To our customers,

Old Company Name in Catalogs and Other Documents

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

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

April 1st, 2010
Renesas Electronics Corporation

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

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

Notice

1. All information included in this document is current as of the date this document is issued. Such information, however, is subject to change without any prior notice. Before purchasing or using any Renesas Electronics products listed herein, please confirm the latest product information with a Renesas Electronics sales office. Also, please pay regular and careful attention to additional and different information to be disclosed by Renesas Electronics such as that disclosed through our website.
2. Renesas Electronics does not assume any liability for infringement of patents, copyrights, or other intellectual property rights of third parties by or arising from the use of Renesas Electronics products or technical information described in this document. No license, express, implied or otherwise, is granted hereby under any patents, copyrights or other intellectual property rights of Renesas Electronics or others.
3. You should not alter, modify, copy, or otherwise misappropriate any Renesas Electronics product, whether in whole or in part.
4. Descriptions of circuits, software and other related information in this document are provided only to illustrate the operation of semiconductor products and application examples. You are fully responsible for the incorporation of these circuits, software, and information in the design of your equipment. Renesas Electronics assumes no responsibility for any losses incurred by you or third parties arising from the use of these circuits, software, or information.
5. When exporting the products or technology described in this document, you should comply with the applicable export control laws and regulations and follow the procedures required by such laws and regulations. You should not use Renesas Electronics products or the technology described in this document for any purpose relating to military applications or use by the military, including but not limited to the development of weapons of mass destruction. Renesas Electronics products and technology may not be used for or incorporated into any products or systems whose manufacture, use, or sale is prohibited under any applicable domestic or foreign laws or regulations.
6. Renesas Electronics has used reasonable care in preparing the information included in this document, but Renesas Electronics does not warrant that such information is error free. Renesas Electronics assumes no liability whatsoever for any damages incurred by you resulting from errors in or omissions from the information included herein.
7. Renesas Electronics products are classified according to the following three quality grades: "Standard", "High Quality", and "Specific". The recommended applications for each Renesas Electronics product depends on the product's quality grade, as indicated below. You must check the quality grade of each Renesas Electronics product before using it in a particular application. You may not use any Renesas Electronics product for any application categorized as "Specific" without the prior written consent of Renesas Electronics. Further, you may not use any Renesas Electronics product for any application for which it is not intended without the prior written consent of Renesas Electronics. Renesas Electronics shall not be in any way liable for any damages or losses incurred by you or third parties arising from the use of any Renesas Electronics product for an application categorized as "Specific" or for which the product is not intended where you have failed to obtain the prior written consent of Renesas Electronics. The quality grade of each Renesas Electronics product is "Standard" unless otherwise expressly specified in a Renesas Electronics data sheets or data books, etc.

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

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

   "Specific": Aircraft; aerospace equipment; submersible repeaters; nuclear reactor control systems; medical equipment or systems for life support (e.g. artificial life support devices or systems), surgical implantations, or healthcare intervention (e.g. excision, etc.), and any other applications or purposes that pose a direct threat to human life.
8. You should use the Renesas Electronics products described in this document within the range specified by Renesas Electronics, especially with respect to the maximum rating, operating supply voltage range, movement power voltage range, heat radiation characteristics, installation and other product characteristics. Renesas Electronics shall have no liability for malfunctions or damages arising out of the use of Renesas Electronics products beyond such specified ranges.
9. Although Renesas Electronics endeavors to improve the quality and reliability of its products, semiconductor products have specific characteristics such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Further, Renesas Electronics products are not subject to radiation resistance design. Please be sure to implement safety measures to guard them against the possibility of physical injury, and injury or damage caused by fire in the event of the failure of a Renesas Electronics product, such as safety design for hardware and software including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other appropriate measures. Because the evaluation of microcomputer software alone is very difficult, please evaluate the safety of the final products or system manufactured by you.
10. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. Please use Renesas Electronics products in compliance with all applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive. Renesas Electronics assumes no liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.
11. This document may not be reproduced or duplicated, in any form, in whole or in part, without prior written consent of Renesas Electronics.
12. Please contact a Renesas Electronics sales office if you have any questions regarding the information contained in this document or Renesas Electronics products, or if you have any other inquiries.

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

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

User's Manual

RENESAS

CB38 V.1.00

Programming Manual

Custom Builder for Emulator Debugger PD38

Renesas Electronics
www.renesas.com

Rev.1.00 2003.05

Keep safety first in your circuit designs!

- Renesas Technology Corporation and Renesas Solutions Corporation put the maximum effort into making semiconductor products better and more reliable, but there is always the possibility that trouble may occur with them. Trouble with semiconductors may lead to personal injury, fire or property damage. Remember to give due consideration to safety when making your circuit designs, with appropriate measures such as (i) placement of substitutive, auxiliary circuits, (ii) use of nonflammable material or (iii) prevention against any malfunction or mishap.

Notes regarding these materials

- These materials are intended as a reference to assist our customers in the selection of the Renesas Technology product best suited to the customer's application; they do not convey any license under any intellectual property rights, or any other rights, belonging to Renesas Technology Corporation, Renesas Solutions Corporation or a third party.
- Renesas Technology Corporation and Renesas Solutions Corporation assume no responsibility for any damage, or infringement of any third-party's rights, originating in the use of any product data, diagrams, charts, programs, algorithms, or circuit application examples contained in these materials.
- All information contained in these materials, including product data, diagrams, charts, programs and algorithms represents information on products at the time of publication of these materials, and are subject to change by Renesas Technology Corporation and Renesas Solutions Corporation without notice due to product improvements or other reasons. It is therefore recommended that customers contact Renesas Technology Corporation, Renesas Solutions Corporation or an authorized Renesas Technology product distributor for the latest product information before purchasing a product listed herein. The information described here may contain technical inaccuracies or typographical errors. Renesas Technology Corporation and Renesas Solutions Corporation assume no responsibility for any damage, liability, or other loss rising from these inaccuracies or errors. Please also pay attention to information published by Renesas Technology Corporation and Renesas Solutions Corporation by various means, including the Renesas home page (http://www.renesas.com).
- When using any or all of the information contained in these materials, including product data, diagrams, charts, programs, and algorithms, please be sure to evaluate all information as a total system before making a final decision on the applicability of the information and products. Renesas Technology Corporation and Renesas Solutions Corporation assume no responsibility for any damage, liability or other loss resulting from the information contained herein.
- Renesas Technology semiconductors are not designed or manufactured for use in a device or system that is used under circumstances in which human life is potentially at stake. Please contact Renesas Technology Corporation, Renesas Solutions Corporation or an authorized Renesas Technology product distributor when considering the use of a product contained herein for any specific purposes, such as apparatus or systems for transportation, vehicular, medical, aerospace, nuclear, or undersea repeater use.
- The prior written approval of Renesas Technology Corporation and Renesas Solutions Corporation is necessary to reprint or reproduce in whole or in part these materials.
- If these products or technologies are subject to the Japanese export control restrictions, they must be exported under a license from the Japanese government and cannot be imported into a country other than the approved destination. Any diversion or reexport contrary to the export control laws and regulations of Japan and/or the country of destination is prohibited.
- Please contact Renesas Technology Corporation or Renesas Solutions Corporation for further details on these materials or the products contained therein.

For inquiries about the contents of this document or product, fill in the text file the installer generates in the following directory and email to your local distributor.

¥SUPPORT¥Product-name¥SUPPORT.TXT

Renesas Tools Homepage http://www.renesas.com/en/tools

1. Overview

1.1 Outline of this manual

This manual describes how to write a program when creating custom command or custom window programs using CB38. For details on how to use CB38, please refer to the "CB38 V.1.00 User's Manual."

1.2 What is CB38?

CB38 is an entirely new development environment that allows you to create your exclusive commands or windows that operate on PD38, the emulator debugger for the 740 Family.

1.3 What can be done with CB38?

By using CB38 you can easily create (program) custom commands and custom windows for PD38.

