

To our customers,

Old Company Name in Catalogs and Other Documents

On April 1st, 2010, NEC Electronics Corporation merged with Renesas Technology Corporation, and Renesas Electronics Corporation took over all the business of both companies. Therefore, although the old company name remains in this document, it is a valid Renesas Electronics document. We appreciate your understanding.

Renesas Electronics website: <http://www.renesas.com>

April 1st, 2010
Renesas Electronics Corporation

Issued by: Renesas Electronics Corporation (<http://www.renesas.com>)

Send any inquiries to <http://www.renesas.com/inquiry>.

Notice

1. All information included in this document is current as of the date this document is issued. Such information, however, is subject to change without any prior notice. Before purchasing or using any Renesas Electronics products listed herein, please confirm the latest product information with a Renesas Electronics sales office. Also, please pay regular and careful attention to additional and different information to be disclosed by Renesas Electronics such as that disclosed through our website.
2. Renesas Electronics does not assume any liability for infringement of patents, copyrights, or other intellectual property rights of third parties by or arising from the use of Renesas Electronics products or technical information described in this document. No license, express, implied or otherwise, is granted hereby under any patents, copyrights or other intellectual property rights of Renesas Electronics or others.
3. You should not alter, modify, copy, or otherwise misappropriate any Renesas Electronics product, whether in whole or in part.
4. Descriptions of circuits, software and other related information in this document are provided only to illustrate the operation of semiconductor products and application examples. You are fully responsible for the incorporation of these circuits, software, and information in the design of your equipment. Renesas Electronics assumes no responsibility for any losses incurred by you or third parties arising from the use of these circuits, software, or information.
5. When exporting the products or technology described in this document, you should comply with the applicable export control laws and regulations and follow the procedures required by such laws and regulations. You should not use Renesas Electronics products or the technology described in this document for any purpose relating to military applications or use by the military, including but not limited to the development of weapons of mass destruction. Renesas Electronics products and technology may not be used for or incorporated into any products or systems whose manufacture, use, or sale is prohibited under any applicable domestic or foreign laws or regulations.
6. Renesas Electronics has used reasonable care in preparing the information included in this document, but Renesas Electronics does not warrant that such information is error free. Renesas Electronics assumes no liability whatsoever for any damages incurred by you resulting from errors in or omissions from the information included herein.
7. Renesas Electronics products are classified according to the following three quality grades: “Standard”, “High Quality”, and “Specific”. The recommended applications for each Renesas Electronics product depends on the product’s quality grade, as indicated below. You must check the quality grade of each Renesas Electronics product before using it in a particular application. You may not use any Renesas Electronics product for any application categorized as “Specific” without the prior written consent of Renesas Electronics. Further, you may not use any Renesas Electronics product for any application for which it is not intended without the prior written consent of Renesas Electronics. Renesas Electronics shall not be in any way liable for any damages or losses incurred by you or third parties arising from the use of any Renesas Electronics product for an application categorized as “Specific” or for which the product is not intended where you have failed to obtain the prior written consent of Renesas Electronics. The quality grade of each Renesas Electronics product is “Standard” unless otherwise expressly specified in a Renesas Electronics data sheets or data books, etc.
 - “Standard”: Computers; office equipment; communications equipment; test and measurement equipment; audio and visual equipment; home electronic appliances; machine tools; personal electronic equipment; and industrial robots.
 - “High Quality”: Transportation equipment (automobiles, trains, ships, etc.); traffic control systems; anti-disaster systems; anti-crime systems; safety equipment; and medical equipment not specifically designed for life support.
 - “Specific”: Aircraft; aerospace equipment; submersible repeaters; nuclear reactor control systems; medical equipment or systems for life support (e.g. artificial life support devices or systems), surgical implantations, or healthcare intervention (e.g. excision, etc.), and any other applications or purposes that pose a direct threat to human life.
8. You should use the Renesas Electronics products described in this document within the range specified by Renesas Electronics, especially with respect to the maximum rating, operating supply voltage range, movement power voltage range, heat radiation characteristics, installation and other product characteristics. Renesas Electronics shall have no liability for malfunctions or damages arising out of the use of Renesas Electronics products beyond such specified ranges.
9. Although Renesas Electronics endeavors to improve the quality and reliability of its products, semiconductor products have specific characteristics such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Further, Renesas Electronics products are not subject to radiation resistance design. Please be sure to implement safety measures to guard them against the possibility of physical injury, and injury or damage caused by fire in the event of the failure of a Renesas Electronics product, such as safety design for hardware and software including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other appropriate measures. Because the evaluation of microcomputer software alone is very difficult, please evaluate the safety of the final products or system manufactured by you.
10. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. Please use Renesas Electronics products in compliance with all applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive. Renesas Electronics assumes no liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.
11. This document may not be reproduced or duplicated, in any form, in whole or in part, without prior written consent of Renesas Electronics.
12. Please contact a Renesas Electronics sales office if you have any questions regarding the information contained in this document or Renesas Electronics products, or if you have any other inquiries.

(Note 1) “Renesas Electronics” as used in this document means Renesas Electronics Corporation and also includes its majority-owned subsidiaries.

(Note 2) “Renesas Electronics product(s)” means any product developed or manufactured by or for Renesas Electronics.



Application Note

μ PD78F0730

8-bit Single-Chip Microcontroller

USB-to-Serial Conversion Software

[MEMO]

MINICUBE is a registered trademark of NEC Electronics Corporation in Japan and Germany or a trademark in the United States of America.

Windows and Windows Vista are registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

PC/AT is a trademark of International Business Machines Corporation.

Other company names and product names described in this document are trademarks or registered trademarks of the respective company.

- **The information in this document is current as of March, 2009. The information is subject to change without notice. For actual design-in, refer to the latest publications of NEC Electronics data sheets or data books, etc., for the most up-to-date specifications of NEC Electronics products. Not all products and/or types are available in every country. Please check with an NEC Electronics sales representative for availability and additional information.**

- No part of this document may be copied or reproduced in any form or by any means without the prior written consent of NEC Electronics. NEC Electronics assumes no responsibility for any errors that may appear in this document.
- NEC Electronics does not assume any liability for infringement of patents, copyrights or other intellectual property rights of third parties by or arising from the use of NEC Electronics products listed in this document or any other liability arising from the use of such products. No license, express, implied or otherwise, is granted under any patents, copyrights or other intellectual property rights of NEC Electronics or others.
- Descriptions of circuits, software and other related information in this document are provided for illustrative purposes in semiconductor product operation and application examples. The incorporation of these circuits, software and information in the design of a customer's equipment shall be done under the full responsibility of the customer. NEC Electronics assumes no responsibility for any losses incurred by customers or third parties arising from the use of these circuits, software and information.
- While NEC Electronics endeavors to enhance the quality, reliability and safety of NEC Electronics products, customers agree and acknowledge that the possibility of defects thereof cannot be eliminated entirely. To minimize risks of damage to property or injury (including death) to persons arising from defects in NEC Electronics products, customers must incorporate sufficient safety measures in their design, such as redundancy, fire-containment and anti-failure features.
- NEC Electronics products are classified into the following three quality grades: "Standard", "Special" and "Specific".

The "Specific" quality grade applies only to NEC Electronics products developed based on a customer-designated "quality assurance program" for a specific application. The recommended applications of an NEC Electronics product depend on its quality grade, as indicated below. Customers must check the quality grade of each NEC Electronics product before using it in a particular application.

"Standard": Computers, office equipment, communications equipment, test and measurement equipment, audio and visual equipment, home electronic appliances, machine tools, personal electronic equipment and industrial robots.

"Special": Transportation equipment (automobiles, trains, ships, etc.), traffic control systems, anti-disaster systems, anti-crime systems, safety equipment and medical equipment (not specifically designed for life support).

"Specific": Aircraft, aerospace equipment, submersible repeaters, nuclear reactor control systems, life support systems and medical equipment for life support, etc.

The quality grade of NEC Electronics products is "Standard" unless otherwise expressly specified in NEC Electronics data sheets or data books, etc. If customers wish to use NEC Electronics products in applications not intended by NEC Electronics, they must contact an NEC Electronics sales representative in advance to determine NEC Electronics' willingness to support a given application.

(Note)

- (1) "NEC Electronics" as used in this statement means NEC Electronics Corporation and also includes its majority-owned subsidiaries.
- (2) "NEC Electronics products" means any product developed or manufactured by or for NEC Electronics (as defined above).

PREFACE

Readers	This application note is intended for users who understand the features of the μ PD78F0730, and are going to develop application systems using this product.	
Purpose	This application note is intended to give users an understanding of the specifications of the sample software provided for using the USB function controller incorporated in the μ PD78F0730.	
Organization	This application note is broadly divided into the following four sections: <ul style="list-style-type: none">• An overview of the μPD78F0730 USB function controller• An overview of the USB standard• The specifications for the sample software• How to use the sample software	
How to Read This Manual	It is assumed that the readers of this application note have general knowledge in the fields of electrical engineering, logic circuits, and microcontrollers. <ul style="list-style-type: none">• To learn about the hardware features and electrical specifications of the μPD78F0730 → See the separately provided μPD78F0730 Hardware User's Manual.• To learn about the instructions of the μPD78F0730 → See the separately provided 78K/0 Series Instructions User's Manual.	
Conventions	Data significance:	Higher digits on the left and lower digits on the right
	Note:	Footnote for item marked with Note in the text
	Caution:	Information requiring particular attention
	Remark:	Supplementary information
	Numeric representation:	Binary or decimal ... XXXX Hexadecimal ... 0XXXXX
	Prefix indicating power of 2 (address space, memory capacity):	K (kilo): $2^{10} = 1,024$ M (mega): $2^{20} = 1,024^2$ G (giga): $2^{30} = 1,024^3$ T (tera): $2^{40} = 1,024^4$ P (peta): $2^{50} = 1,024^5$ E (exa): $2^{60} = 1,024^6$

Related Documents

The related documents indicated in this publication may include preliminary versions. However, preliminary versions are not marked as such.

- Documents Related to the μ PD78F0730

Document Name	Document No.
μ PD78F0730 User's Manual	U19014E
μ PD78F0730 USB-to-Serial Conversion Driver User's Manual	U19340E
78K/0 Series Instructions User's Manual	U12326E

- Documents Related to Development Tools (User's Manuals)

Document Name		Document No.
CC78K0 Ver. 3.70 C Compiler	Operation	U17201E
	Language	U17200E
RA78K0 Ver. 3.80 Assembler Package	Operation	U17199E
	Language	U17198E
	Structured Assembly Language	U17197E
SM+ System Simulator	Operation	U18601E
	User Open Interface	U18212E
SM78K Ver. 2.52 System Simulator	Operation	U16768E
PM plus Ver. 5.10 Project Manager		U16569E
ID78K0-NS Ver. 2.70 Integrated Debugger	Operation	U17729E
ID78K0-QB Ver. 3.00 Integrated Debugger	Operation	U18492E
QB-780731 In-Circuit Emulator		U17804E
QB-MINI2 On-Chip Debug Emulator with Programming Function		U18371E
PG-FP5 Flash Memory Programmer		U18865E
PG-FP4 Flash Memory Programmer		U15260E

CONTENTS

CHAPTER 1 OVERVIEW.....	9
1.1 Overview	9
1.1.1 Features of the USB function controller.....	9
1.1.2 Features of the sample software	10
1.1.3 Files included in the sample software.....	10
1.2 Overview of the μPD78F0730.....	11
CHAPTER 2 OVERVIEW OF USB.....	12
2.1 Transfer Format	12
2.2 Endpoints.....	13
2.3 Device Class.....	13
2.4 Requests	13
2.4.1 Types	14
2.4.2 Format.....	15
2.5 Descriptor	15
2.5.1 Types	15
2.5.2 Format.....	16
CHAPTER 3 SAMPLE SOFTWARE SPECIFICATIONS.....	18
3.1 Overview	18
3.1.1 Features	18
3.1.2 System setup	19
3.1.3 Processing flow	20
3.1.4 Supported requests	22
3.1.5 Descriptor settings.....	24
3.2 CPU Initialization Processing	28
3.3 USB Control Processing	29
3.3.1 USBF initialization processing	30
3.3.2 USBF interrupt servicing (INTUSB0B).....	33
3.3.3 USBF reception interrupt servicing (INTUSB1B)	35
3.3.4 USB transmission data storage processing.....	36
3.3.5 USB data transmission.....	37
3.4 UART Control Processing.....	38
3.4.1 UART initialization processing.....	39
3.4.2 UART operation mode specification processing.....	40
3.4.3 UART reception interrupt servicing.....	42
3.4.4 UART reception error interrupt servicing	42
3.4.5 UART data transmission	43
3.4.6 UART operation mode.....	44
3.5 Bridge Processing Between the UART and USB.....	45
3.5.1 Storing the data received by the UART into the USB transmission buffer	45
3.5.2 Transmitting the data in the USB reception FIFO from the UART	45
3.5.3 Main routine.....	46
3.6 Vendor Request Format	47
3.6.1 LINE_CONTROL.....	47

3.6.2	SET_DTR_RTS	48
3.6.3	SET_XON_XOFF_CHR	48
3.6.4	OPEN_CLOSE	49
3.6.5	SET_ERR_CHR	49
3.7	Function Specifications.....	50
3.7.1	Functions.....	50
3.7.2	Correlation of the functions.....	51
3.7.3	Function features.....	53
3.8	Data Structures.....	69
CHAPTER 4	DEVELOPMENT ENVIRONMENT	70
4.1	Used Products	70
4.1.1	System components	70
4.1.2	Program development	71
4.1.3	Debugging	71
4.2	Setting Up the Environment.....	72
4.2.1	Preparing the environment	72
4.2.2	Preparing the host environment.....	73
4.3	On-Chip Debugging	81
4.3.1	Generating a load module	81
4.3.2	Loading and executing the load module	81
4.3.3	Connecting the USB port (virtual COM port).....	83
4.3.4	Connecting the RS-232C port.....	87
4.3.5	Checking the operation.....	88
4.4	Cautions	93
4.4.1	Recommended communication speed.....	93
4.4.2	Causes of data loss	93
CHAPTER 5	USING THE SAMPLE SOFTWARE.....	94
5.1	Overview	94
5.2	Customizing the Sample Software	94
5.2.1	Application section.....	94
5.2.2	Setting up the registers.....	95
5.2.3	Descriptor information.....	96
5.2.4	Setting up the virtual COM port host driver.....	96
APPENDIX A	TARGET BOARD.....	101
A.1	Overview	101
A.2	Circuit Example	102

CHAPTER 1 OVERVIEW

This application note describes the USB-to-serial conversion sample software created for the USB function controller incorporated in the μ PD78F0730 microcontroller.

This application note provides the following information:

- The specifications for the sample software
- Information about the environment used to develop an application program by using the sample software
- The reference information provided for using the sample software

This chapter provides an overview of the sample software and describes the microcontroller for which the sample software can be used.

1.1 Overview

1.1.1 Features of the USB function controller

The USB function controller (USBF) incorporated in the μ PD78F0730 has the following features:

- Conforms to the Universal Serial Bus Specification.
- Supports 12 Mbps (full speed) transfer.
- Incorporates transfer endpoints.

Table 1-1. Configuration of the Endpoints of the USB Function Controller Incorporated in the μ PD78F0730

Endpoint Name	FIFO Size (Bytes)	Transfer Type	Remark
Endpoint0 Read	64	Control transfer (IN)	–
Endpoint0 Write	64	Control transfer (OUT)	–
Endpoint1	64×2	Bulk transfer 1 (IN)	Dual-buffer configuration
Endpoint2	64×2	Bulk transfer 1 (OUT)	Dual-buffer configuration

- The internal or external clock can be selected ($f_{\text{USB}} = 48 \text{ MHz}$)^{Note}

The clock signal generated by using the X1 oscillator ($f_x = 12$ or 16 MHz) is multiplied by 4 or 3.

An external input clock ($f_{\text{EXCLK}} = 12$ or 16 MHz) is multiplied by 4 or 3.

Note The sample software selects the internal clock.

1.1.2 Features of the sample software

The sample software has the features below. For details about the features and operations, see **CHAPTER 3 SAMPLE SOFTWARE SPECIFICATIONS**.

- Operates as a virtual COM port.
- Uses a dedicated host driver.
- Directly transmits the data received by the USB function controller from the UART.
- Directly transmits the data received by the UART from the USB function controller.
- The baud rate, stop bit, data length, and parity bit can be changed by using the terminal software.
- Operates as the vendor class and uses three endpoints (Control, Bulk In, and Bulk Out).
- Does not support suspending or resuming.
- Operates as a bus-powered device.
- Exclusively uses the following amounts of memory (excluding the vector table):
ROM: About 4.2 KB
RAM: About 0.3 KB

1.1.3 Files included in the sample software

The sample software includes the following files:

Table 1-2. Files Included in the Sample Software

Folder	File	Overview
src	main.c	Initialization and main routine
	usbf78k.c	USBF initialization, interrupt servicing, bulk transfer, and control transfer
	uart_ctrl.c	UART communication control
	usbf78k_vendor.c	Vendor class processing
	boot.asm	Boot processing routines
include	errno.h	Error code definitions
	main.h	main.c function prototype declarations
	Types.h	User-defined type declarations
	uart_ctrl.h	uart_ctrl.c function prototype declarations
	usbf78k.h	usbf78k.c function prototype declarations
	usbf78k_desc.h	Descriptor definitions
	usbf78k_sfr.h	Macro definitions for accessing the USB function controller registers
	usbf78k_vendor.h	usbf78k_vendor.c function prototype declarations

Remark In addition, the project-related files generated when creating a development environment by using the USB-to-serial conversion host driver or PM+ (an integrated development tool made by NEC Electronics) are also included. For details, see **4.2.2 Preparing the host environment**.

1.2 Overview of the μ PD78F0730

This section describes the μ PD78F0730, which is controlled by using the sample software.

The μ PD78F0730 is an 8-bit single-chip microcontroller made by NEC Electronics. It has peripherals such as ROM, RAM, timers, counters, a serial interface, an A/D converter, a D/A converter, a DMA controller, and a USB function controller. For details, see the **μ PD78F0730 8-bit Single-Chip Microcontroller User's Manual**.

The μ PD78F0730 has the following main features:

- Can execute instructions at high speeds (in 0.125 μ s when it operates at 16 MHz based on the high-speed system clock).
- General-purpose registers: 8 bits \times 32 registers (8 bits \times 8 registers \times 4 banks)
- ROM and RAM capacities

Part Number	Item	Data Memory	
	Program Memory (ROM)	Internal High-Speed RAM ^{Note}	Internal Expansion RAM ^{Note}
μ PD78F0730	Flash Memory ^{Note} 16 KB	1 KB	2 KB

Note The capacity of the internal flash memory, internal high-speed RAM, and internal expansion RAM can be changed using registers.

- Has a USB function controller (USBF).
- Has a single-power-supply flash memory.
- Can perform self programming (boot swapping).
- Can perform on-chip debugging.
- Has a power-on-clear (POC) circuit and a low-voltage detector (LVI).
- Has a watchdog timer (that can operate based on the internal low-speed oscillation clock).
- Has 19 I/O ports (including two N-ch open drain ports).
- Has the following five timer channels:
 - One 16-bit timer/event counter channel
 - Two 8-bit timer/event counter channels
 - One 8-bit timer channel
 - One watchdog timer channel
- Has the following three serial interface channels:
 - One UART channel
 - One CSI channel
 - One USB channel

CHAPTER 2 OVERVIEW OF USB

This chapter provides an overview of the USB standard, which the sample software conforms to.

USB (Universal Serial Bus) is an interface standard for connecting various peripherals to a host by using the same type of connector. The USB interface is more flexible and easier to use than older interfaces in that it can connect up to 127 devices by adding a branching point known as a hub, and supports the hot-plug feature, which enables devices to be recognized by Plug & Play. The USB interface is provided in most current PCs and has become the standard for connecting peripherals to a PC.

The USB standard is formulated and managed by the USB Implementers Forum (USB-IF). For details about the USB standard, see the official USB-IF website (www.usb.org).

2.1 Transfer Format

Four types of transfer formats (interrupt, bulk, isochronous, and control) are defined in the USB standard. Table 2-1 shows the features of each transfer format.

Table 2-1. USB Transfer Format

Transfer Format		Control Transfer	Bulk Transfer	Interrupt Transfer	Isochronous Transfer
Item					
Feature		Transfer format used to exchange information required for controlling peripheral devices	Transfer format used to aperiodically handle large amounts of data	Periodic data transfer format that has a low band width	Transfer format used for a real-time transfer
Specifiable packet size	High speed 480 Mbps	64 bytes	512 bytes	1 to 1,024 bytes	1 to 1,024 bytes
	Full speed 12 Mbps	8, 16, 32, or 64 bytes	8, 16, 32, or 64 bytes	1 to 64 bytes	1 to 1,023 bytes
	Low speed 1.5 Mbps	8 bytes	—	1 to 8 bytes	—
Transfer priority		3	3	2	1

2.2 Endpoints

An endpoint is an information unit that is used by the host to specify a communicating device and is specified using a number from 0 to 15 and a direction (IN or OUT). An endpoint must be provided for every data communication path that is used for a peripheral device and cannot be shared by multiple communication paths^{Note}. For example, a device that can write to and read from an SD card and print out documents must have a separate endpoint for each purpose. Endpoint 0 is used to control transfers for any type of device.

During data communication, the host uses a USB device address, which specifies the device, and an endpoint (a number and direction) to specify the communication destination in the device.

Peripheral devices have buffer memory that is a physical circuit to be used for the endpoint and functions as a FIFO that absorbs the difference in speed of the USB and communication destination (such as memory).

Note An endpoint can be exclusively switched by using the alternative setting.

2.3 Device Class

Various device classes, such as the mass storage class (MSC), printer class, and human interface device class (HID), are defined according to the functions of the peripheral devices connected via USB (the function devices). A common host driver can be used if the connected devices conform to the standard specifications of the relevant device class, which is defined by a protocol. A separate driver is not necessary for each device, enabling users to connect any device and vendors to save labor hours for developing application programs.

2.4 Requests

For the USB standard, communication starts with the host issuing a command, known as a request, to all function devices. A request includes data such as the direction and type of processing and address of the function device. Each function device decodes the request, judges whether the request is addressed to it, and responds only if the request is addressed to it.

2.4.1 Types

There are three types of requests: standard requests, class requests, and vendor requests.

For details about requests that the sample software supports, see **3.1.4 Supported requests**.

(1) Standard requests

Standard requests are used for all USB-compatible devices. A request is a standard request if the values of bits 6 and 5 in the `bmRequestType` field are both 0. For details about the processing of standard requests, see the **Universal Serial Bus Specification Rev. 2.0**.

Table 2-2. Standard Requests

Request Name	Target Descriptor	Overview
GET_STATUS	Device	Reads the settings of the power supply (self or bus) and remote wakeup.
	Endpoint	Reads the halt status.
CLEAR_FEATURE	Device	Clears remote wakeup.
	Endpoint	Cancels the halt status (DATA PID = 0).
SET_FEATURE	Device	Specifies remote wakeup or test mode.
	Endpoint	Specifies the halt status.
GET_DESCRIPTOR	Device, configuration, string	Reads the target descriptor.
SET_DESCRIPTOR	Device, configuration, string	Changes the target descriptor (optional).
GET_CONFIGURATION	Device	Reads the currently specified configuration values.
SET_CONFIGURATION	Device	Specifies the configuration values.
GET_INTERFACE	Interface	Reads the alternatively specified value among the currently specified values of the target interface.
SET_INTERFACE	Interface	Specifies the alternatively specified value of the target interface.
SET_ADDRESS	Device	Specifies the USB address.
SYNCH_FRAME	Endpoint	Reads frame-synchronous data.

(2) Class requests

Class requests are unique to the device class. Class requests can be supported by using a common host driver. A request is a class request if the values of bits 6 and 5 in the `bmRequestType` field are 0 and 1, respectively.

(3) Vendor requests

Vendor requests are requests that are uniquely defined by each vendor. To make vendor requests available for use, the vendor must provide a host driver that supports the requests. A request is a vendor request if bits 6 and 5 in the `bmRequestType` field are 1 and 0, respectively.

2.4.2 Format

USB requests have an 8-byte length and consist of the following fields:

Table 2-3. USB Request Format

Offset	Field		Description
0	bmRequestType		Request attribute
		Bit 7	Data transfer direction
		Bits 6 and 5	Request type
		Bits 4 to 0	Target descriptor
1	bRequest		Request code
2	wValue	Lower	Any value used by the request
3		Higher	
4	wIndex	Lower	Index or offset used by the request
5		Higher	
6	wLength	Lower	Number of bytes transferred at the data stage (the data length)
7		Higher	

2.5 Descriptor

For the USB standard, a descriptor is information that is specific to a function device and is encoded in a specified format. A function device transmits a descriptor in response to a request transmitted from the host.

2.5.1 Types

The following five types of descriptors are defined:

- Device descriptor

This descriptor exists in every device and includes basic information such as the supported USB specification version, device class, protocol, maximum packet length that can be used when transferring data to endpoint 0, vendor ID, and product ID.

This descriptor is transmitted in response to a GET_DESCRIPTOR_Device request.

- Configuration descriptor

At least one configuration descriptor exists in every device and includes information such as the device attribute (power supply method) and power consumption.

This descriptor is transmitted in response to a GET_DESCRIPTOR_Configuration request.

- Interface descriptor

This descriptor is required for each interface and includes information such as the interface identification number, interface class, and supported number of endpoints.

This descriptor is transmitted in response to a GET_DESCRIPTOR_Configuration request.

- Endpoint descriptor

This descriptor is required for each endpoint specified for an interface descriptor and defines the transfer type (direction), maximum packet length that can be used for a transfer, and transfer interval. However, endpoint 0 does not have this descriptor.

This descriptor is transmitted in response to a GET_DESCRIPTOR_Configuration request.

- String descriptor

This descriptor includes any character string.

This descriptor is transmitted in response to a GET_DESCRIPTOR_String request.

2.5.2 Format

The size and fields of each descriptor type vary as described below.

Remark The data sequence of each field is in little endian format.

Table 2-4. Device Descriptor Format

Field	Size (Bytes)	Description
bLength	1	Descriptor size
bDescriptorType	1	Descriptor type
bcdUSB	2	USB specification release number
bDeviceClass	1	Class code
bDeviceSubClass	1	Subclass code
bDeviceProtocol	1	Protocol code
bMaxPacketSize0	1	Maximum packet size of endpoint 0
idVendor	2	Vendor ID
idProduct	2	Product ID
bcdDevice	2	Device release number
iManufacturer	1	Index to the string descriptor representing the manufacturer
iProduct	1	Index to the string descriptor representing the product
iSerialNumber	1	Index to the string descriptor representing the device production number
bNumConfigurations	1	Number of configurations

Remark Vendor ID: The identification number each company that develops a USB device acquires from USB-IF

Product ID: The identification number each company assigns to a product after acquiring the vendor ID

Table 2-5. Configuration Descriptor Format

Field	Size (Bytes)	Description
bLength	1	Descriptor size
bDescriptorType	1	Descriptor type
wTotalLength	2	Total number of bytes of the configuration, interface, and endpoint descriptors
bNumInterfaces	1	Number of interfaces in this configuration
bConfigurationValue	1	Identification number of this configuration
iConfiguration	1	Index to the string descriptor specifying the source code for this configuration
bmAttributes	1	Features of this configuration
bMaxPower	1	Maximum current consumed in this configuration (in 2 μ A units)

Table 2-6. Interface Descriptor Format

Field	Size (Bytes)	Description
bLength	1	Descriptor size
bDescriptorType	1	Descriptor type
bInterfaceNumber	1	Identification number of this interface
bAlternateSetting	1	Whether the alternative settings are specified for this interface
bNumEndpoints	1	Number of endpoints of this interface
bInterfaceClass	1	Class code
bInterfaceSubClass	1	Subclass code
bInterfaceProtocol	1	Protocol code
iInterface	1	Index to the string descriptor specifying the source code for this interface

Table 2-7. Endpoint Descriptor Format

Field	Size (Bytes)	Description
bLength	1	Descriptor size
bDescriptorType	1	Descriptor type
bEndpointAddress	1	Transfer direction of this endpoint Address of this endpoint
bmAttributes	1	Transfer type of this endpoint
wMaxPacketSize	2	Maximum packet size of this transfer
bInterval	1	Polling interval of this endpoint

Table 2-8. String Descriptor Format

Field	Size (Bytes)	Description
bLength	1	Descriptor size
bDescriptorType	1	Descriptor type
bString	Any	Any data string

CHAPTER 3 SAMPLE SOFTWARE SPECIFICATIONS

This chapter provides details about the features and processing of the USB-to-serial conversion sample software for the μ PD78F0730 and the specifications of the functions provided in the μ PD78F0730.

3.1 Overview

3.1.1 Features

The sample software can perform the following processing:

(1) Initialization

This processing specifies the memory size and clock of the μ PD78F0730. For details, see **3.2 CPU Initialization Processing**.

(2) USB control processing

This processing initializes the USBF, executes an interrupt handler, and performs data transmission and reception. For details, see **3.3 USB Control Processing**.

This processing also responds to USB requests to which the μ PD78F0730 does not automatically respond. For details, see **3.1.4 Supported requests**.

(3) UART control processing

This processing initializes the UART, executes an interrupt handler, and performs data transmission and reception. For details, see **3.4 UART Control Processing**.

(4) Bridge processing between the UART and USB

During the processing to store the data received by the UART into the USB transmission buffer, the received data that is read during the UART reception completion interrupt servicing is stored in the USB transmission buffer.

