NEC

Preliminary User's Manual

IMAPCAR2 Family

1DC Library

Software

IMAPCAR2 series

IMAPCAR2 – 300

IMAPCAR2 – 200

IMAPCAR2 – 100

IMAPCAR2 – 50

Document No. U19888EE1V0UM00

Date Published: July 2009

Legal Notes

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

No part of this document may be copied or reproduced in any form or by any means without the prior written consent of NEC Electronics. NEC Electronics assumes no responsibility for any errors that may appear in this document.

- NEC Electronics does not assume any liability for infringement of patents, copyrights or other intellectual property rights of third parties by or arising from the use of NEC Electronics products listed in this document or any other liability arising from the use of such products. No license, express, implied or otherwise, is granted under any patents, copyrights or other intellectual property rights of NEC Electronics or others.
- Descriptions of circuits, software and other related information in this document are provided for illustrative purposes in semiconductor product operation and application examples. The incorporation of these circuits, software and information in the design of a customer's equipment shall be done under the full responsibility of the customer. NEC Electronics assumes no responsibility for any losses incurred by customers or third parties arising from the use of these circuits, software and information.
- While NEC Electronics endeavors to enhance the quality, reliability and safety of NEC Electronics products, customers agree and acknowledge that the possibility of defects thereof cannot be eliminated entirely. To minimize risks of damage to property or injury (including death) to persons arising from defects in NEC Electronics products, customers must incorporate sufficient safety measures in their design, such as redundancy, fire-containment and anti-failure features.
- NEC Electronics products are classified into the following three quality grades: "Standard", "Special" and "Specific".
  The "Specific" quality grade applies only to NEC Electronics products developed based on a customer-designated "quality assurance program" for a specific application. The recommended applications of an NEC Electronics product depend on its quality grade, as indicated below. Customers must check the quality grade of each NEC Electronics product before using it in a particular application.
  "Standard": Computers, office equipment, communications equipment, test and measurement equipment, audio and visual equipment, home electronic appliances, machine tools, personal electronic equipment and industrial robots.
  "Special": Transportation equipment (automobiles, trains, ships, etc.), traffic control systems, anti-disaster systems, anti-crime systems, safety equipment and medical equipment (not specifically designed for life support).
  "Specific": Aircraft, aerospace equipment, submersible repeaters, nuclear reactor control systems, life support systems and medical equipment for life support, etc.

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

(Note)

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

(2) "NEC Electronics products" means any product developed or manufactured by or for NEC Electronics (as defined above).

Table of Contents

1 Overview....9
1.1 Standard Function Library and Startup Routine Files....10
1.2 Include Files....11
1.3 Startup Processing....12
1.4 Program Creation Procedure....13
1.4.1 Creating a source file....13
1.4.2 Assembling or compiling....13
1.4.3 Linking....13
2 Overview of functions....14
2.1 External Memory Transfer Functions ((Line Transfer)....15
2.1.1 Foreground line transfer functions....15
2.1.2 Background line transfer functions....16
2.1.3 Line transfer functions for which a window register set is defined....17
2.1.4 Functions to directly set up line transfers....18
2.1.5 Low level functions that support line transfers....19
2.1.6 Line transfer functions that use division....19
2.2 Others....20
2.2.1 Dynamic memory allocation....20
2.2.2 ROI transfer and cache operations....20
2.2.3 Basic functions for the MP mode....21
2.2.4 Message sending and receiving functions for MP mode....22
2.2.5 Sep data operations....23
2.2.6 Data transfers between sep-type and scalar data....23
2.2.7 Sep data transfers....24
2.2.8 Inline functions for absolute values calculation....24
2.2.9 Inline functions for accumulator addition....24
2.2.10 Inline functions for saturation arithmetic....25
2.2.11 Inline functions for maximization and minimization....26
2.2.12 Inline functions for bit manipulation....26
2.2.13 Inline functions for special registers read or write....26
2.2.14 Integer square root functions....27
2.2.15 Inline functions for port I/O....27
2.2.16 Inline functions for floating point arithmetic....27
2.2.17 Inline and normal functions to prohibit, permit and enable interrupts....27
2.2.18 Inline functions for data transfer between connected PEs....27
2.2.19 Operation stop function....28
2.2.20 Debugger command execution function....28
2.2.21 Functions for data copy within memory and values specification....28

2.2.22 Basic image processing functions 28
2.2.23 Other functions....28
3 Function details 29
3.1 Functions to transfer data (lines) between EMEM and IMEM 29
3.1.1 Ix_ememrw_init....31
3.1.2 Ix_ememrw{_hp}....33
3.1.3 Ix_ememrw{_hp}_blk 34
3.1.4 Ix_ememrw{_hp}_wno 35
3.1.5 Ix_ememrw{_hp}_wno_blk....36
3.1.6 Ix_ememrw{_hp}_req....37
3.1.7 Ix_wclose 37
3.1.8 Ix_wclose_all....38
3.1.9 Ix_wget0, Ix_wget1, Ix_wget2....38
3.1.10 Ix_wopen....39
3.1.11 Ix_wopen_all 39
3.1.12 Ix_wset0, Ix_wset1, Ix_wset2 40
3.1.13 Ix_wreq0, Ix_wreq1....40
3.1.14 Ix_wwait 41
3.1.15 IxEmemBlk....41
3.1.16 IxEmemBlk_Lno....42
3.1.17 IxEmemBlk_So 45
3.1.18 IxEmemBlk_So 48
3.1.19 IxEmemrd{_hp}, IxEmemrd{hp}_start, IxEmemrd_end....50
3.1.20 IxEmemrd{_hp}_offset, IxEmemrd{_hp}_offset_start 51
3.1.21 IxEmemrd{_hp}_blk, IxEmemrd{_hp}_blk_start....51
3.1.22 IxEmemrw{_hp}, IxEmemrw{_hp}_start, IxEmemrw_end....52
3.1.23 IxEmemrw{_hp}_offset, IxEmemrw{_hp}_offset_start 53
3.1.24 IxEmemrw{_hp}_blk, IxEmemrw{_hp}_blk_start....53
3.1.25 IxEmemwr{_hp}, IxEmemwr{_hp}_start, IxEmemwr_end....54
3.1.26 IxEmemwr{_hp}_offset, IxEmemwr{_hp}_offset_start 55
3.1.27 IxEmemwr{_hp}_blk, IxEmemwr{_hp}_bl 55
3.2 Functions to transfer data between DMEM and IMEM 56
3.2.1 IxDmem2imem_comm....57
3.2.2 IxDmem2imem{_u2, _u4} 57
3.2.3 IxDmem2PE{_u2} 58
3.2.4 IxImem2dmem{_u2, _u4}....59
3.2.5 IxPE2dmem{_u2}....61
3.2.6 IxPEs2dmem{_u2, _u4}....62
3.3 Functions to transfer data between DMEM and shared RAM 63
3.3.1 IxDmem2shared_u4{_nc} 63

3.3.2 IxShared2dmem_u4{_nc} 63
3.4 Functions to transfer data between memory area of the same type (simple copy and initialization) 64
3.4.1 ix_memcpy{_nc, _nc2c, _c2nc, 16, 16_nc, 16_nc2c, 16_c2nc} 64
3.4.2 ix_memset{_u2, _nc, _nc_u2, _u4, _nc_u4} 65
3.4.3 Ix_memcpy{16} 65
3.4.4 Ix_memset{_u2, _u4} 66
3.4.5 Ix_memset_sep{_u2, _u4} 66
3.4.6 IxEmem2emem 66
3.5 Functions to transfer data within DMEM (ROI transfers) 67
3.5.1 ix_roi_read 67
3.5.2 ix_roi_write 67
3.6 Data cache manipulation functions 68
3.6.1 ix_civd_all 68
3.6.2 ix_civda 68
3.6.3 ix_civda2 68
3.6.4 ix_cwb_all 68
3.6.5 ix_cwba 68
3.6.6 ix_cwba2 68
3.6.7 ix_cwbivda 68
3.7 Dynamic Memory Allocation 69
3.7.1 ix_malloc, ix_calloc, ix_free, ix_realloc, ix_heap_overflow 69
3.7.2 Ix_eheap, Ix_eheap_available, Ix_eheap_overflow 70
3.7.3 Ix_eheap_return 71
3.7.4 Ix_iheap, Ix_iheap_available, Ix_iheap_overflow 71
3.7.5 Ix_iheap_return 72
3.8 Sep Data Manipulation Functions 73
3.8.1 IxAddsep, IxAddsepl, IxAddsepll 73
3.8.2 IxCountsep 73
3.8.3 IxGathersep 73
3.8.4 IxGetleftsep 74
3.8.5 IxGetrightsep 74
3.8.6 IxMaxsep 75
3.8.7 IxMinsep 75
3.8.8 IxPack8, IxPack8i, IxPack8all 75
3.8.9 IxPack8s, IxPack8si 76
3.8.10 IxOrdersep 76
3.9 Sep Data Transposition Functions 77
3.9.1 IxRemap_rx1_1xr 78
3.9.2 IxRemap_rec_1pe 80

3.9.3 IxRemap_sqr_sqr 82
3.9.4 IxRot90 83
3.10 Basic functions for MP mode 84
3.10.1 ix_cp_id 84
3.10.2 Ix_start_mode 84
3.10.3 Ix_cp_start_pu 85
3.10.4 Ix_cp_start_multi_pu 86
3.10.5 Ix_fork 87
3.10.6 Ix_fork_again 87
3.10.7 Ix_forkd 88
3.10.8 Ix_forkinit 88
3.10.9 Ix_forkp 88
3.10.10 Ix_initm 89
3.10.11 ix_is_MIXED, ix_is_MP, ix_is_SIMD 89
3.10.12 Ix_join 89
3.10.13 Ix_mutex_init, Ix_mutex_destroy 90
3.10.14 ix_mutex_lock, ix_mutex_unlock 91
3.10.15 ix_modify_dstkp 91
3.10.16 ix_pu_id 91
3.10.17 ix_pu_local_area 92
3.10.18 ix_TLSread_{u1, u2, u4, s1}, ix_TLSwrite{u1, u2, u4} 94
3.10.19 ix_total_pu_number, ix_max_total_pu_number 95
3.11 Functions to send and receive messages in MP mode 96
3.11.1 Message format 96
3.11.2 Patterns to send and receive messages, and coding examples 97
3.11.3 Restriction on the number of message buffers 99
3.11.4 ix_msg_header 101
3.11.5 ix_msg_header_intr 102
3.11.6 ix_msg_header_sync 103
3.11.7 ix_msg_id 103
3.11.8 ix_msg_receiver 103
3.11.9 ix_msg_sender 103
3.11.10 ix_msg_type 104
3.11.11 ix_recv 104
3.11.12 ix_recv_any 104
3.11.13 ix_recv_msg 105
3.11.14 ix_recv_msg_blk 105
3.11.15 ix_send 106
3.11.16 ix_send_any 106
3.11.17 ix_send_any_sync 106

3.11.18 ix_send_msg 107
3.11.19 ix_send_msg_blk 107
3.12 Inline Operation Functions 108
3.12.1 Absolute difference (ix_asub) 108
3.12.2 Addition of an absolute difference to an accumulator (ix_acc_asub) 108
3.12.3 Addition of a product to an accumulator (ix_acc_mul) 109
3.12.4 Reading the result of adding to an accumulator (ix_acc_read) 110
3.12.5 Adding to an accumulator (ix_acc) 110
3.12.6 Saturation arithmetic (ix_sadd, ix_ssub) 111
3.12.7 Maximization and minimization operations (ix_max, ix_min) 111
3.12.8 Bit manipulation operations (ix_bitr, ix_bts, ix_skz) 112
3.12.9 Packing operations (Ix_pack) 112
3.13 Inline functions to transfer data between PEs 113
3.13.1 Ix_mvlc, Ix_mvrc, Ix_mvlc_u4, Ix_mvrc_u4 113
3.13.2 Ix_mvlz, Ix_mvrz, Ix_mvlz_u4, Ix_mvrz_u4 113
3.14 Integer Square Root Functions 114
3.14.1 ixSqrt, ixSqrtl, ixSqrtll 114
3.14.2 IxSqrtsep, IxSqrtsepl, IxSqrtsepll 114
3.15 Functions of obtaining the number of PEs and PE numbers 115
3.15.1 Ix_max_total_pe_number 115
3.15.2 Ix_pe_id 115
3.15.3 Ix_real_pe_number 115
3.15.4 Ix_total_pe_number 115
3.15.5 PEMAX, PEMIN 116
3.15.6 PENO 116
3.15.7 PENUM, _PENUM 116
3.16 Image Processing Functions 117
3.16.1 IxPk8tlup 117
3.16.2 IxVote_u1u2 118
3.16.3 IxVoteCollect_u2u4 118
3.17 Interrupt functions 119
3.17.1 ix_idis, ix_iena 119
3.17.2 ix_interrupt_return 119
3.17.3 ix_push_creg_all, ix_pop_creg_all 120
3.17.4 Ix_push_ireg_all, Ix_pop_ireg_all 121
3.18 Other functions 122
3.18.1 ix_halt 122
3.18.2 ix_exec 122
3.18.3 ix_exec2 123
3.18.4 Ix_in, Ix_ou 124

3.18.5 ix_nopwait 124
3.18.6 ix_spec_read, ix_spec_write, lx_spec_read, lx_spec_write 124
3.18.7 IxPtime 125
3.18.8 IxPtimeAcc 126
3.19 Imported Standard C Functions 127
3.20 Imported C Functions for Mathematical Operations 128
4 Performance 132
4.1 Mathematical Operations 132
4.2 Sep Data Manipulation Functions 133
4.3 Functions to transfer data between sep-type data and scalar data 133
4.4 Function to transpose sep-type data 133

1 Overview

The 1DC standard function library described in this document includes:

- functions for basic processing (memory operations and interrupts)
- standard C functions pre-compiled using the 1DC compiler.

The functions in this library are normally linked with when the 1DC compiler (cc1dc) is used to create an object for the IMAP Series processor.

In IMAP assembly programs that support the standard interface used for the 1DC compiler, these functions can also be used by linking with the ldimap linker (except for inline functions).

Except for standard C functions, the names of the functions in this library start with `ix` or `Ix`.

- functions starting with `ix` can be used from a control processor (CP) or processing unit (PU)
- those starting with `Ix` can only be used from a CP (contain operations that can only be performed on a CP or processing element PE array)

Note that operations are not guaranteed when a PU is calling functions starting with `Ix`.

1.1 Standard Function Library and Startup Routine Files

Standard 1DC library functions and standard C functions precompiled using the 1DC compiler are combined into a library using the UNIX command `ar`. This standard function library file is named as follows:

- `lib1dc.ia` Standard function library

This file is stored in the `1dc/cc1dc/imap/lib/`*processor-core-name/ number-of-PEs* directory.