CB38 provides an integrated support for all operations from programming custom commands and windows to compiling and debugging them.

The custom commands and windows thus created can be used readily on PD38.

In short, CB38 lets you upgrade the functions of PD38 and customize it easily for yourself.

Since the custom commands and windows created using CB38 can control the emulators (PC4701HS/L) directly, various debug functions you may wish to have such as those listed below can easily be obtained.

- Reference and modify target memory contents
- Reference and analyze real-time trace data
- Control target program execution by running and stopping program and single-stepping source lines
- Build automatic target system testing environment

1.4 Features of CB38

1. A window design similar to that of PD38 is adopted for operational integrity with PD38.
2. An integrated development environment for programming, compiling, and debugging is provided.
3. The commands and windows that operate on PD38 can be created by yourself.
4. The program description language supported for CB38 is the C-language subset.
5. Various libraries like those listed below are available for CB38:
   - Standard function library (stdlib.lib)
   - Emulator operating function library (system.lib)
   - Window operating function library (winlib.lib)

2. Custom Command Programming

This chapter describes how to program the custom commands of PD38.

2.1 Procedure for creating custom commands

To create a custom command using CB38, follow the procedure described below.

1. Creating a project

   A project is a set of the source programs necessary to create custom commands. Create one project for one custom command to be created. For details on how to create a project, refer to the "CB38 V.1.00 User's Manual," Section 3.1.1, "Creating New Project for Custom Command Program".

2. Creating source programs

   Write the operation of a custom command in a source file. For details on how to create a source file, refer to the "CB38 V.1.00 User's Manual," Section 3.1.2, "Creating New Source File". For details on how to add the source file to a project, refer to the "CB38 V.1.00 User's Manual," Section 3.1.3, "Adding Source File to Project".

3. Building a command program

   The term "build" refers to creating a custom command program by compiling the source programs created above. For details about this operation, refer to the "CB38 V.1.00 User's Manual," Section 3.1.4, "Building a Program".

4. Debugging a command program

   If the custom command program created does not work as intended, debug it. For details on how to debug, refer to the "CB38 V.1.00 User's Manual," Section 2.1, "CB38 Window".

5. Adding custom command to PD38

   To use the custom command thus completed, add it to PD38. For details on how to add, refer to "Customize Functions" in the "PD38 V.2.00 User's Manual."

Described in this manual is the method for programming in 2, "Creating source programs" outlined above. For other details, refer to the corresponding sections in the "CB38 V.1.00 User's Manual."

2.2 Example of source program for the simplest custom command

This section explains the method of programming with CB38 by using a source program for the simplest custom command as an example.

- Example of custom command

Command name	hello
Format	hello address <RET>
Content	● Display a character string "Hello CB38 World!" in the script window. ● Then input the character string from the script window. ● After entering the character string, store it at the address that is specified in the first parameter of the command. ● If any error occurs during processing, terminate the command.

- Example of source program

```
#include <stdlib.h>
#include <system.h>

int main(int argc, char **argv)                 /* 1. Program is executed */
                                                /* from main() function */
{
        char    str[128];
        int     val, i, len;

        if(argc != 2){                          /* 2. Command is terminated */
                                                /* if one parameter is nonexistent */
                exit(1);
        }
        if(_exp_eval(argv[1], EXP_DEFAULT, EXP_LABEL, &val) == FALSE){
                                                /* 3. Assembler expression is */
                                                /* analyzed to get value */
                exit(1);
        }
        printf("Hello CB38 World!¥n");          /* 4. Character string is output */
                                                /* to script window */
        if(gets(str) == NULL){                  /* 5. Character string is input */
                                                /* from script window */
                exit(1);
        }
        len = strlen(str);
        for(i = 0; i < len; i++){
                                                /* 6. Memory contents are modified */
                if(_mem_put(val + i, 1, &(str[i])) == FALSE){
                        exit(1);
                }
        }
        exit(0);
}
```

- Explanation

1. The source program of the custom command begins from the main() function. The programming language used is the C-language subset specifically designed for use in CB38. Specifications of this language are detailed in the "CB38 V.1.00 User's Manual," Section 4, "Programming Language Specifications". The major differences with the C language are as follows:
    - Aggregates (structures and unions) are not supported.
    - Real types (float and double) are not supported.

   Stored in argc is the number of arguments, and what is stored in argv is the address that contains the pointer array that contains a pointer to the area at which the character string specified in the argument is stored. This is the same as the arguments of the main() function in the standard C language are handled.

   Note that although the main() function is the only function used in this example, multiple user-defining functions can be used in the same way as in the C language.

2. To quit the command in the middle, use the exit() function. Specifications of this function are detailed in the "CB38 V.1.00 User's Manual," Section 5.1.9, "exit: Terminate program execution".

3. To analyze an assembler expression to get a value, use the _exp_eval() function. Labels and symbols can be used in the expressions analyzed by this function. Specifications of this function are detailed in the "CB38 V.1.00 User's Manual," Section 5.2.59, "_exp_eval: Analyze assembler expression".

4. To display a character string in the script window, use the printf() function. Specifications of this function are detailed in the "CB38 V.1.00 User's Manual," Section 5.1.17, "printf: Output characters with format (to Script Window)".

5. To input a character string from the script window, use the gets() function. This function is used in almost the same way as the gets() function in the C language. Specifications of this function are detailed in the "CB38 V.1.00 User's Manual," Section 5.1.8, "gets: Input character string (from Script Window)".

6. To set a value in the target CPU memory, use the _mem_put() function. Specifications of this function are detailed in the "CB38 V.1.00 User's Manual," Section 5.2.189, "_mem_put: Set memory value".

2.3 Functions that can be used in programming a custom command

The functions that can be used in programming a custom command can broadly be classified into the following two groups.

1. Standard functions
   The functions similar to the standard C-language functions that are assumed to be relatively frequently used are supported.

2. Debugger operating functions
   The functions necessary to operate the debugger are supported.

2.4 Method for using the standard functions

When using the standard functions, include the header file “stdlib.h” in the function you are going to use. Specifications of the standard functions are detailed in the "CB38 V.1.00 User's Manual," Section 5.1, "Standard Functions (stdlib.lib)".

2.4.1 Using heap area manipulating functions

This section explains how to use the functions for manipulating the heap area by using the function shown below as an example.

Function name	Description
malloc	Allocate memory from heap area

[Program example]

```
char *regist_name(char *name)
{
        char        *p;
        int         len;

        if(name != NULL){                       /* This line is processed only when “name” is */
                                                /* not NULL */
                len = strlen(name);             /* Get length of character string */
                p = malloc(len + 1);            /* Allocate area of len + 1 bytes */
                if(p == NULL){                  /* Memory allocation failed if “p” is NULL */
                        return NULL;            /* Error */
                }
                strcpy(p, name);                /* Duplicate character string */
                return p;                       /* Return stored area */
        }
        return NULL;                            /* Error */
}
```

Shown above is a program example used to create a user-defined function that stores the character string specified by the argument “name” in the heap area using the malloc() function.

2.4.2 Using character string manipulating functions

This section explains how to use the functions for manipulating character strings by using the functions shown below as an example.

Function name	Description
strcmp	Compare character strings
strtoi	Convert character string into value
sprintf	Output character string with format (to memory)

[Program example]

```
int eval_str(char *str1, char *str2, char *str3)
{
        int     value;

        if(strcmp(str1, "go") == 0){            /* When str1 is "go" */
                if(strtoi(str2, 0, &value) == TRUE){
                                                /* When str2 was converted into value */
                                                /* Output to str3 with format included */
                        sprintf(str3, "%06X(%d)", value, value);
                        return TRUE;    /* Succeeded */
                }
        }
        return FALSE;                           /* Error */
}
```

Shown above is a program example used to create a user-defined function that when the argument str1 is "go," converts the numeral-representing character string specified by the argument str2 into a numeric value and outputs it to the area specified by the argument str3 with a format included.

