④ After the installation ends, the dialog shown in “Figure 1-25 Completing the PC Writer installation” appears to tell the completion of installation; click “Finish”.

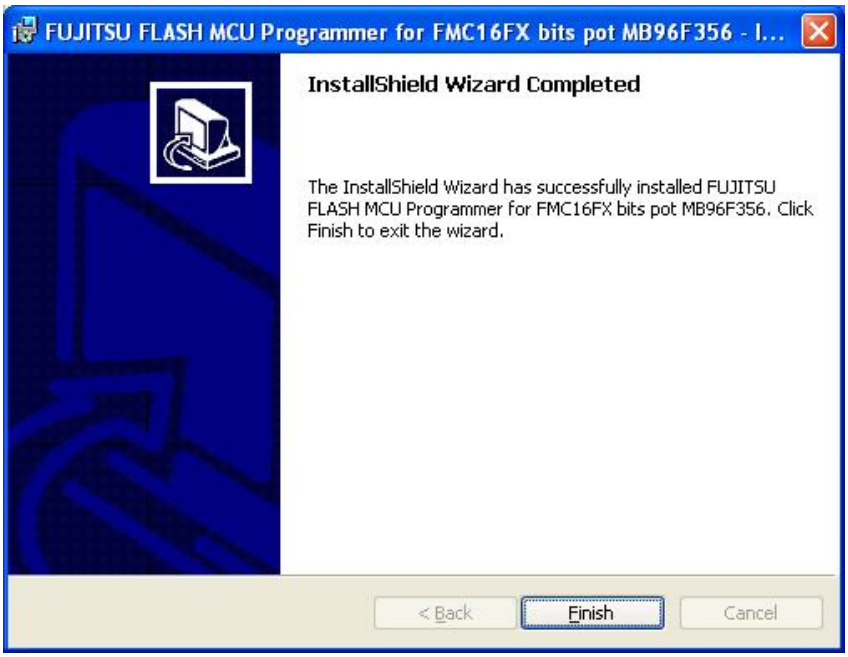


Figure 1-25 Completing the PC Writer installation

This completes the installation of PC Writer.

1.1.5 Installing EUROScope (evaluation version)

Start installing EUROScope (evaluation version). Look for the following file in the folder where you decompressed in “Section 1.1.1 Downloading the software”:

¥softwares¥euroscope¥euroscope _setup.exe

① Double-click the “euroscope_setup.exe” file you downloaded to begin the installation.

② Follow the on-screen directions to proceed with the installation. Click the “Next” button.

Figure 1-26 EUROScope installation dialog

③ If you agree, check the checkbox and click the “Next” button.

(If you do not agree, you cannot use the software.)

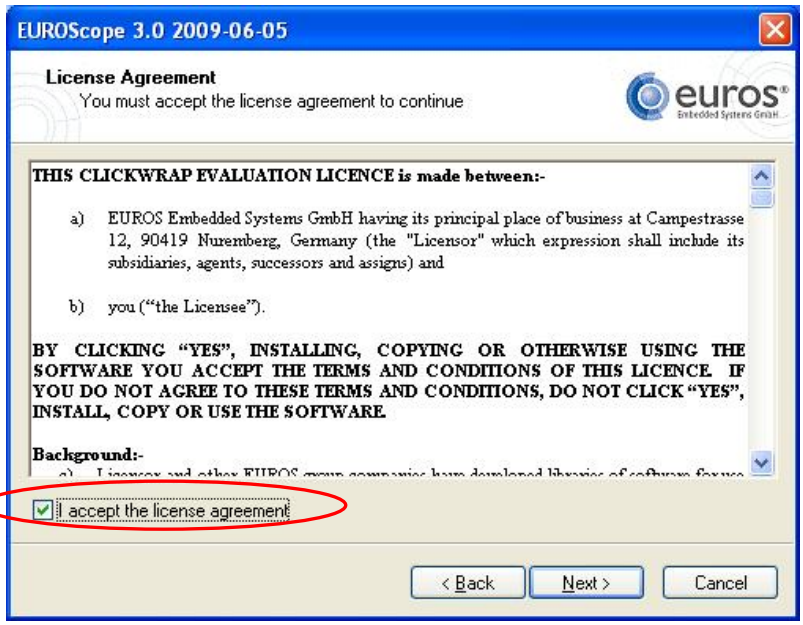


Figure 1-27 EUROScope/License agreement

④ Click the “Next” button.

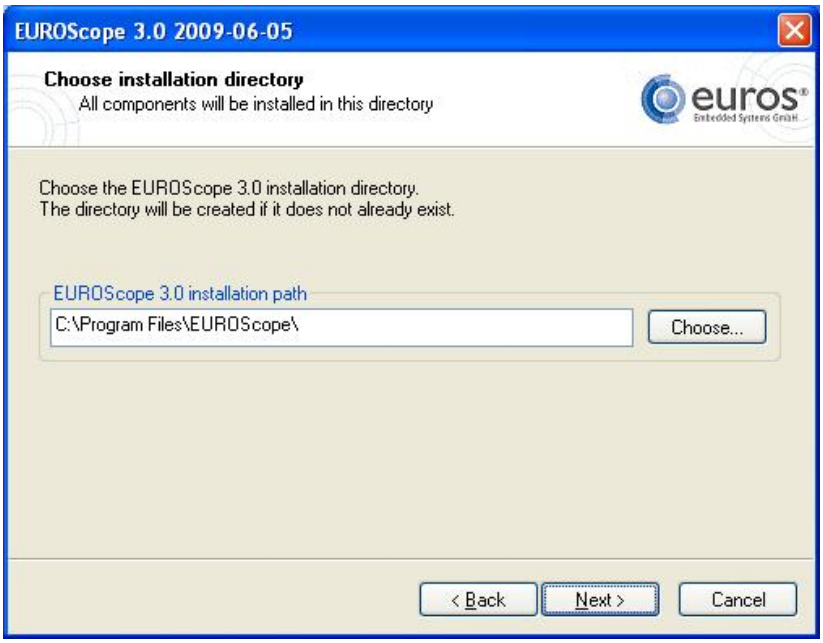


Figure 1-28 EUROScope/Install path

⑤ Check the “Fujitsu F16LX/F16FX” checkbox, and then click the “Next” button.

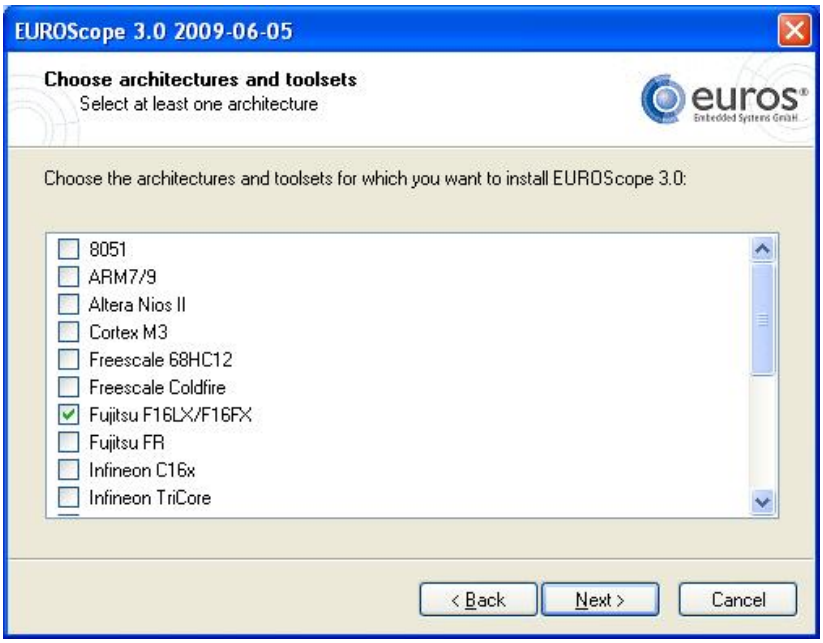


Figure 1-29 EUROScope/Selecting the architecture

⑥ Click the “Next” button.

Figure 1-30 EUROScope/Selecting the setup components

⑦ Click the “Next” button.

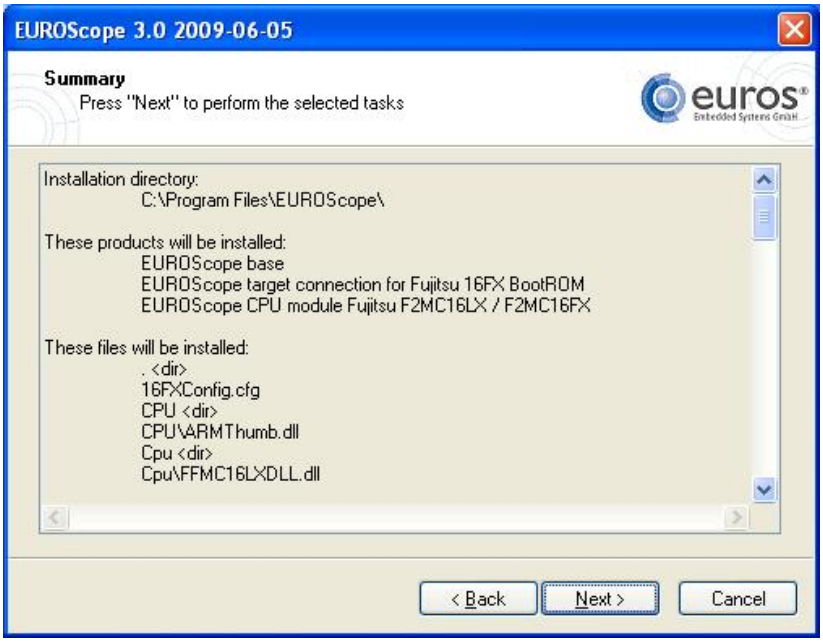


Figure 1-31 EUROScope/Confirming the setup information

⑧The installation is performed.

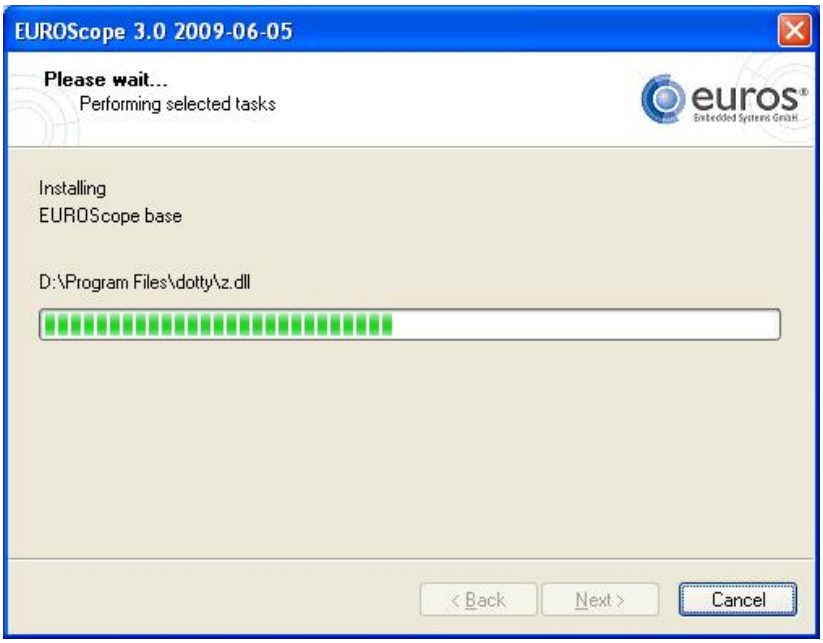


Figure 1-32 EUROScope/Executing setup

⑨Once the installation has finished, click the “Finish” button.

This completes the installation of EUROScope.

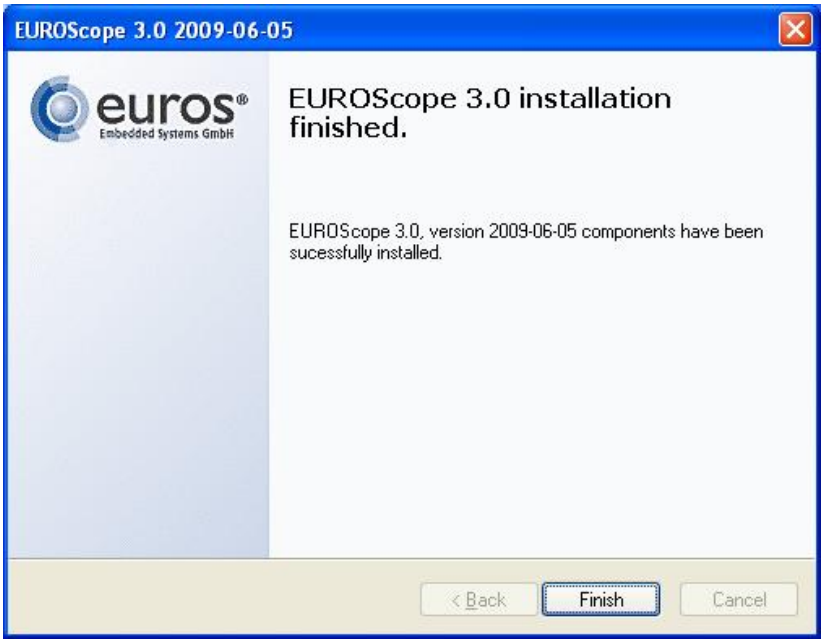


Figure 1-33 EUROScope/Setup complete

Next, activate Euroscope.

Select “All Programs”→ “EUROS”→ “EUROScope” from the Windows Start menu to activate Euroscope. Because there is no license for the first time, the following screen is displayed.

⑩ To obtain a license, note down the Host ID (Fig. 1-34 (1)), and for customers outside of Europe, click the “Request EUROScope lite 16FX key-other countries” button (Fig. 1-34 (2)).

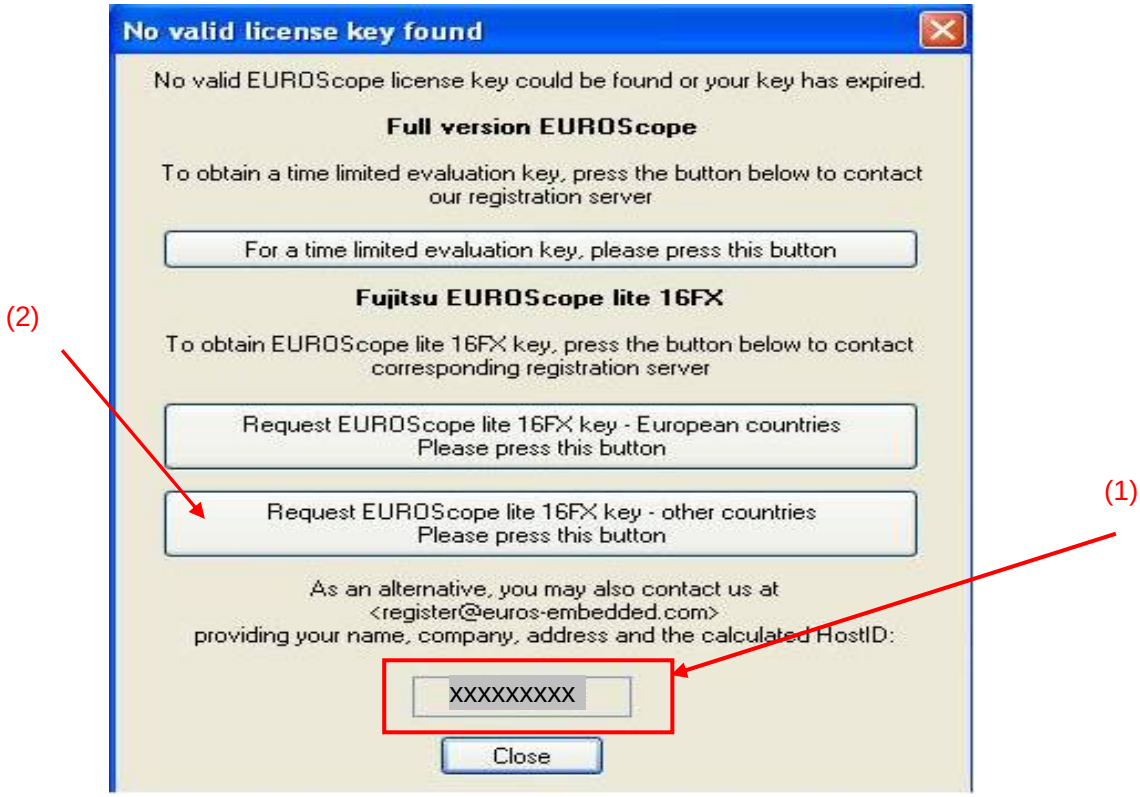


Figure 1-34 EUROScope/License information screen

(Note:The screen in Figure 1-34 outputs and Euroscope can not be started case, though the installation of Euroscope is completed. Such a situation occurs,when there are two or more MAC addresses of PC and installing. Please set the MAC address to one and install and start.)

⑪ When the following screen is displayed, fill in the Host ID (Fig. 1-34 (1)) you noted down and the required fields using single-byte alphanumeric characters, then click the “Request Keys” button. The license keys will be sent by e-mail at some later stage.

(Note: The situation may occur where “This page cannot be displayed” appears after you press the “Request Keys” button. However, this is not a problem as long as the license keys are sent by e-mail.)

Figure 1-35 EUROScope/Information input screen

⑫E-mail is sent by the EUROScope(Subject: “EUROScope Lite keys”).The license key is appended. Please decompressed the attachment(**euros-license.zip**).
There is euros-license.key when the attachment is decompressed. euros-license.key save in the EUROScope installation folder (the default is C:¥Program Files¥EUROScope).
(Note: The content of the key is different for each application.)

At this point, press the “Close” button in the screen shown in Fig. 1-35. Close the screen shown in Fig. 1-34 also to finish.

This completes the installation of EUROScope.

1.1.6 Configuring the board and connecting it to the PC

After installing SOFTUNE and EUROScope, configure the switches on the board and connect the board to the PC.

① Set the mode switch on the board to "**PROG**".

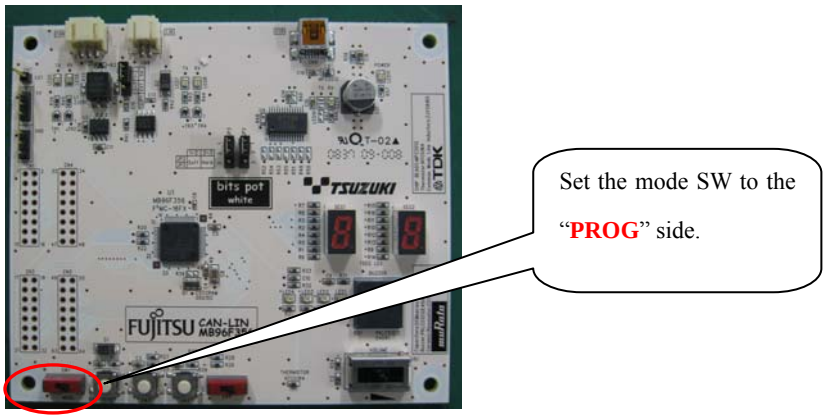


Figure 1-36 Mode switch

Mode switch	Operation mode
PROG	FLASH memory serial write mode → Used to write programs into the microcontroller.
RUN	Single chip mode → Used to run the program written into it.

Then, connect it to the PC.

② Connect the USB cable included in the kit to a USB port on the PC and the USB port on the board. Be sure to directly connect between them without using a USB hub.

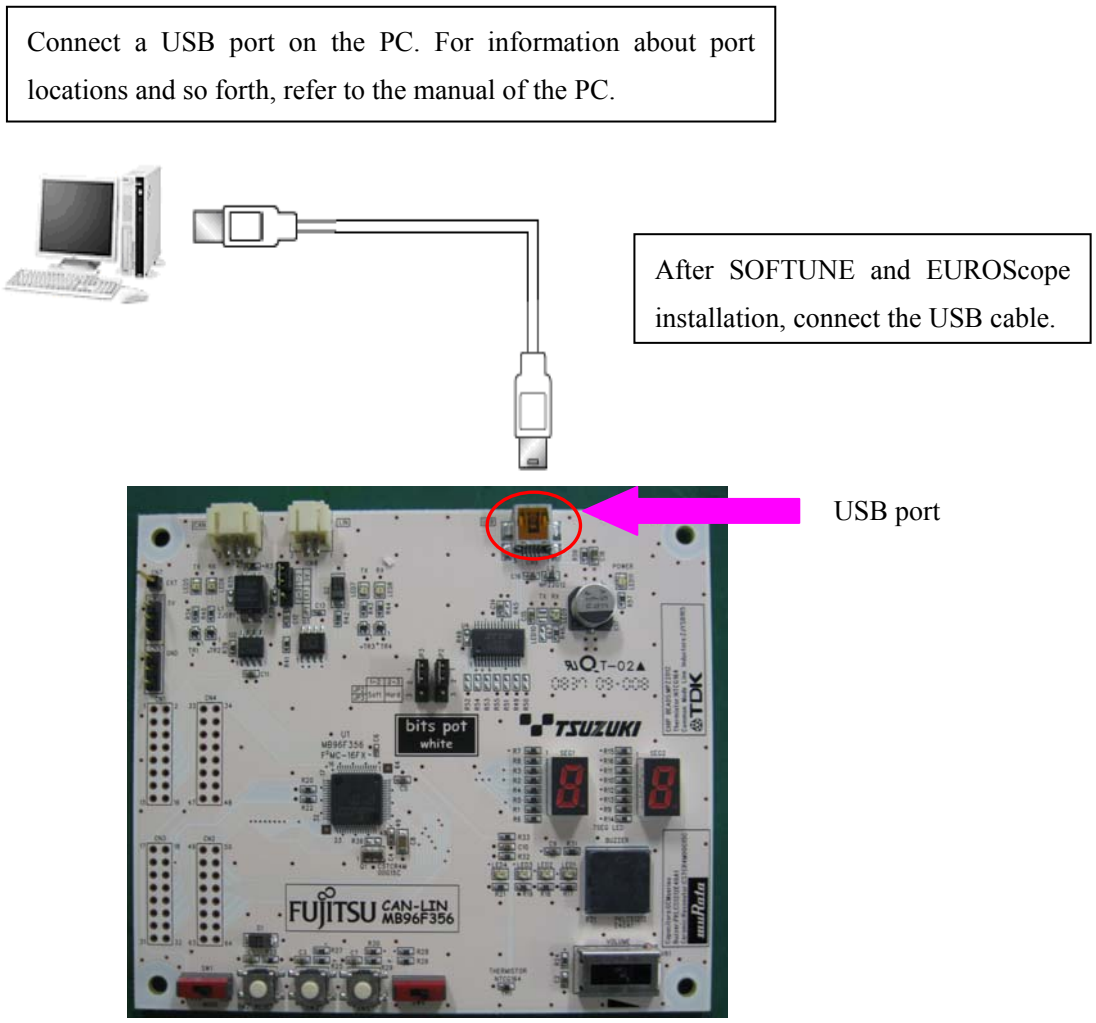


Figure 1-37 Connection between the PC and the board

The power of the board is supplied via USB (USB bus power).

[Note]

If a driver installation dialog is displayed after connecting the board to the PC, USB drivers may be incorrectly installed. Return to Section 1.1.2, “Installing the USB driver” and begin the installation again from the start.

2 Running the program

To run a program with the starter kit, use either of the following procedures.

2.1 Executing in single chip mode

In single chip mode, use the following procedures.

① Building a project

② Writing the program into the microcontroller

2.1.1 Building a project

Preparation

Decompress the following file in advance from within the folder you decompressed in Section 1.1.1, “Downloading the software”.

¥sample program¥bitspot_white_SampleProgram.zip

① Activate SOFTUNE.

Select “All Programs” → “Softune V3” → “FFMC-16 Family Softune Workbench” from the Windows Start menu to activate Softune.

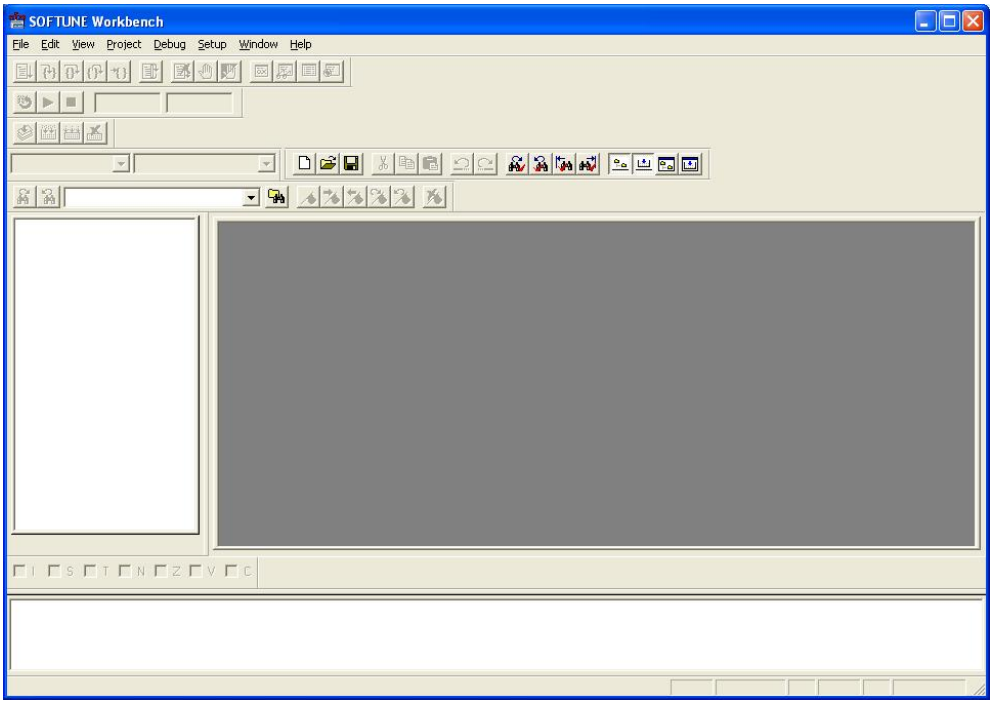


Figure 2-1 Activating SOFTUNE

② Open the workspace file for the sample program.

Select “Open Workspace” from the “File” menu.

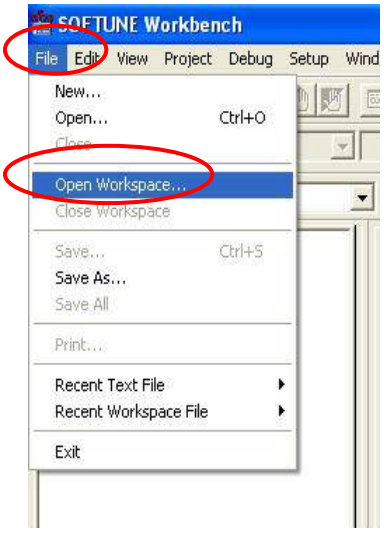


Figure 2-2 Opening a workspace

③Open the “bitspot_white_SampleProgram.wsp” file in
the bitspot_white_SampleProgram folder of the sample program.
¥bitspot_white_SampleProgram¥bitspot_white_SampleProgram.wsp

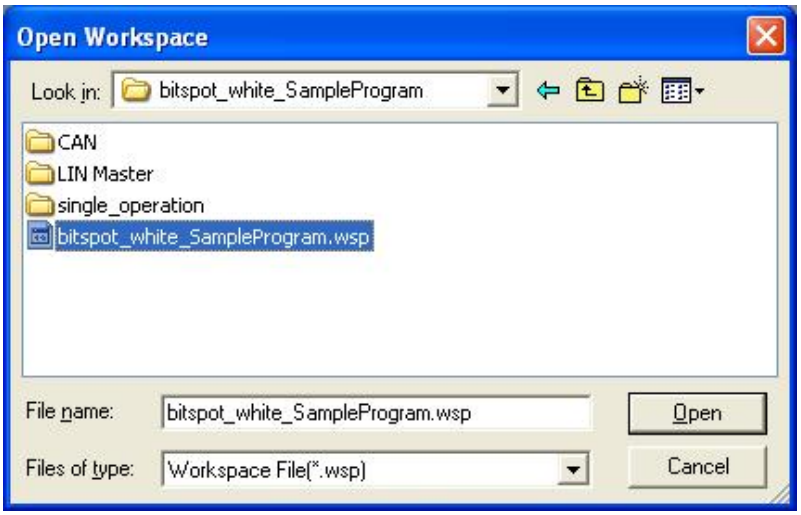


Figure 2-3 Selecting a workspace

At this point, the workspace file for running the sample program is open. (The sample program is not executed when this workspace file is opened.)

④ The workspace screen is opened. Select the project you want to run from among (1) to (3) below, and set it as the active project. Next, click “Project” → “Build” from the menu to build the project. (The method for selecting the active project is by “right-clicking on the project name” and then selecting “Set as Active Project”.)

(1) For single-unit operation

“single_operation.abs” ⇒ Active project

(2) For CAN communication

“CAN.abs” ⇒ Active project

(3) For LIN communication

“LIN MASTER.abs” ⇒ Active project

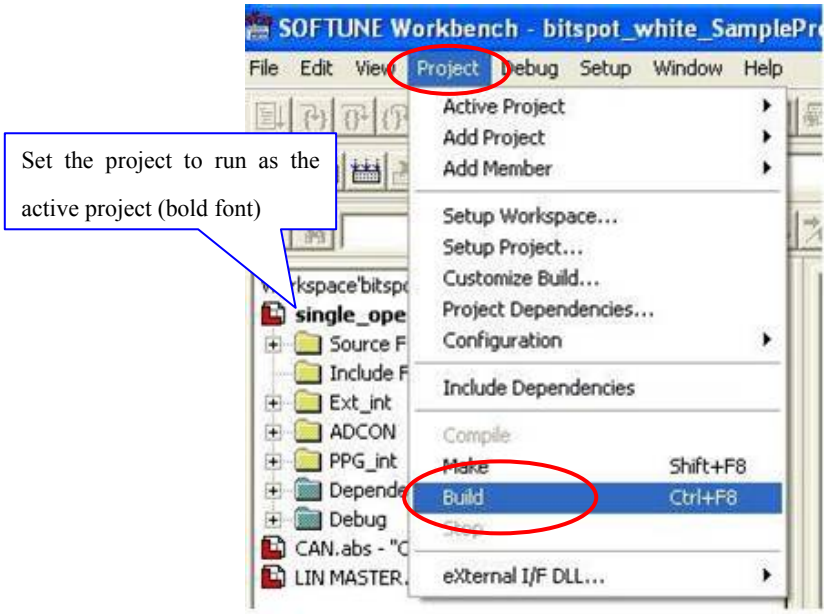


Figure 2-4 Building a project

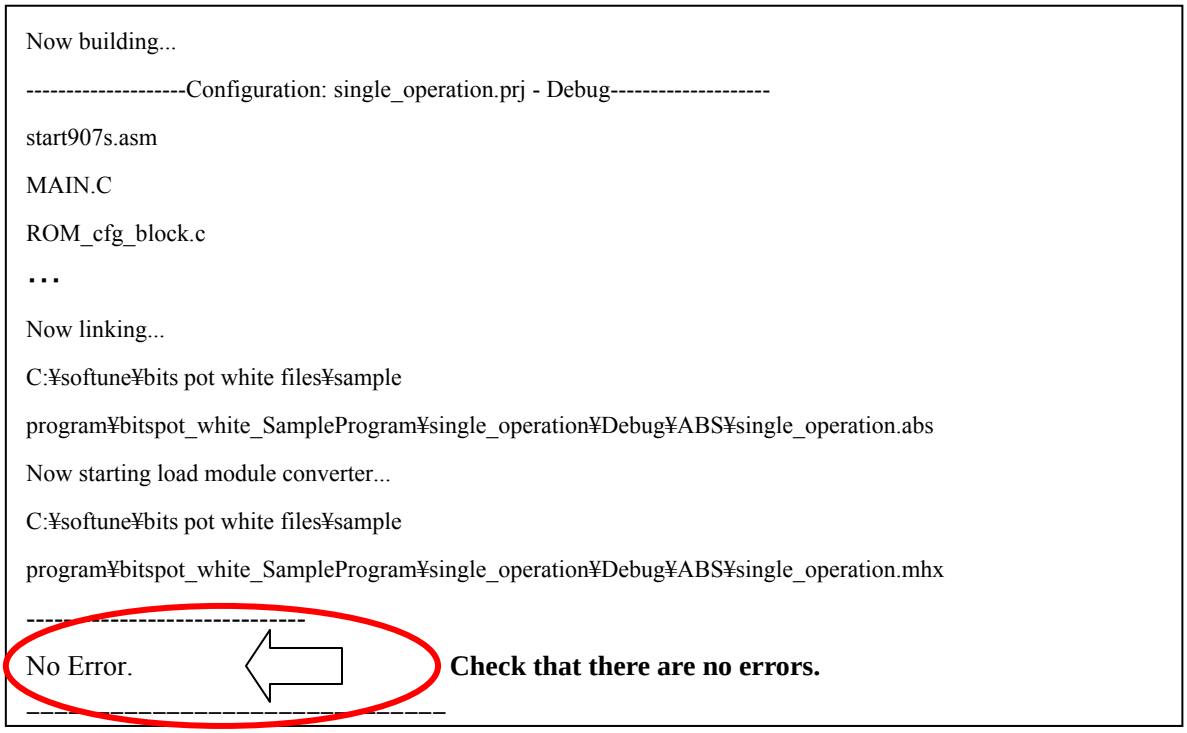


Figure 2-5 Completing the build

2.1.2 Writing the program into the microcontroller

Preparation

The mode switch on the board is required to be set to “PROG” in order to write the program. If the mode switch is not set to “PROG”, switch the mode switch to “PROG”.

