NEC

Application Note

RX77016

Real-Time Operating System

HOST API

Target devices

- μPD77016
- μPD77017
- μPD77018
- μPD77018A
- μPD77019
- μPD77110
- μPD77111
- μPD77112
- μPD77113
- μPD77114

Document No. U14371EJ1V0ANJ1 (1st edition)
Date Published October 2000 N CP(K)

© NEC Corporation 1999

Printed in Japan

[MEMO]

Windows is either a registered trademark or a trademark of Microsoft Corporation in the United States and/or other countries.

- **The information in this document is current as of October, 1999. The information is subject to change without notice. For actual design-in, refer to the latest publications of NEC's data sheets or data books, etc., for the most up-to-date specifications of NEC semiconductor products. Not all products and/or types are available in every country. Please check with an NEC 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 prior written consent of NEC. NEC assumes no responsibility for any errors that may appear in this document.
- NEC 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 semiconductor 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 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 customer's equipment shall be done under the full responsibility of customer. NEC assumes no responsibility for any losses incurred by customers or third parties arising from the use of these circuits, software and information.
- While NEC endeavours to enhance the quality, reliability and safety of NEC semiconductor 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 semiconductor products, customers must incorporate sufficient safety measures in their design, such as redundancy, fire-containment, and anti-failure features.
- NEC semiconductor products are classified into the following three quality grades:
  "Standard", "Special" and "Specific". The "Specific" quality grade applies only to semiconductor products developed based on a customer-designated "quality assurance program" for a specific application. The recommended applications of a semiconductor product depend on its quality grade, as indicated below. Customers must check the quality grade of each semiconductor 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 semiconductor products is "Standard" unless otherwise expressly specified in NEC's data sheets or data books, etc. If customers wish to use NEC semiconductor products in applications not intended by NEC, they must contact an NEC sales representative in advance to determine NEC's willingness to support a given application.
  (Note)
  (1) "NEC" as used in this statement means NEC Corporation and also includes its majority-owned subsidiaries.
  (2) "NEC semiconductor products" means any semiconductor product developed or manufactured by or for NEC (as defined above).

M8E 00.4

Regional Information

Some information contained in this document may vary from country to country. Before using any NEC product in your application, please contact the NEC office in your country to obtain a list of authorized representatives and distributors. They will verify:

- Device availability
- Ordering information
- Product release schedule
- Availability of related technical literature
- Development environment specifications (for example, specifications for third-party tools and components, host computers, power plugs, AC supply voltages, and so forth)
- Network requirements

In addition, trademarks, registered trademarks, export restrictions, and other legal issues may also vary from country to country.

NEC Electronics Inc. (U.S.)
Santa Clara, California
Tel: 408-588-6000
800-366-9782
Fax: 408-588-6130
800-729-9288

NEC Electronics (Germany) GmbH
Duesseldorf, Germany
Tel: 0211-65 03 02
Fax: 0211-65 03 490

NEC Electronics (UK) Ltd.
Milton Keynes, UK
Tel: 01908-691-133
Fax: 01908-670-290

NEC Electronics Italiana s.r.l.
Milano, Italy
Tel: 02-66 75 41
Fax: 02-66 75 42 99

NEC Electronics (Germany) GmbH
Benelux Office
Eindhoven, The Netherlands
Tel: 040-2445845
Fax: 040-2444580

NEC Electronics (France) S.A.
Velizy-Villacoublay, France
Tel: 01-30-67 58 00
Fax: 01-30-67 58 99

NEC Electronics (France) S.A.
Madrid Office
Madrid, Spain
Tel: 91-504-2787
Fax: 91-504-2860

NEC Electronics (Germany) GmbH
Scandinavia Office
Taeby, Sweden
Tel: 08-63 80 820
Fax: 08-63 80 388

NEC Electronics Hong Kong Ltd.
Hong Kong
Tel: 2886-9318
Fax: 2886-9022/9044

NEC Electronics Hong Kong Ltd.
Seoul Branch
Seoul, Korea
Tel: 02-528-0303
Fax: 02-528-4411

NEC Electronics Singapore Pte. Ltd.
United Square, Singapore
Tel: 65-253-8311
Fax: 65-250-3583

NEC Electronics Taiwan Ltd.
Taipei, Taiwan
Tel: 02-2719-2377
Fax: 02-2719-5951

NEC do Brasil S.A.
Electron Devices Division
Guarulhos-SP Brasil
Tel: 55-11-6462-6810
Fax: 55-11-6462-6829

J00.7

INTRODUCTION

Target Readers

This application note is intended for users who wish to understand the functions of the μPD77016 Family and to design application programs using these microcontrollers.

The μPD77016 Family comprises the μPD77016, 77017, 77018, 77018A, 77019, 77110, 77111, 77112, 77113, and 77114. Unless otherwise specified, the μPD77016 is treated as the representative product in this document in describing functions that are shared by all the above-listed products.

Purpose

This application note is intended to give users an understanding of HOST API, which is an API (Application Program Interface) that supports data communication between a μPD77016 Family product that mounts the RX77016 and the host processor. The configuration of the loaded program is indicated for illustration purposes only, and is not to be used for mass-production design.

Organization

This application note describes the configuration and mechanism of HOST API, as well as the API functions and BIOS functions.

How to Read This Manual

It is assumed that the reader of this manual has general knowledge in the fields of logic circuits, microcontrollers, C language, and Windows™.

To learn about the functions of the RX77016:
→ Refer to the **RX77016 User's Manual Functions**.

To learn how to use the RX77016 Configuration Tool:
→ Refer to the **RX77016 User's Manual Configuration Tool**.

To learn about the hardware functions of the μPD77016 Family:
→ Refer to the **μPD7701X Family User's Manual Architecture**.

To learn about the instruction functions of the μPD77016 Family
→ Refer to the **μPD7701X Family User's Manual Instructions**.

Conventions

Data Significance:	Higher digits on the left and lower digits on the right
Active low representation:	$\overline{\times\times\times}$ (overscore over pin or signal name)
Note:	Footnote for item marked with Note in the text
Caution:	Information requiring particular attention
Remark:	Supplementary information
Numerical representation:	Binary... ×××× or 0b××××
Decimal:	××××
Hexadecimal:	0x××××

Expression Format

In this application note, API functions are expressed by appending "()" to indicate the fact that they are C language function, and BIOS functions are expressed by appending ":" to indicate that they are μPD77016 assembly language level.
Example API function "OpenDEV" → "OpenDEV()"
BIOS function "EchoWord" → "EchoWord:"

In HOST API for RX77016, the μPD77016 Family is assumed as the digital signal processor (DSP) on the target system, but in this application note, the digital signal processor is simply described as the DSP.

In this application note, the July 1999 version of HOST API for RX77016 is described as the current version.

*××× indicates the memory contents having ××× as their address.
Example If the contents of the DP0 register are 0x1000, *DP0 indicates the memory contents of address 0×1000.

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

- **Documents Related to Devices**

Document Name / Product Name	Pamphlet	Data Sheet	User's Manual		Application Note
			Architecture	Instructions	Basic Software
μPD77016	U12395E	U10891E	U10503E	U13116E	U11958E
μPD77017		U10902E			
μPD77018					
μPD77018A		U11849E			
μPD77019					
μPD77019-013		U13053E			
μPD77110		U12801E	To be prepared		
μPD77111					
μPD77112					
μPD77113		U14373E			
μPD77114					

- **Documents Related to Development Tools**

Document Name		Document Number
IE-77016-98/PC User's Manual	Hardware	EEU-1541
IE-77016-CM-LC User's Manual		U14139E
RX77016 User's Manual	Functions	U14397E
	Configuration Tool	U14371E
RX77016 Application Note	HOST API	This manual

Caution **The above documents are subject to change without prior notice. Be sure to use the latest version document when starting design.**

[MEMO]

CONTENTS

CHAPTER 1 OVERVIEW 13
- **1.1 Outline of API Functions** 13
- **1.2 Outline of BIOS Functions** 14

CHAPTER 2 HOST API CONFIGURATION 15
- **2.1 Software Configuration** 15
- **2.2 Hardware Configuration** 17

CHAPTER 3 HOST API FUNCTIONS 19
- **3.1 Opening and Closing Communication Path** 19
- **3.2 Data Setting to OS Communication Area** 20
- **3.3 Reboot** 21
- **3.4 Data Memory Access** 22
- **3.5 Communication Buffer** 23
  - 3.5.1 Securing and releasing communication buffer area 24
  - 3.5.2 Communication buffer access 26
- **3.6 Task Management** 28

CHAPTER 4 INPUT FILES 29
- **4.1 Information Files** 29
  - 4.1.1 Device section 29
  - 4.1.2 IO RESOURCE section 29
  - 4.1.3 Task section 32
  - 4.1.4 TaskXX section 32
  - 4.1.5 SubTaskXXYY section 32
  - 4.1.6 Information file description example 33
- **4.2 HEX File** 34

CHAPTER 5 HOST API SYSTEM 35
- **5.1 BIOS Mode** 35
- **5.2 BIOS Function Call** 36

CHAPTER 6 API FUNCTIONS 37
- **6.1 Data Definition** 37
- **6.2 API Application Functions** 39
  - 6.2.1 Initial setting functions 39
  - 6.2.2 Task management functions 44
  - 6.2.3 Memory access functions 47
  - 6.2.4 HOST API management functions 64

CHAPTER 7 BIOS FUNCTIONS 67
- **7.1. Action upon Occurrence of HI Interrupt** 67
- **7.2. BIOS Functions** 67

CHAPTER 8 PREPARATION OF HOST API EXECUTION FILE....... 77
8.1 Preparation of Execution File Using API Functions 77
8.1.1 hapiucfg.h....... 77
8.1.2 spx.c....... 78
8.2 Preparation of Execution File Using BIOS Functions 79
8.2.1 os_udef.h 79
8.2.2 Target system initialization block (_MOS_TargetSysInit:)....... 79
8.2.3 HI interrupt handler code block (_MOS_ivHI:)....... 79
8.2.4 biosucfg.h....... 80

LIST OF FIGURES

Figure No.	Title	Page
2-1	Software Configuration Image	15
2-2	Hardware Configuration Image	17
2-3	Byte Access Image	18
3-1	Image of Communication Path Opening and Closing (Support of Multiple Devices)	19
3-2	Image of Data Setting to OS Communication Area	20
3-3	Reboot Image	21
3-4	Data Memory Access Image	22
3-5	Communication Buffer Image	23
3-6	Image of Securing Communication Buffer Area	24
3-7	Image of Releasing Usage Area	25
3-8	Image of Communication Buffer Access	26
3-9	Task Management Image	28
4-1	Information File Description Example	33
4-2	Image of Reeboot Using Extension Intel HEX File	34
5-1	Change in BIOS Mode When Calling OpenDEV(), CloseDEV()	35
5-2	BIOS Kernel Operation Flow	36
6-1	Read Communication Buffer Read Destination Address	53
6-2	Write Communication Buffer Write Destination Address	55

LIST OF TABLES

Figure No.	Title	Page
3-1	List of Communication Buffer Access States	27
6-1	Data Types	37
6-2	Character Constants	38
7-1	Shift Amount and Jump Destination	68
7-2	*_MHA_PCmdPtr:MEM Contents and Jump Destination	69
7-3	Call Addresses and Reboot Routines	70

CHAPTER 1 OVERVIEW

HOST API is a program interface provided for user applications for performing data communication between the μPD77016 Family and the host processor[Note]. Data communications for all user applications are performed via HOST API.

Note In the current version of HOST API for RX77016, a personal computer (PC) is assumed as the host processor.

1.1 Outline of API Functions

The API functions provided by the HOST API system are outlined below. For more details about API functions, refer to **CHAPTER 6 API FUNCTIONS**.

(1) Initial setting functions

The following initial setting functions are provided for target devices.

OpenDEV() ... Opens communication path
CloseDEV() ... Closes communication path
RebootDEV() ... Performs host boot/self boot
ResetDEV() ... Resets system
SetCommunicateValue() ... Sets data to OS communication area

(2) Task management functions

The following task management functions are provided for specific tasks of target devices.

ShowTask() ... Outputs task information
ResumeSyncTask() ... Changes value of frame counter A
SuspendSyncTask() ... Clears value of frame counter A to 0

(3) Memory access functions

The following memory access functions are provided for specific data memory or communication buffers of target devices (refer to **3.5 Communication Buffers**.)

CreateCBuf() ... Secures communication buffer area
FreeCBuf() ... Releases communication buffer area
GetCBufCaps() ... Displays communication buffer information
InitCBuf() ... Clears communication buffer to 0
ReadFromTMem() ... Reads 1 data from X or Y memory
WriteToTMem() ... Writes 1 data to X or Y memory
ReadFromCBuf() ... Reads 1 data from communication buffer
WriteToCBuf() ... Writes 1 data to communication buffer
CopyTMemToHost() ... Copies from memory on host to X or Y memory
CopyHostToTMem() ... Copies from X or Y memory to memory on host
CopyCBufToHost() ... Copies from communication buffer to memory on host
CopyHostToCBuf() ... Copies from memory on host to communication buffer
CopyCBufToTMem() ... Copies from communication buffer to memory on host
CopyTMemToCBuf() ... Copies from memory on host to communication buffer

(4) HOST API management functions

The following HOST API management functions are provided for enabling smooth HOST API system use.

LoadInformationFile() ... Loads information file
ShowInfoContentsAll() ... Displays contents information
GetVersion() ... Displays version information
HexToBuf() ... Buffers HEX file

1.2 Outline of BIOS Functions

The HOST API system provides the following BIOS functions. For further information on BIOS functions, refer to **CHAPTER 7 BIOS FUNCTIONS**.

EchoWord: ... Action corresponding to HI interrupt upon failure to establish communication route
Interpreter: ... Specifies action corresponding to HI interrupt
SelPCmd: ... Specifies jump destination function address
PuttoCmdBuf: ... Jumps upon occurrence of HI interrupt the specified number of times
IDTagCmd: ... Jumps to specified user defined function
Reboot: ... Performs reboot according to called address
Direct_RX: ... Reads 1 data from X memory
Direct_RY: ... Reads 1 data from Y memory
Direct_WX: ... Writes 1 data to X memory
Direct_WY: ... Writes 1 data to Y memory
Copy_XtoX: ... Copies data in X memory
Copy_YtoY: ... Copies data in Y memory
Copy_XtoY: ... Copies data from X memory to Y memory
Copy_YtoX: ... Copies data from Y memory to X memory
ReadFromCBuf: ... Reads data from write communication buffer
WriteToCBuf: ... Writes data to read communication buffer

CHAPTER 2 HOST API CONFIGURATION

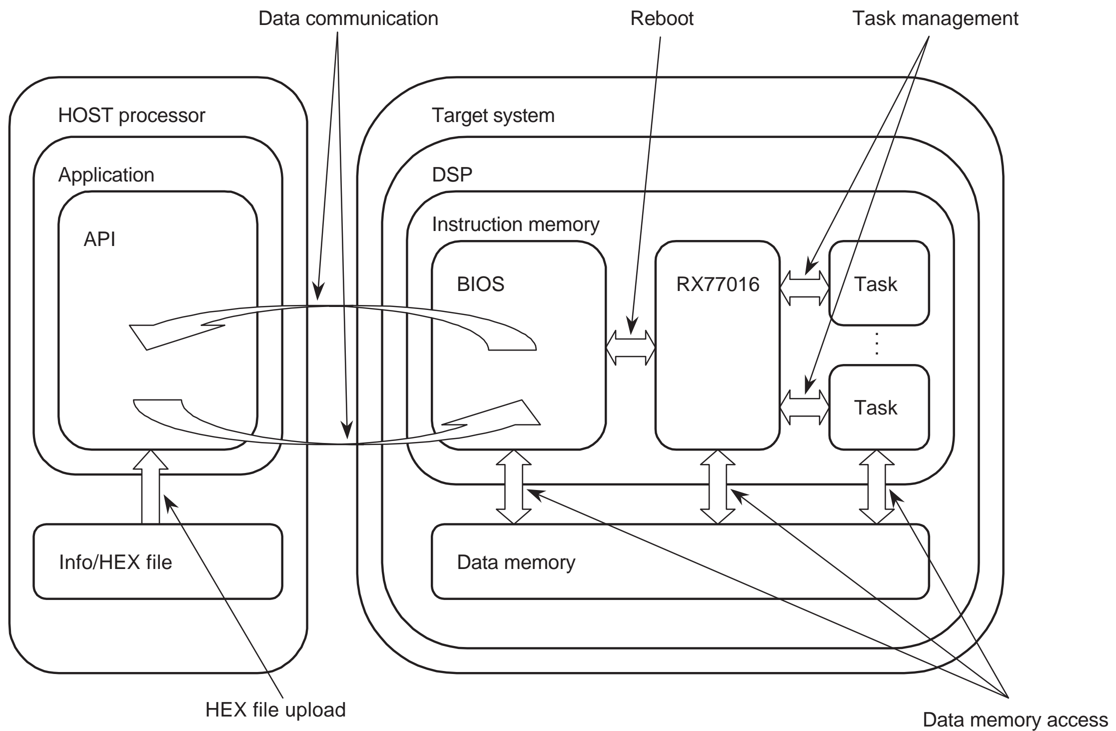
This chapter explains the configuration of the HOST API system.