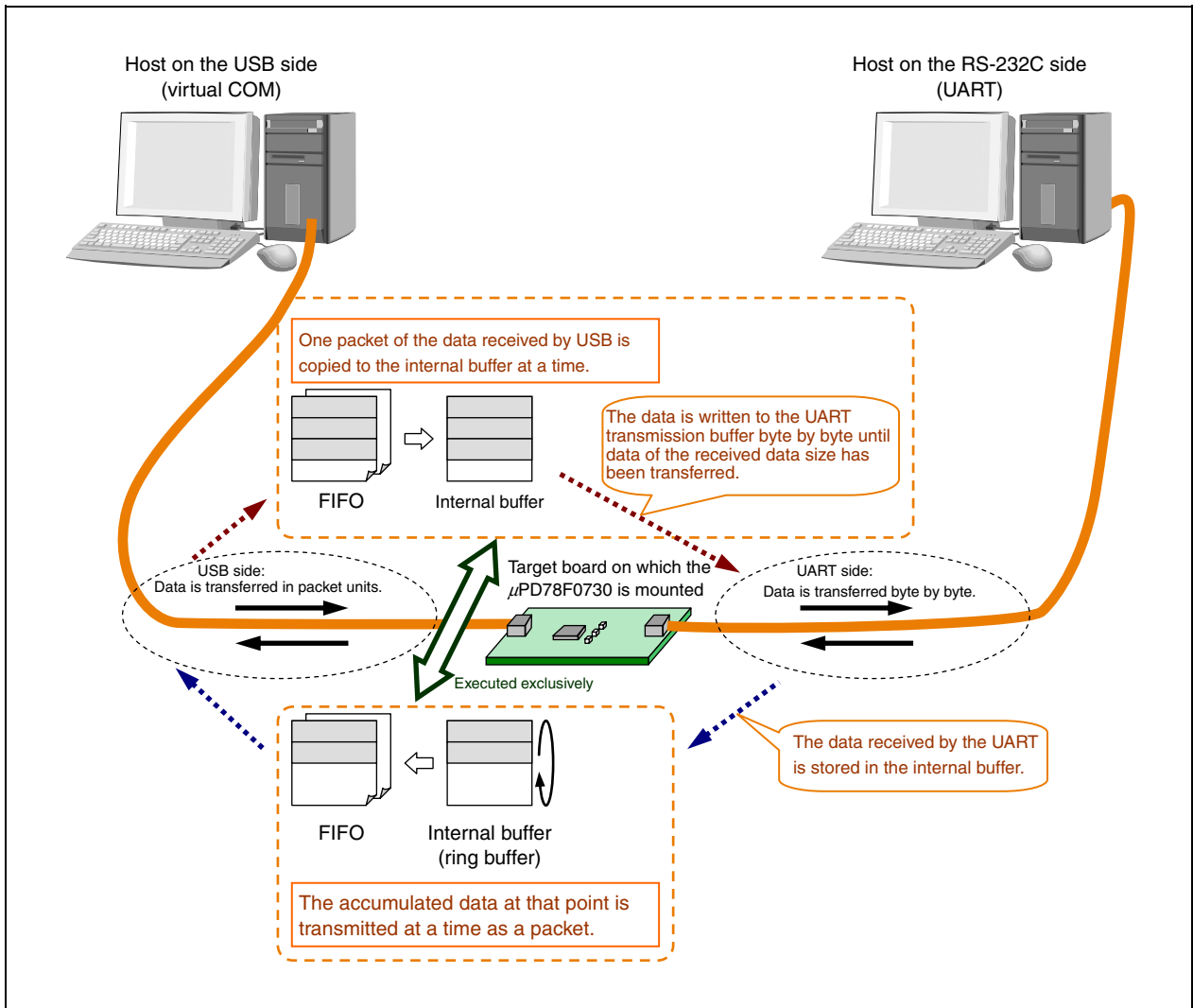
During the processing to transmit the data in the USB reception FIFO from the UART, the data at the endpoint for a bulk out transfer (reception) is read and transmitted from the UART.

For details, see **3.5 Bridge Processing Between the UART and USB**.

3.1.2 System setup

Figure 3-1 shows the setup and processing overview of a system that uses the sample software.

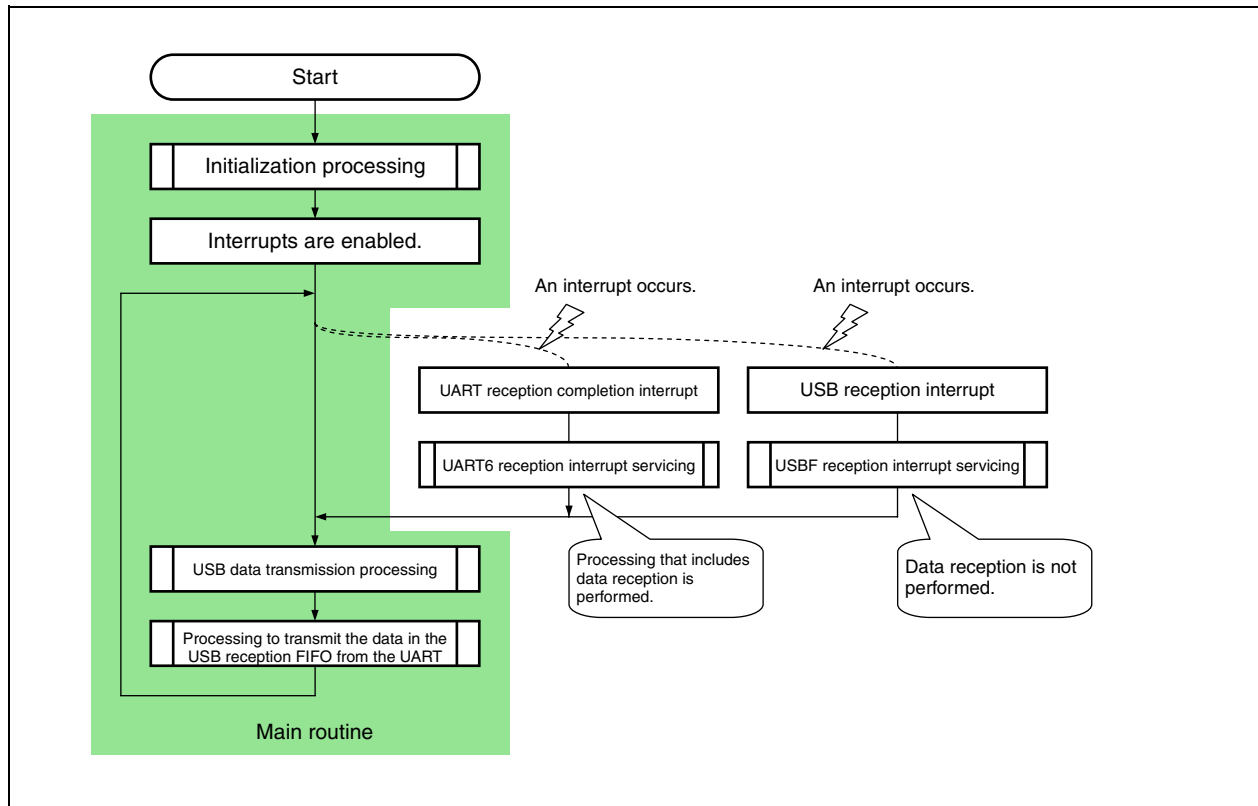
Figure 3-1. System Setup and Processing Overview



3.1.3 Processing flow

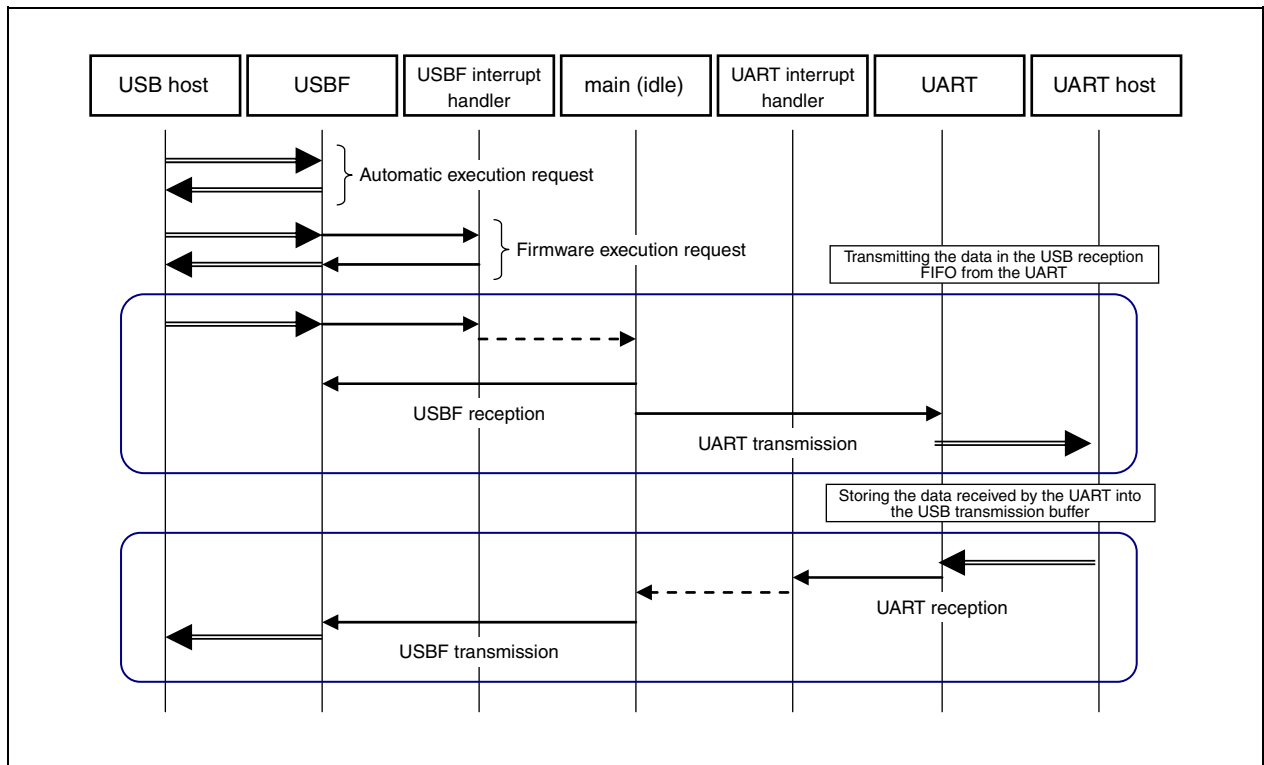
The processing shown in Figures 3-2 and 3-3 is performed by executing the sample software.

Figure 3-2. Processing Flow of the Sample Software (1)



- <1> When the initialization performed when the board is turned on ends, loop processing (within the main function (main)) and interrupt servicing by using various interrupt signals start. After the host and target board are connected via USB, data transmission and reception between the UART and USB starts.
- <2> If data is received via USB, the USBF reception interrupt handler starts when a USB reception completion interrupt occurs. This handler sets only the reception completion flag and cannot read data. After the interrupt servicing ends, the function for transmitting the data in the USB reception FIFO from the UART (`usbf78k_usb_to_uart`) is called within the main function (main) and USB reception and UART transmission are performed.
- <3> If data is received via UART, the UART reception completion interrupt handler starts when a UART reception completion interrupt occurs. This handler reads the received data, calls the function for storing the data received by the UART into the USB transmission buffer (`usbf78k_uart_to_usb`), and stores the data in the internal buffer. The stored data is transmitted from USB by calling the function for transmitting USB data to the ring buffer (`usbf78k_send_txbuf`) within the main function (main).

Figure 3-3. Processing Flow of the Sample Software (2)



<1> Automatic execution request

The hardware (the μ PD78F0730) automatically responds to this request.

<2> Firmware execution request

The firmware (the sample software) responds to this request when executing the USBF interrupt handler.

<3> Transmitting the data in the USB reception FIFO from the UART

Reception is identified by executing the USBF interrupt handler and the data received via USB is transmitted from the UART.

<4> Storing the data received by the UART into the USB transmission buffer

The received data is stored in the buffer and transmitted from USB by executing the UART reception completion interrupt handler.

3.1.4 Supported requests

Table 3-1 shows the USB requests defined by the hardware (the μ PD78F0730) and firmware (the sample software).

Table 3-1. USB Request Processing

Request Name	Codes								Processing
	0	1	2	3	4	5	6	7	
Standard request									
GET_INTERFACE	0x81	0x0A	0x00	0x00	0xXX	0xXX	0x01	0x00	Automatic hardware response
GET_CONFIGURATION	0x80	0x08	0x00	0x00	0x00	0x00	0x01	0x00	Automatic hardware response
GET_DESCRIPTOR Device	0x80	0x06	0x00	0x01	0x00	0x00	0xXX	0xXX	Automatic hardware response
GET_DESCRIPTOR Configuration	0x80	0x06	0x00	0x02	0x00	0x00	0xXX	0xXX	Automatic hardware response
GET_DESCRIPTOR String	0x80	0x06	0x00	0x03	0x00	0x00	0xXX	0xXX	Firmware response
GET_STATUS Device	0x80	0x00	0x00	0x00	0x00	0x00	0x02	0x00	Automatic hardware response
GET_STATUS Interface	0x81	0x00	0x00	0x00	0xXX	0xXX	0x02	0x00	Automatic hardware STALL response
GET_STATUS Endpoint n	0x82	0x00	0x00	0x00	0xXX	0xXX	0x02	0x00	Automatic hardware response
CLEAR_FEATURE Device	0x00	0x01	0x01	0x00	0x00	0x00	0x00	0x00	Automatic hardware response
CLEAR_FEATURE Interface	0x01	0x01	0x00	0x00	0xXX	0xXX	0x00	0x00	Automatic hardware STALL response
CLEAR_FEATURE Endpoint n	0x02	0x01	0x00	0x00	0xXX	0xXX	0x00	0x00	Automatic hardware response
SET_DESCRIPTOR	0x00	0x07	0xXX	0xXX	0xXX	0xXX	0xXX	0xXX	Firmware STALL response
SET_FEATURE Device	0x00	0x03	0x01	0x00	0x00	0x00	0x00	0x00	Automatic hardware response
SET_FEATURE Interface	0x02	0x03	0xXX	0xXX	0xXX	0xXX	0x00	0x00	Automatic hardware STALL response
SET_FEATURE Endpoint n	0x02	0x03	0x00	0x00	0xXX	0xXX	0x00	0x00	Automatic hardware response
SET_INTERFACE	0x01	0x0B	0xXX	0xXX	0xXX	0xXX	0x00	0x00	Automatic hardware response
SET_CONFIGURATION	0x00	0x09	0xXX	0xXX	0x00	0x00	0x00	0x00	Automatic hardware response
SET_ADDRESS	0x00	0x05	0xXX	0xXX	0x00	0x00	0x00	0x00	Automatic hardware response
Vendor request									
LINE_CONTROL	0x40	0x0B	0x00	0x00	0x00	0x00	0x06	0x00	Firmware response
SET_DTR_RTS	0x40	0x0B	0x00	0x00	0x00	0x00	0x02	0x00	Firmware response
SET_XON_XOFF_CHR	0x40	0x0B	0x00	0x00	0x00	0x00	0x03	0x00	Firmware response
OPEN_CLOSE	0x40	0x0B	0x00	0x00	0x00	0x00	0x02	0x00	Firmware response
SET_ERR_CHR	0x40	0x0B	0x00	0x00	0x00	0x00	0x03	0x00	Firmware response
Other requests	Other than the above								Firmware STALL response

Remark 0xXX: Undefined value

(1) Standard requests

The sample software performs the following response processing for requests to which the hardware (the μ PD78F0730) does not automatically respond.

(a) GET_DESCRIPTOR_string

The host issues this request to acquire the string descriptor of a function device.

If this request is received, the sample software transmits the requested string descriptor to the host (by performing a control read transfer).

(b) SET_DESCRIPTOR

The host issues this request to specify the descriptor of a function device.

If this request is received, the sample software returns a STALL response.

(2) Vendor requests

The sample software responds to the following five types of requests:

- LINE_CONTROL
- SET_DTR_RTS
- SET_XON_XOFF_CHR
- OPEN_CLOSE
- SET_ERR_CHR

For details about each request, see **3.6 Vendor Request Format**.

(3) Undefined requests

If an undefined request is received, the sample software returns a STALL response.

3.1.5 Descriptor settings

The settings of each descriptor specified by the sample software are shown below. These settings are included in the header file `usb78k_desc.h`.

(1) Device descriptor

This descriptor is transmitted in response to a `GET_DESCRIPTOR_device` request.

The settings are stored in the `UF0DDn` registers (where $n = 0$ to 17) when the USB function controller is initialized, because the hardware automatically responds to a `GET_DESCRIPTOR_device` request.

Table 3-2. Device Descriptor Settings

Field	Size (Bytes)	Specified Value	Description
<code>bLength</code>	1	0x12	Descriptor size: 18 bytes
<code>bDescriptorType</code>	1	0x01	Descriptor type: device
<code>bcdUSB</code>	2	0x0200	USB specification release number: USB 2.0
<code>bDeviceClass</code>	1	0xFF	Class code: vendor class
<code>bDeviceSubClass</code>	1	0x00	Subclass code: none
<code>bDeviceProtocol</code>	1	0x00	Protocol code: No unique protocol is used.
<code>bMaxPacketSize0</code>	1	0x40	Maximum packet size of endpoint 0: 64
<code>idVendor</code>	2	0x0409	Vendor ID: NEC
<code>idProduct</code>	2	0x01CD	Product ID: μ PD78F0730
<code>bcdDevice</code>	2	0x0001	Device release number: 1st version
<code>iManufacturer</code>	1	0x01	Index to the string descriptor representing the manufacturer: 1
<code>iProduct</code>	1	0x02	Index to the string descriptor representing the product: 2
<code>iSerialNumber</code>	1	0x03	Index to the string descriptor representing the device production number: 3
<code>bNumConfigurations</code>	1	0x01	Number of configurations: 1

(2) Configuration descriptor

This descriptor is transmitted in response to a GET_DESCRIPTOR_configuration request.

The settings are stored in the UF0CIEn registers (where n = 0 to 255) when the USB function controller is initialized, because the hardware automatically responds to a GET_DESCRIPTOR_configuration request.

Table 3-3. Configuration Descriptor Settings

Field	Size (Bytes)	Specified Value	Description
bLength	1	0x09	Descriptor size: 9 bytes
bDescriptorType	1	0x02	Descriptor type: configuration
wTotalLength	2	0x0020	Total number of bytes of the configuration, interface, and endpoint descriptors: 32 bytes
bNumInterfaces	1	0x01	Number of interfaces in this configuration: 1
bConfigurationValue	1	0x01	Identification number of this configuration: 1
iConfiguration	1	0x00	Index to the string descriptor specifying the source code for this configuration: 0
bmAttributes	1	0x80	Features of this configuration: bus-powered, no remote wakeup
bMaxPower	1	0x32	Maximum current consumed in this configuration: 100 mA

(3) Interface descriptor

This descriptor is transmitted in response to a GET_DESCRIPTOR_configuration request.

The settings are stored in the UF0CIEn registers (where n = 0 to 255) when the USB function controller is initialized, because the hardware automatically responds to a GET_DESCRIPTOR_configuration request.

Table 3-4. Interface Descriptor Settings for Interface 0

Field	Size (Bytes)	Specified Value	Description
bLength	1	0x09	Descriptor size: 9 bytes
bDescriptorType	1	0x04	Descriptor type: interface
bInterfaceNumber	1	0x00	Identification number of this interface: 0
bAlternateSetting	1	0x00	Whether the alternative settings are specified for this interface: no
bNumEndpoints	1	0x02	Number of endpoints of this interface: 2
bInterfaceClass	1	0xFF	Class code: vendor class
bInterfaceSubClass	1	0x00	Subclass code: none
bInterfaceProtocol	1	0x00	Protocol code: No unique protocol is used.
iInterface	1	0x00	Index to the string descriptor specifying the source code for this interface: 0

(4) Endpoint descriptor

This descriptor is transmitted in response to a GET_DESCRIPTOR_configuration request.

The settings are stored in the UF0CIEn registers (where n = 0 to 255) when the USB function controller is initialized, because the hardware automatically responds to a GET_DESCRIPTOR_configuration request.

Two descriptor types are specified because the sample software uses two endpoints.

Table 3-5. Endpoint Descriptor Settings for Endpoint 2

Field	Size (Bytes)	Specified Value	Description
bLength	1	0x07	Descriptor size: 7 bytes
bDescriptorType	1	0x05	Descriptor type: endpoint
bEndpointAddress	1	0x02	Transfer direction of this endpoint: OUT Address of this endpoint: 2
bmAttributes	1	0x02	Transfer type of this endpoint: bulk
wMaxPacketSize	2	0x0040	Maximum packet size of this transfer: 64 bytes
bInterval	1	0x00	Polling interval of this endpoint: 0 ms

Table 3-6. Endpoint Descriptor Settings for Endpoint 1

Field	Size (Bytes)	Specified Value	Description
bLength	1	0x07	Descriptor size: 7 bytes
bDescriptorType	1	0x05	Descriptor type: endpoint
bEndpointAddress	1	0x81	Transfer direction of this endpoint: IN Address of this endpoint: 1
bmAttributes	1	0x02	Transfer type of this endpoint: bulk
wMaxPacketSize	2	0x0040	Maximum packet size of this transfer: 64 bytes
bInterval	1	0x00	Polling interval of this endpoint: 0 ms

(5) String descriptor

This descriptor is transmitted in response to a GET_DESCRIPTOR_string request.

If a GET_DESCRIPTOR_string request is received, the sample software extracts the settings of this descriptor from the header file usb78k_desc.h and stores them into the UF0E0W register of the USB function controller.

Table 3-7. String Descriptor Settings**(a) String 0**

Field	Size (Bytes)	Specified Value	Description
bLength	1	0x04	Descriptor size: 4 bytes
bDescriptorType	1	0x03	Descriptor type: string
bString	2	0x09, 0x04	Language code: English (U.S.)

(b) String 1

Field	Size (Bytes)	Specified Value	Description
bLength ^{Note 1}	1	0x2A	Descriptor size: 42 bytes
bDescriptorType	1	0x03	Descriptor type: string
bString ^{Note 2}	40	–	Vendor: NEC Electronics Corporation

Notes 1. The specified value depends on the size of the bString field.

2. The vendor can freely set up the size and specified value of this field.

(c) String 2

Field	Size (Bytes)	Specified Value	Description
bLength ^{Note 1}	1	0x16	Descriptor size: 22 bytes
bDescriptorType	1	0x03	Descriptor type: string
bString ^{Note 2}	12	–	Product type: VirtualCom (virtual COM driver)

Notes 1. The specified value depends on the size of the bString field.

2. The vendor can freely set up the size and specified value of this field.

(d) String 3

Field	Size (Bytes)	Specified Value	Description
bLength ^{Note 1}	1	0x16	Descriptor size: 22 bytes
bDescriptorType	1	0x03	Descriptor type: string
bString ^{Note 2}	20	–	Serial number: 0_98765432

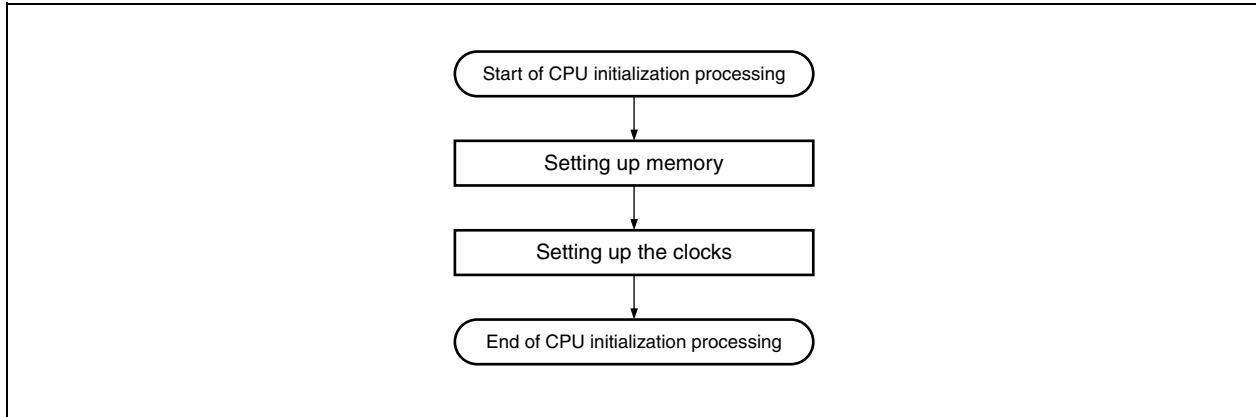
Notes 1. The specified value depends on the size of the bString field.

2. The vendor can freely set up the size and specified value of this field.

3.2 CPU Initialization Processing

The settings necessary to use the μ PD78F0730 are specified.

Figure 3-4. CPU Initialization Processing Flow



(1) Setting up memory

The sizes of the ROM and internal expansion RAM are specified.

- <1> 0xC4 is written to the IMS register. This sets the ROM size to 16 KB.
- <2> 0x08 is written to the IXS register. This sets the RAM size to 3 KB (high speed: 1 KB, expansion: 2 KB).

(2) Setting up the clocks

The clock signal to supply to the high-speed system clock, CPU clock, and PLL is specified.

- <1> 0x41 is written to the OSCCTL register.
- <2> 0x00 is written to the MOC register.
- <3> 0x01 is written to the OSTS register.
- <4> 0x00 is written to the PCC register.
- <5> 0x00 is written to the RCM register.
- <6> 0x01 is written to the PLLC register.
- <7> 0x05 is written to the MCM register.
- <8> 0x03 is written to the PLLC register.
- <9> 0x02 is written to the PLLC register.

If these settings are specified, the μ PD78F0730 operates based on the internal high-speed oscillation clock ($f_{RH} = 16$ MHz) and the USB operation clock (f_{USB}) is set to 48 MHz.

3.3 USB Control Processing

This processing initializes the USB function controller (USBF), executes an interrupt handler, and performs data transmission and reception.

The sample software provides a simple driver to use the USBF incorporated in the μ PD78F0730.

Caution A processing program for the USBF incorporated in the μ PD78F0730 is provided, but it only performs the minimum processing for the sample software. The use of the simple driver as a general-purpose USB driver is not guaranteed.

- USBF initialization

This processing is called by the main routine and initializes the USBF incorporated in the μ PD78F0730.

- USBF interrupt handler

This routine is dedicated to interrupt servicing and is called every time a USBF interrupt occurs.

Caution For this sample software, unnecessary interrupts are masked.

The following interrupts are used for this sample software:

- The RSUSPD, BUSRST, SETRQ, and CPUDEC interrupts reported by INTUSB0B
- The BKO1DT interrupt reported by INTUSB1B

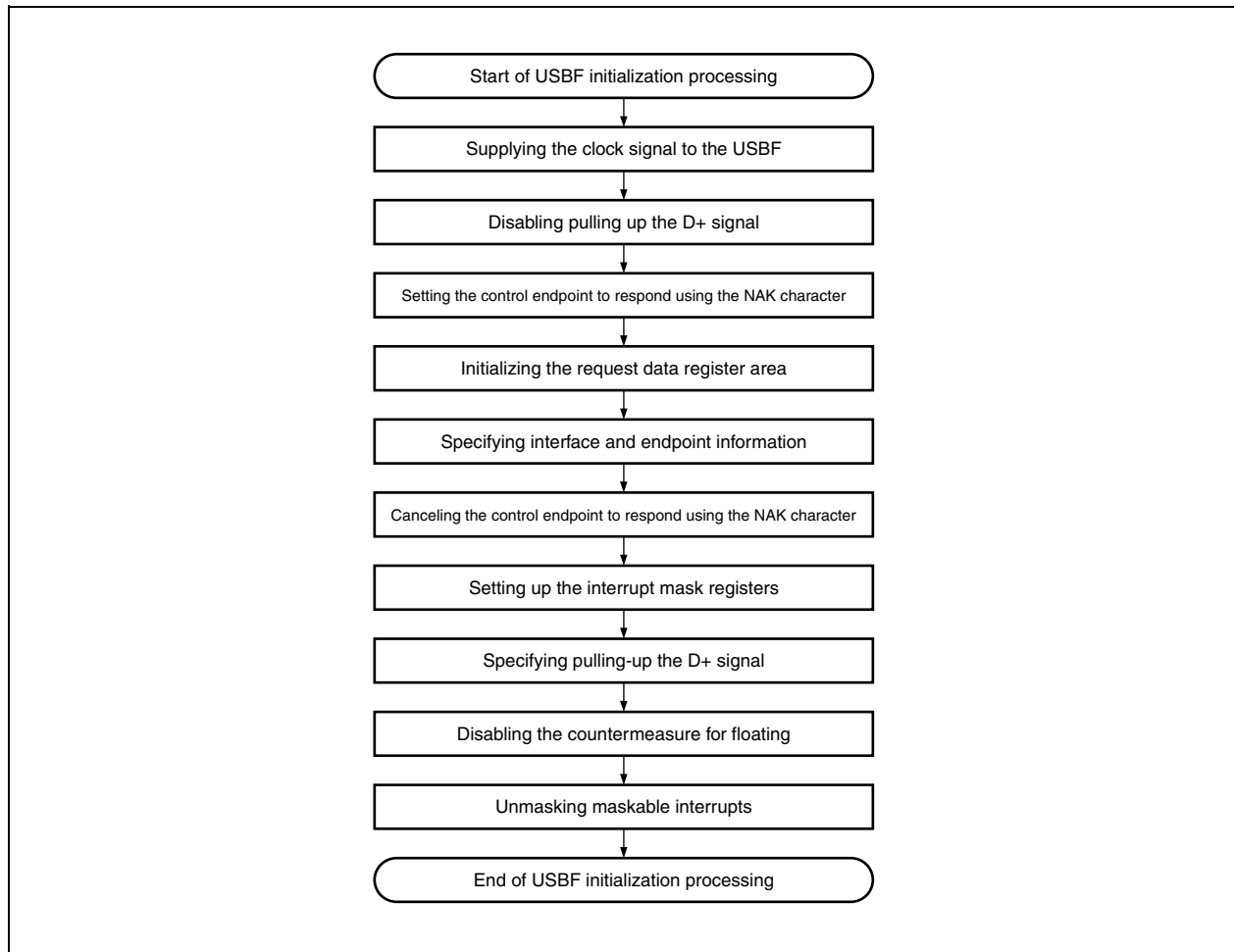
- General-purpose USB functions

USB data reception and transmission functions are provided as general-purpose functions for USB control processing.

3.3.1 USBF initialization processing

The settings necessary to use the USB function controller (USBF) are specified.

Figure 3-5. USBF Initialization Processing Flow



(1) Supplying the clock signal to the USBF

Supplying the clock signal to the USB function controller (USBF) is enabled.

- 1 is written to the UCKCNT bit of the UCKC register.

(2) Disabling pulling up the D+ signal

Reporting connection to the USB host or hub is disabled by disabling pulling up the D+ signal.

- 0 is written to the CONNECT bit of the UF0GPR register.

(3) Setting the control endpoint to respond using the NAK character

The control endpoint is set up so that it responds to any request by the NAK character, including automatically executed requests.

- 1 is written to the EP0NKA bit of the UF0E0NA register.

This setting prevents the hardware from returning unintended data to requests that are automatically responded to until the data to be used by these requests is added.