2.4.3 Using input/output functions

This section explains how to use the input/output functions by using the functions shown below as an example.

Function name	Description
gets	Input character string (from Script Window)
printf	Output character string with format (to Script Window)

[Program example]

```
int echo_str()
{
        char    str[1024];

        if(gets(str) != NULL){                          /* Character string was obtained */
                printf("Your input is [%s].¥n", str);   /* Output with format included */
                return TRUE;                            /* Succeeded */
        }
        return FALSE;                                   /* Error */
}
```

Shown above is a program example used to create a user-defined function that outputs the character string entered in the input area of the Script Window to the window's display area with a format included.

2.4.4 Using file manipulating functions

This section explains how to use the file manipulating functions by using the functions shown below as an example.

Function name	Description
fopen	Open file
fclose	Close file
fprintf	Output data with format (to file)

[Program example]

```
int put_file(char *filename, int data1, int data2, int data3, int data4)
{
        int         fd;

        if((fd = fopen(filename, "w")) == NULL){        /* Open file */
                return FALSE;                           /* Error */
        }
        fprintf(fd, "Data1 = %d¥n", data1);             /* Output data1 */
        fprintf(fd, "Data2 = %d¥n", data2);             /* Output data2 */
        fprintf(fd, "Data3 = %d¥n", data3);             /* Output data3 */
        fprintf(fd, "Data4 = %d¥n", data4);             /* Output data4 */
        fclose(fd);                                     /* Close file */
        return TRUE;                                    /* Succeeded */
}
```

Shown above is a program example used to create a user-defined function that creates the file specified by the argument "filename" and outputs the data specified by arguments "data1" through "data4" to the file with a format included.

2.5 Method for using the debugger operating functions

When using the debugger operating functions, include the header file system.h in the function you are going to use. Specifications of the debugger operating functions are detailed in the "CB38 V.1.00 User's Manual," Section 5.2, "System Call Functions for Debugger Operation (system.lib)".

2.5.1 Using execution controlling functions

This section explains how to use the functions or controlling program execution by using the functions shown below as an example.

Function name	Description
_cpu_go	Execute program free-run
_cpu_stop	Stop program execution
_cpu_reset	Reset target system

[Program example]

```
int go_stop_10()
{
        int         i;

        for(i = 0; i < 10; i++){                        /* Repeat 10 times */
                if(_cpu_go() == FALSE){                 /* Execute program in */
                                                        /* free-run mode */
                        return FALSE;                   /* Error in the above */
                }
                if(_cpu_stop() == FALSE){               /* Stop program execution */
                        return FALSE;                   /* Error in the above */
                }
        }
        if(_cpu_reset() == FALSE){                      /* Reset target system */
                return FALSE;                           /* Error in the above */
        }
        return TRUE;                                    /* Succeeded */
}
```

Shown above is a program example used to create a user-defined function that repeats program execution and stopping in free-run mode 10 times before resetting the target system.

2.5.2 Using register manipulating functions

This section explains how to use the functions for manipulating registers by using the functions shown below as an example.

Function name	Description
_reg_get_pc	Get program counter value
_reg_put_reg	Set register value

[Program example]

```
int pc_inc_intb()
{
        int         reg;

        if(_reg_get_pc(&reg) == FALSE){                    /* Get PC value */
                return FALSE;                              /* Error in the above */
        }
        reg = (reg & 0xff) + 1;                            /* Low-order 8 bits + 1 */
        if(_reg_put_reg(reg, IN1_REG_1_A) == FALSE){/* Set value in table A */
                return FALSE;                              /* Error in the above */
        }
        return TRUE;                                       /* Succeeded */
}
```

Shown above is a program example used to create a user-defined function that sets the current value of the program counter in the table A register after incrementing the value of its 8 low-order bits by 1.

2.5.3 Using memory manipulating functions

This section explains how to use the memory manipulating functions by using the functions shown below as an example.

Function name	Description
_mem_get	Get memory value
_mem_put	Set memory value

[Program example]

```
int inc_1000H()
{
        char        data[128];
        int         i;

        if(_mem_get(0x1000, 128, data) == FALSE){      /* Get 128 bytes beginning with */
                                                       /* address 1000H */
                return FALSE;                          /* Error in the above */
        }
        for(i = 0; i < 128; i++){                      /* Repeat 128 times */
                (data[i])++;                           /* Increment data */
        }
        if(_mem_put(0x1000, 128, data) == FALSE){      /* Set 128 bytes beginning with */
                                                       /* address 1000H */
                return FALSE;                          /* Error in the above */
        }
        return TRUE;                                   /* Succeeded */
}
```

Shown above is a program example used to create a user-defined function that increments 128 bytes of memory values beginning with address 1000H by 1.

2.5.4 Using software break manipulating functions

This section explains how to use the functions for manipulating software breaks by using the functions shown below as an example.

Function name	Description
_break_set	Set/enable software break
_break_reset	Clear software break

[Program example]

```
int go_F000H()
{
        if(_break_set(0xF000) == FALSE){      /* Set software break at address F000H */
                return FALSE;                 /* Error in the above */
        }
        if(_cpu_gb() == FALSE){               /* Execute program with break */
                return FALSE;                 /* Error in the above */
        }
        _cpu_wait();                          /* Wait until target execution is stopped */
        if(_break_reset(0xF000) == FALSE){    /* Clear software break at address F000H */
                return FALSE;                 /* Error in the above */
        }
        return TRUE;                          /* Succeeded */
}
```

Shown above is a program example used to create a user-defined function that executes a program until it is stopped at address F000H.

2.5.5 Using debug information manipulating functions

This section explains how to use the functions for manipulating debug information by using the functions shown below as an example.

Function name	Description
_line_addr2line	Get source line of indicated address
_exp_eval	Analyze assembler expression
_c_exp_eval	Analyze C-language expression

[Program example]

```
int str_eval(char *str, int *is_c, char *filename, int *line, int *find_line)
{
        int     value, val;
        char    s1[128], s2[128], s3[128];

        if(_exp_eval(str, EXP_DEFAULT, EXP_LABEL, &value) == TRUE){
                *is_c = FALSE;                  /* Assembler expression */
        }else if(_c_exp_eval(str, &value, &val, s1, s2, s3) == TRUE){
                *is_c = TRUE;                   /* C-language expression */
        }else{
                return FALSE;                   /* Error in analyzing expression */
        }
        if(_line_addr2line(value, line, filename) == TRUE){
                *find_line = TRUE;              /* Source file name and line number */
                                                /* found */
        }else{
                *find_line = FALSE;             /* Source file name and line number */
                                                /* nonexistent */
        }
        return TRUE;
}
```

Shown above is a program example used to create a user-defined function that determines whether the character string specified by the argument str is an assembler or a C-language expression and gets the source file name and line number that corresponds to the address obtained by analyzing the expression.

2.5.6 Using scrip command executing functions

This section explains how to use the functions for executing script commands by using the functions shown below as an example.

Function name	Description
_syscom	Execute PD38's script command

[Program example]

```
int DB(int addr)
{
        char    str[128];

        sprintf(str, "DumpByte %X", addr);      /* Create script command character */
                                                /* string */
        if(_syscom(str) == FALSE){              /* Execute script command */
                return FALSE;                   /* Error in the above */
        }
        return TRUE;                            /* Succeeded */
}
```

Shown above is a program example used to create a user-defined function that executes a DumpByte script command using the address specified by the argument addr as the first argument.

2.5.7 Using DOS command executing functions

This section explains how to use the functions for executing DOS commands by using the functions shown below as an example.

Function name	Description
_doscom	Execute DOS command

[Program example]

```
int CP(char *src, char *dest)
{
        char    str[256];

        sprintf(str, "copy %s¥¥*.* %s¥¥*.*", src, dest);  /* Create DOS command character */
                                                          /* string */
        if(_doscom(str) == FALSE){                        /* Execute DOS command */
                return FALSE;                             /* Error in the above */
        }
        return TRUE;                                      /* Succeeded */
}
```

Shown above is a program example used to create a user-defined function that executes a DOS command to copy a file from the directory specified by the argument src to the directory specified by the argument dest.