2.1 Software Configuration

The HOST API system consists of API functions, which exist on the host processor and form the interface between the user application and the DSP, and BIOS functions, which exist on the DSP and consist of the actual HOST API functions. Reboot, data memory access, and task management, among other things, are performed by passing data between these two codes.

Figure 2-1 shows an image of the software configuration.

Figure 2-1. Software Configuration Image



Remark The host processor in the above figure is assumed to be a personal computer.

The API functions are provided as C language functions, and are used by issuing a call to the user application, which is the center of control. The API function for which a call is issued to the user application issues a command to the DSP to realize that function.

The BIOS functions are provided as μPD77016 Family assembly language functions. Which function is called is determined according to the result of decoding the commands issued by API functions.

In this application note, API functions are expressed by appending "()" to indicate the fact that they are C language function, and BIOS functions are expressed by appending ":" to indicate that they are uPD77016 assembly language level.

Example API function "OpenDEV" → "OpenDEV()"
BIOS function "EchoWord" → "EchoWord:"

(1) API functions

API functions consist of the following 6 files. The user application that calls an API function is compiled by the C compiler, and an execution file can be created by linking with the mos_hapi.c and spx.c object files (which must be compiled beforehand).

hostapi.h ... Header file required when a user application calls an API application
hapiucfg.h ... Header file for user definition of mos_hapi.c, spx.c
mos_hapi.h ... Header file of mos_hapi.c
mos_hapi.c ... Source file of API function
spx.h ... Header file of spx.c
spx.c ... Source file of interface block between API function and DSP

(2) BIOS functions

BIOS functions consist of the following 4 files. An executable file can be created by linking with the bios_fns.asm object file (which must be assembled beforehand) following insertion of the required code to the initialization block (_MOS_TargetSysInit:) for the target system of the RX77016 and the HI interrupt handler code block (_MOS_ivHI:) (refer to 8.2.3. HI interrupt Handler Code block (_MOS_ivHI:) and assembly in WB77016.

bios_fns.h ... Header file for target system initialization block (_MOS-TargetSysInit:) and HI interrupt handler code block (_MOS_ivHI:) of RX77016
bios_mac.h ... Macro definition file for target system initialization block (_MOS_TargetSysInit:) and bios_fns.asm of RX77016
biosucfg.h ... Header file for target system initialization block (_MOS_TargetSysInit:) and HI interrupt handler code block (_MOS_ivHI:) and bios_fns.asm of RX77016
bios_fns.asm ... Source file of BIOS function

2.2 Hardware Configuration

In the HOST API system, control is performed from the host processor (personal computer or microcontroller such as V85x) toward a DSP consisting of a single device.

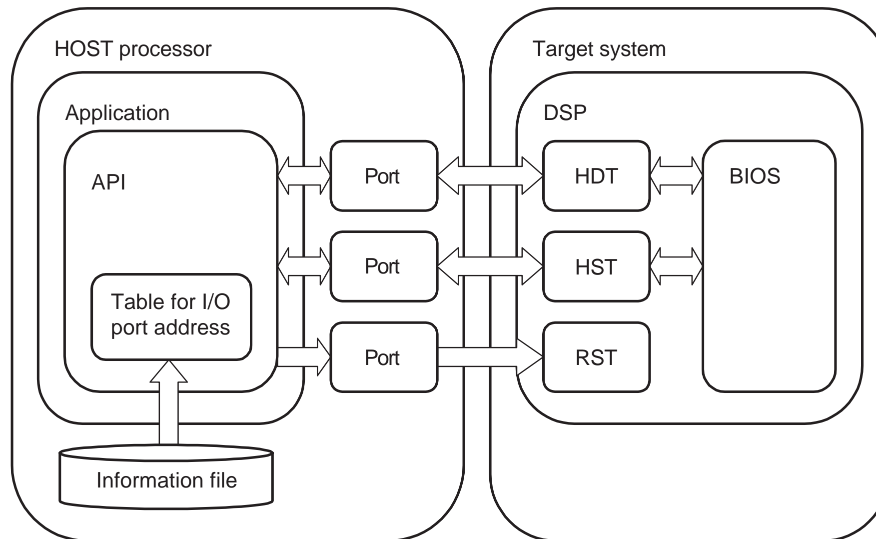
However, the HOST API system is designed to enable transition to a new version without changing the user program, when upgrading the version of the HOST API system while realizing multiple device support for RX77016 in the future.

Data communication is performed between the I/O port of the host processor and the host interface of the DSP[Note]. When the I/O port address that is used is described in the IO RESOURCES section of the information file, which is a HOST API information file (refer to **4.1 Information File**), it is acquired by the LoadInformationFile() and stored in a buffer. API functions that perform some kind of communication with the DSP, use this buffer information for data communication.

Figure 2-2 shows an image of the hardware configuration.

Note If the BIOS function of the HOST API is loaded to the DSP, other applications cannot use the HDT register, in order to acquire the host data register (HDT register). In HOST API, it is assumed that all data communication between the host processor and DSP are performed via the HOST API.

Figure 2-2. Hardware Configuration Image



Remark The host processor in the above figure is assumed to be a personal computer.

The host data register (HDT register) and host interface status register (HST register), which form the DSP host interface, can be accessed with either of two methods, byte access or word access[Note].

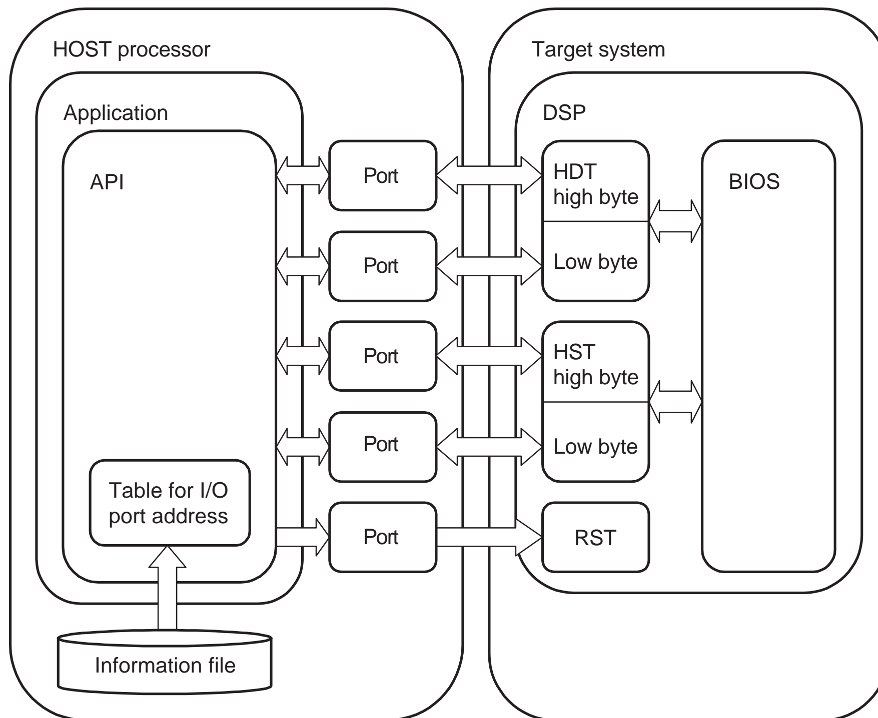
Byte access is done by dividing 16 bits (1 word) into higher 8 bits and lower 8 bits, and using a different I/O port for each to read/write data.

Word access is done through data read/write of 16 bits (1 word) using a single I/O port.

Figure 2-3 shows the image of byte access.

Note To switch the access method by condition compilation through definition of HDTAccess, HSTAccess, and RSTAccess of hapiucfg.h, a second compilation is required.

Figure 2-3. Byte Access Image



Remark The host processor in the above figure is assumed to be a personal computer.

CHAPTER 3 HOST API FUNCTIONS

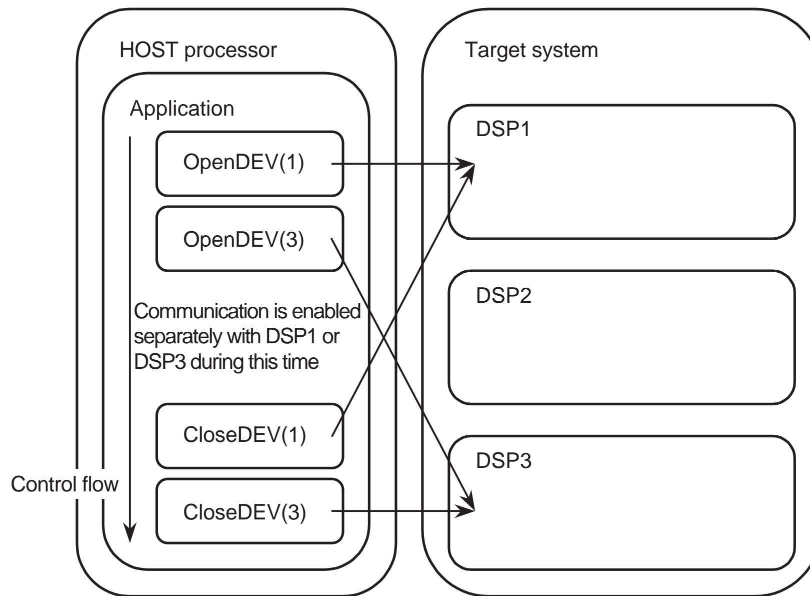
This chapter describes the functions of the HOST API system.

3.1 Opening and Closing Communication Path

In the HOST API system, data communication with the DSP can be performed until a communication path with a particular DSP is opened through OpenDEV() and then closed through CloseDEV(). OpenDEV() allocates a device handle corresponding to a particular DSP upon opening of a communication path. All the API functions use device handles to perform data communication with particular DSPs.

Figure 3-1 shows the image of communication path opening and closing when R77016 and HOST API support multiple devices.

Figure 3-1. Image of Communication Path Opening and Closing (Support of Multiple Devices)



Remark The host processor in the above figure is assumed to be a personal computer.

Caution Although this function will become truly valuable when the RX77016 supports multiple devices in the future, this function exists in the current version in order to minimize changes in user programs when this support is realized (This version of HOST API does not support multiple devices).

3.2 Data Setting to OS Communication Area

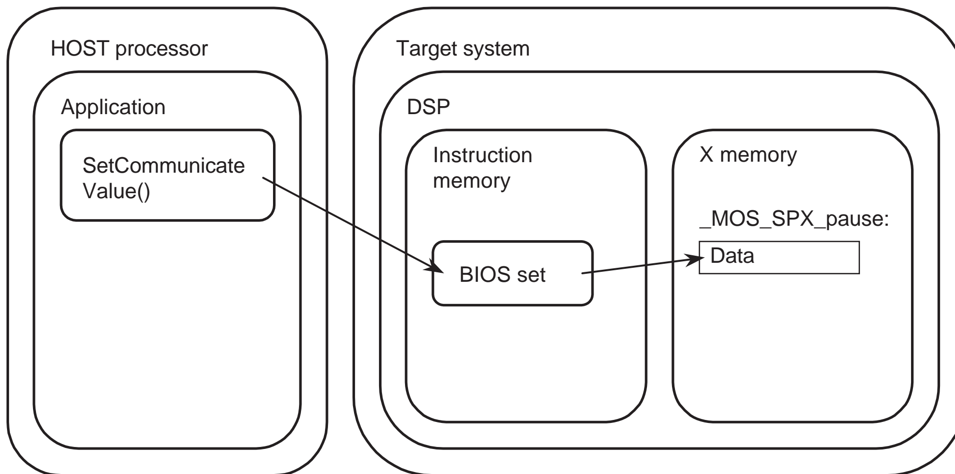
In the HOST API system, any desired data can be set to the RX77016's OS communication area (*_MOS_SPX_pause:X) using SetCommunicateValue(). Through this, it is possible to pause[Note] following the startup of the RX77016 system.

Data setting to the OS communication area is required when any kind of data communication is performed from external by a future RX77016 or individual applications.

Figure 3-2 shows the image of data setting to the OS communication area.

Note Pause following the startup of the RX77016 system refers to the state in which the execution of the next processing (application program start) is held until all the initialization processing of the OS are completed and an instruction is received from external (such as from the host processor). Pause is performed when *_MOS_SPX_pause:X is 0xffff, and when another value is written, pause is released. However, if a value other than 0xffff is written for the initial state, pause is not implemented and the next processing is performed.

Figure 3-2. Image of Data Setting to OS Communication Area



Remark The host processor in the above figure is assumed to be a personal computer.

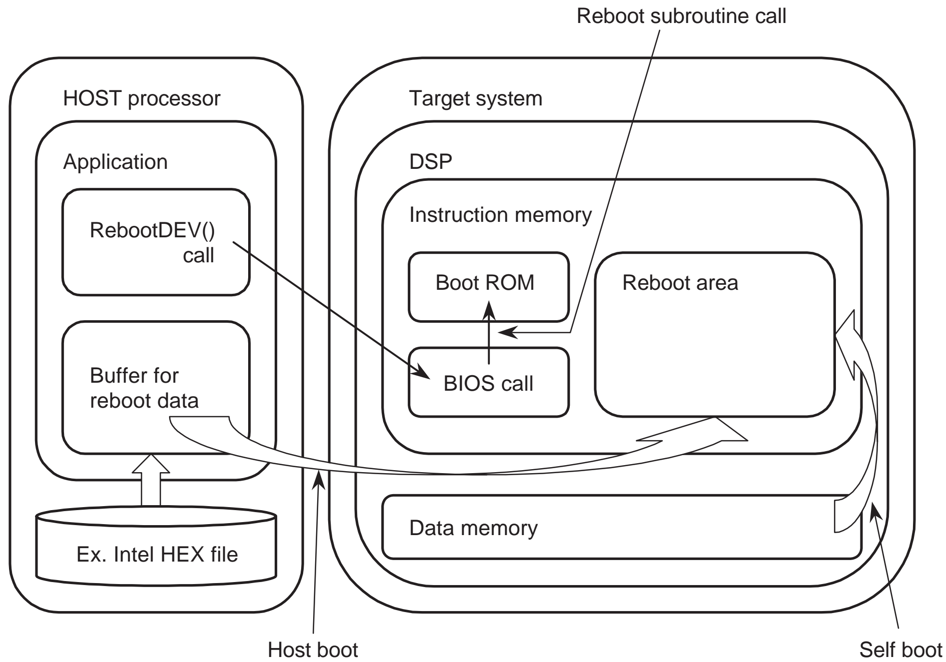
3.3 Reboot

In the HOST API system, it is possible to perform X-word reboot, X-byte reboot, Y-word reboot, Y byte reboot, and host reboot, using the reboot subroutine of the DSP reboot ROM with RebootDEV().

Moreover, it is also possible to use an extension Intel HEX file as the data source by using the HexToBuf() function for host reboot (however, this is possible only when a personal computer is used as the host processor).

Figure 3-3 shows the reboot image.

Figure 3-3. Reboot Image



Remark The host processor in the above figure is assumed to be a personal computer.