(4) Initializing the request data register area

The descriptor data transmitted in response to a GET_DESCRIPTOR request is added to registers.

The data to add includes the device status, endpoint 0 status, device descriptor, configuration descriptor, interface descriptor, and endpoint descriptor.

<1> 0x00 is written to the UF0DSTL register.

This setting disables remote wakeup and operates the USB function controller as a bus-powered device.

<2> 0x00 is written to the UF0EnSL registers (where n = 0 to 2).

This setting indicates that endpoint n operates normally.

<3> The total data length (number of bytes) of the required descriptor is written to the UF0DSCL register.

This setting determines the range of the UF0CIEn registers (where n = 0 to 255).

<4> The device descriptor data is written to the UF0DDn registers (where n = 0 to 17).

<5> The data of the configuration, interface, and endpoint descriptors is written to the UF0CIEn registers (where n = 0 to 255).

<6> 0x00 is written to the UF0MODC register.

This setting enables automatic responses to GET_DESCRIPTOR_configuration requests.

(5) Specifying interface and endpoint information

Information such as the number of supported interfaces, whether the alternative setting is used, and the relationship between the interfaces and endpoints is specified for registers.

<1> 0x00 is written to the UF0AIFN register.

This setting enables only interface 0.

<2> 0x00 is written to the F0AAS register.

This setting disables the alternative setting.

<3> 0x20 is written to the UF0E1IM and UF0E2IM registers.

This setting links endpoints 1 and 2 to interface 0.

(6) Canceling the control endpoint to respond using the NAK character

The setting specifying that the control endpoint responds by using the NAK character is canceled when the data for automatically executed requests has been added.

- 0 is written to the EP0NKA bit of the UF0E0NA register.

This setting resumes responding to all requests, including requests that are automatically responded to.

(7) Setting up the interrupt mask registers

Masking is specified for each interrupt source indicated by the interrupt status register of the USB function controller.

<1> 1 is written to all valid bits of the UF0ICn registers (where n = 0 to 4).

This setting clears all interrupt sources.

<2> 1 is written to all valid bits of the UF0FIC0 and UF0FIC1 registers.

This setting clears all transfer FIFOs.

<3> 0x07 is written to the UF0IM0 register.

This setting masks interrupt sources indicated by the UF0IS0 register other than those of the RSUSPDM and BUSRSTM interrupts.

<4> 0x7E is written to the UF0IM1 register.

This setting masks interrupt sources indicated by the UF0IS1 register other than those of the CPUDEC interrupt.

<5> 0x30 is written to the UF0IM2 register.

This setting masks all interrupt sources indicated by the UF0IS2 register.

<6> 0x0E is written to the UF0IM3 register.

This setting masks interrupt sources indicated by the UF0IS3 register other than those of the BKO1DT interrupt.

<7> 0x20 is written to the UF0IM4 register.

This setting masks all interrupt sources indicated by the UF0IS4 register.

(8) Specifying pulling-up the D+ signal

The D+ signal is pulled up so that the host recognizes that a device has been connected.

- 1 is written to the CONNECT bit of the UF0GPR register.

(9) Disabling the countermeasure for floating

The countermeasure for floating is disabled.

- 0x03 is written to the UF0BC register.

This setting disables the countermeasure for floating and enables the USB buffer.

(10) Unmasking maskable interrupts

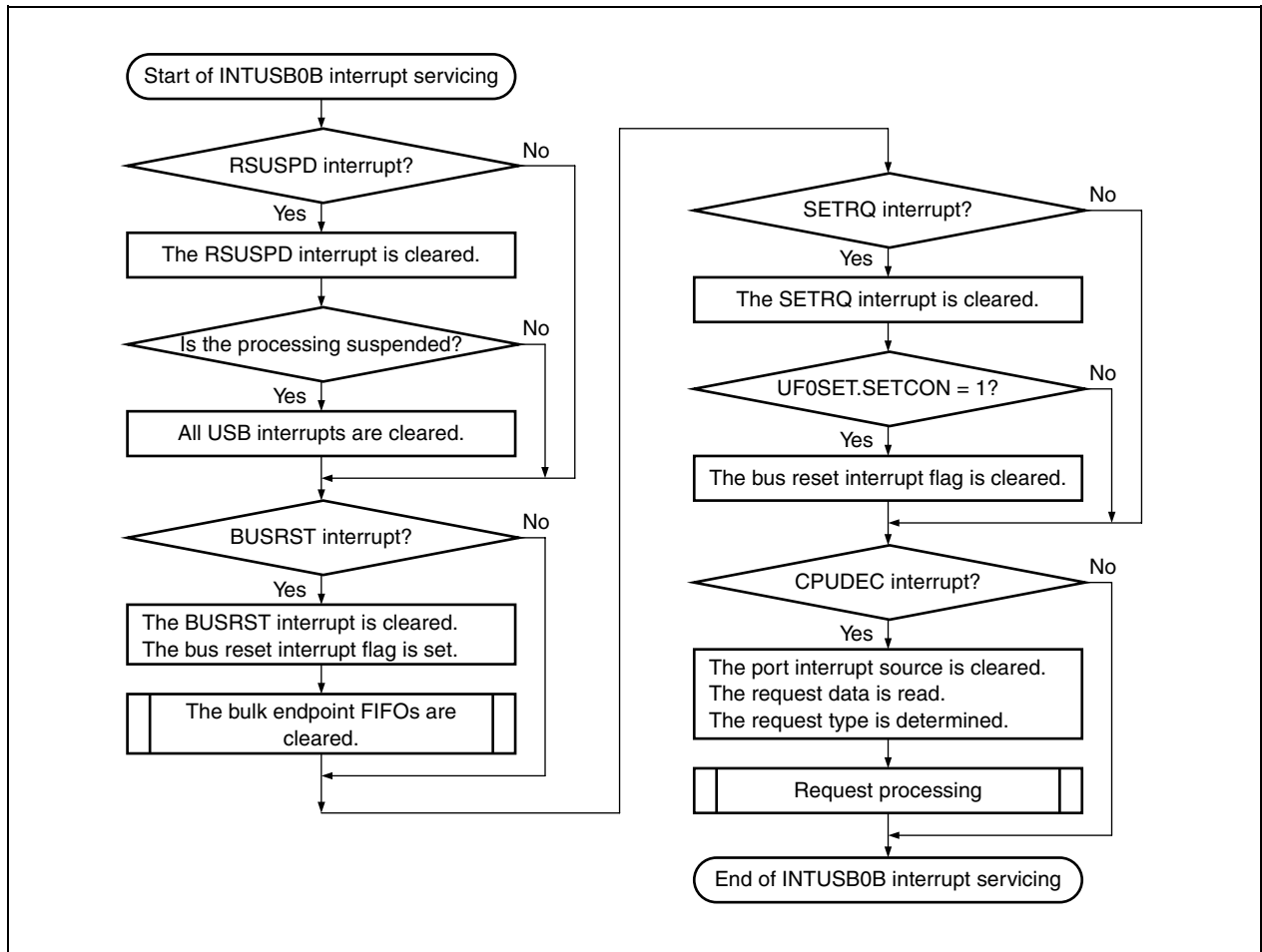
The interrupt sources INTUSB0 and INTUSB1 are unmasked.

- 0 is written to the USBMK0 and USBMK1 bits of the interrupt mask flag MK0L.

3.3.2 USBF interrupt servicing (INTUSB0B)

The INTUSB0B interrupt handler services the RSUSPD, BUSRST, SETRQ, and CPUDEC interrupts.

Figure 3-6. INTUSB0B Interrupt Handler Processing Flow



(1) RSUSPD interrupt servicing

If the RSUSPD bit of the UF0IS0 register is 1, an RSUSPD interrupt is judged to have occurred.

If an RSUSPD interrupt occurred, the following processing is performed:

- The interrupt source is cleared. (0 is written to the RSUSPDC bit of the UF0IC0 register.)
- Whether the processing is suspended or has resumed is determined.

(2) Suspend processing

If the RSUM bit of the UF0EPS1 register is 1, the processing is judged to have been suspended.

If the processing is suspended, all USB interrupt sources are cleared. This omits all subsequent INTUSB0B interrupt servicing.

(3) BUSRST interrupt servicing

If the BUSRST bit of the UF0IS0 register is 1, a BUSRST interrupt is judged to have occurred.

If a BUSRST interrupt occurred, the following processing is performed:

- The interrupt source is cleared. (0 is written to the BUSRST bit of the UF0IC0 register.)
- The bus reset interrupt flag (usb78k_busrst_flg) is set to 1.
- The bulk endpoint FIFOs are cleared.

(4) Clearing the bulk endpoint FIFOs

The FIFO clearing function (usb78k_clearFIFO) is called to clear all bulk endpoint FIFOs.

(5) SETRQ interrupt servicing

If the SETRQ bit of the UF0IS0 register is 1, an SETRQ interrupt is judged to have occurred.

If a SETRQ interrupt occurred, the following processing is performed:

- The interrupt source is cleared. (0 is written to the SETRQ bit of the UF0IC0 register.)
- A request that is automatically responded to (SET_XXXX) is processed.

(6) Processing an automatically responded request (SET_XXXX)

If the SETCON bit of the UF0SET register is 1, a SET_CONFIGURATION request is received and automatic processing is judged to have been performed.

If automatic processing was performed, the bus reset interrupt flag (usb78k_busrst_flg) is set to 0.

Caution To check whether a configured status has been entered, check the values of the UF0CNF register.

(7) CPUDEC interrupt servicing

If the CPUDEC bit of the UF0IS1 register is 1, a CPUDEC interrupt is judged to have occurred.

If a CPUDEC interrupt occurred, the following processing is performed:

- The port interrupt source is cleared. (0 is written to the PORT bit of the UF0IC1 register.)
- The received data is read from the FIFOs and request data is created.
- Request processing

(8) Request processing

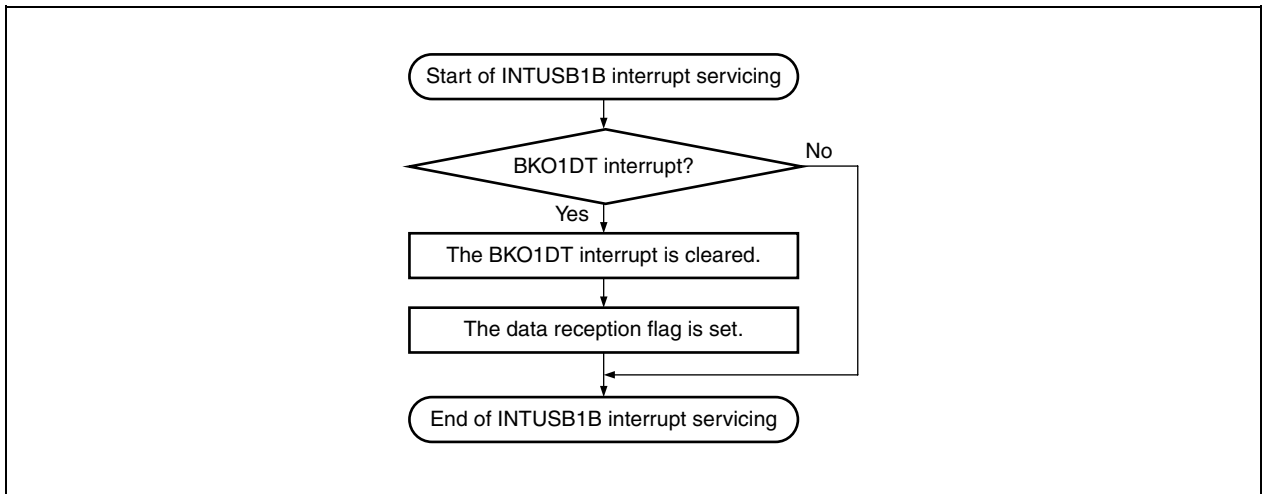
Whether the request is one to which the hardware does not automatically respond (a standard, class, or vendor request) is determined and processing according to the type of request is executed.

Endpoint 0 is used for a control transfer. During the enumeration processing when a device is plugged in, almost all standard device requests are automatically processed by the hardware. Here, the standard, class, and vendor requests that are not automatically processed are processed.

3.3.3 USBF reception interrupt servicing (INTUSB1B)

The INTUSB1B interrupt handler services the BKO1DT interrupt.

Figure 3-7. INTUSB1B Interrupt Handler Processing Flow



(1) Judging the BKO1DT interrupt

If the BKO1DT bit of the UF0IS3 register is 1, a BKO1DT interrupt is judged to have occurred.

(2) Clearing the BKO1DT interrupt

The interrupt source is cleared by writing 0 to the BKO1DTC bit of the UF0IC3 register.

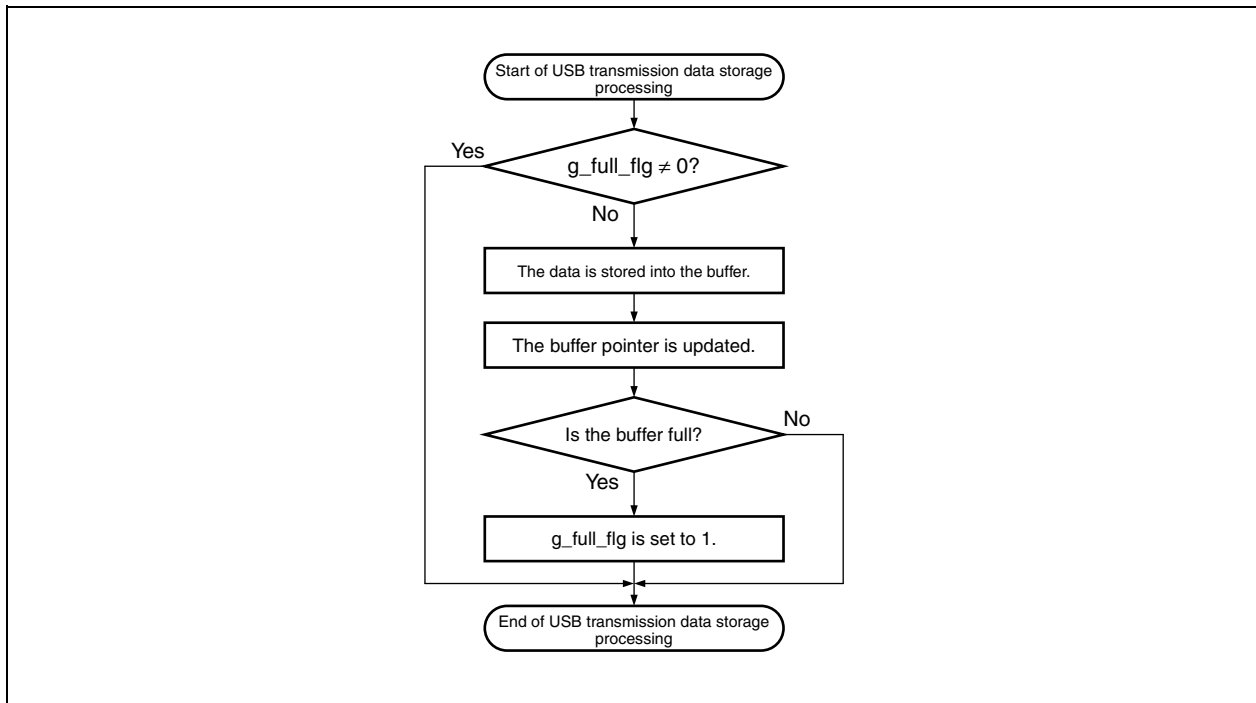
(3) Setting the data reception flag

The data reception flag (usb78k_rdata_flg) is set to 1.

3.3.4 USB transmission data storage processing

The data to transmit to the USB device is stored into the transmission ring buffer.

Figure 3-8. USB Transmission Data Storage Processing Flow



(1) If the USB transmission data storage buffer has a vacancy (g_full_flg = 0)

The data is stored into the USB transmission data storage buffer and the buffer pointer is updated.

If the buffer is full after the data is stored, the buffer full flag (g_full_flg) is set to 1.

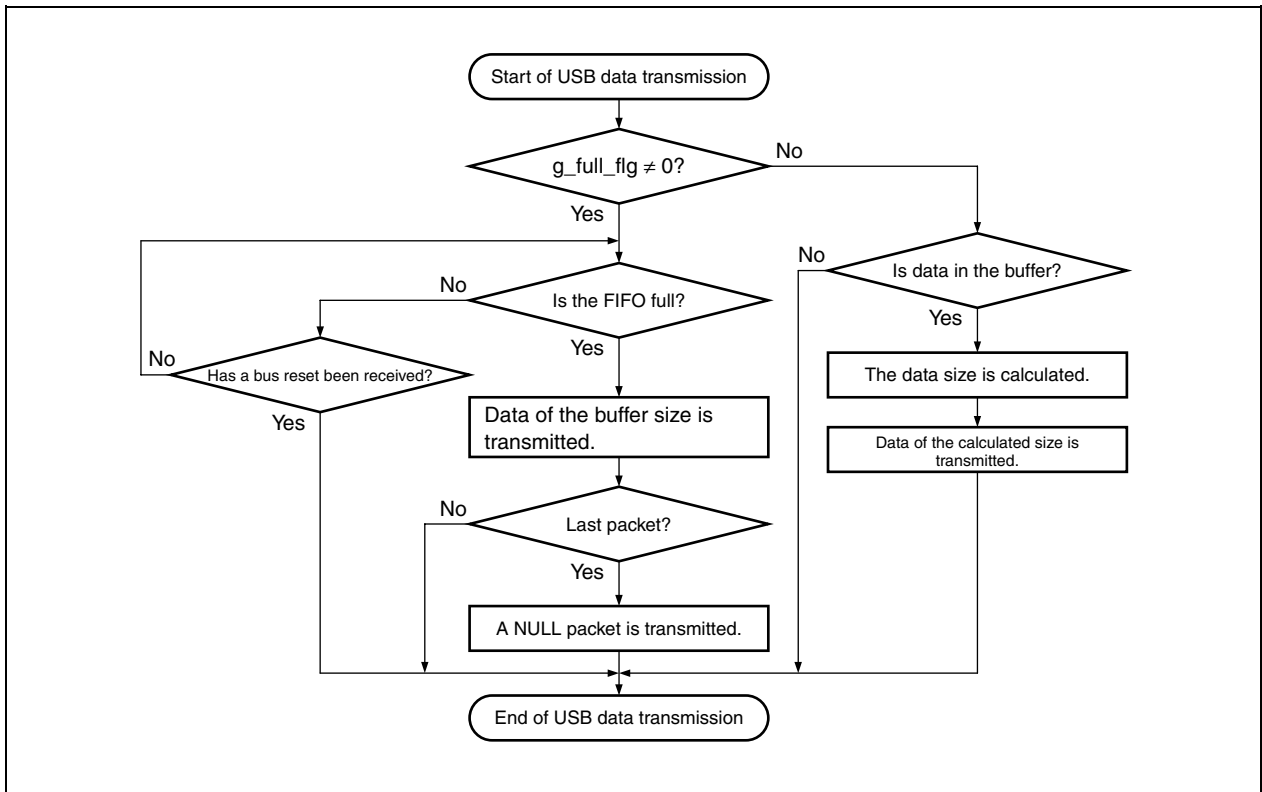
(2) If the USB transmission data storage buffer has no vacancy (g_full_flg ≠ 0)

USB transmission data storage processing ends without storing the data.

3.3.5 USB data transmission

The data stored in the USB transmission ring buffer is transmitted to the USB host.

Figure 3-9. USB Data Transmission Flowchart



(1) If the transmission ring buffer has a vacancy ($g_full_flag = 0$)

The size of the data stored in the transmission ring buffer is calculated and data of that size is transmitted.

(2) If the transmission ring buffer has no vacancy ($g_full_flag \neq 0$)

All the data stored in the transmission ring buffer is transmitted.

If the transmitted data is the last packet, a NULL packet is transmitted after the data is transmitted.

(3) Bus reset

Whether a bus reset has been received is judged only if the transmission ring buffer has no vacancy and the transmission FIFO is full.

If the bus reset interrupt flag (`usb78k_busrst_flg`) is 1, a bus reset is judged to have been received.

If a bus reset was received, USB data transmission ends without transmitting the data.

3.4 UART Control Processing

This processing initializes the serial interface (UART6), executes an interrupt handler, and performs data transmission and reception.

The sample software provides a simple driver to use the UART6 incorporated in the μ PD78F0730.

Caution A processing program for the UART incorporated in the μ PD78F0730 is provided, but it only performs the minimum processing for the sample software. The use of the simple driver as a general-purpose UART driver is not guaranteed.

- UART initialization

This processing is called by the main routine and initializes the UART6 incorporated in the μ PD78F0730.

- UART interrupt handler

This routine is dedicated to interrupt servicing and is called every time a UART interrupt occurs.

Caution For this sample software, unnecessary interrupts are masked.

The following two interrupts are used for this sample software:

- The reception completion interrupt reported by INTSR6
- The reception error interrupt reported by INTSRE6

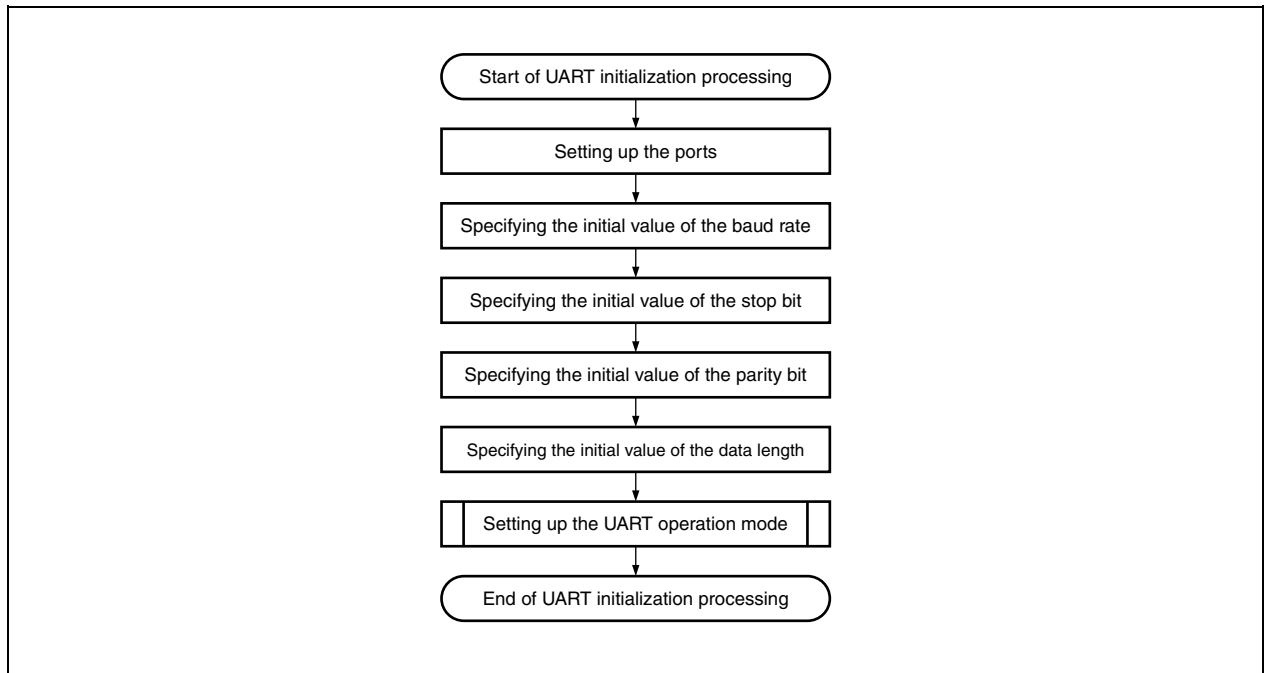
- General-purpose UART functions

The UART data transmission functions and operation mode specification functions are provided as general-purpose functions for the UART processing block.

3.4.1 UART initialization processing

This processing specifies the initial values of the μ PD78F0730 ports and the UART6 operation mode.

Figure 3-10. UART Initialization Processing Flow



(1) Setting up the ports

The serial data I/O ports of the serial interface UART6 are set up.

(2) Specifying the initial value of the baud rate

The initial baud rate value is stored into the UART communication setting structure (UART_MODE_TBL).

(3) Specifying the initial value of the stop bit

The initial stop bit value is stored into the UART communication setting structure (UART_MODE_TBL).

(4) Specifying the initial value of the parity bit

The initial parity bit value is stored into the UART communication setting structure (UART_MODE_TBL).

(5) Specifying the initial value of the data length

The initial data length value is stored into the UART communication setting structure (UART_MODE_TBL).

(6) Setting up the UART operation mode

The UART operation mode specification function (`uart78k_uartmode_set`) is called to set up the UART6 registers.

3.4.2 UART operation mode specification processing

This processing specifies the UART operation mode according to the value stored in the UART communication setting structure (UART_MODE_TBL). The transfer speed, parity bit, data length, and stop bit are specified.

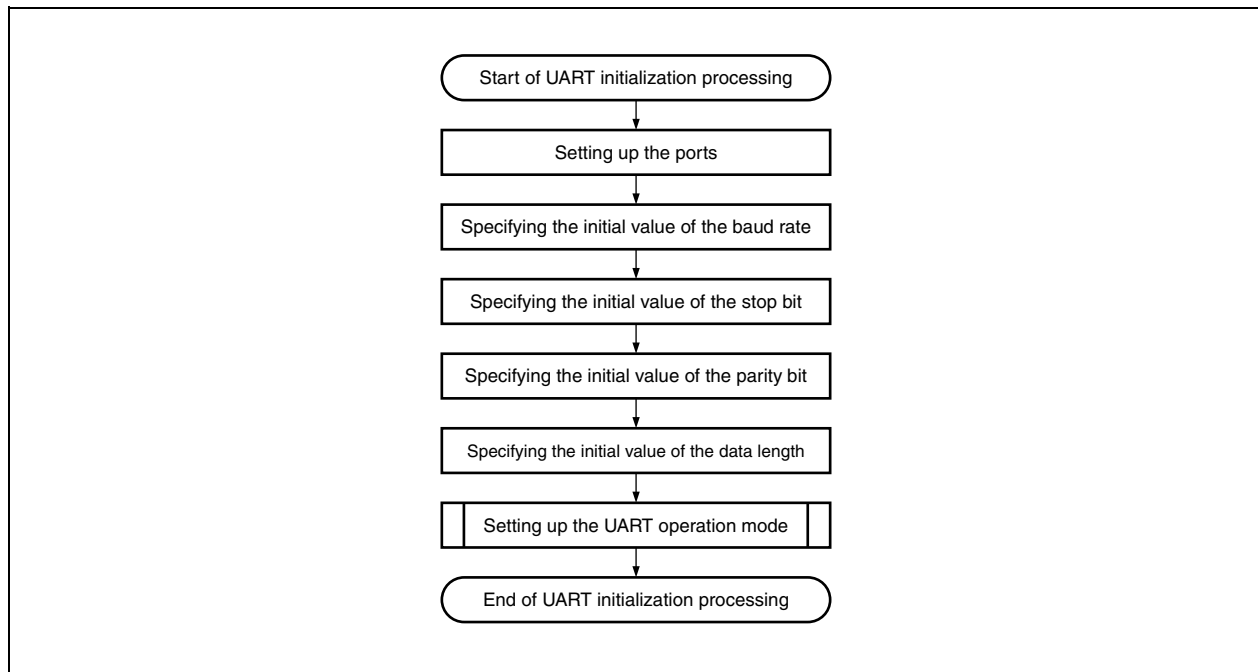
This processing is called during UART initialization processing or LINE_CONTROL request processing.

During UART initialization processing, the initially specified values are stored into the UART communication setting structure (UART_MODE_TBL), and then the UART operation mode specification processing is called.

During LINE_CONTROL request processing, the value of the operation mode to be specified by the host is stored into the UART communication setting structure (UART_MODE_TBL) and the UART operation mode is changed by calling the UART operation mode specification processing.

Remark For details about the UART communication setting structure (UART_MODE_TBL) and the UART specification values that can be used for the sample software, see **3.4.6 UART operation mode**.
For details about the LINE_CONTROL request, see **3.6.1 LINE_CONTROL**.

Figure 3-11. UART Operation Mode Specification Processing Flow



(1) Disabling transmission and reception

The POWER6, TXE6, and RXE6 bits of asynchronous serial interface operation mode register 6 (ASIM6) are set to 0 to disable UART6 transmission and reception.

(2) Specifying the baud rate

The value specified for the baud rate is read from the UART communication setting structure, and then clock selection register 6 (CKSR6) and baud rate generator control register 6 (BRGC6) are set up.

(3) Specifying the stop bit

The value specified for the stop bit is read from the UART communication setting structure and the SL6 bit of asynchronous serial interface operation mode register 6 (ASIM6) is set up.

(4) Specifying the parity bit

The value specified for the parity bit is read from the UART communication setting structure and the PS60 and PS61 bits of asynchronous serial interface operation mode register 6 (ASIM6) are set up.

(5) Specifying the data length

The value specified for the data length is read from the UART communication setting structure and the CL6 bit of asynchronous serial interface operation mode register 6 (ASIM6) is set up.

(6) Setting up the reception error interrupt signal

The generation of a reception completion interrupt is enabled for when an error occurs by setting the ISRM6 bit of asynchronous serial interface operation mode register 6 (ASIM6) to 0.

(7) Enabling transmission and reception

The POWER6, TXE6, and RXE6 bits of asynchronous serial interface operation mode register 6 (ASIM6) are set to 1 to enable UART6 transmission and reception.

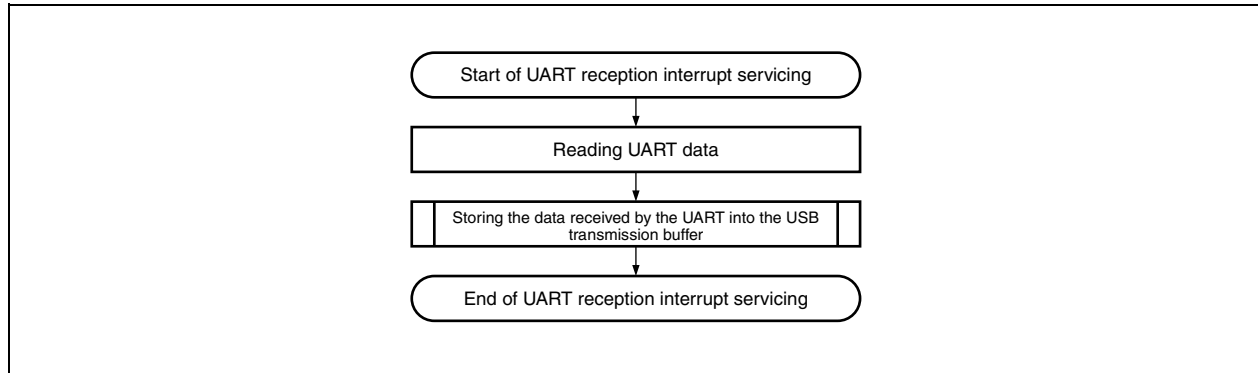
(8) Enabling interrupts

The SRIF6 and SREIF6 interrupts are enabled by clearing the interrupt request flag and canceling masking.