3. Custom Window Programming

This chapter explains how to program the custom windows of PD38.

3.1 Procedure for creating a custom window

To create a custom window using CB38, follow the procedure described below.

1.Creating a project

A project is a set of the source programs necessary to create custom windows. Create one project for one custom window to be created. For details on how to create a project, refer to the "CB38 V.1.00 User's Manual," Section 3.2.1, "Creating New Project for Custom Window Program".

2.Creating source programs

Write the operation of a custom window in the framework source file that is automatically generated by CB38 when creating a project. For details on how to edit the framework source file, refer to the "CB38 V.1.00 User's Manual," Section 3.2.2, "Editing Automatically Created Framework Source File".

3.Building a window program

The term "build" refers to creating a custom window program by compiling the source programs created above. For details about this operation, refer to the "CB38 V.1.00 User's Manual," Section 3.1.4, "Building a Program".

4.Debugging a window program

If the custom window program created does not work as intended, debug it. For details on how to debug, refer to the "CB38 V.1.00 User's Manual," Section 2.1, "CB38 Window".

5.Adding custom window to PD38

To use the custom window thus completed, add it to PD38. For details on how to add, refer to "Customize Functions" in the "PD38 V.2.00 User's Manual."

Described in this manual is the method for programming in 2, "Creating source programs" outlined above. For other details, refer to the corresponding sections in the "CB38 V.1.00 User's Manual."

3.2 Example of source program for the simplest custom window

This section explains the method of programming with CB38 by using a source program for the simplest custom window as an example.

- Example of custom window

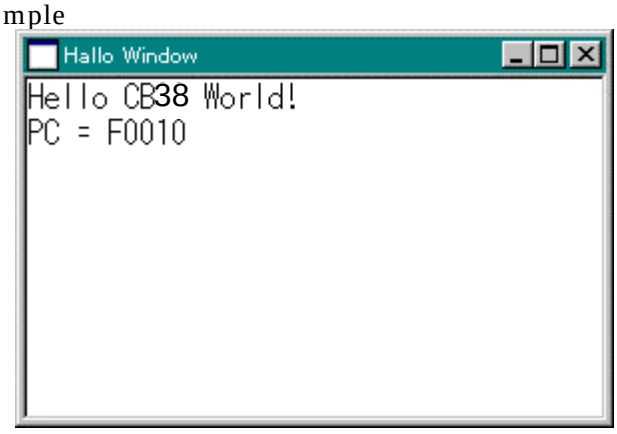
Window name	Hello Window
Content	● Display "Hello Window" in title ● Window size is 300 x 200 pixels ● Display characters "Hello CB38 World!" and start a new line ● Get current PC value and display it in hex after "PC = "

- Example of source program (excerpt from framework source file)

```
OnCreate()                    /* 1. OnCreate() function in framework source file */
{
        /* Write message handler code here, please. */
                /* 2. Display "Hello Window" in title */
        _win_set_window_title("Hallo Window");
                /* 3. Set window size to 300 x 200 (pixels) */
        _win_set_window_size(300, 200);
}

OnDraw()              /* 4. OnDraw() function in framework source file */
{
        /* Write message handler code here, please. */
        int         pc;
                /* 5. Set drawing start position at (0, 0) (cursor coordinates) */
        _win_set_cursor(0, 0);
                /* 6. Display "Hello CB38 World!" in window */
        _win_printf("Hello CB38 World!¥n");
        _reg_get_pc(&pc);
                /* 7. Display PC value in window */
        _win_printf("PC = %05X¥n", pc);
}
```

The lines printed in plain style are the codes automatically generated in the framework source file by CB38. The lines printed in bold face are the codes added by the user.

- Execution example



- Explanation

1. The source file (called the "framework source file") automatically generated by CB38 when creating a custom window project contains several functions written in it beforehand. These functions are special ones that are automatically called by PD38 when any operation is performed on the custom window. These functions are called the "handle functions."

   The OnCreate() handle function is called immediately before creating a window in order to initialize the window frame size, title, and necessary variables. This handle function is executed first of all functions when you start up a custom window program. Specifications of this function are detailed in the "CB38 V.1.00 User's Manual," Section 5.4.4, "OnCreate Handle Function".

2. To set the title of a custom window, use the _win_set_window_title() function. Specifications of this function are detailed in the "CB38 V.1.00 User's Manual," Section 5.3.23, "_win_set_window_title: Set custom window title".

3. To set the size of a custom window, use the _win_set_window_size() function. Specifications of this function are detailed in the "CB38 V.1.00 User's Manual," Section 5.3.38, "_win_set_window_size: Set custom window size".

4. The OnDraw() handle function is called when, for example, displaying part (or the whole) of a window that is hidden behind some other window in order to redraw the window. Specifications of this function are detailed in the "CB38 V.1.00 User's Manual," Section 5.4.6, "OnDraw Handle Function".

5. To set the cursor position, use the _win_set_cursor() function. Specifications of this function are detailed in the "CB38 V.1.00 User's Manual," Section 5.3.3, "_win_set_cursor: Set cursor position".

6. To output a character string at the current cursor position of the window, use the _win_printf() function. In this case, the cursor is moved to a position next to the last character that is output. Specifications of this function are detailed in the "CB38 V.1.00 User's Manual," Section 5.3.1, "_win_printf: Output character string with format (to Custom Window)".

3.3 About the handle functions

A custom window functions as one of PD38 windows. Therefore, a custom window exchanges information with PD38 and the OS as it goes on operating.

When an operation is performed on the custom window or an elapsed time is notified by the system timer, PD38 calls the corresponding handle function of the custom window program for the operation performed or the notification received. The processing written in a handle function is executed when such an operation is performed on the custom window that requires calling the handle function.

All handle functions do not have an argument. Nor are their returned values evaluated by PD38.

For the OnMouseMove() handle function that is called when the mouse is moved, and the OnSize() handle function that is called when the window size is changed, for example, the data showing the current mouse position and the window size before the change are stored in the area indicated by a global variable _HandleMsgBlock located in a library immediately before calling the handle function.

By referencing the data stored in _HandleMsgBlock within the handle function, it is possible to get the information associated with the operation performed or the notification received. The procedure for getting such information is automatically written in the framework source file.

For details about the handle functions, refer to the "CB38 V.1.00 User's Manual," Section 5.4, "Handle Functions for Custom Window".

[Precaution] Since the handle functions are special functions called by PD38**, do not try to call them freely like a user-defined function. This is because such an operation could make it impossible for** PD38 **to call them correctly.**

3.4 About framework source file

CB38 automatically generates a source file (i.e., the framework source file) when creating a project. All of the handle functions called by PD38 and the procedures for acquiring data are written in this file.

Among such procedures written in the framework source file are the codes to get the information associated with the operation performed on the custom window or the received notification from _HandleMsgBlock and copy it to the local variable(s) of the handle function.

If processing for the operations performed on the custom window is wanted, write it in each corresponding handle function in the framework source file after the comment shown below.

```
/* Write message handler code here, please. */
```

[Precaution] Do not delete the handle functions written in the framework source file. CB38 **will become unable to build correctly. Do not modify the local variable setup procedures written in the handle functions either. The custom window program may become unable to operate correctly.**

3.5 Method for using handle functions

This section describes how to use the handle functions written in the framework source file.

[Precaution] Since the handle functions are special functions called by PD38**, do not try to call them freely like a user-defined function. This is because such an operation could make it impossible for** PD38 **to call them correctly.**

3.5.1 Using the OnCreate handle function (to start creating a window)

When PD38 starts executing a custom window program, the OnCreate() function is called only once immediately before creating a custom window. In this function, set the position at which a window opens, the window size when opened, and the window title, as well as generate control items (e.g., buttons).