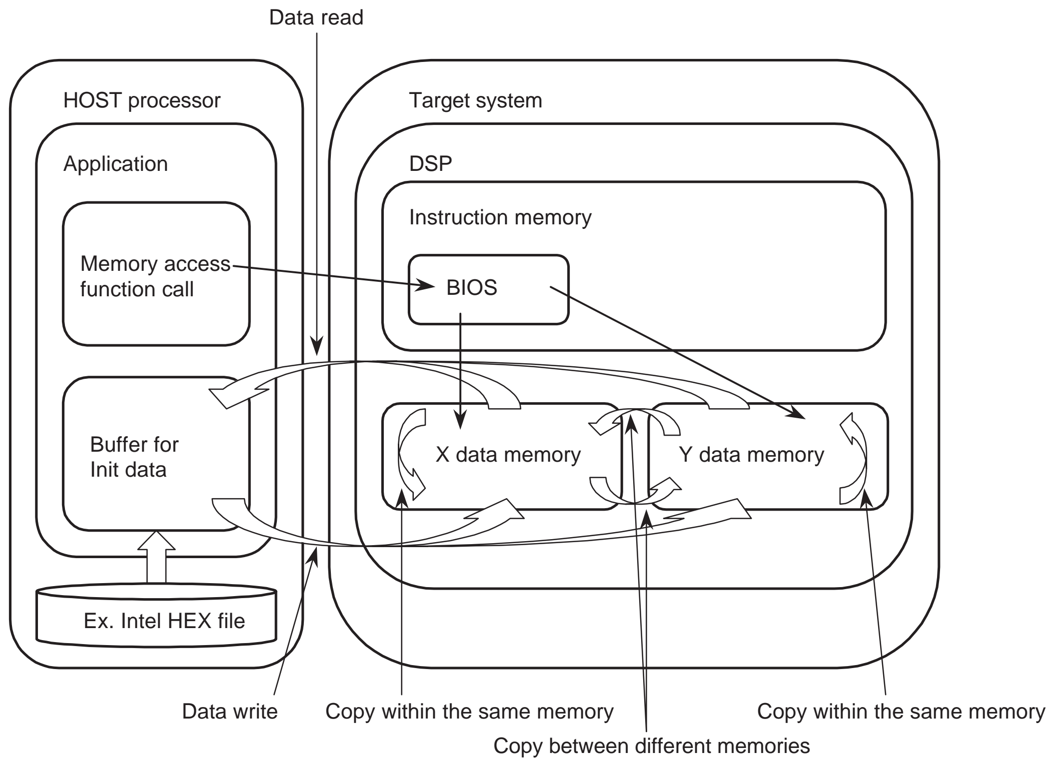
3.4 Data Memory Access

In the HOST API system, it is possible to read/write data to any address in data memory, copy data within the same memory, and copy data between different memories using ReadFromTMem(), WriteToMem(), CopyTMemToHost(), CopyHostToTMem(), or CopyTMemToTMem().

Moreover, with CopyHostToTMem(), it is also possible to use an extension Intel HEX file as the data source by using the HexToBuf() function (however, this is possible only when a personal computer is used as the host processor).

Figure 3-4 shows the data memory access image.

Figure 3-4. Data Memory Access Image



Remark The host processor in the above figure is assumed to be a personal computer.

3.5 Communication Buffer

In the HOST API system, it is possible to perform data communication between a user application and an application on the DSP by creating areas called communication buffers on the X or Y memory of the DSP.

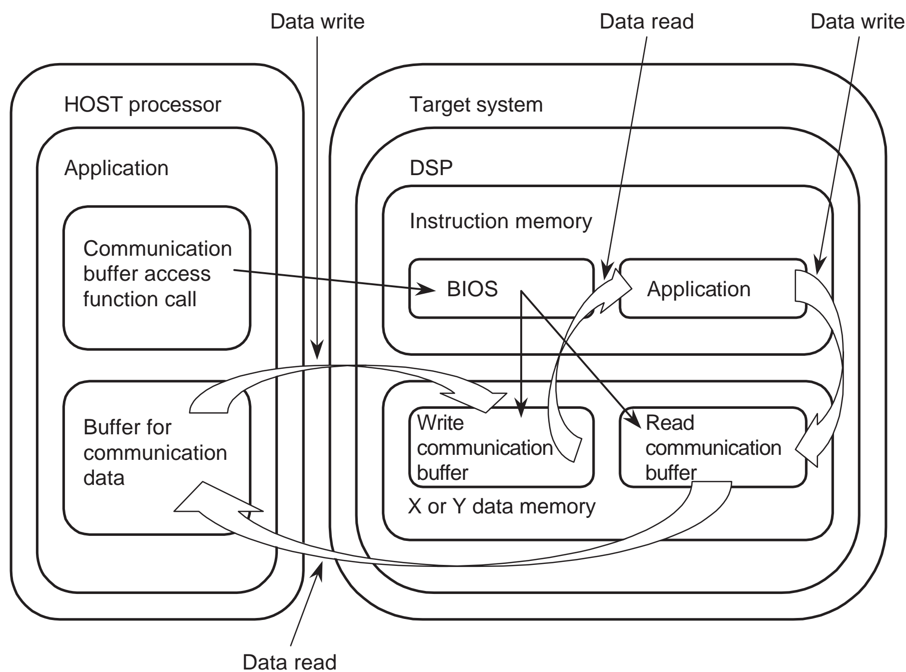
There are two types of communication buffers, a write communication buffer used to transfer data from the user application to the application on the DSP, and a read communication buffer used to transfer data from the application on the DSP to the user application. The size[Note] of these buffers can be freely set by the user.

Moreover, the communication buffers can perform data communication to any address in X or Y data memory.

Figure 3-5 shows the image of the communication buffers.

Note The communication buffer size is set by defining HICBUF_HEAP_SIZE and HOCBUF_HEAP_SIZE of biosucfg.h.

Figure 3-5. Communication Buffer Image



Remark The host processor in the above figure is assumed to be a personal computer.

3.5.1 Securing and releasing communication buffer area

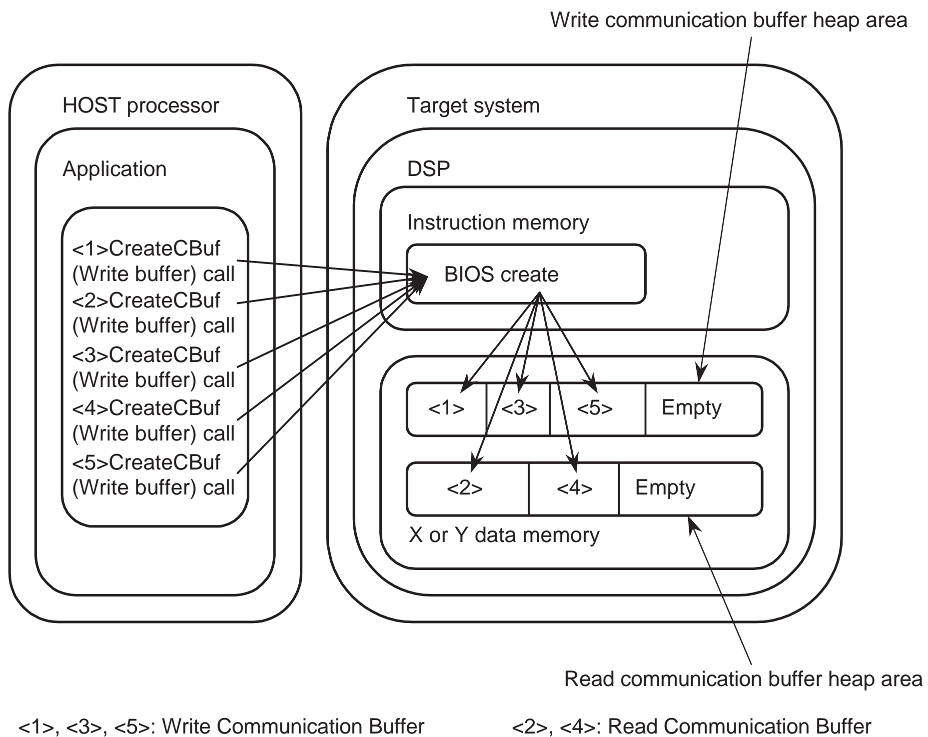
It is possible to secure and release the area to be used in small increments as required within previously secured[Note] heap areas (separate heap area for read and for write). Each area to be used is secured with CreateCBuf() and released with FreeCBuf().

The range can be freely set when securing the area to be used with CreateCBuf(), as long as the sum of the size already secured and the newly secured size does not exceed the remaining heap area size, and the number of already secured buffers + 1 does not exceed the planned number of buffers. Moreover, when securing an area, the handle corresponding to the area that is being secured is assigned. Later, this handle is used to access specific areas.

Figure 3-6 shows image of securing the communication buffer areas.

Note The required area is automatically secured by defining HICBUF_HEAP_SIZE, NUM_HICBUF, HOCBUF_HEAP_SIZE, and NUM_HOCBUF to perform assembly.

Figure 3-6. Image of Securing Communication Buffer Area



<1>, <3>, <5>: Write Communication Buffer   <2>, <4>: Read Communication Buffer

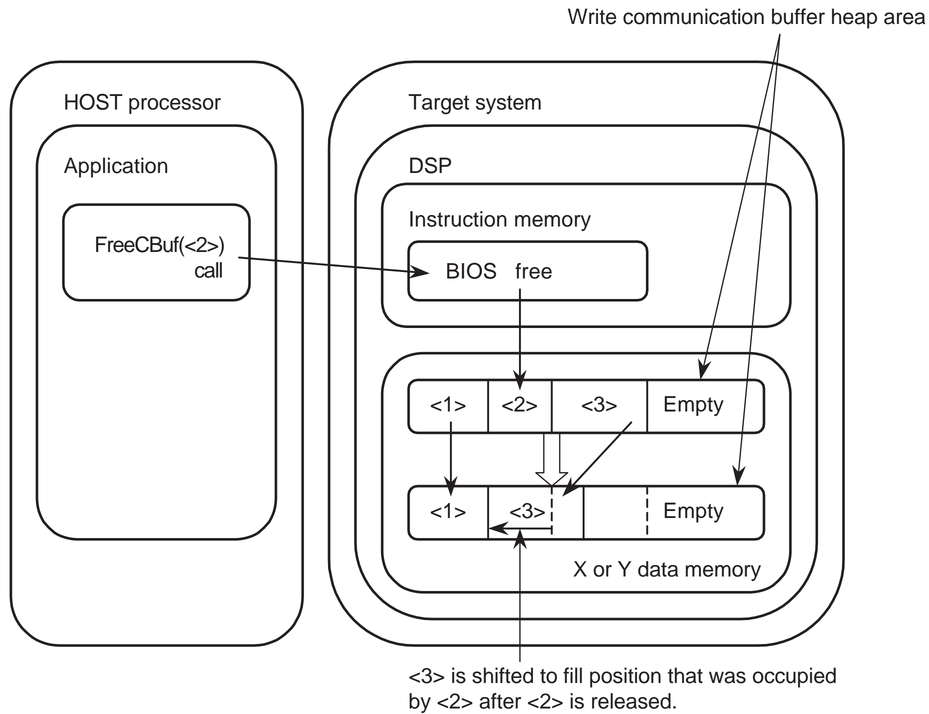
Remark The host processor in the above figure is assumed to be a personal computer.

When area release is performed with FreeCBuf(), the existing usage area is released and changes into a reusable area. Moreover, after the area is released, memory compaction is performed so that the usage area is placed continuously from the start of the heap area.

For example, when usage area <2> in Figure 3-7 is released, a blank results between <1> and <3>, so that the position of <3> is changed so as to fill the position that was occupied by <2>.

Figure 3-7 shows the image of releasing a usage area.

Figure 3-7. Image of Releasing Usage Area



Remark The host processor in the above figure is assumed to be a personal computer.

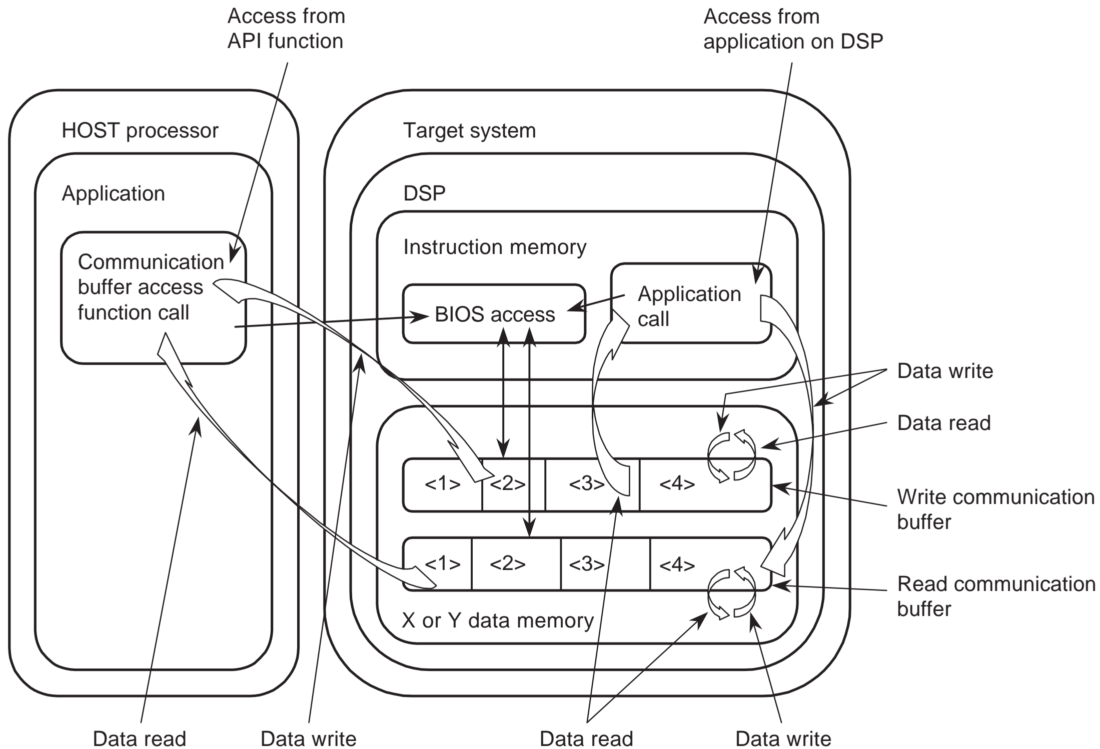
3.5.2 Communication buffer access

Communication buffers can be accessed in two ways: Access from an API function, and access from an application on the DSP.

In the case of access from an API application, one of InitCBuf(), ReadFromCBuf(), WriteToCBuf(), CopyCBufToHost(), CopyHostToCBuf(), CopyCBufToTMem(), or CopyTMemToCBuf() is called, and in the case of access from an application on the DSP, ReadFromCBuf: or WriteToCBuf: is called (for additional information on there functions, refer to **CHAPTER 6 API FUNCTIONS** and **CHAPTER 7 BIOS FUNCTIONS**).

Figure 3-8 shows the image of communication buffer access (in the case where <2> is current).

Figure 3-8. Image of Communication Buffer Access



Remark The host processor in the above figure is assumed to be a personal computer.

The basic access procedure for the write communication buffer is as follows.

<1> Secure the usage area with CreateCBuf().
<2> Write data to the usage area secured in step <1> with InitCBuf(), WriteToCBuf(), CopyHostToCBuf(), or CopyTMemToCBuf().
<3> Read the data of the usage area to which data has been written in step <2> with CopyCBufToTMem() or ReadFromCBuf:.

The basic access procedure for the read communication buffer is as follows.

<1> Secure the usage area with CreateCBuf().
<2>Write data to the usage area secured in step <1> with CopyTMemToCBuf() or WriteToCBuf():.
<3> Read the data of the usage area to which data has been written in step <2> with ReadFromCBuf(), CopyCBufToHost(), or CopyCBufToTMem().

However, due to the nature of the communication buffers, there may be accessible at times and not at other times. Communication buffer accessibility is summarized in the following table.

Table 3-1. List of Communication Buffer Access States

State of Usage Buffer / Function	Write Communication Buffer		Read Communication Buffer	
	Written Data is Read	Written Data is Not Read	Written Data is Read	Written Data is Not Read
InitCBuf()	Accessible	Not accessible	Access prohibited	Access prohibited
ReadFromCBuf()	Access prohibited	Access prohibited	Not accessible	Accessible
WriteToCBuf()	Accessible	Not accessible	Access prohibited	Access prohibited
CopyCBufToHost()	Access prohibited	Access prohibited	Not accessible	Accessible
CopyHostToCBuf()	Accessible	Not accessible	Access prohibited	Access prohibited
CopyCBufToTMem()	Not accessible	Accessible	Not accessible	Accessible
CopyTMemToCBuf()	Accessible	Not accessible	Accessible	Not accessible
ReadFromCBuf:	Not accessible	Accessible	Access prohibited	Access prohibited
WriteToBuf	Access prohibited	Access prohibited	Accessible	Not accessible

When an API function or BIOS function attempts to access a communication buffer that is not accessible, an error value is returned as the return value, and it is possible to decide the next processing based on this value.