① Select “All Programs” → “FUJITSU FLASH MCU Programmer” → “MB96F356” from the Windows Start menu to activate PC Writer.

② To select a file to be written as shown in “Figure 2-6 Opening the file to write”, click the “Open” button.

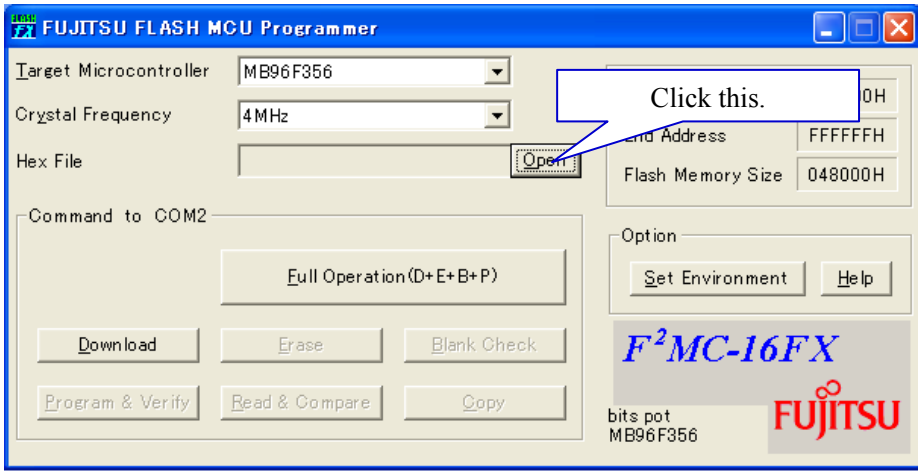


Figure 2-6 Opening the file to write

③ The dialog that allows you to select the file is displayed as shown in “Figure 2-7 Selecting the file to write”; select the file built in Section “2.1.1, ④ Building a project” and then click “Open”.

(1) For single-unit operation

¥bitspot_white_SampleProgram¥single_operation¥Debug¥ABS¥single_operation.mhx

(2) For CAN communication

¥bitspot_white_SampleProgram¥CAN¥Debug¥ABS¥CAN.mhx

(3) For LIN communication

¥bitspot_white_SampleProgram¥LIN Master¥Debug¥ABS¥LIN MASTER.mhx

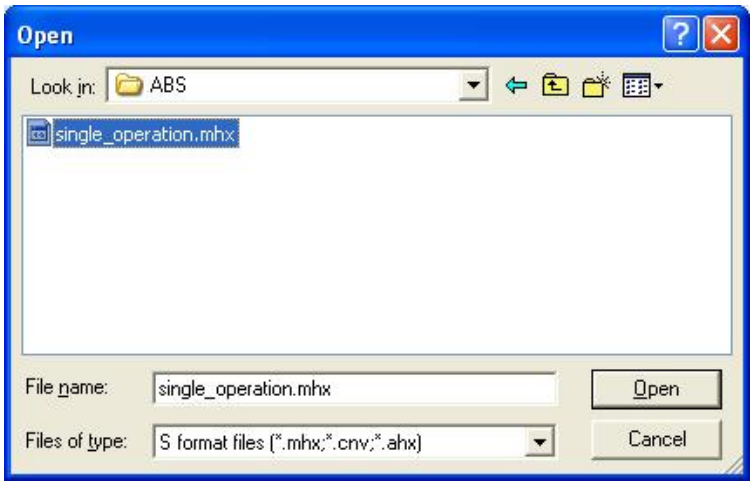
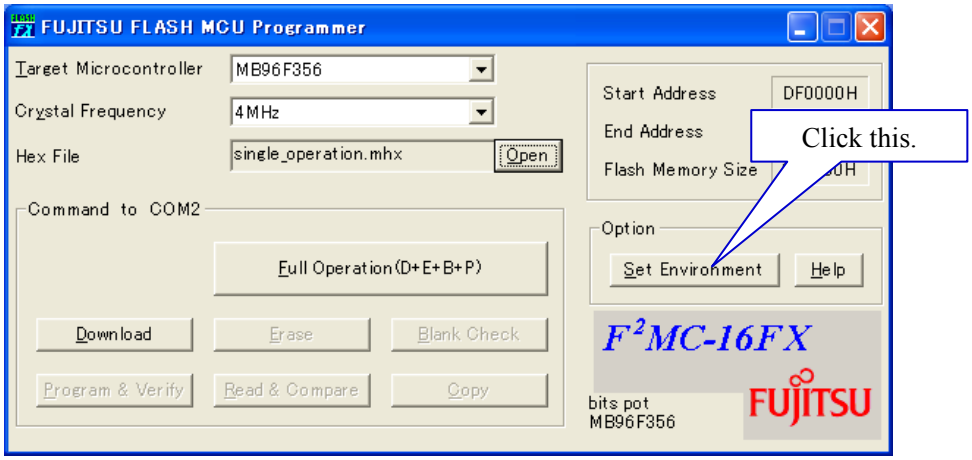


Figure 2-7 Selecting the file to write

④ Then, select the COM port to be used for the writing. Click the “Set Environment” button; the COM port selection dialog appears. Select the COM port with which the board is connected, and then click the “OK” button.



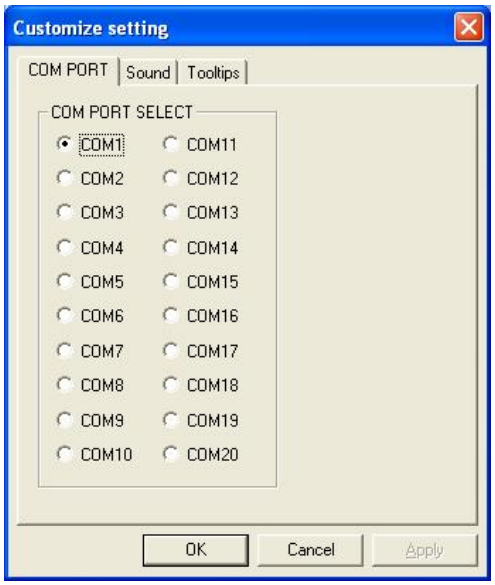


Figure 2-8 Selecting the COM port to be used for writing

Note: To check the COM port in use, right-click “My Computer” and then select “Properties”; the system properties are displayed. Select the “Hardware” tab and then click the “Device Manager” button. After Device Manager activates, check the COM port number in the parentheses of “USB Serial Port (COM n)” under “Port (COM and LPT)” in the tree shown in “Figure 2-9 Checking the COM port”.

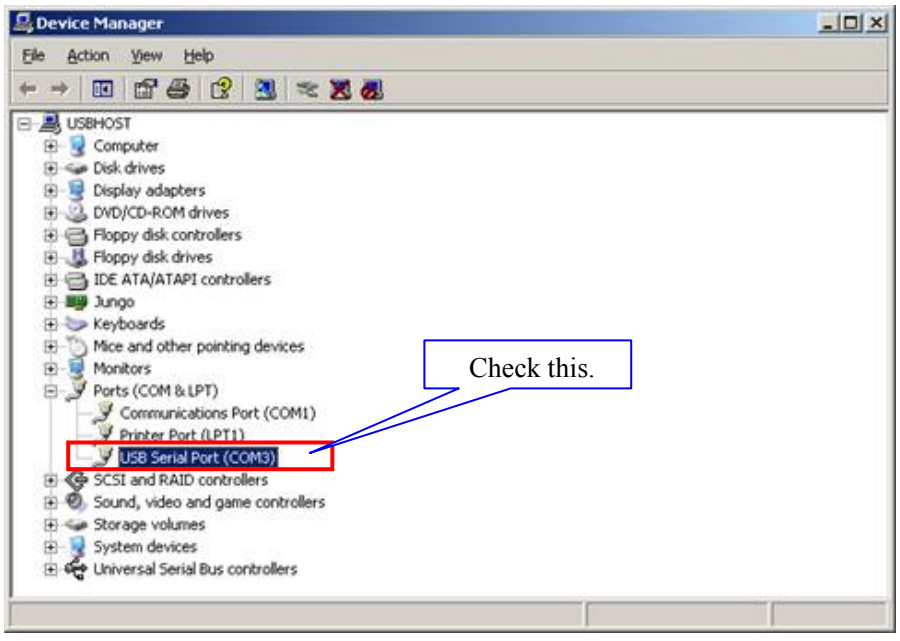


Figure 2-9 Checking the COM port

⑤ As shown in “Figure 2-10 Writing the program”, press the “Full Operation” button to start writing the program; the dialog that asks you to press the Reset switch is displayed. Press the Reset SW on the board, and then click the “OK” button on the dialog; the program write sequence starts. For the location of the Reset SW, see “Figure 1-1 External board view”.

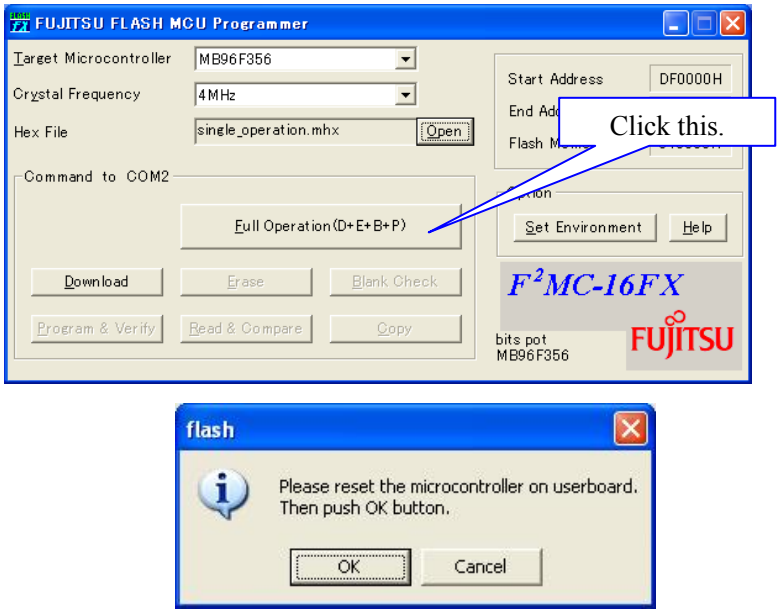


Figure 2-10 Writing the program

⑥ The dialog shown in “Figure 2-11 Completing the program writing” is displayed to notify you of the completion of the program writing; press the “OK” button to quit PC Writer.

Figure 2-11 Completing the program writing

Set the MODE switch on the board to “RUN” and then press the Reset button; the program starts running. (Note: To perform CAN communication and LIN communication, the bits pot red and bits pot yellow need to be connected. Refer to “Figure 1-3 System connection diagram”. In addition, refer to the respective manuals for the connections and settings in the starter kit.)

2.2 Debugging by using Monitor Debugger

To debug by using Monitor Debugger, use the following procedures.

① Activating and configuring SOFTUNE

② Changing the source file to activate with EUROScope

③ Writing the program into the microcontroller

④ Activating and configuring EUROScope

2.2.1 Activating and configuring SOFTUNE

Preparation

Decompress the following file from within the folder you decompressed in “Section 1.1.1, Downloading the software” in advance.

¥sample program¥bitspot_white_SampleProgram.zip

① Activate SOFTUNE.

Click “All Programs” → “Softune V3” → “FFMC-16 Family Softune Workbench” from the Windows Start menu to activate Softune.

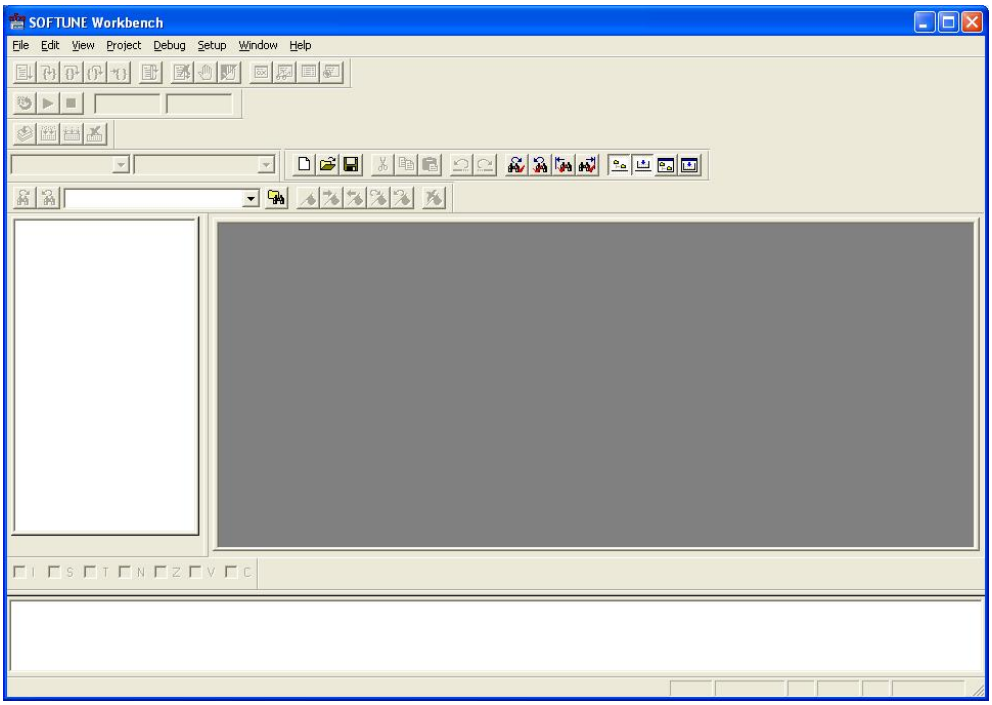


Figure 2-12 Activating SOFTUNE

② Open the workspace file for the sample program.

Select “Open Workspace” from the “File” menu.

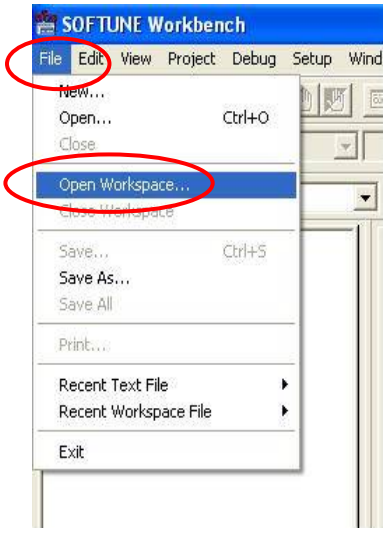


Figure 2-13 Opening a workspace

③ Open the “bitspot_white_SampleProgram.wsp” file in the bitspot_white_SampleProgram folder of the sample program.

¥bitspot_white_SampleProgram¥bitspot_white_SampleProgram.wsp

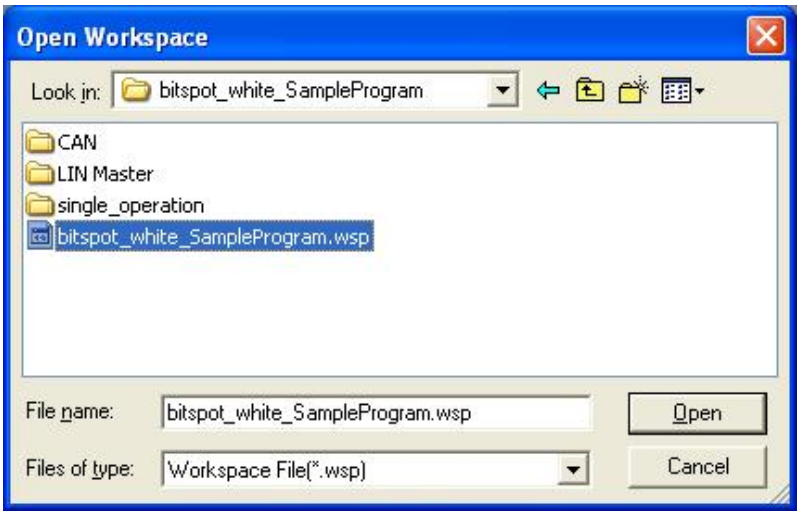


Figure 2-14 Selecting a workspace

At this point, the workspace file for running the sample program is open. (The sample program is not executed when this workspace file is opened.)

2.2.2 Changing the source file to activate with EUROScope

To run the sample programs using EUROScope, the source files need to be changed and built in SOFTUNE. Use the following procedures.

① As shown in “Figure 2-15 Opening the ROM_cfg_block.c file”, select the project to execute from the list of SOFTUNE projects.
Select “ROM_cfg_block.c” from within the project, and double-click to open the file.
(Note: The “ROM_cfg_block.c” file of each project is shown (1) to (3) below. Furthermore, ensure that the project for each of these operations is set as the active project (shown in bold font). (The method for setting the active project is by “right-clicking the project name” and select “Set Active Project”.)

(1) For single-unit operation
“single_operation.abs” (Active project) → “Source Files” → “ROM_cfg_block.c”
(2) For CAN communication
“CAN.abs” (Active project) → “Vct” → “ROM_cfg_block.c”
(3) For LIN communication
“LIN MASTER.abs” (Active project) →“Source Files” → “APPL” → “ROM_cfg_block.c”

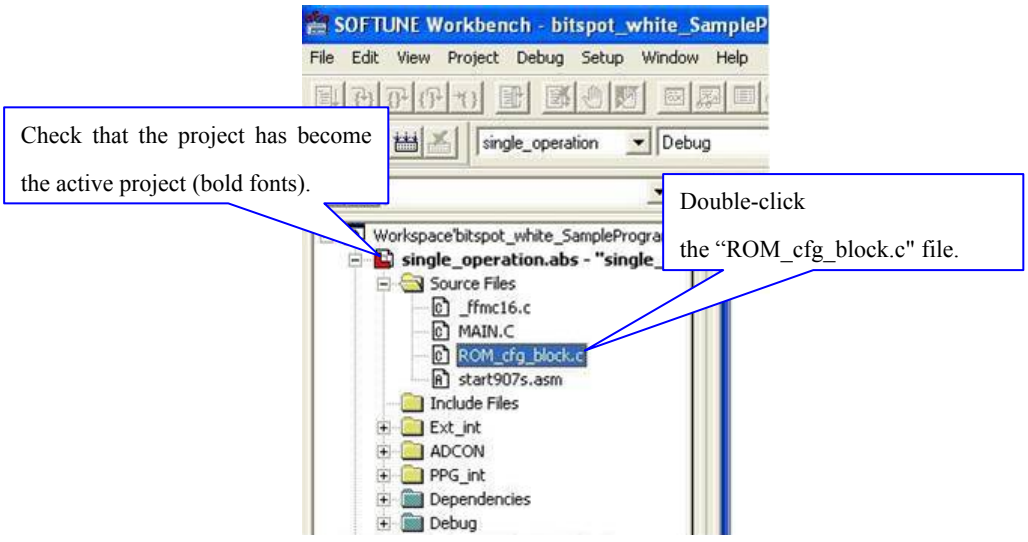


Figure 2-15 Opening the ROM_cfg_block.c file

② Change the “OFF” part to “ON” in the line
“#set BACKGROUND_DEBUGGING OFF;” (default state) under the
“4.18 Enable Background Debugging Mode” line in the ROM_cfg_block.c file
(Fig. 2-16 (1)).

③ Check that the value of the second digit from the right in the line
“#set BDM_CONFIGURATION B’000000000000010;” (default state) shown in
Fig.2-16 (2) is “1”.

④ Check that the value (BAUDRATE) is “115200” in the line
“#set BDM_BAUDRATE 115200;” shown in Fig. 2-16 (3).

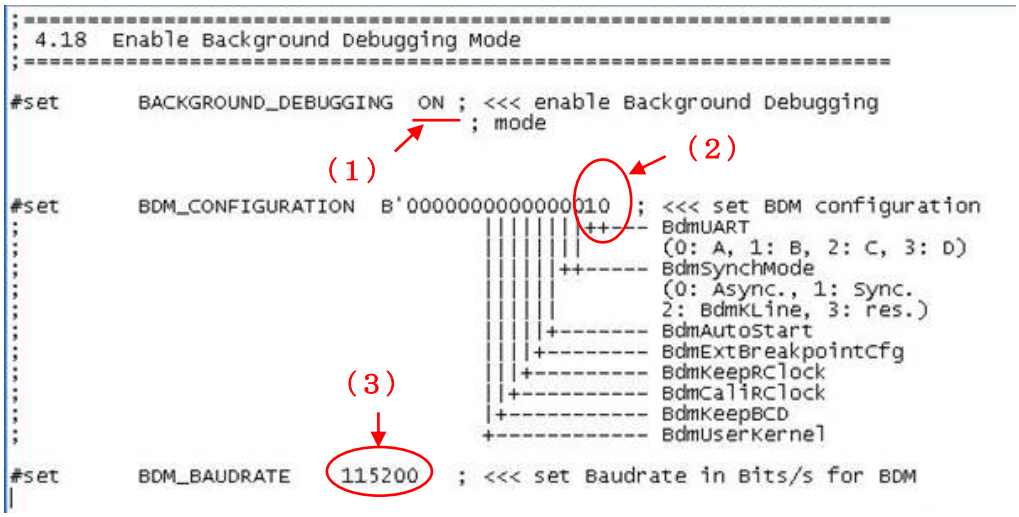


Figure 2-16 ROM_cfg_block.c file

⑤ Select “File” → “Save All” from the SOFTUNE menu to save the changes to the “ROM_cfg_block.c” as shown in Fig. 2-16.

⑥ Select “Project” → “Build” from the SOFTUNE menu to build the project.

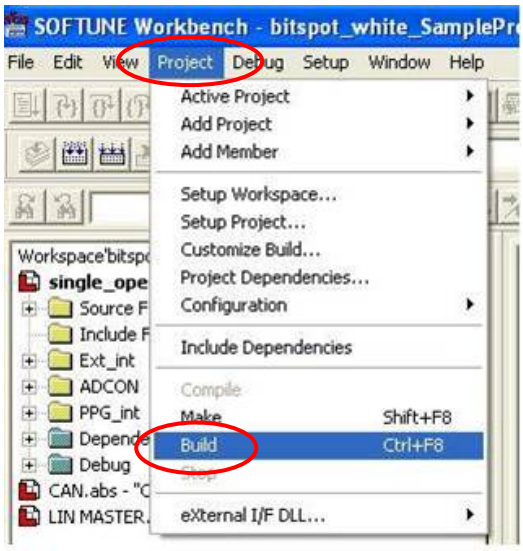


Figure 2-17 Building the project

⑦ Once the build is completed, check that no errors are output, and check the .mhx file is created in the following locations.

(1) For single-unit operation

¥bitspot_white_SampleProgram¥single_operation¥Debug¥ABS¥single_operation.mhx

(2) For CAN communication

¥bitspot_white_SampleProgram¥CAN¥Debug¥ABS¥CAN.mhx

(3) For LIN communication

¥bitspot_white_SampleProgram¥LIN Master¥Debug¥ABS¥LIN MASTER.mhx

2.2.3 Writing the program into the microcontroller

Preparation

In order to write the program, the mode switch on the board needs to be set to “PROG”. If the mode switch is not set to “PROG”, switch the mode switch to “PROG”.

① Select “All Programs” → “FUJITSU FLASH MCU Programmer” → “MB96F356” from the Windows Start menu to activate PC Writer.

② In order to select the file to write as shown in “Figure 2-18 Opening the file to write”, click the “Open” button.

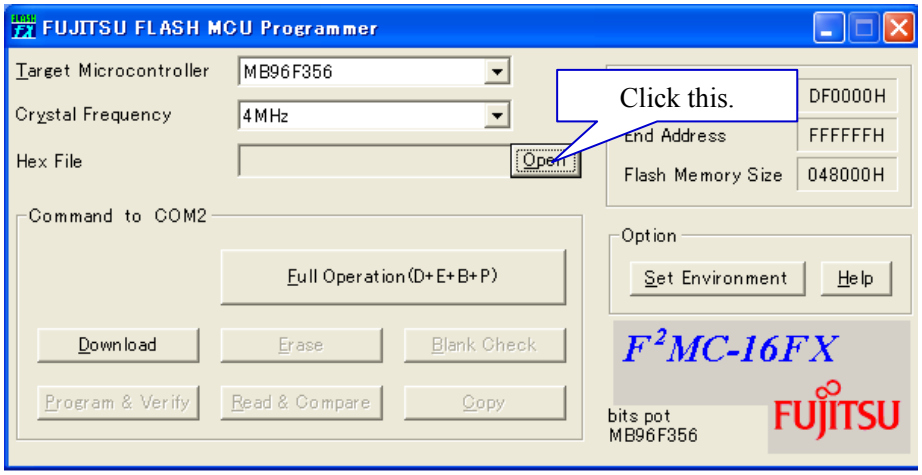


Figure 2-18 Opening the file to write

③ The dialog that allows you to select the file is displayed as shown in “Figure 2-19 Selecting the file to write”; select the file built in Section 2.2.2, “⑦ Building the project” and then click “Open”.

(1) For single-unit operation

¥bitspot_white_SampleProgram¥single_operation¥Debug¥ABS¥single_operation.mhx

(2) For CAN communication

¥bitspot_white_SampleProgram¥CAN¥Debug¥ABS¥CAN.mhx

(3) For LIN communication

¥bitspot_white_SampleProgram¥LIN Master¥Debug¥ABS¥LIN MASTER.mhx

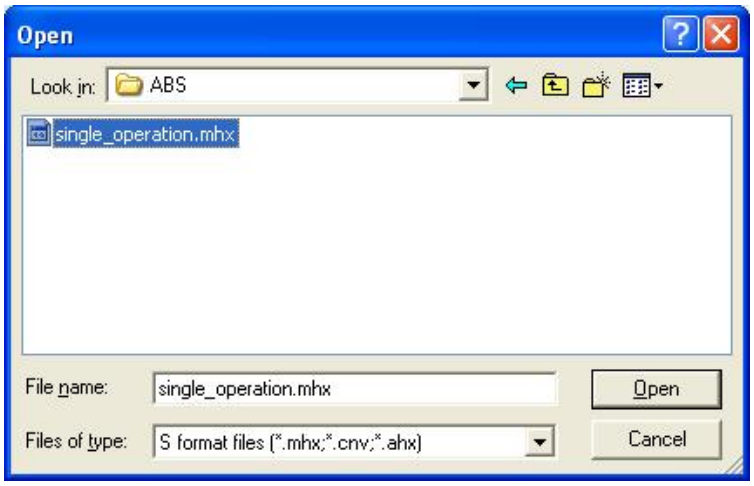
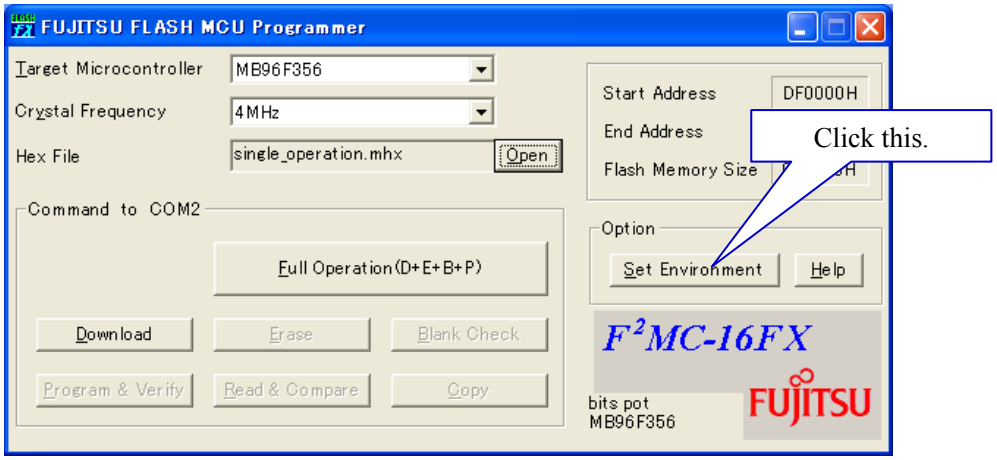


Figure 2-19 Selecting the file to write

Then, select the COM port to be used for the writing. Click the “Set Environment” button; the COM port selection dialog appears. Select the COM port with which the board is connected, and then click the “OK” button.



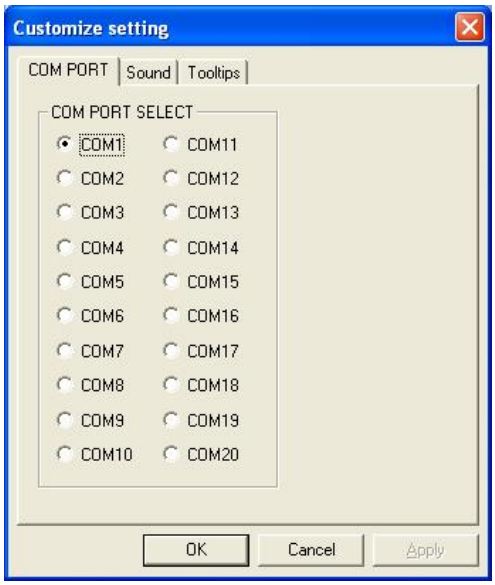


Figure 2-20 Selecting the COM port to be used for writing

Note: To check the COM port in use, right-click “My Computer” and then select “Properties”; the system properties are displayed. Select the “Hardware” tab and then click the “Device Manager” button.

After Device Manager activates, check the COM port number in the parentheses of “USB Serial Port (COM n)” under “Port (COM and LPT)” in the tree shown in “Figure 2-21 Checking the COM port”.