To link with this library, its functions can be used by using the linker ldimap. The source codes of the library functions are in the `1dc/imap/imap/src/lib1dc` directory.

The following object file performs startup processing for 1DC programs (including the first processing performed when the programs are executed and calling the user-defined `main` function):

- `crt0.io` Standard startup routine

This file is stored in the `1dc/cc1dc/imap/lib/`*processor-core-name/ number-of-PEs* directory.

When the 1DC compiler is used to consecutively compile a program and then link with its files, the above file is automatically linked first. However, when only using the IDC compiler to compile a program and using the linker to manually link the program with its files, you might have to explicitly link with `crt0.io` first or link with a different file that performs equivalent startup processing.

1.2 Include Files

The following 1DC library header files include the prototypes and constants for functions in the standard 1DC library:

- `stdimap.h` 1DC standard library functions and prototypes
- `imapio.h` Functions used for video I/O, macro functions, and constants
- `inline.h` Inline functions and prototypes

These files are stored in the `1dc/cc1dc/lib/include/imap` directory.

When using functions from this library, the required header files containing these functions must be included. Note that these header files include the following header files:

- If `stdimap.h` is included, `inline.h` is automatically included.
- If `imapio.h` is included, `stdimap.h` is automatically included.

The following header files for standard C functions include the prototypes and constants for standard C functions supported by the 1DC compiler:

- `float.h` Floating point operations
- `limits.h` Definitions of values such as integer sizes
- `math.h` Mathematical functions and prototypes
- `signal.h` Signal operations
- `stdarg.h` Allows functions to accept a variable number of arguments
- `stddef.h` Definitions of common constants such as `NULL`, `PEN0`, and `_PENUM`
- `stdio.h` Standard I/O functions and prototypes
- `stdlib.h` General utility functions and prototypes
- `string.h` Functions and prototypes for strings manipulations
- `ctype.h` Functions and prototypes for characters manipulations (in development)

These files are stored in the `1dc/cc1dc/lib/include` directory.

To use standard C functions included in the 1DC standard library, specify the required header files in your source code.

1.3 Startup Processing

The standard startup routine in `crt0.io` contains the processing performed first when a program is executed. This processing specifies the initial settings required for program execution and starts the user-defined `main` function.

Basic procedures to run a program when using the low level debugger dbimap:

- (dbimap) creset Reset
- (dbimap) open test.io Opens the program.
- (dbimap) run Runs the program and waits for completion.
- (dbimap)

Or

- (dbimap) creset Reset
- (dbimap) open test.io Opens the program.
- (dbimap) Runs the program and waits for completion.
- (dbimap) start Starts the program.
- (dbimap) cont Restarts the program

1.4 Program Creation Procedure

This section describes the procedure to develop a typical 1DC program.

1.4.1 Creating a source file

Create a source file using 1DC or assembly language. The first function called in the program must use the name below and have external linkage.

- For 1DC: `main` (Do not declare the function as `static`)
- For assembly: `Fmain` (Do not forget to declare the function as `global`)

1.4.2 Assembling or compiling

Use `asimap`, assembler of the IMAP Series processor, to assemble the program, or use the 1DC compiler `cc1dc` to compile it. Note that, if you used assembly language and included files, the path where those files exist must be specified during assembly.

Example of compiling:
```
cc1dc -c file.lc
```

Example of assembling:
```
asimap -i -I 1dc/cc1dc/lib/macro/imap file.is -o file.io
```

1.4.3 Linking

Use the cc1dc compiler or the ldimap linker to link the assembled or compiled object file with the standard library. (If cc1dc* is used, ldimap* is called)

When using the compiler:
```
cc1dc file1.io file2.io
```

When using the linker:
```
ldimap 1dc/cc1dc/imap/lib/processor-core-name/crt0.io
file1.io
file2.io 1dc/cc1dc/imap/lib/processor-core-name/lib1dc.ia
```

When using assembly language, the startup routine can be included in a specified source file. In this case, `crt0.io` does not have to be linked with.

When using the compiler, you can either compile and link at the same time, or compile and then explicitly specify a linker for linking.

Example of compiling and linking:
```
cc1dc file1.lc file2.lc
```

Example of only compiling:
```
cc1dc -c file1.lc file2.lc
```

Performing the above procedure generates an executable object file.

2 Overview of functions

The standard 1DC compiler provides functions (standard 1DC functions) which are considered particularly useful for IMAP Series processor programs. These functions are grouped into the following main categories:

- Memory functions
  - o Functions that transfer data between IMEM and EMEM (line transfer)
  - o Functions that dynamically allocate memory
  - o Functions for performing ROI transfers
- Operation functions
  - o Functions for bit manipulation, transfers, transposition, accumulation, absolute values, square roots, maximums, and minimums
- MP mode functions
  - o Basic functions and functions for sending and receiving messages

In addition, this document uses the following data type abbreviations

Data type	Abbreviation
unsigned char	uchar
unsigned int	uint
unsigned long	ulong

2.1 External Memory Transfer Functions ((Line Transfer)

Functions for data transfer between IMEM and EMEM (a process referred as *line transfer*) are divided into functions which transfer lines in foreground and those which transfer lines in background.

These functions are used to perform a burst transfer of a defined number of lines between IMEM and EMEM (each line contains one byte per PE, so that the total amount of line data in bytes is equal to the number of PEs).

Line transfers are performed for blocks of data (each block contains 4 lines, which is equal to the number of PEs $\times$ 4 bytes).

Therefore, if the number of lines to transfer is not a multiple of 4, the hardware controls the transfer to ensure that the defined number of lines is transferred.

In addition, when a transfer extends across blocks, you must specify whether you want to automatically calculate the increment (in blocks), or to add a user-defined number of blocks to determine the starting address of the next block.

For automatic calculation, one block is added to determine the starting address used for the next block transfer.

2.1.1 Foreground line transfer functions

The execution of foreground line transfer operations and any other operation processing cannot overlap. All transfer requests are issued at once, and other processing is performed after the transfer is complete. The following functions are used:

- lx_ememrw — Transfers lines between EMEM and IMEM (when 1 is assigned to the wait parameter).
- lx_ememrw_hp — High priority version of lx_ememrw

The following macro functions are defined using the lx_ememrw[_hp] functions in imapio.h. Because these macro functions define constants for frequently used parameters, less parameters have to be defined as compared with directly calling the lx_ememrw[_hp] functions.

- lxEmemrd — Transfers lines from EMEM to IMEM.
- lxEmemrd_offset — Transfers lines from EMEM to IMEM (with a defined block offset)
- lxEmemrd_blk — Transfers lines from EMEM to IMEM (in vertical block units)
- lxEmemrd_hp — High priority version of lxEmemrd
- lxEmemrd_hp_offset — High priority version of lxEmemrd_offset
- lxEmemrd_hp_blk — High priority version of lxEmemrd_blk
- lxEmemwr — Transfers lines from IMEM to EMEM.
- lxEmemwr_offset — Transfers lines from IMEM to EMEM (with a defined block offset)
- lxEmemwr_blk — Transfers lines from IMEM to EMEM (vertically)
- lxEmemwr_hp — High priority version of lxEmemwr
- lxEmemwr_hp_offset — High priority version of lxEmemwr_offset
- lxEmemwr_hp_blk — High priority version of lxEmemwr_blk

2.1.2 Background line transfer functions

Background line transfer functions only issue transfer requests.

When using these functions, data needed for next processing is automatically transferred while other data is processed.

Therefore, data transfer time is concealed by overlapping background data transfer and data processing.

The background line transfer functions are as follows:

- lx_ememrw — Transfers lines between EMEM and IMEM (when 0 is specified for the wait parameter).
- lx_ememrw_wno — Transfers lines between EMEM and IMEM (requires the specification of a window register).
- lx_ememrw_hp — High priority version of lx_ememrw
- lx_ememrw_hp_wno — High priority version of lx_ememrw_hp

The following macro functions are defined using the lx_ememrw[_hp][wno] functions in imapio.h. Because these macro functions define constants for frequently used parameters, less parameters have to be specified as compared with directly calling the lx_ememrw[_hp][wno] functions.

- lxEmemrd_start — Transfers lines from EMEM to IMEM.
- lxEmemrd_offset_start — Transfers lines from EMEM to IMEM (with a block offset defined).
- lxEmemrd_blk_start — Transfers lines from EMEM to IMEM (in vertical block units).
- lxEmemrd_hp_start — High priority version of lxEmemrd_start
- lxEmemrd_hp_offset_start — High priority version of lxEmemrd_offset_start
- lxEmemrd_hp_blk_start — High priority version of lxEmemrd_blk_start
- lxEmemrd_end — Waits for the transfer from EMEM to IMEM to finish and then releases the window register.

- lxEmemwr_start — Transfers lines from IMEM to EMEM.
- lxEmemwr_offset_start — Transfers lines from IMEM to EMEM (with a block offset defined).
- lxEmemwr_blk_start — Transfers lines from IMEM to EMEM (in vertical block units).
- lxEmemwr_hp_start — High priority version of lxEmemwr_start
- lxEmemwr_hp_offset_start — High priority version of lxEmemwr_offset_start
- lxEmemwr_hp_blk_start — High priority version of lxEmemwr_blk_start
- lxEmemwr_end — Waits for the transfer from IMEM to EMEM to finish and then releases the window register

2.1.3 Line transfer functions for which a window register set is defined

Whenever one of the line transfer functions described in the previous sections starts a transfer, a hardware resource called *window register set* is internally allocated, then released after the transfer terminates.

When performing repeatedly background line transfers in a loop, transfers are more efficient if the same window register is used for each transfer instead of allocating and releasing the window register set for each transfer.

The line transfer functions below are used by specifying a user-allocated window register set as parameter. Note that the transfer direction is determined based on whether the window register set number is odd or even.

- lx_wopen — Allocates a window register set (for which the transfer direction is specified as a parameter).
- lx_wclose — Releases a window register set allocated using lx_wopen.
- lxEmemrw_start — Transfers lines between EMEM and IMEM.
- lxEmemrw_offset_start — Transfers lines between EMEM and IMEM (with a block offset defined).
- lxEmemrw_blk_start — Transfers lines between EMEM and IMEM (in vertical block units).
- lxEmemrw_hp_start — High priority version of lxEmemrw_start
- lxEmemrw_hp_offset_start — High priority version of lxEmemrw_offset_start
- lxEmemrw_hp_blk_start — High priority version of lxEmemrw_blk_start
- lxEmemrw_end — Waits for a transfer between EMEM and IMEM to finish.

The procedure for using the above functions is as follows:

1) Allocate the window register set wno by defining wno=lx_wopen(...).

2) When defining wno for functions such as lxEmemrw_start, do the following:

   2-1) Wait for a line transfer using wno.

   2-2) Define settings for the window registers in wno that are necessary to transfer the lines.

   2-3) Issue a line transfer request.

3) Perform any other processing here. Lines are transferred in the background of this processing.

4) Wait for the transfer to finish by defining lxEmemrw_end().

5) Release the window register set wno back to the system by defining lx_wclose(wno).

The following shows an example of the code to use:

```
int direction= 0;       // Specify 0 to transfer from EMEM to IMEM or 1 to transfer
                        // from IMEM to EMEM.

int lines = 240;        // Number of lines to transfer

int wno = Ix_wopen(direction);  // Allocate the window register set.

for ( ... ; ... ; ... ) {

        IxEmemrw_start (.....);         // Wait for the transfer to finish, set up the
                                        // window register, and issue a transfer //request.

        //  ----------------------------------------------------------------

        //  If operation processing is specified here, a background

        //  transfer is performed, and, if there is no such processing,

        //  a foreground transfer is performed.

        //  ----------------------------------------------------------------

}

IxEmemrw_end();     // Wait for the transfer to end.

Ix_wclose(wno);     // Release the window register set.
```

2.1.4 Functions to directly set up line transfers

Before starting either a foreground or a background transfer, a window register set must be internally allocated, then released when the transfer terminates.

In addition, the following information must be specified before a transfer for each register in the window register set:

- The transfer source address
- The transfer destination address
- The address increment for each block
- The number of lines to transfer
- The left and right edge mask settings
- The transfer direction (either from EMEM to IMEM or from IMEM to EMEM)

The following functions can be used to directly define the above settings without using the standard line transfer functions. For example, the effects of the previously described Ix_ememrw function can be achieved by calling the following functions:

- Ix_ememrw_init — Sets up the window registers for transferring lines between EMEM and IMEM.
- Ix_ememrw_req — Issues a transfer request for a specified window register.
- Ix_ememrw_hp_req — High priority version of Ix_ememrw_req

2.1.5 Low level functions that support line transfers

Actually, the previously described functions for transferring lines use the following inline functions:

- Ix_wset0 — Assigns the value of window register 0 in the specified set.
- Ix_wget0 — Extracts the value of window register 0 in the specified set.
- Ix_wset1 — Assigns the value of window register 1 in the specified set.
- Ix_wget1 — Extracts the value of window register 1 in the specified set.
- Ix_wset2 — Assigns the value of window register 2 in the specified set.
- Ix_wget2 — Extracts the value of window register 2 in the specified set.
- Ix_wreq0 — Issues a high priority transfer request for window registers in the specified set.
- Ix_wreq1 — Issues a normal priority transfer request for window registers in the specified set.

For example, the IxEmemrw_end(wno) function is actually written using the following sort of code, which uses the Ix_wget2 function:

```
while (Ix_wget2(wno) & 0xffff) ix_nopwait();
```

2.1.6 Line transfer functions that use division

Line transfer function that use division automatically divide images that are wider than the number of PEs into multiple data blocks equivalent to the width of the number of PEs. It performs line transfers to sequentially read each block into IMEM, uses a user-specified function, and then performs line transfer to write the results of processing performed in the IMEM area to EMEM. If there is space in the IMEM heap, the above line transfer operations are automatically performed in the background.

- IxEmemrw_app — Divides an image into data blocks equivalent to the width of the number of PEs, and then uses a user-specified function.
- IxEmemrw_Lno — Macro function for the IxEmemrw_app function when performing local neighborhood operations
- IxEmemrw_Po — Macro function for the IxEmemrw_app function when performing point operations
- IxEmemrw_So — Macro function for the IxEmemrw_app function when performing statistical operations

2.2 Others

Several functions for special purpose and inline functions including accumulation adding are available. For details about the purpose of each function, see the following chapters.