3.4.3 UART reception interrupt servicing

A vector is added so that UART data reception processing starts when a UART reception completion interrupt occurs.

Figure 3-12. UART Reception Interrupt Servicing Flowchart



(1) Reading UART data

Data is read from the UART reception buffer register. This enables the reception of new data.

(2) Storing the data received by the UART into the USB transmission buffer

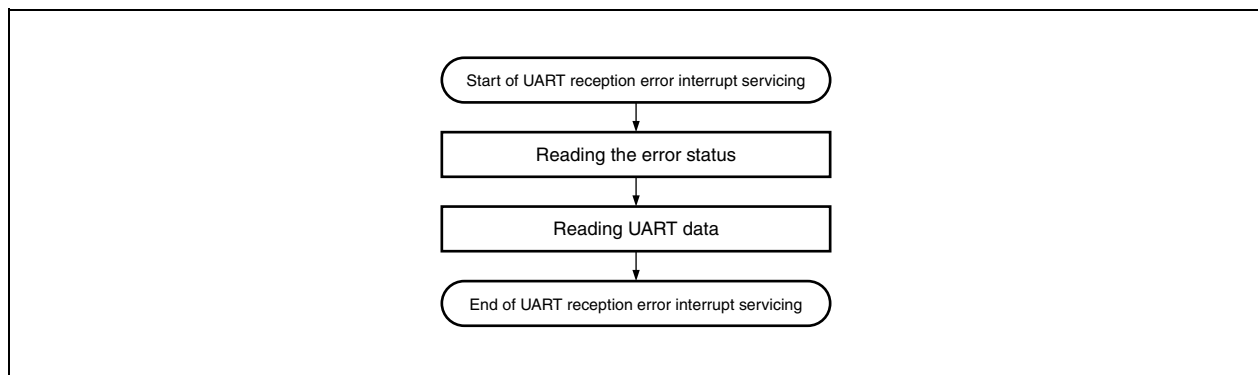
The read data is stored into the USB transmission ring buffer.

For details, see **3.5.1 Storing the data received by the UART into the USB transmission buffer**.

3.4.4 UART reception error interrupt servicing

If an error occurs during UART reception, the received data that caused the error is discarded.

Figure 3-13. UART Reception Error Interrupt Servicing Flowchart



(1) Reading the error status

The reception error status register is read. This clears the error flag.

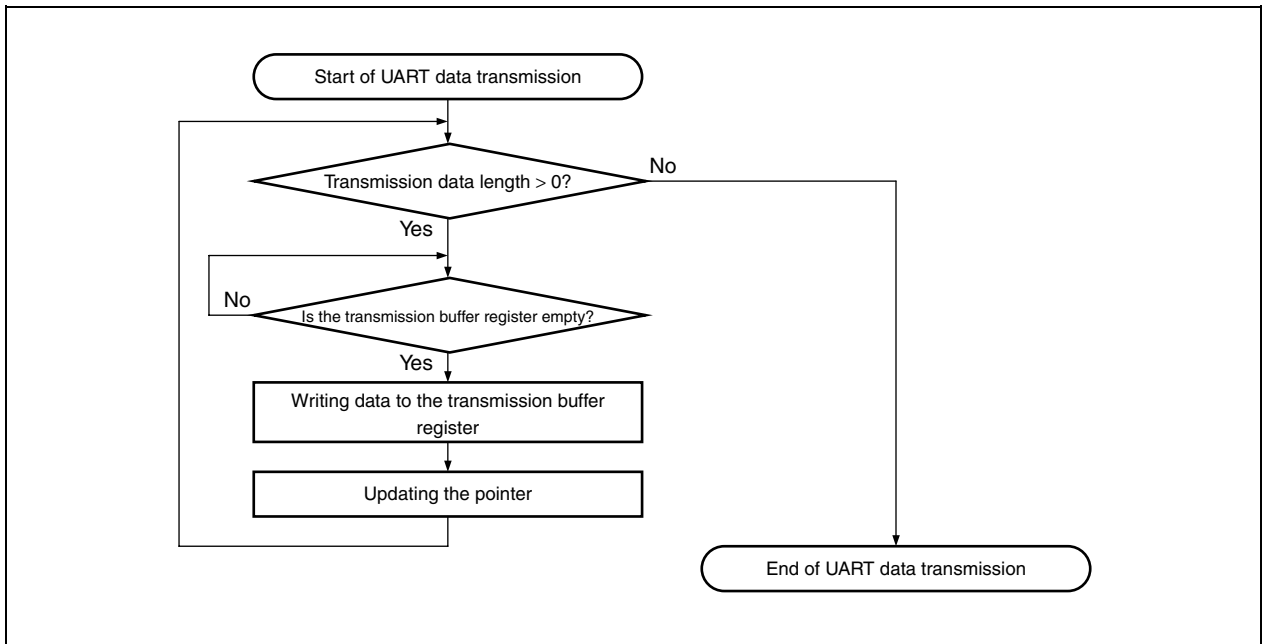
(2) Reading UART data (reading and then discarding)

Data is read from the UART reception buffer register. This enables the reception of new data.

3.4.5 UART data transmission

Data of the size specified by the UART is transmitted.

Figure 3-14. UART Data Transmission Flowchart



(1) Checking the transmission data length

Data is transmitted until there is none left.

(2) Checking the transmission buffer register

The system waits until there is a vacancy in the UART transmission buffer register.

(3) Transmitting data

Transmission data is written to the UART transmission buffer register in 1-byte units.

(4) Updating the pointer

The transmission data pointer and the transmission data length are updated.

3.4.6 UART operation mode

The values that can be specified for the UART operation mode settings of the sample software are limited. Table 3-8 shows the values that can be specified for the sample software.

Table 3-8. UART Specification Values

Setting	Value	Remark
Transfer speed	2400 bps	—
	4800 bps	—
	9600 bps	Initial value
	19200 bps	—
	38400 bps	—
	57600 bps	—
	76800 bps	—
	115200 bps	—
	Other than the above	9,600 bps is specified.
Parity bit	None	Initial value
	Odd numbers	—
	Even numbers	—
	(Space)	There is no corresponding feature on the host driver side.
	Other than the above	No parity is specified.
Data length	7 bits	—
	8 bits	Initial value
	Other than the above	8 bits are specified.
Stop bit	1 bit	Initial value
	2 bits	—
	Other than the above	1 bit is specified.

The UART operation mode settings are retained in the UART communication setting structure (UART_MODE_TBL). The UART communication setting structure (UART_MODE_TBL) is defined as follows:

List 3-1. UART Communication Setting Structure (UART_MODE_TBL)

```
typedef struct UART_MODE_TBL{
    UINT8      DTERate[4];          /* transfer rate(bps) */
    UINT8      STOPBIT;             /* length of the stop bit - 0:1bit 2:2bits */
    UINT8      PARITYType;          /* parity bit - 0:None 1:Odd 2:Even 4:Space */
    UINT8      DATABits;            /* data size (number of the bits:7 or 8) */
} UART_MODE_TBL , *PUART_MODE_TBL;
```

3.5 Bridge Processing Between the UART and USB

Data is transferred between the UART and USB.

3.5.1 Storing the data received by the UART into the USB transmission buffer

This processing is called by UART reception completion interrupt servicing.

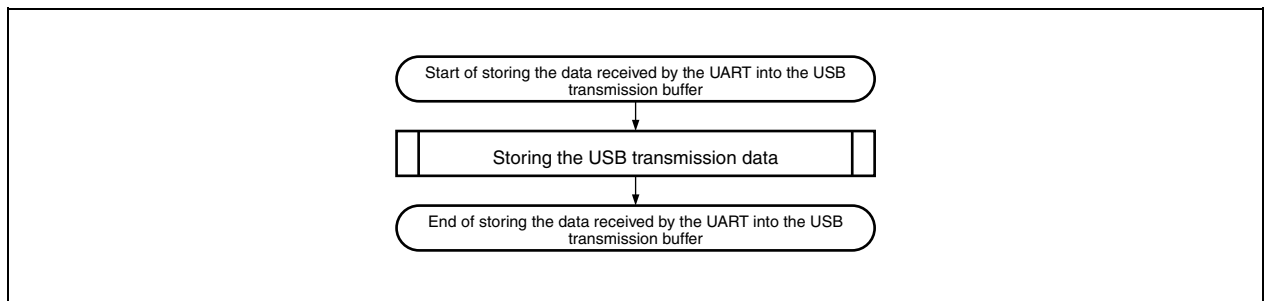
During this processing, the data received by the UART is stored into the USB transmission ring buffer.

For details about UART reception completion interrupt servicing, see **3.4.3 UART reception interrupt servicing**.

For details about the transmission of data in the transmission ring buffer, see **3.3.5 USB data transmission**.

For details about USB transmission data storage processing, see **3.3.4 USB transmission data storage processing**.

Figure 3-15. Flowchart of Storing the Data Received by the UART into the USB Transmission Buffer



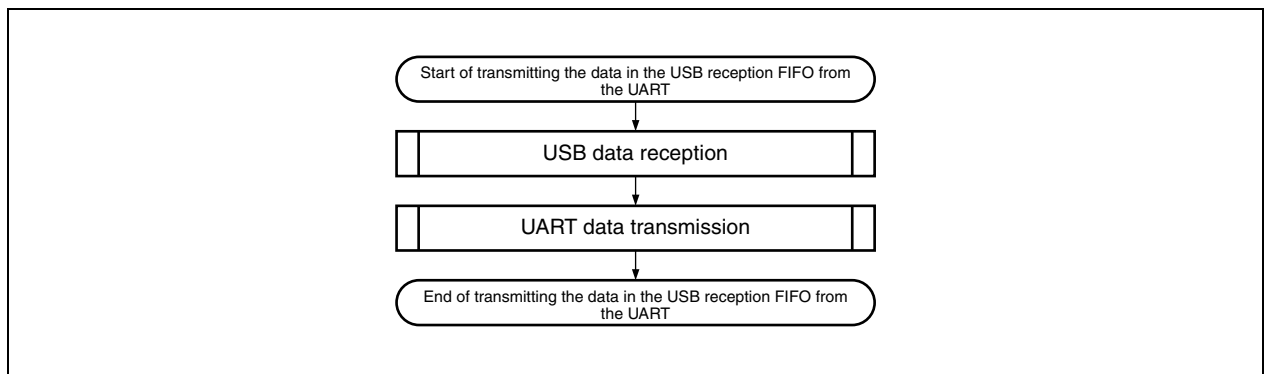
3.5.2 Transmitting the data in the USB reception FIFO from the UART

During this processing, the data at the endpoint (FIFO) for a bulk out transfer (reception) is read and transmitted from the UART.

During USB data reception, the data in the FIFO is read and stored into the reception data storage buffer.

During UART data transmission, the data in the reception data storage buffer is transmitted from the UART byte by byte. For details, see **3.4.5 UART data transmission**.

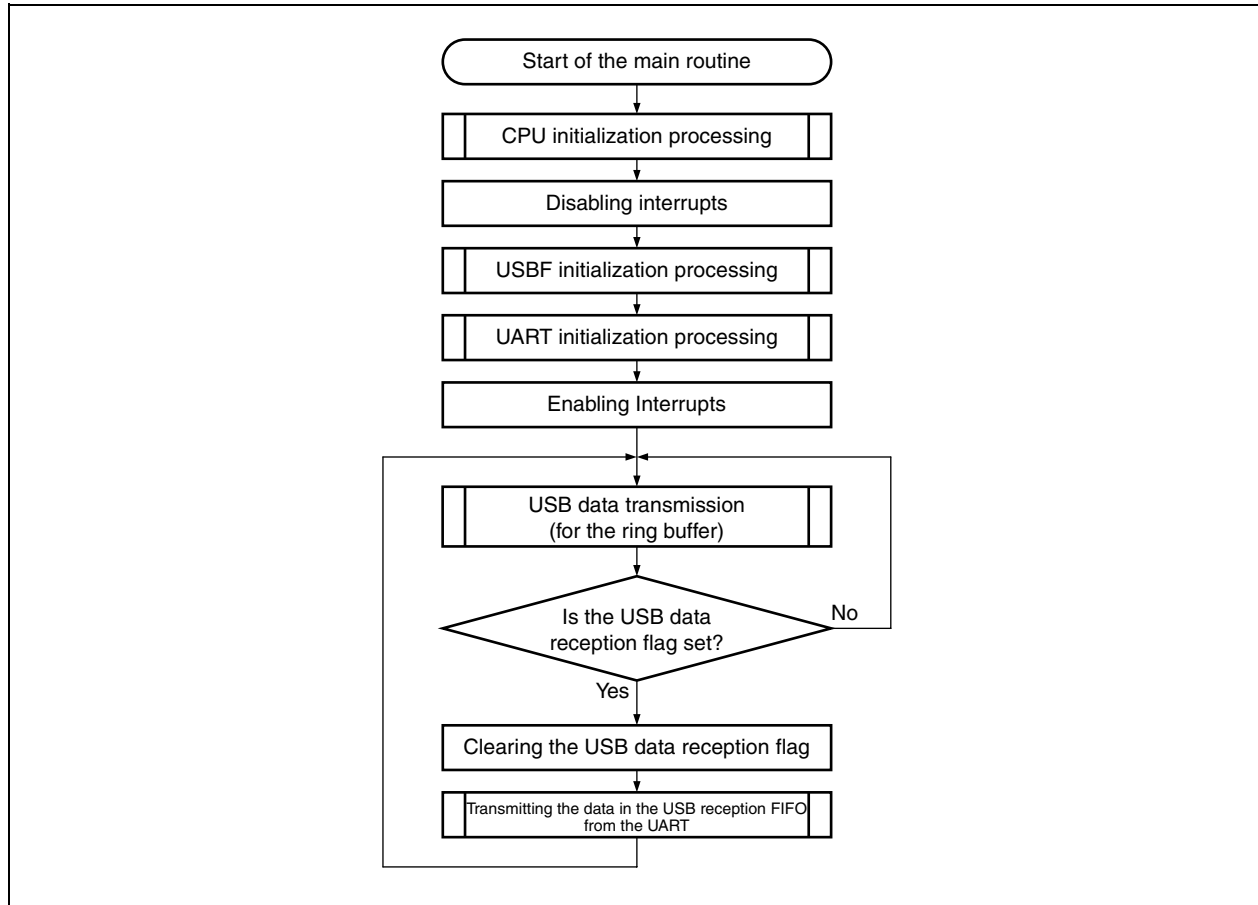
Figure 3-16. Flowchart of Transmitting the Data in the USB Reception FIFO from the UART



3.5.3 Main routine

During the main routine, the data received by the USB function controller (USBF) is transmitted to the serial interface (UART6) and the data received by the serial interface is transmitted to the USB function controller.

Figure 3-17. Main Routine Processing Flow



(1) USB data transmission

The function for transmitting USB data to the ring buffer (`usb78k_send_txbuf`) is called to transmit the data stored in the USB transmission ring buffer to the USB host.

(2) Identification of USB data reception

The data reception flag set by the sample software (`usb78k_rdata_flg`) is monitored. If this flag is set to 1, there is reception data in the USB function controller.

If there is reception data, the data reception flag (`usb78k_rdata_flg`) is cleared (0) and the data in the USB reception FIFO is transmitted from the UART.

(3) Transmitting the data in the USB reception FIFO from the UART

The function for transmitting the data in the USB reception FIFO from the UART (`usb78k_usb_to_uart`) is called to transmit the data in the USB reception FIFO from the UART.

3.6 Vendor Request Format

The sample software can perform processing to respond to the five types of requests below.

This section describes the format of each vendor request.

3.6.1 LINE_CONTROL

This request is used by the host to report the settings of the baud rate, flow control, parity bit, and data size to the device.

Table 3-9. LINE_CONTROL Request Format

(a) Request codes

Field	Size	Specified Value
bmRequestType	1	Request type: 0x40
bRequest	1	Request identifier: 0x00
wValue	2	Unused: 0x0000
wIndex	2	Unused: 0x0000
wLength	2	Data length: 0x0006 (the number of data stage bytes)

(b) Data

Field	Size	Specified Value
bRequest	1	Request identifier: 0x00 (LINE_CONTROL)
bBaud	4	Baud rate ^{Note} : 0x00000960 (2400 bps) 0x000012c0 (4800 bps) 0x00002580 (9600 bps) 0x00004b00 (19200 bps) 0x00009600 (38400 bps) 0x0000e100 (57600 bps) 0x00012c00 (76800 bps) 0x0001c200 (115200 bps)
bParams	1	D7 and D6 (reserved): 00 D5 and D4 (flow control): 00 (none) 01 (hardware (RTS or CTS)) 10 (software (Xon or Xoff)) D3 and D2 (parity): 00 (none) 01 (even number) 10 (odd number) D1 (stop bit): 0 (1 bit) 1 (2 bits) D0 (data size): 0 (7 bits) 1 (8 bits)

Note Any value can be specified for bBaud, but the baud rate at which the device actually operates depends on the device.

3.6.2 SET_DTR_RTS

This request is used by the host to report the enabling or disabling of the DTR or RTS settings.

Table 3-10. SET_DTR_RTS Request Format**(a) Request codes**

Field	Size	Specified Value
bmRequestType	1	Request type: 0x40
bRequest	1	Request identifier: 0x00
wValue	2	Unused: 0x0000
wIndex	2	Unused: 0x0000
wLength	2	Data length: 0x0002 (the number of data stage bytes)

(b) Data

Field	Size	Specified Value
bRequest	1	Request identifier: 0x01 (SET_DTR_RTS)
bParams	1	D7 to D2 (Reserved): 000000 D1 (DTR): 0 (off) 1 (on) D0 (RTS): 0 (off) 1 (on)

3.6.3 SET_XON_XOFF_CHR

This request is used by the host to report the Xon or Xoff character code settings.

Table 3-11. SET_XON_XOFF_CHR Request Format**(a) Request codes**

Field	Size	Specified Value
bmRequestType	1	Request type: 0x40
bRequest	1	Request identifier: 0x00
wValue	2	Unused: 0x0000
wIndex	2	Unused: 0x0000
wLength	2	Data length: 0x0003 (the number of data stage bytes)

(b) Data

Field	Size	Specified Value
bRequest	1	Request identifier: 0x02 (SET_XON_XOFF_CHR)
XonChr	1	Xon character code
XoffChr	1	Xoff character code

3.6.4 OPEN_CLOSE

This request is used by the host to report whether ports are opened or closed.

Table 3-12. OPEN_CLOSE Request Format**(a) Request codes**

Field	Size	Specified Value
bmRequestType	1	Request type: 0x40
bRequest	1	Request identifier: 0x00
wValue	2	Unused: 0x0000
wIndex	2	Unused: 0x0000
wLength	2	Data length: 0x0002 (the number of data stage bytes)

(b) Data

Field	Size	Specified Value
bRequest	1	Request identifier: 0x03 (OPEN_CLOSE)
bOpen	1	Port status: 0x00 (The port is closed.) 0x01 (The port is opened.)

3.6.5 SET_ERR_CHR

This request is used by the host to report the settings of replacement character codes when an error occurs.

Table 3-13. SET_ERR_CHR Request Format**(a) Request codes**

Field	Size	Specified Value
bmRequestType	1	Request type: 0x40
bRequest	1	Request identifier: 0x00
wValue	2	Unused: 0x0000
wIndex	2	Unused: 0x0000
wLength	2	Data length: 0x0003 (the number of data stage bytes)

(b) Data

Field	Size	Specified Value
bRequest	1	Request identifier: 0x04 (SET_ERR_CHR)
bOpen	1	Port status: 0x00 (Replacing ErrChr is disabled.) 0x01 (Replacing ErrChr is enabled.)
ErrChr	1	Replacement character code if a parity error occurs

3.7 Function Specifications

This section describes the functions implemented in the sample software.

3.7.1 Functions

The functions of each source file included in the sample software are described below.

Table 3-14. Functions Implemented in the Sample Software

Source File	Function Name	Description
main.c	main	Main routine
	cpu_init	Initializes the CPU.
usbf78k.c	usbf78k_init	Initializes the USB function controller.
	usbf78k_standardreq	Processes standard requests.
	usbf78k_getdesc	Processes GET_DESCRIPTOR requests.
	usbf78k_data_send	Transmits data to the USB host.
	usbf78k_rdata_length	Acquires the USB reception data length.
	usbf78k_data_receive	Receives data from the USB host.
	usbf78k_clearFIFO	Clears the endpoint (FIFO) data.
	usbf78k_sendnullEP0	Issues a NULL packet for endpoint 0.
	usbf78k_sendstallEP0	Performs a STALL response for endpoint 0.
	usbf78k_put_txbuf	Copies the data received by the UART into the dedicated buffer.
	usbf78k_send_txbuf	Calls the usbf78k_data_send function.
	usbf78k_intusb0b	Serves an INTUSB0B interrupt.
	usbf78k_intusb1b	Serves an INTUSB1B interrupt (by using a BULK or INTERRUPT endpoint).
usbf78k_vendor.c	usbf78k_vendorreq	Responds to a vendor-specific request.
	usbf78k_line_control	Responds to a LINE_CONTROL request.
	usbf78k_set_dtr_rts	Responds to a SET_DTR_RTS request by issuing NULL.
	usbf78k_set_xon_xoff_chr	Responds to a SET_XON_XOFF_CHR request by issuing NULL.
	usbf78k_open_close	Responds to an OPEN_CLOSE request by issuing NULL.
	usbf78k_set_err_chr	Responds to a SET_ERR_CHR request by issuing NULL.
	usbf78k_usb_to_uart	Transfers data from the USB host to UART.
	usbf78k_uart_to_usb	Transfers data from the UART to the USB host.
uart_ctrl.c	uart78k_init	Initializes the serial interface UART6.
	uart78k_data_send	Calls the uart_send function.
	uart78k_uartmode_set	Responds to a SET_LINE_CODING request (and specifies the UART parameters.)
	uart_send	Transmits data to the UART terminal.
	uart_receive	Receives data from the UART terminal (and responds to interrupts).
	uart_receive_error	Indicates a failure to receive data from the UART terminal (and responds to interrupts).
boot.asm	—	Boot processing routine

3.7.2 Correlation of the functions

Some functions call other functions during the processing. The following figures show the correlation of the functions.

Figure 3-18. Functions Called During USB Interrupt Servicing

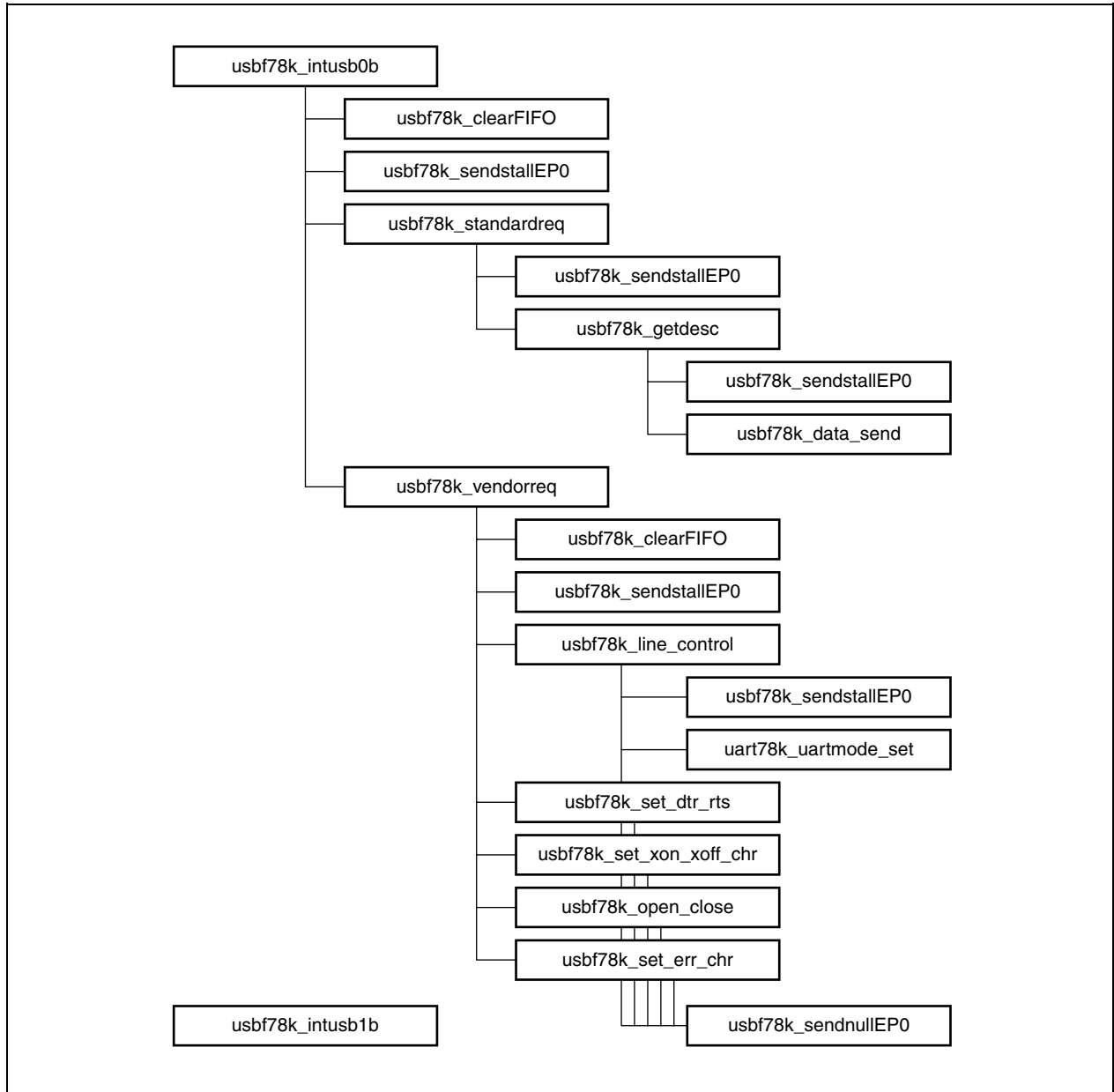
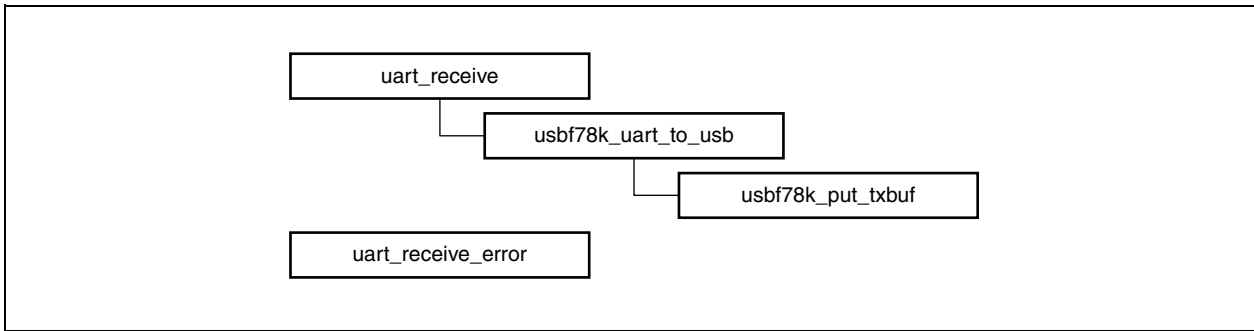
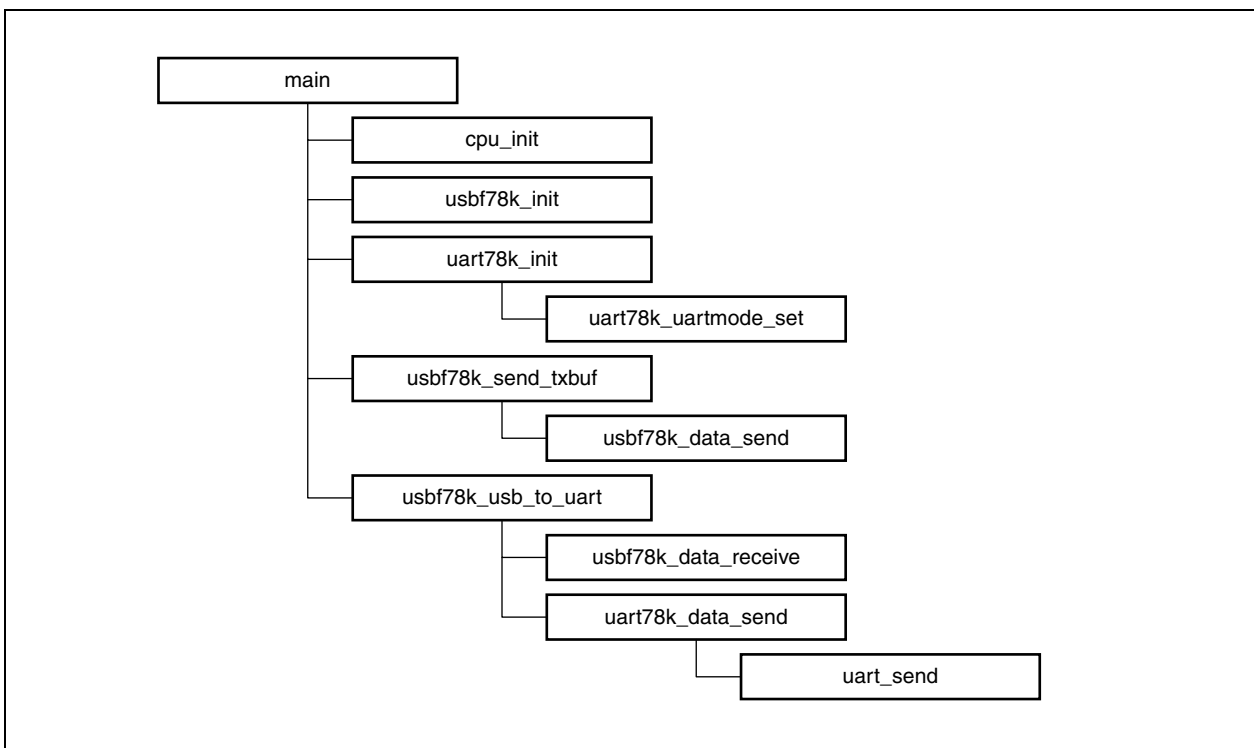


Figure 3-19. Functions Called During UART Interrupt Servicing**Figure 3-20. Functions Called in the Main Routine**

3.7.3 Function features

This section describes the features of the functions implemented in the sample software.