The OnCreate handle function that is automatically created is shown below.
(There is no information associated with it.)

```
OnCreate()
{
        /* Write message handler code here, please. */

}
```

The OnCreate() handle function is the first function executed among all functions in the custom window program source file.

When creating a window, the handle functions are called in order of OnCreate -> OnSize -> OnDraw.

3.5.2 Using OnDestroy handle function (to start destroying a window)

When a system menu is selected to close a custom window that is open, the OnDestroy() handle function is called only once immediately before destroying the custom window. In this function, free the heap area and system timer and perform related other operations. The control items are automatically destroyed after this function is executed.

The OnDestroy handle function that is automatically created is shown below.
(There is no information associated with it.)

```
OnDestroy()
{
        /* Write message handler code here, please. */

}
```

The OnDestroy() handle function is the last function executed among all functions in the custom window program source file.

After processing of the OnDestroy() handle function is terminated, PD38 frees the control items used and destroys the custom window before it finishes executing the custom window program.

3.5.3 Using the OnDraw handle function (to request redrawing a window)

The OnDraw() handle function is called in the cases described below. In this function, PD38 draws a window in the window drawing area.

- When part (or the whole) of a custom window is hidden behind some other window and the hidden part is exposed

  In this case, the window drawing area is cleared immediately before calling the OnDraw() handle function.

- When one of the window manipulating functions to redraw a window is called

  There are following two redraw functions:

  1. _win_redraw_clear()

     If this function is called, the window drawing area is cleared immediately before calling the OnDraw() handle function.

  2. _win_redraw()

     If this function is called, the window drawing area is not cleared immediately before calling the OnDraw() handle function.

The OnDraw handle function that is automatically created is shown below.
(There is no information associated with it.)

```
OnDraw()
{
        /* Write message handler code here, please. */

}
```

Since the OnDraw() handle function is called rather frequently, Mitsubishi recommends that this function be used for only drawing a window, and that the OnEvent() handle function, etc. be used to get or process the data required for drawing (e.g., memory and register values) that takes time.

3.5.4 Using the OnEvent handle function (for status change of PD38)

The OnEvent() handle function is called when the status of PD38 has changed. In this function, processing is performed that corresponds to a change of the PD38 status.

A change of the PD38 status refers to one of the following events:

1.When a new target program is downloaded
2.When the target program is single-stepped
3.When a register value is modified
4.When the information to be displayed by PD38 is modified

The type of change that has occurred to the status of PD38 is passed to the local variable nEventID. Types of status are detailed in the "CB38 V.1.00 User's Manual," Section 5.4.7, "OnEvent Handle Function".

Since processing need to be performed for multiple status changes by one handle function, a procedure that is taken normally is that a switch statement, etc. is used at the beginning of the function to determine the type of change indicated by the local variable nEventID and control is made to branch off to processing that corresponds to the status change that has occurred.

The OnEvent handle function that is automatically created is shown below.
(The information associated with it is nEventID.)

```
OnEvent()
{
        int     nEventID;

        nEventID = ((int *)_HandleMsgBlock)[0];

        /* Write message handler code here, please. */

}
```

If the target memory value is modified in PD38's dump window, etc., the OnEvent() handle function is called for nEventID == EVENT_PUT_MEM. For a custom window where memory values are displayed, the memory values are reacquired to update the display.

Note that when the target program is executing, the OnEvent() handle function is periodically called for nEventID == EVENT_TIME_10MS. It is recommended that processing which need to be performed periodically only when the target program is executing (e.g., processing based on sampling) be written at a place to which control branches for nEventID == EVENT_TIME_10MS.

[About window drawing processing]

The following explains how processing is performed to draw a window using the OnDraw() and OnEvent() handle functions described above.

There are following two ways in which the OnDraw() handle function normally is used:

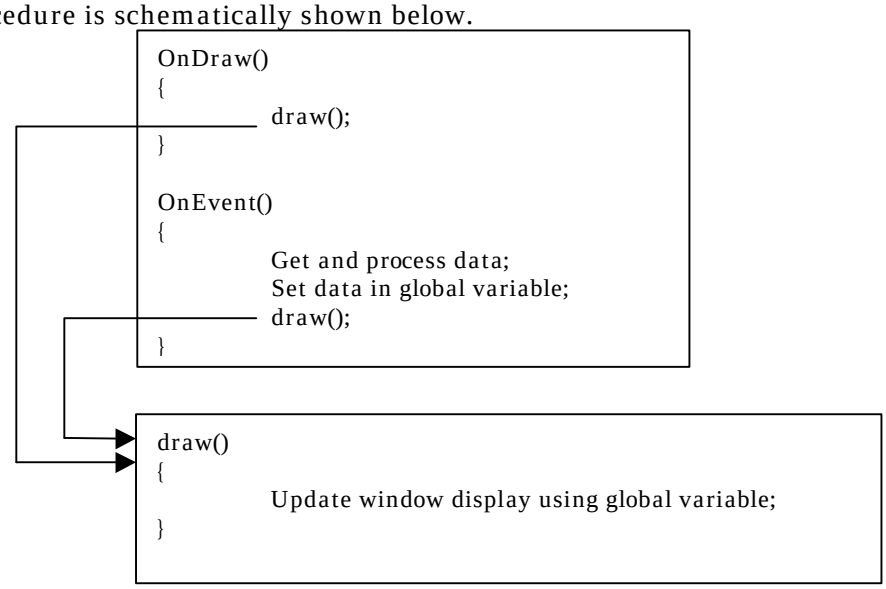
1. Drawing processing is not performed in this function; instead, some other function that performs the entire drawing processing is called.
   (The OnDraw() handle function works merely as one that receives a redrawing request. All drawing processing is performed by calling some other function.)
2. Drawing processing is performed in this function. (In some cases, a "subcontract" function may be called.)
   (The OnDraw() function must always be executed to perform the processing to draw a window.)

The following explains the difference between these two methods and how each method is used.

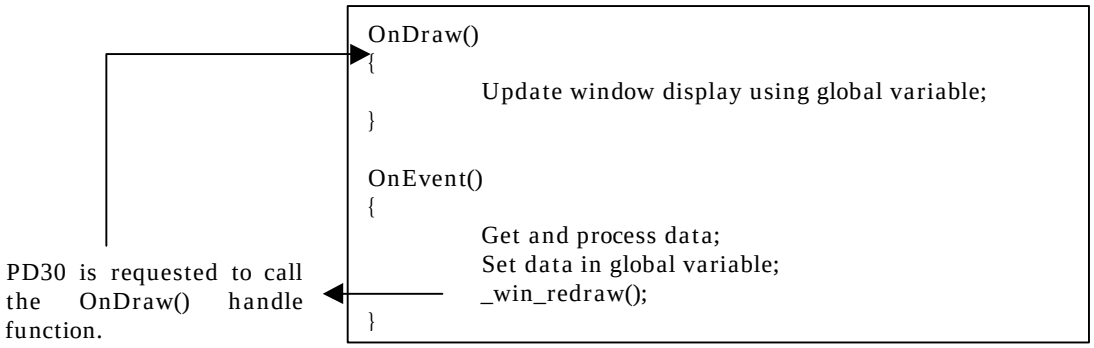
Drawing processing need to be performed at times other than when the OnDraw() function is called. For example, assume a window (e.g., PD38's memory window) in which a specific memory content is displayed successively. For such a window, every time the target memory value is modified, the window display must be updated with a new memory value.

In such a case, the OnEvent() handle function may be used to get the target memory value which is then stored in a global variable and drawn to the window by a drawing function.

If method 1 is used for drawing to a window, get a memory value in the OnEvent() handle function and then call a function directly from it that performs drawing processing. Thus, method 1 is that when drawing is required, a function to perform drawing processing is called directly after acquiring the data necessary for the drawing. This procedure is schematically shown below.