3.6 Task Management

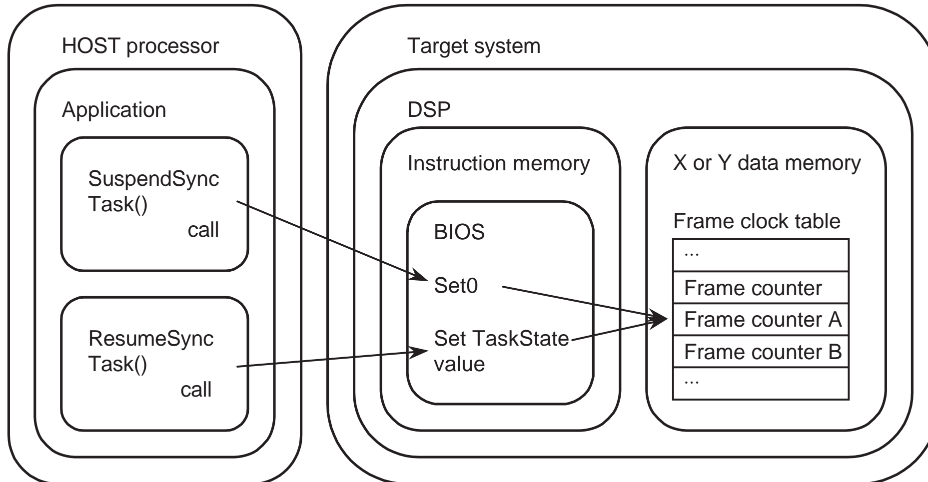
In the HOST API system, an operating task can be changed to a stopped task, and a stopped task can be changed to an operating task by using SuspendSyncTask() and ResumeSyncTask(). "0" is set to the frame counter A[Note] of the specified task by SuspendSyncTask(). If the value of the frame counter prior to setting was other than 0, the change that is made is from an operating task to a stopped task.

The value specified with the TaskState key of the SubTaskXXYY section of the information file is set to frame counter A of the frame clock table of RX77016. If the value of frame counter A was 0, the change that is made is from a stopped task to an operating task.

Figure 3-9 shows the image of task management.

Note This value is the calculation value for the frame counter value. The timer is referenced for the calculation, and the calculation is performed during a fixed cycle. (For details, refer to **RX77016 User's Manual Functions**.)

Figure 3-9. Task Management Image



Remark The host processor in the above figure is assumed to be a personal computer.

CHAPTER 4 INPUT FILES

4.1 Information Files

The HOST API loads up information files to obtain I/O port addresses, communication buffer information, task information, and so on. Information files are in the Windows INI format, and are defined with the following sections and keys.

4.1.1 Device section

(1) Specification of device identifier

- **DeviceDescription key**

  Specify the device identifier with 31 characters or less (2-byte characters are counted as 2 character). The identifier specified here becomes the target device name when changing the BIOS to a communication enabled mode (OpenDEV() call). Characters over the 32nd character are truncated if a device identifier longer than 32 characters is specified.

  In the following example, "SPX0" is specified as the device identifier.

  DeviceDescription = SPX0

4.1.2 IO RESOURCE section

(1) Specification of I/O port address of HDT register

Depending on whether byte access or word access of the HDT register is performed, both the IOPortHDTLow and IOPortHDTHigh keys, or the IOPortHDT key is used.

- **IOPortHDTLow, IOPortHDTHigh keys**

  Specify the I/O port address when performing byte access of the HDT register with 4 or fewer hexadecimal digits. Specify HDT register bits 0 to 7 to IOPortHDTLow, and I/O port address bits 8 to 15 to IOPortHDTHigh.

  In the following description example, 0x8020 is specified to IOPortHDTLow, and 0x8021 is specified to IOPortHDTHigh.

  IOPortHDTLow = 0x8020
  IOPortHDTHigh = 0x8021

- **IOPortHDT key**

  Specify the I/O port address when performing word access of the HDT register with 4 or fewer hexadecimal digits. IOPortHDTLow key definitions and IPOPortHDTHigh key definitions become invalid after the IOPortHDT key.

  In the following description example, 0x8020 is specified to IOPortHST.

  IOPortHDT = 0x8020

(2) Specification of I/O port address of HST register

Depending on whether byte access or word access of the HST register is performed, both the IOPortHSTLow and IOPortHSTHigh keys, or the IOPortHST key is used.

- **IOPortHSTLow, IOPortHSTHigh keys**

  Specify the I/O port address when performing byte access of the HST register with 4 or fewer hexadecimal digits.

  Specify HST register bits 0 to 7 to IOPortHSTLow, and I/O port bits 8 to 15 to IOPortHSTHigh.

  In the following description example, 0x8022 is specified to IOPortHSTLow, and 0x8023 is specified to IOPortHSTHigh.

  ```
  IOPortHSTLow = 0x8022
  IOPortHSTHigh = 0x8023
  ```

- **IOPortHST key**

  Specify the I/O port address when performing word access of the HST register with 4 or fewer hexadecimal digits.

  In the following description example, 0x8022 is specified to IOPortHST.

  ```
  IOPortHST = 0x8022
  ```

(3) Specification of I/O port address for RST pin access

Depending on whether byte access or word access of the I/O port is performed, both the IOPortRSTLow and IOPortRSTHigh keys, or the IOPortRST key is used.

- **IOPortRSTLow, IOPortRSTHigh keys**

  Specify the I/O port when performing byte access of the I/O port for RST pin access in 4 or fewer hexadecimal digits. Specify bits 0 to 7 of the word including the RST pin to IOPortRSTLow, and bits 8 to 16 to IOPortRSTHigh.

  In the following example, 0x8024 is specified to IOPortRSTLow, and 0x8025 is specified to IOPortRSTHigh.

  ```
  IOPortRSTLow = 0x8024
  IOPortRSTHigh = 0x8025
  ```

- **IOPortRST key**

  Define the I/O port address when performing word access of the I/O port for RST pin access with 4 or fewer hexadecimal digits.

  In the following description example, 0x8024 is specified to IOPortRST.

  ```
  IOPortRST = 0x8024
  ```

(4) Specification of RST pin bit assign

Depending on whether byte access or word access of the I/O port is performed, both the IOPortRSTLow and IOPortRSTHigh keys, or the IOPortRST key is used.

- **BitAssignRSTLow, BitAssignRSTHigh keys**

  Specify the bit position of the RST pin when performing byte access of the I/O port for RST pin access as 2 or fewer hexadecimal digits. Specify bit assign of bits 0 to 7 of the word including the RST pin to BitAssignRSTLow, bits bit assign of bits 8 to 15 to BitAssignRSTHigh. Specify 1 for the corresponding position, and 0 for other positions.

  In the following example, 0x01 is specified to BitAssignRSTLow, and 0x00 is specified to BitAssignRSTHigh.

  BitAssignRSTLow = 0x01
  BitAssignRSTHigh = 0x00

- **BitAssignRST key**

  Specify the bit position of the RST pin when performing word access of the I/O port for RST pin access as 4 or fewer hexadecimal digits. Specify 1 for the corresponding position, and 0 for other positions.

  In the following example, 0x0001 is specified to BitAssignRST.

  BitAssignRST = 0x0001

(5) Mask data of I/O port for RST pin access

Depending on whether byte access or word access of the I/O port for RST pin access is performed, both the MaskDataRSTLow and MaskDataRSTHigh keys, or the MaskRST key is used.

- **MaskDataRSTLow, MaskDataRSTHigh keys**

  Specify the OR mask data of other than the RST pin when performing byte access of the I/O port for RST pin access as 2 or fewer hexadecimal digits. Specify bits 0 to 7 of the word including the RST pin to MaskDataRSTLow, and bits 8 to 15 of the mask data to MaskDataRSTHigh (in the case of access to the RST pin, masking is not performed even if 1 is set to the RST pin bit position).

  In the following description example, 0x30 is specified to MaskDataRSTLow, and 0x05 is specified to MaskDataRSTHigh.

  MaskDataRSTLow = 0x30
  MaskDataRSTHigh = 0x05

- **MaskDataRST key**

  Specify the OR mask data of other than the RST pin when performing word access of the I/O port for RST pin access as 4 or fewer hexadecimal digits (in the case of access to the RST pin, masking is not performed even if 1 is set to the RST pin bit position).

  In the following description example, 0x3005 is specified to MaskDataRST.

  MaskDataRST = 0x3005

4.1.3 Task section

(1) Specification of number of tasks

- **nTask key**

  Specify the number of tasks controlled by the RX77016 on the target device as 4 or fewer decimal digits.
  In the following description example, 2 is specified for nTask.
  nTask = 2

4.1.4 TaskXX section

In the TaskXX section, the number of sections specified with the nTask key must be described. Continuous numbers starting from 01 are allocated to XX. If 3 is specified to the nTask key, 01 to 03 is allocated to XX, and the Task01, Task02, and Task03 sections are described. Moreover, TaskXX must be indicated to the same task as TaskX of the RX77016.

(1) Specification of number of subtasks

- **nSubTask key**

  Specify the number of subtasks of TaskX as 4 or fewer decimal digits.
  In the following description example, 1 is specified for nSubTask of Task01, and 3 is specified for nSubTask of Task02.

```
[Task01]
  nSubtask = 1
[Task02]
  nSubTask = 3
```

4.1.5 SubTaskXXYY section

In the SubTaskXXYY section, the number of sections specified with the nSubTask key must be described. The same value as for the TaskXX section is allocated to XX. Continuous numbers starting from 00 according to the number specified to the nSubTask key are allocated to YY.

If 2 is specified to the nTask key of the Task section, 2 is specified to the nSubTask key of the Task01 section, and 3 is specified to the nSubTask key of the task02 section, 0100, 0101, 0200, 0201, and 0202 are allocated to XXYY, and the SubTask0100, SubTask0101, SubTask0200, SubTask0201, and SubTask0202 sections are described. Moreover, SubTaskXXYY must be indicated to the same task as SubTaskXY of the RX77016.

(1) Specification of subtask name

- **TaskName key**

  Specify the subtask name of SubTaskXXYY as 31 or fewer characters. If the subtask name is specified as 32 or more characters, all characters past the 31st will be cut off.
  In the following description example, "SWAPIO" is specified to the TaskName of the SubTask0101 section.

```
[SubTask0101]
  TaskName = SWAPIO
```

(2) Specification of subtask state

- **TaskState key**

  Specify the subtask state[Note] of SubTaskXXYY as 4 or fewer hexadecimal digits.

  In the following description example, 1 is specified to TaskState of the SubTask0101 section.

  ```
  [SubTask0101]
   TaskState = 1
  ```

Note Value set by ResumeSyncTask() to frame counter A of the frame clock table.

4.1.6 Information file description example

Figure 4-1 shows a description example of an information file.

Figure 4-1. Information File Description Example

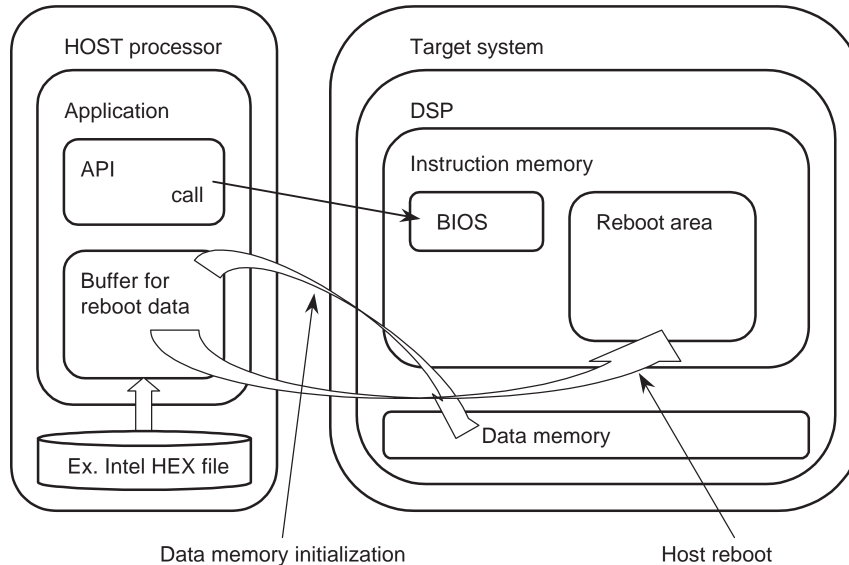
```
;**********************************************************
;* uPD77016 series processor information file For example  *
;**********************************************************
[Device]
DeviceDescription=SPX0
[IO RESOURCE]
IOPortHDT=0x8020
IOPortHST=0x8022
IOPortRST=0x8024
BitAssignRST=0x0001
MaskDataRST=0x3005
[Task]
nTask =2
[Task01]
nSubTask=1
[Task02]
nSubTask=3
[SubTask0100]
TaskName=SWAPIO
TaskState=1
[SubTask0200]
TaskName=REVERB
TaskState=1
[SubTask0201]
TaskName=CHORUS
TaskState=1
[SubTask0202]
TaskName=SURROUND
TaskState=1
```

4.2 HEX File

When rebooting instructions or data, HOST API must accumulate load data beforehand in a buffer for the DSP. An extension Intel HEX file (such as .HXI, .HDX, or .HDY files, which are output by the WB77016) can be specified as that data source. In the HOST API system, the API function to accumulate the extension Intel HEX file data source to the specified buffer is supported.

Figure 4-2 shows the image of Reboot using an extension Intel HEX file.

Figure 4-2. Image of Reeboot Using Extension Intel HEX File



Remark The host processor in the above figure is assumed to be a personal computer.

CHAPTER 5 HOST API SYSTEM

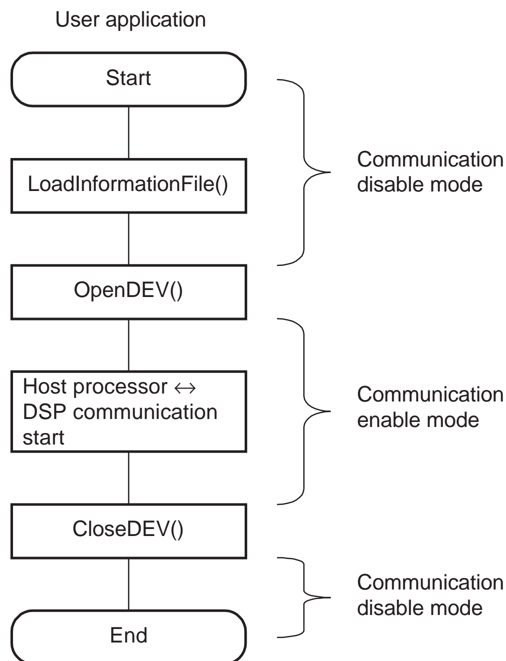
5.1 BIOS Mode

There are two BIOS modes, the communication enabled mode and the communication disabled mode. The communication enabled mode is a mode that enables communication to be performed freely between API applications and BIOS applications. When communication is not possible, a communication path is established by calling OpenDEV(), and the mode changes to the communication enabled mode. At this time, the BIOS function that has received the HDT register data written by the API function judges this data as a command or a parameter, and performs the appropriate processing.

The communication disabled mode is a mode in which communication between API functions and BIOS functions cannot be performed. When communication is possible, the communication path is closed by calling CloseDEV() and the mode changes to the communication disabled mode. The BIOS initial status is the communication disabled mode. At this time, the data written to the HDT register is judged to be meaningless, and it is not decoded.

Figure 5-1 shows the change of the BIOS mode when the user application calls OpenDEV() and the CloseDEV().

Figure 5-1. Change in BIOS Mode When Calling OpenDEV(), CloseDEV()



5.2 BIOS Function Call

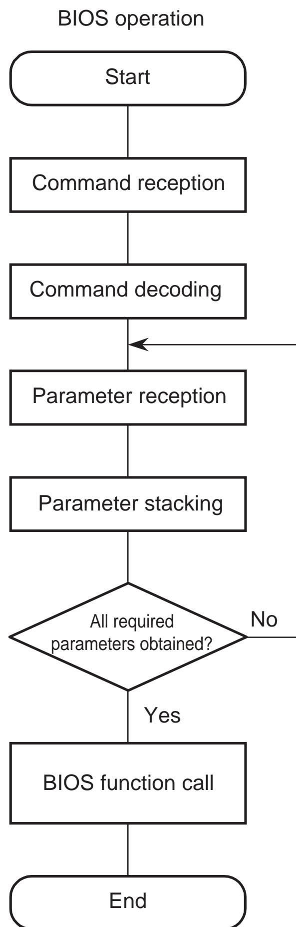
In the HOST API system, all API functions except for the HOST API management functions send commands and parameters to the DSP in order to call the BIOS functions that realize functions.