(1) Function description format

The functions are described in the following format.

<i>Function name</i>

[Overview]

An overview of the function is provided.

[C description format]

The format in which the function is written in C is provided.

[Parameters]

The parameters (arguments) of the function are described.

Parameter	Description
<i>Parameter type and name</i>	<i>Parameter summary</i>

[Return values]

The values returned by the function are described.

Symbol	Description
<i>Return value type and name</i>	<i>Return value summary</i>

[Description]

The feature of the function is described.

(2) Functions for the main processing**main****[Overview]**

Main processing

[C description format]

```
void main(void)
```

[Parameters]

None

[Return values]

None

[Description]

This function is called first when the sample software is executed.

This function executes initialization by calling the CPU initialization function (`cpu_init`), the USBF initialization function (`usbf78k_init`), and then the UART initialization function (`uart78k_init`).

Next, this function calls the following functions during its loop:

- Function for transmitting USB data to the ring buffer (`usbf78k_send_txbuf`) (transmits data from USB)
- Function for transmitting the data in the USB reception FIFO from the UART (`usbf78k_usb_to_uart`) (transmits the data received by USB from the UART)

cpu_init**[Overview]**

Initializes the CPU.

[C description format]

```
void cpu_init(void)
```

[Parameters]

None

[Return values]

None

[Description]

This function is called during the main processing.

This function specifies the settings required for using the μ PD78F0730, such as the memory sizes and clock frequency.

(3) Functions for the USB function controller**usbf78k_init****[Overview]**

Initializes the USB function controller.

[C description format]

```
void usbf78k_usbf_init(void)
```

[Parameters]

None

[Return values]

None

[Description]

This function is called during the main processing.

This function specifies the settings required for using the USB function controller, such as allocating and specifying the data area, and masking interrupt requests.

usbf78k_intusb0b**[Overview]**

Executes the INTUSB0B interrupt handler.

[C description format]

```
void usbf78k_intusb0b(void)
```

[Parameters]

None

[Return values]

None

[Description]

This function is executed when an INTUSB0B interrupt occurs.

If the interrupt source is RSUSPD, BUSRST, SETRQ, or CPUDEC, this function executes processing corresponding to each request.

If the interrupt source is CPUDEC, the request data (8 bytes) is retrieved and decoded. The request type is determined based on the result of decoding, and then the corresponding function is called to perform response processing.

usbf78k_intusb1b**[Overview]**

Executes the INTUSB1B interrupt handler.

[C description format]

```
void usbf78k_intusb1b(void)
```

[Parameters]

None

[Return values]

None

[Description]

This function is executed when an INTUSB1B interrupt occurs.

If the interrupt source is BKO1DT, the data reception flag (usbf78k_rdata_flg) is set to 1.

usbf78k_data_send**[Overview]**

Transmits USB data.

[C description format]

```
INT32 usbf78k_data_send(UINT8 *data, INT32 len, INT8 ep)
```

[Parameters]

Parameter	Description
UINT8 *data	Transmission data buffer pointer
INT32 len	Transmission data length
INT8 ep	Data transmission endpoint number

[Return values]

Symbol	Description
DEV_OK	Normal completion
DEV_ERROR	Abnormal termination

[Description]

This function stores the data stored in the transmission data buffer into the FIFO for the specified endpoint, byte by byte.

usbf78k_rdata_length**[Overview]**

Acquires the USB reception data length.

[C description format]

```
void usbf78k_rdata_length(INT32 *len , INT8 ep)
```

[Parameters]

Parameter	Description
INT32 *len	Address pointer for storing the reception data length
INT8 ep	Data reception endpoint number

[Return values]

None

[Description]

This function reads the reception data length of the specified endpoint.

usbf78k_data_receive**[Overview]**

Receives USB data.

[C description format]

```
INT32 usbf78k_data_receive(UINT8 *data, INT32 len, INT8 ep)
```

Parameter	Description
UINT8 *data	Reception data buffer pointer
INT32 len	Reception data length
INT8 ep	Data reception endpoint number

[Return values]

Symbol	Description
DEV_OK	Normal completion
DEV_ERROR	Abnormal termination

[Description]

This function reads data from the FIFO for the specified endpoint byte by byte and stores the data into the reception data buffer.

usbf78k_clearFIFO**[Overview]**

Clears FIFOs.

[C description format]

```
void usbf78k_clearFIFO(INT8 ep)
```

[Parameters]

Parameter	Description
INT8 ep	Endpoint number

[Return values]

None

[Description]

This function clears all FIFOs of the specified endpoint.

usbf78k_sendnullEP0**[Overview]**

Transmits a NULL packet for endpoint 0.

[C description format]

```
void usbf78k_sendnullEP0(void)
```

[Parameters]

None

[Return values]

None

[Description]

This function clears the FIFO for endpoint 0 and transmits a NULL packet from the USB function controller by setting the bit that indicates the end of data to 1.

usbf78k_sendstallEP0**[Overview]**

Performs a STALL response for endpoint 0.

[C description format]

```
void usbf78k_sendstallEP0(void)
```

[Parameters]

None

[Return values]

None

[Description]

This function makes the USB function controller perform a STALL response by setting the bit that indicates the use of STALL handshaking to 1.

usbf78k_standardreq**[Overview]**

Processes standard requests to which the USB function controller does not automatically respond.

[C description format]

```
void usbf78k_standardreq(void)
```

[Parameters]

None

[Return values]

None

[Description]

This function is called during USB interrupt servicing (INTUSB0B).

If a GET_DESCRIPTOR request is decoded, this function calls the GET_DESCRIPTOR request processing function (`usbf78k_getdesc`). For other requests, this function calls the function for processing STALL responses for endpoint 0 (`usbf78k_sendstallEP0`).

usbf78k_getdesc**[Overview]**

Processes GET_DESCRIPTOR requests.

[C description format]

```
void usbf78k_getdesc(void)
```

[Parameters]

None

[Return values]

None

[Description]

This function is called during the processing of standard requests to which the USB function controller does not automatically respond.

If a decoded request requests a string descriptor, this function calls the USB data transmission function (usbf78k_data_send) and transmits a string descriptor from endpoint 0. If a decoded request requests any other descriptor, this function calls the function for processing STALL responses for endpoint 0 (usbf78k_sendstallEP0).

usbf78k_put_txbuf**[Overview]**

Stores data into the transmission buffer.

[C description format]

```
void usbf78k_put_txbuf(UINT8 *data)
```

[Parameters]

Parameter	Description
UINT8 *data	Pointer of the data to be stored

[Return values]

None

[Description]

This function is called by the function for storing the data received by the UART into the USB transmission buffer (usbf78k_uart_to_usb).

This function stores the data received by the UART into the USB transmission ring buffer.

usbf78k_send_txbuf**[Overview]**

Transmits USB data (for the ring buffer).

[C description format]

```
INT32 usbf78k_send_txbuf(void)
```

[Parameters]

None

[Return values]

Symbol	Description
DEV_OK	Normal completion
DEV_ERROR	Abnormal termination

[Description]

This function transmits the data stored in the USB transmission ring buffer to the USB host.

(4) Functions for processing vendor requests and for communication between the UART and USB**usbf78k_vendorreq****[Overview]**

Processes vendor requests.

[C description format]

```
void usbf78k_vendorreq(void)
```

[Parameters]

None

[Return values]

None

[Description]

This function is called by the INTUSB0B interrupt handler function (`usbf78k_intusb0b`) when request data is received.

This function determines the type of vendor request and executes the corresponding processing.

usbf78k_line_ctrl**[Overview]**

Processes LINE_CONTROL requests.

[C description format]

```
void usbf78k_line_ctrl(void)
```

[Parameters]

None

[Return values]

None

[Description]

This function stores UART operation specification parameters specified at the LINE_CONTROL request data stage into the UART communication setting structure (`UART_MODE_TBL`).

This function calls the UART operation mode specification function (`uart78k_uartmode_set`) to change the UART operation mode.

When the operation mode has been changed, this function calls the function for processing NULL responses for endpoint 0 (`usbf78k_sendnullEP0`) to issue a NULL response.

usbf78k_set_dtr_rts**[Overview]**

Processes SET_DTR_RTS requests.

[C description format]

```
void usbf78k_set_dtr_rts(void)
```

[Parameters]

None

[Return values]

None

[Description]

This function calls the function for processing NULL responses for endpoint 0 (usbf78k_sendnullEP0).

This function responds normally and then ends because it is not used by the sample software.

usbf78k_set_xon_xoff_chr**[Overview]**

Processes SET_XON_XOFF_CHR requests.

[C description format]

```
void usbf78k_set_xon_xoff_chr(void)
```

[Parameters]

None

[Return values]

None

[Description]

This function calls the function for processing NULL responses for endpoint 0 (usbf78k_sendnullEP0).

This function responds normally and then ends because it is not used by the sample software.

usbf78k_open_close**[Overview]**

Processes OPEN_CLOSE requests.

[C description format]

```
void usbf78k_open_close(void)
```

[Parameters]

None

[Return values]

None

[Description]

This function calls the function for processing NULL responses for endpoint 0 (usbf78k_sendnullEP0).

This function responds normally and then ends because it is not used by the sample software.

usbf78k_set_err_chr**[Overview]**

Processes SET_ERR_CHR requests.

[C description format]

```
void usbf78k_set_err_chr(void)
```

[Parameters]

None

[Return values]

None

[Description]

This function calls the function for processing NULL responses for endpoint 0 (usbf78k_sendnullEP0).

This function responds normally and then ends because it is not used by the sample software.

usbf78k_usb_to_uart**[Overview]**

Transmits the data in the USB reception FIFO from the UART.

[C description format]

```
void usbf78k_usb_to_uart(UINT8 len)
```

[Parameters]

Symbol	Description
UINT8 len	Data length

[Return values]

None

[Description]

This function is called during the main processing.

This function calls the USB data reception function (`usbf78k_data_receive`) and stores the received data into a buffer.

This function calls the UART data transmission function (`uart78k_data_send`) and transmits the data stored in the buffer from the UART.

usbf78k_uart_to_usb**[Overview]**

Stores the data received by the UART into the USB transmission buffer.

[C description format]

```
void usbf78k_uart_to_usb(UINT8 *data)
```

[Parameters]

Symbol	Description
UINT8 *data	Reception data buffer pointer

[Return values]

None

[Description]

This function is called by the UART reception completion interrupt handler function (`uart_receive`).

This function calls the function for storing data into the transmission buffer (`usbf78k_put_txbuf`) and stores the data received by the UART into the USB transmission data buffer (which is a ring buffer).

(5) UART functions**uart78k_init****[Overview]**

Specifies the default settings for UART communication by initializing the UART.

[C description format]

```
void uart78k_init(void)
```

[Parameters]

None

[Return values]

None

[Description]

This function adds UART communication settings such as for the ports, baud rate, STOP bit, parity bit, and data length to a structure.

This function calls the UART operation mode specification function (`uart78k_uartmode_set`) to start the UART.

uart78k_uartmode_set**[Overview]**

Specifies the UART operation mode.

[C description format]

```
void uart78k_uartmode_set(void)
```

[Parameters]

None

[Return values]

None

[Description]

This function is called by the UART communication default setup function (`uart78k_init`) and LINE_CONTROL request function (`usbf78k_line_ctrl`).

This function stops the UART and sets up registers according to the UART communication settings saved in the structure.

Next, this function starts the UART and enables UART interrupts.

uart78k_data_send**[Overview]**

Transmits UART data.

[C description format]

```
void uart78k_data_send(UINT8 *buffer, UINT32 size)
```

[Parameters]

Symbol	Description
UINT8 *buffer	Data buffer pointer
UINT32 size	Data size

[Return values]

None

[Description]

This function calls the UART transmission function (`uart_send`) to execute UART transmission.

uart_send**[Overview]**

Transmits UART data.

[C description format]

```
void uart_send(UINT8 *sendString, UINT32 len)
```

[Parameters]

Symbol	Description
UINT8 *sendString	Transmission data buffer pointer
UINT32 len	Data length

[Return values]

None

[Description]

This function transmits every byte if the data length is 1 or more.

uart_receive**[Overview]**

Executes the UART reception completion interrupt handler.

[C description format]

```
__interrupt void uart_receive(void)
```

[Parameters]

None

[Return values]

None

[Description]

This function reads the data received by the UART and then calls the function for storing the data received by the UART into the USB transmission buffer (usb78k_uart_to_usb) to store the data received by the UART into the USB transmission buffer.

uart_receive_error**[Overview]**

Executes the UART reception error interrupt handler.

[C description format]

```
__interrupt void uart_receive_error(void)
```

[Parameters]

None

[Return values]

None

[Description]

This function clears error flags and then reads (and discards) the received data.

3.8 Data Structures

The sample software uses the following structures:

(1) USB device request structure

This structure is defined in the usb78k.h file.

The global variable UsbSetup_Data in the program is an instance of this structure.

```

/*-----
 * SETUP DATA structure
 *-----*/
typedef struct {
    UINT8  ReqType;           /* bmRequestType */
    UINT8  Request;          /* bRequest      */
    UINT16 Value;            /* wValue        */
    UINT16 Index;            /* wIndex        */
    UINT16 Length;           /* wLength       */
    UINT8* Data;             /* index to Data */
} Usb_Setup_st;

/*-----
 *   global variable
 *-----*/
extern Usb_Setup_st    UsbSetup_Data;

```

(2) UART communication setting structure

This structure is defined in the usb78k_vendor.h file.

```

typedef struct _UART_MODE_TBL{
    UINT8 DTERate[4];        /* transfer rate(bps) */
    UINT8 STOPBIT;           /* length of the stop bit - 0:1bit (1:1.5bits) 2:2bits */
    UINT8 PARITYType;        /* parity bit - 0:None 1:Odd 2:Even (3:Mark) 4:Space */
    UINT8 DATAbits;         /* data size (number of the bits:5,6,7,8,16) */
} UART_MODE_TBL , *PUART_MODE_TBL;

```

CHAPTER 4 DEVELOPMENT ENVIRONMENT

This chapter provides an example of creating an environment for developing an application program that uses the USB-to-serial conversion sample software for the μ PD78F0730 and the procedure for debugging the application.

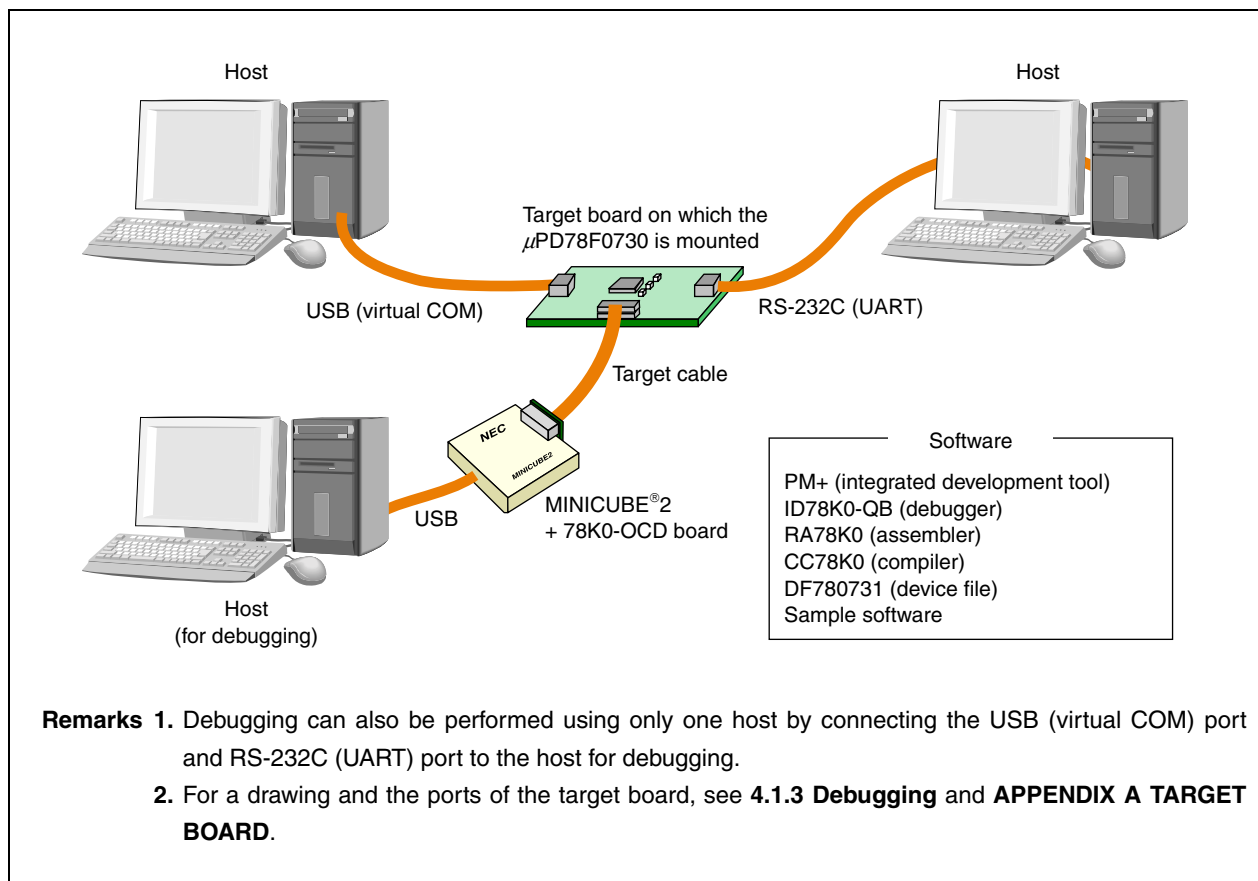
4.1 Used Products

This section describes the used hardware and software tool products.

4.1.1 System components

Figure 4-1 shows the components used in a system that uses the sample software.

Figure 4-1. Example of the System Components (During Debugging)



4.1.2 Program development

The following hardware and software are necessary to develop a system that uses the sample software:

Table 4-1. Example of the Components Used in a Program Development Environment

Components		Product Example	Remark
Hardware	Host	–	A PC/AT™-compatible PC using Windows™ XP or Windows Vista™
Software	Integrated development tool	PM+	V6.30
	Assembler	RA78K0	W4.00
	Compiler	CC78K0	W4.01
Files	Device file	DF780731	V1.10 (for the μ PD78F0730)
	Source files	Sample software	
	Include files		

4.1.3 Debugging

The following hardware and software are necessary to debug a system that uses the sample software:

Table 4-2. Example of the Components Used in a Debugging Environment

Components		Product Example	Remark
Hardware	Host	–	A PC/AT-compatible PC using Windows XP or Windows Vista
	Target	QB-78F0730-TB ^{Note 1}	A μ PD78F0730-mounted board that has USB and RS-232C (UART) interfaces
	On-chip debugging emulator	MINICUBE2	
	Cables	–	A USB cable, RS-232C cable, etc.
Software	Integrated development tool	PM+	V6.30
	Debugger	ID78K0-QB	V3.00
	Terminal	–	Note 2
Files	USB-to-serial conversion host driver	–	Included with the sample software
	Device file	DF780731	V1.10 (for the μ PD78F0730)
	Source files	Sample software	
	Include files		
	Project file		Note 3

Notes 1. An RS-232C interface must be connected to QB-78F0730-TB, because QB-78F0730-TB does not have an RS-232C interface.

2. A Windows terminal emulator, such as HyperTerminal (which is standard software provided in Windows), can be used.

3. A file that is used when creating a system using PM+ is included with the sample software.

4.2 Setting Up the Environment

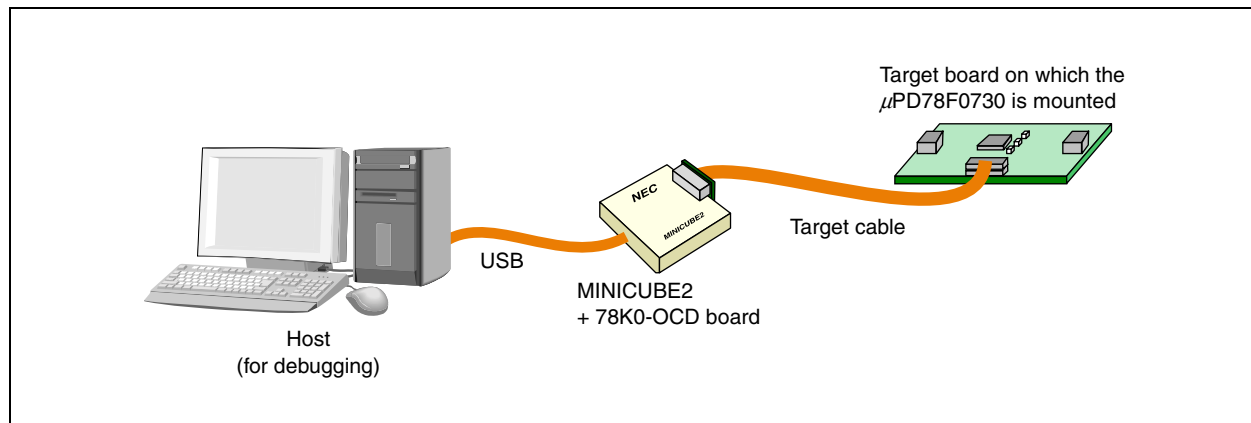
This section describes the preparations required for developing and debugging a system by using the products described in 4.1 Used Products.

4.2.1 Preparing the environment

Connect the target board to MINICUBE2, and connect MINICUBE2 to the host for debugging.

For details about how to connect MINICUBE2, see the **QB-MINI2 User's Manual (U18371E)**.

Figure 4-2. Target Board Connections



The procedure when using QB-78F0730-TB as the target board is described below.

(1) Setting up the switches

Flip the mode selection switch and power supply switch of MINICUBE2 to position M2 (78K0 microcontroller) and position 5 (supplying 5 V to the target system), respectively.

Caution Do not flip the MINICUBE2 switches while a USB cable is connected. Flip the switches after removing the USB cable.

(2) Setting up the 78K0-OCD board

Mount a 20 MHz oscillator. (A 20 MHz oscillator is mounted by default.)
Connect the 78K0-OCD board to MINICUBE2.

(3) Connecting the target board

Connect QB-78F0730-TB to the 78K0-OCD board by using a 10-pin cable.

(4) Connecting via USB

Connect MINICUBE2 to the host for debugging.

4.2.2 Preparing the host environment

Create a dedicated workspace on the host for debugging.

(1) Installing an integrated development tool

Install PM+. For details, see the **PM+ User's Manual**.

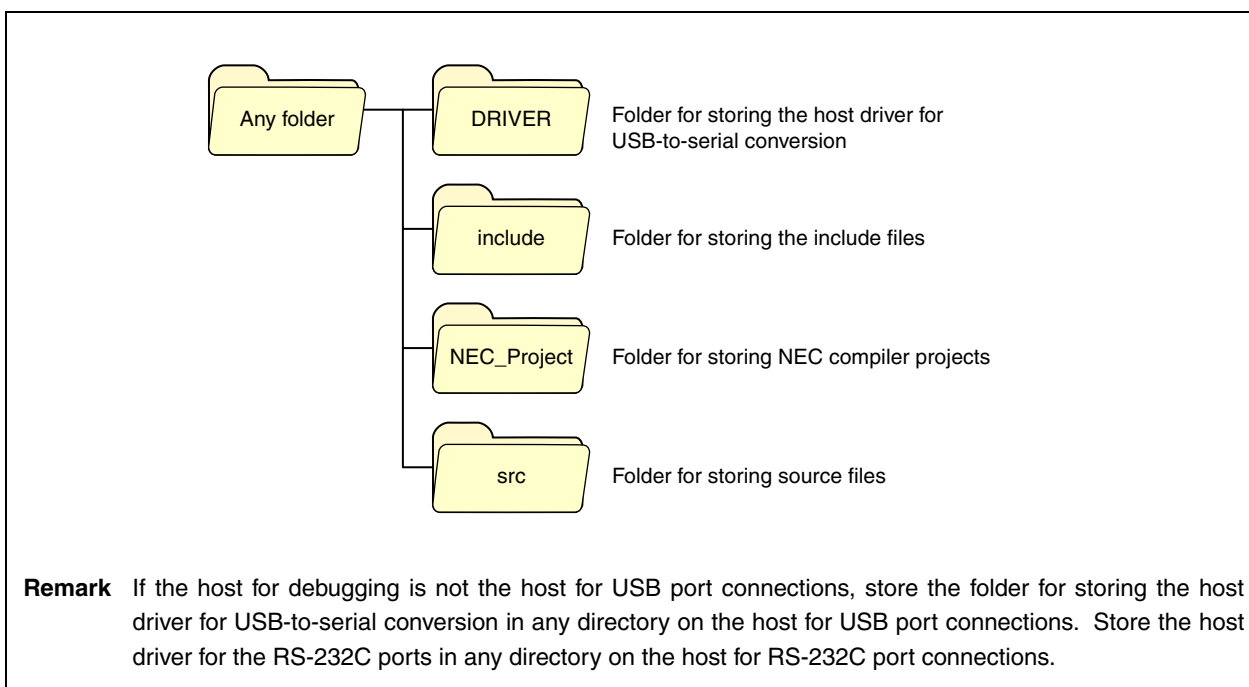
(2) Installing a debugger

Install ID78K0-QB. For details, see the **ID78K0-QB User's Manual**.

(3) Downloading the sample software

Store the complete set of sample software files into any directory without changing the folder structure.

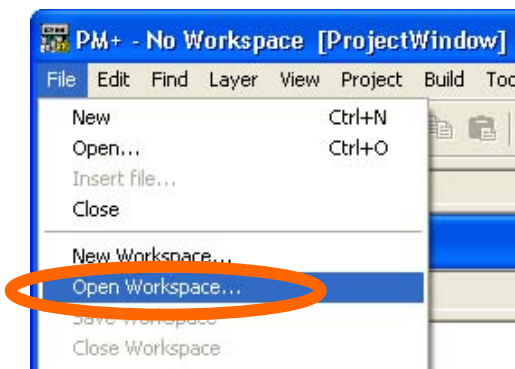
Figure 4-3. Folder Structure of the Sample Software



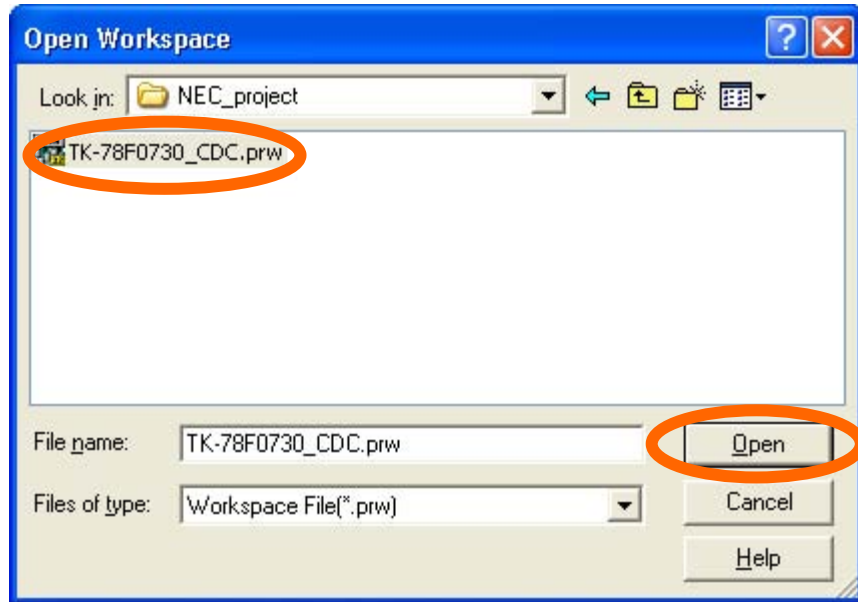
(4) Setting up the workspace

The procedure for using project files included with the sample software is described below.

<1> Start PM+, and then select **Open Workspace** in the **File** menu.



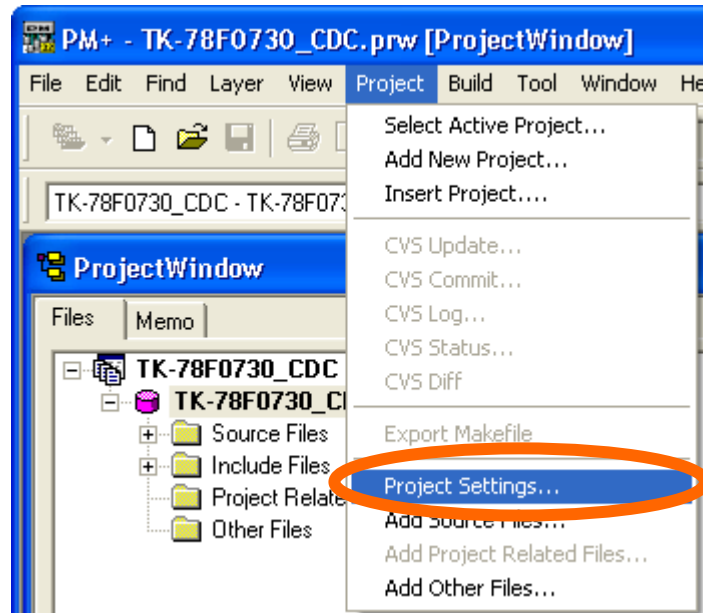
- <2> In the **Open Workspace** dialog box, specify the workspace file in the NEC_project folder, which is the sample software installation directory.