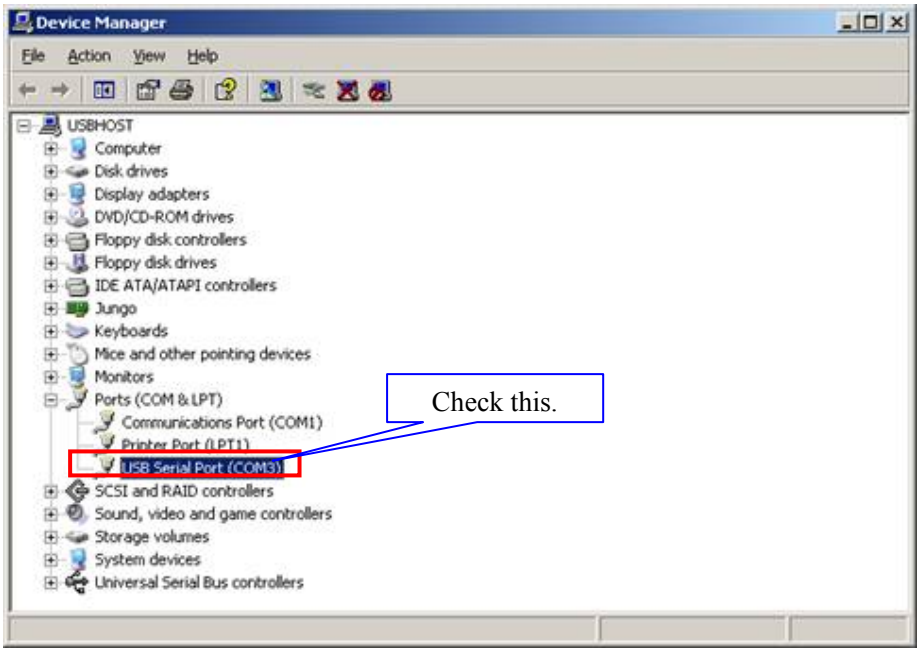


Figure 2-21 Checking the COM port

⑤ As shown in “Figure 2-22 Writing the program”, press the “Full Operation” button to start writing the program; the dialog that asks you to press the Reset switch is displayed. Press the Reset SW on the board, and then click the “OK” button on the dialog; the program write sequence starts. For the location of the Reset SW, see “Figure 1-1 External board view”.

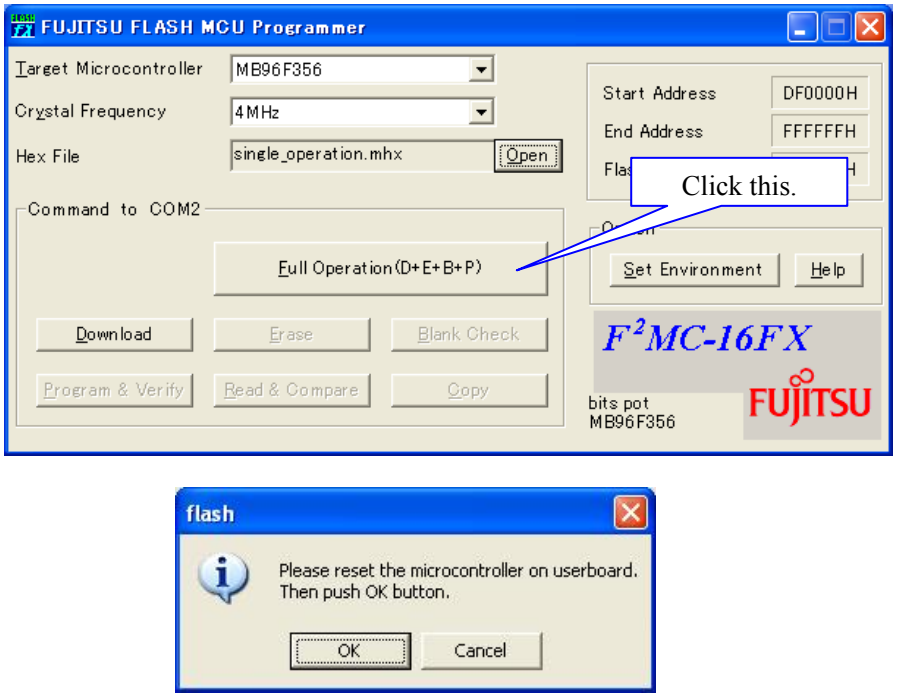


Figure 2-22 Writing the program

⑥ The dialog shown in “Figure 2-23 Completing the program writing” is displayed to notify you of the completion of the program writing; press the “OK” button to quit PC Writer.

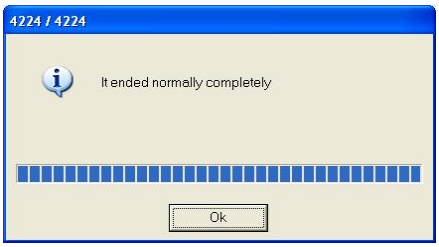


Figure 2-23 Completing the program writing

⑦ Remove the USB cable from the board and set the mode switch to “RUN”. After this, reconnect the USB cable. (Note: To perform CAN communication and LIN communication, the bits pot red and bits pot yellow need to be connected. Refer to “Figure 1-3 System connection diagram”. In addition, refer to the respective manuals for the connections and settings in the starter kit.)

2.2.4 Activating and configuring EUROScope

① Activating EUROScope.

Click “All Programs” → “EUROS” → “EUROScope” from the Window Start menu. (Alternatively, double-click the “EUROScope” shortcut on the desktop.)

The EUROScope debug window is displayed.

② Specify the information file (single_operation.abs) necessary for execution.

Select “File” → “Open Application…”.

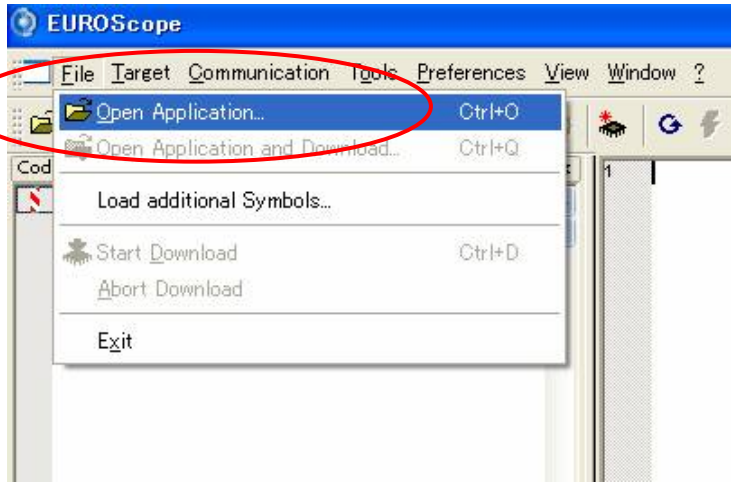


Figure 2-24 Opening a file

③ Read the file (.abs).

Select the .abs file as specified in the following locations and click the “Open” button.

(1) For single-unit operation

¥bitspot_white_SampleProgram¥single_operation¥Debug¥ABS¥single_operation.abs

(2) For CAN communication

¥bitspot_white_SampleProgram¥CAN¥Debug¥ABS¥CAN. abs

(3) For LIN communication

¥bitspot_white_SampleProgram¥LIN Master¥Debug¥ABS¥LIN MASTER. abs

(Note: It is not a problem if “No source for module ‘_ffmc16’” is displayed on the EUROScope debug screen after reading the file.)

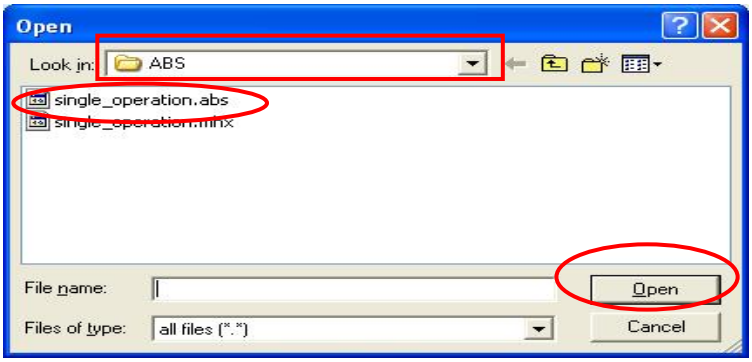


Figure 2-25 Selecting the abs file

④ Select the PC connection setting for the board.

Select “Preferences” → “Select Target Connection…” from the menu.

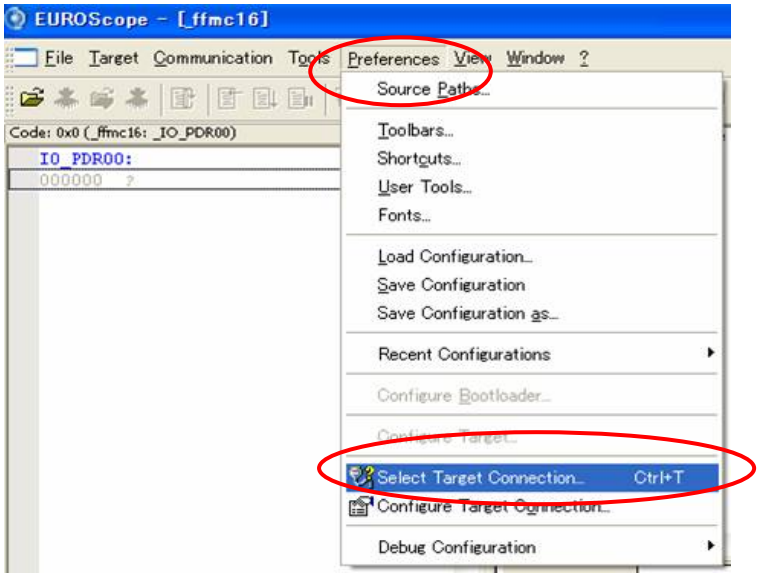


Figure 2-26 Board connection setting menu

⑤ “Select target connection” (Fig. 2-27) is displayed.

Check the “Fujitsu 16FXBootROM (RS232)” is selected. Then, press the “Configure” button.

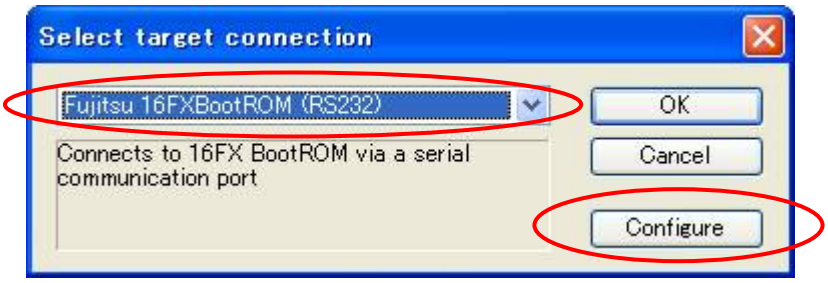


Figure 2-27 Board connection setting

⑥ “Configure 16FX BootROM connec…” (Fig. 2-28) is displayed. Set each of the fields as follows.

- **Port: Select the port number of the connected COM port.**

  (Note: Set this to the COM port number you verified in Step ④ of Section 2.2.3, “Writing the program into the microcontroller”. However, for COM port numbers of “10” or higher, because the setting as shown in Fig. 2-28 cannot be configured, change the PC port number to “9” or less before making this setting.)

- **Baudrate: 115200**
- **Communication options: Asynchronous communication**

  : Int/Ext vector mode

When you have finished all of the settings, click the “OK” button.

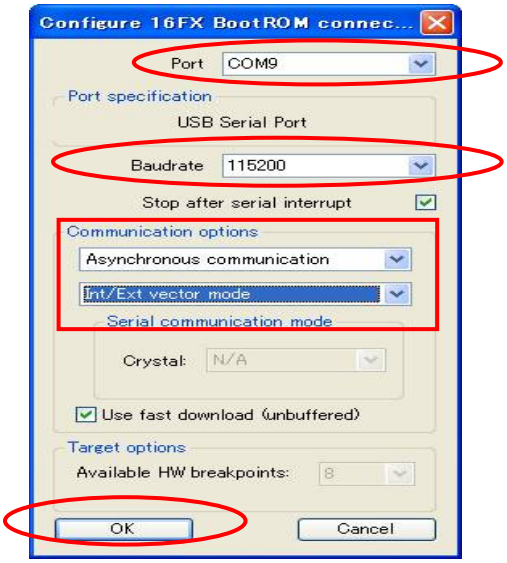


Figure 2-28 Board connection settings

⑦ In “Select target connection”, click the “OK” button.

Figure 2-29 Board connection settings complete

⑧ Press the “RESET” button on the board, and then select “Communication” → “Open” from the menu,

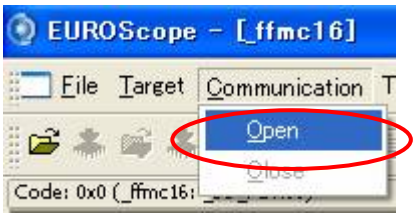


Figure 2-30 Opening a file

(Note: After doing the above mentioned procedure, The message in Figure 2-31 outputs. Please push "Yes".)

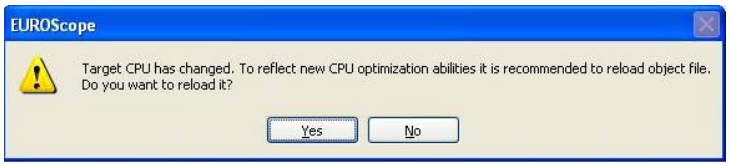


Figure 2-31 Output message

If EUROScope is correctly connected to the board, the general-purpose registers, code, and memory values are displayed in the EUROScope window. (Figure 2-32 Debug screen) Furthermore, the button for starting execution is enabled. (Figure 2-33 Debug execution button (1))

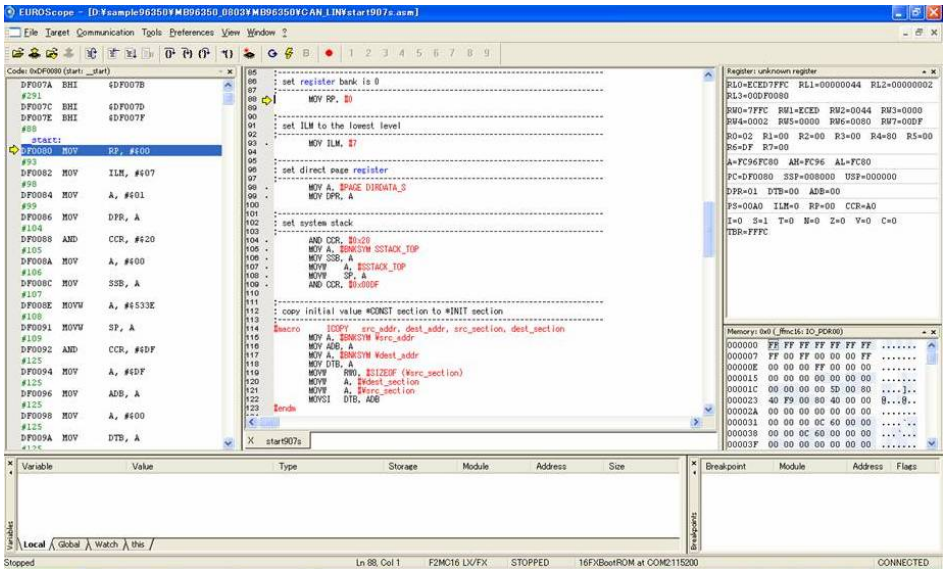


Figure 2-32 Debug screen

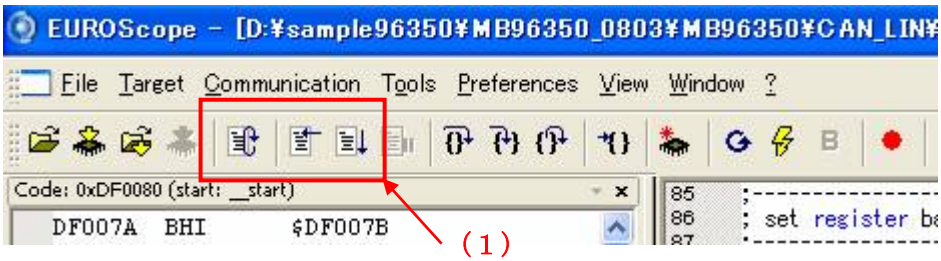


Figure 2-33 Debug execution button

⑨ Run EUROScope. (Note: To perform CAN communication or LIN communication, check that the bits pot red or bits pot yellow is connected.)
Select “Target” → “Initialize” from the menu.

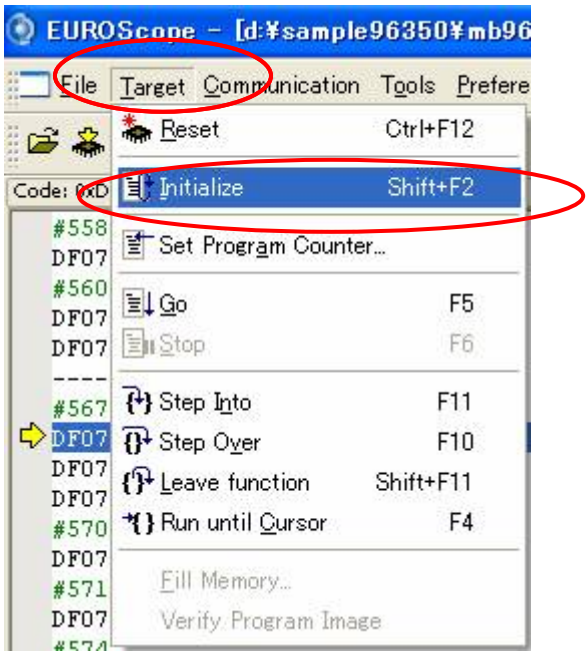


Figure 2-34 Initializing debug execution

Check that there is a yellow arrow at the start of the main routine (under void main (void)) in the program window displayed in the center of the EUROScope screen.

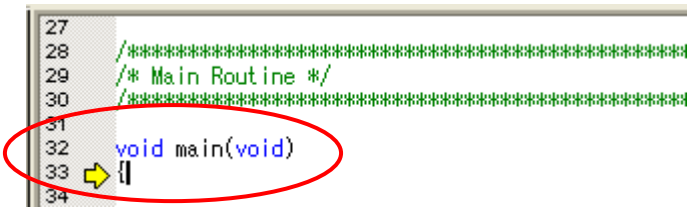


Figure 2-35 Beginning of the program

⑩ Once you have confirmed the yellow arrow, select “Target” → “Go” from the menu to run EUROScope.
(This makes it possible to perform operations on the board.)

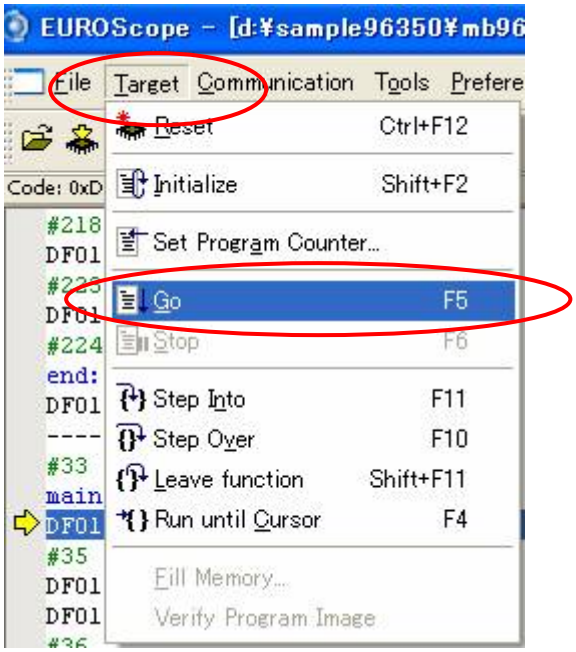


Figure 2-36 Starting debug execution

⑪ To stop debug execution on the board, stop from EUROScope.
Select “Target” → “Stop” from the menu to stop execution. (This stops the operation of this board.)

Figure 2-37 Stopping debug execution

(If you want to run again without quitting EUROScope, repeat the procedure from step ⑨ above.)

2.3 Exiting EUROScope

The procedure for exiting EUROScope is as follows.

① Select “Communication” → “Close” from the menu.

Figure 2-38 Ending the execution program

② Select “File” → “Exit” from the menu. The EUROScope debug window closes.

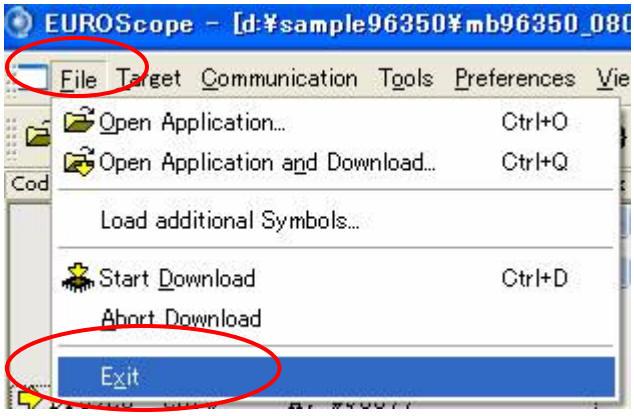


Figure 2-39 Exiting EUROScope

(Note: There is a case that the message to save EUROScope configuration of Figure 2-40 outputs after doing the above mentioned procedure. Please put the check in "Don't show again" and push "Yes" for the following messages. Next, figure 2-41 outputs. Please input an arbitrary name to the “File name” and push “save”.Next time, when EUROScope is started, the message doesn't output.)

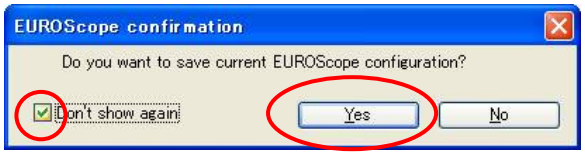


Figure 2-40 Output message

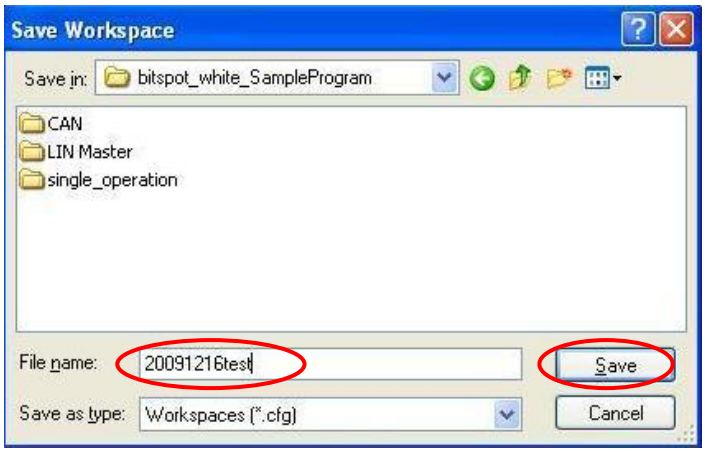


Figure 2-41 Configuration save

2.4 Exiting SOFTUNE

The procedure for exiting SOFTUNE is as follows.

① Select "File" → “Close Workspace” from the menu.

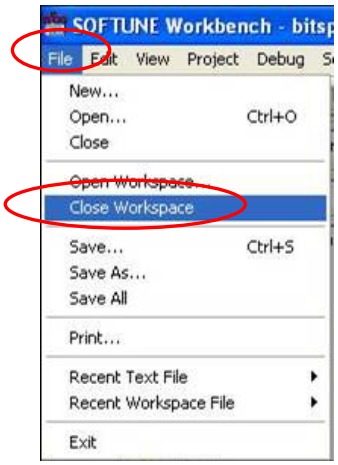


Figure 2-42 Closing a workspace

② A message pops up asking “Save changes to project?”. Click the “Yes” button.

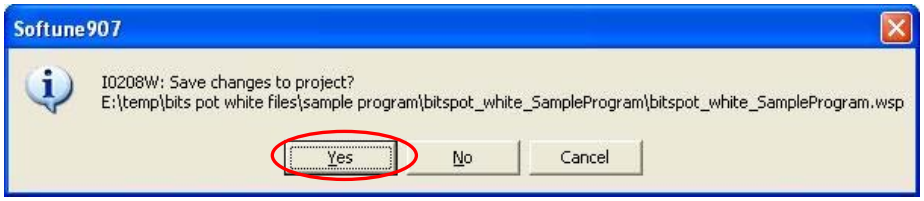


Figure 2-43 Saving a workspace

③ Select “File” → “Exit” from the menu to exit SOFTUNE.

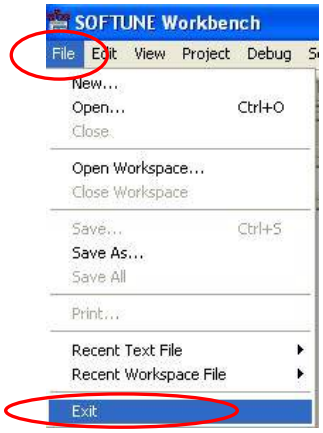


Figure 2-44 Exiting SOFTUNE

This completes basic operations up to starting and exiting debug execution, using the sample program.

3 Operation of the sample program

This section describes the operation of the sample program. The operation of the sample is classified into the following three categories.

① bits pot white single-unit operation
② CAN communication operation (CAN communication operation with bits pot red)
③ LIN communication operation (LIN communication operation with bits pot yellow)

3.1 bits pot white single-unit operation

The operation of programs operating bits pot white as a single unit is shown in this section.

"Figure 3-1 Controls and operations during single-unit operation" shows each control and mechanicals supported in single-unit operation. Descriptions of each of the controls and mechanicals are given in Table 3-1.

In single-unit operation, the LED (red) 7SEG LED, and buzzer are activated by controlling the SW3, SW4, SW5, volume SW, and temperature sensor on the board.

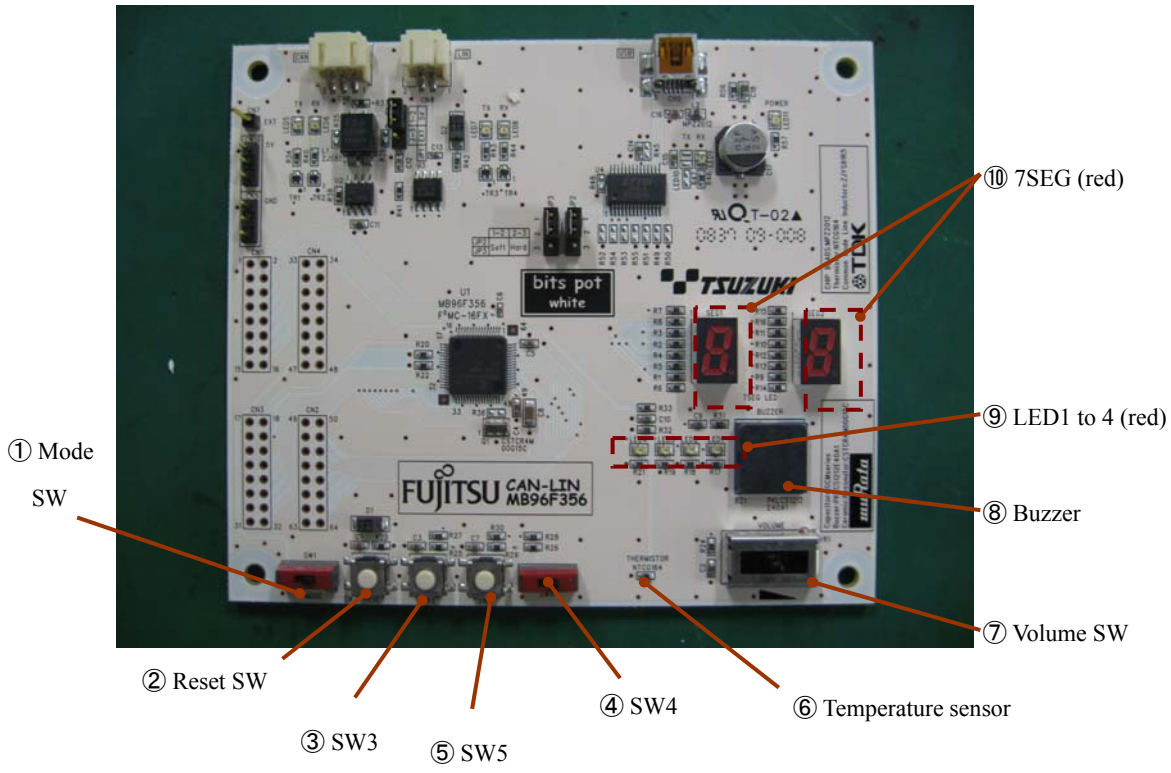


Figure 3-1 Controls and operations during single-unit operation

No.	Name	Function	Description
①	Mode SW	Control	Switches between PROG mode and RUN mode. PROG: Write a programs RUN: Run the program
②	Reset SW	Control	Resets the MCU when pressed.
③	SW3	Control	Lights up all of LED1, 2 and the 7SEG when pressed.
④	SW4	Control	Selects the operation mode. Left: Enables operation of SW3, SW5, and the volume SW, and LEDs 1 to 2, 7SEG and buzzer operate. Right: The temperature sensor is enabled and the temperature sensor information is displayed on the 7SEG LEDs.
⑤	SW5	Control	Switch the buzzer output on/off each time it is pressed. The sound is changed by controlling the volume SW.
⑥	Temperature sensor	Control	The temperature sensor information is displayed on the 7SEG LEDs. The 7SEG display changes every 5°C.
⑦	Volume SW	Control	Changes the 7SEG display and the buzzer sound.
⑧	Buzzer	Mechanical	Produces a sound when SW5 is pressed.
⑨	LED (red)	Mechanical	Lights up when SW3 is pressed.
⑩	7SEG (red)	Mechanical	Displays according to SW3, volume SW, and temperature sensor.

Table 3-1 Single-unit operation/Descriptions of the controls and mechanicals

3.2 CAN communication operation (CAN communication operation with the bits pot red)

"Figure 3-2 CAN communication operation/Controls and mechanicals" shows the controls and mechanicals, and "Table 3-2 CAN communication operation/Descriptions of the controls and mechanicals" provides descriptions about them. The bits pot red performs CAN communication, controls the motor mounted on the bits pot red, and displays the motor rotation information and the information received from temperature sensor.