2.2.1 Dynamic memory allocation

- lx_iheap — Dynamically allocates memory from the IMEM heap (by only performing a pointer operation)
- lx_iheap_return — Releases memory dynamically allocated from the IMEM heap (by only performing a pointer operation)
- lx_iheap_overflow — Identifies where a standard program stopped based on insufficient space in the IMEM heap
- lx_eheap — Dynamically allocates memory from the EMEM heap (by only performing a pointer operation)
- lx_eheap_return — Releases memory dynamically allocated from the EMEM heap (by only performing a pointer operation)
- lx_eheap_overflow — Identifies where a standard program stopped based on insufficient space in the EMEM heap
- ix_malloc — Dynamically allocates memory from the DMEM heap (which is managed using a list)
- ix_calloc — Dynamically allocates memory from the DMEM heap (which is managed using a list) and initializes the memory block to 0
- ix_free — Releases memory dynamically allocated from the DMEM heap (which is managed using a list)
- ix_realloc — Changes the amount of allocated memory (which is managed using a list)

2.2.2 ROI transfer and cache operations

- ix_civd_all — disables all data caches
- ix_civda — disables a cache line hit using the specified address
- ix_civda2 — disables the cache lines hit using the specified range of addresses
- ix_cwb_all — writes back all dirty data caches
- ix_cwba — writes back dirty data in the cache if it is at the specified address
- ix_cwba2 — writes back dirty cache lines in the specified range of addresses
- ix_cwbivda — Executes ix_cwba() and then ix_civda()
- ix_roi_read — Transfers data in the rectangular area that starts at the specified address to the data cache
- ix_roi_write — Transfers data in the data cache to the rectangular area that starts at the specified address

2.2.3 Basic functions for the MP mode

• lx_cp_start_pu	Starts up PU
• lx_cp_start_multipu	Starts up PUs
• lx_fork	Inline function used to start up PUs
• lx_fork_again	Waits for multiple PUs to stop, then executes the lx_forkp or lx_forkd function and restarts the PUs as necessary
• lx_forkp	Inline function used to fill the explicit program cache before a PU starts up
• lx_forkd	Inline function used to fill the explicit data cache before a PU starts up
• lx_fork_init	PU startup inline function
• lx_start_mode	Switches the operation mode
• lx_mutex_init	Initializes the lock control variable
• lx_mutex_destroy	Discards the lock control variable
• lx_mutex_lock	Locks the lock control variable
• lx_mutex_unlock	Unlocks the lock control variable
• ix_pu_id	Macro function that returns the IDs of execution resources (PU, CP)
• ix_total_pu_number	Macro function that returns the total number of PUs
• ix_max_total_pu_number	Macro function that returns the maximum number or PUs that the program supports

2.2.4 Message sending and receiving functions for MP mode

- ix_msg_header — Macro function that generates an asynchronous response message header
- ix_msg_header_intr — Macro function that generates an interrupt message header
- ix_msg_header_sync — Macro function that generates a synchronized data message header
- ix_msg_id — Macro function that returns the message ID
- ix_msg_receiver — Macro function that returns information about the message recipient
- ix_msg_sender — Macro function that returns information about the message sender
- ix_msg_type — Macro that returns information about the message type
- ix_recv — Inline function used to receive a message header
- ix_recv_any — Macro function used to receive a message from any sender
- ix_recv_msg — Receives a message
- ix_recv_msg_blk — Sequentially receives a message as multiple 32-bit words
- ix_send — Inline function that sends a message header
- ix_send_any — Macro function that sends a message to any destination (broadcast)
- ix_send_any_sync — Macro function that sends a message to any destination (broadcast) and uses synchronized message requests
- ix_send_msg — Sends a message
- ix_send_msg_blk — Sequentially sends a message as multiple 32-bit words

2.2.5 Sep data operations

- IxAddsep — Sums up the 8 to 16 bits of sep data for all PEs (and returns a long value).
- IxAddsepl — Sums up the 32 bits of sep data for all PE (and returns a long value).
- IxAddsepll — Sums up the 32 bits of sep data for all PEs (and returns a long long value)
- IxCountsep — Returns the number of PEs that satisfy a specified condition
- IxGathersep — Extracts sep data (16 or 32 bits) that satisfies a specified condition
- IxGetleftsep — Extracts the leftmost sep data (16 bits) that satisfies a specified condition
- IxGetrightsep — Extracts the rightmost sep data (16 bits) that satisfies a specified condition
- IxOrdersep — Sorts sep data (16 or 32 bits)
- IxMaxsep — Extracts the maximum value from sep data (16 bits)
- IxMinsep — Extracts the minimum value from sep data (16 bits)
- IxPack8si — Packs (binarizes) the 8 pixels adjacent to a position in a 16-bit sep array that differs for each PE into 1 byte
- IxPack8s — Packs (binarizes) the 8 pixels adjacent to a position in an 8-bit sep array that differs for each PE into 1 byte

2.2.6 Data transfers between sep-type and scalar data

- IxDmem2imem_comm — Stores the contents of the CP array (in 16-byte units) in the IMEM array (same value for all PEs)
- IxDmem2imem{_u1/4} — Going through PEs, sequentially stores the contents of the CP array in the IMEM array
- IxImem2dmem{_u1/4} — Going through PEs, stores the specified number of lines in the IMEM array in the CP array
- IxDmem2PE{_u1} — Prioritizing PEs, copy the contents of the CP array to the IMEM array
- IxPE2dmem{_u1} — Prioritizing PEs, copy the contents of the IMEM array to the CP array
- IxPEs2dmem{_u1} — Prioritizing PEs, copy the contents of the IMEM array to the CP array (PEs can have different numbers of elements.)

2.2.7 Sep data transfers

- IxRemap_rx1_1xr — Transposes the contents of a sep array (8, 16, or 32 bits), using the specified number of PEs as unit
- IxRemap_rec_1pe — Transfers data between a specified rectangular area in a sep array (8, 16, or 32 bits) and one PE or vice versa
- IxRemap_sqr_sqr — Rotates a square area of the contents of a sep array (8, 16, or 32 bits) by 90 degrees or transposes the area
- IxRot90 — Rotates the contents of a sep array (8, 16, or 32 bits) by 90 degrees

2.2.8 Inline functions for absolute values calculation

- ix_asub — Absolute difference (for a PE array)
- Ix_asub_u2 — Absolute difference (for a PE array)
- Ix_asub_s2 — Absolute difference (for a PE array)
- ix_asub_u2 — Absolute difference
- ix_asub_s2 — Absolute difference
-

2.2.9 Inline functions for accumulator addition

- Ix_acc_mul8 — Adds the lower 16 bits of a product to the accumulation register ra10 (for a PE array)
- Ix_acc_mul_u2_u4 — Adds the lower 16 bits of a product to the accumulation register ra10 (for a PE array)
- Ix_acc_mul_s2_s4 — Adds the lower 16 bits of a product to the accumulation register ra10 (for a PE array)
- ix_acc_mul_u2_u4 — Adds the lower 16 bits of a product to the accumulation register cra10
- ix_acc_mul_s2_s4 — Adds the lower 16 bits of a product to the accumulation register cra10
- Ix_acc_mul_u2_u8 — Adds a 32-bit product to the accumulation register ra3210 (for a PE array)
- Ix_acc_mul_s2_s8 — Adds a 32-bit product to the accumulation register ra3210 (for a PE array)
- ix_acc_mul_u2_u8 — Adds a 32-bit product to the accumulation register cra3210
- ix_acc_mul_s2_s8 — Adds a 32-bit product to the accumulation register cra3210
- Ix_acc_read_u8 — Returns the value stored in the accumulation register ra3210 and simultaneously clears this register (for a PE array)
- ix_acc_read_u8 — Returns the value stored in the accumulation register and simultaneously clears this register

- lx_acc_asub_u2 — Adds an absolute difference to the accumulation register ra10 (for a PE array)
- lx_acc_asub_s2 — Adds an absolute difference to the accumulation register ra10 (for a PE array)
- ix_acc_asub_u2 — Adds an absolute difference to the accumulation register cra10
- ix_acc_asub_s2 — Adds an absolute difference to the accumulation register cra10
- lx_acc_s2_s4 — Adds a defined 16-bit value to the accumulation register ra10 (for a PE array)
- ix_acc_s2 — Adds a defined 16-bit value to the accumulation register cra10
- lx_acc_s4_s8 — Adds a defined 32-bit value to the accumulation register ra3210 (for a PE array)
- ix_acc_s4_s8 — Adds a defined 32-bit value to the accumulation register cra3210
- lx_acc_read — Returns the value stored in the accumulation register ra10 and simultaneously clears this register (for a PE array)
- lx_acc_read_u4 — Returns the value stored in the accumulation register and simultaneously clears this register (for a PE array)
- ix_acc_read_u4 — Returns the value stored in the accumulation register cra10 and simultaneously clears this register

2.2.10 Inline functions for saturation arithmetic

- lx_sadd — Performs saturated addition of 8-bit values for a PE array (where 255 is the saturated value)
- lx_sadd_u2_255 — Performs saturated addition of 16-bit values for a PE array (where 255 is the saturated value)
- lx_sadd_u2_65535 — Performs saturated addition of 16-bit values for a PE array (where 65535 is the saturated value)
- ix_sadd_u2_255 — Performs saturated addition of 16-bit values (where 255 is the saturated value)
- ix_sadd_u2_65535 — Performs saturated addition of 16-bit values (where 65535 is the saturated value)
- lx_ssub — Performs saturated subtraction of 8-bit values for a PE array (where 0 is the saturated value)
- lx_ssub_u2 — Performs saturated subtraction of 16-bit values for a PE array (where 0 is the saturated value)
- ix_ssub_u2 — Performs saturated subtraction of 16-bit values (where 0 is the saturated value)

2.2.11 Inline functions for maximization and minimization

- lx_max3 — Returns the maximum of three 8-bit values (for a PE array)
- lx_max_u2 — Returns the maximum of two 16-bit values (for a PE array)
- ix_max_u2 — Returns the maximum of two 16-bit values
- lx_min3 — Returns the minimum of three 8-bit values (for a PE array)
- lx_min_u2 — Returns the minimum of two 16-bit values (for a PE array)
- ix_min_u2 — Returns the minimum of two 16-bit values

2.2.12 Inline functions for bit manipulation

- lx_bitr — Returns the value at the specified bit position of the specified value (for a PE array).
- lx_bitr_u2 — Returns the value at the specified bit position of the specified value (for a PE array).
- ix_bitr_u2 — Returns the value at the specified bit position of the specified value.
- lx_bts_u2 — Sets defined bit for a PE array
- ix_bts_u2 — Sets defined bit for a PE array
- lx_skz_u2 — Returns the number of 0s in a defined value up to the first bit position where there is a 1, starting with the MSB (for a PE array)
- ix_skz_u2 — Returns the number of 0s in a defined value up to the first bit position where there is a 1, starting with the MSB
- lx_pack_u2 — Packs the values of 8 adjacent pixels of a binarized image into 1 byte (for a PE array)
- lx_pack — Packs the values of 8 adjacent pixels of a binarized image into 1 byte (for a PE array)

2.2.13 Inline functions for special registers read or write

- lx_spec_read — Returns the value of the defined special PE register (for a PE array)
- lx_spec_write — Writes a value to the defined special PE register (for a PE array)
- ix_spec_read — Returns the value of the defined special register
- ix_spec_write — Writes a value to the defined special register

2.2.14 Integer square root functions

- ixSqrt — Returns the uint square root
- ixSqrtl — Returns the ulong square root
- ixSqrtll — Returns the ulong long square root
- IxSqrtsep — Returns the sep uint square root (for a PE array)
- IxSqrtsepl — Returns the sep ulong square root (for a PE array)
- IxSqrtsepll — Returns the sep ulong long square root (for a PE array)

2.2.15 Inline functions for port I/O

- Ix_in — Reads from a port
- Ix_ou — Writes to a port

2.2.16 Inline functions for floating point arithmetic

- ix_fsqrt — Returns the square root of a floating point value.
- ix_fmodf — Returns the remainder after dividing a floating point value.

2.2.17 Inline and normal functions to prohibit, permit and enable interrupts

- ix_idis — Prohibits subsequent interrupts.
- ix_iena — Cancels the specification of prohibited interrupts by the most recent call to the ix_idis function.
- ix_interrupt_return — Enables the subsequent calling of interrupt functions.

2.2.18 Inline functions for data transfer between connected PEs

- Ix_mvlc — Shifts a sep variable that is 2 bytes or less between connected PEs (right to left).
- Ix_mvrc — Shifts a sep variable that is 2 bytes or less between connected PEs (left to right).
- Ix_mvlc_u4 — Shifts a 4-byte sep variable between connected PEs (right to left).
- Ix_mvrc_u4 — Shifts a 4-byte sep variable between connected PEs (left to right).
- Ix_mvlz — Shifts a zero-supplied-type sep variable that is 2 bytes or less between PEs (right to left).
- Ix_mvrz — Shifts a zero-supplied-type sep variable that is 2 bytes or less between PEs (left to right).
- Ix_mvlz_u4 — Shifts a 4-byte zero-supplied-type sep variable between PEs (right to left).
- Ix_mvrz_u4 — Shifts a 4-byte zero-supplied-type sep variable between PEs (left to right).

2.2.19 Operation stop function

- ix_halt — This function stops execution. When called from a PU, this function also defines the LSB of its parameter for the mpjoin bit at the position corresponding to the PU

2.2.20 Debugger command execution function

- ix_exec — This function executes a debugger command defined by a character string. This allows the execution of debugger commands that include data load to EMEM or IMEM, EMEM or IMEM data save to files, and interrupt timers start.

2.2.21 Functions for data copy within memory and values specification

- ix_memset — Writes the specified number of characters to the area starting at the defined DMEM address
- ix_memcpy — Copies data within DMEM
- ix_memcpy16 — Copies data within DMEM (between 4-byte aligned addresses in 16-byte units)
- lx_memset — Writes the specified number of lines to the area starting at the defined IMEM address
- lx_memcpy — Copies data within IMEM
- lx_memcpy16 — Copies data within IMEM (between 4-byte aligned addresses in 16-line units)
- lxEmem2emem — Copies data within EMEM

2.2.22 Basic image processing functions

- lxPk8tlup — Uses the packed value of 8 adjacent pixels to reference a table for binary image processing
- lxVote_u1u2 — Polls the values in a sep array as indexes
- lxVoteCollect_u2u4 — Totals various specified types of data in a sep array

2.2.23 Other functions

- ix_nopwait — wait function used to define macros (not moved during compiler optimization)
- lx_pe_id — Returns a PE number (for a PE array).
- lx_real_pe_number — Macro that returns the number of usable PEs.
- lx_total_pe_number — Returns the total number of PEs.
- lx_max_total_pe_number — Returns the maximum number of PEs usable by the program
- lxPtime — Measures the processing time

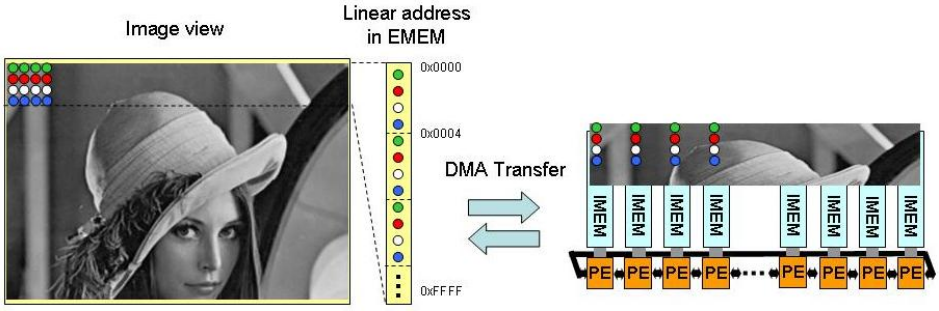
3 Function details

3.1 Functions to transfer data (lines) between EMEM and IMEM

As shown in the figure below, transferring lines means performing a DMA transfer to transfer lines between EMEM and IMEM.