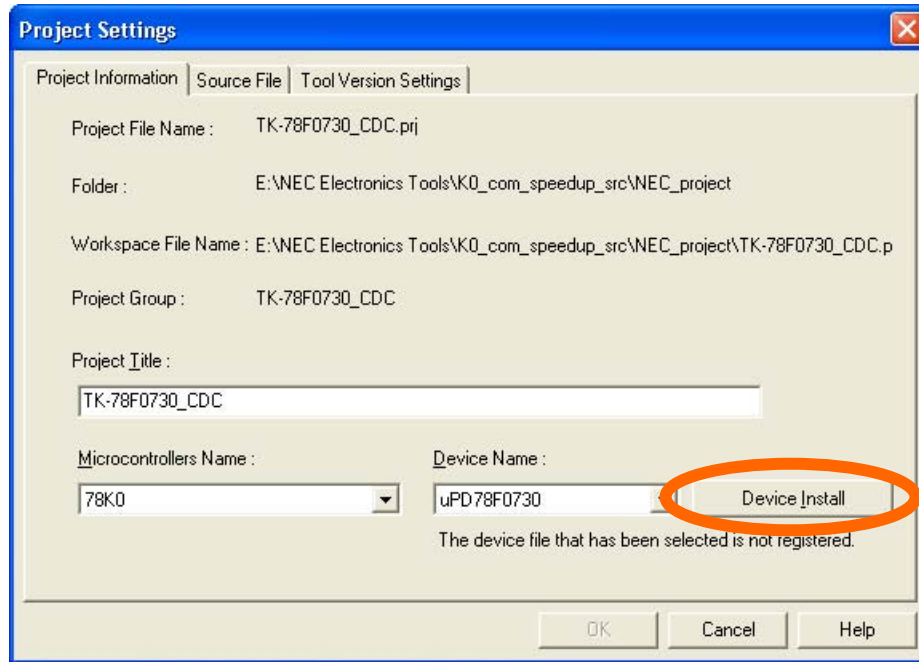
(5) Installing a device file

The procedure for using a device file for the μ PD78F0730 is described below.

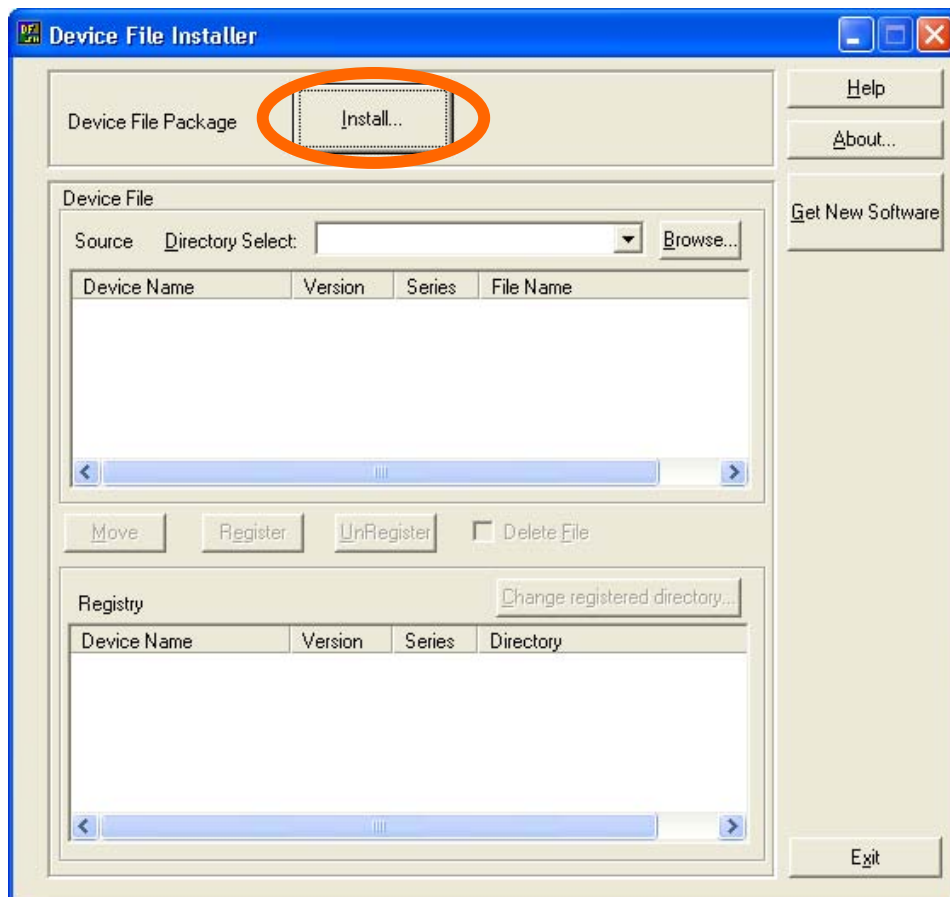
- <1> Select **Project Settings** in the PM+ **Project** menu.



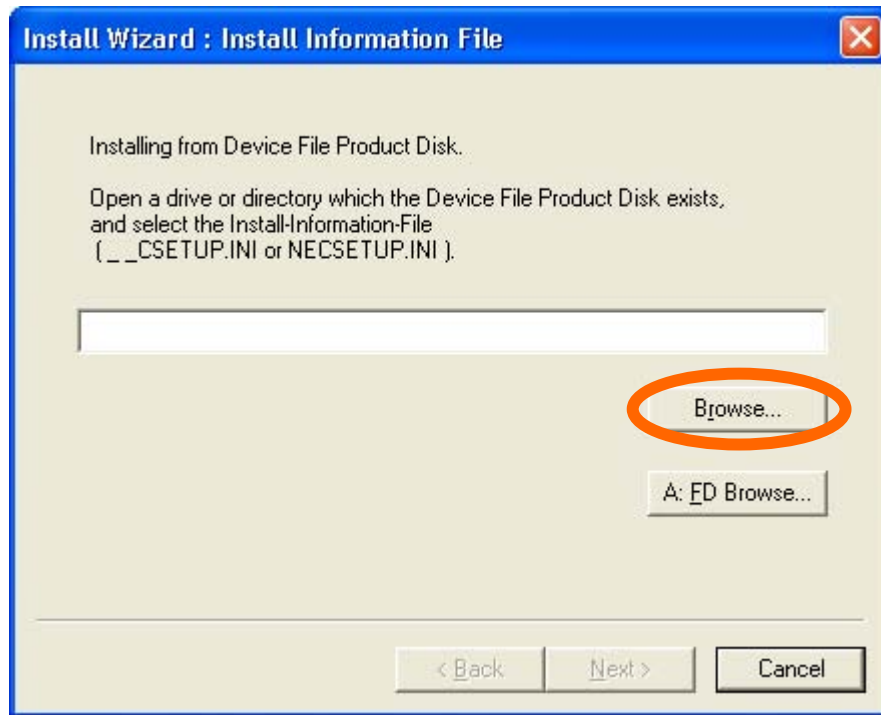
- <2> In the **Project Settings** dialog box, click the **Device Install** button on the **Project Information** tab to start the Device File Installer.



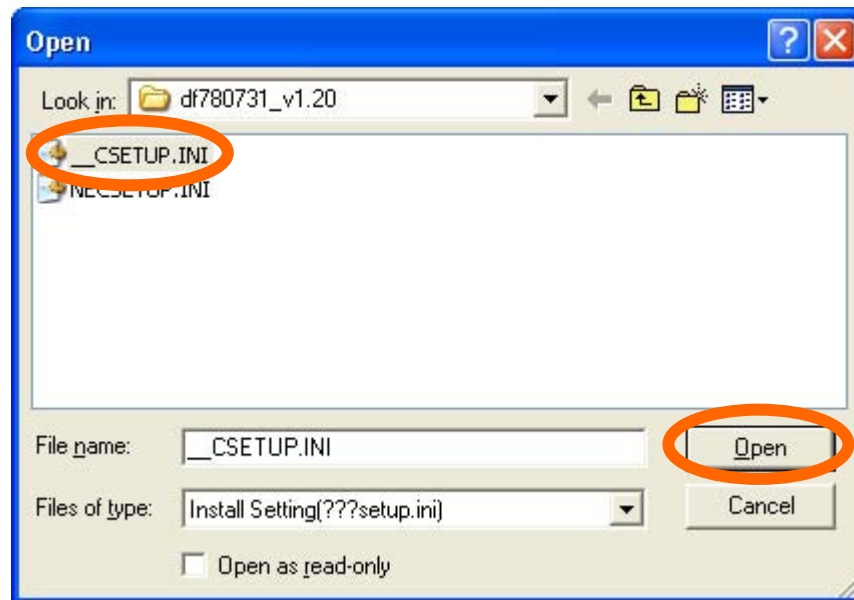
- <3> In the **Device File Installer** dialog box, click the **Install...** button to start the installation wizard.



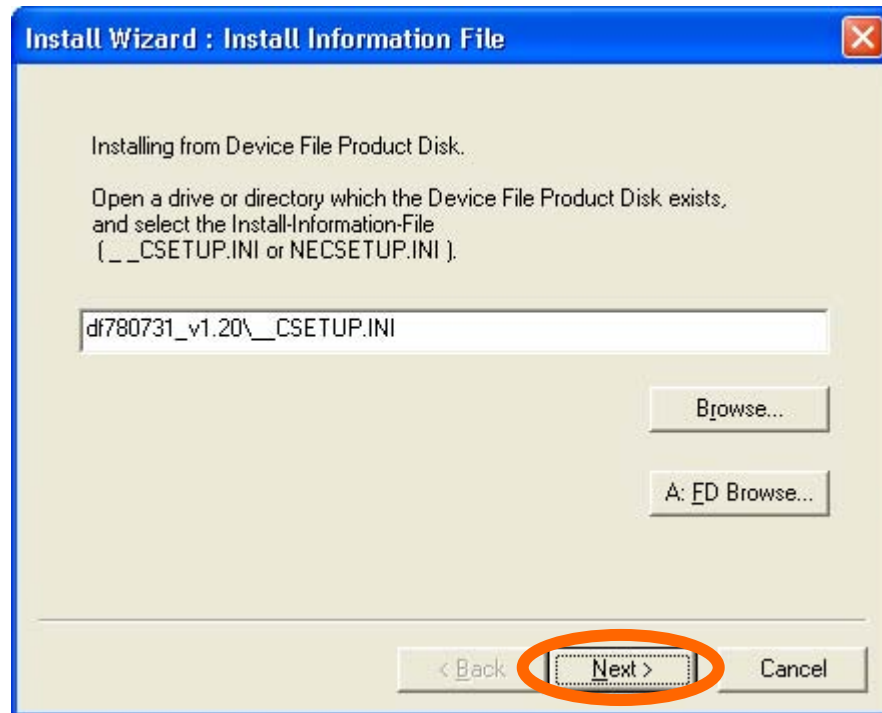
<4> In the **Install Information File** dialog box, click the **Browse** button.



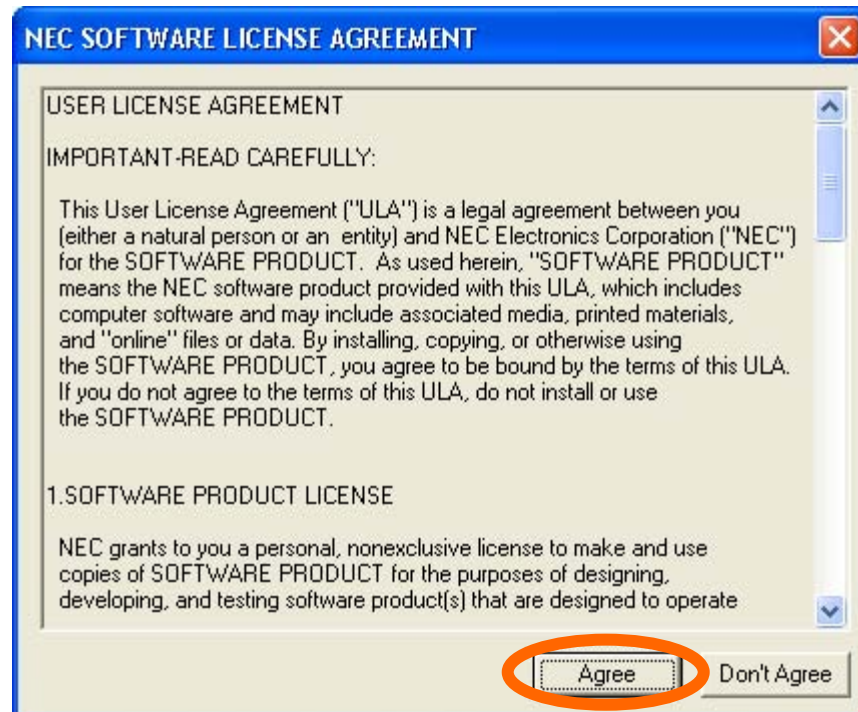
<5> In the **Open** dialog box, open the directory in which the device file was stored, select the **_CSETUP.INI** file, and then click the **Open** button.



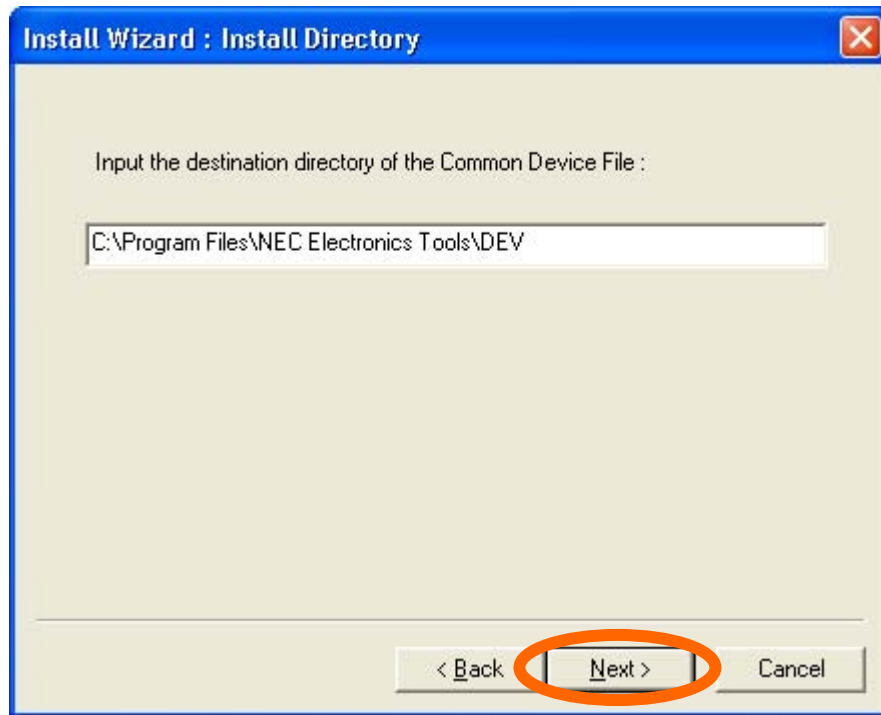
<6> In the **Install Information File** dialog box, click the **Next** button.



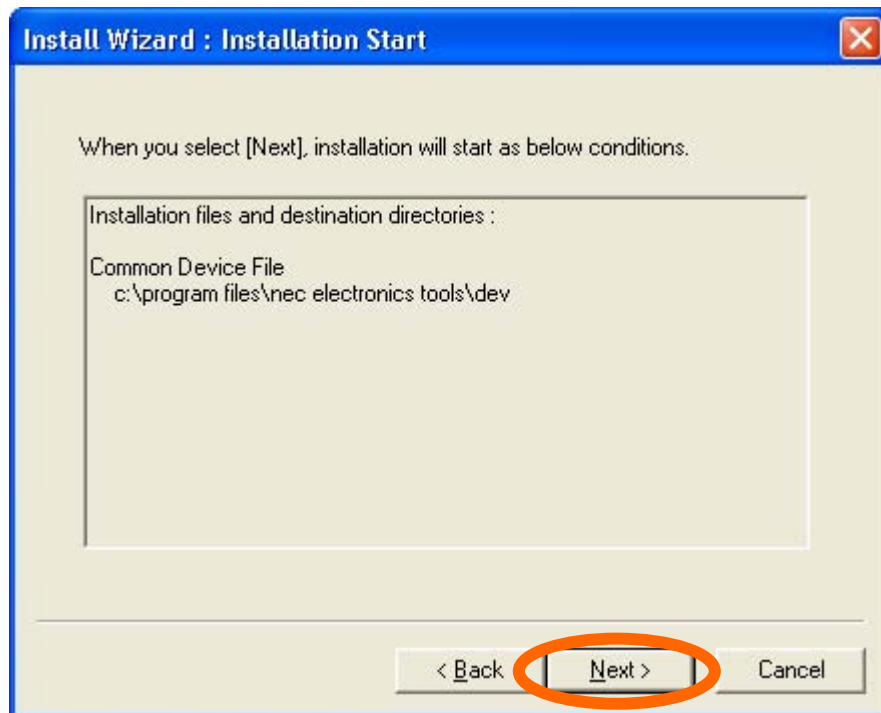
<7> In the **NEC SOFTWARE LICENSE AGREEMENT** dialog box, read the license agreement, and then click the **Agree** button if you agree with the terms.



<8> In the **Install Directory** dialog box, confirm that a path is displayed, and then click the **Next** button.

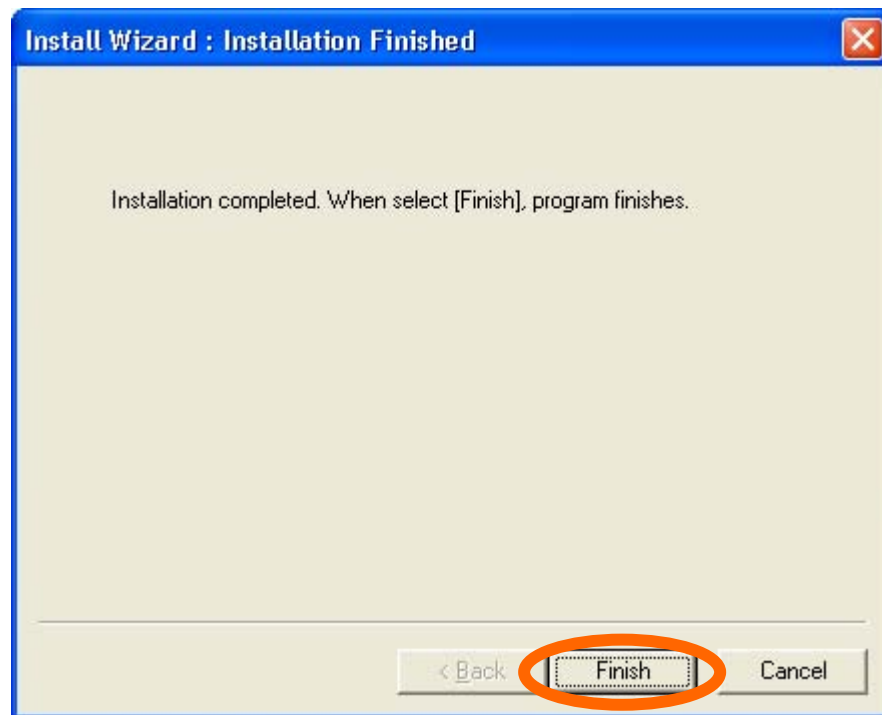


<9> In the **Installation Start** dialog box, click the **Next** button.



<10> A device file is installed to the project. This might take a while depending on the environment.

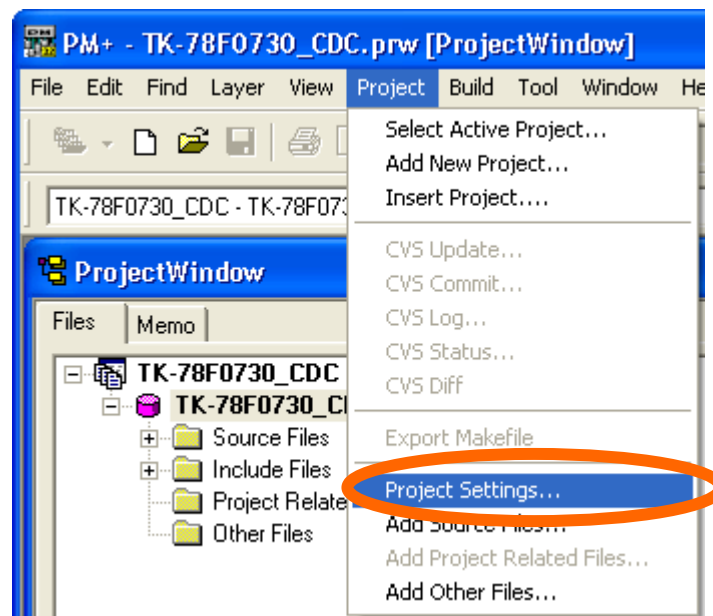
<11> In the **Installation Finished** dialog box, click the **Finish** button.



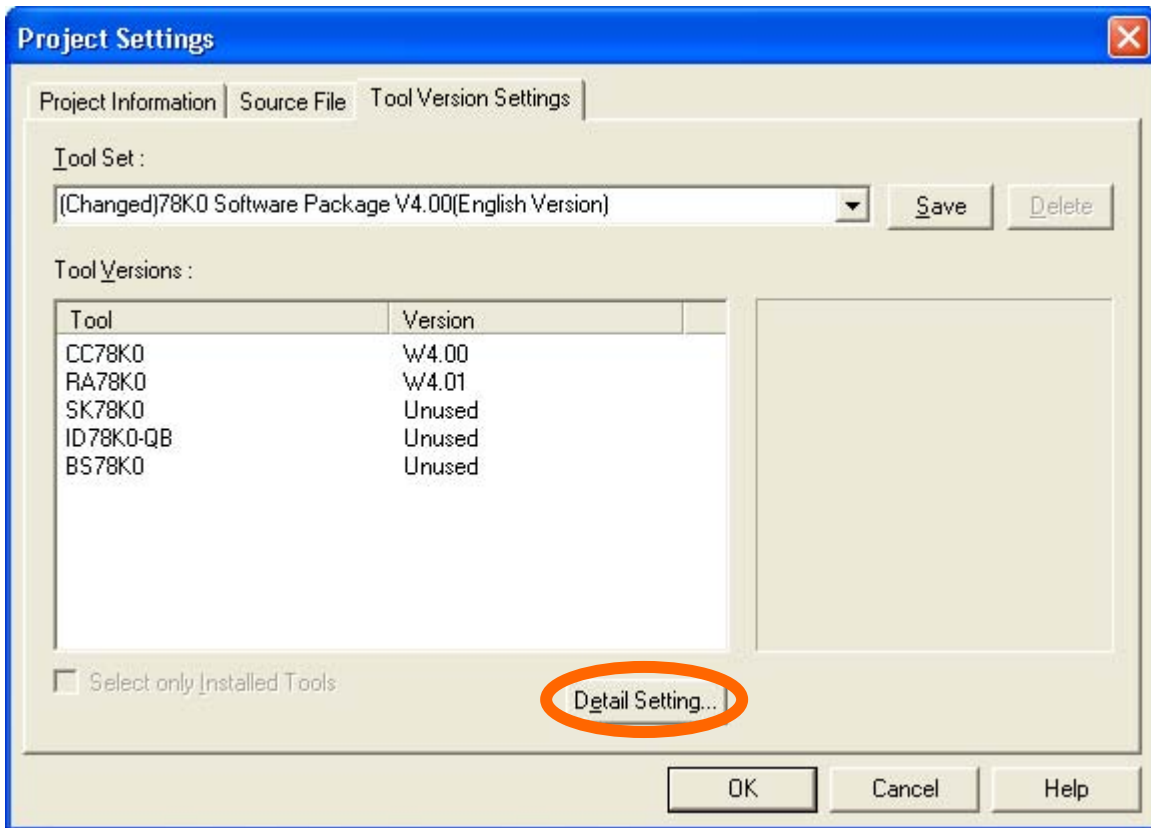
(6) Setting up the building tool

The procedure for using CC78K0 as the building tool and ID78K0-QB as the debugging tool is described below.

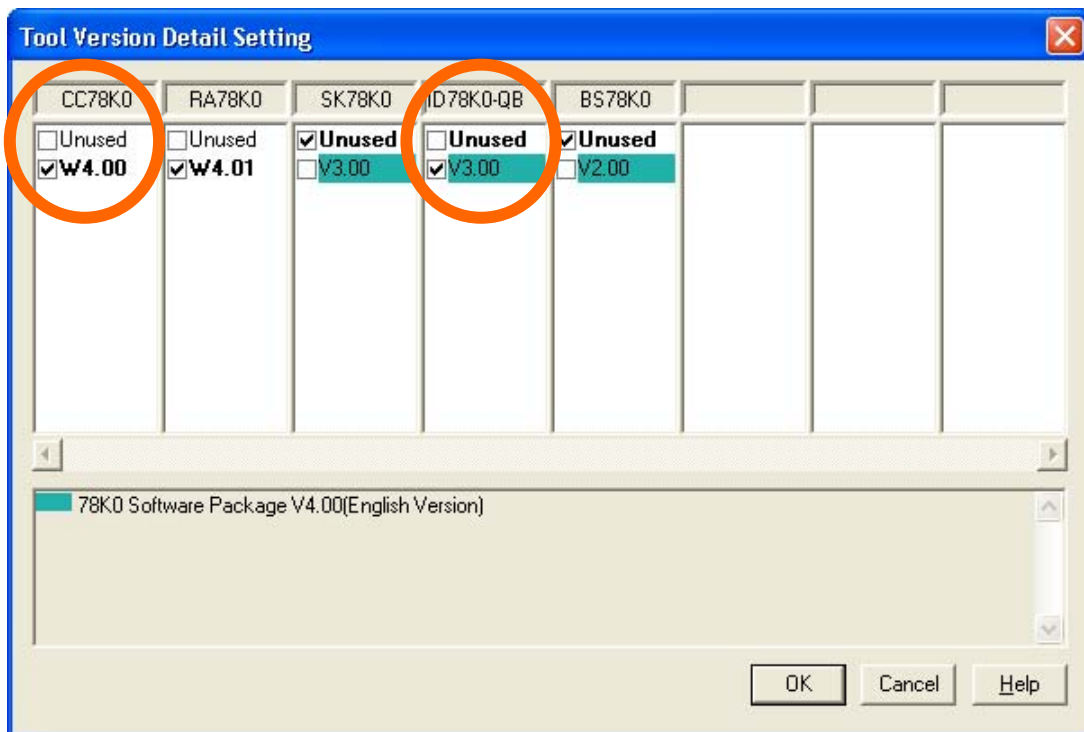
<1> Select **Project Settings** in the PM+ **Project** menu.



<2> In the **Project Settings** dialog box, click the **Detail Setting...** button on the **Tool Version Settings** tab.



<3> In the **Tool Version Detail Setting** dialog box, select the compiler version to use in the CC78K0 column and the debugger version to use in the ID78K0-QB column.



4.3 On-Chip Debugging

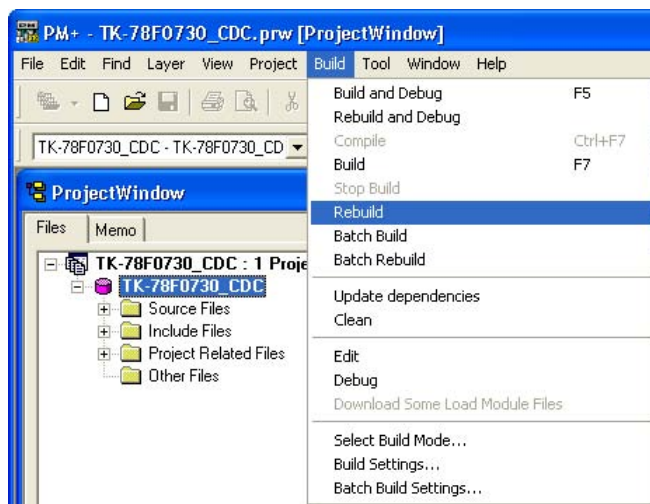
This section describes the procedure for debugging an application program that was developed using the workspace described in **4.2 Setting Up the Environment**.

For the μ PD78F0730, a program can be written to its internal flash memory and the program operation can be checked by directly executing the program by using a debugger (on-chip debugging).

4.3.1 Generating a load module

To write a program to the target device, use a C compiler to generate a load module by converting a file written in C or assembly language.

For PM+, generate a load module by selecting **Rebuild** in the **Build** menu.



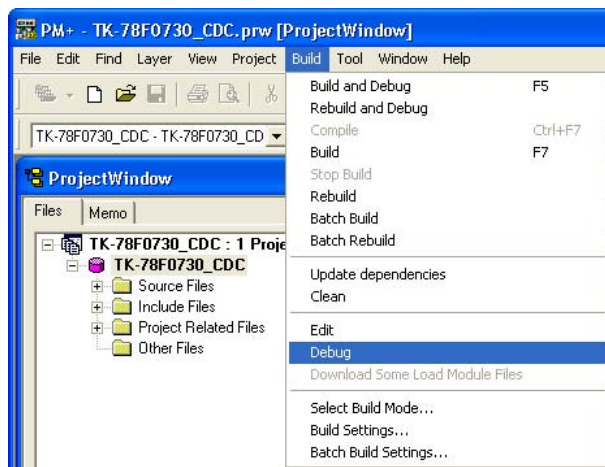
4.3.2 Loading and executing the load module

Execute the generated load module by writing (loading) it to the target.

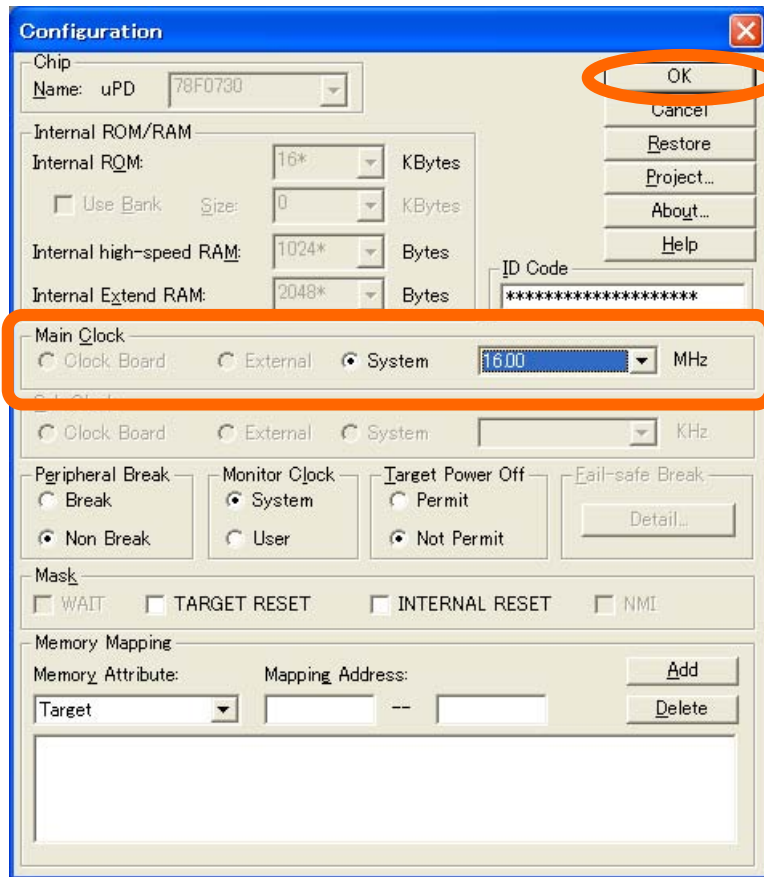
(1) Writing the load module

The procedure for writing the load module to the μ PD78F0730 on the target board by using PM+ is described below.