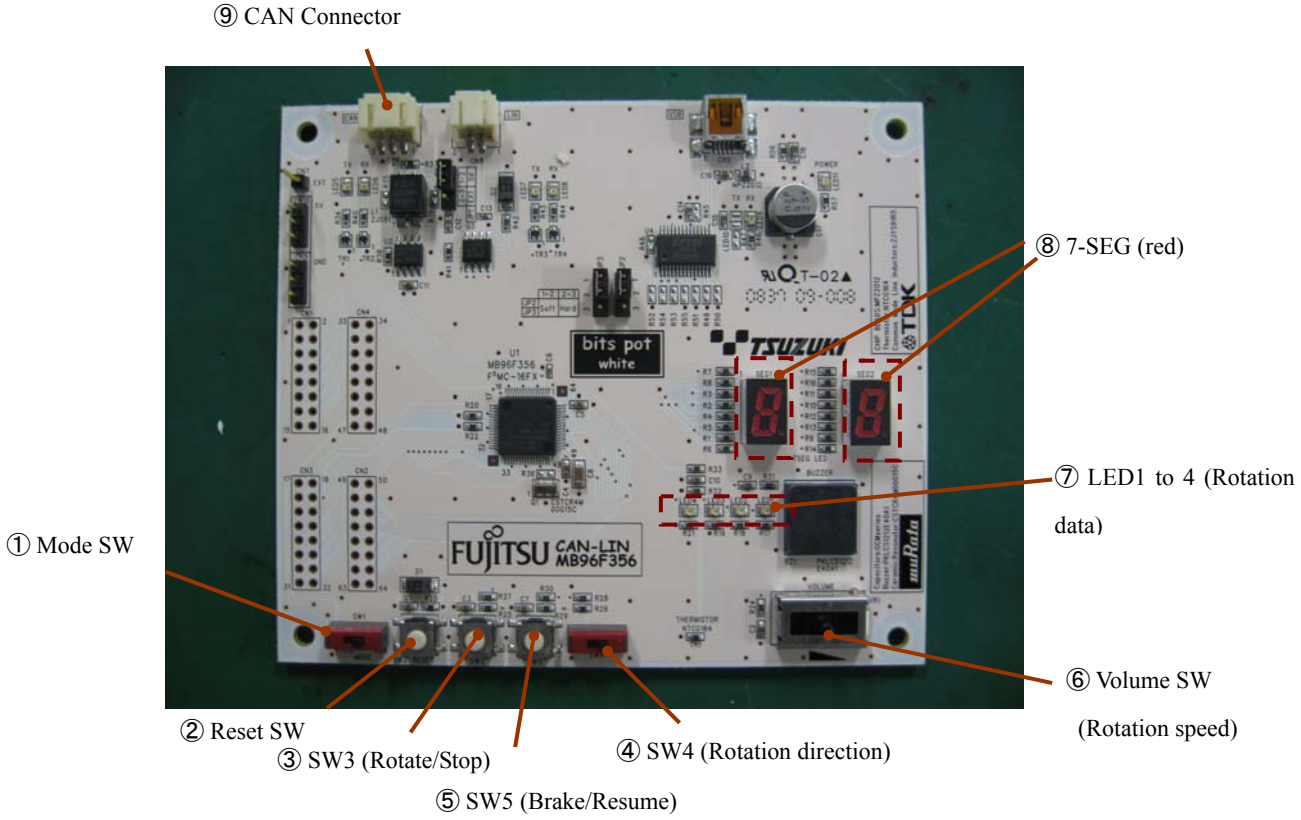


Figure 3-2 CAN communication operation/Controls and mechanicals

No.	Name	Function	Description
①	Mode SW	Control	Switches between PROG mode and RUN mode. PROG: Write a program RUN: Run the programs
②	Reset SW	Control	Resets the MCU when pressed.
③	SW3	Control	Rotates/stops the motor in turn when pressed. The motor rotates if it is stopped and stops if it is rotating when this switch is pressed.
④	SW4	Control	Selects the direction of the motor rotation. Right: The motor rotates clockwise. Left: The motor rotates counterclockwise.
⑤	SW5	Control	Brakes/resumes the motor in turn when pressed. The temperature measurement mode transition command is issued when pressed for a long time.
⑥	Volume SW	Control	Changes the rotation speed of the motor.
⑦	LED (red)	Display	Indicates motor rotation information.
⑧	7SEG (red)	Display	Displays temperature information acquired while in temperature measurement mode.
⑨	CAN connector	Mechanical	During actual operation, needs to be connected by wire for CAN communication with the bits pot red.

Table 3-2 CAN communication operation/Descriptions of the controls and mechanicals

3.3 LIN communication operation (LIN communication operation with the bits pot yellow)

"Figure 3-3 LIN communication operation/Controls and mechanicals" shows the controls and mechanicals, and "Table 3-3 LIN communication operation/Descriptions of the controls and mechanicals" provides descriptions about them.

Performs LIN communication with the bits pot yellow. This starter kit operates as the LIN master and sends the operation mode using the ID field. This starter kit and the bits pot yellow outputs to the LED (red), 7SEG display, and buzzer on the starter kit based on the SW operations, temperature sensor, and volume SW of the starter kit. Furthermore, if an error occurs during LIN communication, the buzzer is output (Note 1).

Note 1: If the sample program for LIN communication is operated while not communicating with the bits pot yellow, this is treated as a LIN communication error, and the starter kit outputs a buzzer sound.

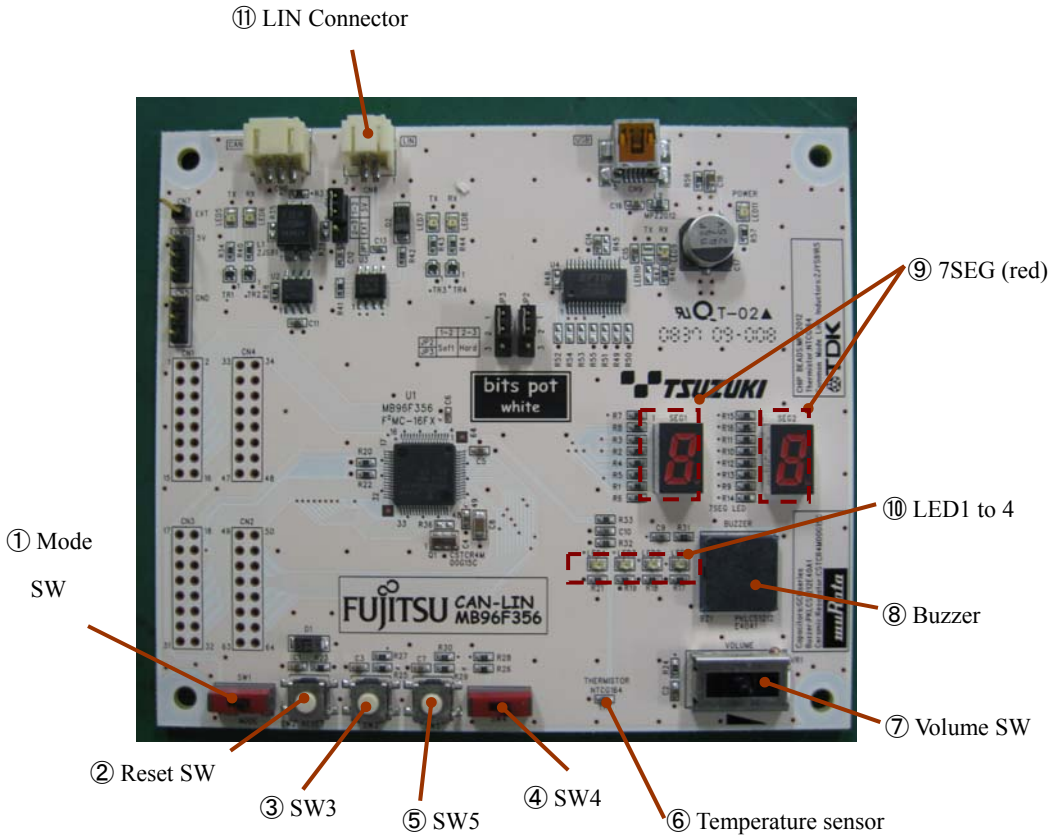


Figure 3-3 LIN communication operation/Controls and mechanicals

Table 3-3 LIN communication operation/Descriptions of the controls and mechanicals

No.	Name	Function	Description
①	Mode SW	Control	Switches between RPG mode and RUN mode. PROG: Write a program RUN: Run the program
②	Reset SW	Control	Resets the MCU when pressed.
③	SW3	Control	If this is press when SW4 is on the left, the currently displayed values of the LEDs and 7SEG display count up, and the bits pot yellow LEDs also counts up. When SW4 is on the right, only the LEDs of the starter kit count up.
④	SW4	Control	Selects the operation mode. Left: The LED, 7SEG display, and LEDs on the bits pot yellow operate when SW3 or SW5 are pressed. Right: Only the LEDs of the starter kit operate when SW3 or SW5 are pressed. The 7SEG display and buzzer operate based on each SW, volume SW, and the temperature sensor on the board.
⑤	SW5	Control	If this is pressed when SW4 is on the left, the currently displayed values of the LEDs and 7SEG display count down, and the bits pot yellow LEDs also counts down. When SW4 is on the right, only the LEDs of the starter kit count down.
⑥	Temperature sensor	Control	Sends the temperature sensor temperature when SW4 is on the right.
⑦	Volume SW	Control	When SW4 is on the left, sends volume SW information for the starter kit.
⑧	Buzzer	Mechanical	When SW4 is on the right, outputs a buzzer sound. Furthermore, the sound that is output is changed by controlling the volume SW on the bits pot yellow. The buzzer also sounds if an error occurs during LIN communication.
⑨	7SEG (red)	Mechanical	When SW4 is on the left, counts up or counts down synchronized with the LEDs when SW3 or SW5 are pressed. When SW4 is on the right, this changes by 5°C based on

			the temperature sensor temperature.
⑩	LED (red)	Mechanical	When SW4 is on the left, counts up or counts down synchronized with the 7SEG display when SW3 or SW5 are pressed. When SW4 is on the right, only these LEDs are controlled.

4 Try to implement single-unit operation

4.1 Overview of single-unit operation

The board operates as a single-unit board controlled by the switches (SW3, SW4, SW5, volume SW) and temperature sensor as shown below.

4.1.1 Controlling the SW inputs to light up the LEDs

The board is fitted with SW3 as shown in “Figure 4-1 Switches when the board is in single-unit operation” which is connected to pins of the microcontroller.

This section describes how the microcontroller detects the state of SW3 when SW3 is operated in order to turn the LED on and off.

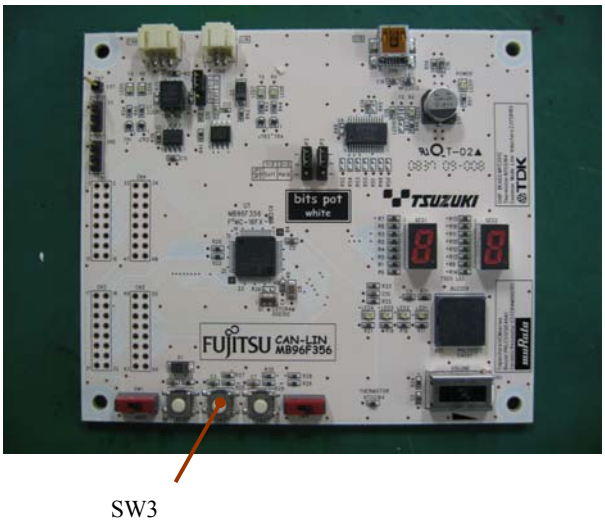


Figure 4-1 Switches when the board is in single-unit operation

First, a diagram (schematic) of the connections between SW3 and the microcontroller pins on the board is shown in Figure 4-2 Connection configuration between SW3 and the microcontroller pins (schematic diagram)”.

On the board, the SW3 is connected to the P05_5 pin, which is a general-purpose I/O port of the microcontroller. When SW3 is not pressed (OFF), the voltage on pin P05_5 of the microcontroller is Vcc (5V) and the input is high. Furthermore, when SW is pressed (ON), the voltage to pin P05_5 is GND and the input to the P05_5 pin is low. The input state of pin P05_5 thus changes depending on the control of When SW3 is OFF

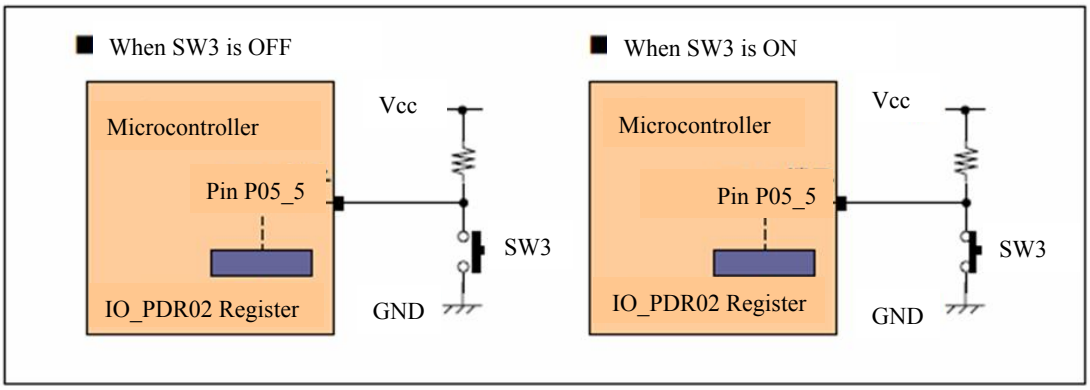


Figure 4-2 Connection configuration between SW3 and the microcontroller pins (schematic diagram)

This change in the state of the pin can be detected by the program running on the microcontroller. In the microcontroller that is mounted on the board, the state of pin P05_5 can be determined from the value of the PDR register (IO_PDR05) for the I/O port within the microcontroller.

The register is memory that stores the microcontroller control state and operating state. The CPU and peripheral functions can be controlled by writing data to and reading data from the registers. Furthermore, peripheral functions refer to functions such as I/O ports, timers, and A/D converters that are built into the microcontroller.

The values of registers can be read using instructions from the microcontroller program. In other words, the microcontroller program can determine the control state of SW3 by reading the value of this PDR register (IO_PDR5). The values indicating the state of the pin are “1” for high, and “0” for low. As a result, the state of the P05_5 pin, and therefore the control state of SW3, can be viewed by reading the value of the PDR register (IO_PDR05).

However, when using the PDR register (IO_PDR05), it is necessary to set whether to use as an input or an output. This register is called the DDR register (IO_DDR05).

In this case, P05_5 is used as an “input” pin in order to sample the SW3 input signal. As a result, “0” needs to be written to the DDR register (IO_DDR05) corresponding to the PDR register (IO_PDR05).

Next, for the LED control, pins P02_0 to P02_3 of the microcontroller are connected to LED 1 to 4. In this case, the specifications are to set the PDR register (IO_PDR02) to low (“0”) to turn the

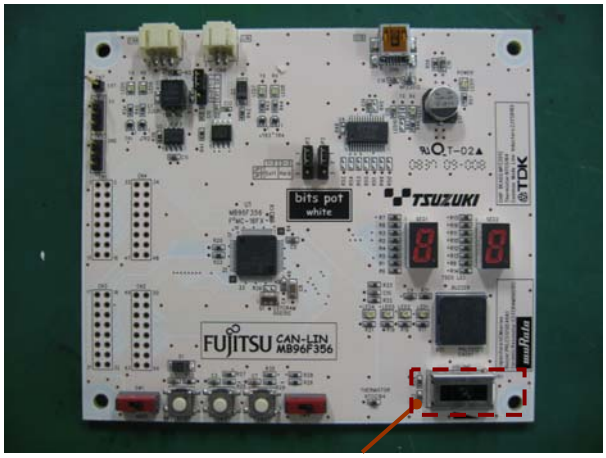
LED on, and set the register to high (“1”) to turn the LED off. Because the P02_0 to P02_3 pins therefore need to be set as “output” pins, “1” needs to be written to the corresponding DDR register (IO_DDR02).

Summarizing the above, the microcontroller program performs the processing to detect the SW control state and turn the LEDs on and off by writing “0” to IO_DDR05 to set pin P05_5 as an input, and reading the value of IO_PDR05 corresponding to the P05_5 pin. Furthermore, the program writes “1” to IO_DDR02 to set pins P02_0 to P02_3 as outputs and writes values to IO_PDR02 corresponding to pins P02_0 to P02_3.

4.1.2 Changing the buzzer sound using the volume SW

This section introduces the process that converts the analog signal into digital using an A/D converter by operating the volume SW, reads the signal internally as a digital signal, and changes the buzzer sound according to changes in the digital signal. The A/D converter is a function that takes an analog value, divides it within specifications based on a set of rules, and converts it to a digital value. Furthermore, this function is built into the microcontroller, and the conversion process is called A/D conversion.

In the board, the voltage value applied to the analog pin for the A/D converter can be controlled by using the volume SW. The analog signal is input to the microcontroller by using this knob. The input analog signal is converted into a digital signal by the A/D converter and is then processed by the microcontroller.



Volume SW

Figure 4-3 Volume SW when the board is in single-chip operation

Thus, in order to explain the mechanism of the volume SW, the symbol for a variable resistor is shown in “Figure 4-4 Variable resistor”.

In fact, volume SW is actually a variable resistor. The board has the circuit configuration shown

in “Figure 4-5 Connection configuration of the volume SW (voltage adjustment knob) (schematic diagram)” which changes the applied voltage value depending on this knob, and applies this voltage to the pin that performs A/D conversion. The applied voltage is converted to digital in 1024 levels, and can be handled as an internal signal.

Figure 4-4 Variable resistor

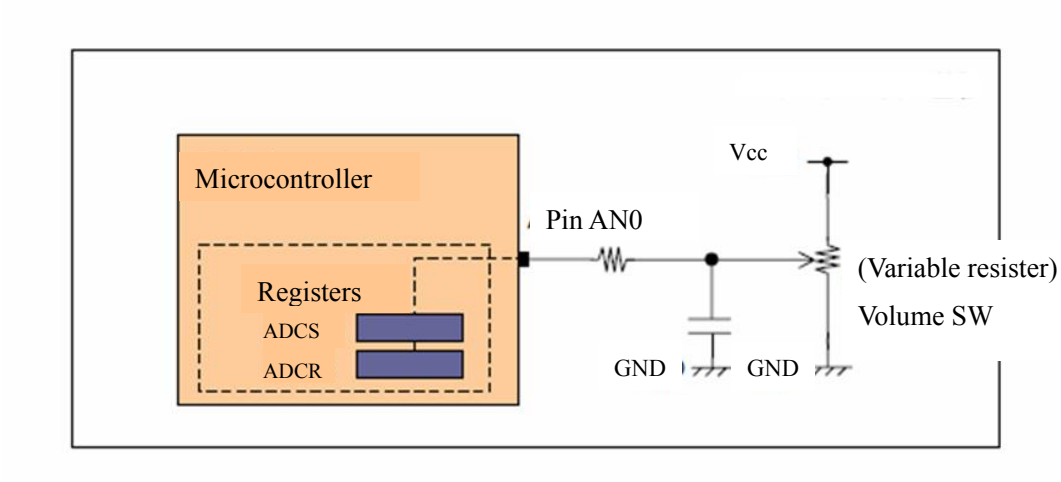


Figure 4-5 Connection configuration of the volume SW (voltage adjustment knob) (schematic diagram)

The method for changing the buzzer sound is described next.

The buzzer mounted on the board is an external-drive buzzer, which is able to change the sound because it generates a sound with an arbitrary frequency for the given voltage. A pulse wave is therefore output from the microcontroller to produce an arbitrary frequency.

Basically, the PPG timer built into the microcontroller is used to produce output by setting the “H” and “L” widths of the pulses. Therefore, if the output pulse is changed, the buzzer sound also changes. (In the program that runs on the board, the PPG timer output pulse is already configured.)

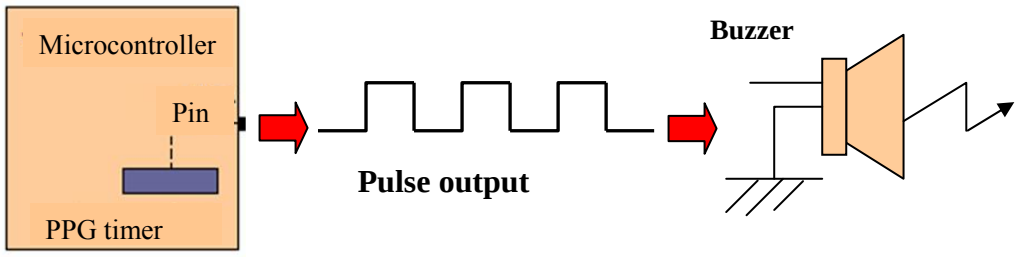


Figure 4-6 Sound produced by the external-drive buzzer (Schematic diagram)

Summarizing the above, if the analog value (voltage value) that is changed by controlling the volume SW is converted into a digital value internally within the microcontroller and a pulse corresponding to the digital value is output to the buzzer from the PPG timer, then the buzzer sound is changed by controlling the volume SW.

4.1.3 7SEG display by temperature sensor operation

The board has a built-in temperature sensor. This section describes how the 7SEG display is controlled by using the temperature sensor.

The temperature sensor is a sensor that detects changes in temperature. Put simply, it is a thermometer for measuring the temperature. Although there are a variety of ways to measure temperature, the temperature sensor mounted on this board is a thermistor. A thermistor is temperature sensor where the resistance changes depending on the temperature by using a resistance element that employs the temperature characteristics of semiconductors.

A diagram of the circuitry around the temperature sensor on the board (schematic diagram) is shown in “Figure 4-7 Circuit diagram around the temperature sensor (schematic diagram)”. When the resistance value of the temperature sensor changes, the input voltage to the A/D converter of the microcontroller also changed due to this circuitry.

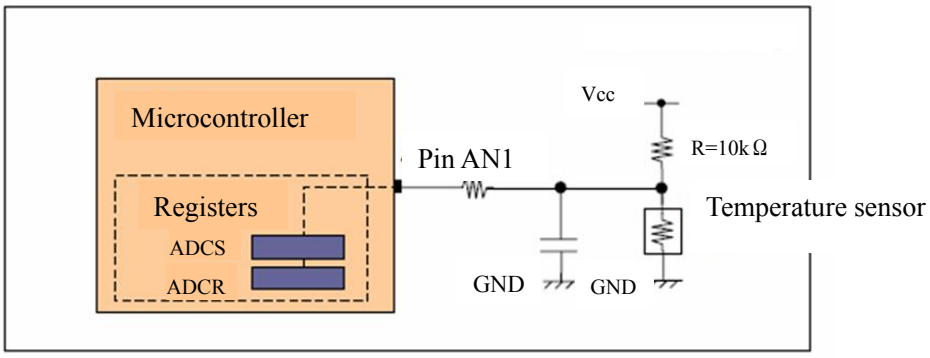


Figure 4-7 Circuit diagram around the temperature sensor (schematic diagram)

Next, the 7SEG display is basically the same as turning on and off the LEDs.
Each segment (each single thin, long display element) in the 7SEG display is treated the same as a single LED, and the display method for displaying numbers using the 7SEG display is to write a value (“0” for on, “1” for off) to the PDR register corresponding to each segment.

Summarizing the above, the input voltage is changed by changes in the resistance of the temperature sensor. The 7SEG display can then change in response to the temperature of the sensor by performing A/D conversion of the input voltage using the A/D converter of the microcontroller and changing the 7SEG display according to the A/D converted digital value.

4.1.4 Sample Programs

The flowcharts of the sample programs are shown in the following diagrams ("Figure 4-8 Flowchart of main routine" to "Figure 4-11 Flowchart of SW5 operation").

First, the microcontroller internal operating clock is initialized, and the LED and 7SEG display pins are initialized. Next, the external interrupts corresponding to the SW operation are initialized. After this, the volume SW side and temperature sensor side are divided by the state of SW4, each of the A/D converters are initialized, and A/D conversion processing is performed to display the 7SEG display. Furthermore, if an interrupt occurs at this time due to the operation of SW3 or SW5, all of the LEDs and the 7SEG display are turned on, or the buzzer is sounded, or these are stopped.

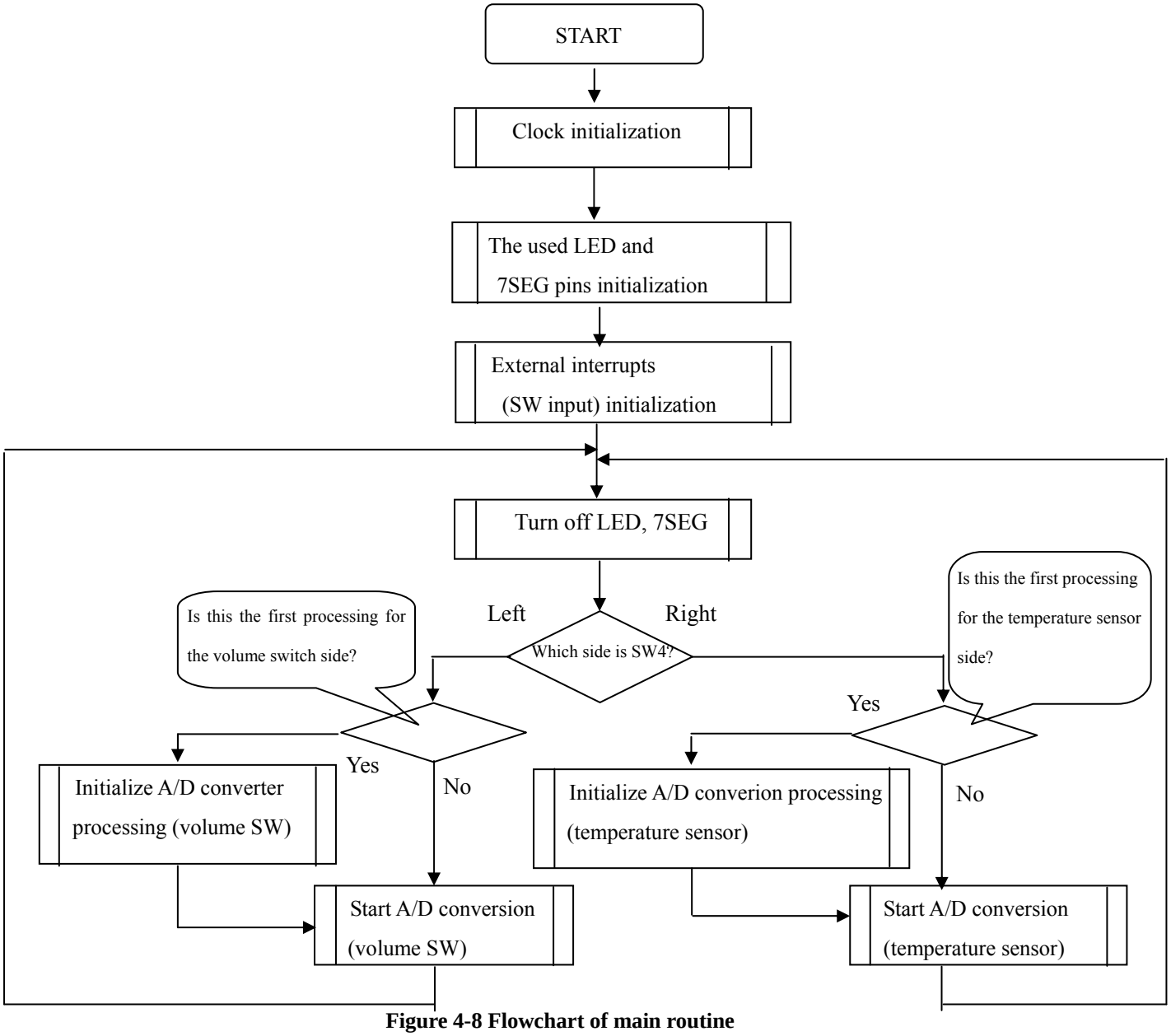


Figure 4-8 Flowchart of main routine

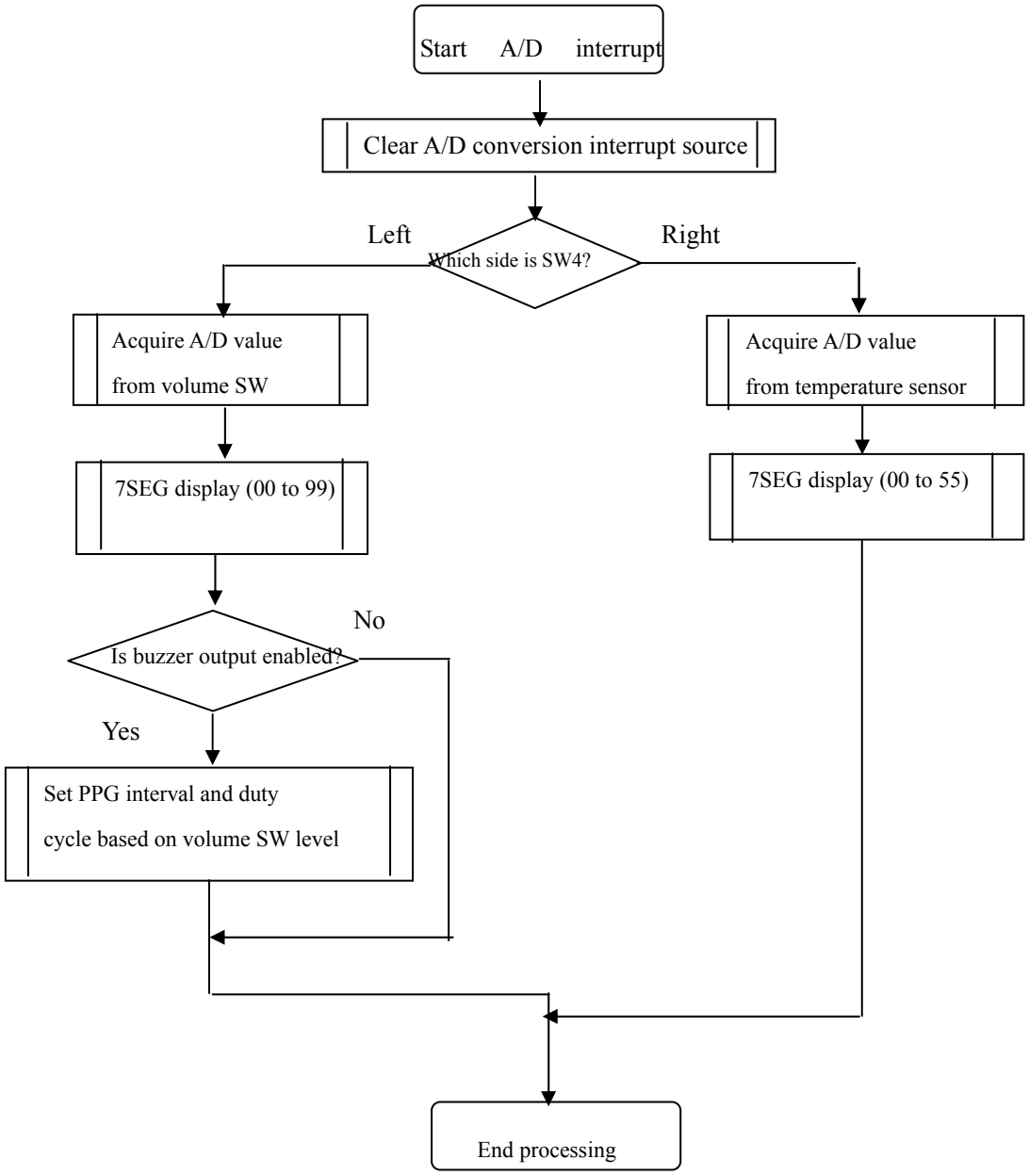


Figure 4-9 Flowchart of A/D conversion processing of the volume SW and temperature sensor

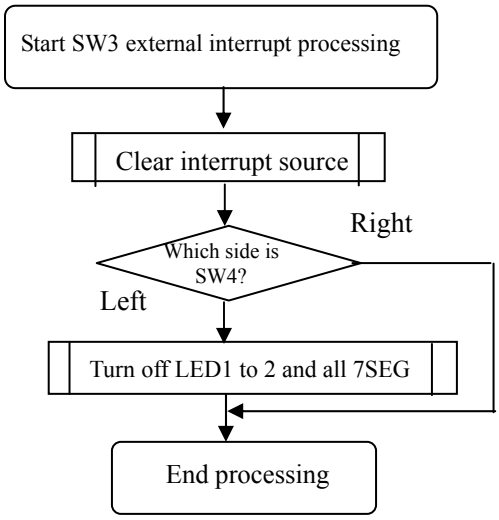


Figure 4-10 Flowchart of SW3 operation

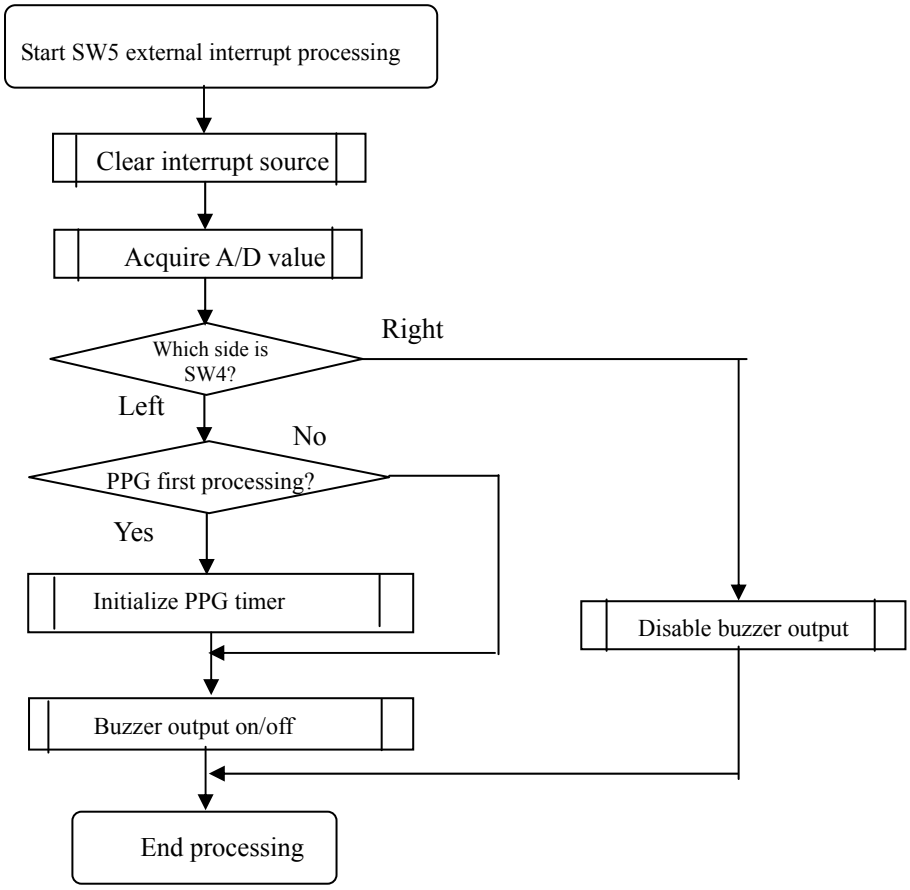


Figure 4-11 Flowchart of SW5 operation

Next, we will take a look at an actual program.