During a line transfer, 4 sequential bytes of data starting at an address (which is multiple of 4 in EMEM) are stored in 4 sequential bytes of a PE (which start at a multiple of 4), and one transfer is performed for each PE.

Note that the unit for a user-specified transfer is a line (which consists of one byte of data for each PE) and multiples of 4 do not have to be used. Similarly, the addresses at which an EMEM or IMEM transfer starts do not have to be multiples of 4, but using multiples of 4 improves the line transfer efficiency.



Line transfer functions are executed by calling the lx_ememrw_init macro function described below.

This macro function allows parameters to be defined for various operations performed when transferring lines. It is used to define the parameters for each window register included in the eight window register sets (see table below).

Functions other than the lx_ememrw_init macro function are available as macro functions that define constants for seldom used parameters or as normal functions.

Table 1 Register set format for line transfer

Name	No of bits	Value range	Bit position	Description
wr0	12	0 to 4095	[11..0]	IMEM starting address (unit: lines, which consists of one byte per PE)
	3	-	[14..12]	Ignored when defined and 0 when read
	1	0 or 1	[15]	Defines the offset addition mode
	7	0 to 127	[22..16]	Defines the left edge mask (write prohibited area) Unit: number of PEs However, only [21..16] are considered in case of 64-PE configuration, and only [20..16] are considered in case of 32-PE configuration
	1	-	[23]	Ignored when defined and 0 when read
	7	0 to 127	[30..24]	Defines the right edge mask (write prohibited area) Unit: number of PEs
	1	-	[31]	Ignored when defined and 0 when read
wr1	28	0 to 256M	[27..0]	EMEM starting address (unit: bytes)
wr2	16	0 to 65535	[15..0]	Total number of lines to transfer
	16	0 to 65535	[31..16]	Block offset (unit: blocks which consist of 4*PENO bytes)

3.1.1 lx_ememrw_init

Synopsis

```
#include <imapio.h>
void lx_ememrw_init(iadr, eadr, maskl, maskr, mode, boffset, lines, wno);
separate void *iadr;  /* IMAP internal memory address */
ulong eadr;           /* external memory absolute address */
uint maskl;           /* number of PE to skip from left  (ex. 0) */
uint maskr;           /* number of PE to skip from right (ex. 0) */
uint mode;            /* block offset addition mode (0:auto 1:request) */
uint boffset;         /* block offset value */
uint lines;           /* number of lines to transfer */
uint wno;             /* window-register ID(even: EMEM -> IMEM, odd: IMEM -> EMEM */
```

Description This function is used to define settings for the wno window registers which are used to transfer lines between EMEM and IMEM. This function is only used to initialize settings for transferring lines and cannot be used on its own to actually perform the transfer.

- iadr defines the IMEM line address.
- eadr defines the absolute EMEM starting address (defines the value by which the number of PEs is multiplied for an outside separate type pointer)
- maskl defines the number of PEs to skip from the left edge to each line when storing or reading data in IMEM.
- maskr defines the number of PEs to skip from the right edge to each line when storing or reading data in IMEM

Note that, if a value greater than or equal to the number of PEs (= PENO) is assigned for maskl or maskr, it is assumed that maskl%PENO or maskr%PENO is assigned

- mode parameter defines the block offset addition method.
  - o When mode = 0: after 4 lines (the word data for the number of PE, where each word size is 32 bits)
  - o When mode = 1: At each new transfer request (the lx_ememrw_req function is called once)

Note For the first calculation, if eadr is not a multiple of 4, the (4 – (eadr%4)) lines starting at eadr are transferred, and then *boffset* × *4* × *the number of PEs* is added to *eadr& 0xfffffffc*

- boffset defines the block offset described below.

A product of *boffset* × *4* × *the number of PEs* is added to the current absolute EMEM starting address (in the units specified by this parameter), and the obtained address is used as the absolute EMEM starting address for the next line transfer

- lines defines the number of lines to transfer. Up to 65635 can be specified for lines, but only up to 4095 lines can be transferred for one transfer request (for details, see the descriptions of lx_wreq0 and lx_wreq1)

- wno defines the ID of the specified window register set. An even ID indicates a transfer from EMEM to IMEM (read), and an odd ID indicates a transfer from IMEM to EMEM (write). wno is normally obtained using the lx_wopen function

Transfers are normally performed extending across all PEs in word units (where one word is 4 bytes). lines does not have to be a multiple of 4, but, if it is a multiple of 4 (such as 4, 8, 16, or 32), transfer speed might increase. Transfer speed might similarly increase if eadr is a multiple of 4, maskl is 3 or less, and maskr is 0.

This function is included among the macro definitions in the following header file:

1DC/cc1dc/lib/include/imap/imapio.h

3.1.2 lx_ememrw{_hp}

Synopsis

```
#include <imapio.h>
int lx_ememrw{_hp}(iadr,eadr,maskl,maskr,boffset,lines,direction,wait);
separate void *iadr;  /* IMEM address */
ulong eadr;           /* EMEM absolute address */
uint maskl;           /* number of PE to skip from left  (ex. 0) */
uint maskr;           /* number of PE to skip from right (ex. 0) */
uint boffset;         /* block offset value */
uint lines;           /* number of lines to transfer */
uint direction;       /* 0: EMEM -> IMEM (read), 1: IMEM -> EMEM (write) */
uint wait;            /* 0: background transfer, otherwise: foreground transfer */
```

Description These functions are used to transfer data (lines) between EMEM and IMEM.

- direction defines the transfer direction (as 0 for a transfer from EMEM to IMEM or 1 for a transfer from IMEM to EMEM)
- Up to 4,095 can be specified for lines.
- If 0 is assigned to wait, a background transfer is performed, and, if any other value is assigned, a foreground transfer is performed (and no other operation is done until the end of the transfer)

To transfer lines, window registers, which are system resources, must be used, and these functions allocate a new set of window registers and return the window register number. If the return value is from 0 to 7, the window registers were successfully allocated, and, if the return value is -1, the window registers could not be allocated (which means that there were no empty window registers).

For a background transfer (when 0 is specified for wait), these functions end immediately after issuing a transfer request, and the actual transfer is performed in the background.

When performing a background transfer, call the lx_wwait function to separately confirm that the transfer has finished before referencing EMEM or IMEM data at the transfer destination.

Be sure to assign the value returned by these functions when calling the lx_wwait function.

In addition, after the end of the transfer, do not forget to return the window register resources to the system if they are no longer necessary.

Note that, if 1 is specified for the close parameter of the lx_wwait function, these resources are automatically returned to the system when the transfer terminates.

For a foreground transfer (when 1 is specified for wait), the allocated window register resources are automatically returned to the system when the transfer terminates.

For the meanings of other parameters, see the description of the lx_ememrw_init register.

The parameters descriptions for lx_ememrw_hp are the same as those for lx_ememrw, but the lx_ememrw_hp transfer requests are placed in a higher priority queue.

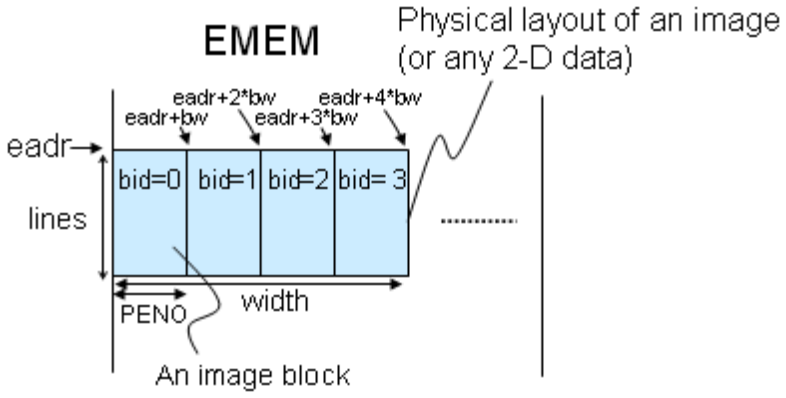
3.1.3 lx_ememrw{_hp}_blk

Synopsis

```
#include <imapio.h>
int lx_ememrw_blk(iadr, eadr, maskl, maskr, bid, bw, width, lines, direction, wait);
int lx_ememrw_hp_blk(iadr, eadr, maskl, maskr, bid, bw, width, lines, direction, wait);
separate void *iadr;  /* IMEM address */
ulong eadr;           /* EMEM absolute address */
uint maskl;           /* number of PE to skip from left (ex. 0) */
uint maskr;           /* number of PE to skip from right (ex. 0) */
uint bid;             /* block id */
uint bw;              /* block width*/
uint width;           /* image width */
uint lines;           /* number of lines to transfer */
uint direction;       /* 0: EMEM->IMEM (read), 1: IMEM->EMEM (write) */
uint wait;            /* 0: background transfer, otherwise: foreground transfer */
```

Description lx_ememrw_blk differs from lx_ememrw in that bid is used to define the position of the transfer target image block instead of offset, bw can be used to define the distance between image blocks (in pixels in a 2D layout), and width can be used to define the width of images

The meanings of bid and bw are shown in the figure below. Note that, no matter what is assigned to bw, the width of the image block that is actually loaded into IMEM is PEN0 (the number of PEs)



The parameters descriptions for lx_ememrw_hp_blk are the same as those for lx_ememrw_blk, but the lx_ememrw_hp_blk transfer requests are placed in a higher priority queue.

This function is included among the macro definitions in the following header file:

`1DC/cc1dc/lib/include/imap/imapio.h`

3.1.4 lx_ememrw{_hp}_wno

Synopsis

```
#include <imapio.h>
void lx_ememrw_wno(iadr, eadr, maskl, maskr, boffset, lines, wno);
void lx_ememrw_hp_wno(iadr, eadr, maskl, maskr, boffset, lines, wno);
separate void *iadr;  /* IMEM address */
ulong eadr;           /* EMEM absolute address */
uint maskl;           /* number of PE to skip from left  (ex. 0) */
uint maskr;           /* number of PE to skip from right (ex. 0) */
uint boffset;         /* block offset value */
uint lines;           /* number of lines to transfer */
uint wno;             /* window register set number */
```

Description The parameters for lx_ememrw_wno differ from those for lx_ememrw in that wno is defined instead of direction and wait.

Operations are performed to check whether the transfer using wno has finished before starting to set up the transfer.

The transfer direction is specified according to whether wno is odd or even. If wno is even, data is transferred from EMEM to IMEM, and, if wno is odd, data is transferred from IMEM to EMEM. The function ends after only issuing the transfer request.

For wno, lx_wopen defines the window register set number obtained. Note that the parameter for the lx_wopen function defines the line transfer direction.

If there is already a transfer using wno, these functions wait for the transfer to finish before setting up another transfer and issuing a transfer request.

To check whether a transfer started by these functions has finished, either reissue these functions or use the lxEmemwr_end or lx_wwait function. Finally, if wno is no longer necessary, use the lx_wclose function to return it to the system.

The lx_ememrw_hp_wno function is a high priority transfer version of the lx_ememrw_wno function, and transfer requests issued by lx_ememrw_hp_wno are stored in a higher priority transfer queue.

3.1.5 Ix_ememrw{_hp}_wno_blk

Synopsis

```
#include <imapio.h>
void Ix_ememrw_wno_blk(iadr, eadr, maskl, maskr, bid, bw, width, lines, wno);
void Ix_ememrw_hp_wno_blk(iadr, eadr, maskl, maskr, bid, bw, width, lines, wno);
separate void *iadr;  /* IMEM address */
ulong eadr;           /* EMEM absolute address */
uint maskl;           /* number of PE to skip from left  (ex. 0) */
uint maskr;           /* number of PE to skip from right (ex. 0) */
uint bid;             /* block id */
uint bw;              /* block width*/
uint width;           /* image width */
uint lines;           /* number of lines to transfer */
uint wno;             /* window register set number */
```

Description

Ix_ememrw_wno_blk differs from Ix_ememrw_wno in that bid defines the position of the transfer target image block (instead of offset), bw defines the blocks width, and width can be used to define the images width.

Ix_ememrw_hp_wno_blk is a high priority transfer version of Ix_ememrw_wno_blk, and transfer requests issued by Ix_ememrw_hp_wno_blk are stored in a higher priority transfer queue.

This function is included among the macro definitions in the following header file:

`1DC/cc1dc/lib/include/imap/imapio.h`

3.1.6 lx_ememrw{_hp}_req

Synopsis

```
#include <imapio.h>
void lx_ememrw_req(wno, lines);
void lx_ememrw_hp_req(wno, lines);
uint wno;          /* window register ID */
uint lines;        /* number of lines to transfer */
```

Description These functions issue transfer requests using the window register set wno to transfer data between EMEM and IMEM. Up to 4,095 can be assigned to lines.

These functions are normally used to start transferring data, after initializing the window register set number wno with lx_ememrw_init.

wno is usually obtained using the lx_wopen() function.

To check whether the transfer has finished for an issued transfer request, use lx_wwait function.

Except for the fact that it issues higher priority transfer requests, the lx_ememrw_hp_req function is the same as the lx_ememrw_ req function.

This function is included among the macro definitions in the following header file:

`1DC/cc1dc/lib/include/imap/imapio.h`

3.1.7 lx_wclose

Synopsis

```
#include <stdimap.h>
int lx_wclose(uint wno);        /* wno: window register number */
```

Description This function releases an allocated window register set.

It releases the window register set whose number is wno, allocated by lx_wopen. If this function terminates successfully, it returns 0. If an invalid register number is assigned, an error occurs, and the function returns −1.

This function is also called in the lx_wwait function described below.

3.1.8 lx_wclose_all

Synopsis

```
#include <stdimap.h>
uint lx_wclose_all();
```

Description This function sets the system variable (which controls the 8 window register sets) to the value indicating that the sets are unused.