<1> Start ID78K0-QB-EZ by selecting **Debug** in the **Build** menu.

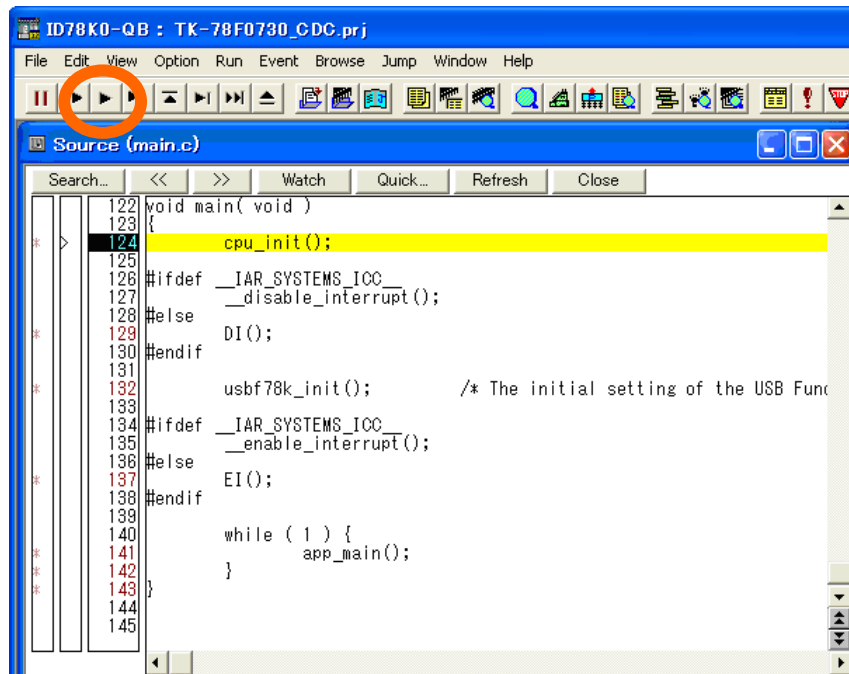


<2> In the **Configuration** dialog box, check the **Main Clock** setting, and then click the **OK** button.



(2) Executing the program

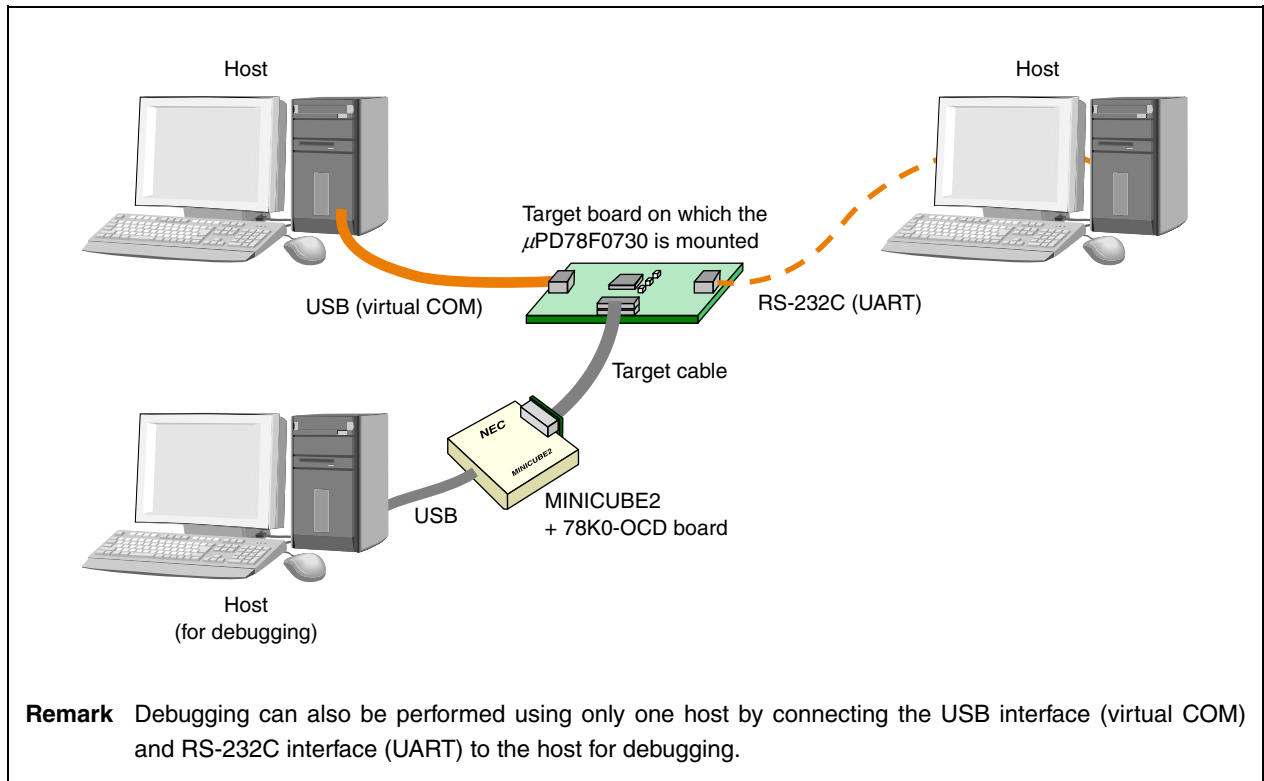
Click the  button in the **ID78K0-QB** window or select **Run Without Debugging** in the **Run** menu.



4.3.3 Connecting the USB port (virtual COM port)

Connect the USB port of the target board to the USB port of the host while the sample program is running.

Figure 4-4. Connecting the USB Port (Virtual COM Port)



(1) Installing a host driver

A USB-to-serial conversion host driver must be installed on the host.

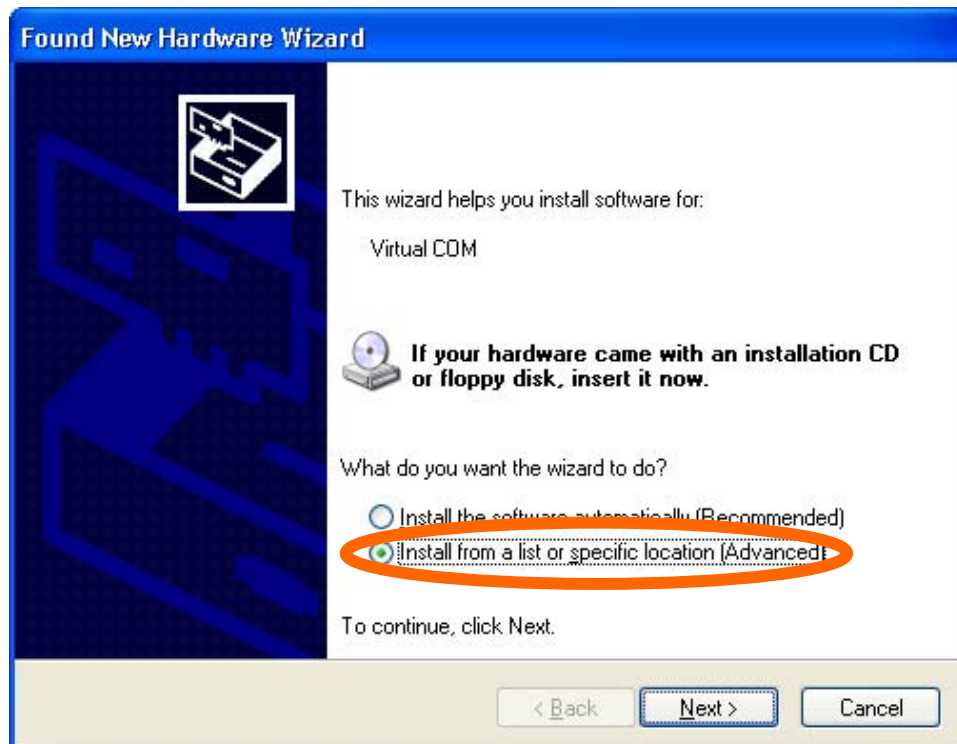
The procedure for using the USB-to-serial conversion host driver included with the sample software is described below.

- <1> When the connection of the target board is recognized by the host, the Found New Hardware message is displayed, and then the **Found New Hardware Wizard** starts.

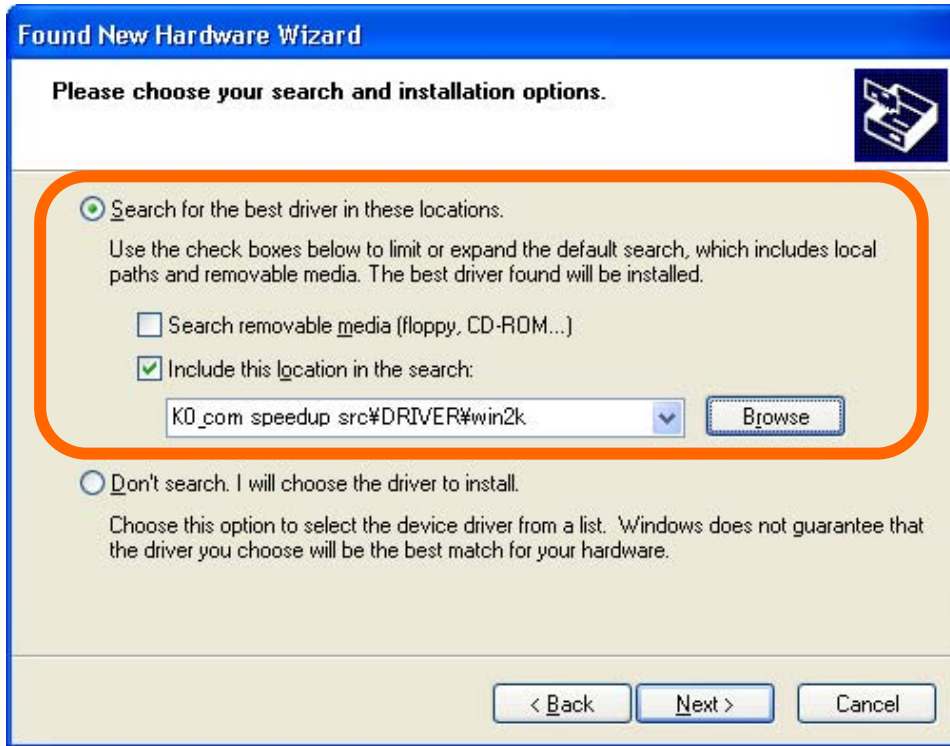
- <2> On the first page of the **Found New Hardware Wizard**, select **No, not this time**, and then click the **Next** button.



- <3> On the next page, select **Install from a list or specific location (Advanced)** and then click the **Next** button.



- <4> On the next page, select **Search for the best driver in these locations**, select **Include this location in the search**, specify the host driver folder, and then click the **Next** button.

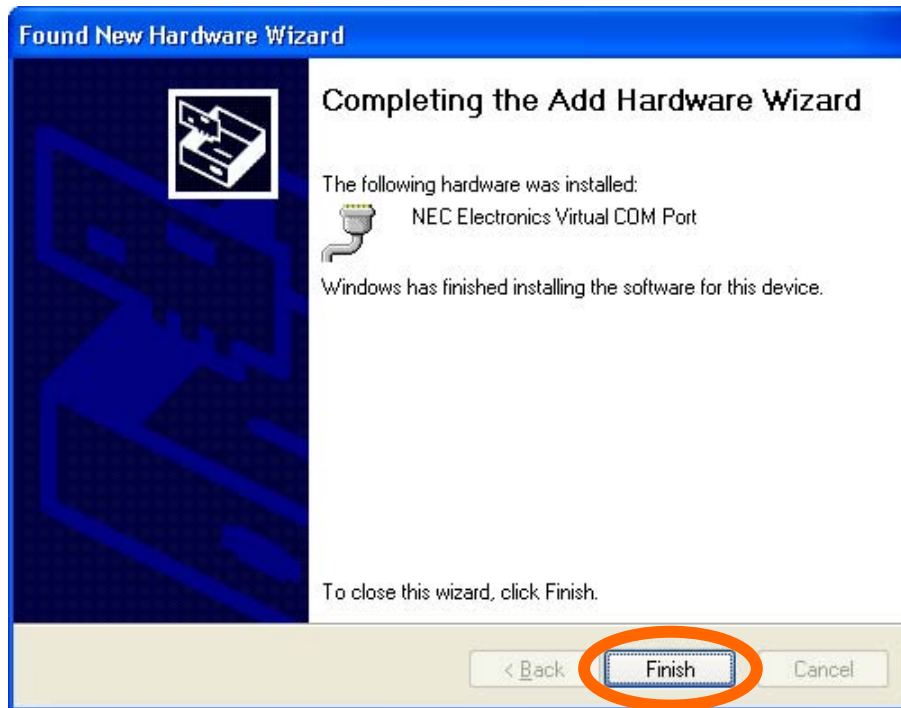


Remark Select a host driver that matches the OS on the host. For example, select win2k, which is in the DRIVER folder, for a 32-bit version of Windows XP or select wlh_amd64, which is in the DRIVER folder, for a 64-bit version of Windows XP.

- <5> In the **Hardware Installation** dialog box, click the **Continue Anyway** button.

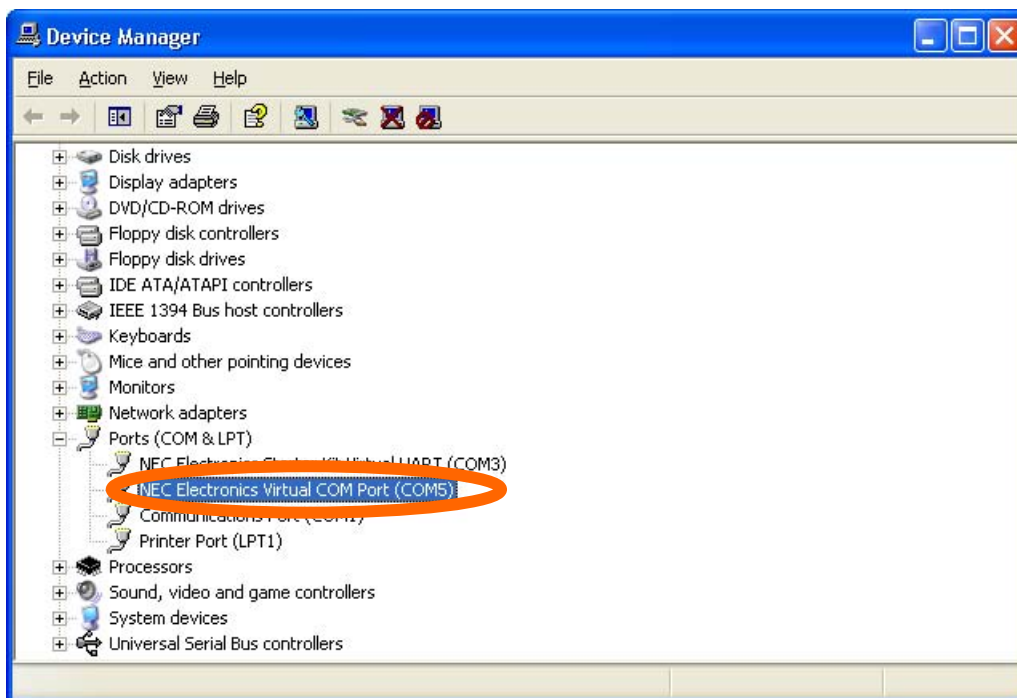


<6> On the next page of the **Found New Hardware Wizard**, click the **Finish** button.



(2) **Checking the device assignment**

Open the Windows Device Manager. In the **Ports (COM & LPT)** category, make sure that **NEC Electronics Virtual COM Port** is displayed and check the assigned COM port number.

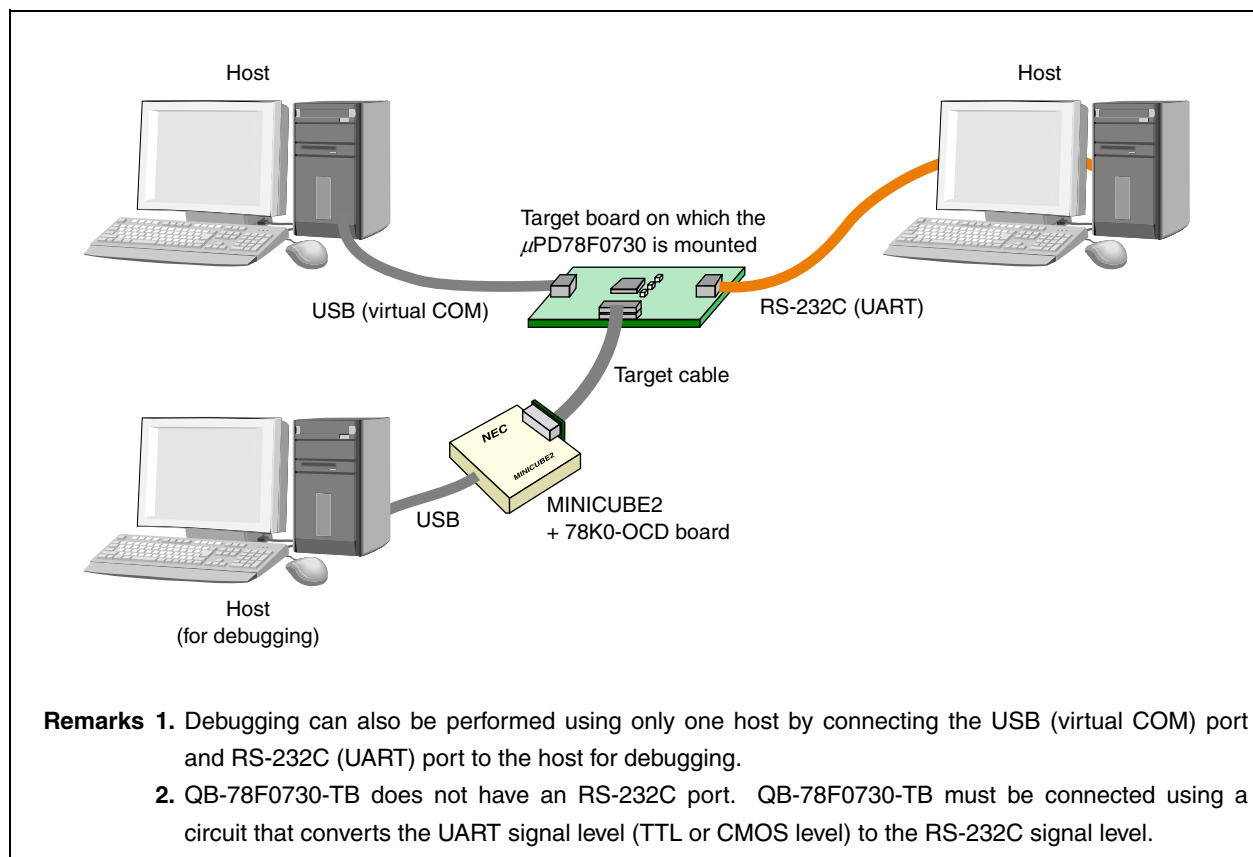


Remark Device names and port numbers can be changed. For details, see **5.2 Customizing the Sample Software**.

4.3.4 Connecting the RS-232C port

Connect the RS-232C port of the target board to the host.

Figure 4-5. Connecting the RS-232C Port



(1) Checking the device assignment

Open the Windows Device Manager. In the **Ports (COM & LPT)** category, check the COM port number assigned to the RS-232C port.

The displayed name differs depending on the installed driver.

4.3.5 Checking the operation

The operation of the sample software is checked by using terminal software on the host.

An example in which HyperTerminal, which is installed as standard Windows software, is used is provided below.

(1) Setting up HyperTerminal for the USB side

Set up the host port that is connected to the target board via USB.

- <1> Start HyperTerminal by selecting **Start, All Programs, Accessories, Communications**, and then **HyperTerminal**.
- <2> In the **Connection Description** dialog box, enter **USB Port** in the Name field and then click the **OK** button. Any icon can be selected.



- <3> In the **Connect To** dialog box, select the virtual COM port number of the USB port for **Connect using**, and then click the **OK** button.

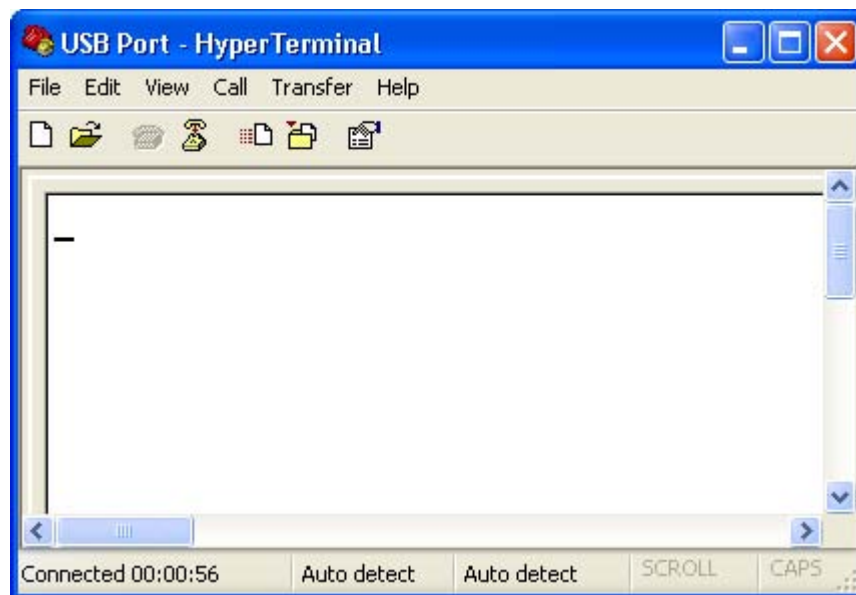
The virtual COM port number is displayed for **NEC Electronics Virtual COM Port** in the Device Manager. For details about how to check this number, see **4.3.3 (2) Checking the device assignment**.



- <4> In the dialog box displayed for the selected COM port, select a transfer speed (in bits per second) from 115200, 76800, 38400, 19200, 9600, 4800, and 2400 bps, and then click the **OK** button. Leave the other items as they are set by default.



- <5> The terminal window of the USB port side is created.

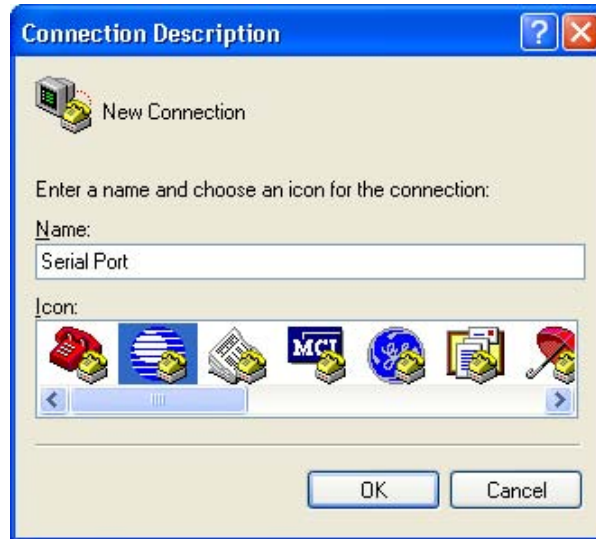


(2) Setting up HyperTerminal for the RS-232C side

Set up the host port that is connected to the target board via RS-232C.

Remark A host can also be connected to the target board by using both USB and RS-232C.

- <1> Start HyperTerminal by selecting **Start, All Programs, Accessories, Communications, and then HyperTerminal**.
- <2> In the **Connection Description** dialog box, enter **Serial Port** in the Name field and then click the **OK** button. Any icon can be selected.



- <3> In the **Connect To** dialog box, select the COM port number of the RS-232C port for **Connect using**, and then click the **OK** button.

The COM port number is displayed for the debugging port in the Device Manager. For details about how to check this number, see **4.3.3 (2) Checking the device assignment**.

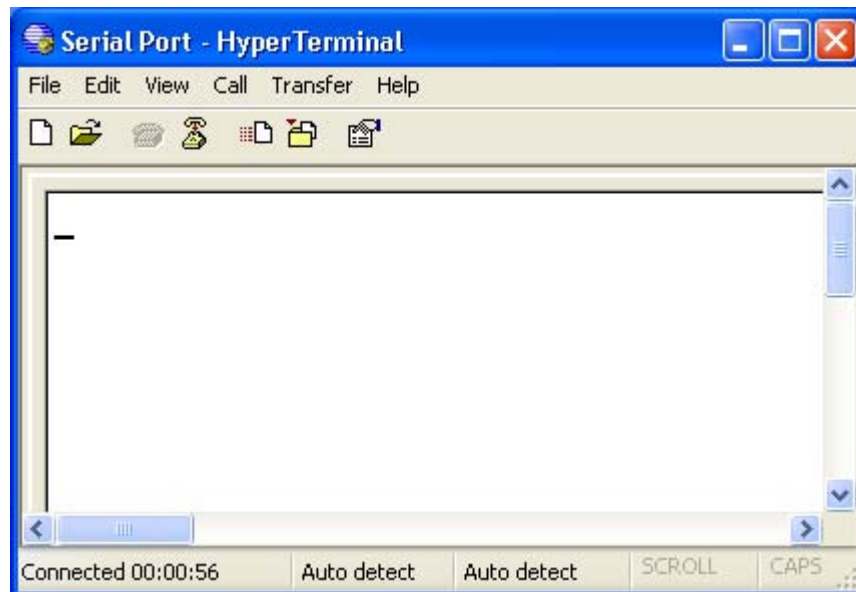


- <4> In the dialog box displayed for the selected COM port, select a transfer speed (in bits per second), and then click the **OK** button.

Caution Be sure to select the same transfer speed for the USB port and serial port.



- <5> The terminal window of the serial port side is created.

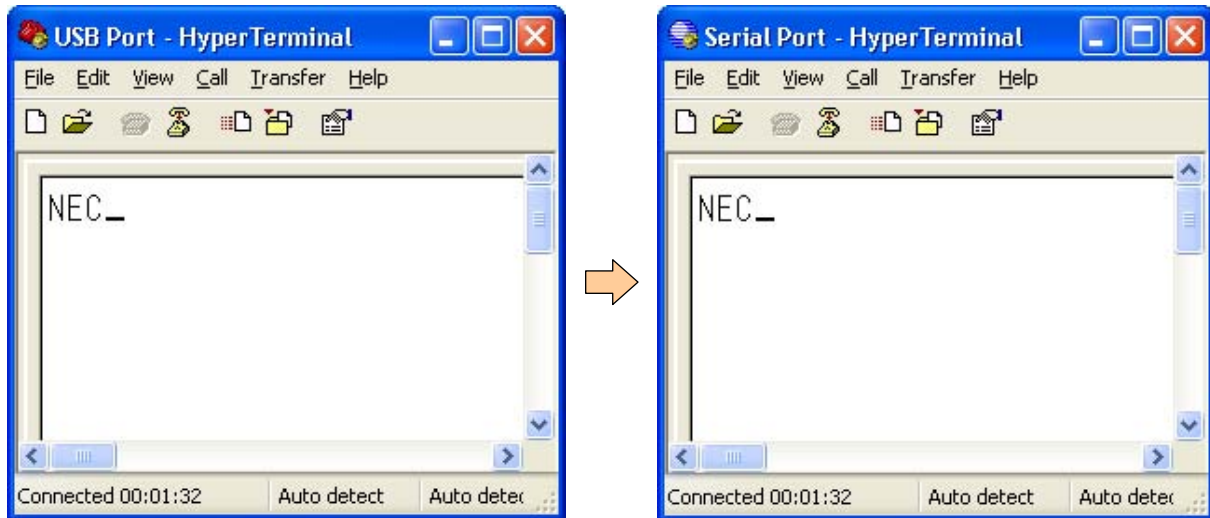


(3) Checking communication

Check the communication status by entering alphabetic characters (1-byte characters) into HyperTerminal.

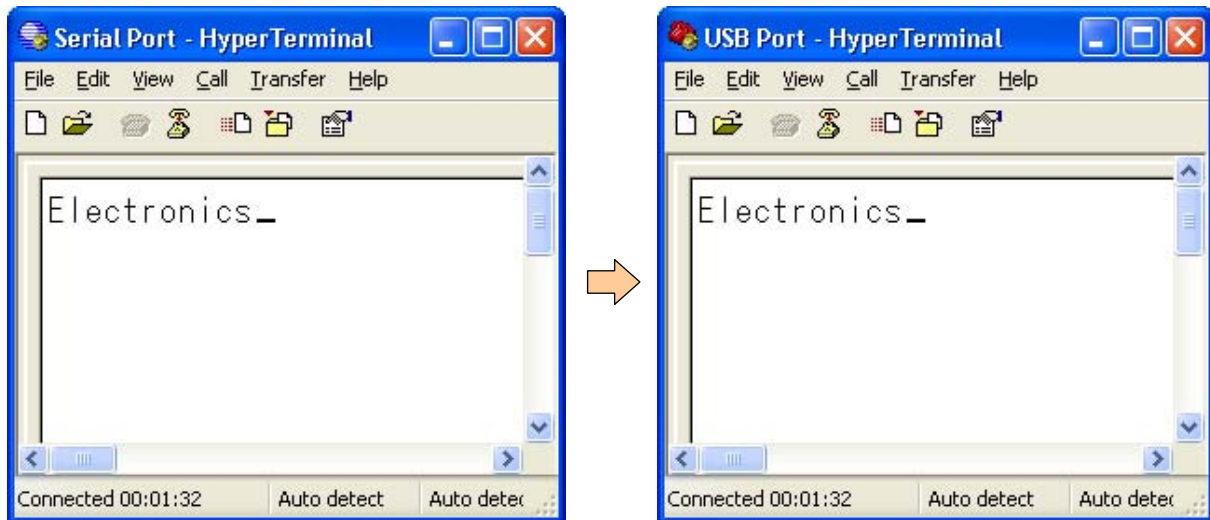
The transfer from the USB port to the serial port is normal if characters entered in the **USB Port - HyperTerminal** window are displayed in the **Serial Port - HyperTerminal** window.