Check the following folder in the sample programs. There are several files stored in this folder. Among these, we will first open MAIN.C.

¥bitspot_white_SampleProgram¥single_operation

First there is the main function shown in “Figure 4-12 main routine program (MAIN.C)”. Within this function, processing is performed to “the microcontroller internal clock initialization", “LED and 7SEG pins initialization”, “External interrupts initialization”, and “Infinite loop that performs A/D conversion processing”.

```
void main(void)
{
        (omitted)
    initial_clock();        ←internal clock initializaton

    IO_PDR02.bit.P3=1;
    IO_DDR02.bit.D23=1;     ←LED pin initializaton


     (omitted)

     Ext_initial();   ← External interrupt initializaton


     IO_PDR00.byte = 0xff;
     IO_DDR00.byte = 0xff;   ← 7SEG pins initializaton

      (omitted)

     while (1)
     {          ← Loop initializaton

       if(IO_PDR05.bit.P6==1)        ← Temperature sensor (A/D conversion)
            {                        processing
                ADC_initial_tmp();

                (omitted)

                ADC_start();

       }
       else if(IO_PDR05.bit.P6==0)   ← Volume switch (A/D conversion)
            {                        processing
                ADC_initial_bsw();

                (omitted)

                ADC_start();
            }

    }/* while loop */
}
```

Figure 4-12 main routine program (MAIN.C)

Next, the program for A/D conversion processing is shown in “Figure 4-13 A/D conversion program for volume SW and temperature sensor operation (ADC.C)”.
On the temperature sensor side and volume SW side, the digital values are obtained and divided up to change the 7SEG display or buzzer sound by the A/D conversion interrupt routine.

```
__interrupt void interrupt_AD(void)
{
        IO_ADCSH.bit.INT=0;   ← Clear A/D conversion interrupt source

        if(IO_PDR05.bit.P6==1)   ← Temperature sensor side
        {
                ad_data_tmp=IO_ADCRLH.DATA8;   ← Acquire temperature sensor A/D value

                if(ad_data_tmp>214){   ← Separate processing by A/D value

                IO_PDR00.bit.P0=0; /* a */   ← 7SEG display
                IO_PDR00.bit.P1=0; /* b */
                IO_PDR00.bit.P2=0; /* c */
                (omitted);

                }
                else if(ad_data_tmp>170){

                (omitted)
        }
        else if(IO_PDR05.bit.P6==0)   ← Volume SW side

        {
                ad_data=IO_ADCRLH.DATA8;   ← Acquire volume SW A/D value

                if(ad_data>215){   ← Separate processing by A/D value

                IO_PDR00.bit.P0=0; /* a */   ← 7SEG display
                IO_PDR00.bit.P1=0; /* b */
                IO_PDR00.bit.P2=0; /* c */
                (omitted)


                if(IO_PCN1.bit.OE==1){   ← Buzzer output enabled

                  IO_PCSR1 = 56000;   ← Set PPG interval and duty value
                  IO_PDUT1 = 56000/2;

                  }

                }
                 else if(ad_data>191){
                (omitted)
                }
         }
 }
```

Figure 4-13 A/D conversion program for volume SW and temperature sensor operation (ADC.C)

Next, the program for processing the external interrupts from the operation of SW3 and SW5 is shown in “Figure 4-14 Program for SW3 operation (Ext_int.c)“ and “Figure 4-15 Program for SW5 operation (Ext_int.c)”.

All of LED 1 and 2 and the 7SEG display are turned on by pressing SW3, the buzzer sounds by pressing SW5, or these actions are stopped.

```
__interrupt void interrupt_sw3(void)
{
        IO_EIRR0.bit.ER0=0;     ← Clear external interrupt source

        if(IO_PDR05.bit.P6==0)      ← Enable SW3 when on volume SW side
        {

        IO_PDR02.bit.P3=~IO_PDR02.bit.P3;      ← Turn on LED 1 and 2
        IO_PDR02.bit.P2=~IO_PDR02.bit.P2

          if((IO_PDR02.bit.P3==0)&(IO_PDR02.bit.P2==0))
            {
            IO_PDR00.byte=0x00;     ← Turn on all 7SEG
            IO_PDR03.byte=0x00;
                    (omitted)
            }
        }
    }
```

Figure 4-14 Program for SW3 operation (Ext_int.c)

```
__interrupt void interrupt_sw5(void)
{
        IO_EIRR0.bit.ER2=0;     ← Clear external interrupt source

        sw5_ad_data=IO_ADCRLH.DATA8;     ← Acquire A/D value

        if(IO_PDR05.bit.P6==0)     ← Enable SW5 when on volume SW side
        {

          if(sw5_flag==0)
                {

                PPG_initial();  ← PPG timer initialization
                }

                IO_PCN1.bit.OE=~IO_PCN1.bit.OE;     ← Sound or stop buzzer sound

                }

            else if(IO_PDR05.bit.P6==1){     ← When temperature sensor side

              (omitted)

              IO_PCN1.bit.OE=0;      ← Set buzzer output disabled

               (omitted)
              } }
```

Figure 4-15 Program for SW5 operation (Ext_int.c)

5 Try to use CAN communication

Communication is to send/receive information. There are various types of communication such as utterance/hearing of spoken words, writing/reading of written letters, and electrical transmission of information.

Among them, there are various standards for communication based on electrical transmission. This chapter describes a communication standard called CAN.

CAN is a global standard of the ISO (International Organization for Standardization).

5.1 What is CAN?

CAN stands for Controller Area Network, which is an on-board LAN specification proposed by Bosch in Germany. It is the most popular on-board control LAN and used in various parts of a vehicle as shown in “Figure 5-1 Example of on-board CAN application”.

It is now also used not only in vehicles but also in many industries.

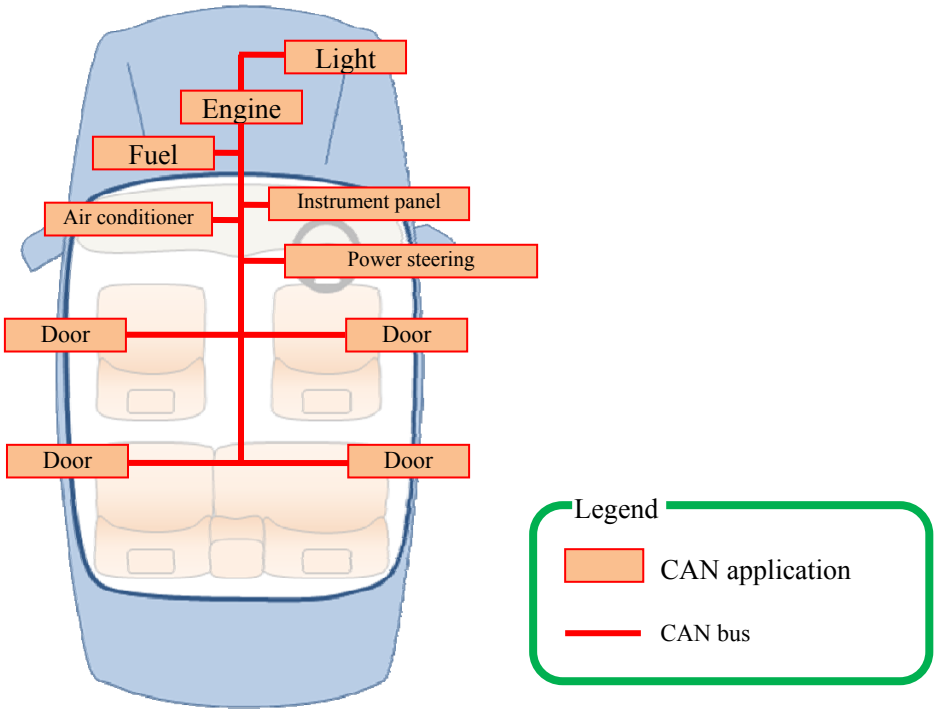


Figure 5-2 Example of on-board CAN application

The features of CAN can be classified into the following five points.

1. Multi-master communication

   CAN employs the multi-master system in which each node is allowed to start communication as desired. The timing of a start of communication is occurrence of an event. The word “event” mentioned here indicates an occasion at which a node needs to start communication.

   CAN avoid conflicts in communication through mediation with node signals if more than one event occurs on nodes simultaneously. This mediation is called arbitration.

2. Bus-type topology

   The CAN topology is the bus type. The maximum number of nodes depends on the communication speed; in the case of 1M bits/sec, up to 30 nodes are allowed. This is specified as a regulation.

3. Differential transmission system

   Taking account of influence from noise on the transmission paths, CAN employs the differential transmission system in which the voltage difference between two signal lines is used to determine “0”/”1”. The signal lines are respectively called CANH and CANL and the voltage difference between them is used to determine the bus level. The differential is used to determine logical “0”/”1”. As shown in “Figure 5-3 CAN bus signal levels”, the bus status of logical “0” is called dominant and the bus status of logical “1” is called recessive. The communicable distance depends on the communication speed; in the case of 1M bits/sec, up to 40 m is allowed. This is also specified by a regulation.

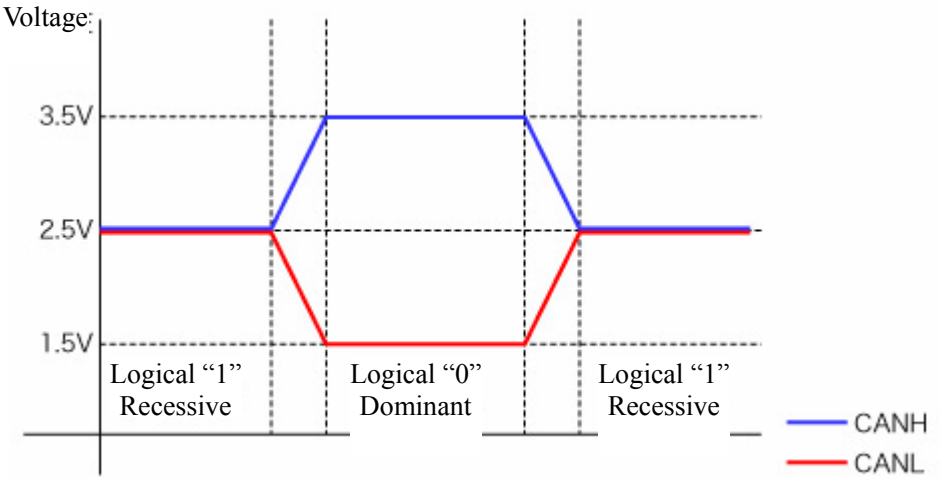


Figure 5-3 CAN bus signal levels

4. High-speed version and low-speed version

   There are two CAN specifications with different communication speeds. One of them is High-speed-CAN. High-speed-CAN is standardized as ISO11898 and its maximum and minimum communication speeds are 1 Mbits/sec and 125 kbits/sec. The other is Low-speed-CAN. Low-speed-CAN is standardized as ISO11519 and its maximum communication rate is 125 kbits/sec. The communication speeds currently popular are, in order of rates, 500 kbits/sec, 250 kbits/sec, 125 kbits/sec, 83.3 kbits/sec, 33.3 kbits/sec and so forth.

5. Node control with error counters against errors

   CAN supports five types of error detection. Each node has error counters. If an error occurs, either counter is increased by a specified count. On the contrary, when communication is successful, the counter is decreased by a specified constant. The communication status of each node is determined by the values of the error counters. This mechanism serves to limit communication by node.

5.2 CAN specifications

This section provides brief descriptions of the CAN specifications.

For more information about the specifications, access the web site of the CAN promoting organization CiA (CAN in Automation) (**http://www.can-cia.org/**) and make a registration; you can get the specifications.

5.2.1 CAN frame configurations

This section describes frames that are the fundamental communication unit of CAN.

CAN provides four types of frames, which are respectively named the data frame, remote frame, error frame, and overload frame as shown in “Figure 5-4 CAN frame configurations”.

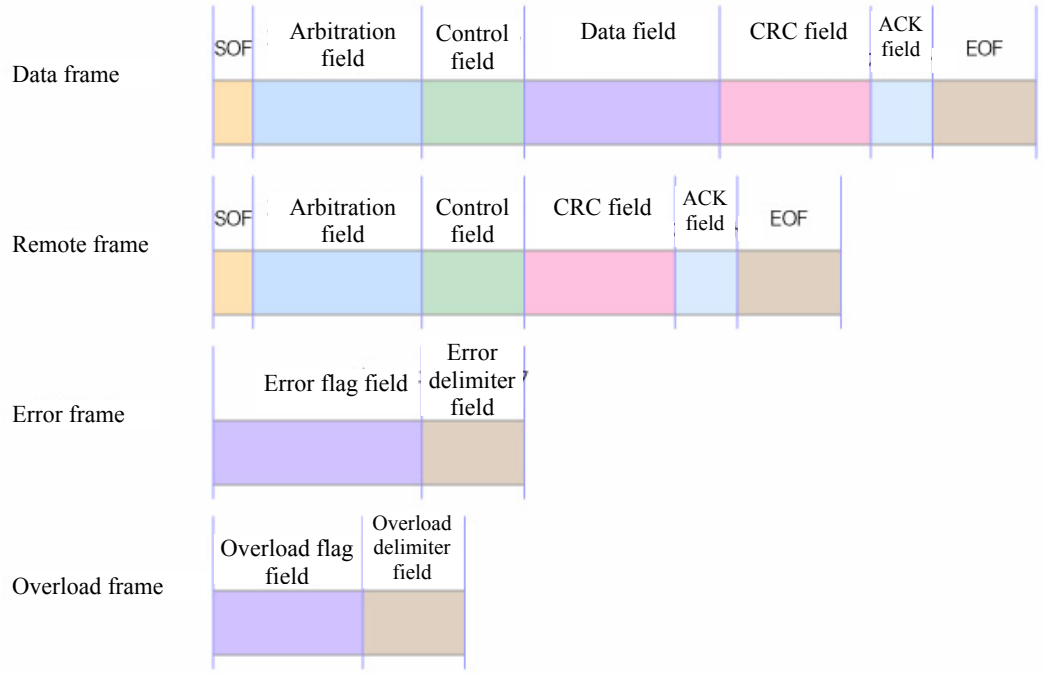


Figure 5-4 CAN frame configurations

1. Data frame

   Transfer format for data transmit. It consists of seven fields.

Field name	Description
Start of frame (SOF)	1-bit field containing “0” that indicates the start of a data frame
Arbitration field	Field that determines the priority of the data. This field is also called the ID field and there are two types of format; standard format and extended format. The standard format is 12 bits and extended format is 32 bits.
Control field	6-bit field that indicates the length of the data field.
Data field	0-byte to 8-byte field that stores real data.
CRC field	16-bit field that serves to allow a check of the transmitted frame validity.
ACK field	2-bit field that is used to notify of successful reception.
End of frame (EOF)	7-bit field containing “1” that indicates the end of the data frame.

Table 5-1 Data frame structure

2. Remote frame

   Usually, in CAN, a form of transmit of communication information to a node is generally used, but it is also allowed to request a specific node to transmit specific data. For this purpose, the remote frame is available.

   The remote frame has almost the same configuration with the data frame; it consists of six fields except the data field. The control field of the remote frame indicates the length of the data field for the requested data.

3. Error frame

   Transfer format immediately sent on error detection on a node. The error frame consists of two fields.

Field name	Description
Error flag	This is a 6 to 12 bit field that indicates the error type.
Error delimiter	This is a field where the 8th bit is “1” to indicate the end of the error frame.

Table 5-2 Error frame structure

4. Overload frame

   Transfer format sent to indicate that the node is in unreceivable status.

Field name	Description
Overload flag	6-bit to 12-bit field that indicates the type of overload.
Overload delimiter	8-bit field containing “1” that indicates the end of the error frame.

Table 5-3 Overload frame structure

5.2.2 Arbitration

CAN employs the multi-master communication system, so any node can start communication. But, the number of communication sessions actually allowed on one bus is only one. Each node is cyclically checking whether the bus is the status of transmission. When there is no transmission on the bus, communication is started, but if more than one node starts transmission, they conflict. Against this, CAN performs arbitration to give priority to one with a lower ID for transmission. This section describes the arbitration.

The arbitration is carried out by comparison between the ID and the bus level by bit as shown in "Figure 5-5 Operation of the arbitration". Bit 10 to 7 of Node 1 and Node 2 are the same as the bus level. This indicates that both Node 1 and Node 2 are transmitting signals. But, Bit 6 of Node 1 is set to "0" and that of Node 2 is set to "1". The bus level is "0", so Node 2 recognizes that the frame is not of its own communication and stops the transmission immediately. Node 1 keeps on transmitting. After Node 1 ends its communication, Node 2 resumes transmission.

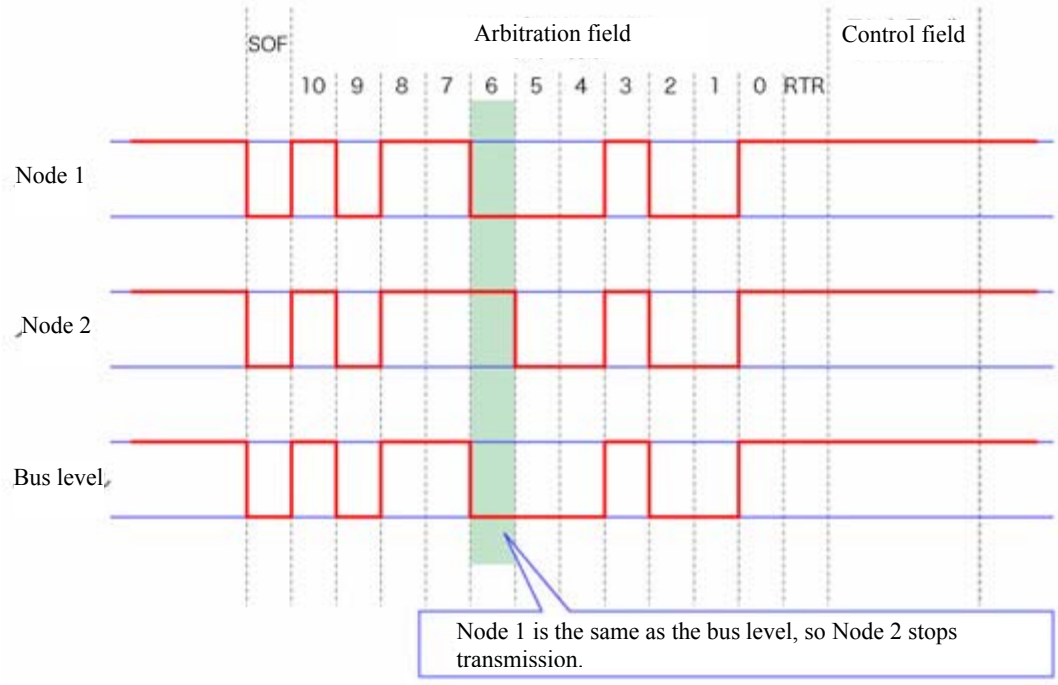


Figure 5-5 Operation of the arbitration

The bus status is determined according to the logical product of IDs, so "0" is prior to "1". This means that a lower ID takes priority.

A practical communication flow shown in “Figure 5-6 Example of arbitration among nodes” is as described below. First, Node 1 and Node 2 starts transmission simultaneously. The arbitration results in giving priority to the Node 1 transmission with a lower ID. After Node 1 ends its transmission, Node 2 resumes transmission.

After that, Node 1 and Node 3 starts transmission simultaneously. The arbitration is also performed and results in giving priority to the Node 3 transmission. After that, Node 4 starts transmission as soon as Node 3 ends its transmission. On this occasion, arbitration between Node 1 retransmission and Node 4 transmission is performed. This results in transmission in order of Node 4 to Node 1. That is, setting a lower ID to those of preference allows priorities to be settled for communication.

The ID is assigned by the command, information, and type of transmit data. The ID settings can be configured as desired.

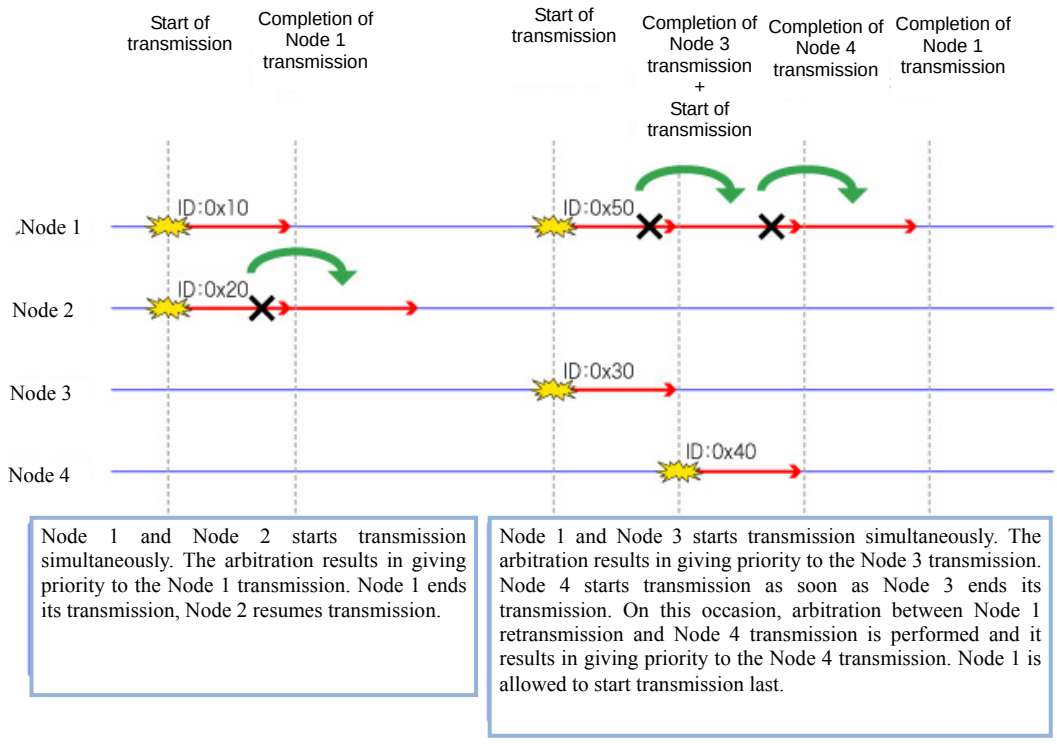


Figure 5-6 Example of arbitration among nodes

5.2.3 Error management

CAN error management is defined in its protocol. Five types of error detection and three types of status are used.

1. Error detection

   As shown in "Table 5-4 Description of the error types", errors that can be detected depends on whether the node is transmitting or receiving.

Table 5-4 Description of the error types

Error type	Transmitting node	Receiving node	Description
Bit error	○	—	Detected if there is a difference between the transmitted data and the bus level.
ACK error	○	—	Detected if an acknowledgement to transmission cannot be obtained.
Stuff error	—	○	Detected if bit stuffing is not applied. Bit stuffing is to set an inverted bit by 5 bits if the number of successive bits with the same level is 5 or more. This prevents bits with the same level from being successive over 6 bits.
CRC error	—	○	Detected if CRC (cyclic redundancy check) fails on the received data.
Format error	—	○	Detected if the received data does not confirm to any of the frame formats.

2. Statuses

   Each node has error counters whose value depends on the status. The error counters of the nodes are named TEC (transmit error count) and REC (receive error count) intending transmission and reception. The three statuses are as described below.

Table 5-5 Three statuses

Status	Description
Error active	The node is normally joining in the bus.
Error passive	The node has frequent errors so it is influencing the bus.
Bus off	The node is disconnected from the bus. To restore to the bus, the bus needs to satisfy the restoration condition.

Transition between the statuses is described below along the example shown in “Figure 5-7 CAN status transition”. The initial status of a node is error active. In this status, occurrence of errors increases the TEC/REC counters.

If either of the TEC/REC counters comes to 127 or higher, the status of the node changes to error passive. In this status, the node remains communicable and the values of the counters decrease whenever a communication session is normally carried out.

When both the TEC/REC counters decrease to 127 or less, the status of the node returns to error active.

If the TEC counter increases after the node comes to error passive and the count comes to 255 or higher, the status of the node changes to bus off.

If the status of the node becomes bus off, the node cannot be restored to error active unless the restoration condition that successive 11-bit recessive is received 128 times is satisfied.

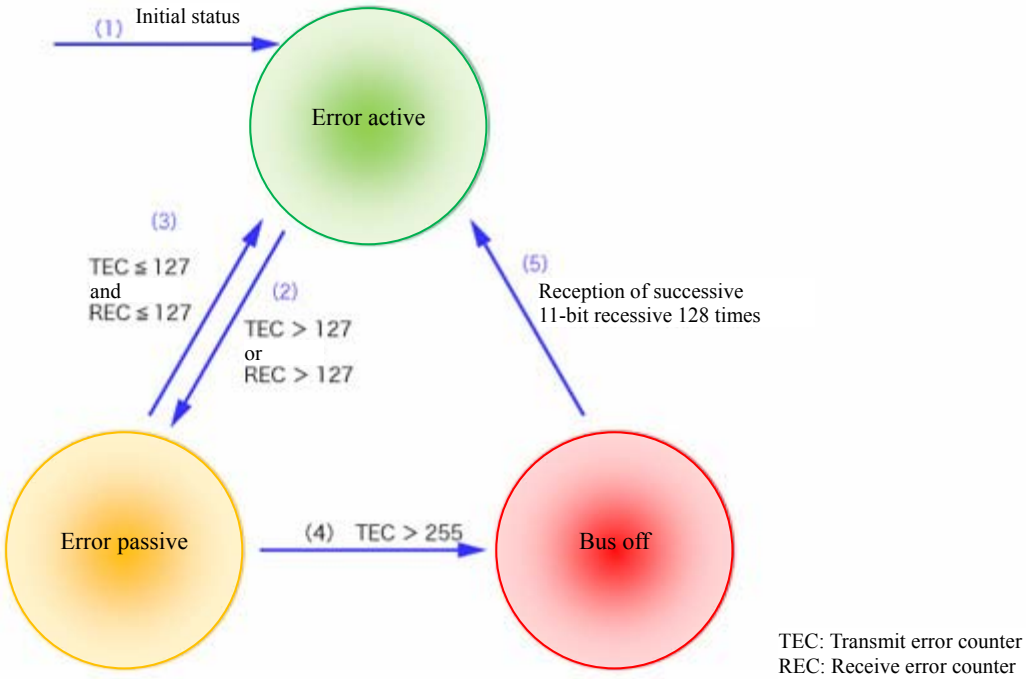


Figure 5-7 CAN status transition

5.3 Using the microcontroller to perform CAN communication

This section describes how to perform practical CAN communication with the microcontroller.

On the board, as shown in “Figure 5-8 CAN circuit”, the microcontroller is connected with the CAN transceiver (MAX3058). TX on the microcontroller is used for transmission and RX is used for reception. Signals transmitted/received are transferred to CAN-High and CAN-Low as the differential signals on the bus through the CAN transceiver.

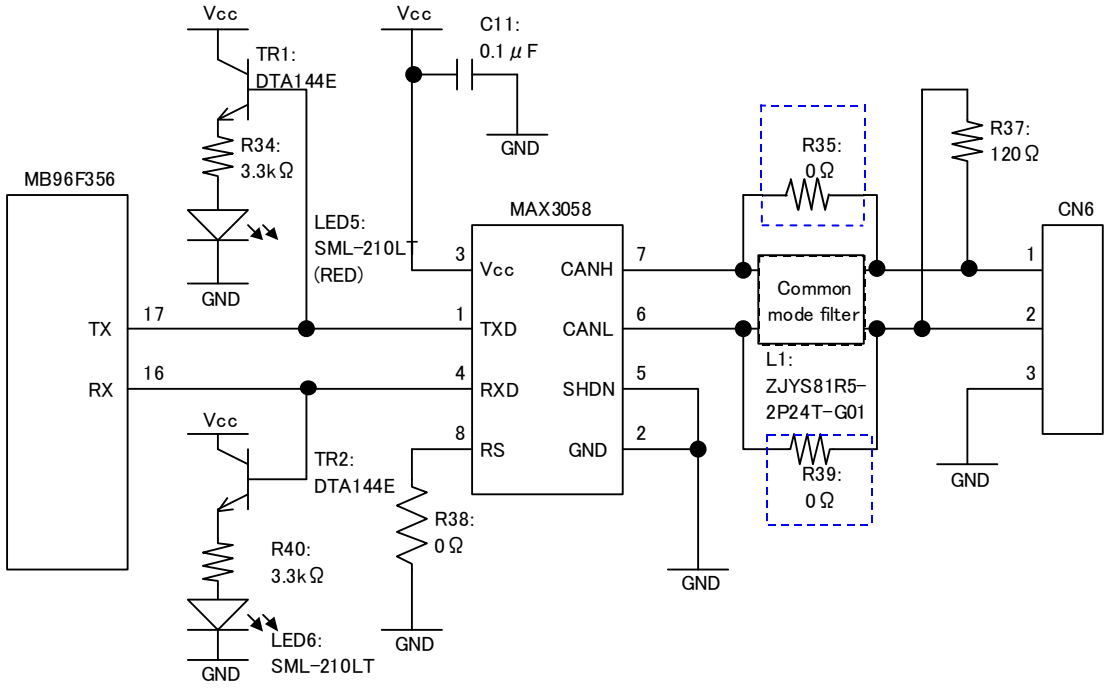


Figure 5-8 CAN circuit

The registers used for entire CAN communication control on the microcontroller are as shown in the following table.