If method 2 is used for drawing to a window, get a memory value in the OnEvent() handle function and then call the _win_redraw() or _win_redraw_clear() function in order to request PD38 to call the OnDraw() handle function. Thus, method 2 is that when drawing is required, the OnDraw() function is called indirectly by requesting it from PD38 after acquiring the data necessary to perform drawing processing. This procedure is schematically shown below.



The OnDraw() handle function is frequently called, even on an unintended occasion such as when some other window crosses in front of a window. Therefore, it is desirable that the OnDraw() handle function be used for only drawing, and that time-consuming processing (e.g., acquiring and processing data) be performed at other timing.

3.5.5 Using the OnSize handle function (to change window size)

The OnSize() handle function is called when the size of a custom window is changed. In this function, PD38 performs processing that corresponds to a change of the customer window size.

The type of size change (e.g., maximize or iconify) and the new width and height of the client area are passed to local variables nType, cx, and cy, respectively. Types of size changes are detailed in the "CB38 V.1.00 User's Manual," Section 5.4.18, "OnSize Handle Function".

The OnSize handle function that is automatically created is shown below.
(The information associated with it are nType, cx, and cy.)

```
OnSize()
{
        int     nType;
        int     cx;
        int     cy;

        nType = ((int *)_HandleMsgBlock)[0];
        cx = ((int *)_HandleMsgBlock)[1];
        cy = ((int *)_HandleMsgBlock)[2];

        /* Write message handler code here, please. */

}
```

For custom window programs that change drawing, etc. by using a window's size information, use global variables to hold the window sizes (cx, cy) acquired by the OnSize() handle function.

Since the OnSize() handle function is called following the OnCreate() handle function when creating a window, it can also be used to get the window size when created.

3.5.6 Using the OnCommand handle function (to manipulate control items (buttons))

The OnCommand() handle function is called when one of the generated control items (buttons) is operated on. In this function, perform the processing that corresponds to the control item that is operated on.

The control item's command ID, notification code, and handle respectively are passed to local variables nID, nMsg, and hHandle. The OnCommand() handle function is detailed in the "CB38 V.1.00 User's Manual," Section 5.4.3, "OnCommand Handle Function".

The OnSize handle function that is automatically created is shown below.
(The information associated with it are nId, nMsg, and hHandle.)

```
OnCommand()
{
        int     nId;
        int     nMsg;
        int     nHandle;

        nId = ((int *)_HandleMsgBlock)[0];
        nMsg = ((int *)_HandleMsgBlock)[1];
        nHandle = ((int *)_HandleMsgBlock)[2];

        /* Write message handler code here, please. */

}
```

Buttons are the only control item supported by CB38 V.1.00.

The local variable nMsg is not used for buttons; nMsg is reserved for use in future versions of CB38.

3.5.7 Using the OnHScroll and other handle functions (to manipulate scroll bars)

The OnHScroll() handle function is called when the horizontal scroll bar is operated on.

Similarly, the OnVScroll() handle function is called when the vertical scroll bar is operated on. In these functions, perform the processing that corresponds to the scroll bar that is operated on.

The operation code for the scroll bar (e.g., drag, page scroll) and the scroll thumb (slider) position respectively are passed to local variables “nSBCode” and “nPos”. The operation code and scroll thumb position detailed in the "CB38 V.1.00 User's Manual," Section 5.4.8, "OnHScroll Handle Function".

The OnHScroll handle function that is automatically created is shown below.
(The information associated with it are nSBCode and nPos.)

```
OnHScroll()
{
        int     nSBCode;
        int     nPos;

        nSBCode = ((int *)_HandleMsgBlock)[0];
        nPos = ((int *)_HandleMsgBlock)[1];

        /* Write message handler code here, please. */

}
```

The local variable nPos is used only when nSBCode is SB_THUMBPOSITION or SB_THUMBTRACK.

When the scroll operation is completed, these functions are called for nSBCode == SB_ENDSCROLL to notify the end of scroll operation to window.

3.5.8 Using the OnLButtonDblClk and other handle functions (to manipulate mouse)

Following handle functions are called when the mouse is operated on.

Handle function	Cases where the function is called
OnLButtonDblClk	When the left mouse button is double-clicked.
OnLButtonDown	When the left mouse button is pressed.
OnLButtonUp	When the left mouse button is released.
OnMouseMove	When the mouse cursor is moved.
OnRButtonDblClk	When the right mouse button is double-clicked.
OnRButtonDown	When the right mouse button is pressed.
OnRButtonUp	When the right mouse button is released.

In these functions, perform the processing that corresponds to the kind of operation performed on the mouse.

The key code that is pressed at the same time the mouse is operated on and the mouse cursor's x and y coordinates respectively are passed to local variables nFlags, x, and y. The key code is detailed in the "CB38 V.1.00 User's Manual," Section 5.4.11, "OnLButtonDblClk Handle Function".

The OnHScroll handle function that is automatically created is shown below.
(The information associated with it are nFlags, x, and y.)

```
OnLButtonDblClk()
{
        int     nFlags;
        int     x;
        int     y;

        nFlags = ((int *)_HandleMsgBlock)[0];
        x = ((int *)_HandleMsgBlock)[1];
        y = ((int *)_HandleMsgBlock)[2];

        /* Write message handler code here, please. */

}
```

When the mouse button is double-clicked, the above handle functions are called in order of OnXButtonDown -> OnXButtonUp -> OnXButtonDblClik -> OnXButtonUp. (X is L when the left button is concerned or R when the right button is concerned.)

3.5.9 Using the OnChar and other handle functions (to manipulate keys)

Following handle functions are called when a key is operated on the keyboard.

Handle function	Cases where the function is called
OnChar	When a WM_KEYDOWN message is converted into character code. Stored in the key code is the converted ASCII code.
OnKeyDown	When any key other than the system key is pressed. Stored in the key code is the virtual key code of the pressed key.
OnKeyUp	When any key other than the system key is released. Stored in the key code is the virtual key code of the released key.

In these functions, perform the processing that corresponds to the kind of operation performed on the keyboard.

The key code, repeat count value, and scan code value of the pressed key are passed to local variables nChar, nRepCnt, and nFlags, respectively. The key code and the repeat count and scan code values are detailed in the "CB38 V.1.00 User's Manual," Section 5.4.9, "OnKeyDown Handle Function".

The OnChar handle function that is automatically created is shown below.
(The information associated with it are nChar, nRepCnt, and nFlags.)

```
OnChar()
{
        int     nChar;
        int     nRepCnt;
        int     nFlags;

        nChar = ((int *)_HandleMsgBlock)[0];
        nRepCnt = ((int *)_HandleMsgBlock)[1];
        nFlags = ((int *)_HandleMsgBlock)[2];

        /* Write message handler code here, please. */

}
```

When a key that can be converted into character code is pressed, the above handle functions are called in order of OnKeyDown -> OnChar -> OnKeyUp. If a key is held down, the handle functions are called in order of OnKeyDown -> (OnChar) -> OnKeyDown (OnChar) -> ... -> OnKeyUp. (The OnChar handle function is called only when a key is pressed that can be converted into character code.)

If the pressed key corresponds to one of ASCII characters, the corresponding ASCII code is stored in nChar. For keys that do not correspond to ASCII characters such as in the case of function keys, a corresponding virtual key code value is stored in nChar. For details about virtual key code, refer to the "CB38 V.1.00 User's Manual," Section 5.4.9, "OnKeyDown Handle Function".

3.5.10 Using the OnTimer handle function

The OnTimer() handle function is called at preset intervals when the system timer provided by Windows is used. In this function, write the processing that is executed at preset intervals.

The timer's identification number is passed to local variable nIDEvent. This identification number can be set as desired by the user when using Windows' system timer.

Since processing need to be performed for multiple timer-related processing by one handle function, a procedure that is taken normally is that a switch statement, etc. is used at the beginning of the function to determine the type of timer indicated by the local variable nIDEvent and control is made to branch off to processing that corresponds to the timer concerned.