Commands refer to codes for calling the desired BIOS function, and by decoding these commands, it is possible to determine the function to be called and the required parameters.

Parameters refer to the data required when calling a function corresponding to a command. For example, the write address and write data for memory write are two such parameters. The BIOS performs a BIOS function call when the required number of parameters is obtained when it stacks parameters in the stack area called command buffer.

Figure 5-2 shows the operation flow of the BIOS kernel.

Figure 5-2. BIOS Kernel Operation Flow



CHAPTER 6 API FUNCTIONS

6.1 Data Definition

The data types shown in Table 6-1 are defined for API functions.

Table 6-1. Data Types

Defined Data Type	C Language Expression Data Type	Application
BYTE	unsigned char	Unsigned 1-byte data
LPBYTE	unsigned char*	Unsigned 1-byte data pointer
SWORD	short	Signed 2-byte data
WORD	unsigned short	Unsigned 2-byte data
LPWORD	unsigned short*	Unsigned 2-byte data pointer
DWORD	unsigned long	Unsigned 4-byte data
devHnd	unsigned short	Device handle
cbHnd	unsigned short	CB handle
lpcbHnd	unsigned short FAR*	CB handle pointer
SubTaskInfo	typedef struct { WORD ID ; BYTE Name[MAX_NAMES] ; WORD FrameSubValue ; WORD FrameSubAddress ; } SubTaskInfo ;	 Subtask ID Subtask name FrameSub value FrameSub address
RebootParam	typedef struct { WORD CmdID ; WORD NumInstSize ; WORD SrcStarAddr ; WORD *SrcPointer ; WORD DestStartAddr ; } RebootParam ;	 Reboot command ID Boot size Source start address (during self boot) Source pointer (during host boot) Destination start address
CBCaps	Typedef struct { WORD StartAddr ; WORD nSize ; WORD RWFlag ; DWORD StructAddr ; } CBCaps ;	 CB start address CB size CB R/W flag CB information address

Remark CB: Communication Buffer

The following character constants are defined for API functions.

Table 6-2. Character Constants

Defined Constant String	C Language Expression Data	Application
SUCCESS	(SWORD)0x0000	API function return value
FILE_NOTFOUND	(SWORD)0xfff5	API function return value
SUBTASKTBL_FULL	(SWORD)0xfff6	API function return value
CBTBL_FULL	(SWORD)0xfff7	API function return value
CB_FULL	(SWORD)0xfff8	API function return value
CB_EMPTY	(SWORD)0xfff9	API function return value
FUNCID_ERROR	(SWORD)0xfffa	API function return value
ALREADY_OPEN	(SWORD)0xfffb	API function return value
CBHND_ERROR	(SWORD)0xfffc	API function return value
DEVHND_ERROR	(SWORD)0xfffd	API function return value
DEVNAME_ERROR	(SWORD)0xfffe	API function return value
FAIL	(SWORD)0xffff	API function return value
_MHA_SELPCMD_CMD	0x8000	Command ID
_MHA_IDTAGCMD_CMD	0x4000	Command ID
_MHA_REBOOT_SUBCMD	0x0000	Sub-command ID
_MHA_DIRECT_RX_SUBCMD	0x0002	Sub-command ID
_MHA_DIRECT_RY_SUBCMD	0x0004	Sub-command ID
_MHA_DIRECT_WX_SUBCMD	0x0006	Sub-command ID
_MHA_DIRECT_WY_SUBCMD	0x0008	Sub-command ID
_MHA_COPY_XTOX_SUBCMD	0x000a	Sub-command ID
_MHA_COPY_YTOY_SUBCMD	0x000c	Sub-command ID
_MHA_COPY_XTOY_SUBCMD	0x000e	Sub-command ID
_MHA_COPY_YTOX_SUBCMD	0x0010	Sub-command ID
_MHA_REBOOT_HOST_CMD	0x0005	Reboot command ID
_MHA_REBOOT_X_BYTE_CMD	0x0004	Reboot command ID
_MHA_REBOOT_Y_BYTE_CMD	0x0003	Reboot command ID
_MHA_REBOOT_X_WORD_CMD	0x0002	Reboot command ID
_MHA_REBOOT_Y_WORD_CMD	0x0001	Reboot command ID
READ	0x0000	R/W flag
WRITE	0x0001	R/W flag
CB_NOTUSED	0xffff	CB usage status flag
WAIT	0x0000	CB access restriction flag
NON_WAIT	0x0001	CB access restriction flag
INST	0x0000	Hex file judgment flag
DATA	0x0001	Hex file judgment flag

Remark CB: Communication Buffer

6.2 API Application Functions

6.2.1 Initial setting functions

(1) OpenDEV()

[Description]

Establishes a communication path to the device set to DEV_Name and writes that device handle to the device.

```
SWORD OpenDEV(
  devHnd *device          /* device handle storage destination pointer */
  LPBYTE DEV_Name         /* target device name pointer */
);
```

When this function is called, the BIOS goes into the communication enabled mode. When an API function is used to perform some data communication with a target device, this function must be called beforehand.

DEV_Name describes the device name specified with the DeviceDes key of the Device section of the Information File.

This function will display its worth when the HOST API supports multiple devices in the future. It is included in the current version in order to minimize user program changes once the HOST API starts supporting multiple devices.

[Return Value]

SUCCESS	...	Communication path was successfully established.
DEVNAME_ERROR	...	A device name that cannot be recognized has been specified.
ALREADY_OPEN	...	The specified device is already open.
FAIL	...	Communication with the DSP failed.

[Code Example]

```
int main()
{
   devHnd device1 ;
   if(LoadInformationFile("devinfo.ini")!=SUCCESS) return -1 ;
   if(OpenDEV(&device1,"Device1")!=SUCCESS ){
      puts("It failed to Open device.") ;
      return -1 ;
   }
   if(SetCommunicateValue(device1,0x0000)!=SUCCESS) return -1 ;

   if(CloseDEV(device1)!=SUCCESS) return -1 ;
   return 0;
}
```

(2) CloseDEV()

[Description]

Closes the communication path with the target device.

SWORD CloseDEV(
 devHnd device /* target device handle */
);

When this function is called, the BIOS goes into the communication disabled mode. This function is used when the API, which performs some data communication with a target device, is not used further.
This function will display its worth when the HOST API supports multiple devices in the future. It is included in the current version in order to minimize user program changes once the HOST API starts supporting multiple devices.

[Return Value]

SUCCESS ... Communication path was successfully closed.
DEVHND_ERROR ... A device handle that cannot be recognized has been specified.
FAIL ... Communication with the DSP failed.

[Code Example]

```
int main()
{
   devHnd device1 ;
   if(LoadInformationFile("devinfo.ini")!=SUCCESS)  return -1 ;
   if(OpenDEV(&device1,"Device1")!=SUCCESS )  return -1 ;
   if(SetCommunicateValue(device1,0x0000)!=SUCCESS) return -1 ;

   if(CloseDEV(device1)!=SUCCESS){
      puts("It failed to Close device.") ;
      return -1 ;
   }
   return 0;
}
```

(3) RebootDEV()

[Description]

Performs X word/byte reboot, Y word/byte reboot, or self boot for the target device.

```
SWORD RebootDEV(
  devHnd device,                /* target device handle */
  RebootParam*SpxRbtParam       /* RebootParam structure pointer */
);
```

When host boot is performed using this function, it is necessary to accumulate boot data in the buffer with head address SpxRbtParam → SrcStartAddr. If an extension Intel HEX file is used as the data source, the HexToBuf() function can be used to perform buffering.

[Return Value]

SUCCESS	...	Reboot was successful.
DEVHND_ERROR	...	A device handle that cannot be recognized has been specified.
FAIL	...	Communication with the DSP failed.

[Code Example]

```
WORD IBuff[]={ 0x0000,0x0000,0x0000,0x0000 } ;
RebootParam RebootParam1={
   _MHA_REBOOT_HOST_CMD, // Reboot Function ID
   2, // Size of reboot instruction
   0, // Start address of source
   IBuff, // Pointer of source
   1024, // Start address of destination
} ;
int main()
{
   devHnd device1 ;
   if(LoadInformationFile("devinfo.ini")!=SUCCESS) return -1 ;
   if(OpenDEV(&device1,"Device1")!=SUCCESS ) return -1 ;
   if(SetCommunicateValue(device1,0x0000)!=SUCCESS) return -1 ;

   if(RebootDEV(device1,&RebootParam1)!=SUCCESS){
      puts("It failed to reboot device.") ;
   }

   if(CloseDEV(device1)!=SUCCESS) return -1 ;
   return 0;
}
```

(4) ResetDEV()

[Description]

Performs system reset for the target device.

```
SWORD ResetDEV(
  devHnd device            /* target device handle */
);
```

[Return Value]

SUCCESS ... System reset was successful.

DEVHND_ERROR ... A device handle that cannot be recognized has been specified.

[Code Example]

```
int main()
{
   devHnd device1 ;
   if(LoadInformationFile("devinfo.ini")!=SUCCESS) return -1 ;
   if(OpenDEV(&device1,"Device1")!=SUCCESS ) return -1 ;
   if(SetCommunicateValue(device1,0x0000)!=SUCCESS) return -1 ;

   if(ResetDev(device1)!=SUCCESS){
      puts("It failed to reset device.") ;
      return -1 ;
   }

   return 0;
}
```

(5) SetCommunicateValue()

[Description]

Sets the data specified to the OS communication area (*_MOS_SPX-pause:X) of the RX77016.

```
SWORD SetCommunicate Value(
  DevHnd device,          /* target device handle */
  WORD Value              /* setting data */
);
```

[Return Value]

SUCCESS ... Setting was successful

DEVHND_ERROR ... A device handle that cannot be recognized has been specified.

FAIL ... Communication with the DSP failed.

[Code Example]

```
int main()
{
   devHnd device1 ;
   if(LoadInformationFile("devinfo.ini")!=SUCCESS) return -1 ;
   if(OpenDEV(&device1,"Device1")!=SUCCESS ) return -1 ;
   if(SetCommunicateValue(device1,0x0000)!=SUCCESS){
      puts("It failed to set Communicate Value.") ;
      return -1 ;
   }

   if(CloseDEV(device)!=SUCCESS) return -1 ;
   return 0;
}
```

6.2.2 Task management functions

(1) ShowTask()

[Description]

Shows the task information loaded up with the loadInformationFile(). If RETURN_TO_WINDOW is defined, the information is displayed in the message box on Windows. If RETURN_TO_WINDOW is not defined, the information is displayed on the console, and the Func structure pointer is copied to pFunc.

```
SWORD ShowTask(
  devHnd device,            /* target device handle */
  WORD SubTaskID,           /* subtask ID */
  SubTaskInfo*pSubTask      /* Func structure copy destination pointer */
);
```

The displayed items consist of the subtask name, subtask ID, current state, and frame counter A.

[Return Value]

SUCCESS	...	Display was successful
DEVHND_ERROR	...	A device handle that cannot be recognized has been specified.
FUNCID_ERROR	...	A subtask ID that cannot be recognized has been specified.

[Code Example]

```
int main()
{
   Func Func1 ;
   devHnd device1 ;
   if(LoadInformationFile("devinfo.ini")!=SUCCESS) return -1 ;
   if(OpenDEV(&device1,"Device1")!=SUCCESS ) return -1 ;
   if(SetCommunicateValue(device1,0x0000)!=SUCCESS) return -1 ;

   if(ShowTask(&device1,100,&Func1)!=SUCCESS ){
      puts("It failed to show task information.") ;
   }

   if(CloseDEV(device1)!=SUCCESS) return -1 ;
   return 0;
}
```

(2) ResumeSyncTask()

[Description]

Writes the devInf[device].subtask[SubTaskID]->FrameSubValue value to frame counter A of the frame clock table of the RX77016 that corresponds to the specified task. If a value other than 0 is written when the original frame counter A value is 0, that task becomes an executable task.

```
SWORD ResumeSyncTask(
  devHnd device,          /* target device handle */
  WORD SubTaskID          /* subtask ID */
```

[Return Value]

SUCCESS	... Update of the frame counter A value was successful
DEVHND_ERROR	... A device handle that cannot be recognized has been specified.
FUNCID_ERROR	... A subtask ID that cannot be recognized has been specified.
FAIL	... Communication with the DSP failed.

[Code Example]

```
int main()
{
   devHnd device1 ;
   if(LoadInformationFile("devinfo.ini")!=SUCCESS) return -1 ;
   if(OpenDEV(&device1,"Device1")!=SUCCESS ) return -1 ;
   if(SetCommunicateValue(device1,0x0000)!=SUCCESS) return -1 ;

   if(ResumeSyncTask(device1,100)!=SUCCESS ){
      puts("It failed to resume task.") ;
   }

   if(CloseDEV(device1)!=SUCCESS) return -1 ;
   return 0;
}
```

(3) SuspendSyncTask()

[Description]

Writes 0 to frame counter A of the frame clock table of the RX77016 that corresponds to the specified task. If the original frame counter A value is other than 0, that task becomes a stopped task.

```
SWORD SuspendSyncTask(
  devHnd device,            /* target device handle */
  WORD SubTaskID            /* subtask ID */
);
```

[Return Value]

SUCCESS	...	Update of the frame counter A value was successful
DEVHND_ERROR	...	A device handle that cannot be recognized has been specified.
FUNCID_ERROR	...	A subtask ID that cannot be recognized has been specified.
FAIL	...	Communication with the DSP failed.

[Code Example]

```
int main()
{
   devHnd device1 ;
   if(LoadInformationFile("devinfo.ini")!=SUCCESS) return -1 ;
   if(OpenDEV(&device1,"Device1")!=SUCCESS ) return -1 ;
   if(SetCommunicateValue(device1,0x0000)!=SUCCESS) return -1 ;

   if(SuspendSyncTask(device1,100)!=SUCCESS ){
      puts("It failed to suspend task.") ;
   }

   if(CloseDEV(device1)!=SUCCESS) return -1 ;
   return 0;
}
```

6.2.3 Memory access functions

(1) CreateCBuf()

[Description]

Creates an nSize used area from the communication buffer heap area previously secured. Moreover, allocates a communication buffer handle that corresponds to the prepared used area.

```
SWORD CreateCBuf(
  devHnd device,        /* target device handle */
  WORD nSize,           /* communication buffer size */
  WORD RWFlag,          /* R/W flag */
  cbHnd*cbuf            /* communication buffer handle storage destination pointer */
);
```

If the nSIZE value is larger than the remaining heap area, or if the planned number of buffers is exceeded, no new area is created.

"READ" or "WRITE" is specified for RWFlag, in the case of the read communication buffer and in the case of the write communication buffer, respectively.

[Return Value]

SUCCESS	...	Creation was successful
DEVHND_ERROR	...	A device handle that cannot be recognized has been specified.
CBHND_ERROR	...	The number of buffers specified with biosucfg.h has been exceeded.
CBTBL_FULL	…	The number of buffers specified with hapiucfg.h has been exceeded.
FAIL	...	Communication with the DSP failed.

[Code Example]

```
int main()
{
   devHnd device1 ;
   cbHnd cbHnd1 ;
   if(LoadInformationFile("devinfo.ini")!=SUCCESS) return -1 ;
   if(OpenDEV(&device1,"Device1")!=SUCCESS ) return -1 ;
   if(SetCommunicateValue(device1,0x0000)!=SUCCESS) return -1 ;

   if(CreateCBuf(device1,10,READ,&cbHnd1)!=SUCCESS ){
      puts("It failed to create communication buffer.") ;
   }

   if(CloseDEV(device1)!=SUCCESS) return -1 ;
   return 0;
}
```

(2) FreeCBuf()

[Description]

Releases the used areas of the communication buffer that have become unnecessary.

```
SWORD FreeCBuf(
  devHnd device,        /* target device handle */
  cbHnd cbuf            /* communication buffer handle releasing area */
);
```

[Return Value]

SUCCESS	... Area was successfully released
DEVHND_ERROR	... A device handle that cannot be recognized has been specified.
CBHND_ERROR	... A communication buffer handle that cannot be recognized has been specified.
FAIL	... Communication with the DSP failed.