Before this function is executed, the 8-lower bits of the above system variable (1: window register sets are used, 0: unused) are stored in the 8-lower bits of the value returned by this function.

The following table shows the correspondences between each of the 8-lower bits in the system variable and the window register set status.

Purpose	Read				Write			
Bit position	7	6	5	4	3	2	1	0
	MSB				LSB			
Corresponding window register number	6	4	2	0	7	5	3	1

3.1.9 lx_wget0, lx_wget1, lx_wget2

Synopsis

```
#include < stdimap.h>
ulong lx_wget0(uint wno);      /* wno: window register number */
ulong lx_wget1(uint wno);      /* wno: window register number */
ulong lx_wget2(uint wno);      /* wno: window register number */
```

Description These functions read and then return the value of the defined register in the window register sets (numbered wno0 to wno7): lx_wget0 reads register 0, lx_wget1 reads register 1, and lx_wget2 reads register 2

3.1.10 lx_wopen

Synopsis

```
#include <stdimapo.h>
int lx_wopen(uint rw);          /* If rw = 0: EMEM -> IMEM, If rw = 1: IMEM -> EMEM */
```

Description This function allocates an unused register set and returns its number (which is normally from 0 to 7).

If the parameter is 0 (for a read) or an even number, an even number is returned, and, if the parameter is 1 (for a write) or any other odd number, an odd window register set number is returned.

If there is no unused window register set, the function returns −1.

There are 8 register sets, which are numbered from 0 to 7, and each contains registers used to define settings to transfer blocks of data between IMEM and EMEM.

The window register set number must be specified as a parameter when using line transfer functions (such as lx_wset and lx_wreq). By using this function to obtain the window register set, you can ensure that such sets are not duplicated. For example, window register sets can be obtained by calling this function within the lx_ememrw function.

Window register sets allocated using this function must be released by defining 1 for the close parameter of the lx_wclose or lx_wwait function. Note that, if the lx_wopen function is repeatedly called without releasing the window register sets, there will be insufficient sets and errors will occur.

3.1.11 lx_wopen_all

Synopsis

```
#include <stdimap.h >
void lx_wopen_all(uint status);
```

Description This function defines the 8-lower bits of status (8-lower bits of the system variable that controls all the 8 window register sets).

For example, if the value returned by the lx_wclose_all function is defined for status, the value of the above system variable can be restored to the value from before the lx_wclose_all function was used.

The following table shows the correspondences between each of the 8-lower bits in the system variable and the window register set status.

Purpose	Read				Write			
Bit placement	7	6	5	4	3	2	1	0
	MSB				LSB			
Corresponding window register number	6	4	2	0	7	5	3	1

3.1.12 lx_wset0, lx_wset1, lx_wset2

Synopsis

```
#include < stdimap.h >
void lx_wset0(uint wno, ulong value);     /* wno: window register number */
void lx_wset1(uint wno, ulong value);     /* wno: window register number */
void lx_wset2(uint wno, ulong value);     /* wno: window register number */
```

Description These functions write the value of the defined register in the window register sets (numbered wno0 to wno7): lx_wset0 writes to register 0, lx_wset1 writes to the register 1, and lx_wset2 writes to register 2.

3.1.13 lx_wreq0, lx_wreq1

Synopsis

```
#include <stdimap.h>
void lx_wreq0(uint wno, uint reqnum);
void lx_wreq1(uint wno, uint reqnum);
```

Description These functions issue transfer requests to transfer lines between IMEM and EMEM.

wno specifies a window register set from 0 to 7, and reqnum defines how many lines to transfer (consists of 1 byte per PE).

If a value other than 0 to 7 is assigned to wno, wno&7 is assumed.

Any value from 1 to 4095 can be assigned to reqnum, and, if a value outside this range is specified, reqnum&4095 is assumed.

Issued requests are stored in one of two FIFO queues that have different priorities. If lx_wreq1() is used, transfer requests are stored in the normal priority queue, and, if lx_wreq0() is used, they are stored in the high priority queue.

If there are transfer requests in both queues, the requests stored in the high priority queue are processed first. Requests in a queue are processed in the order they were placed in that queue.

3.1.14 Ix_wwait

Synopsis

```
#include <imapio.h>
int  Ix_wwait(uint wno, uint close);
```

Description This function makes the system wait until line transfers that use the defined window register set wno finish. If the value of the close parameter is not 0, the window register resources are released when the transfer terminates.

If this function terminates successfully, it returns 0. If an invalid register number is defined for the function, or the close parameter is not 0 and the window register resources cannot be released, the function returns −1.

This function is also used by the following three macros defined in imapio.h:

```
#define IxEmemrd_end(wno) Ix_wwait((wno),1)
#define IxEmemwr_end(wno) Ix_wwait((wno),1)
#define IxEmemrw_end(wno) Ix_wwait((wno),0)
```

3.1.15 IxEmemBlk

Synopsis

```
#include <stdimap.h>
int  IxEmemBlk (void(*func)(),
                ulong ein, ulong eou,
                uint lines, uint width,
                uint src_height, uint dst_height,
                ulong *args);
```

Description This function is the basis for the line transfer functions that use division, and all functions IxEmemBlk_Lno, IxEmemBlk_Po, and IxEmemBlk_So are defined as macro functions that execute this function.

The table below shows the relationship between this function and the above macro functions. For details about this function, see the sections that describe each of these macro functions.

Macro function name	How IxEmemBlk is used
IxEmemBlk_Lno	msizex and msizey parameters of IxEmemBlk_Lno are used to define ((msizey&0xff)<<8) + (msizex&0xff) for args[0], and then IxEmemBlk is called
IxEmemBlk_Po	args[0] is set to 0, and then IxEmemBlk is called
IxEmemBlk_So	0 is set for args[0] and dst_height, then IxEmemBlk is called

3.1.16 IxEmemBlk_Lno

Synopsis

```
#include <stdimap.h>
int IxEmemBlk_Lno(void(*func)(),
                  ulong ein, ulong eou,
                  uint lines, uint width,
                  uint src_height, uint dst_height,
                  uint msizex, uint msizey,
                  ulong *args);
```

Description

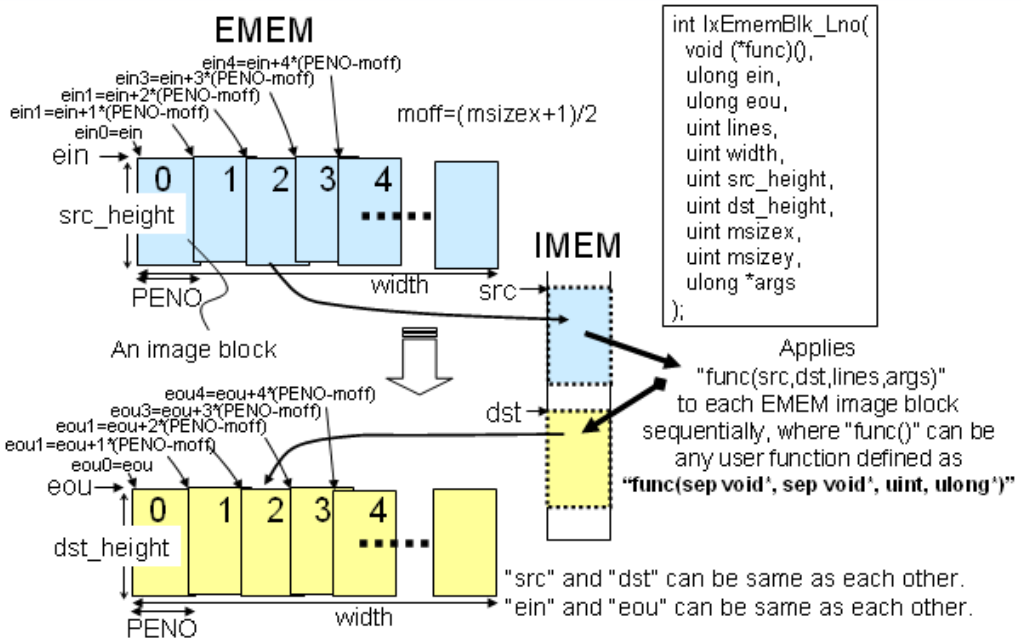
This function writes the PENO × src_height bytes of data read from the absolute EMEM address ein to the IMEM area (src).

Then it calls the func function using the format func(src, dst, lines, args).

The PENO × dst_height bytes of data in the IMEM area (dst returned from func) are written to the absolute EMEM address eou the required number of times (slightly more than (width + PENO – 1) / PENO and proportional to msizex) in order to use filter processing for which the local neighborhood distance is msizex × msizey on an image width–pixels wide (as shown in the figure below).

For src and dst, an amount of memory in the range from max{src_height + barea, dst_height} to 2×(src_height + barea + dst_height) is allocated to the IMEM heap according to the remaining IMEM heap memory. Here, max{*a*, *b*} returns the larger of *a* and *b*, and barea is equal to (msizey + 1) / 2.

When this function terminates, memory allocated to the IMEM heap is released.



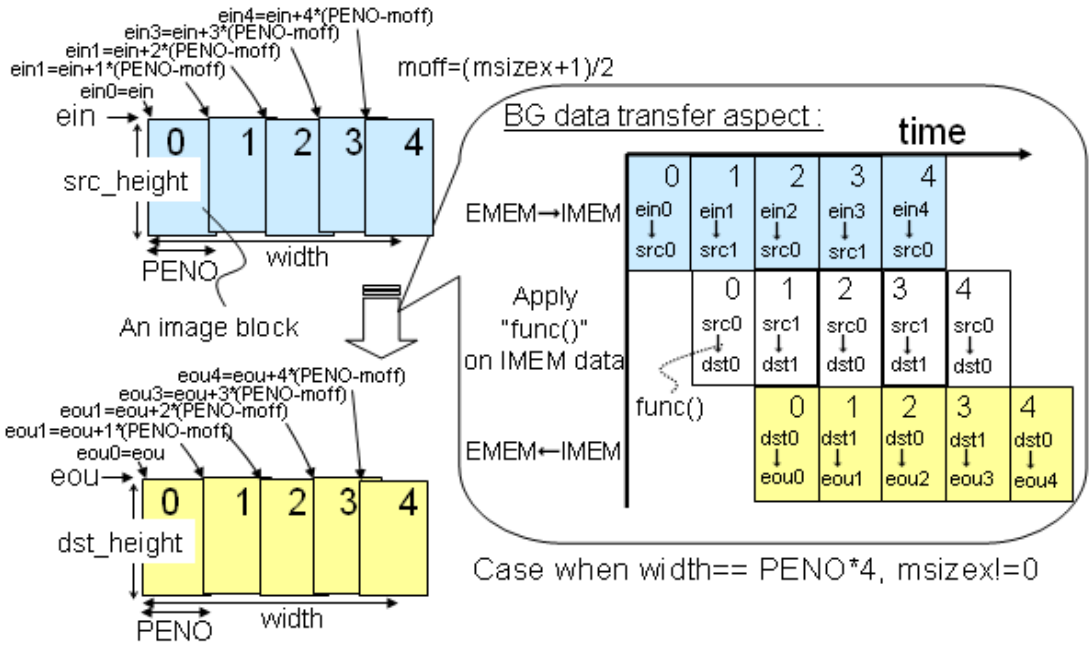
This function determines whether to transfer data between EMEM and IMEM in background (BG) or foreground (FG) according to how much memory is successfully allocated to the IMEM area when the function is executed (as shown in the table below).

Therefore, to perform transfer operations in the background, free up a specific amount of memory in the IMEM heap (for example, by backing up data to the EMEM area) according to the following table before executing this function.

Relationship between ein **and** eou	**Memory successfully allocated from IMEM heap by executing the function**	**Relationship between the** src **and** dst **parameters defined for** func	**Selected line transfer method**
When ein!=eou	2*(src_height + barea + dst_height)	src != dst	BG: ein to src BG: dst to eou
	src_height + barea + dst_height		All transfers are performed in the FG
When ein!=eou	2*Max(src_height + barea, dst_height)	src == dst	BG: ein to src BG: dst to eou
	Max(src_height + barea, dst_height)		All transfers are performed in the FG

The processing performed during a BG transfer is shown in the figure below:

- The image blocks read at the absolute EMEM addresses ein0, ein1, ... are alternately stored in the src0 and src1 buffers allocated to the IMEM heap.
- Then func used the data in those buffers.
- Next, data is alternately read in a similar way from the dst0 and dst1 buffers allocated to the IMEM heap, and then this data is written to the absolute EMEM addresses eou0, eou1, etc…
-



The args parameter passed to the user-specified function func points to an address.

- Its first element, args[0] stores:
  - the left edge mask (= lmask) in the 8 higher bits of its 16 lower bits
  - the right edge mask (= rmask) in the 8 lower bits of its 16 lower bits
  - the processing function counter (= *b*) in its higher 16 bits (can be referenced within the func function)
- the elements args[1] and later can be used to store parameters that the user wants to pass to the func function

To summarize:

- 8 lower bits of args[0]: right edge mask value (= rmask)
- 8 higher bits of the 16 lower bits of args[0]: left edge mask value (= lmask)
- 8 lower bits of the 16 higher bits of args[0]: block number (indicated as 0, 1, 2, 3, or 4 in the above figure)
- 8 higher bits of args[0]: number of blocks (indicated as 5 in the above figure)
- Elements args[1] and later: additional area usable by the user

If the value returned to the IxEmemBlk_Lno function by func is not 0, IxEmemBlk_Lno immediately terminates and the returned value is assimilated to the value returned by the function.

In other cases (when func terminates successfully each time it is called), 0 is returned.

Because the IxEmemBlk_Lno function determines which operation to perform based on the value returned by func, the func function must return 0 when it terminates normally.