The OnTimer handle function that is automatically created is shown below.
(The information associated with it is nIDEvent.)

```
OnTimer()
{
        int     nIDEvent;

        nIDEvent = ((int *)_HandleMsgBlock)[0];

        /* Write message handler code here, please. */

}
```

Note that when the target program is executing, the OnEvent() handle function is periodically called for nEventID == EVENT_TIME_10MS. It is recommended that processing which need to be performed periodically only when the target program is executing (e.g., processing based on sampling) be serviced by the OnEvent() handle function.

[Precaution] The total number of system timers is limited by the OS used. Use of too many system timers than necessary could affect the operation of other applications.

3.6 Functions that can be used in programming a custom window

The functions that can be used in programming a custom window are broadly classified into the following three groups:

1. Standard functions
   The functions similar to the standard C-language functions that are assumed to be relatively frequently used are supported.

2. Debugger operating functions
   The functions necessary to operate the debugger are supported.

3. Window manipulating functions
   The functions necessary to manipulate a window are supported.

3.7 Method for using window manipulating functions

When using the window manipulating functions, include the header file “winlib.h” in the function you are going to use.

Specifications of the window manipulating functions are detailed in the "CB38 V.1.00 User's Manual," Section 5.3, "System Call Functions for Window Operation (winlib.lib)".

3.7.1 Using drawing functions

This section explains how to use the drawing functions by using the functions shown below as an example.

Function name	Description
_win_printf	Output text with format
_win_set_cursor	Set cursor position
_win_set_color	Set text color
_win_set_bkcolor	Set background color
_draw_frame_rect	Draw rectangle

The functions whose name begins with _win draw an object on cursor coordinates (the coordinate system specified by row and column). One character of system font is output to one column of cursor coordinate.

The functions whose name begins with _draw draw an object on pixel coordinates (the coordinate system specified by a dot position).

The following shows an example where character strings "Hello world" are output to a custom window.

```
OnDraw()
{
        /* Write message handler code here, please. */
        _win_set_cursor(3, 1);                  /* Set cursor position to (3, 1) */
        _win_printf("Hello World.");            /* Output character string */
}
```

The lines printed in plain style are the codes automatically generated in the framework source file by CB38. The lines printed in bold face are the codes to be added by the user. (The same applies in the examples that may follow.)

Display example

The following shows an example where "Hello world" is output in inverse video.

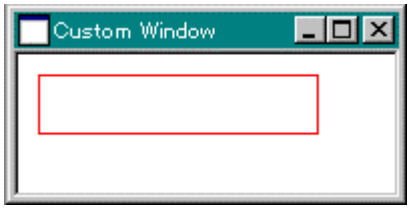
```
OnDraw()
{
        /* Write message handler code here, please. */
        int         old_color;/* Variable used to save text color before change */
        int         old_bkcolor;        /* Variable used to save background color before change */

        _win_set_cursor(3, 1);                      /* Set cursor position to (3, 1) */
                                                    /* Set text color to white */
        old_color = _win_set_color(COLOR_WHITE);
                                                    /* Set background color to black */
        old_bkcolor = _win_set_bkcolor(COLOR_BLACK);
        _win_printf("Hello World.");                /* Output character string */
        _win_set_color(old_color);                  /* Restore text color */
        _win_set_bkcolor(old_bkcolor);              /* Restore background color */
}
```

Display example

The following shows an example where a red rectangular area is drawn.

```
OnDraw()
{
        /* Write message handler code here, please. */
                                                    /* Draw a red rectangle whose upper left */
                                                    /* coordinate is (10, 10) and lower right */
                                                    /* coordinate is (150, 40) */
        _draw_frame_rect(10, 10, 150, 40, COLOR_RED);
}
```

Display example



3.7.2 Using functions to manipulate control items (buttons)

Buttons are supported as the control item that can be attached to a custom window created by CB38. This section explains how to use the control item manipulating functions by using the functions shown below as an example.

Function name	Description
_win_button_create	Create button
_win_button_set_text	Change button text

The following shows an example where the button is assigned a label "button" when created and the label is changed between uppercase and lowercase each time the button is entered.

```
#define     IDB_BUTTON    (1000)    /* Define button ID number */
int         hButton;                /* Variable to store button handle */
int         count;                  /* Variable to store button-pressed count */

OnCommand()
{
    int     nId;
    int     nMsg;
    int     nHandle;

    nId = ((int *)_HandleMsgBlock)[0];
    nMsg = ((int *)_HandleMsgBlock)[1];
    nHandle = ((int *)_HandleMsgBlock)[2];

    /* Write message handler code here, please. */
    switch(nId){
    case IDB_BUTTON:                /* If button ID is IDB_BUTTON */
        if(++count % 2){            /* If button-pressed count is odd number */
                                    /* Change label to "BUTTON" */
            _win_button_set_text(hButton, "BUTTON");
        }else{                      /* If button-pressed count is even number */
                                    /* Change label to "button" */
            _win_button_set_text(hButton, "button");
        }
        break;
    }
}

OnCreate()
{
    /* Write message handler code here, please. */
    count = 0;                      /* Initialize button-pressed count to 0 */
                                    /* Create button whose upper left coordinate */
                                    /* is (10, 10) and lower right coordinate is */
                                    /*   (100, 4) that has a label "button" and */
                                    /* IDB_BUTTON as its ID and hold button's */
                                    /* handle in hButton */
    hButton = _win_button_create(10, 10, 100, 40, "button", IDB_BUTTON);
}
```

Display example (when created)

Display example (when button is entered)

3.7.3 Using functions to manipulate the status bar

This section explains how to use the status bar manipulating functions by using the functions shown below as an example.

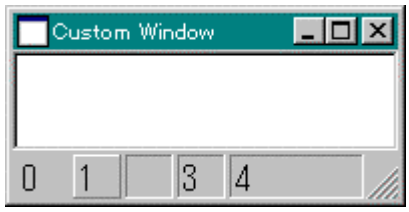
Function name	Description
_win_statusbar_create	Create status bar
_win_statusbar_set_pane	Set items of status bar
_win_statusbar_set_text	Set text of status bar

The following shows an example for a status bar that has five items.

```
OnCreate()
{
        /* Write message handler code here, please. */
        _win_statusbar_create(5);/* Create status bar that has five items */
                                 /* et 0th item (leftmost item) in SBPS_NOBORDER */
                                 /* style and in size of 20 pixels */
        _win_statusbar_set_pane(0, SBPS_NOBORDERS, 20);
                                 /* Draw "0" in 0th item */
        _win_statusbar_set_text(0, "0");
                                 /* Set 1st item in SBPS_POPOUT style and in size of */
                                 /* 20 pixels */
        _win_statusbar_set_pane(1, SBPS_POPOUT, 20);
                                 /* Draw "1" in 1st item */
        _win_statusbar_set_text(1, "1");
                                 /* Set 2nd item in SBPS_DISABLED style and in size of */
                                 /* 20 pixels */
        _win_statusbar_set_pane(2, SBPS_DISABLED, 20);
                                 /* Set 3rd item in SBPS_NORMAL style and in size of */
                                 /* 20 pixels */
        _win_statusbar_set_pane(3, SBPS_NORMAL, 20);
                                 /* Draw "3" in 3rd item */
        _win_statusbar_set_text(3, "3");
                                 /* Set 4th item in SBPS_STRETCH | SBPS_NORMAL */
                                 /* style Since SBPS_STRETCH style is set, 4th item */
                                 /* stretches as window is expanded or reduced */
        _win_statusbar_set_pane(4, SBPS_STRETCH | SBPS_NORMAL, 0);
                                 /* Draw "4" in 4th item */
        _win_statusbar_set_text(4, "4");
}
```

Specify the style of the status bar item in the third argument of the _win_statusbar_set_pane() function. Styles are detailed in the "CB38 V.1.00 User's Manual," Section 5.3.32, "_win_statusbar_set_pane: Set status bar items".