[Code Example]

```
int main()
{
  devHnd device1 ;
  cbHnd cbHnd1 ;
  if(LoadInformationFile("devinfo.ini")!=SUCCESS) return -1 ;
  if(OpenDEV(&device1,"Device1")!=SUCCESS ) return -1 ;
  if(SetCommunicateValue(device1,0x0000)!=SUCCESS) return -1 ;

  if(CreateCBuf(device1,10,READ,&cbHnd1)!=SUCCESS ){
    puts("It failed to create communication buffer.") ;
  }

  if(CloseDEV(device1)!=SUCCESS) return -1 ;
  return 0;
}
```

(3) GetCBufCaps()

[Description]

Copies communication buffer information to the CBCaps structure. If RETURN_TO_WINDOW is defined, displays this information to the message box on Windows.

```
SWORD GetCbufCaps(
  devHnd device,           /* target device handle */
  cbHnd cbuf,              /* target communication handle */
  CBCaps *pCommBufCaps     /* CBCaps structure copy destination pointer */
```

[Return Value]

SUCCESS	...	Copy or display was successful.
DEVHND_ERROR	...	A device handle that cannot be recognized has been specified.
CBHND_ERROR	...	A communication buffer handle that cannot be recognized has been specified.
FAIL	...	Communication with the DSP failed.

[Code Example]

```
int main()
{
   devHnd device1 ;
   CBCaps CBCaps2 ;
   if(LoadInformationFile("devinfo.ini")!=SUCCESS) return -1 ;
   if(OpenDEV(&device1,"Device1")!=SUCCESS ) return -1 ;
   if(SetCommunicateValue(device1,0x0000)!=SUCCESS) return -1 ;
   if(CreateCBuf(device1,10,READ,&cbHnd1)!=SUCCESS ) return -1 ;

   if(GetCBufCaps(device1,cbHnd1,&CBCaps2)!=SUCCESS){
      puts("It failed to get communication buffer capabilities.") ;
   }

   if(CloseDEV(device1)!=SUCCESS) return -1 ;
   return 0;
}
```

(4) InitCBuf()

[Description]

Clears to 0 the contents of the specified used area of the write communication buffer.

```
SWORD InitCBuf(
  devHnd device,          /* target device handle */
  cbHnd cbuf              /* target communication buffer handle */
);
```

If the data written previously has not yet been read, this function returns CB_FULL and clearing cannot be performed.

[Return Value]

SUCCESS	...	Clearing to 0 was successful.
DEVHND_ERROR	...	A device handle that cannot be recognized has been specified.
CBHND_ERROR	...	A communication buffer handle that cannot be recognized has been specified.
CB_FULL	...	Previously written data has not been read.
FAIL	...	Communication with the DSP failed.

[Code Example]

```
int main()
{
   devHnd device1 ;
   if(LoadInformationFile("devinfo.ini")!=SUCCESS) return -1 ;
   if(OpenDEV(&device1,"Device1")!=SUCCESS ) return -1 ;
   if(SetCommunicateValue(device1,0x0000)!=SUCCESS) return -1 ;
   if(CreateCBuf(device1,10,WRITE,&cbHnd1)!=SUCCESS ) return -1 ;

   if(InitCBuf(device1,cbHnd1)!=SUCCESS ){
      puts("It failed to initialize communication buffer.") ;
   }

   if(CloseDEV(device1)!=SUCCESS) return -1 ;
   return 0;
}
```

(5) ReadFromTMem()

[Description]

Writes the SrcAddr address data of the data memory on the target device to the lpRData address of the memory on the host processor.

```
SWORD ReadFromTMem(
  devHnd device,          /* target device handle */
  DWORD SrcAddr,          /* source address (SrcAddr+0x10000 in case of YMEM) */
  LPWORD lpRData          /* read data write destination pointer */
);
```

In the case of read from X memory, specify the address as is, and in the case of read from Y memory, specify address + 0x10000 for SrcAddr.

[Return Value]

SUCCESS ... Read was successful.
DEVHND_ERROR ... A device handle that cannot be recognized has been specified.
FAIL ... Communication with the DSP failed, or a parameter is abnormal.

[Code Example]

```
int main()
{
   devHnd device1 ;
   unsigned short RData ;
   if(LoadInformationFile("devinfo.ini")!=SUCCESS) return -1 ;
   if(OpenDEV(&device1,"Device1")!=SUCCESS ) return -1 ;
   if(SetCommunicateValue(device1,0x0000)!=SUCCESS) return -1 ;

   if(ReadFromTMem(device1,0x10000,&RData)!=SUCCESS ){
      puts("It failed to read data .") ;
   }

   if(CloseDEV(device1)!=SUCCESS) return -1 ;
   return 0;
}
```

(6) WriteToMem()

[Description]

Writes WData to the DestAddr address of the data memory on the target device.

```
SWORD WriteToTMem(
  devHnd device,        /* target device handle */
  DWORD DestAddr,       /* destination address (DestAddr + 0x10000 in case of YMEM) */
  WORD WData            /* write data */
```

In the case of write to X memory, specify the address as is, and in the case of write to Y memory, specify address + 0x10000 for DestAddr.

[Return Value]

SUCCESS	...	Write was successful.
DEVHND_ERROR	...	A device handle that cannot be recognized has been specified.
FAIL	...	Communication with the DSP failed, or a parameter is abnormal.

[Code Example]

```
int main()
{
   devHnd device1 ;
   if(LoadInformationFile("devinfo.ini")!=SUCCESS) return -1 ;
   if(OpenDEV(&device1,"Device1")!=SUCCESS ) return -1 ;
   if(SetCommunicateValue(device1,0x0000)!=SUCCESS) return -1 ;

   if(WriteToTMem(device1,0x0000,0x369c)!=SUCCESS ){
      puts("It failed to write data .") ;
   }

   if(CloseDEV(device1)!=SUCCESS) return -1 ;
   return 0;
}
```

(7) ReadFromCBuf()

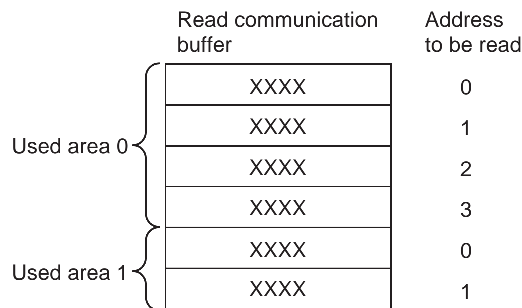
[Description]

Writes the data of any address of the specified used area of the read communication buffer to the IpRData address of the memory on the host processor.

```
SWORD ReadFromCBuf(
  devHnd device,          /* target device handle */
  cbHnd cbuf,             /* target communication buffer handle */
  WORD RelAddr,           /* read destination address */
  LPWORD IpRData          /* destination pointer */
  WORD Flag               /* Access restriction flag */
);
```

For the read destination address (RelAddr), specify the number of the data in the specified used area that is to be read (refer to Figure 6-1).

Figure 6-1. Read Communication Buffer Read Destination Address



Specify either WAIT or NON_WAIT for the access restriction flag (Flag).

In the case of WAIT, data write using CopyTMemToCBuf() or WriteToCBuf: is not enabled.

In the case of NON_WAIT, data write using CopyTMemToCBuf() or WriteToCBuf: is enabled.

For example, in the case of Figure 6-1, it is possible to perform write upon reading 1 data even in used area 0 (size 4). To read all the data of size 4 used area, the following 4 steps are required.

```
ReadFromCBuf(device,cbuf,0,RData++,WAIT) ;
ReadFromCBuf(device,cbuf,1,RData++,WAIT) ;
ReadFromCBuf(device,cbuf,2,RData++,WAIT) ;
ReadFromCBuf(device,cbuf,3,RData++,NON_WAIT) ;
```

However, in the case of Flag detection for ReadFromCBuf(), WAIT is judged in the case of a value other than 0, and NON_WAIT is judged if the value is 0. Therefore, the following description is possible if one uses the characteristics of the Flag detection method.

```
for(i=0 ; i<=3 ; i++) ReadFromCBuf(device,cbuf,i,RData++,3-i) ;
```

If all the current data has been read, this function returns CB_EMPTY and read is not performed.

[Return Value]

SUCCESS	...	Read was successful.
DEVHND_ERROR	...	A device handle that cannot be recognized has been specified.
CBHND_ERROR	...	A communication buffer handle that cannot be recognized has been specified.
CB_EMPTY	...	The communication buffer contents have either been read or are empty.
FAIL	...	Communication with the DSP failed, or a parameter is abnormal.

[Code Example]

```
int main()
{
   devHnd device1 ;
   unsigned short RData[10] ;
   if(LoadInformationFile("devinfo.ini")!=SUCCESS) return -1 ;
   if(OpenDEV(&device1,"Device1")!=SUCCESS ) return -1 ;
   if(SetCommunicateValue(device1,0x0000)!=SUCCESS) return -1 ;
   if(CreateCBuf(device1,10,READ,&cbHnd1)!=SUCCESS ) return -1 ;

   for(int i=9 ; i>=0 ; i--) {
      if(ReadFromCBuf(device1,cbHnd1,i,RData++,i)!=SUCCESS ){
         puts("It failed to read communication buffer's data.") ;
      }
   }

   if(CloseDEV(device1)!=SUCCESS) return -1 ;
   return 0;
}
```

(8) WriteToCBuf()

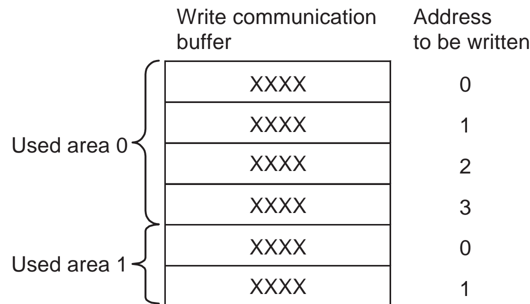
[Description]

Writes WData to any address in the specified used area of the write communication buffer.

```
SWORD WriteToCBuf(
  devHnd device,          /* target device handle */
  cbHnd cbuf,             /* target communication handle */
  WORD RelAddr,           /* write destination address */
  SWORD WData             /* write data */
  WORD Flag               /* access restriction flag */
);
```

The write destination address (RelAddr) specifies the number area of the specified used area to which data is to be written. (refer to **Figure 6-2**.)

Figure 6-2. Write Communication Buffer Write Destination Address



Specify either WAIT or NON_WAIT for the access restriction flag (Flag).

In the case of WAIT, data read using CopyCBufToTMem() or ReadFromCBuf: is not enabled.

In the case of NON_WAIT, read using CopyCBufToTMem() or ReadFromCBuf: is enabled.

When NON_WAIT is set, it is possible to enable read upon write of 1 data, even in the case of used area 0 (size 4) mentioned above. To write all the data of size 4 used area, the following 4 steps are required.

```
WriteToCBuf(device,cbuf,0,*WData++,WAIT) ;
WriteToCBuf(device,cbuf,1,*WData++,WAIT) ;
WriteToCBuf(device,cbuf,2,*WData++,WAIT) ;
WriteToCBuf(device,cbuf,3,*WData++,NON_WAIT) ;
```

However, in the case of WriteToCBuf(), WAIT is judged in the case of a value other than 0 for Flag detection, and NON_WAIT is judged if the value is 0. Therefore, the following description is possible if one uses the characteristics of the Flag detection method.

```
for(i=0 ; i<=3 ; i++) WriteToCBuf(device,cbuf,i,*WData++,3-i) ;
```

If some previously written data remains unread, this function returns CB_FULL and write is not performed.

[Return Value]

SUCCESS ... Read was successful.
DEVHND_ERROR ... A device handle that cannot be recognized has been specified.
CBHND_ERROR ... A communication buffer handle that cannot be recognized has been specified.
CB_FULL ... Previously written data has not been read.
FAIL ... Communication with the DSP failed, or a parameter is abnormal.

[Code Example]

```
 int main()
 {
    devHnd device1 ;
    if(LoadInformationFile("devinfo.ini")!=SUCCESS) return -1 ;
    if(OpenDEV(&device1,"Device1")!=SUCCESS ) return -1 ;
    if(SetCommunicateValue(device1,0x0000)!=SUCCESS) return -1 ;
    if(CreateCBuf(device1,10,WRITE,&cbHnd1)!=SUCCESS  )  return  -1
;

    for(int i=9 ; i>=0 ; i--){
       if(WriteToCBuf(device1,cbHnd1,i,0x369c,i)!=SUCCESS ){
          puts("It   failed   to   write   data   to   communication
buffer.") ;
       }
    }

    if(CloseDEV(device1)!=SUCCESS) return -1 ;
    return 0;
 }
```

(9) CopyTMemToHost()

[Description]

Copies SrcAddr + nSize – 1 address data from the SrcAddr address of data memory on the target device from the DestPtr of memory on the host processor to the DestPtr + nSize - 1 address area.

```
SWORD CopyTMemToHost(
  devHnd device,          /* target device handle */
  DWORD SrcAddr,          /* source address (SrcAddr + 0x10000 in case of YMEM) */
  LPWORD DestPtr,         /* destination pointer */
  WORD nSize              /* copy size */
);
```

In the case of copy from X memory, specify the address as is, and in the case of read from Y memory, specify address + 0x10000 for SrcAddr.

[Return Value]

SUCCESS	...	Copy was successful.
DEVHND_ERROR	...	A device handle that cannot be recognized has been specified.
FAIL	...	Communication with the DSP failed, or a parameter is abnormal.

[Code Example]

```
int main()
{
   devHnd device1 ;
   unsigned short RData[10] ;
   if(LoadInformationFile("devinfo.ini")!=SUCCESS) return -1 ;
   if(OpenDEV(&device1,"Device1")!=SUCCESS ) return -1 ;
   if(SetCommunicateValue(device1,0x0000)!=SUCCESS) return -1 ;

   if(CopyTMemToHost(device1,0x10000,RData,0x10)!=SUCCESS ){
      puts("It failed to copy data.") ;
   }

   if(CloseDEV(device1)!=SUCCESS) return -1 ;
   return 0;
}
```

(10) CopyHostToTMem()

[Description]

Copies SrcPtr + nSize – 1 address data from the SrcPtr address of the memory on the host processor from the DestAddr of memory on the target device to the DestAddr + nSize - 1 address area.

```
SWORD CopyHostToTMem(
  devHnd device,        /* target device handle */
  LPWORD SrcPtr,        /* source address */
  DWORD DestAddr,       /* destination address (DestAddr + 0x1000 in case of YMEM) */
  WORD nSize            /* copy size */
);
```

In the case of copy to X memory, specify the address as is, and in the case of copy to Y memory, specify address + 0x10000 for DestAddr.

[Return Value]

SUCCESS	...	Copy was successful.
DEVHND_ERROR	...	A device handle that cannot be recognized has been specified.
FAIL	...	Communication with the DSP failed, or a parameter is abnormal.

[Code Example]

```
int main()
{
   devHnd device1 ;
   unsigned short WData[]={ 0,1,2,3,4,5,6,7,8,9 } ;
   if(LoadInformationFile("devinfo.ini")!=SUCCESS) return -1 ;
   if(OpenDEV(&device1,"Device1")!=SUCCESS ) return -1 ;
   if(SetCommunicateValue(device1,0x0000)!=SUCCESS) return -1 ;

   if(CopyHostToTMem(device1,WData,0x10000,10)!=SUCCESS ){
      puts("It failed to OpenDEVice.") ;
      return -1 ;
   }

   if(CloseDEV(device1)!=SUCCESS) return -1 ;
   return 0;
}
```

(11) CopyCBufToHost()

[Description]

Enables data write following copying of the specified used area data of the read communication buffer from the DestPtr address of memory on the host processor to the DestPtr+buffer size – 1 address area.

```
SWORD CopyCBufToHost(
  devHnd device,          /* target device handle */
  cbHnd cbuf,             /* target communication handle */
  LPWORD DestPtr,         /* destination pointer */
);
```

If all the current data has been read, this function returns CB_EMPTY and read is not performed.

[Return Value]

SUCCESS	...	Copy was successful.
DEVHND_ERROR	...	A device handle that cannot be recognized has been specified.
CBHND_ERROR	...	A communication buffer handle that cannot be recognized has been specified.
CB_EMPTY	...	The communication buffer contents have either been read or are empty.
FAIL	...	Communication with the DSP failed.