For more information of the registers, refer to the microcontroller hardware manual.

Address	Register name	Abbr.	Access	Initial value	Remarks
Base+0x00	CAN control register	CTRLR	R/W	0x0001	
Base+0x02	CAN status register	STATR	R, R/W	0x0000	Boff, Ewarn, Epass = Read-only RxOk,TxOk,LEC = R/W
Base+0x04	CAN error counter	ERRCNT	R	0x0000	Read-only
Base+0x06	CAN bit timing register	BTR	R/W	0x2301	Writable when Init(CTRLR)=CCE(CTRLR)="1"
Base+0x08	CAN interrupt register	INTR	R	0x0000	Read-only
Base+0x0A	CAN test register	TESTR	R/W	0x0000	Writable when Test(CTRLR)="1" "r" is the CAN_RX level value.
Base+0x0C	CAN prescaler extension register	BRPER	R/W	0x0000	Writable when CCE(CTRLR)="1"

Table 5-4 CAN register list 1

Address	Register name	Abbr.	Access	Initial value	Remarks
Base+0x10	IF1 command request register	IF1CREQ	R, R/W	0x0001	BUSY bit is R/W in Basic mode, and read-only in normal mode Message Number is R/W
Base+0x12	IF1 command mask register	IF1CMSK	R/W	0x0000	
Base+0x14	IF1 mask register 1	IF1MSK1	R/W	0xffff	
Base+0x16	IF1 mask register 2	IF1MSK2	R/W	0xffff	
Base+0x18	IF1 arbitration register 1	IF1ARB1	R/W	0x0000	
Base+0x1A	IF1 arbitration register 2	IF1ARB2	R/W	0x0000	
Base+0x1C	IF1 message control register	IF1MCTR	R/W	0x0000	
Base+0x20	IF1 data register A1	IF1DTA1	R/W	0x0000	
Base+0x22	IF1 data register A2	IF1DTA2	R/W	0x0000	
Base+0x24	IF1 data register B1	IF1DTB1	R/W	0x0000	
Base+0x26	IF1 data register B2	IF1DTB2	R/W	0x0000	
Base+0x40	IF2 command request register	IF2CREQ	R, R/W	0x0001	BUSY bit is R/W in Basic mode, and read-only in normal mode Message Number is R/W
Base+0x42	IF2 command mask register	IF2CMSK	R/W	0x0000	
Base+0x44	IF2 mask register 1	IF2MSK1	R/W	0xffff	
Base+0x46	IF2 mask register 2	IF2MSK2	R/W	0xffff	
Base+0x48	IF2 arbitration register 1	IF2ARB1	R/W	0x0000	
Base+0x4A	IF2 arbitration register 2	IF2ARB2	R/W	0x0000	
Base+0x4C	IF2 message control register	IF2MCTR	R/W	0x0000	
Base+0x50	IF2 data register A1	IF2DTA1	R/W	0x0000	
Base+0x52	IF2 data register A2	IF2DTA2	R/W	0x0000	
Base+0x54	IF2 data register B1	IF2DTB1	R/W	0x0000	
Base+0x56	IF2 data register B2	IF2DTB2	R/W	0x0000	

Table 5-5 CAN register list 2

Address	Register name	Abbr.	Access	Initial value	Remarks
Base+0x80	CAN transmit request register 1	TREQR1	R	0x0000	Read-only
Base+0x82	CAN transmit request register 2	TREQR2	R	0x0000	Read-only
Base+0x90	Data update register 1	NEWDT1	R	0x0000	Read-only
Base+0x92	Data update register 2	NEWDT2	R	0x0000	Read-only
Base+0xA0	CAN interrupt pending register 1	INTPND1	R	0x0000	Read-only
Base+0xA2	CAN interrupt pending register 2	INTPND2	R	0x0000	Read-only
Base+0xB0	CAN message valid register 1	MSGVAL1	R	0x0000	Read-only
Base+0xB2	CAN message valid register 2	MSGVAL2	R	0x0000	Read-only

Table 5-6 CAN register list 3

The steps for initializing CAN in the sample project are introduced simply in the following diagram.

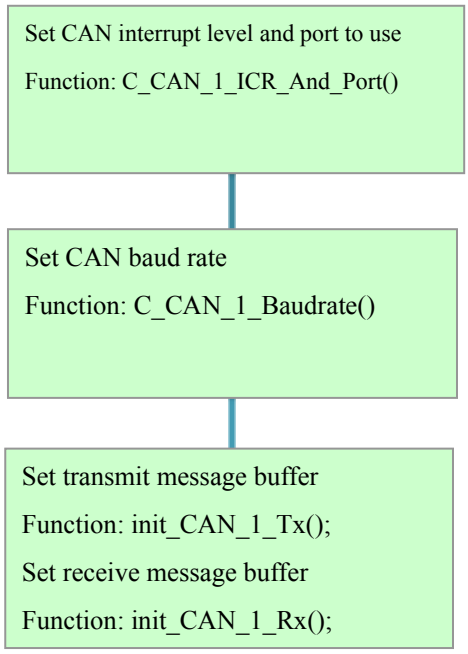


Figure 5-9 Initializing CAN

5.4 Understanding and running the program for CAN communication

This section provides descriptions of the sample program that can serve for practical CAN communication.

5.4.1 CAN communication configuration

"Table 5-7 CAN communication conditions of the sample program" shows the CAN communication conditions of the sample program.

Condition	Value
Communication speed	250 Kbps
CAN clock frequency	16 MHz
Bit time (NBT)	16
Sample point	81.3%
Sync. jump width (SJW)	2
Sample count (SAM)	1
Data length	8 bytes

Table 5-7 CAN communication conditions of the sample program

"Table 5-8 CAN message IDs in the sample program" provides a description of the message IDs used for CAN communication.

ID	Description	Communication direction
0x101	Motor operation start/stop command	Receive
0x102	Motor operation rotation speed/Rotation direction/Brake command	Receive
0x103	Temperature sensor measurement command	Receive
0x201	Motor rotation information	Transmit
0x202	Temperature sensor information	Transmit

Table 5-8 CAN message IDs in the sample program

Details of the IDs are as shown below.

1. ID: 0x101

byte 0	Motor operation instruction
byte 1	Motor rotation direction
byte 2	Motor rotation speed
byte 3	
byte 4	A/D maximum value
byte 5	
byte 6	Reserved
byte 7	Reserved

Field name	Setting value	Remarks
Motor operation command	0: Stop 1: Start	−
Motor rotation direction	0: Clockwise 1: Counterclockwise	−
Motor rotation speed	0 to 65535	The motor rotation speed and A/D maximum value are used for conversion of the speed to a percentage of 0% to 100%.
A/D maximum value	0 to 65535	

2. ID: 0x102

byte 0	Motor rotation direction
byte 1	Brake application
byte 2	Motor rotation speed
byte 3	
byte 4	A/D maximum value
byte 5	
byte 6	Reserved
byte 7	Reserved

Field name	Setting value	Remarks
Motor rotation direction	0: Clockwise 1: Counterclockwise	–
Brake application	0: Brake released 1: Brake applied	–
Motor rotation speed	0 to 65535	The speed is converted to between 0 and 100% using the motor rotation speed and A/D maximum value.
A/D maximum value	0 to 65535	

3. ID: 0x103

byte 0	Temperature
byte 1	Reserved
byte 2	Reserved
byte 3	Reserved
byte 4	Reserved
byte 5	Reserved
byte 6	Reserved
byte 7	Reserved

Field name	Setting value	Remarks
Temperature measurement command	0: Start 1: Stop	–

4. ID: 0x201

byte 0	Motor rotation direction data
byte 1	Brake application information
byte 2	Motor rotation speed information
byte 3	
byte 4	A/D maximum value information
byte 5	
byte 6	Reserved
byte 7	Reserved

Field name	Setting value	Remarks
Motor rotation direction information	0: Clockwise 1: Counterclockwise	–
Brake application information	0: Brake released 1: Brake applied	–
Motor rotation speed information	0 to 65535	The motor rotation speed and A/D maximum value are used for conversion of the speed to a percentage of 0% to 100%.
A/D maximum value information	0 to 65535	

5. ID: 0x202

byte 0	Temperature information
byte 1	Reserved
byte 2	Reserved
byte 3	Reserved
byte 4	Reserved
byte 5	Reserved
byte 6	Reserved
byte 7	Reserved

Field name	Setting value	Remarks
Temperature information	0 to 50	−

5.4.2 Sample program sequence

The flowcharts of the sample program are shown in the following diagrams. First, the internal operating clock of the microcontroller is initialized. Next, the port output for driving the LED is initialized. After this, the A/D converter, external interrupts, CAN, and reload timer are initialized. When an external interrupt occurs due to switch input, CAN transmission and reception (primarily motor control) starts. The reload timer generates interrupts at a fixed interval, which start the A/D converter. The A/D conversion result (used as control data for the motor rotation speed) is then sent via CAN from within the interrupt processing routine for A/D conversion completed.

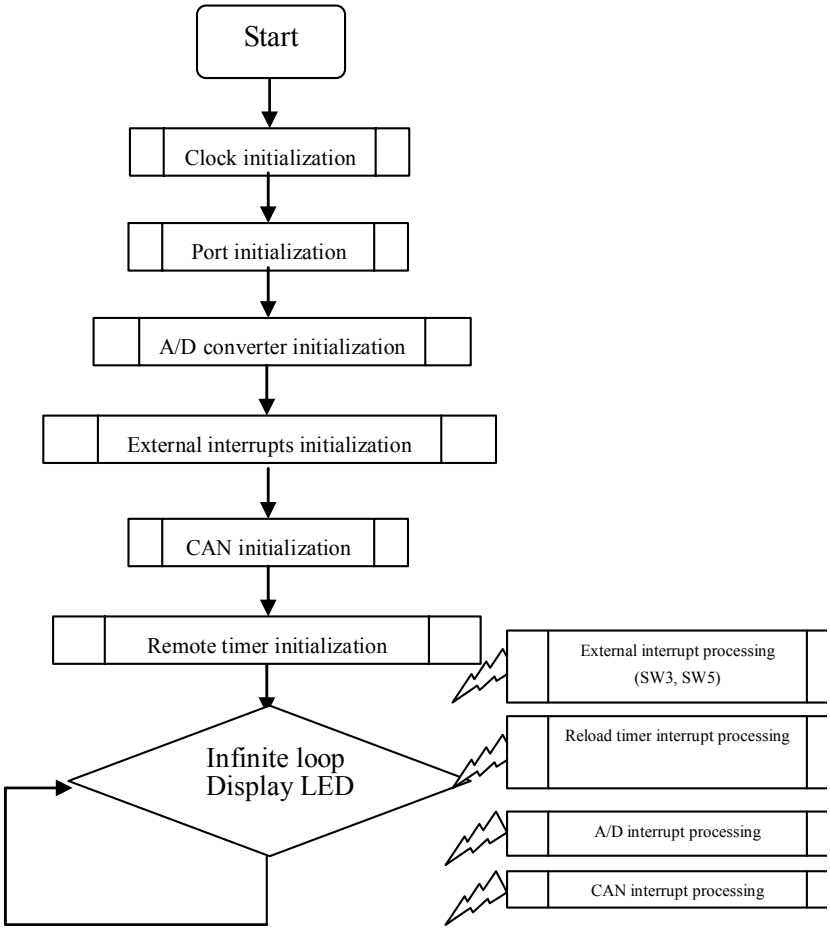


Figure 5-10 CAN communication flowchart

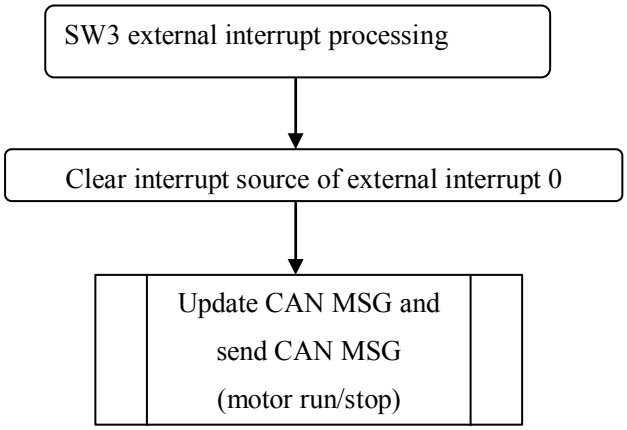


Figure 5-11 SW3 (external interrupt 0) flowchart

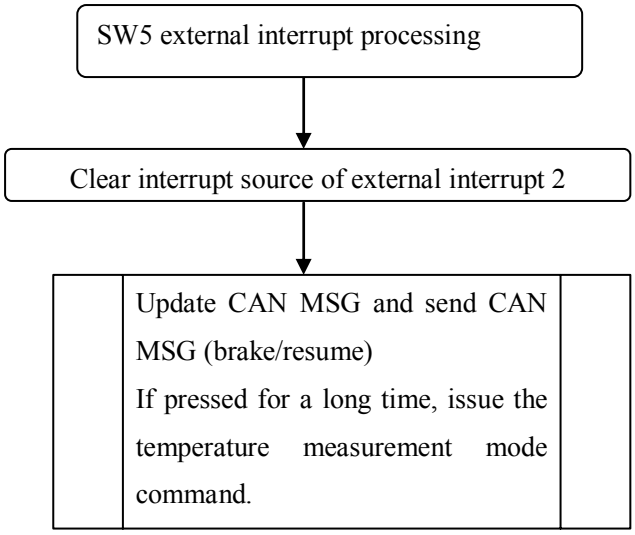


Figure 5-12 SW5 (external interrupt 2) flowchart

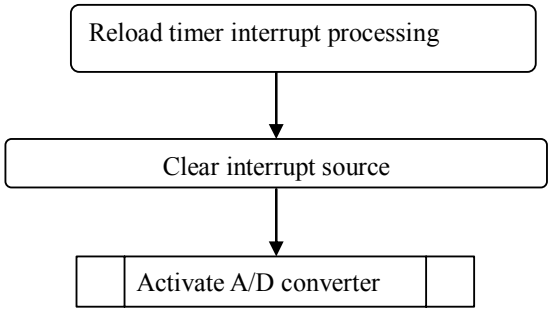


Figure 5-13 the reload timer interrupt flowchart

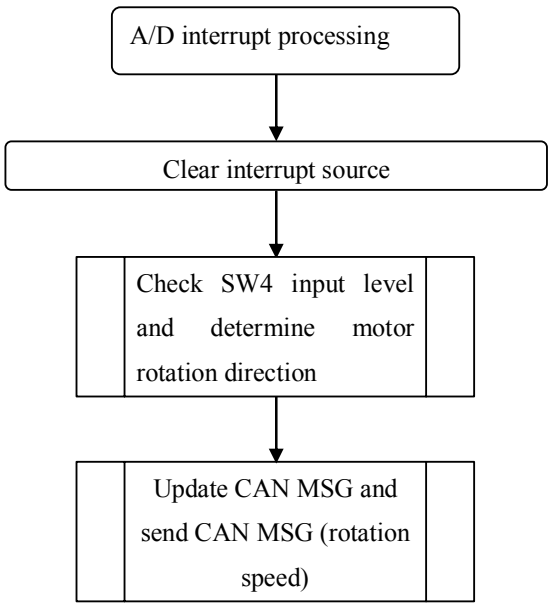


Figure 5-14 the A/D converter interrupt flowchart

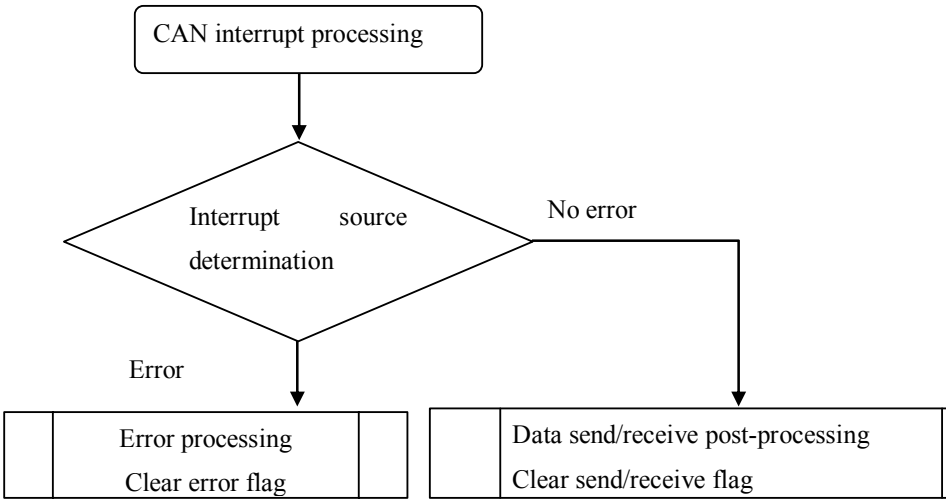


Figure 5-15 the CAN interrupt flowchart

Check the following folder in the sample program. This folder contains several files. First try opening the Main.c file in ¥bits pot_white_SampleProgram¥CAN¥Src.

Main.c:

The main function initializes each peripheral function and then enters an infinite loop that updates the LED display.

```
void main(void)
{
        __set_il(7);                                    // interrupt priority "level 7"
        __EI();                                         // enable interrupt

        set_clock(&Clock_cfg);          // set clock

        Port_Init();                                    // Port Initial

        AD0_Init();                                     // AD initial

        C_CAN_1_Driver();                               // CAN initial

        Ext_Int_Init();                                 // Ext Int initial

        set_reload_timer_1(&Reload_timer_1_init);       // set reload timer 1 to trigger CAN

        while(1)

        {
                __wait_nop();
                LED_display();          ← LED display
                __wait_nop();
        }
}
```

Initialization of each peripheral function

C_CAN_1_Driver_1.c:

The following CAN initialization is performed in C_CAN_1_Driver_1.c.

```
void C_CAN_1_Driver(void)
{
/* initial ICR port baudrate of CAN */
      C_CAN_1_ICR_And_Port();
      C_CAN_1_Baudrate();

/* initial CAN TX and RX in other C file (C_CAN_0_driver_2.c)*/
      init_CAN_1_Tx();                  ← Call CAN transmit buffer configuration function
      init_CAN_1_Rx();                  ← Call CAN receive buffer configuration function
}
/*************** Function definition ********************/
static void C_CAN_1_ICR_And_Port(void)      ↓ CAN interrupt level setting and port Initialization
{
      IO_ICR=0x2006;                      // CAN0 TX/RX/Error status Int level set      ICR_IX=32, ICR_IL=6
      IO_CAN1.COER1.byte=0x01;          // port for CAN Tx is enabled*/
      IO_PIER04.bit.IE2=0x01;             // enable digital input (RX0)
}

/*************** Function definition ********************/
static void C_CAN_1_Baudrate(void)          ↓ CAN baud rate settings
{
/************** CTRLR for baudrate setting ****************************/
      IO_CAN1.CTRLR1.word = 0x0041;                                    // CCE=1 Init=1 for BTR and BRPER setting
      IO_CAN1.BTR1.bit.BRP = ((CAN1_Prescale-0x01)&0x003F);        // to get Lower 6 bit
      IO_CAN1.BRPER1.bit.BRPE = ((CAN1_Prescale&0x03C0)/0x40); // to get Higher 4 bit

/************** bit timing setting ****************************/
      IO_CAN1.BTR1.bit.TSEG1 = CAN1_TSeg1-0x01;
      IO_CAN1.BTR1.bit.TSEG2 = CAN1_TSeg2-0x01;
      IO_CAN1.BTR1.bit.SJW = CAN1_SJW-0x01;

/***************finish CAN baudrate initial*******************/
      IO_CAN1.CTRLR1.word = 0x0001;                                    // CCE=0 Init=1-->over bit timing setting
}
```

C_CAN_1_Driver_2.c:

The CAN transmit and receive buffers are initialized in C_CAN_1_Driver_2.c.

```
/*****************************************************************************
 *   Tx0 initial step:
     1,set IF registor(CMSK,MSK,ARB,Control,Data)
     2,wait transmit RFx to message RAM
*****************************************************************************/
void init_CAN_1_Tx(void)                    ↓ CAN transmit buffer settings
{
/*MSK select*/
            IO_CAN1.IF1CMSK1.word=0x00F7; /*WR/RD=1 Mask=1 Arb=1 Control=1
                                            CIP=0 TxRqst/NewDat=1 DataA=1 DataB=1*/
/*MSK Data*/
            IO_CAN1.IF1MSK1.lword=0xFFFC0000;       /*MXtd=1 MDir=1 res=1 MID28-MID18=1
                                            MID17-MID0=0*/

/*MCTR*/
            IO_CAN1.IF1MCTR1.word=0x1888; /*NewDat=Nouse MsgLst=0 IntPnd=Nouse
                                            UMask=1 TxIE=1 RxIE=0 RmtEn=0
                                            TxRqst=0(Nouse) EoB=1 DLC=8*/

/*CTRLR*/
            IO_CAN1.CTRLR1.word=0x000B;             /*Test=0 CCE=0 DAR=0 EIE=1 SIE=0 IE=1 Init=1*/

}

/*****************************************************************************
 *   Rx0 initial step:
     1,set IF registor(CMSK,MSK,ARB,Control,Data)
     2,transmit RFx to message RAM
     3,Init = 0 enable CAN macro
*****************************************************************************/
void init_CAN_1_Rx(void)                    ↓ CAN receive buffer settings
{
/*MSK select*/
            IO_CAN1.IF2CMSK1.word=0x00F0; /*WR/RD=1 Mask=1 Arb=1 Control=1
                                            CIP=0 TxRqst/NewDat=0 DataA=0 DataB=0*/
/*MSK Data*/
            IO_CAN1.IF2MSK1.lword=0xFFFC0000;       /*MXtd=1 MDir=1 res=1 MID28-MID18 all=1
                                            MID17-MID0 all=0*/
/*Arb Data*/
            IO_CAN1.IF2ARB1.lword=0x88040000;       /*MsgVal=1 Xtd=0 Dir=0 ID(28-18)=0x201*/

/*MCTR*/
            IO_CAN1.IF2MCTR1.word=0x1488; /*NewDat=Nouse MsgLst=0 IntPnd=Nouse
                                            UMask=1 TxIE=0 RxIE=1 RmtEn=0
                                            TxRqst=Nouse EoB=1 DLC=8*/
/*CTRLR*/
            IO_CAN1.CTRLR1.word=0x000B;             /*Test=0 CCE=0 DAR=0 EIE=1 SIE=0 IE=1 Init=1*/

/*CREQ*/
            IO_CAN1.IF2CREQ1.word=0x0004; /*transmit IFx to message RAM
                                            use buffer4*/
            /* for buffer5 */
/*Arb Data*/
            IO_CAN1.IF2ARB1.lword=0x88080000;       /*MsgVal=1 Xtd=0 Dir=0 ID(28-18)=0x202*/

/*CREQ*/
            IO_CAN1.IF2CREQ1.word=0x0005; /*transmit IFx to message RAM use buffer5*/

            IO_CAN1.CTRLR1.bit.INIT = 0;            /*enable CAN controller*/
}
```

C_CAN_1_Int.c：

CAN interrupt processing is performed in C_CAN_1_Int.c. First, the status of CAN communication errors is checked by the State_judge_1() function. If an error has occurred, recovery processing is performed. If there are no errors, the transmission and reception post-processing is performed by the TxRx_Judge_1() function. (For example, the transmission complete flag is cleared or the receive data is saved)

```
/** judge state only when INTR==0x8000 **/          ↓ Check whether or not an error occurred during CAN communication
static void State_judge_1(void)
{
        if(IO_CAN1.STATR1.bit.BOFF==0x01)                       // bus off
        {
                Error_State_1=0x01;

                /*Restart bus*/
                IO_CAN1.CTRLR1.bit.INIT = 0;                    // enable CAN controller
                while((IO_CAN1.ERRCNT1.bit.TEC!=0)||(IO_CAN1.ERRCNT1.bit.REC!=0));
                                                                // see if recovered
        }
        if(IO_CAN1.STATR1.bit.EWARN==0x01)                      // error warning
         {
                Error_State_1=0x02;
        }
        if(!((IO_CAN1.STATR1.bit.BOFF)|(IO_CAN1.STATR1.bit.EWARN)|(IO_CAN1.STATR1.bit.EPASS)))
                                                                // error active
       {
                Error_State_1=0x03;                             // error active
        }
}

/** judge Tx or Rx interrupt **/
static void TxRx_Judge_1(void)                  ↓ CAN transmit/receive processing
{
        MsgNbr1=IO_CAN1.INTR1;                                  // stor MsgNbr

        if(IO_CAN1.STATR1.bit.TXOK==0x01)                       // if TxOK
        {
                IO_CAN1.STATR1.bit.TXOK=0x00;                   // clear TxOK flag

                /* for clear Tx intPnd */
                IO_CAN1.IF1MCTR1.word=0x0888;  //NEWDAT=0 MSGLST=0 INTPND=0 UMASK=0
                                               //TXIE=1 RXIE=0 RMTEN=0 TXRQST=0 EOB=1 DLC=8

                IO_CAN1.IF1CMSK1.word=0x0090;  // WRRD=1 MASK=0 ARB=0 CONTROL=1
                                               // CIP=0 TXREQ=0 DTAA/B=0
                IO_CAN1.IF1CREQ1.bit.MSGN=MsgNbr1;          // IF->RAM

                //IO_CAN0.IF1CMSK0.word=0x0010;
                //IO_CAN0.IF1CREQ0.bit.MSGN=MsgNbr0;

        }
        else if(IO_CAN1.STATR1.bit.RXOK==0x01)                  // if RxOK
        {
                IO_CAN1.STATR1.bit.RXOK=0x00;                   // clear RxOK flag

                /*fetch data from msg RAM*/
                IO_CAN1.IF2CMSK1.word=0x007F;  /*WR/RD=0 Mask=1 Arb=1 Control=1
                                               CIP=1 TxRqst/NewDat=1 DataA=1 DataB=1*/

                IO_CAN1.IF2CREQ1.bit.MSGN=MsgNbr1;          // transmit msgRAM to IF
```

```
                    if(IO_CAN1.IF2MCTR1.bit.MSGLST==0x01)
                    {
                            __wait_nop();                                   // mag lost
                    IO_CAN1.IF2MCTR1.word=0x1488;       // NewDat=0 MSGLST=0 INTPND=0 UMSK=1 TXIE=0
                                                                            // RXIE=1 RMTEN=0 TXRQST=0 EOB=1
                            IO_CAN1.IF2CMSK1.word=0x0090;  // WRRD=1 CONTROL=1 other=0
                                                                            // for clear MSGLST
                            IO_CAN1.IF2CREQ1.bit.MSGN=MsgNbr1;

                    }
                    else
                    {
                            /*ID(28-18)=0x201*/
                            if((IO_CAN1.IF2ARB1.lword&0x1FFC0000)==0x08040000)
                            {
                                    __wait_nop();
                    CAN_Rx1_data0_ID_201.word=IO_CAN1.IF2DTA11;             // save data from buffer to RAM
                    CAN_Rx1_data1_ID_201.word=IO_CAN1.IF2DTA21;             // save data from buffer to RAM
                    CAN_Rx1_data2_ID_201.word=IO_CAN1.IF2DTB11;             // save data from buffer to RAM
                    CAN_Rx1_data3_ID_201.word=IO_CAN1.IF2DTB21;             // save data from buffer to RAM

                            CAN_RX1_data_ID_201_Received=0x01;              // set received flag
                                    __wait_nop();
                            }

                            /*ID(28-18)=0x202*/
                            else if((IO_CAN1.IF2ARB1.lword&0x1FFC0000)==0x08080000)
                            {
                                    __wait_nop();
                    CAN_Rx1_data0_ID_202.word=IO_CAN1.IF2DTA11;             // save data from buffer to RAM
                    CAN_Rx1_data1_ID_202.word=IO_CAN1.IF2DTA21;             // save data from buffer to RAM
                    CAN_Rx1_data2_ID_202.word=IO_CAN1.IF2DTB11;             // save data from buffer to RAM
                    CAN_Rx1_data3_ID_202.word=IO_CAN1.IF2DTB21;             // save data from buffer to RAM

                    CAN_RX1_data_ID_202_Received=0x01;                      // set received flag
                                    __wait_nop();
                            }
                            else
                            {
                                    __wait_nop();
                                    __wait_nop();
                            }
            }
        }
    }
```

6 Try to use LIN communication

Communication is to send/receive information. There are, in fact, various communications formats, such as transmission by people talking, letters written in script, and electronic communications, etc.

Among these, there are various plans for communications using electricity. This chapter explains communications in a standard called LIN.

6.1 What is LIN?

LIN is an acronym for Local Interconnect Network, and is a type of communications protocol for vehicle-mounted LAN. The LIN consortium was proposed in 1999 with the objective of enabling a less expensive configuration than CAN, which is the most widespread control system vehicle-mounted LAN. Thereafter, after several version upgrades, LIN2.0, which has added diagnostic and other functions, was launched in 2003. Further, in 2006, the version was upgraded to LIN2.1.

This section explains LIN applications. Concomitant with multi-function vehicles, the existence of a network in vehicles also became indispensable. Currently, vehicle-mounted LANs are broadly divided into two classifications: control systems, which are concerned with motoring and the vehicle body, and information systems, which connect devices such as the satellite navigation system and audio, and so different LANs are used depending on the application. In particular, vehicle body devices such as electric mirrors and power windows, which are classified as body systems, do not require such fast or detailed control. Consequently, they are also inexpensive. This is where LIN is used.

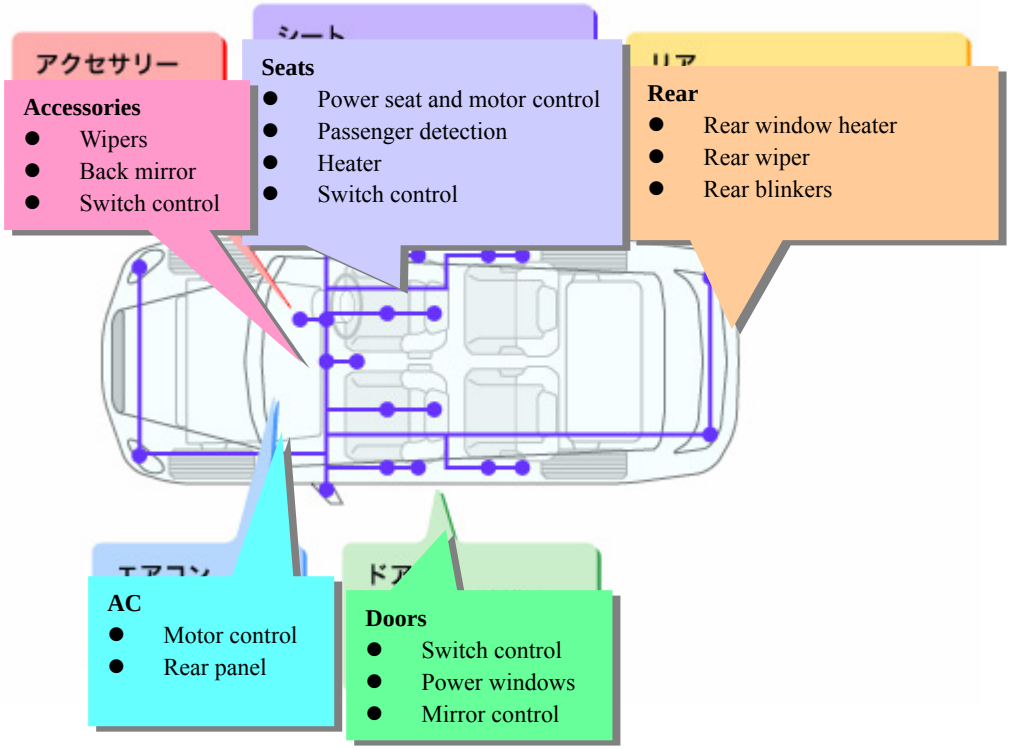


Figure 6-1 Example of vehicle LIN applications

The characteristics of LIN used in the way described above, are collated and introduced in the following five points.