Display example



3.7.4 Using functions to manipulate the scroll bar

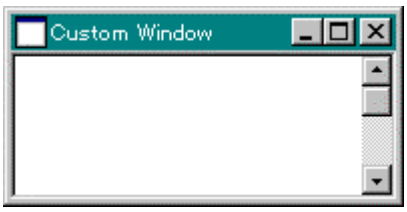
This section explains how to use the scroll bar manipulating functions by using the functions shown below as an example.

Function name	Description
_win_vscroll_range	Set scroll range of vertical scroll bar
_win_vscroll_pos	Set position of vertical scroll box

The following shows an example for displaying a vertical scroll bar in the window.

```
OnCreate()
{
        /* Write message handler code here, please. */
        _win_vscroll_range(0, 100);              /* Set scroll range to 0 through 100 */
}
```

Display example



Shown below is an example of the OnVScroll() handle function in which processing is written that corresponds to the up/down operation of the scroll bar.

```
int   VScrollPageSize;                  /* Contain number of lines per page */
int   VScrollPos;                       /* Store thumb position */

OnVScroll()
{
      int       nSBCode;
      int       nPos;

      nSBCode = ((int *)_HandleMsgBlock)[0];
      nPos = ((int *)_HandleMsgBlock)[1];

      /* Write message handler code here, please. */
      switch(nSBCode){
      case SB_BOTTOM:           /* Scroll to the bottom */
                VScrollPos = 100;
                break;
      case SB_ENDSCROLL:                /* Finish scrolling */
                break;
      case SB_LINEDOWN:                 /* Scroll one line down */
                VScrollPos++;
                break;
      case SB_LINEUP:                   /* Scroll one line up */
                VScrollPos--;
                break;
      case SB_PAGEDOWN:                 /* Scroll one page down */
                VScrollPos += VScrollPageSize;
                break;
      case SB_PAGEUP:                   /* Scroll one page up */
                VScrollPos -= VScrollPageSize;
                break;
      case SB_THUMBPOSITION:/* Scroll to absolute position */
                                        /* (Current position is specified by nPos)*/
      case SB_THUMBTRACK:               /* Drag scroll box to specified position */
                                        /* (Current position is specified by nPos) */
                VScrollPos = nPos;
                break;
      case SB_TOP:                      /* Scroll to the top */
      default:
                VScrollPos = 0;
      }
      if(VScrollPos < 0)                /* Processing performed when output of scroll */
                                        /* range */
                VScrollPos = 0;
      if(VScrollPos > 100)
                VScrollPos = 100;
      _win_vscroll_pos(VScrollPos);     /* Set scroll thumb position */
      _win_redraw_clear();              /* Redraw custom window */
}
```

3.7.5 Using functions to manipulate dialog box

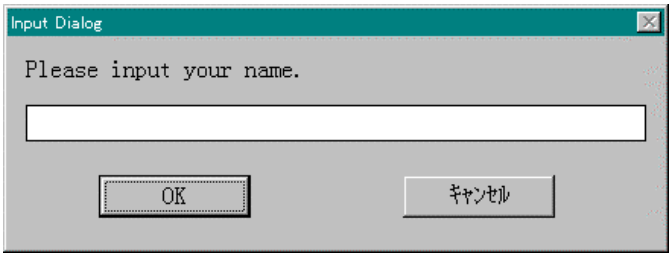
This section explains how to use the dialog box manipulating functions by using the functions shown below as an example.

Function name	Description
_win_dialog	Create input dialog box
_win_message_box	Create message box

The following shows an example where an input dialog box is used to get a value. This function opens the input dialog box, asking the user to input a value, and when "eisuke" is input, returns TRUE; otherwise, it returns FALSE after displaying an error in the message box.

```
int check_name()
{
        char        name[100];

                                                /* Open input dialog box, asking for input */
        if(_win_dialog("Please input your name.", name) == TRUE){
                                                /* OK button is input */
                if(strcmp(name, "eisuke") == 0){
                                                /* Character string is "eisuke" */
                        return TRUE;
                }
        }
                                                /* Display error in message box */
        _win_message_box(" Name is incorrect ", "Error!",
                                                MB_ICONEXCLAMATION | MB_OK);
        return FALSE;
}
```

Display example



Input dialog box

Message box

3.7.6 Using functions to manipulate the window frame

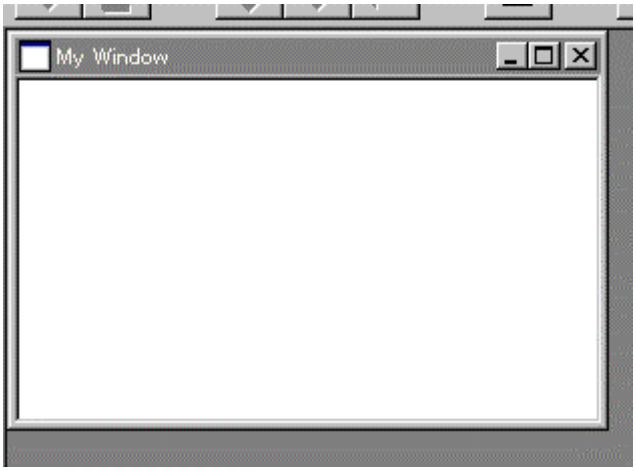
This section explains how to use the window frame manipulating functions by using the functions shown below as an example.

Function name	Description
_win_redraw	Redraw custom window
_win_redraw_clear	Redraw custom window (with area clear)
_win_set_window_title	Set title of custom window
_win_set_window_pos	Set position of custom window
_win_set_window_size	Set size of custom window

The following shows an example where the custom window is titled "My Window" and its size when opened is (300 x 200) pixels and the window is opened at a position (0, 0). The window position is referenced to the upper left corner of the PD38 window's client area as its origin (0, 0), with pixels in the rightward direction defined by “x” and those in the downward direction defined by y.

```
OnCreate()
{
        /* Write message handler code here, please. */
        _win_set_window_title("My Window"); /* Set window title */
        _win_set_window_size(300, 200);              /* Set window size */
        _win_set_window_pos(0, 0);                   /* Set window position */
}
```

Display example



3.7.7 Using functions to operate the system timer

This section explains how to use the system timer operating functions by using the functions shown below as an example.

Function name	Description
_win_timer_set	Set system timer
_win_timer_kill	Clear system timer

The following shows an example where the system timer is used to increment a counter every 200 ms.

```
#define     IDT_1    (10)                    /* Define timer ID number */
int         count;                           /* Counter */

OnCreate()
{
    /* Write message handler code here, please. */
    _win_timer_set(IDT_1, 200);          /* Set system timer */
    count = 0;                           /* Initialize counter to 0 */
}

OnDestroy()
{
    /* Write message handler code here, please. */
    _win_timer_kill(IDT_1);              /* Clear system timer */
}

OnTimer()
{
    int     nIDEvent;

    nIDEvent = ((int *)_HandleMsgBlock)[0];

    /* Write message handler code here, please. */
    if(nIDEvent == IDT_1){               /* If system timer is IDT_1 */
        count++;                         /* Increment counter by 1 */
    }
}
```

Note that when the target program is executing, the OnEvent() handle function is periodically called for nEventID == EVENT_TIME_10MS. It is recommended that processing which need to be performed periodically only when the target program is executing (e.g., processing based on sampling) be serviced by the OnEvent() handle function.

[Precaution] The total number of system timers is limited by the OS used. Use of too many system timers than necessary could affect the operation of other applications.

MEMO

CB38 V.1.00 Programming Manual

Rev. 1.00
May 1, 2003
REJ10J0096-0100Z

CB38 V.1.00
Programming Manual

Renesas Electronics Corporation
1753, Shimonumabe, Nakahara-ku, Kawasaki-shi, Kanagawa 211-8668 Japan

REJ10J0096-0100Z