3.1.17 IxEmemBlk_So

Synopsis

```
#include <stdimap.h>
int IxEmemBlk_Po(void(*func)(),
        ulong ein, ulong eou,
        uint lines, uint width,
        uint src_height, uint dst_height,
        ulong *args);
```

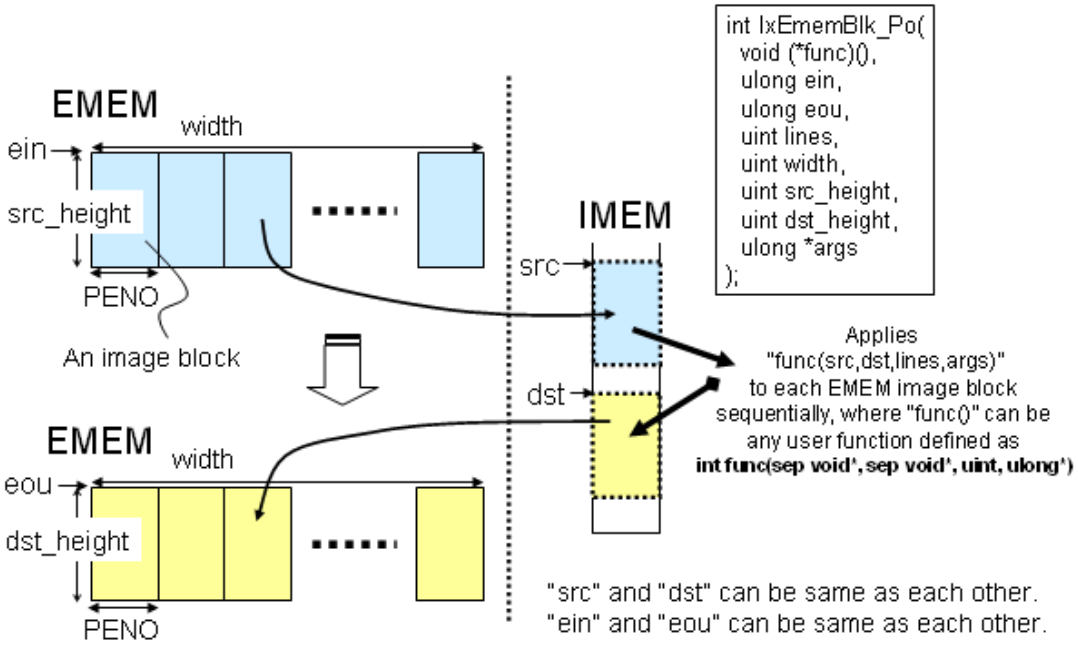
Description This function writes the PENO × src_height bytes of data read from the absolute EMEM address ein to the IMEM area (src).

Then it calls the func function using the format func(src,dst,lines,args).

The PENO × dst_height bytes of data in the IMEM area (dst returned from func) are written to the absolute EMEM address eou, *(width + PENO – 1) /PENO* times (as shown in the figure below).

For src and dst, an amount of memory in the range from Max{src_height + barea, dst_height} to 2×(src_height + barea + dst_height) is allocated to the IMEM heap according to the remaining IMEM heap memory. Here, Max{*a*, *b*} returns the larger of *a* and *b*.

When this function terminates, memory allocated to the IMEM heap is released.



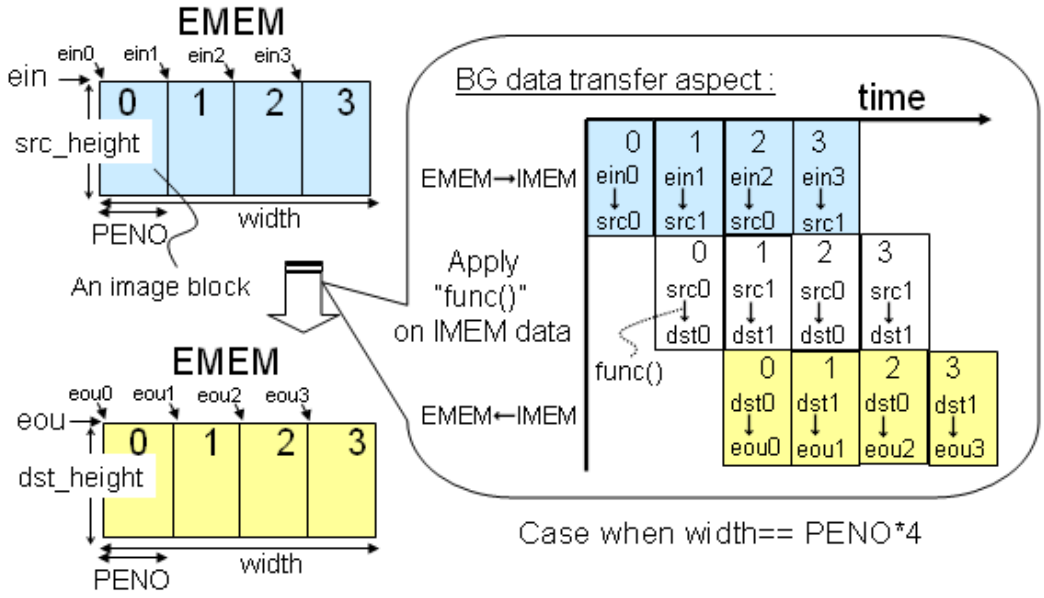
This function determines whether to transfer data between EMEM and IMEM in the background (BG) or foreground (FG) according to how much memory is successfully allocated from the IMEM area when the function is executed (as shown in the table below).

Therefore, to perform transfer operations in the background, free up a specific amount of memory in the IMEM heap (for example, by backing up data to the EMEM area) according to the following table before executing this function.

Relationship between ein **and** eou	**Memory successfully allocated to the IMEM heap by executing the function**	**Relationship between** src **and** dst **parameters defined for** func	**Selected line transfer method**
When ein!=eou	2 x (src_height + dst_height)	src != dst	BG: ein to src BG: dst to eou
	src_height + dst_height		All transfers are performed in FG
When ein==eou	2 x Max{src_height+barea, dst_height}	src == dst	BG: ein to src BG: dst to eou
	Max{src_height+barea, dst_height}		All transfers are performed in FG

The processing performed during a BG transfer is shown in the figure below:

- Image blocks read at the absolute EMEM addresses ein0, ein1, …are alternately stored in the src0 and src1 buffers allocated to the IMEM heap
- Then func is applied to the data in src
- Next, data is alternately read in a similar way from the dst0 and dst1 buffers allocated to the IMEM heap, and then this data is written to the absolute EMEM addresses eou0, eou1, ...



The args parameter passed to the user-specified function func points to an address:

- Its first element, args[0], stores:
  - o the right edge mask value (= rmask) in its 16 lower bits
  - o the processing function counter (= *b*) in its16 higher bits.

Note that the rmask value is PEN0 – (width & (PEN0 – 1) if width is not divided by PEN0 and is 0 otherwise.

- The elements args[1] and later can be used to store parameters that the user wants to pass to the func function.

To summarize:

- 8 lower bits of args[0]: right edge mask value (= rmask)
- 8 higher bits of the 16 lower bits of args[0]: left edge mask value (= lmask)
- 8 lower bits of the 16 higher bits of args[0]: block number (0,1,2 and 3 in the previous figure)
- 8 higher bits of args[0]: number of blocks (4 in the previous figure)
- Elements args[1] and later: additional area usable by the user

If the value returned to the IxEmemBlk_Po function by func is not 0, IxEmemBlk_Po immediately terminates and the returned value is assimilated to the value returned by the function.

In other cases (when func terminates successfully each time it is called), 0 is returned.

Because the IxEmemBlk_Po function determines which operation to perform based on the value returned by func, the func function must return 0 when it terminates normally.

3.1.18 IxEmemBlk_So

Synopsis

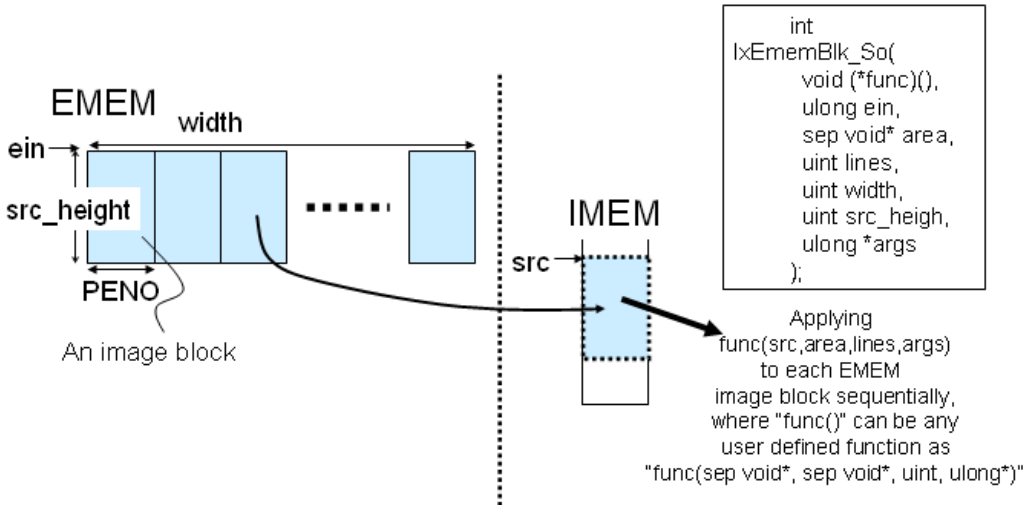
```
#include <stdimap.h>
ulong IxEmemBlk_So(void(*func)(),
        ulong ein, sep void *area,
        uint lines, uint width,
        uint src_height,
        ulong *args);
```

Description This function writes the PENO × src_height bytes of data read from the absolute EMEM address ein to the IMEM area (src).

Then it calls the func function using the format func(src,area,lines,args), *(width + PENO – 1) / PENO* times (as shown in the figure below).

For src, an amount of memory equal to src_height or 2×src_height is allocated to the IMEM heap according to the remaining IMEM heap memory.

When this function terminates, memory allocated to the IMEM heap is released.



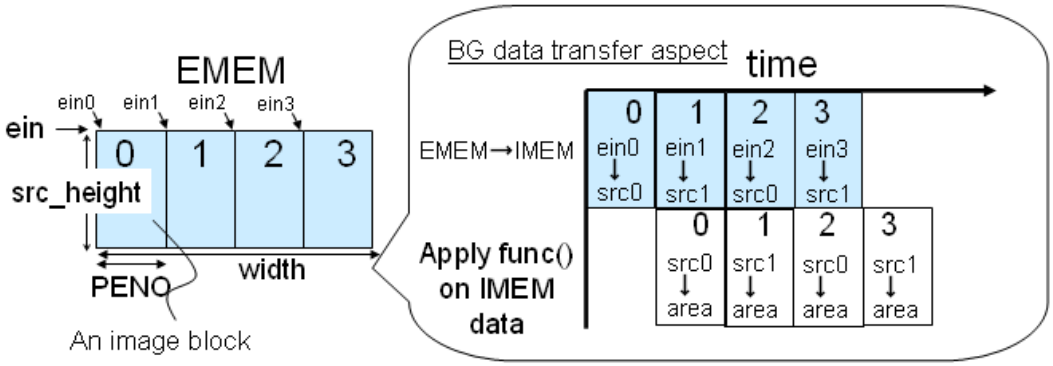
This function determines whether to transfer data between EMEM and IMEM in background (BG) or foreground (FG) in according to how much memory is successfully allocated from the IMEM area when the function is executed (as shown in the table below).

Therefore, to perform transfer operations in background, free up a specific amount of memory in the IMEM heap (for example, by backing up data to the EMEM area) according to the following table before executing this function.

Memory successfully allocated to the IMEM heap by executing the function	Selected line transfer method
2 x src_height	Transfers are performed in BG
src_height	Transfers are performed in FG

The processing performed during a BG transfer is shown in the figure below:

- Image blocks read at the absolute EMEM addresses ein0, ein1, ... are alternately stored in the src0 and src1 buffers allocated to the IMEM heap
- Then, func is applied to the data in src. For the func function, it is assumed that the contents of src are referenced
- Next, the calculated result is stored in area.



The args parameter passed to the user-specified function func points to an address:

- Its first element, args[0], stores:
  - the right edge mask value (= rmask) in its 16 lower bits
  - the processing function counter (= *b*) in its 16 higher bits.

Note that the rmask value is PEN0 – (width & (PEN0 – 1) if width is not divided by PEN0 and is 0 otherwise.

- The elements args[1] and later can be used to store parameters that the user wants to pass to func

To summarize:

8 lower bits of args[0]: right edge mask value (= rmask)

8 higher bits of the 16 lower bits of args[0]: left edge mask value (= lmask)

8 lower bits of the 16 higher bits of args[0]: block number (0, 1, 2, or 3 in the previous figure)

8 higher bits of args[0]: number of blocks (4 in the previous figure)

Elements args[1] and later: additional area usable by the user

If the value returned to the IxEmemBlk_So function by func is not 0, IxEmemBlk_So immediately terminates and the returned value is assimilated to the value returned by the function.

In other cases (when func terminates successfully each time it is called), 0 is returned.

Because the IxEmemBlk_So function determines which operation to perform based on the value returned by func, the func function must return 0 when it terminates normally.

3.1.19 IxEmemrd{_hp}, IxEmemrd{hp}_start, IxEmemrd_end

Synopsis

```
#include <imapio.h>
int IxEmemrd(iadr, eadr, lines);
int IxEmemrd_start(iadr, eadr, lines);
int IxEmemrd_hp(iadr, eadr, lines); //High priority version
int IxEmemrd_hp_start(iadr, eadr, lines); //High priority version

sep void *iadr;      /* internal memory address for destination */
ulong eadr;          /* external memory absolute address for source */
uint lines;          /* number of lines to transfer */

int IxEmemrd_end(uint wno);
```

Description

The IxEmemrd functions transfer lines from EMEM to IMEM.

- iadr defines a pointer to separate-type data, which corresponds to the line address of the starting position in IMEM
- eadr defines the byte address at the starting position of EMEM. The byte address is obtained by multiplying the outside separate type pointer by the number of PEs
- lines defines the number of lines to transfer (up to 4,095).

The return value indicates the window register that was used. If the window register could not be allocated, -1 is returned.

IxEmemrd{_hp} wait until the end of the transfer. The internally used window register set is automatically released.

IxEmemrd{_hp}_start perform the same operation as IxEmemrd{_hp}, except that they return a value immediately after a line transfer request is issued without waiting for the end of the transfer.

By calling IxEmemrd_end() to wait for the end of a transfer, make sure to release the window registers.

IxEmemrd_end is used to make the system wait for the end of the line transfer using the window registers defined by wno.

Assign the value returned by the IxEmemrd{_hp}{_offset}_start functions as the wno parameter.

If the function terminates successfully, IxEmemrd_end returns 0. If an invalid register number is defined, an error occurs and the function returns −1.

The macro definitions for the above functions are in the following header file:

`1DC/cc1dc/lib/include/imap/imapio.h`

Note that the IxEmemrd_end function is defined as follows in imapio.h, so that Ix_wwait can be used directly without problems:

```
#define IxEmemrd_end(wno) Ix_wwait((wno),1)
```

3.1.20 IxEmemrd{_hp}_offset, IxEmemrd{_hp}_offset_start

Synopsis

```
#include <imapio.h>
int IxEmemrd_offset(iadr, eadr, boffset, lines);
int IxEmemrd_offset_start(iadr, eadr, boffset, lines);
int IxEmemrd_hp_offset(iadr, eadr, boffset, lines); // High priority version
int IxEmemrd_hp_offset_start(iadr, eadr, boffset, lines); // High priority version
separate void *iadr;  /* destination internal memory address */
ulong eadr;           /* source external memory absolute address */
uint boffset;         /* block offset value */
uint lines;           /* number of lines to transfer */
```

Description These functions are used in the same way as IxEmemrd{_hp} and IxEmemrd{_hp}_start functions, except that an offset value can be added.