Figure 4-6. Checking the Transfer from the USB Port to the Serial Port



The transfer from the serial port to the USB port is normal if characters entered in the **Serial Port - HyperTerminal** window are displayed in the **USB Port - HyperTerminal** window.

Figure 4-7. Checking the Transfer from the Serial Port to the USB Port



4.4 Cautions

When transmitting data from the serial port (RS-232C) to the USB port, some data might be lost depending on conditions such as the communication speed and status of the host OS.

4.4.1 Recommended communication speed

Specify the communication speed in whichever of the following ranges corresponds to the host OS:

- Windows 2000: 38,400 bps or less
- Windows XP or Vista: 115,200 bps or less

If data is communicated at a speed that exceeds the corresponding range, data is more likely to be lost. Data is also more likely to be lost if it is transferred from the USB port to the serial port and from the serial port to the USB port at the same time.

4.4.2 Causes of data loss

Two FIFOs (two 64-byte FIFOs) are provided for the endpoints for bulk-in transfer (transmission) performed by the USB function controller of the μ PD78F0730. Each of these FIFOs stores data of one packet to be transmitted as the USB packet. Therefore, for a transfer such as one that uses serial communication, in which the data sizes are not fixed, the maximum size of data might not be buffered.

The sample software provides an internal buffer for buffering data. This ring buffer can store serially communicated data of the maximum buffer size, regardless of the data transfer unit. (The buffer size is defined in the `usb78k.h` file and can be changed.)

(1) Delay in issuing requests

USB communication is performed in response to requests issued by the host. The sample software stores the data received via serial communication into FIFOs and prepares to transfer the data to the USB host. However, if the request issuance is delayed due to a processing load on the USB host OS, the next serially communicated data is delivered before transfer to the USB port ends. If both the FIFOs and internal buffer fill up in this way, the serially communicated data is read and discarded. Loss of data rarely happens during transfer from the USB port to the serial port, because data can be transmitted from the μ PD78F0730 to the serial port at any time.

(2) Bidirectional communication

During bidirectional communication, the transmission of USB data might be delayed when USB data is received, due to the structure of the sample software. The faster the communication speed, the more likely the buffer of the sample software fills up during this period.

(3) UART reception error

Both when there is a delay in issuing a request and during bidirectional communication, serially received data is read, discarded, and lost when the buffer fills up. If data is delayed at the serial port without reading and discarding it, a UART reception error (overrun error) occurs and the data is lost. The sample software reads and discards data to prevent UART reception errors from occurring. (Depending on the communication speed, data might not be read and discarded in time and an error might occur.)

CHAPTER 5 USING THE SAMPLE SOFTWARE

This chapter describes information that you should know when using the USB-to-serial conversion sample software for the μ PD78F0730.

5.1 Overview

The sample software can be used in the following two ways:

(1) Customizing the sample software

Rewrite the following sections of the sample software as required:

- The application section in the main.c file
- The values specified for the registers in the usbf78k_sfr.h file
- The descriptor information in the usbf78k_desc.h file
- The device names and provider information in the host driver for the virtual COM port (the INF file)

Remark For the list of files included in the sample software, see **1.1.3 Files included in the sample software**.

(2) Using functions

Call functions from within the application program as required. For details about the provided functions, see **3.7 Function Specifications**.

5.2 Customizing the Sample Software

This section describes the sections to rewrite as required when using the sample software.

5.2.1 Application section

The main routine function (`main`) in the main.c file includes a simple example of processing using the sample software. The existing initialization processing and interrupt servicing can be used by including the processing to actually use for the application in this section.

List 5-1. Main Routine

```

1  /*=====
2  Main function
3  void main( void )
4
5  Arguments:
6      N/A
7  Return values:
8      N/A
9  Overview:
10     main routine.
11  =====*/
12 void main( void )
13 {
14     cpu_init();
15
16     #ifdef __IAR_SYSTEMS_ICC__
17         __disable_interrupt();
18     #else
19         DI();
20     #endif
21
22     usbf78k_init();      /* The initial setting of the USB Function
23                          */
24     uart78k_init();      /* The initial setting of the CDC device */
25
26     #ifdef __IAR_SYSTEMS_ICC__
27         __enable_interrupt();
28     #else
29         EI();
30     #endif
31
32     while ( 1 ) {
33         usbf78k_send_txbuf();
34
35         if ( usbf78k_rdata_flg ) {
36             usbf78k_rdata_flg = 0;
37             /* transfers to UART */
38             usbf78k_usb_to_uart(UF0B01L);
39         }
40     }

```

5.2.2 Setting up the registers

The registers the sample software uses (writes to) and the values specified for them are defined in the `usbf78k_sfr.h` file. By rewriting the values in this file according to the actual use for the application, the operation of the target device can be specified by using the sample software.

(1) `usbf78k_sfr.h` file

This file includes the definitions of the USB function controller registers, the register bits used in various types of processing, and the values specified for the bits. (For details, see **3.3.1 USBF initialization processing**.)

5.2.3 Descriptor information

Descriptor information is defined in the `usbf78k_desc.h` file. (For details, see **3.1.5 Descriptor settings**.) To specify the attributes of the target device by using the sample software, rewrite the values in this file according to the application to actually use.

If the vendor ID and product ID of the device descriptor are rewritten, the vendor ID and product ID must also be rewritten in the host driver to install when connecting the target device (the INF file). (For details, see **5.2.4 (3) Changing the vendor and product IDs**.)

Changing the vendor and product IDs.)

Any information can be specified for the string descriptor, so rewrite it as required.

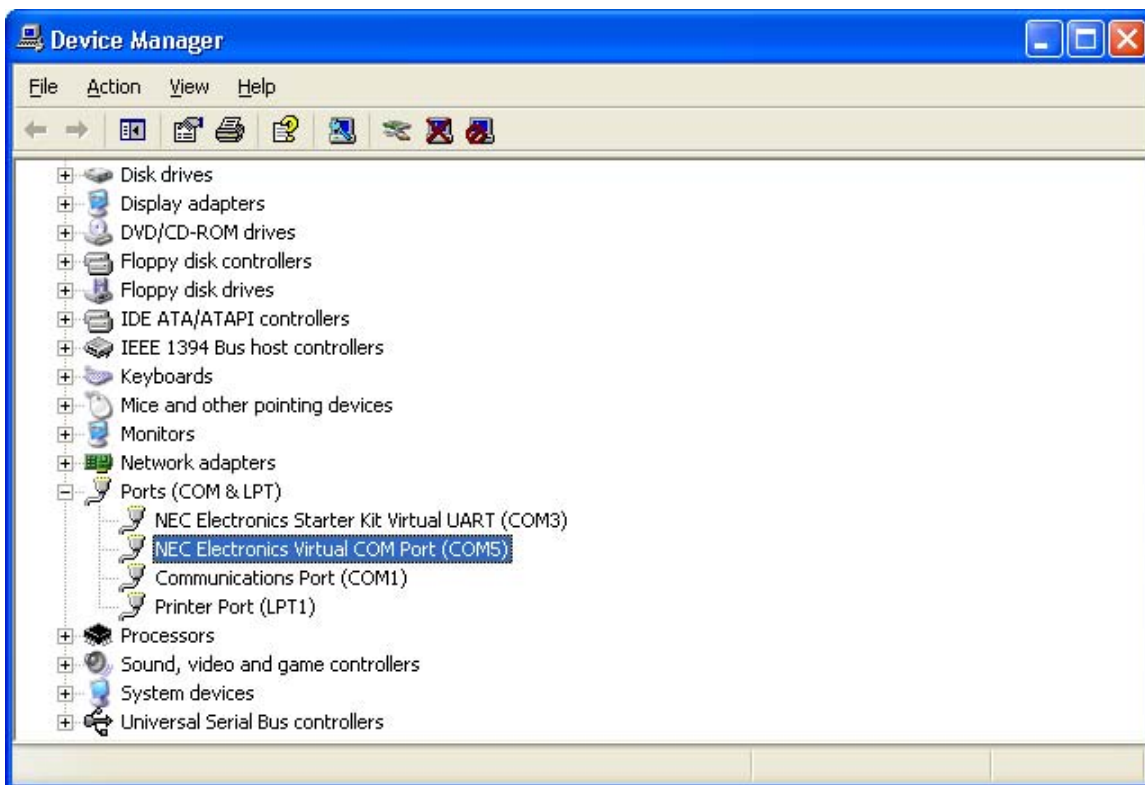
5.2.4 Setting up the virtual COM port host driver

The USB port (virtual COM port) driver can be customized as described below.

(1) Changing the COM port number

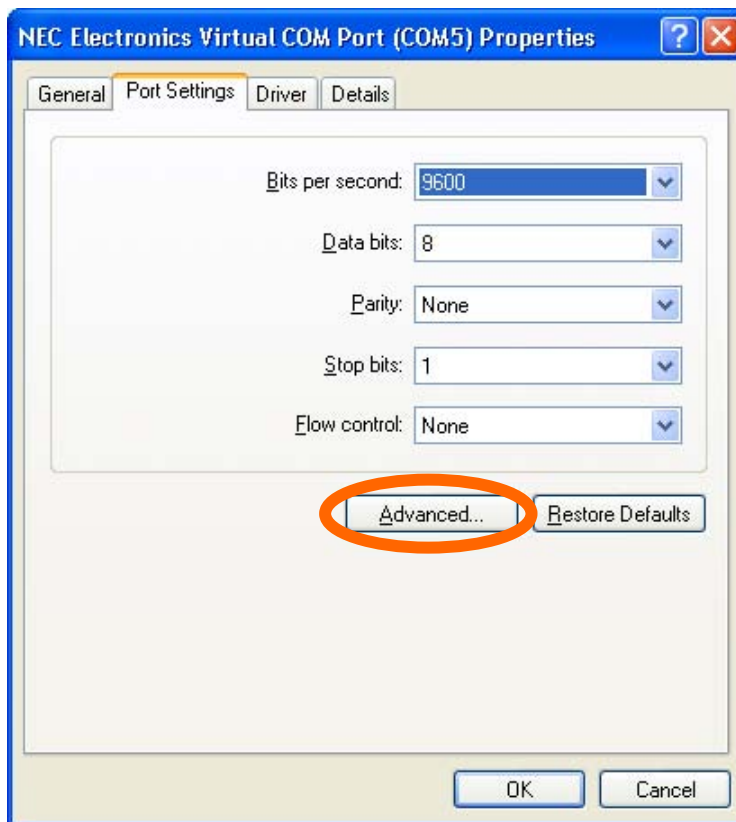
When the connection of a USB device is recognized by the host, the host automatically assigns the COM port number of the device, but the number can be changed to any number. To change the COM port number by using the host, perform the following procedure:

<1> Open the Windows Device Manager and display the items in the **Ports (COM & LPT)** category.

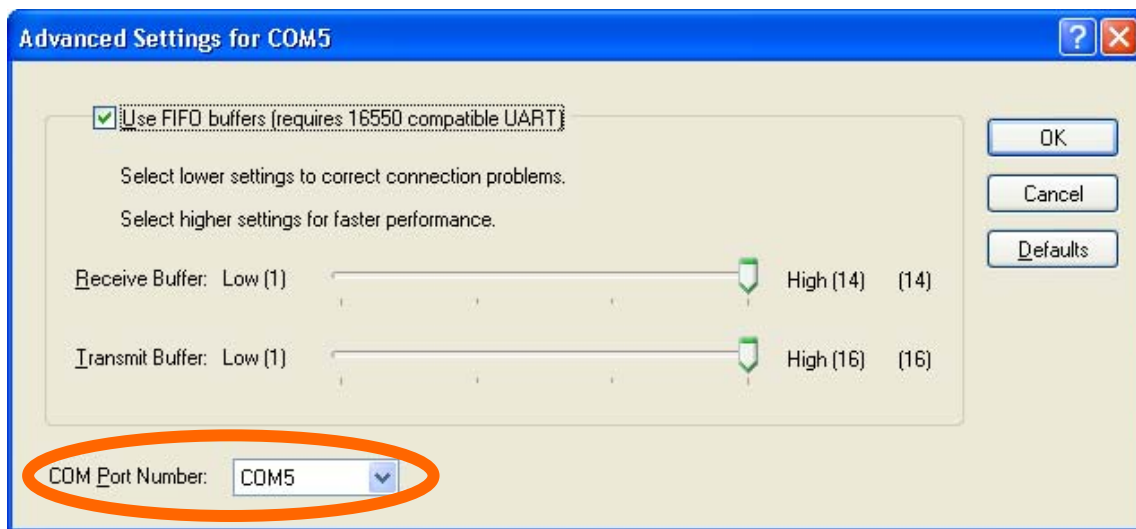


<2> Select **NEC Electronics Virtual COM Port (COMn)** (where n is a number assigned by the host) to display its properties.

<3> Click the **Advanced** button on the **Port Settings** tab.



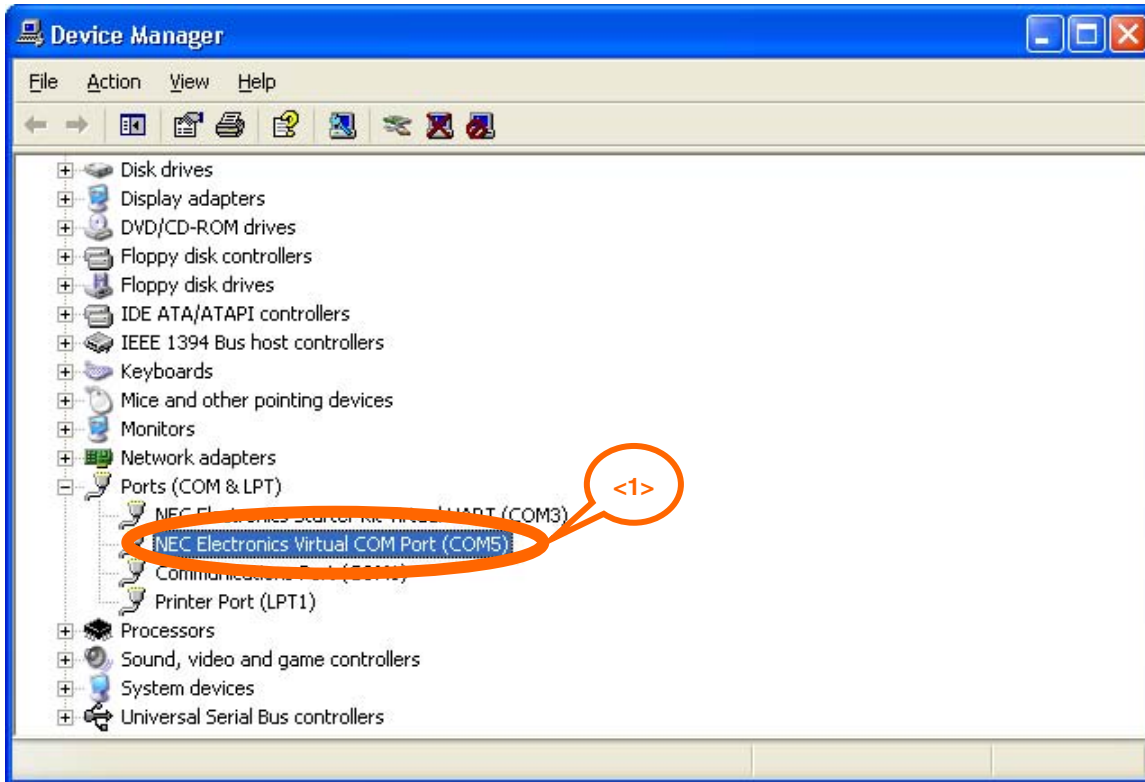
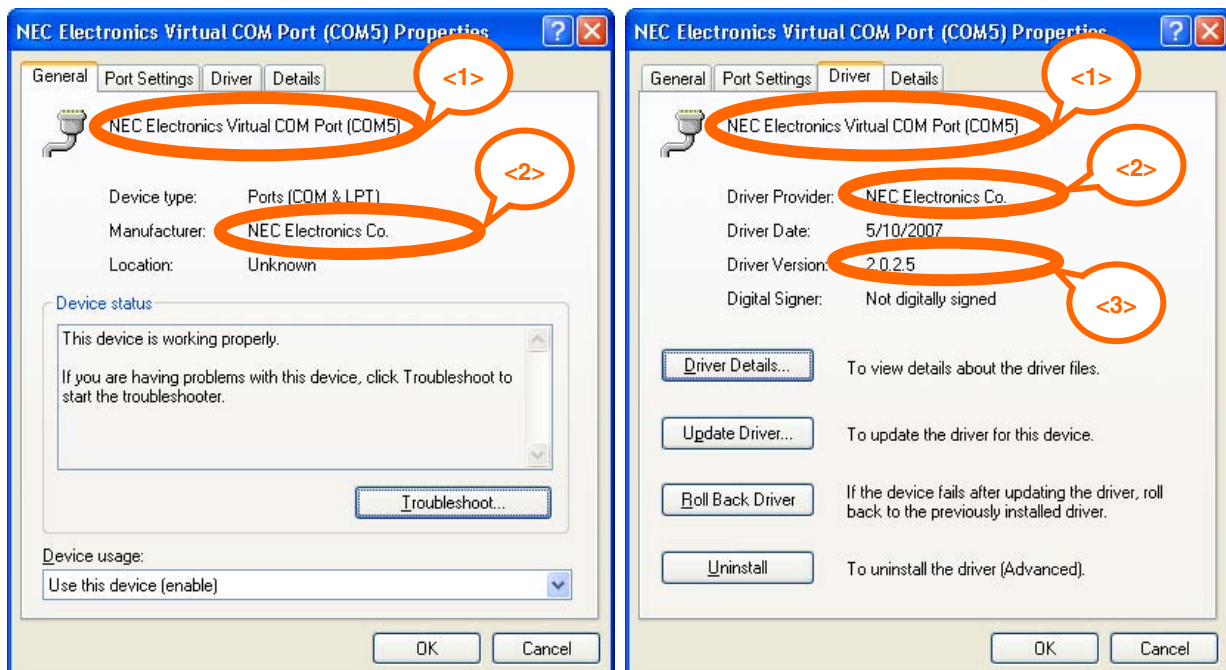
<4> In the **Advanced Settings for COMn** dialog box (where n is a number assigned by the host), select any port number from the drop-down list for **COM Port Number**.



- Remarks**
1. Make sure not to select a port number that is used for a different device.
 2. Immediately after applying this change, the new port number becomes valid but might not be reflected in the Device Manager.

(2) Changing properties

Some information, such as the attributes of the device used by the Windows Device Manager, can be changed. The information that can be changed is shown below.

(a) The device name (devices)**(b) The device name, manufacturer name, and version (device properties)**

Because this information is displayed based on the information included in the host driver (the INF file), it can be changed by rewriting the INF file. The sections in the INF file, which correspond to the numbers in the example on the previous page, are shown below.

List 5-2. INF File (necelusbvcom.inf) (1/2)

```

1  ;/+++
2  ;
3  ;Copyright (c) NEC Electronics co. All rights Reserved
4  ;
5  ;Module Name:
6  ;
7  ;    NECELUSBVCOM.INF
8  ;
9  ;Abstract:
10 ;    NEC Electronics Virtual COM Port Driver for Windows 2000/Xp
11 ;
12 ;--*/
13 [Version]
14 Signature="$Windows NT$"
15 Class=Ports
16 ClassGuid = {4d36e978-e325-11ce-bfc1-08002be10318}
17 Provider=%MfgName%
18 DriverVer=05/10/2007,2.0.2.5
19
20 [SourceDisksNames]
21 1=%disk1.desc%
22
23 [SourceDisksFiles]
24 NECELUSBDV.sys=1
25 NECELVCOM.sys=1
26
27 [Manufacturer]
28 %MfgName%=SECTION_0
29
30 [SECTION_0]
31 %USB\VID_0409&PID_01CD.DeviceDesc%=NECELUSBDV_V2.Dev, USB\VID_0409&PID_01CD
32
33 [DestinationDirs]
34 NECELUSBDV_COPYFILES = 10,System32\Drivers
35
36 [NECELUSBDV_V2.Dev.NT]
37 CopyFiles=NECELUSBDV_COPYFILES
38 AddReg=NECELUSBDV.AddReg
39
40 [NECELUSBDV.AddReg]
41 HKR,,PortSubClass,1,01
42 HKR,,EnumPropPages32,, "MsPorts.dll,SerialPortPropPageProvider"
43
44 [NECELUSBDV_COPYFILES]
45 NECELUSBDV.sys
46 NECELVCOM.sys
47
48 [NECELUSBDV_V2.Dev.NT.HW]
49 AddReg=NECELUSBDV_Common.Dev.NT.HW.AddReg, NECELUSBDV_V2.Dev.NT.HW.AddReg
50
51 [NECELUSBDV_Common.Dev.NT.HW.AddReg]
52 HKR,,"UpperFilters",0x00010000,"NECELVCOM_FILTER"
53 HKR,,"EndPointCaps",0x00010001,4
54 HKR,,"DebugLevel",0x00010001,3

```

<3>

List 5-2. INF File (necelusbvcom.inf) (2/2)

```

55 HKR,, "RawDump", 0x00010001, 0
56
57 [NECEUSBBDV_V2.Dev.NT.HW.AddReg]
58 HKR,, "CtlPollPeriod", 0x00010001, 0
59 HKR,, "RxPollPeriod", 0x00010001, 0
60 HKR,, "RxReqSize", 0x00010001, 512
61 HKR,, "EP1_SegmentSize", 0x00010001, 512
62
63 [NECEUSBBDV_V2.Dev.NT.Services]
64 Addservice = NECEUSBBDV, 2, NECEUSBBDV.AddService
65 Addservice = NECELVCOM_FILTER, , NECELVCOM_FILTER.AddService
66
67 [NECEUSBBDV.AddService]
68 DisplayName = %NECELVCOM_USB.SvcDesc%
69 ServiceType = 1
70 StartType = 3
71 ErrorControl = 1
72 ServiceBinary = %12%\NECEUSBBDV.sys
73 LoadOrderGroup = Base
74
75 [NECELVCOM_FILTER.AddService]
76 DisplayName = %NECELVCOM_FILTER.SvcDesc%
77 ServiceType = 1
78 StartType = 3
79 ErrorControl = 1
80 ServiceBinary = %12%\NECELVCOM.sys
81 LoadOrderGroup = PNP Filter
82
83 [Strings]
84 MfgName="NEC Electronics Co." <2>
85 disk1.desc="NEC Electronics virtual COM port driver install disk"
86 USB\VID_0409&PID_01CD.DeviceDesc="NEC Electronics Virtual COM Port" <1>
87 NECELVCOM_USB.SvcDesc="NECEL USBLIB"
88 NECELVCOM_FILTER.SvcDesc="Virtual COM Port for NECEL USB"

```

(3) Changing the vendor and product IDs

If the vendor and product IDs in the device descriptor are changed, the same changes must be specified in the host driver (the INF file).

Include the vendor and product IDs in the INF file in the following format on line 31 in List 5-2.

Vendor ID: Represented by four digits in hexadecimal format following "VID_"

Product ID: Represented by four digits in hexadecimal format following "PID_"

APPENDIX A TARGET BOARD

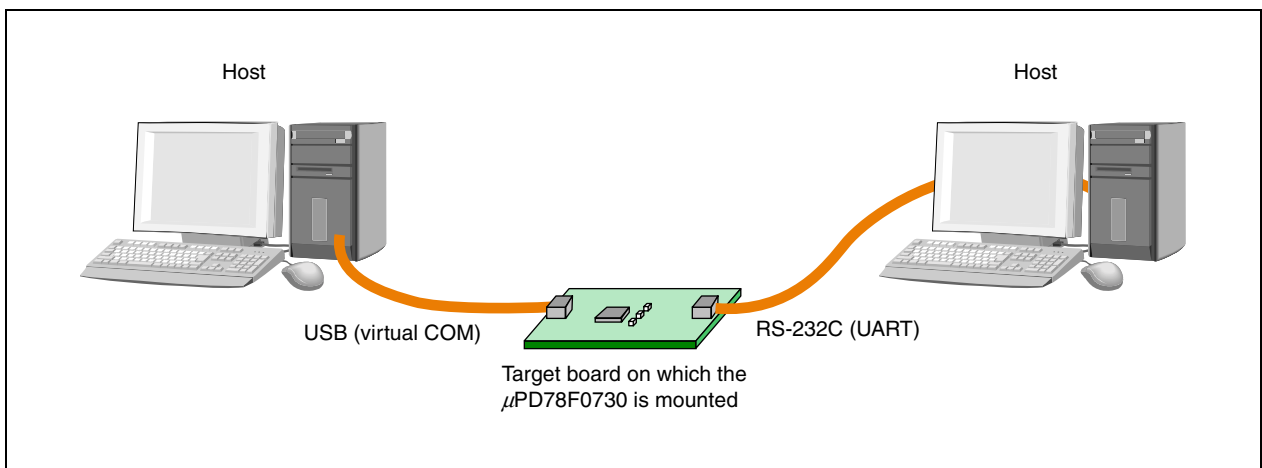
This chapter describes the target board.

A.1 Overview

This sample software transmits the data received by the USB function controller directly from the UART or transmits the data received by the UART directly from the USB function controller.

To perform these operations, a target board that has a USB port and a UART port is required. The UART I/O signals of the μ PD78F0730 must be connected to the host by converting the signal level and using the RS-232C interface, because the UART I/O signals cannot be directly connected to the host.

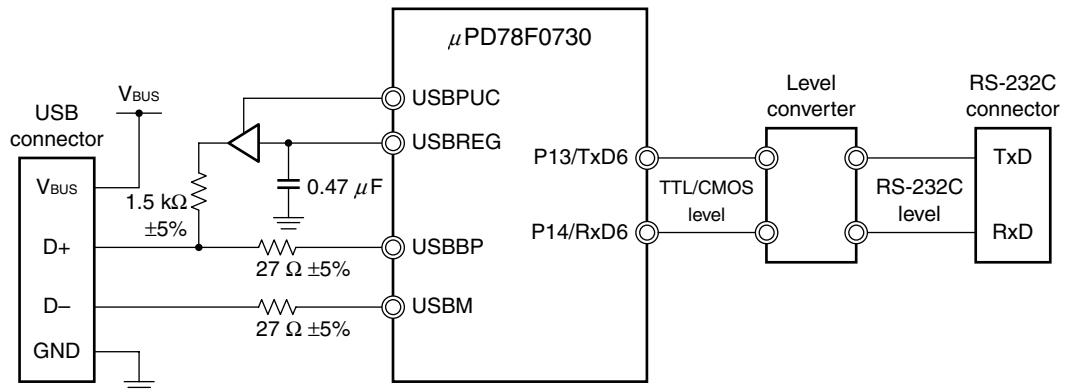
Figure A-1. Target Board Connections



A.2 Circuit Example

A circuit configuration example of the USB and RS-232C sections of a target board is shown below.

Figure A-2. Circuit Configuration Example of the USB and RS-232C Sections



Caution The circuit example and circuit constants are only examples. When actually creating a target board, determine the circuit and circuit constants based on thorough evaluation.

*For further information,
please contact:*

NEC Electronics Corporation
1753, Shimonumabe, Nakahara-ku,
Kawasaki, Kanagawa 211-8668,
Japan
Tel: 044-435-5111
<http://www.necel.com/>

[America]

NEC Electronics America, Inc.
2880 Scott Blvd.
Santa Clara, CA 95050-2554, U.S.A.
Tel: 408-588-6000
800-366-9782
<http://www.am.necel.com/>

[Europe]

NEC Electronics (Europe) GmbH
Arcadiastrasse 10
40472 Düsseldorf, Germany
Tel: 0211-65030
<http://www.eu.necel.com/>

Hanover Office
Podbielskistrasse 166 B
30177 Hannover
Tel: 0 511 33 40 2-0

Munich Office
Werner-Eckert-Strasse 9
81829 München
Tel: 0 89 92 10 03-0

Stuttgart Office
Industriestrasse 3
70565 Stuttgart
Tel: 0 711 99 01 0-0

United Kingdom Branch
Cygnus House, Sunrise Parkway
Linford Wood, Milton Keynes
MK14 6NP, U.K.
Tel: 01908-691-133

Succursale Française
9, rue Paul Dautier, B.P. 52
78142 Velizy-Villacoublay Cédex
France
Tel: 01-3067-5800

Sucursal en España
Juan Esplandiú, 15
28007 Madrid, Spain
Tel: 091-504-2787

Tyskland Filial
Täby Centrum
Entrance S (7th floor)
18322 Täby, Sweden
Tel: 08 638 72 00

Filiale Italiana
Via Fabio Filzi, 25/A
20124 Milano, Italy
Tel: 02-667541

Branch The Netherlands
Steijgerweg 6
5616 HS Eindhoven
The Netherlands
Tel: 040 265 40 10

[Asia & Oceania]

NEC Electronics (China) Co., Ltd
7th Floor, Quantum Plaza, No. 27 ZhiChunLu Haidian
District, Beijing 100083, P.R.China
Tel: 010-8235-1155
<http://www.cn.necel.com/>

Shanghai Branch
Room 2509-2510, Bank of China Tower,
200 Yincheng Road Central,
Pudong New Area, Shanghai, P.R.China P.C:200120
Tel:021-5888-5400
<http://www.cn.necel.com/>

Shenzhen Branch
Unit 01, 39/F, Excellence Times Square Building,
No. 4068 Yi Tian Road, Futian District, Shenzhen,
P.R.China P.C:518048
Tel:0755-8282-9800
<http://www.cn.necel.com/>

NEC Electronics Hong Kong Ltd.
Unit 1601-1613, 16/F., Tower 2, Grand Century Place,
193 Prince Edward Road West, Mongkok, Kowloon, Hong Kong
Tel: 2886-9318
<http://www.hk.necel.com/>

NEC Electronics Taiwan Ltd.
7F, No. 363 Fu Shing North Road
Taipei, Taiwan, R. O. C.
Tel: 02-8175-9600
<http://www.tw.necel.com/>

NEC Electronics Singapore Pte. Ltd.
238A Thomson Road,
#12-08 Novena Square,
Singapore 307684
Tel: 6253-8311
<http://www.sg.necel.com/>

NEC Electronics Korea Ltd.
11F., Samik Lavied'or Bldg., 720-2,
Yeoksam-Dong, Kangnam-Ku,
Seoul, 135-080, Korea
Tel: 02-558-3737
<http://www.kr.necel.com/>