1. Single master communication

   LIN has two types of communication nodes. One is the “master” (sender).This controls the start of all communications. The other is the slave (recipient).The slave responds to commands sent by the master. LIN communication must start from the master, and cannot be started by a slave. Further, the LIN communication mode designated as the master is pre-determined. This format is called a “single master format”.

2. A maximum of 15 slave nodes can be connected using bus wiring.

   The LIN network configuration (topology) is a bus. With single master LINs, the slaves communicate only when they receive commands from the master, so there is no conflict of signals in the bus. A maximum of 15 slave nodes can be connected to one master.

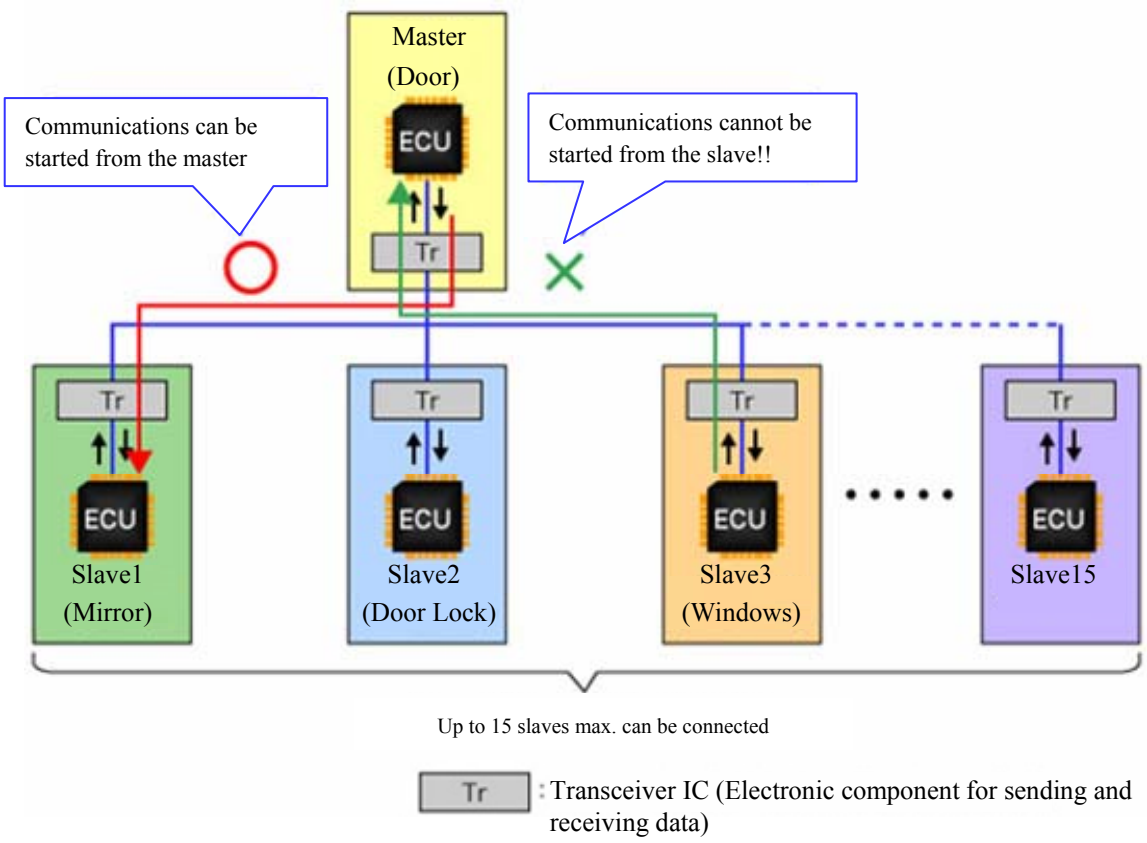


Figure 6-2 Main LIN network configuration

3. Wiring is completed using a single wire

   The on-board ECUs are connected to the LIN network via transceiver ICs (electronic components that send and receive data), and each ECU is connected on the bus from the master to a slave. An ordinary single metal wire is used as the bus cable. CAN combines two opposing metal wires to make one twisted pair cable. FlexRay uses two twisted pair cables. Consequently, LIN has the advantage of using a single cable for numerous network wires, unlike CAN and FlexRay, which use twisted pair cables.

   The communications distance is 40m max. LIN can be used in combination with CAN, and in such cases, CAN is most frequently used as the core network, and LIN is used as the branch network.

4. The baud rate is 20kbps max.

   L The baud rate according to LIN specifications is within the range 1 to 20kbps. Practically, the baud rate of LINs used as LANs depends on the individual vehicle manufacturer’s system

specifications, but generally one of the following is used: 2,400kbps, 9,600kbps, or 19,200kbps.

5. Communications errors are detected only, and subsequent processing depends on the application

   With LIN, communications errors are detected based on information as to whether transmitting and receiving has been performed successfully. Processing after an error has been detected, however, is not specified. Here, LIN error processing can be customized according to the application. CAN and FlexRay management of the communications status depends on the counter value, which is called the error counter, is featured by the specifications, but in LIN, if an error occurs, simple error processing is possible, in which LIN merely waits for the next command.

6.2 LIN specifications

This section explains briefly the LIN specifications.

For detailed specifications, access the LIN consortium website (http://www.lin-subbus.org/), and register your name and e-mail address to get a specifications.

6.2.1 Lin frame configuration

This section explains frames, which is the basic unit of LIN communication.

LIN frames are configured using “headers” and “responses”. As shown in “Figure 6-3 LIN communication flow”, the basic communications flow is a procedure in which the master sends headers to the slaves, and the slaves implement processing according to the contents of the headers received, and then send a response to the master.

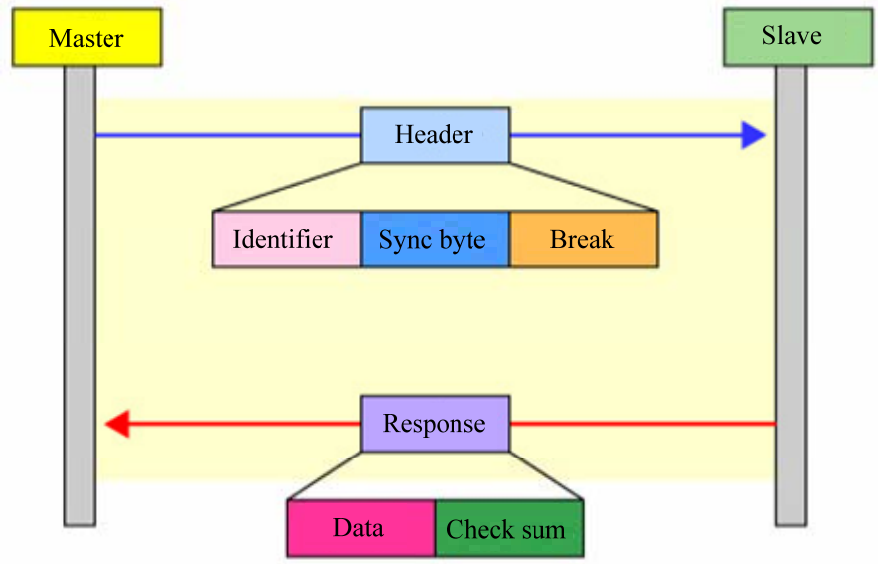


Figure 6-3 LIN communication flow

Further, headers are configured using three fields: Synch break (Break), Synch field (Sync byte), and ID field (Identifier), and responses are configured using two fields: Data field and Checksum field.

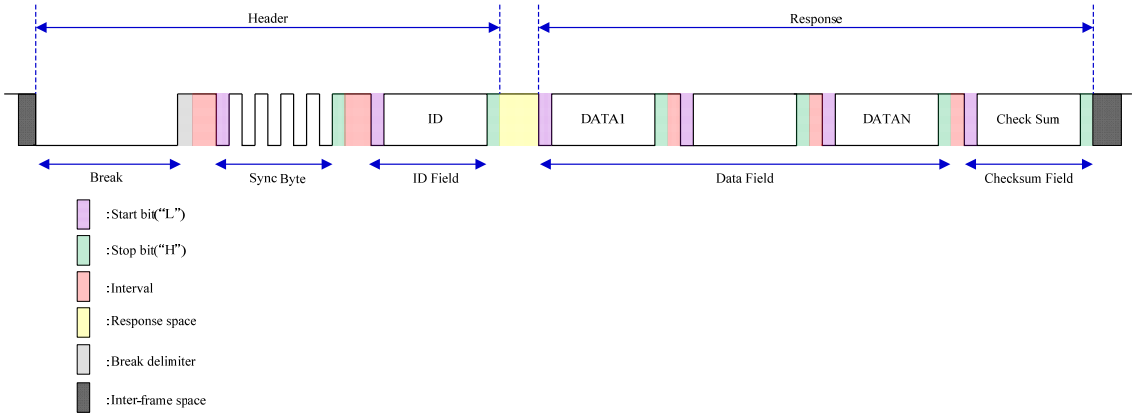


Figure 6-4 LIN frame configuration

1. Break

   Break, which are in the header fields, are variable-length fields that indicate the start of a new frame. They comprise 13 to 16 “0” bits (fixed value zero) min. The general frame length is 13 bits.

2. Sync Byte

   Sync byte, which follow on from breaks, are 10-bit fixed-length fields that synchronize the master and the slaves. Sync byte configurations comprise 1 starter bit (“0”), 8 data bits, and 1 stop bit (“1”).The 8-bit data bit has the fixed value “0x55” (which is expressed as “0x01010101” in binary).If the slave receives the 0x55 in the synch byte send by the master normally, the master and slave are synchronized.

3. ID field

   The “ID field”, which is the final header field and comes after the synchronous byte, is a 10-bit fixed-length field that specifies the frame type and objective. ID fields have values from “0” to “63” (6 bits).This ID field is also used by the master to specify individual slaves. Slaves judge what type of frame has been sent and if it was intended for them according to the ID field sent by the master, and send responses to the master accordingly. Further, the ID field has a 2-bit parity bit following the “0” to “63” (6 bits).This is bracketed by a 1-bit starter bit and 1-bit stop bit in the same way as the synchronous byte, so overall the field is 10 bits in length.

4. Data field

   The “data”, which is in the response header, is a variable-length field that literally transfers data. The data in the number of bytes that has been predetermined (1 to 8 bytes) is sent. As there is a 1-bit start bit and 1-bit stop bit bracketing the 1-byte data in the same way as the header synchronous byte, 1 byte of data is configured from 10 bits. Consequently, the total data field length is “number of bytes x 10 bites”.

5. Checksum field

   The “checksum”, which follows the data, is a 10-bit fixed-length field for checking data. The data recipient checks whether there is an error in the data by comparing the data received with the checksum. The checksum field length is also 10 bits: a start bit and a stop bit added to the 8-bit checksum in the same way as the synchronous byte.

6.3 LIN communication flow

In general LIN communication, one master communicates with numerous slaves. LINs, which adopt a bus topology, connect the master and all the slaves using a single wire, so header electrical signals sent by the master are transmitted by the wire to all the slaves. The slaves check the frame ID, and if the header is addressed to them, sent a response to the master according to the content received. If the header received is addressed to another slave, it is ignored. In this way, 1-to-1 communication between the master and each slave is achieved.

This section explains the actual trading of communications. Currently, functions are allocated to each of the slaves from 1 to 15.The master first communicates with slave 1 and turns the motor ((1) in Figure 6-6-5 Main LIN network configuration and Figure 6-6-6 Example of communication sequence between the master and slaves during normal communication), and next acquires sensor information by communicating with slave 3. ((2) in “Figure 6-6-5 Main LIN network configuration” and Figure 6-6-6 Example of communication sequence between the master and slaves during normal communication.) Thereafter, the motor is turned by communications with slave 2 ((3) in Figure 6-5 Main LIN network configuration and Figure 6-6 Communications sequence between master and slave during normal communications).The master acquires sensor information from slave 3 again ((4) in Figure 6-5 Main LIN network configuration and Figure 6-6 Communications sequence between master and slave during normal communications), and finally turns ON the lamp by communicating with slave 15 ((5) in Figure 5-5 Main LIN network configuration and Figure 6- 6 Communications sequence between master and slave during normal communications).In this chain of communications, communications between the master and slaves 2 and 3 are contiguous, and the master processes the motor turning by communicating with slave 2 using sensor information acquired by communicating with slave 3 first. In this way, during actual communications the master and multiple slaves repeatedly communicate on a 1-to-1 basis.

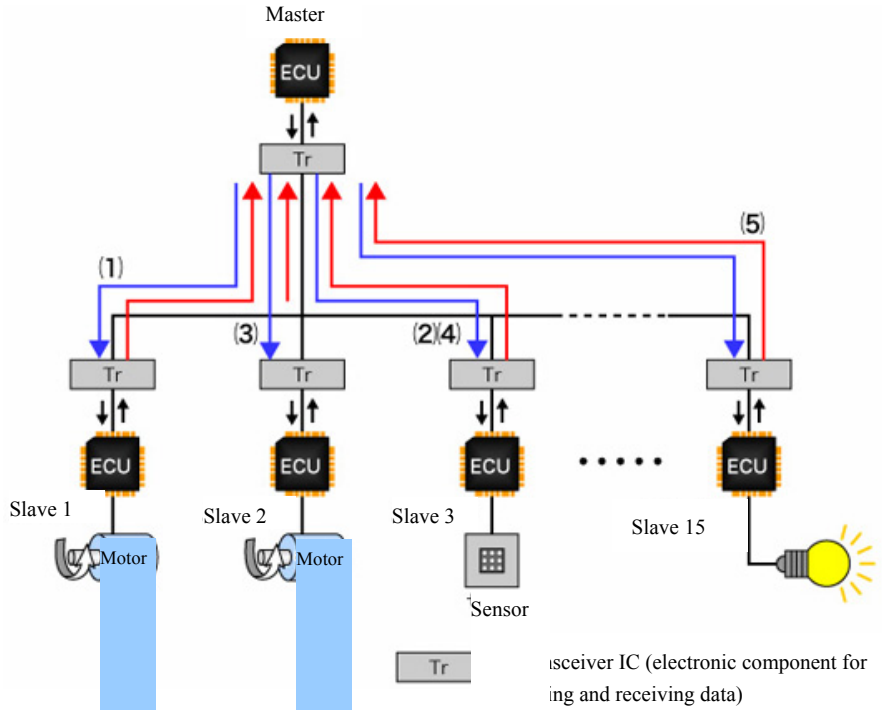


Figure 6-6 ain LIN ne configuration

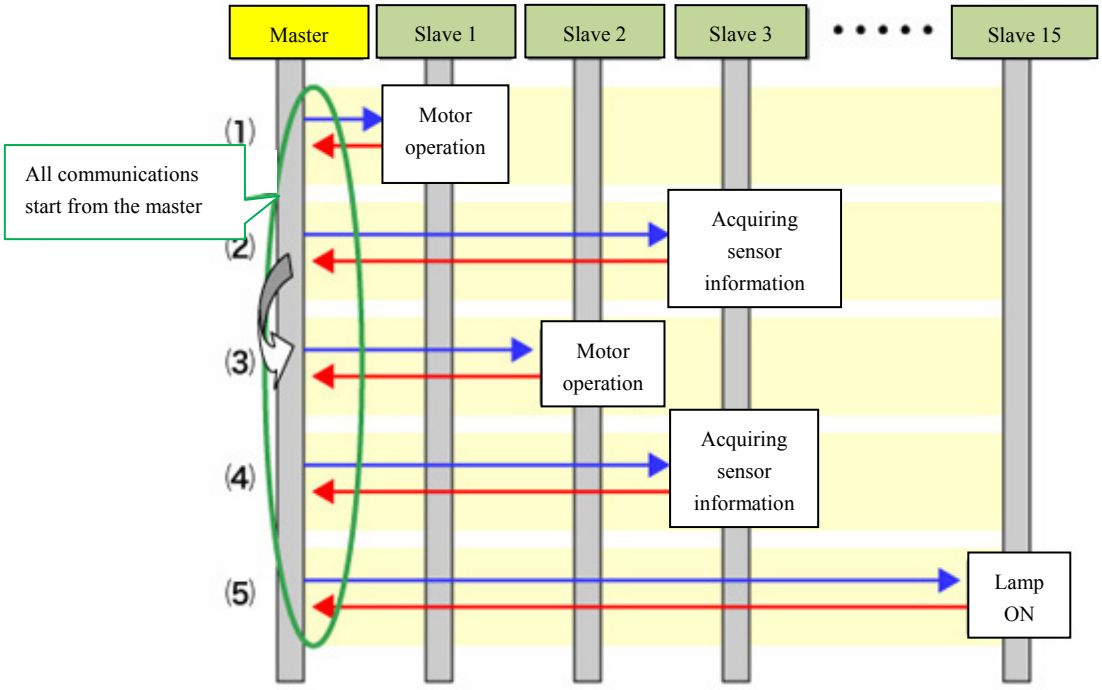


Figure 6-6-6 Example of communication sequence between the master and slaves during normal communication

6.4 Communication between master and slave if an error occurs

LIN error processing is not determined by the protocols, and so depends on the application. Consequently, during design, it is necessary to consider the error detection methods and the process after the error has been processed. As this is not determined by the protocols in the LIN specifications either, however, examples of system design if an error occurs are introduced in the chapter “Status Management”. In the examples introduced, errors are managed by slaves reporting their own status to the master. This mechanism is described below.

The basic master operation is merely to send the header to the next slave when communications with the current slave have ended. On the other hand, the slave operation is to perform error checking when a header is received and when a response is sent. Checksums and other checks are implemented during reception. When sending, checks are performed by comparing the sent data and the bus data that performs the monitoring. In this way, the slave identifies its own status, and inserts the results into the response that is sent to the master. The master identifies the slave status from the response, and if there is a nonconformance, initializes the slave. In this way, the error status is completely cleared.

6.5 LIN communication by using the microcontroller

This section describes how to perform practical LIN communication with the microcontroller.

On the board, as shown in “Figure 6-7 LIN circuit”, the microcontroller is connected with the LIN transceiver IC (TJA1020T). On the microcontroller, SOT is used for transmission and SIN is used for reception, and SCK controls the transceiver IC as a port. Signals transmitted/received pass onto the bus through the LIN transceiver.

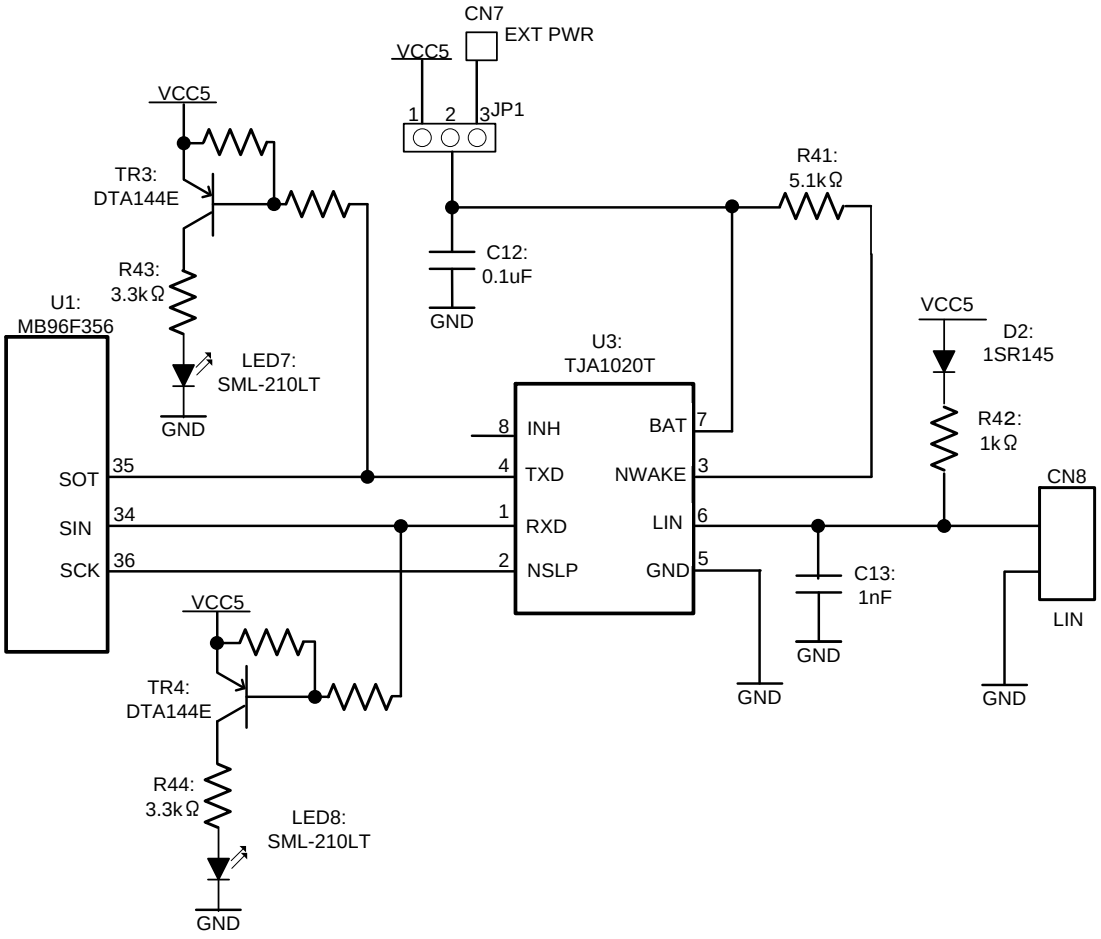


Figure 6-7 LIN circuit

The registers used for entire LIN communication control on the microcontroller are as shown in “Figure 6-8 Entire LIN communication control register”.

A description of the registers and their setting values in the sample program are as described in “ Table 6-1 Description of the entire LIN communication control registers and setting values”. For more information of the registers, refer to the microcontroller hardware manual.

Serial control register

	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
SCR	PEN	P	SBL	CL	AD	CRE	RXE	TXE

Serial mode register

	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
SMR	MD1	MD0	OTO	EXT	REST	UPCL	SCKE	SOE

LIN-UART serial status register

	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
SSR	PE	ORE	FRE	RDRF	TDRE	BDS	RIE	TIE

LIN-UART receive data register/transmit data register

	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
RDR/TDR								

LIN-UART extended status control register

	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
ESCR	LBIE	LBD	LBL1	LBL0	SOPE	SIOP	CCO	SCES

LIN-UART extended communication control register

	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
ECCR	INV	LBR	MS	SCDE	SSM	res	RBI	TBI

LIN-UART baud rate generator register 1

	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
BGR1	-							

LIN-UART baud rate generator register 0

	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
BGR0								

	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
ESIR					TDRE	RDRF	RBI	AICD

Figure 6-8 Entire LIN communication control register

Table 6-1 Description of the entire LIN communication control registers and setting values

Register name	Setting value [function]	Description
SCR_PEN	0 [No parity]	Parity enable bit
SCR_P	0 [Even parity]	Parity selection bit
SCR_SBL	0 [1 bit]	Stop bit length selection bit
SCR_CL	1 [8-bit]	Data length selection bit
SCR_AD	0 [Data frame]	Address/data selection bit
SCR_CRE	1 [Flag clear]	Receive error flag clear bit
SCR_RXE	1 [Receive enabled]	Receive enable bit
SCR_TXE	1 [Transmit enabled]	Transmit enable bit
SMR_MD1	1 [Mode 3]	Operation mode selection bit
SMR_MD0	1 (Asynchronous LIN mode)	
SMR_OTO	0 [Use external clock]	One-to-one external clock selection bit
SMR_EXT	0 [Use baud rate generator]	External clock selection bit
SMR_REST	0	Transmit reload counter restart bit
SMR_UPCL	1 [LIN-UART reset]	USART programmable clear bit (software reset)
SMR_SCKE	0 [General-purpose I/O port or LIN-UART clock input pin]	Serial clock output enable bit
SMR_SOE	1 [LIN-UART serial data output pin]	Serial data output enable bit
SSR_BDS	0 [LSB first (send from least significant bit)]	Transfer direction selection bit
SSR_RIE	1 [Receive interrupt enabled]	Receive interrupt request enable bit
SSR_TIE	0 [Transmit interrupt disabled]	Transmit interrupt request enable bit
ESCR_LBIE	0 [LIN synch break detection interrupt disabled]	LIN synch break detection interrupt enable bit
ESCR_LBD	0 [LIN synch break detection clear flag]	LIN synch break detection flag
ESCR_LBL1	0	LIN synch break length selection bit
ESCR_LBL0	0 [13-bit length]	
ESCR_SOPE	0 [Serial output pin access disabled]	Serial output pin direct access enable
ESCR_SIOP	0	Serial input/output pin direct access
ESCR_CC0	0	Continuous clock output enable bit
ESCR_SCES	0	Serial clock edge selection bit
ECCR_LBR	0 [Do not generated LIN synch break]	LIN synch break generate bit
ECCR_MS	0	Master/slave mode selection bit
ECCR_SCDE	0	Serial clock delay enable bit
ECCR_SSM	0	Start/stop bit mode enable bit
BGR_BGR1	0x16 (When set to 9600bps)	Baud rate generator 1
BGR_BGR0	0x66 (When set to 9600bps)	Baud rate generator 0

6.6 Understanding and running the program for LIN communication

An explanation of the sample program is given as an example of a program that actually performs LIN communication. In the bits pot LIN communication, the starter kit operates as the master and the bits pot yellow operates as the slave.

6.6.1 LIN communication configuration

The LIN communication parameters used by the sample program are summarized in “Table 6-2 LIN communication conditions of the sample program”.

Table 6-2 LIN communication conditions of the sample program

Condition	Value
Communication speed	2400/9600 (default value)/19200bps
Peripheral clock frequency	16MHz
Synch break length	13 bits (Receive is fixed to detect 11 bits)
Data length	8 bits
Data bit format	LSB first
Data byte count	8 bytes

The message IDs used in LIN communication in “Table 6-3 LIN message IDs in the sample program” are described next.

Table 6-3 LIN message IDs in the sample program

ID	Description	Data communication direction
0x00	Temperature measurement command/temperature display command	white → yellow
0x01	Temperature sensor information	white → yellow white ← yellow
0x02	Buzzer output command/volume value measurement command	white → yellow
0x03	Volume SW (VR) information	white → yellow white ← yellow
0x04	LED on/off change command– count up/count down	white → yellow
0x05	LED value	white → yellow white ← yellow

Details of the IDs are as shown below.

1. ID: 0x00

byte 0	Temperature measurement command
byte 1	A/D value (temperature sensor information)
byte 2	Reserved
byte 3	Reserved
byte 4	Reserved
byte 5	Reserved
byte 6	Reserved
byte 7	Reserved

Field name	Setting value	Remarks
Temperature measurement command	0x55: Start 0x0F: Stop	When SW4 is on the right, 0x55 is sent. This acquires and displays the temperature information from the bits pot yellow. When SW4 is on the left, 0x0F is sent. The temperature information is not displayed.
A/D value (temperature sensor information)	0 to 255	The temperature sensor information from the starter kit. The bits pot yellow displays the temperature on the LED using this A/D value.

2. ID: 0x01

byte 0	Reserved
byte 1	Reserved
byte 2	A/D value (temperature sensor information)
byte 3	Reserved
byte 4	Reserved
byte 5	Reserved
byte 6	Reserved
byte 7	Reserved

Field name	Setting value	Remarks
A/D value (temperature sensor information)	0 to 255	The response from the bits pot yellow to the ID 0x00 temperature measurement command. The temperature sensor information is received as an A/D value and is displayed on the 7SEG display.

3. ID: 0x02

byte 0	Volume value acquire command
byte 1	A/D value (VR information)
byte 2	Reserved
byte 3	Reserved
byte 4	Reserved
byte 5	Reserved
byte 6	Reserved
byte 7	Reserved

Field name	Setting value	Remarks
Volume value acquire command	0x55: Start 0x0F: Stop	When SW4 is on the right, 0x55 is sent. Acquires the bits pot yellow volume SW information and outputs the buzzer sound. When SW4 is on the left, 0x0F is sent. The buzzer sound is not output.
A/D value (VR information)	0 to 255	Starter kit volume SW information. The bits pot yellow outputs the buzzer sound based on this A/D value.

4. ID: 0x03

byte 0	Reserved
byte 1	Reserved
byte 2	A/D value (VR information)
byte 3	Reserved
byte 4	Reserved
byte 5	Reserved
byte 6	Reserved
byte 7	Reserved

Field name	Setting value	Remarks
A/D value (VR information)	0 to 255	The response from the bits pot yellow to the ID 0x02 volume value acquire command. The volume SW information is received as an A/D value and output to the buzzer.

5. ID: 0x04

byte 0	LED on/off change command
byte 1	Reserved
byte 2	Reserved
byte 3	LED value
byte 4	Reserved
byte 5	Reserved
byte 6	Reserved
byte 7	Reserved

Field name	Setting value	Remarks
LED on/off change command	0x55: Start 0x0F: Stop	The LED on/off change command from the starter kit. When SW4 is on the left, 0x55 is sent. In addition, if a LED value other than 0xFF is received, the received LED value is displayed by the starter kit LEDs.
LED value	0 to 7 (Or 0xFF)	The value of the LED displayed by the starter kit. When 0xFF is sent, the data is invalid.

6. ID: 0x05

byte 0	Reserved
byte 1	Reserved
byte 2	Reserved
byte 3	LED value
byte 4	Reserved
byte 5	Reserved
byte 6	Reserved
byte 7	Reserved

Field name	Setting value	Remarks
LED value	0 to 7	The value of the LED displayed by the bits pot yellow. If 0xFF is sent, the data is invalid.

6.6.2 Sample program sequence

The flowcharts for the LIN communication in the sample program are shown in “Figure 6-9 LIN communication flowchart (main routine)” and “Figure 6-10 LIN communication flowchart (interrupt routine: USART receive interrupt)”. First, the microcontroller is initialized, the LIN-USART is initialized, and the timer is initialized. Next, the bus connection processing is performed as the LIN master, and the schedule is set. After this, the program enters a loop. Within the loop, headers are sent and responses are sent and received at fixed intervals. Sending of the synch break, synch field, and ID field headers and sending and receiving of responses is processed by the LIN-USART receive interrupt. Processing is performed in response to the master ID (identifier).