[Code Example]

```
int main()
{
   devHnd device1 ;
   WORD RData[10] ;
   if(LoadInformationFile("devinfo.ini")!=SUCCESS) return -1 ;
   if(OpenDEV(&device1,"Device1")!=SUCCESS ) return -1 ;
   if(SetCommunicateValue(device1,0x0000)!=SUCCESS) return -1 ;
   if(CreateCBuf(device1,10,READ,&cbHnd1)!=SUCCESS ) return -1 ;

   if(CopyCBufToHost(device1,cbHnd1,RData)!=SUCCESS ){
      puts("It failed to copy communication buffer's data to host.") ;
   }

   if(CloseDEV(device1)!=SUCCESS) return -1 ;
   return 0;
}
```

(12) CopyHostToCBuf()

[Description]

Enables data read following copying of SrcPtr + buffer size – 1 address data from the SrcPtr address on memory on the host processor to the specified used area of the write communication buffer.

```
SWORD CopyHostToCBuf(
  devHnd device,          /* target device handle */
  cbHnd cbuf,             /* target communication handle */
  SWORD*SrcPtr            /* source address */
```

If the previously written data has not yet been read, this function returns CB_FULL and write is not performed.

[Return Value]

SUCCESS	...	Copy was successful.
DEVHND_ERROR	...	A device handle that cannot be recognized has been specified.
CBHND_ERROR	...	A communication buffer handle that cannot be recognized has been specified.
CB_FULL	...	Previously written data has not been read.
FAIL	...	Communication with the DSP failed.

[Code Example]

```
int main()
{
   devHnd device1 ;
   unsigned short WData[]={ 0,1,2,3,4,5,6,7,8,9 } ;
   if(LoadInformationFile("devinfo.ini")!=SUCCESS) return -1 ;
   if(OpenDEV(&device1,"Device1")!=SUCCESS ) return -1 ;
   if(SetCommunicateValue(device1,0x0000)!=SUCCESS) return -1 ;
   if(CreateCBuf(device1,10,WRITE,&cbHnd1)!=SUCCESS ) return -1 ;

   if(CopyHostToCBuf(device1,cbHnd1,&WData)!=SUCCESS ){
      puts("It failed to copy data to communication buffer.") ;
   }

   if(CloseDEV(device1)!=SUCCESS) return -1 ;
   return 0;
}
```

(13) CopyCBufToTMem()

[Description]

Enables data write following copying of nSize data from the start of the specified used area of the read or communication buffer from the DestAddr address of the memory on the same device to the DestAddr + nSize − 1 address.

```
SWORD CopyCBufToTMem(
  devHnd device,          /* target device handle */
  cbHnd cbuf,             /* target communication handle */
  DWORD DestAddr,         * destination address (DestAddr + 0x10000 in case of YMEM)
  WORD nSize              /* copy size */
);
```

If all the current data has been read, this function returns CB_EMPTY and read is not performed.

[Return Value]

SUCCESS	...	Copy was successful.
DEVHND_ERROR	...	A device handle that cannot be recognized has been specified.
CBHND_ERROR	...	A communication buffer handle that cannot be recognized has been specified.
CB_EMPTY ... CB_EMPTY	...	The communication buffer contents have either been read or are empty.
FAIL	...	Communication with the DSP failed, or a parameter is abnormal.

[Code Example]

```
int main()
{
   devHnd device1 ;

   if(LoadInformationFile("devinfo.ini")!=SUCCESS) return -1 ;
   if(OpenDEV(&device1,"Device1")!=SUCCESS ) return -1 ;
   if(SetCommunicateValue(device1,0x0000)!=SUCCESS) return -1 ;
   if(CreateCBuf(device1,10,READ,&cbHnd1)!=SUCCESS ) return -1 ;

   if(CopyCBufToTMem(device1,cbHnd1,0x10000,10)!=SUCCESS ){
      puts("It failed to copy communication buffer's data to memory.") ;
   }

   if(CloseDEV(device1)!=SUCCESS) return -1 ;
   return 0;
}
```

(14) CopyTMemToCBuf()

[Description]

Enables data read following copying of SrcAddr + nSize – 1 address data from the SrcAddr address of memory on the target device from the start address of the specified used area of the read or write communication buffer.

```
SWORD CopyTMemToCBuf(
  devHnd device,        /* target device handle */
  cbHnd cbuf,           /* target communication handle */
  DWORD SrcAddr,        /* source address (SrcAddr + 0x10000 in case of YMEM) */
  WORD nSize            /* copy size */
  );
```

If the previously written data has not yet been read, this function returns CB_FULL and write is not performed.

[Return Value]

SUCCESS	...	Copy was successful.
DEVHND_ERROR	...	A device handle that cannot be recognized has been specified.
CBHND_ERROR	...	A communication buffer handle that cannot be recognized has been specified.
CB_FULL	...	Previously written data has not been read.
FAIL	...	Communication with the DSP failed, or a parameter is abnormal.

[Code Example]

```
int main()
{
   devHnd device1 ;
   unsigned short WData[]={ 0,1,2,3,4,5,6,7,8,9 } ;
   if(LoadInformationFile("devinfo.ini")!=SUCCESS) return -1 ;
   if(OpenDEV(&device1,"Device1")!=SUCCESS ) return -1 ;
   if(SetCommunicateValue(device1,0x0000)!=SUCCESS) return -1 ;

   if(CreateCBuf(device1,10,WRITE,&cbHnd1)!=SUCCESS ) return -1 ;

   if(CopyTMemToCBuf(device1,cbHnd1,&WData,10)!=SUCCESS ){
      puts("It failed to copy data to communication buffer.") ;
   }

   if(CloseDEV(device1)!=SUCCESS) return -1 ;
   return 0;
}
```

(15) CopyTMemToTMem()

[Description]

Copies SrcAddr + nSize – 1 address data from the SrcAddr address of memory on the target device from the DstAddr address to the DstAddr + nSize – 1 address.

```
SWORD CopyTMemToCBuf(
  devHnd device,           /* target device handle */
  DWORD SrcAddr,           /* source address (SrcAddr + 0x10000 in case of YMEM) */
  DWORD DstAddr,           /* destination address (SrcAddr + 0x10000) */
  WORD n                   /* copy size */
);
```

[Return Value]

SUCCESS ... Copy was successful.

DEVHND_ERROR ... A device handle that cannot be recognized has been specified.

FAIL ... Communication with the DSP failed, or a parameter is abnormal.

[Code Example]

```
int main()
{
  devHnd device1 ;
  if(LoadInformationFile("devinfo.ini")!=SUCCESS) return -1 ;
  if(OpenDEV(&device1,"Device1")!=SUCCESS ) return -1 ;
  if(SetCommunicateValue(device1,0x0000)!=SUCCESS) return -1 ;

  if(CopyTMemToTMem(device1,0x0000,0x0010,0x10)!=SUCCESS ){
    puts("It failed to copy data.") ;
  }

  if(CloseDEV(device1)!=SUCCESS) return -1 ;
  return 0;
}
```

6.2.4 HOST API management functions

(1) LoadInformationFile()

[Description]

Analyzes the information file specified by InFile and saves the analysis data to the previously secured InfFile type devinf.

```
SWORD LoadInformationFile(
  LPCSTR InFile             /* information file pointer */
);
```

Be sure to call this function prior to using the API functions in order to perform data communication with the target device, based on the information obtained from the information file.

Moreover, if the file name specified in InFile is not described as a full path, it is judged to exist in the Windows System folder (drive name: /Windows/System).

[Return Value]

SUCCESS	...	Normal termination
FILE_NOTFOUND	...	Specified file could not be found
SUBTASKBL_FULL	...	Number of subtasks exceeding the number of MAX_SUB_TASK of hapiucfg.h
FAIL	...	Abnormal termination

[Code Example]

```
int main()
{
   devHnd device1 ;

   if(LoadInformationFile("devinfo.ini")!=SUCCESS){
      puts("It failed to get Information.") ;
      return -1 ;
   }

   if(OpenDEV(&device1,"Device1")!=SUCCESS ) return -1 ;
   if(SetCommunicateValue(device1,0x0000)!=SUCCESS) return -1 ;
   if(CloseDEV(device1)!=SUCCESS) return -1 ;
   return 0;
}
```

(2) ShowInfoContentsAll()

[Description]

Displays the information loaded with LoadInformationFile(). If RETURN_TO_WINDOW is defined, the information is displayed to a message box on Windows, and if RETURN_TO_WINDOW is not defined, the information is output as standard output.

```
voidShowInfoContentsAll(
  devHnd device            /* target device handle */
);
```

The displayed items consist of the device name, I/O port address of the HDT/HST register and its bit assignment, and the read/write communication buffer information.

[Return Value]

There are no return values.

[Code Example]

```
int main()
{
   devHnd device1 ;
   if(LoadInformationFile("devinfo.ini")!=SUCCESS) return -1 ;
   if(OpenDEV(&device1,"Device1")!=SUCCESS ) return -1 ;
   if(SetCommunicateValue(device1,0x0000)!=SUCCESS) return -1 ;

   ShowInfoContentsAll(device1) ;

   if(CloseDEV(device1)!=SUCCESS) return -1 ;
   return 0;
}
```

(3) HexToBuf()

[Description]

Analyzes extension Intel HEX files and copies SrcAddr + nSize – 1 address data from the SrcAddr address from the DestPtr address of memory on the host processor to the DestPtr + nSize – 1 address.

```
SWORD HexToBuf(
  WORD HexType              /* Hex file type */
  LPCSTR InFile,            /* file name pointer */
  WORD SrcAddr,             /* source address */
  LPWORD DestPtr,           /* destination pointer */
  WORD nSize                /* copy size */
);
```

Specify INST as the Hex file type in the case of an instruction (for example .HXI file), and DATA in the case of data (for example, .HDX or .HDY file).

[Return Value]

SUCCESS	... Save was successful.
FAIL	... Save failed.

[Code Example]

```
int main()
{
   devHnd device1 ;
   unsigned short Dest[10] ;
   if(LoadInformationFile("devinfo.ini")!=SUCCESS) return -1 ;
   if(OpenDEV(&device1,"Device1")!=SUCCESS ) return -1 ;
   if(SetCommunicateValue(device1,0x0000)!=SUCCESS) return -1 ;

   if(HexToBuf("Device1.hdx",0x10,Dest,10)!=SUCCESS ){
      puts("It failed to OpenDEVice.") ;
      return -1 ;
   }

   if(CloseDEV(device1)!=SUCCESS) return -1 ;
   return 0;
}
```

CHAPTER 7 BIOS FUNCTIONS

7.1. Action upon Occurrence of HI Interrupt

BIOS, through allocation of a specific BIOS function, realizes that function as the action upon occurrence of a HI interrupt or HO interrupt.

In the initial state of the BIOS (communication disabled mode), EchoWord: is allocated as the action upon occurrence of an HI interrupt. At this time, by calling OpenDEV(), which is an API function on the user application, the action upon occurrence of an HI interrupt changes to Interpreter:. From this status onward, the mode is the communication enabled mode until the action upon occurrence of an HI interrupt is changed back to EchoWord:.

In the case of the Interpreter: action, depending on the received HDT value (command), either the action upon occurrence of the next HI interrupt is set to EchoWord:, SelPCmd:, or IDTagCmd: is executed.

In the case of SelPCmd:, following reception of the previously set number of parameters, the specified BIOS function is executed.

In the case of IDTagCmd:, the BIOS function created by the user is executed.

7.2. BIOS Functions

(1) EchoWord:

[Description]

This is an action that corresponds to the HI interrupt when no communication path has been established.

When *HDT:X is 0x5555 (request to establish communication path), the action corresponding to the subsequent HI interrupts is changed from EchoWord: to Interpreter:. If *HDT:X is other than 0x5555, a complement of 1 is written to HDT, and the action corresponding to the HI interrupt does not change.

[Parameters]

*HDT:X ... 0x5555 (request to establish communication path) or other data

[Return Value]

*HDT:X ... Complement of 1 corresponding to input value

(2) Interpreter

[Description]

This is an action that corresponds to the HI interrupt when a communication path is established.

A command is input for the higher 8 bits of *HDT:X, and an operand is input for the lower 8 bits.

Depending on the shift amount used to normalize the command to 32 bits, the operation jumps to the following functions.

Table 7-1. Shift Amount and Jump Destination

Shift Amount	Jump Destination
0	SelPCmd:
1	IDTagCmd:
2 to 7	Empty

Moreover, the user can freely define a function for the empty box in Table 7-1. In this case, write jmp xxxx (xxxx is the start address of the function) to MEDIAOS_HIF_ID_Vect_10 of mos_hapi.asm.

[Parameters]

*HDT:X ... Command, operand

[Return Value]

R0L ... Operand

(3) SelPCmd:

[Description]

The address of the jump destination function indicated in R0L is substituted to *_MHA_PCmdPtr:MEM, and the number of parameters of that function to *_MHA_nRstPCmdPrm:MEM, to change the action upon occurrence of the next HI interrupt from Interpreter: to PuttoCmdBuf:.

[Parameters]

R0L ... Operand

[Return Value]

*_MHA_PCmdPtr:MEM ... Address of jump destination function

*_MHA_nRstPCmdPrm:MEM ... Number of parameters of jump destination function

(4) PuttoCmdBuf:

[Description]

Stacks the *HDT value to the area with _MHA_CmdBuf_Base:MEM as its start.

When there have been HI interrupts *_MHA_nRstPCmdPrm:MEM times, the operation jumps to the *_MHA_PCmdPtr:MEM address after the action upon occurrence of the next HI interrupt is changed from PuttoCmdBuf: to Interpreter:.

Table 7-2. *_MHA_PCmdPtr:MEM Contents and Jump Destination

*_MHA_PCmdPtr: MEM Contents	Jump Destination	Number of Parameters
0	Reboot:	4
2	Direct_RX:	1
4	Direct_RY:	1
6	Direct_WX:	2
8	Direct_WY:	2
10	Copy_XtoX:	3
12	Copy_YtoY:	3
14	Copy_XtoY:	3
16	Copy_YtoX:	3

Moreover, users can freely define new functions in the part past 18 in Table 7-2. In this case, describe DW xxxx (xxxx is the start address of the function) and DW yyyy (yyyy is the number of parameters) to MEDIAOS_HIF_PCmdStruct of mos_hapi.asm.

[Parameters]

*_MHA_PCmdPtr:MEM … Address of jump destination function

*_MHA_nRstPCmdPrm:MEM … Number of parameters of jump destination function

[Return Value]

*_MHA_CmdBuf_Base+?:MEM … Stack data, depends on number of parameters

Remark ? = 0 to number of parameters

(5) IDTagCmd:

[Description]

Jumps to the user defined function indicated by R0L. The definition method is to describe jmp xxxx (xxxx is the start address of the function) to MEDIAOS_HIF_ID_Vect_20 of mos_hapi.asm. The instruction of the address indicated by ID_Tag_20XX:+R0L is executed.

[Parameters]

R0L ... Jump destination function ID

[Return Value]

There are no return values.

(6) Reboot:

[Description]

Sets *_MHA_CmdBuf_Base+2:MEM to DP7, *_MHA_CmdBuf_Base+0:MEM to DP3,
*_MHA_CmdBuf_Base+1:MEM to DP0, and *HDT:x to r71, and executes DP0.

Table 7-3. Call Addresses and Reboot Routines

Call Address	Reboot Routine
1	Y memory → Instruction memory word reboot
2	X memory → Instruction memory word reboot
3	Y memory → Instruction memory byte reboot
4	X memory → Instruction memory byte reboot
5	HDT → Instruction memory reboot

[Parameters]

*MHA_CmdBuf_Base+2:MEM ... dp7 value
*MHA_CmdBuf_Base+1:MEM ... Call address
*MHA_CmdBuf_Base+0:MEM ... dp3 value

[Return Value]

There are no return values.

(7) Direct_RX:

[Description]

Read the data of the *HDT:X address of X memory and writes it to *HDT:X.

[Parameters]

*HDT:X ... Read address

[Return Value]

*HDT:X ... Read data

(8) Direct_RY:

[Description]

Reads the data of the *HDT:X address of Y memory and writes it to *HDT:X.

[Parameters]

*HDT:X ... Read address

[Return Value]

*HDT:X ... Read data

(9) Direct_WX:

[Description]

Writes the *HDT:X value to the *_MHA_CmdBuf_Base+0:MEM of X memory.

[Parameters]

*_MHA_CmdBuf_Base+0:MEM ... Write address
*HDT:X ... Write data

[Return Value]

There are no return values.