The macro definitions for the above functions are in the following header file:

`1DC/cc1dc/lib/include/imap/imapio.h`

3.1.21 IxEmemrd{_hp}_blk, IxEmemrd{_hp}_blk_start

Synopsis

```
#include <imapio.h>
int IxEmemrd_blk(iadr,eadr,bid,bw,width,lines);
int IxEmemrd_blk_start(iadr,eadr,bid,bw,width,lines);
int IxEmemrd_hp_blk(iadr,eadr,bid,bw,width,lines); //High priority version
int IxEmemrd_hp_blk_start(iadr,eadr,bid,bw,width,lines); //High priority version

separate void *iadr;  /* destination IMAP memory address */
ulong eadr;           /* source external memory absolute address */
uint  bid;   /* block id */
uint  bw;    /* block width */
uint  width;          /* image width */
uint  lines; /* number of lines to transfer */
```

Description These functions are used in the same way as IxEmemrd{_hp} and IxEmemrd{_hp}_start functions, except that bid, bw, and width values can be added. For the meaning of bid and width, see the section about the Ix_ememrw_blk function.

The macro definitions for the above functions are in the following header file:

`1DC/cc1dc/lib/include/imap/imapio.h`

3.1.22 IxEmemrw{_hp}, IxEmemrw{_hp}_start, IxEmemrw_end

Synopsis

```
#include <imapio.h>
void IxEmemrw(iadr,eadr,lines,wno);
void IxEmemrw_start(iadr,eadr,lines,wno);
void IxEmemrw_hp(iadr,eadr,lines,wno); //High priority version
void IxEmemrw_hp_start(iadr,eadr,lines,wno); //High priority version

sep void *iadr;      /* source IMAP memory address */
ulong eadr;          /* destination external memory absolute address */
uint lines;          /* number of lines to transfer */
uint wno;            /* window register set */

int IxEmemrw_end(uint wno);
```

Description The IxEmemrw functions transfer lines between IMEM (the IMAP memory) and EMEM (external memory).

- iadr defines a pointer to separate-type data, which corresponds to the line address at the starting IMEM position.
- eadr defines the byte address at the starting position of EMEM. The byte address is obtained by multiplying the outside separate type pointer by the number of PEs
- lines defines the number of lines to transfer (up to 4,095).

The IxEmemrw{_hp} functions wait until the end of the transfer. However, the window register set is not released automatically, so you need to release it manually when it is no longer necessary.

IxEmemrw_start perform the same operation as IxEmemrw, except that they return a value immediately after a line transfer request is issued without waiting for the end of the transfer.

By calling IxEmemrw_end() to wait for the end of the transfer, make sure to release the window register set.

The IxEmemrw_end function is used to make the system wait for the end of the line transfer using the window registers defined by wno.

Assign wno as the value specified in the IxEmemrw_(offset_)start to be waited for.

The macro definitions for the above functions are in the following header file:

`1DC/cc1dc/lib/include/imap/imapio.h`

Note that the IxEmemrw_end function is defined as follows in imapio.h, so that Ix_wwait can be used directly without any problem:

```
#define IxEmemrw_end(wno) Ix_wwait((wno),0)
```

3.1.23 IxEmemrw{_hp}_offset, IxEmemrw{_hp}_offset_start

Synopsis

```
#include <imapio.h>
void IxEmemrw_offset(iadr, eadr, boffset, lines, wno);
void IxEmemrw_offset_start(iadr, eadr, boffset, lines, wno);
void IxEmemrw_hp_offset(iadr, eadr, boffset, lines, wno); // High priority version
void IxEmemrw_hp_offset_start(iadr,eadr,boffset,lines,wno); // High priority version

separate void *iadr;  /* destination internal memory address */
ulong eadr;           /* source external memory absolute address */
uint boffset;         /* block offset value */
uint lines;           /* number of lines to transfer */
uint wno;             /* window register set */
```

Description These functions are used in the same way as IxEmemrw{_hp} and IxEmemrw{_hp}_start functions, except that an offset value can be added.

The macro definitions for the above functions are in the following header file:
`1DC/cc1dc/lib/include/imap/imapio.h`

3.1.24 IxEmemrw{_hp}_blk, IxEmemrw{_hp}_blk_start

Synopsis

```
#include <imapio.h>
int IxEmemrw_blk(iadr,eadr,bid,bw,width,lines,wno);
int IxEmemrw_blk_start(iadr,eadr,bid,bw,width,lines,wno);
int IxEmemrw_hp_blk(iadr,eadr,bid,bw,width,lines,wno); //High priority version
int IxEmemrw_hp_blk_start(iadr,eadr,bid,bw,width,lines,wno); //High priority version

separate void *iadr;  /* destination IMAP memory address */
ulong eadr;           /* source external memory absolute address */
uint  bid;            /* block id */
uint  bw;             /* block width */
uint  width;          /* image width */
uint  lines;          /* number of lines to transfer */
uint  wno;            /* window register set */
```

Description These functions are used in the same way as IxEmemrw{_hp} and IxEmemrw{_hp}_start functions, except that bid, bw, and width values can be added.

For the meaning of bid and width, see the section about the Ix_ememrw_blk function.

3.1.25 IxEmemwr{_hp}, IxEmemwr{_hp}_start, IxEmemwr_end

Synopsis

```
#include <imapio.h>
int IxEmemwr(iadr,eadr,lines);
int IxEmemwr_start(iadr,eadr,lines);
int IxEmemwr_hp(iadr,eadr,lines);  //High priority version
int IxEmemwr_hp_start(iadr,eadr,lines); //High priority version

sep void *iadr;         /* source IMAP memory address */
ulong eadr;             /* destination external memory absolute address */
uint lines;  /* number of lines to transfer */

int IxEmemwr_end(uint wno);
```

Description The IxEmemwr functions transfer lines from IMEM to EMEM.

- iadr defines a pointer to separate-type data, which corresponds to the line address at the starting position of IMEM.
- eadr defines the byte address of the starting position of EMEM. The byte address is obtained by multiplying the outside separate type pointer by the number of PEs
- lines defines the number of lines to transfer (up to 4,095).

The return value indicates the window register that was used. If the window register could not be allocated, -1 is returned.

The IxEmemwr{_hp} functions wait until the end of the transfer. The internally used window register set is automatically released.

IxEmemwr_start functions perform the same operation as IxEmemwr functions, except that they return a value immediately after a line transfer request is issued without waiting for the end of the transfer.

By calling IxEmemwr_end() to wait for the end of a transfer, make sure to release the window register set.

The IxEmemwr_end function is used to make the system wait until the end of the line transfer using the window registers defined by wno.

Assign the value returned by the IxEmemwr_(offset_)start functions for the wno parameter.

If the function terminates successfully, IxEmemwr_end returns 0. If an invalid register number is assigned, an error occurs and the function returns −1.

The macro definitions for the above functions are in the following header file:

`1DC/cc1dc/lib/include/imap/imapio.h`

Note that the IxEmemwr_end function is defined as follows in imapio.h, so that Ix_wwait can be used directly without problems:

```
#define IxEmemwr_end(wno) Ix_wwait((wno),1)
```

3.1.26 IxEmemwr{_hp}_offset, IxEmemwr{_hp}_offset_start

Synopsis

```
#include <imapio.h>
void IxEmemwr_offset(iadr, eadr, boffset, lines);
void IxEmemwr_offset_start(iadr, eadr, boffset, lines);
void IxEmemwr_hp_offset(iadr, eadr, boffset, lines); // High priority version
void IxEmemwr_hp_offset_start(iadr, eadr, boffset, lines); // High priority version

separate void *iadr;  /* destination IMAP memory address */
ulong eadr;           /* source external memory absolute address */
uint boffset;         /* block offset value */
uint lines;           /* number of lines to transfer */
```

Description These functions are used in the same way as IxEmemwr{_hp} and IxEmemwr{_hp}_start functions, except that an offset value can be added.

This function is included among the macro definitions in the following header file:

`1DC/cc1dc/lib/include/imap/imapio.h`

3.1.27 IxEmemwr{_hp}_blk, IxEmemwr{_hp}_bl

Synopsis

```
#include <imapio.h>
int IxEmemwr_blk(iadr, eadr, bid, bw, width, lines);
int IxEmemwr_blk_start(iadr, eadr, bid, bw, width, lines);
int IxEmemwr_hp_blk(iadr, eadr, bid, bw, width, lines); // High priority version
int IxEmemwr_hp_blk_start(iadr, eadr, bid, bw, width, lines); // High priority version

separate void *iadr;  /* destination IMAP memory address */
ulong eadr;           /* source external memory absolute address */
uint  bid;            /* block id */
uint  bw;             /* block width */
uint  width;          /* image width */
uint  lines;          /* number of lines to transfer */
```

Description These functions are used in the same way as IxEmemwr{_hp} and IxEmemwr{_hp}_start functions, except that bid, bw, and width values can be added.

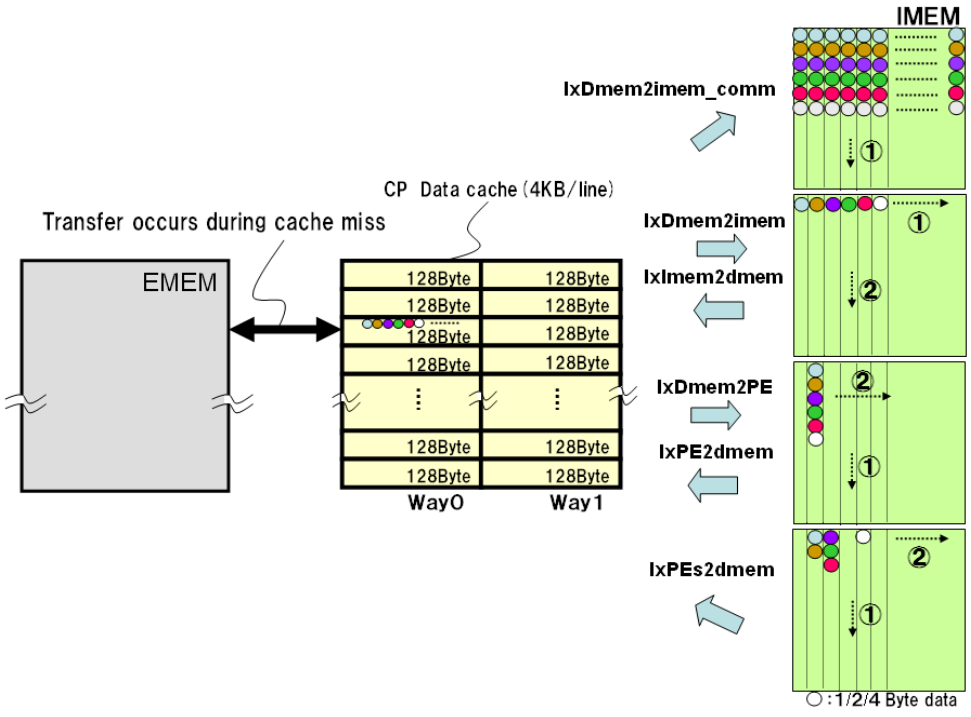
For the meaning of bid and width, see the section about the Ix_ememrw_blk function.

3.2 Functions to transfer data between DMEM and IMEM

These functions are used to transfer data in the CP data cache to IMEM or inversely (see figure below)

All these functions are coded in assembly language and listed in the following table:

Function name	Description
Between CP data cache (D$) and PEs in the direction of PEs (see figure below)	
IxDmem2imem_comm	D$ to IMEM in 1-, 2-, or 4-byte units
IxDmem2imem{_u2,_u4}	D$ to IMEM in 1-, 2-, or 4-byte units
IxImem2dmem{_u2,_u4}	IMEM to D$ in 1-, 2-, or 4-byte units
Between D$ and each PE in the direction of IMEM addressing (see figure below)	
IxDmem2PE{_u2}	D$ to IMEM in 1- or 2-byte units. If more bytes are specified, they are transferred as series of 2-byte units
IxPE2dmem{_u2}	IMEM to D$ in 1- or 2-byte units. If more bytes are specified, they are transferred as series of 2-byte units
IxPEs2dmem{_u2,_u4}	IMEM to D$ in 1-, 2- or 4-byte units. The number of elements that differ for each PE can be defined



3.2.1 IxDmem2imem_comm

Synopsis

```
#include <stdimap.h>
void IxDmem2imem_comm(sep void * sepdata, void *dmem, uint bytes);
```

Description For all PEs, this function stores the same bytes bytes of data that are in the dmem array into the IMEM area starting from the sep array sepdata.

For example, this function can be used to store tables saved in DMEM into the IMEM area for each PE. Then, perform table lookup processing for sepdata that accesses a different address for each PE.

The process is equivalent to the following 1DC code:

```
for (int i=0; i< lines; i++) (sep uchar*) sepdata [i] = *(uchar*)dmem++;
```

3.2.2 IxDmem2imem{_u2, _u4}

Synopsis

```
#include <stdimap.h>
uint IxDmem2imem(separate void *sepdata,void * dmem,uint loff,uint lines);
uint IxDmem2imem_u2(separate uint * sepdata,uint *dmem,uint loff,uint lines);
uint IxDmem2imem_u4(separate ulong * sepdata,ulong *dmem,uint loff,uint lines);
```

Description Starting at the beginning of dmem, these functions read the "lines × BYTES × number of PEs" bytes necessary to fill up the sep array sepdata.

Then the elements are stored in the array one-by-one in the direction PEs are arranged, starting at the beginning of sepdata.

The value obtained by adding "(PEN0 + loff) × BYTES" to the current starting address is the starting address used to store the next line.

The return value is the number of stored lines (lines), and each line contains "BYTES × number of PEs" bytes of data.

The value of BYTES is 1 for the IxDmem2imem function, 2 for the IxDmem2imem_u2 function, and 4 for the IxDmem2imem_u4 function.

Note that, when transferring the same number of bytes, the IxDmem2imem_u2 and IxDmem2imem_u4 functions are faster than the IxDmem2imem function.

The process of these functions (especially IxDmem2imem) is equivalent to the following 1DC code:

```
for (int i=0;i<lines;i++)
   for (int j=0; j<PENO; j++) (sep uchar*) sepdata[i]:[j:]=*(uchar*) dmem++;
return lines;
```

3.2.3 IxDmem2PE{_u2}

Synopsis

```
#include <stdimap.h>
uint IxDmem2PE(separate void*sepdata,void *dmem,uint thisPE,uint lines);
uint IxDmem2PE_u2(separate uint *sepdata,uint *dmem,uint thisPE,uint lines);
```

Description

If thisPE is not PENO, which is the maximum physical PE number, these functions read "lines × BYTES" bytes starting at the beginning of dmem.