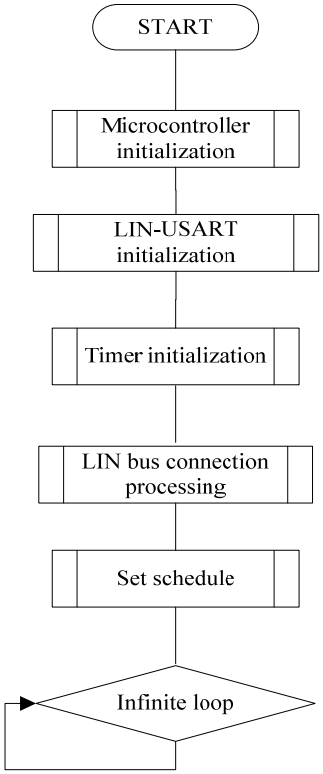


Figure 6-9 LIN communication flowchart (main routine)

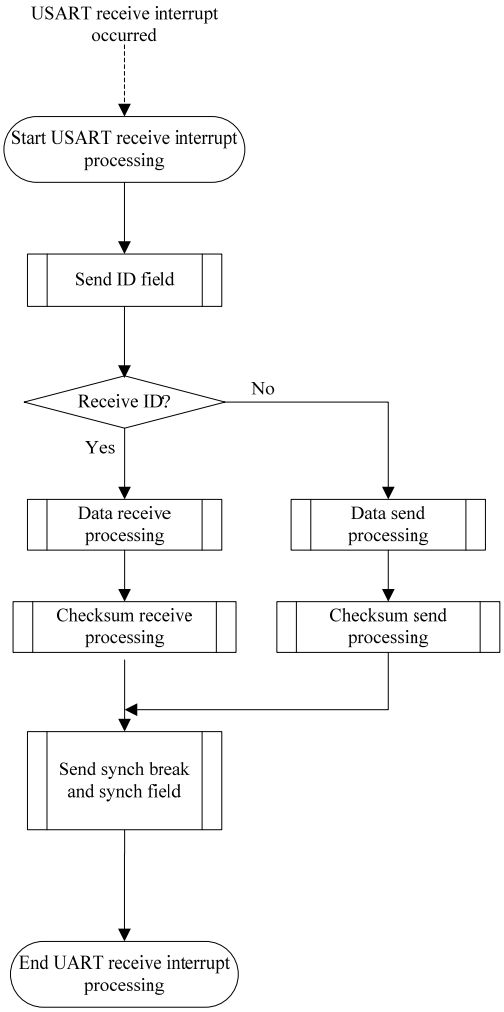


Figure 6-10 LIN communication flowchart (interrupt routine: USART receive interrupt)

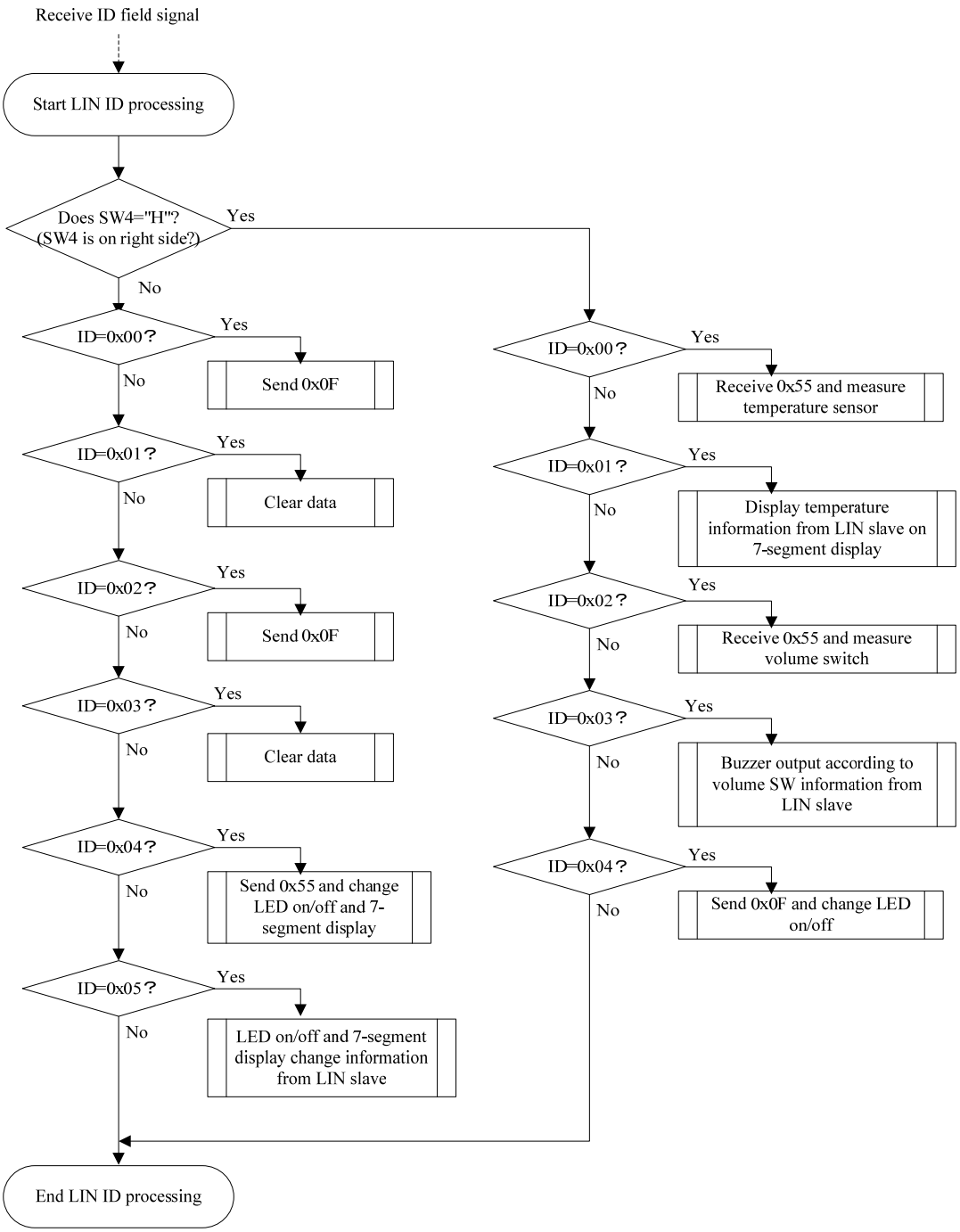


Figure 6-11 LIN communication flowchart (data processing by ID)

The sample program is explained next. However, the sample program contains sections that are not used during communication with the bits pot yellow. These sections have been made extensible, and may be included in programs that meet the LIN specifications and programs that operate as the slave. However, the operation of these sections has not been completely verified. Please take care if you use these sections.

The points where this sample program operates in the LIN protocol during LIN communication are shown below. Because the sample program is the LIN master, the LIN bus connection processing and schedule registration are performed first.

```
void        main(void)
{
            (omitted)
            l_ifc_connect(hLIN_NORMAL_WAKEUP);          ← LIN bus connection processing
            l_sch_set(hSchedule1, Schedule1_DATA00);    ← Set schedule
```

Figure 6-12 LIN bus initial settings

Parts of the section that sets the LIN ID and sets the schedule is shown below.
In the sample program, one schedule table and eight IDs are used. The IDs that actually use a response are ID 0x00 to ID 0x05.

```
typedef     enum {
            Schedule1_DATA00 = 0,                                   ← Register 8 IDs
            Schedule1_DATA01,
            Schedule1_DATA02,
            Schedule1_DATA03,
            Schedule1_DATA04,
            Schedule1_DATA05,
            Schedule1_DATA06,
            Schedule1_DATA07,
            (omitted)
#define                 Schedule1Count      8                       ← Register 8 IDs
            (omitted)
__far const     l_u8    Schedule1_IdList[Schedule1Count] =          ← Register 8 IDs
                {       ID_00, ID_01, ID_02, ID_03, ID_04, ID_05, ID_06, ID_07    };
```

Figure 6-13 ID registration – Lindbmaster.h

```
l_u8*     __far const          LinTxDataPtr[64] = {
/*        0  1  2  3  4  5  6  7  8  9  */
/* 0*/    ucDATA00, 0, ucDATA02, 0, ucDATA04, 0, ucDATA06, 0, 0, 0,     ← Register send response
/*10*/    0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
          (omitted)
l_u8*     __far const          LinRxDataPtr[64] = {
/*        0  1  2  3  4  5  6  7  8  9  */
/* 0*/    0, ucDATA01, 0, ucDATA03, 0, ucDATA05, 0, ucDATA07, 0, 0,     ← Register receive response
```

Figure 6-14 Send and receive response registration – Lindbmsg.h

This sample program operates by processing multiple interrupts, as shown in "Figure 6-15 Points where the processing of each interrupt is performed". We will now look at the processing performed by the sample program for each field of the LIN protocol.

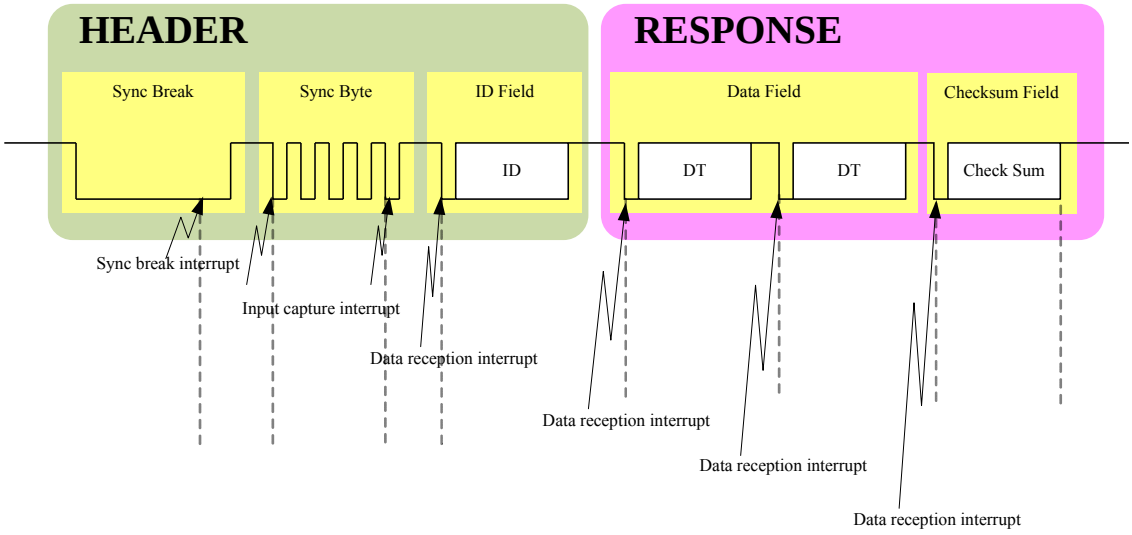
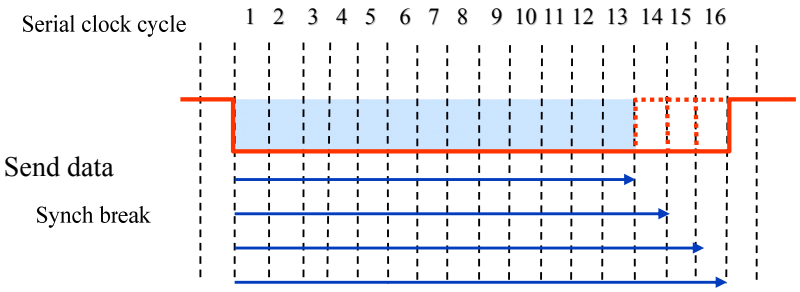


Figure 6-15 Points where the processing of each interrupt is performed

① Synch break

On synch break, a synch break signal (a low signal for 13 to 16 bits) is sent. In the sample program, 13 bits are sent. Receive processing is also performed at the same time, and if the

bus is “0” for 11 bit times or more, a synch break interrupt occurs. When a synch break interrupt is detected, the synch break interrupt is set to disabled, and processing to determine whether a synch break was received is performed within l_ifc_rx(data).



LBR	LIN synch break generate bit
0	No effect
1	Generate LIN synch break

LBL0	LBL1	LIN synch break length selection bits
0	0	13 bits long
1	0	14 bits long
0	1	15 bits long
1	1	16 bits long

Figure 6-16 Synch break data setting

```
__interrupt void    _LinUartRx(void)
{
        if ((ssr & 0xE0) != 0) {                          ← Error check
        (omitted)
        } else if (IO_UART3_ESCR3_bit_LBD == SET) {       ← Synch break detection
        #if         (LIN_MASTER==1)
                    IO_UART3_ESCR3_bit_LBD = CLEAR;       ← Clear synch break detection flag
        (omitted)
    }else{
        l_ifc_rx(data);                          ← Receive processing
```

Figure 6-17 Synch break interrupt control

The processing for synch break reception, synch field reception, ID reception, DATA transmission and reception, and WAKEUP transmission are split up according to the status, as shown in “Figure 6-23 ID receive determination processing”. In the processing to determine whether a synch break was received, the extended status control register (ESCR) is cleared and the program enters a state of waiting for reception of the synch field.

```
void      l_ifc_rx(l_ifc_handle rx_data)
{
  switch(ucLinStatus){
    case  LIN_TRANSMIT:                          ← Transmit DATA FIELD
          (omitted)
    case  LIN_DATA_RECEPTION:                    ← Receive DATA FIELD
          (omitted)
    case  LIN_ID_RECEPTION:                      ← Wait to receive ID FIELD
          (omitted)
    case   LIN_WAIT_SYNCH_FIELD:                 ← Wait to receive Synch field
          (omitted)
    case  LIN_MS_WAIT_SYNCH_BREAK:               ← Wait to receive Synch break
#if       (LINUART_CH==3)
              IO_UART3_ESCR3_byte = 0x00;        ← Clear ESCR register
          (omitted)
               ucLinStatus = LIN_WAIT_SYNCH_FIELD;  ← Switch to wait to receive
          (omitted)                              Sync field state
    case  LIN_WAKEUP_TRANSMIT:                   ← Transmit WAKEUP state
          (omitted)
  }

}
```

Figure 6-18 Processing to determine whether synch break was received

② Synch field

After a synch break is detected, processing is performed to send and receive the synch field. 0x55 is sent in the synch field. If this data is successfully received by the slave, the slave becomes synchronized.

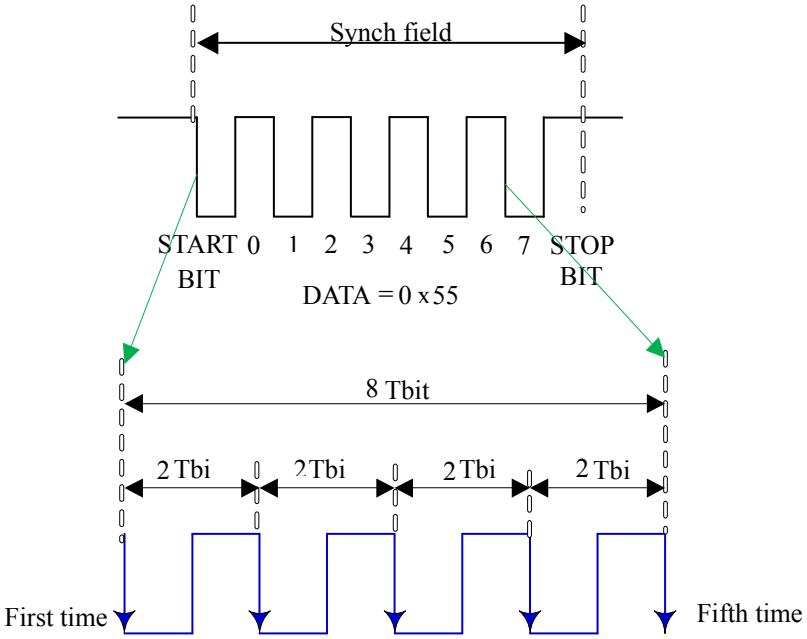


Figure 6-19 Synch field interrupt control

The processing to receive the synch field, ID field, and Data field is performed within the LIN-UART interrupt function _LinUART. If there are no errors when an interrupt occurs and the interrupt source is not synch break, received processing is performed.

```
__interrupt void    _LinUART (void)
{
    (omitted)
    if ((ssr & 0xE0) != 0) {                    ← Error check
            (omitted)
    } else if (IO_UART3_ESCR3_bit_LBD == SET) {    ← Check whether or not interrupt
            (omitted)                                 by synch break
    }else{
        l_ifc_rx(data);                         ← Receive processing
    }
}
```

Figure 6-20 Synch field interrupt control

In “Figure 6-23 ID receive determination processing”, once it has been confirmed that the data is 0x55, the program enters a state of waiting to receive the ID field and performs ID send processing.

```
void    l_ifc_rx(l_ifc_handle rx_data)
{
        (omitted)
        case    LIN_WAIT_SYNCH_FIELD:                   ← Wait to receive sync field
                if (rx_data == SYNCH_FIELD_CHAR) {      ← Check if Sync field is 0x55
        (omitted)
                ucLinStatus = LIN_ID_RECEPTION;         ← Switch to wait to receive ID state
                l_ifc_tx(ucLinMsScheduleCurrentId);     ← Transmit ID processing
        (omitted)
```

Figure 6-21 Synch field receive determination processing

The send data is stored in a register within the send start processing function l_ifc_tx(l_ifc_handle tx_data) as shown in “Figure 6-23 ID receive determination processing”.

```
void    l_ifc_tx(l_ifc_handle tx_data)
{
#if     (LINUART_CH == 3)
        IO_UART3_RDR3 = tx_data;                ← Set send data in register
        (omitted)
```

Figure 6-22 UART send start processing

③ ID field

In order to transition to the state of waiting to receive the ID field, the ID receive determination processing is performed in the normal sequence in the ID receive processing, as shown in “Figure 6-23 ID receive determination processing”. In the ID receive determination processing, a judgment is made as to whether the acquired ID is for sending or for receiving and a parity check is performed. If the ID is for sending, the status is changed to the send preparation state and the send data is copied to a buffer. If the ID is for receiving, the status is changed to the DATA receive wait state in preparation for receiving a response (data) from the slave.

```
void      l_ifc_rx(l_ifc_handle rx_data){
          (omitted)
   case   LIN_ID_RECEPTION:                  ← Wait to receive ID FIELD state
          ucCurrentId.byte    = rx_data;     ← Store received ID
          if( ucCurrentId.fields.parity != ucRightParity[ucCurrentId.fields.id] ) {   ← Parity check
          l_flg_tst(hBIT_ERR);              ← Error processing
          l_flg_clr(hBIT_ERR);
          (omitted)
          } else if( LinRxDataPtr[ucCurrentId.fields.id] != 0 ) {   ← If receive ID
              ucLinStatus = LIN_DATA_RECEPTION;   ← DATA receive wait state
          (omitted)
              vSetLinFreerunTimersCompare(ucRxCount);   ← Set free-running timer
          } else if ( LinTxDataPtr[ucCurrentId.fields.id] != 0 ) {   ← If transmit ID
          (omitted)
                                                ↓ Copy send data to buffer
              vLinWordCopy(ucUartTxBuffer, LinTxDataPtr[ucCurrentId.fields.id], ucTxCount);
              vSetLinFreerunTimersCompare(hTINFRAME_SPACE_IND);   ← Set free-running timer
          }
          (omitted)
   }
}
```

Figure 6-23 ID receive determination processing

④ DATA field

This section explains how data is sent and received using the DATA field.

First, for sending DATA, if the ID received using the ID field is the ID for sending, the vTimeoutCheckTask function as shown in “Figure 6-24 Timeout detection processing” is called by a free-running timer interrupt. This function is called when the timeout value configured in the free-running timer is detected, and in this case, the function is called when the timeout value from receiving the header to sending the response (response space) is detected.

The vTimeoutCheckTask function is divided into send pre-processing, initialization processing, etc. depending on the status information. When the status is the send pre-processing state, the first byte of the data is sent.

```
void        vTimeoutCheckTask(void){
    (omitted)
    if ( uiIntDemandCounter == 0 ) {
        switch ( ucLinStatus ) {
        case        LIN_PRETRANSMIT:          ← Pre-transmit state
            ucLinStatus = LIN_TRANSMIT;       ← State transition: Transmit DATA FIELD state
            ucSaveData = ucUartTxBuffer[0];   ← Transmit data: Acquire 1 byte
            l_ifc_tx(ucUartTxBuffer[0]);      ← Data transmit processing
            (omitted)
        case        LIN_UART_INITIAL:
                    (omitted)
        case        LIN_ID_RECEPTION:
                    (omitted)
        case        LIN_DATA_RECEPTION:
                    (omitted)
        case        LIN_TRANSMIT:
                    (omitted)
        case        LIN_WAIT_SYNCH_FIELD_START:
                    (omitted)
    }
}
```

Figure 6-24 Timeout detection processing

When the first byte of data is sent, a receive interrupt occurs due to receiving the data that the program itself sent. Therefore, in the same way as the operation for the ID field, the receive determination processing function _ifc_rx(l_ifc_handle rx_data) is called, and processing to send the second and subsequent data is performed based on the send state of the DATA FIELD as shown in “Figure 6-25 DATA send processing”, with this same process repeated over and over. Because the data byte count is set to 8 in the current LIN communication, after the 8th bytes of DATA has been sent the Checksum is finally sent and the send processing finishes.

```
void    l_ifc_rx(l_ifc_handle rx_data){
    switch(ucLinStatus){
    case  LIN_TRANSMIT:        ← Transmit DATA FIELD state
          if ( ucTxCurrentIndex < ucTxCount ){          ← If transmit DATA remains
          (omitted)
          l_ifc_tx(ucUartTxBuffer[ucTxCurrentIndex]);   ← Transmit processing
          (omitted)
          } else if ( ucTxCurrentIndex == ucTxCount ){  ← If all transmit data has been sent
          (omitted)
          l_ifc_tx(((unsigned char)~uiTxCheckSum));     ← Transmit checksum processing
          (omitted)
          }
    case  LIN_DATA_RECEPTION:
          (omitted)
    case  LIN_ID_RECEPTION:
          (omitted)
    case  LIN_WAKEUP_TRANSMIT:
          (omitted)
    }
}
```

Figure 6-25 DATA send processing

The data receive processing is explained next.

If the ID acquired in the ID receive processing is for receiving, the status is changed to the DATA receive state and the program waits to receive data from the bits pot yellow. If an interrupt occurs due to receiving data from the bits pot yellow, the receive processing is performed within the receive processing function l_ifc_rx(data) as shown in “Figure 6-26 DATA receive processing”. For the case of receiving data, the receive processing is performed by l_ifc_rx(l_ifc_handle rx_data) for each single byte of data received, and once all 8 bytes of data have been received, the receive success flag is set if there are no checksum errors and the receive processing finishes.

```
void      l_ifc_rx(l_ifc_handle rx_data){
    switch(ucLinStatus){
    case  LIN_TRANSMIT:
          (omitted)
    case  LIN_DATA_RECEPTION:                ← Receive DATA FIELD state
          if ( ucRxCurrentIndex >= ucRxCount ) {     ← If all data has been received
              if ( (uiRxCheckSum + rx_data) == 0xFF ) {    ← If checksum calculation is correct
                  (omitted)
                  flagsLinTxRx.bit.SucceedReception = SET;      ← Set receive success flag
                  memcpy( &ucUartRxFixedBuffer[0], &ucUartRxBuffer[0], ucRxCount );
                  (omitted)                                  ↑ Copy received data
              } else {                                       ← If there is a checksum error
                  l_flg_tst(hCHECKSUM_ERR);          ← Error processing
          } else {                                   ← If there is receive data remaining
              ucUartRxBuffer[ucRxCurrentIndex] = rx_data;      ← Store receive data in buffer
              (omitted)
    case  LIN_ID_RECEPTION:
          (omitted)
    case  LIN_WAKEUP_TRANSMIT:
          (omitted)
    }
}
```

Figure 6-26 DATA receive processing

Finally, the vBaseTimeTask function in main.c is for processing each received ID. This function is called periodically at fixed intervals, and primarily checks whether sending and receiving has finished. If this function is called when all of the data reception has finished (flagsLinTxRx.bit.SucceedReception is set), the sub_control function in submain.c as shown in “Figure 6-27 Submain processing 1” and “Figure 6-28 Submain processing 2” is called as receive completion processing that performs temperature sensor measurement processing, buzzer output processing, LED on/off processing, 7SEG display processing, and storage of sent data.

```
void sub_control(void){
 if(SW4 == SET){          ← Check the state of SW4 (for right side)
  switch (ucCurrentId.fields.id){
   case  0x00:            ← ID: 0x00
         ad_input2();                ← Start A/D interrupt (Acquire temperature sensor information)
         ucDATA00[0] = 0x55;
         ucDATA00[1] = temp2;        ← Transmit temperature sensor information
     break;
   case  0x01:            ← ID: 0x01
         LIN_temp_value = ucDATA01[2];          ← Acquire temperature information from slave
         led_seg2_display(LIN_temp_value);      ← Display temperature information on 7SEG
    break;
   case  0x02:            ← ID: 0x02
         ad_input0();                ← Start A/D interrupt (Acquire volume SW information)
         ucDATA02[0] = 0x55;
         ucDATA02[1] = Buzzer0;      ← Transmit volume SW information
    break;
   case  0x03:            ← ID: 0x03
         LIN_Buzzer_value  = ucDATA03[2];  ← Acquire volume SW data from slave
         buzzer_control();     ← Set PPG timer and output to buzzer
          if(LIN_Buzzer_value == 0){
           IO_PCN1.bit.OE = CLEAR;      ← No buzzer output
          }else{
           IO_PCN1.bit.OE = SET;        ← Buzzer output
          }
    break;
  case   0x04:            ← ID: 0x04
         led_control();              ← Check the state of SW3 and SW5, and display on LED
         ucDATA04[0] = 0x0F;
         count_clear();
    break;
  default:
    break;
```

Figure 6-27 Submain processing 1

```
void sub_control(void){
  (omitted)
 else if(SW4 == CLEAR){                ← Check state of SW4 (for left side)
  switch (ucCurrentId.fields.id){
  (omitted)
  case    0x04:                        ← ID: 0x04
          ucDATA04[0] = 0x55;
          led_control();   ← Check state of SW3 and SW5, and display on LEDs
           if(countup == SET || countdown == SET){
                  ucDATA04[3] = LED_current_value;        ← Send display data to LEDs
                  led_seg1_display(LED_current_value);    ← Display on 7SEG
           }
           else{
                  ucDATA04[3] = 0xff;
           }
          count_clear();
    break;
  case    0x05:
          if(ucDATA05[3] != 0xff){
                  LED_current_value = ucDATA05[3];        ← Receive display data for LED
                  led_display(LED_current_value);         ← Display on LED
                  led_seg1_display(LED_current_value);    ← Display on 7SEG
                  count_clear();
          }
    break;
  default:
    break;
```

Figure 6-28 Submain processing 2

7 Appendix

7.1 Sample program folder/file configuration

The folder/file configuration of the sample programs is shown below.

File/folder name	Description
bitspot_white_SampleProgram	
bitspot_white_SampleProgram.wsp	Softune workspace file
single_operation	Folder for single-unit operation
Debug	
ABS	
single_operation.abs	Sample program abs file
single_operation.mhx	Sample program Hex file
LST	
OBJ	
OPT	
Ext_int	
ADC.c	A/D converter initialization file
Ext_int.c	External interrupt initialization file
initial_clock.c	Internal clock initialization file
PPG_int.c	PPG timer initialization file
MB96350_IO	
_ffmc16.c	For header file definitions
_ffmc16.h	For header file definitions
_ffmc16_a.asm	Assembly language I/O definition file
mb96350.h	For header file definitions
mb96350_a.inc	I/O register definition file (assembly language)
ioreg.txt	I/O register usage guide (C language file, English version)
ioreg_a.txt	I/O register file usage guide (assembly language file, English version)
ioregj.txt	I/O register file usage guide (C language file, Japanese version)
ioregj_a.txt	I/O register file usage guide (assembly language file, Japanese version)
single_operation.dat	Softune settings file
single_operation.prj	Softune project file
main.c	Main source file
ROM_cfg_block.c	Settings file for running the monitor debugger
start907s.asm	Microcontroller start assembly file
CAN	Folder for CAN operation
Debug	
ABS	
CAN.abs	Sample program abs file
CAN.mhx	Sample program Hex file
LST	
OBJ	
OPT	
CAN__LIN_Board.sup	Debugger file
sim.sup	Debugger file

ADC	
AD.c	A/D converter initialization file
C_CAN_driver	
C_CAN_1_Driver_1.c	CAN initialization (interrupt levels, baud rate, etc.)
C_CAN_1_Driver_2.c	CAN initialization (message buffers)
C_CAN_1_Int.c	CAN interrupt processing function
Ext_Int	
Ext_Int.c	External interrupt initialization
Inc	
Ext_func_decla.h	External function declaration file
Ext_para_decla.h	External variable declaration file
MB96350	
_ffmc16.c	For header file definitions
_ffmc16.h	For header file definitions
_ffmc16_a.asm	Assembly language I/O definition file
mb96350.h	For header file definitions
mb96350_a.inc	I/O register definition file (assembly language)
ioreg.txt	I/O register usage guide (C language file, English version)
ioreg_a.txt	I/O register file usage guide (assembly language file, English version)
ioregj.txt	I/O register file usage guide (C language file, Japanese version)
ioregj_a.txt	I/O register file usage guide (assembly language file, Japanese version)
MCU_init	
Clock_config.c	MCU clock initialization
Port_LED	
Port_LED.c	Defines the function for driving the port
Reload_timer	
Reload_timer.c	Reload timer initialization and interrupt functions
Src	
_ffmc16.c	For header file definitions
main.c	Main source file
start907s.asm	Microcontroller start assembly file
Vct	
Intvect.c	Interrupt vector definitions
ROM_cfg_block.c	Settings file for running the monitor debugger
CAN.prj	
CAN__LIN_Board.dat	Softune settings file
LIN Master	
Debug	
ABS	
LIN MASTER.abs	Sample program abs file
LIN MASTER.mhx	Sample program Hex file
LST	
OBJ	
OPT	
MASTER.sup	Debugger file
sim.sup	Debugger file
INCLUDE	
jpn_eur.h	Header definition conversion file
define.h	Header definition file
lin.h	Header file for the LIN driver
linapi.h	Header file for the data communication API code

lindbcpu.h	Header file for CPU compatibility definitions
lindbmsg.h	Header file for LIN communication definitions (baud rate settings, ID setting, signal registration, etc.)
linhibios.h	LIN driver high level header file
linlobios.h	LIN driver low level header file
lindbmaster.h	Header file for separate LIN communication node definitions
Vector.h	Microcontroller header file
APPL	
start907s.asm	Microcontroller startup assembly file
ad.c	A/D converter file
extint.c	External interrupt processing function
main.c	Main source file
portled.c	Function definition file for driving the ports
ppg.c	PPG processing function
ROM_cfg_block.c	Display file for monitor debugger settings
submain.c	Driver low level source file (CPU resource control)
main.h	Main source header file
extern.h	External reference definition header file
DRIVER	
linapi.c	Data communication API code header file
linhibios.c	Driver high level source file (LIN protocol control)
linlobios.c	Driver low level source file (CPU resource control)
linmaster.c	LIN master communication node definition file
IOREG	
_ffmc16.c	For header file definitions
_ffmc16.h	For header file definitions
_ffmc16_a.asm	Assembly language I/O definition file
mb96350.h	For header file definitions
mb96350_a.inc	I/O register definition file (assembly language)
ioreg.txt	I/O register usage guide (C language file, English version)
ioreg_a.txt	I/O register file usage guide (assembly language file, English version)
ioregj.txt	I/O register file usage guide (C language file, Japanese version)
ioregj_a.txt	I/O register file usage guide (assembly language file, Japanese version)
LIN MASTER.prj	Softune project file
MASTER.dat	Softune settings file

Table 7-1 Folder/file structure of the sample programs