(10) Direct_WY:

[Description]

Writes the *HDT:X value to the *_MHA_CmdBuf_Base+0:MEM address of Y memory.

[Parameters]

*_MHA_CmdBuf_Base+0:MEM ... Write address
*HDT:X ... Write data

[Return Value]

There are no return values.

(11) Copy_XtoX:

[Description]

Copies the data of the *HDT:X size area from the *_MHA_CmdBuf_Base+0:MEM address of X memory from the *_MHA_CmdBuf_Base+1:MEM address of the same memory to the *HDT:X size area.

[Parameters]

*MHA_CmdBuf_Base+1:MEM ... Source start address
*MHA_CmdBuf_Base+0:MEM ... Destination start address
*HDT:X ... Copy size

[Return Value]

There are no return values.

(12) Copy_YtoY:

[Description]

Copies the data of the *HDT:X size area from the *_MHA_CmdBuf_Base+0:MEM address of Y memory from the *_MHA_CmdBuf_Base+1:MEM address of the same memory to the *HDT:X size area.

[Parameters]

*_MHA_CmdBuf_Base+1:MEM+0 ... Source start address
*_MHA_CmdBuf_Base+0:MEM+1 ... Destination start address
*HDT:X ... Copy size

[Return Value]

There are no return values.

(13) Copy_XtoY:

[Description]

Copies the data of the *HDT:X size area from the *_MHA_CmdBuf_Base+0:MEM address of X memory from the *_MHA_CmdBuf_Base+1:MEM address of Y memory to the *HDT:X size area.

[Parameters]

*_MHA_CmdBuf_Base+1:MEM+0 ... Source start address
*_MHA_CmdBuf_Base+0:MEM+1 ... Destination start address
*HDT:X ... Copy size

[Return Value]

There are no return values.

(14) Copy_YtoX:

[Description]

Copies the data of the *HDT:X size area from the *_MHA_CmdBuf_Base+0:MEM address of Y memory from the *_MHA_CmdBuf_Base+1:MEM address of X memory to the *HDT:X size area.

[Parameters]

*_MHA_CmdBuf_Base+1:MEM+0 ... Source start address
*_MHA_CmdBuf_Base+0:MEM+1 ... Destination start address
*HDT:X ... Copy size

[Return Value]

There are no return values.

(15) ReadFromCBuf:

[Description]

Reads 1 data from the used area indicated by R0L of the write communication buffer, and substitutes it to R1L. This function is called directly from a subtask on the DSP.
Moreover, when this function is called by several tasks, it is necessary to prohibit interrupts used as a timer (interrupt defined with _USED_TimerINT of os_udef.h) beforehand.

The value specified by R0L is determined in the order that the used areas are prepared. If, a new used area is to be prepared following the deletion of a used area, that deleted value is allocated.
For example, if three used areas have been created with the CreateCBuf() API function, the creation order, 0, 1, 2, is used.

CreateCBuf(device,3,WRITE,&cbHndA); ... R0L = 0 if this used area is to be read
CreateCBuf(device,5,WRITE,&cbHndB); ... R0L = 1 if this used area is to be read
CreateCBuf(device,4,WRITE,&cbHndC); ... R0L = 2 if this used area is to be read

Then, even if the used area that was created second (cbHndB) is deleted, the values specified to R0L of the used areas created first and second remain 0 and 2.
FreeCBuf(cbHndB); ... If the buffers created first and third are to be read, R0L remains 0 and 2.

Then, if a new fourth used area is created, the value specified for R0L is 1.
CreateCBuf(device,6,WRITE,&cbHndD); ... If this buffer is to be read, R0L = 1

Moreover, if a new fifth used area is created, the value specified for R0L is 3.
CreateCBuf(device,7,WRITE,&cbHndE); ... If this buffer is to be read, R0L is 3.

In the same way, if a used area is deleted thereafter, the values specified until now for R0L do not change, and if a new used area is created thereafter, the vacant value is filled. If there is no vacant value, a new value is allocated.

Since the R0L value allocation matches the contents of cbHnd at the time the used area is created, it is possible to use that value to subtasks.
CreateCBuf(device,3,WRITE,&cbHndA); ... If this buffer is to be read, R0L = cbHndA, so that a program that in one way or another directs the subtasks must be created.

In the case of this function, the data corresponding to the used area must all be read. After this function has read the data corresponding to the used area, data write to that used area is enabled. In the following example, the contents of a size 5 used area are read and stored to the address displayed with dp0.

```
R0L=0 ;
Loop 0x5{
   Call ReadFromCBuf ;
   *dp0++=R1L ;
}
```

In the case of this function, if the data of the used area to be read has already been read, or if it has not been written, 1 is set to R2L as the return value and data read is not performed for the used area. In this case, it is not necessary to call this function the number of times corresponding to the used area size. In the following example, a loop is exited in case read is not possible.

```
CLR(R2L) ;
R0L=0 ;
Loop 0x5{
   Call ReadFromCBuf ;
   if(R2==0) jmp $+2
   LPOP ;
   *dp0++=R1L ;
   NOP ;
}
```

[Parameters]

R0L ... Write communication buffer handle

[Return Value]

R1L ... Read data

R2L ... 0: Normal termination, 1: Abnormal termination (data has already been read or data cannot be read because there is no data.)

(16) WriteToCBuf:

[Description]

Writes the contents of R1L to the used area indicated by R0L of the read communication buffer. This function is called directly from a subtask on the DSP.

Moreover, when this function is called by several tasks, it is necessary to prohibit interrupts used as a timer (interrupt defined with _USED_TimerINT of os_udef.h) beforehand.

The value specified by R0L is determined in the order that the used areas are prepared. If, a new used area is to be prepared following the deletion of a used area, that deleted value is allocated.

For example, if three read communication buffer used areas have been created with the CreateCBuf() API function, the creation order, 0, 1, 2, is used.

CreateCBuf(device,3,READ,&cHndA); ... R0L = 0 if this buffer is to be written

CreateCBuf(device,5,READ,&cHndB); ... R0L = 1 if this buffer is to be written

CreateCBuf(device,4,READ,&cHndC); ... R0L = 2 if this buffer is to be written

Then, even if the used area that was created second (cBHndB) is deleted, the values specified to R0L of the used areas created first and third remain 0 and 2.

FreeCBuf(dbHndB); ... If the buffers created first and third are to be written, R0L remains 0 and 2.

Then, if a new fourth used area is created, the value specified for R0L is 1.

CreateCBuf(device,6,READ,&cbHndD); ... If this buffer is to be written, R0L = 1

Moreover, if a new fifth used area is created, the value specified for R0L is 3.

CreateCBuf(device,7,READ,&cbHndE); ... If this buffer is to be written, R0L = 3

In the same way, if a used area is newly created thereafter without changing the value specified until now to R0L, the vacant value is filled. If there is no vacant value, a new value is allocated.

Since the above value allocations match the AND masked contents with 0x7fff to cbHnd at the time the used area is created, it is possible to use that value to subtasks.

CreateCBuf(device,3,READ,&cbHndA); ... If this buffer is to be written, R0L = cbHndA&0x7fff, so that a program that in one way or another directs the subtasks must be created.

In the case of this function, the data corresponding to the used area must all be read. After this function has read the data corresponding to the used area, data write to that used area is enabled. In the following example, the contents of a size 5 used area are read and stored to the address displayed with dp0.

```
R0L=0 ;
Loop 0x5{
   R1L=*dp0++ ;
   Call WriteToCBuf ;
}
```

In the case of this function, if the data of the used area to be read has not yet been read, 1 is set to R2L as the return value and data read is not performed. In this case, it is not necessary to call this function the number of times corresponding to the used area size. In the following example, a loop is exited in case read is not possible.

```
CLR(R2L) ;
R0L=0 ;
Loop 0x5{
   R1L=*dp0++ ;
   Call WriteToCBuf ;
   if(R2==0) jmp $+2
   LPOP ;
   NOP ;
   NOP ;
}
```

[Parameters]

R0L ... Read communication buffer handle

R1L ... Write data

[Return Value]

R2L ... 0: Normal termination, 1: Abnormal termination (Write cannot be performed because data that has been written previously remains.)

[MEMO]

CHAPTER 8 PREPARATION OF HOST API EXECUTION FILE

8.1 Preparation of Execution File Using API Functions

To prepare an execution file using API functions, compile user applications that call API functions with a C compiler, and link them with mos_hapi.c, spx.c object files (which must be compiled beforehand).

Moreover, user settings or a program adjusted to the target system must be prepared in the next file.

8.1.1 hapiucfg.h

hapiucfg.h is a user definition header file of mos_hapi.c, spx.c. The following settings adjusted to the target system are required.

(1) WAIT_TIME, TIM_CONST definition

#define WAIT_TIMExx ... Write a value for xx.
#define TIM_CONSTyy ... Write a value for yy.

WAIT_TIME and TIM_CONST definitions define the time to be waited until the host read enable flag or host write enable flag of the HST register changes to 1 (read, write enable), using functions that access the HDT register of the DSP within spx.c (inhdtw(), outhdtw()). The wait time is determined by WAIT_TIME * TIM_CONST. If the character constants WAIT_TIME and TIM_CONST in inhdtw(), outhdtw() are not used, these definition statements can be deleted.

(2) HDTAccess, HSTAccess, RSTAccess definitions

#define HDTAccess x ... Describe 0:Byte access or 1:Word access for x.
#define HSTAccess y ... Describe 0:Byte access or 1:Word access for y.
#define RSTAccess z ... Describe 0:Byte access or 1:Word access for z.

The HDTAccess, HSTAccess, and RSTAccess definitions define whether the HDT register, HST register, and RST pin are to be accessed using byte access or word access. If the HDTAccess, HSTAccess, RSTAccess character constants are not used for the functions accessing the HDT register, HST register, and RST pin (inhdtw(), outhdtw(), systemreset()), these definition statements can be deleted.

(3) MAX_SUB_TASK definition

#define MAX_SUB_TASK xx ... Write a value for xx.

The MAX_SUB_TASK definition defines the maximum number of used subtasks. The defined value can be larger than the number of subtasks that is actually used, but if it is smaller, the SUBTASKBL_FULL error occurs upon execution of LoadInformationFile().

(4) NUM_HICBUF, NUM_HOCBUF definitions

#define NUM_HICBUF xx ... Write a value for xx.
#define NUM_HOCBUF yy ... Write a value for yy.

NUM_HICBUF and NUM_HOCBUF definitions define the number of used areas for the write communication buffer and read communication buffer, respectively, that can be prepared. The defined value can be larger than the number of communication buffers that are actually prepared, but if it is smaller, the CBTBL_FUL error occurs upon execution of CreateCBuf().

8.1.2 spx.c

spx.c is the source file for the interface between API functions and the DSP. Regarding the functions listed below, a program adjusted to the target system must be prepared.

Word inhdtw(Config cmConfig)	...	Function to write data to HDT
WORD outhdtw(Config cmCOnfig, WORD dataword)	...	Function to read data from HDT
void systemreset(Config cmConfig)	...	Function to perform system reset

8.2 Preparation of Execution File Using BIOS Functions

To prepare an execution file that uses BIOS functions, insert the necessary code to the initialization block of the target system of the RX77016 (_MOS_TargetSysInit:) and HI interrupt handler code block (_MOS_ivHI:), and following assembly in the WB77016, perform linking of the RX77016 with the bios_fns.asm object file (which must be assembled beforehand), in order to create the execution file.

The blocks necessary for inserting code or changing definition are as follows.

8.2.1 os_udef.h

(1) _USED_ivHI definition

```
#DEFINE _USED_ivHI  TRUE          ; HI   (ITN9)  TRUE:Used  FALSE:Not Used
```

The _USED_ivHI definition must be TRUE.

(2) _IVAL_EIR, _IVAL_SRorMASK, _IVAL_SRandMASK definitions

```
#DEFINE _IVAL_EIR   xx            ; EIR
#DEFINE _IVAL_SRorMASK  xx        ; OR mask value for SR.
#DEFINE _IVAL_SRandMASK  xx       ; AND mask value for SR.
```

The _IVAL_EIR, _IVAL_SRorMASK, and _IVAL_SRandMASK definitions must be definitions that enable HI interrupts.

8.2.2 Target system initialization block (_MOS_TargetSysInit:)

In the target system initialization block, the HDT access wait enable bit of the HST register is set and %InitHiState(EchoWord) is executed. Moreover, biosucfg.h, bios_fns.h, and bios_mac.h must be included prior to %InitHiState(EchoWord). Next, a description example for the execution of %InitHiState(EchoWord) is shown.

```
R0L = *HST:x ;
R0 = R0 | 0x0400 ;
*HST:x = R0L ;

%InitHiState(EchoWord) ;
```

8.2.3 HI interrupt handler code block (_MOS_ivHI:)

In the HI interrupt handler code block (_MOS_ivHI:), the following 4 states are described. Moreover, biosucfg.h and bios_fns.h must be included prior to r0l=*_MHA_MdRg_In:MEM.

```
clr(r0) ;
r0l = *_MHA_MdRg_In:MEM ;
dp0 = r0l;
jmp dp0;;
```

8.2.4 biosucfg.h

biosucfg.h is a header file for the initialization block (_MOS_TargetSysInit:) of the target system of the RX77016, HI interrupt handler code block (_MOS_ivHI:), and bios_fns.asm, and the following settings adjusted to the target system are required.

(1) _MHA_DATAMEM definition

#define_MHA_DATAMEMx ... Write 0:xmem or 1:ymem for x.

In the _MHA_DATAMEM definition, allocations to xmem or ymem of the data area required for execution of the communication buffer information and heap area, and other BIOS functions can be done.

(2) HAVE_EXRAM definition

#define HAVE_EXRAMx ... Write 0:Internal RAM or 1:External RAM for x.

In the HAVE_EXRAM definition, allocation of all instructions and data required for execution of BIOS functions can be done to internal RAM or external RAM.

(3) CMD_BUF_SIZE definition

#define CMD_BUF_SIZEx ... Write a value for x.

In the CMD_BUF_SIZE definition, the stack area size of the parameters used in PuttoCmdBuf: is defined. Describe the maximum number of parameter stacks required for the execution of BIOS functions.

(4) HICBUF_HEAP_SIZE, NUM_HICBUF definitions

#define HICBUF_HEAP_SIZEx ... Write a value for x.
#define NUM_HICBUFy ... Write a value for y.

The HICBUF_HEAP_SIZE and NUM_HICBUF definitions define the write communication buffer heap area size and the number of buffers that can be prepared.

(5) HOCBUF_HEAP_SIZE, NUM_HOCBUF definitions

#define HOCBUF_HEAP_SIZEx ... Write a value for x.
#define NUM_HOCBUFy ... Write a value for y.

The HOCBUF_HEAP_SIZE and NUM_HOCBUF definitions define the read communication buffer heap area size and the number of buffers that can be prepared.

NEC

Facsimile Message

From:

Name

Company

Tel. FAX

Address

Although NEC has taken all possible steps to ensure that the documentation supplied to our customers is complete, bug free and up-to-date, we readily accept that errors may occur. Despite all the care and precautions we've taken, you may encounter problems in the documentation. Please complete this form whenever you'd like to report errors or suggest improvements to us.

Thank you for your kind support.

North America
NEC Electronics Inc.
Corporate Communications Dept.
Fax: 1-800-729-9288
1-408-588-6130

Europe
NEC Electronics (Europe) GmbH
Technical Documentation Dept.
Fax: +49-211-6503-274

South America
NEC do Brasil S.A.
Fax: +55-11-6462-6829

Hong Kong, Philippines, Oceania
NEC Electronics Hong Kong Ltd.
Fax: +852-2886-9022/9044

Korea
NEC Electronics Hong Kong Ltd.
Seoul Branch
Fax: 02-528-4411

Taiwan
NEC Electronics Taiwan Ltd.
Fax: 02-2719-5951

Asian Nations except Philippines
NEC Electronics Singapore Pte. Ltd.
Fax: +65-250-3583

Japan
NEC Semiconductor Technical Hotline
Fax: 044-435-9608

I would like to report the following error/make the following suggestion:

Document title: ______

Document number: ______ Page number: ______

If possible, please fax the referenced page or drawing.

Document Rating	Excellent	Good	Acceptable	Poor
Clarity	❏	❏	❏	❏
Technical Accuracy	❏	❏	❏	❏
Organization	❏	❏	❏	❏

CS 00.6