Then these bytes are used to fill the sep array sepdata for thisPE, starting at the beginning of sepdata.

These functions return the value lines.

If thisPE is equal to PENO, these functions read "lines × BYTES × number of PEs" bytes starting at the beginning of dmem. These bytes are used to write "lines × BYTES" bytes to each PE. Then proceed to the beginning of sepdata for the next PE, starting at the beginning of sepdata for the PE whose PE number is 0.

These functions return the value "PENO × lines".

The value of BYTES is 1 for the IxDmem2PE function and 2 for the IxDmem2PE_u2 function.

When transferring the same number of bytes, the IxDmem2PE_u2 function is faster than the IxDmem2PE function.

If BYTES is 4 or more, specify an equivalent multiple for the lines value of the IxDmem2PE_u2 function when transferring.

The process of these functions (especially IxDmem2PE_u2) is equivalent to the following 1DC code:

```
int i
if (thisPE<PENO) {
   for (i=0; i<words; i++) (sep uint*) sepdata[i]:[thisPE:]=*(uint*) dmem++;
   return words;
} else {
   for (thisPE=0; thisPE<PENO; thisPE++)
      for (i=0; i<words; i++) (sep uint*)sepdata[i]:[thisPE:]=*(uint*) dmem++;
   return PENO*words;
}
return 0;
```

3.2.4 IxImem2dmem{_u2, _u4}

Synopsis

```
#include <stdimap.h>
uint IxImem2dmem (sep void *src, void * dmem, sep uchar *msk, uint loff, uint lines);
uint IxImem2dmem_u2(sep uint *src, uint* dmem, sep uchar *msk, uint loff, uint lines);
uint IxImem2dmem_u4(sep ulong *src, ulong* dmem, sep uchar *msk, uint loff, uint lines);
```

Description

These functions read lines lines of data (total of "lines × BYTES × number of unmasked PEs" bytes), starting at the beginning of the sep array src for unmasked PEs (defined below).

Then, data is stored in dmem.

However, if off is not 0, the functions increment the address by "off × BYTES" when the "lines × BYTES" bytes of data are stored, and then store the next "lines × BYTES" of data.

These functions return the number of words stored in dmem (where one word is BYTES bytes).

The value of BYTES is 1 for the IxImem2dmem function, 2 for the IxImem2dmem_u2 function, and 4 for the IxImem2dmem_u4 function.

When transferring the same number of bytes, the IxImem2dmem_u2 and IxImem2dmem_u4 functions are faster than the IxImem2dmem function.

Unmasked PEs are defined using the mif/mifa function if msk is 0 (NULL). If the msk element of a PE is 1, that PE is unmasked, and, if the element is 0, that PE is masked and out of range.

The following shows an example of using the mif/mifa function to define unmasked PEs:

```
mifa(5<_PENUM && _PENUM<PENO-5) // Masks the leftmost 5 PEs and the rightmost 5 PEs
IxImem2dmem_u2(sepdata, dmemdata, (sep uchar*)NULL, 0, 120);
```

Note that the PE mask definitions are effective even across functions. Therefore, to use the IxImem2dmem{_u2,_u4} functions when all PEs are always unmasked, the following sort of code is required:

```
mifa(1)  IxImem2dmem_u2(sepdata, dmemdata, (sep uchar*)NULL, 0, 120);
```

The process of these functions (especially IxImem2dmem) are equivalent to the following 1DC code:

```
 int i,j;
sep int k;
if (msk == NULL) {
        k:=0;
        k =1;
        for (i=0; i<lines; i++) {
                for (j=0; j<PENO; j++)
                        if (k:[j:]) *(uchar*) dmem++ = (sep uchar*) src[i]:[j:];
                dmem += off;
        }
} else {
        for (i=0; i<lines; i++){
                for (j=0; j<PENO; j++)
                        if (msk[i]:[j:]) *(uchar*) dmem++ = (sep uchar*) src[i]:[j:];
                dmem += off;
        }
}
```

3.2.5 IxPE2dmem{_u2}

Synopsis

```
#include <stdimap.h>
uint IxPE2dmem(sep void *src,void *dmem,uint thisPE,uint lines);
uint IxPE2dmem_u2(sep uint * src,uint *dmem,uint thisPE,uint lines);
```

Description

If thisPE is not PENO, which is the maximum physical PE number, these functions read "lines × BYTES" bytes starting at the beginning of the sep array src for thisPE&PENO-1.

These bytes are used to fill dmem, starting at the beginning.

These functions return the value lines.

If thisPE is equal to PENO, these functions read "lines × BYTES" bytes starting at the beginning of the sep array src for each PE. Then it proceeds to the beginning of src for the next PE and read the "lines × BYTES × PENO" bytes

Then data are written to dmem.

These functions return the value "PENO × lines".

The value of BYTES is 1 for the IxPE2dmem function and 2 for the IxPE2dmem_u2 function.

When transferring the same number of bytes, the IxPE2dmem_u2 function is faster than the IxPE2dmem function.

If BYTES is 4 or more, define an equivalent multiple for the lines value of the IxPE2dmem_u2 function when transferring.

The process of these functions (specifically the IxPE2dmem_u2 function) is equivalent to the following 1DC code:

```
int i
if (thisPE != PENO) {
   for (i=0; i<words; i++) *(uint*) dmem++ = (sep uint*) src[i]:[thisPE &(PENO-1):];
   return words;
} else {
   for (thisPE=0; thisPE<PENO; thisPE++)
      for (i=0; i<words; i++) *(uint*) dmem++ = (sep uint*) src[i]:[thisPE:];
      return PENO*words;
}
```

3.2.6 IxPEs2dmem{_u2, _u4}

Synopsis

```
#include <stdimap.h>
uint IxPEs2dmem(sep void *src, void * dmem, sep uint lines);
uint IxPEs2dmem_u2(sep uint *src, uint * dmem, sep uint lines);
uint IxPEs2dmem_u4(sep ulong*src, ulong* dmem, sep uint lines);
```

Description

These functions store lines words from each PE of the sep array src into dmem (where 1 word is BYTES bytes).

A different value can be defined for lines for each PE.

The return value is the number of words stored in dmem.

The value of BYTES is 1 for the IxPEs2dmem function, 2 for the IxPEs2dmem_u2 function, and 4 for the IxPEs2dmem_u4 function.

When transferring the same number of bytes, the IxPEs2dmem_{u2,u4} functions are faster than the IxPEs2dmem function.

The process of these functions (especially IxPEs2dmem_u2) is equivalent to the following 1DC code:

```
 int x,i,num,k;
 for (i=0, x=0; x<PENO; x++){
     k=0;
    num = lines:[x:];
    while(k != num)
        (uint*) dmem [i++] = (sep uint*) src[k++]:[x:];
}
return i;
```

3.3 Functions to transfer data between DMEM and shared RAM

The CP contains 2 KB of scratchpad RAM (commonly known as shared RAM). The functions below can quickly transfer data between shared RAM and DMEM.

Although the shared RAM can be accessed for data read by using the lx_in function or for data write by using the lx_ou function, using the functions below can save time when accessing a lot of data.

Note that the shared RAM is mapped to the I/O part of the address map, so you have to manually define the addresses you want to access following the table below.

This table lists the main I/O address assignments:

Address	Write permission	Resource
0x8000 to 0x87FC	Can be read or written	Shared RAM (total of 2KB
0xC010	Read only	pend mirror (when read: value of the pend system register)
0xC014	Read only	dend mirror (when read: value of the dend system register)
0xC018	Read only	estklm mirror (when read: value of estklm system register)
0XC01C	Read only	istklm mirror (when read: value of the istklm system register)

3.3.1 lxDmem2shared_u4{_nc}

Synopsis

```
#include <stdimap.h>

void lxDmem2shared_u4(ulong dst, ulong *src, uint words);

void lxDmem2shared_nc_u4(ulong dst, outside ulong *src, uint words);
```

Description The lxDmem2shared_u4 function transfers words words from the area starting at src to the shared RAM starting at dst. 1 word is 4 bytes. dst can only use data multiples of 4.

The lxDmem2shared_nc_u4 function transfers words words from the uncached area (of the outside type) starting at src to the shared RAM starting at dst. 1 word is 4 bytes. dst can only use data multiples of 4.

3.3.2 lxShared2dmem_u4{_nc}

Synopsis

```
#include <stdimap.h>

void lxShared2dmem_u4(ulong src,ulong *dst,uint words);

void lxShared2dmem_u4(ulong src,outside ulong *dst,uint words);
```

Description The lxDmem2shared_u4 function transfers words words from the shared RAM starting at src to the area starting at dst. 1 word is 4 bytes. src can only use data multiples of 4.

The lxShared2dmem_nc_u4 function transfers words words from the shared RAM starting at src to the uncached area (of the outside type) starting at dst. 1 word is 4 bytes. src can only use data multiples of 4.

3.4 Functions to transfer data between memory area of the same type (simple copy and initialization)

3.4.1 ix_memcpy{_nc, _nc2c, _c2nc, 16, 16_nc, 16_nc2c, 16_c2nc}

Synopsis

```
#include <stdimap.h>
void ix_memcpy(void *dst_p, void *src_p, uint copy_cnt);
void ix_memcpy_nc(outside void *dst_p, outside void *src_p, uint copy_cnt);
void ix_memcpy_nc2c(void *dst_p, outside void *src_p, uint copy_cnt);
void ix_memcpy_c2nc(outside void *dst_p, void *src_p, uint copy_cnt);
void ix_memcpy16(ulong *dst_p, ulong *src_p, uint copy_cnt);
void ix_memcpy16_nc(outside ulong *dst_p, outside ulong *src_p, uint copy_cnt);
void ix_memcpy16_nc2c(ulong *dst_p, outside ulong *src_p, uint copy_cnt);
void ix_memcpy16_c2nc(outside ulong *dst_p, ulong *src_p, uint copy_cnt);
```

Description

The ix_memcpy function copies copy_cnt bytes from the area starting at src_p to the area starting at dst_p.

ix_memcpy_nc is the same as ix_memcpy, except that it copies data from an uncached area (of the outside type) to another uncached area (of the outside type).

ix_memcpy_nc2c function is the same as ix_memcpy, except that it copies data from an uncached area (of the outside type) to a cached area.

ix_memcpy_c2nc is the same as ix_memcpy, except that it copies data from a cached area to an uncached area (of the outside type).

ix_memcpy16 copies copy_cnt bytes from the area starting at src_p to the area starting at dst_p in 16-byte units.

Both dst_p and src_p have to be multiples of 4, and copy_cnt must be a multiple of 16. ix_memcpy16() is faster than ix_memcpy().

ix_memcpy16_nc is the same as ix_memcpy16, except that it copies data from an uncached area (of the outside type) to an uncached area (of the outside type).

ix_memcpy16_nc2c is the same as ix_memcpy16, except that it copies data from an uncached area (of the outside type) to a cached area.

ix_memcpy16_c2nc is the same as ix_memcpy16, except that it copies data from a cached area to an uncached area (of the outside type).

3.4.2 ix_memset{_u2, _nc, _nc_u2, _u4, _nc_u4}

Synopsis

```
#include <stdimap.h>
void ix_memset(void *area_p, uint fill_ch, uint set_cnt);
void ix_memset_nc(outside void *area_p, uint fill_ch, uint set_cnt);
void ix_memset_u2(uint *area_p, uint fill_ch, uint set_cnt);
void ix_memset_nc_u2(outside uint *area_p, uint fill_ch, uint set_cnt);
void ix_memset_u4(ulong *area_p, ulong fill_ch, uint set_cnt);
void ix_memset_nc_u4(outside ulong *area_p, ulong fill_ch, uint set_cnt);
```

Description ix_memset writes set_cnt bytes of the lower byte of fill_ch to the area starting at area_p.

ix_memset_nc is the same as ix_memset, except that it writes to an uncached area (of the outside type).

ix_memset_u2 writes set_cnt words (where 1 word is 2 bytes) of fill_ch to the area starting at area_p.

ix_memset_nc_u2 is the same as ix_memset_u2, except that it writes to an uncached area (of the outside type).

ix_memset_u4 writes set_cnt words (where 1 word is 4 bytes) of fill_ch to the area starting at area_p.

ix_memset_nc_u4 is the same as ix_memset_u4, except that it writes to an uncached area (of the outside type).

3.4.3 lx_memcpy{16}

Synopsis

```
#include <stdimap.h>
void lx_memcpy(sep void *dst_p, sep void *src_p, uint lines);
void lx_memcpy16(sep ulong*dst_p, sep ulong *src_p, uint lines);
```

Description lx_memcpy function copies lines bytes of data starting at the src_p area to the area starting at dst_p.

The lx_memcpy16 function copies lines bytes of data starting at the src_p area to the area starting at dst_p in 16-byte units.

Both dst_p and src_p have to be multiples of 4, lines has to be a multiple of 16.

The lx_memcpy16 function is faster than the lx_memcpy function.

3.4.4 Ix_memset{_u2, _u4}

Synopsis

```
#include <stdimap.h>
void Ix_memset(sep void *area_p, uint fill_ch, uint lines);
void Ix_memset_u2(sep uint *area_p, uint fill, uint lines);
void Ix_memset_u4(sep ulong *area_p, ulong fill, uint lines);
```

Description The Ix_memset function writes lines bytes of the lower byte of fill_ch to the area starting at area_p.

The Ix_memset_u2 function writes lines words (where 1 word is 2 bytes) of the 2-byte value defined by fill to the area starting at area_p.

The Ix_memset_u4 function writes lines words (where 1 word is 4 bytes) of the 4-byte value defined by fill to the area starting at area_p.

3.4.5 Ix_memset_sep{_u2, _u4}

Synopsis

```
#include <stdimap.h>
void Ix_memset_sep(sep void *area_p, sep uint fill_ch, uint lines);
void Ix_memset_sep_u2(sep uint *area_p, sep uint fill, uint lines);
void Ix_memset_sep_u4(sep ulong *area_p, sep ulong fill, uint lines);
```

Description The Ix_memset_sep function writes lines bytes of the lower byte of fill_ch to the area starting at area_p.

The Ix_memset_sep_u2 function writes lines words (where 1 word is 2 bytes) of the 2-byte value defined by fill to the area starting at area_p.

The Ix_memset_sep_u4 function writes lines words (where 1 word is 4 bytes) of the 4-byte value defined by fill to the area starting at area_p.

3.4.6 IxEmem2emem

Synopsis

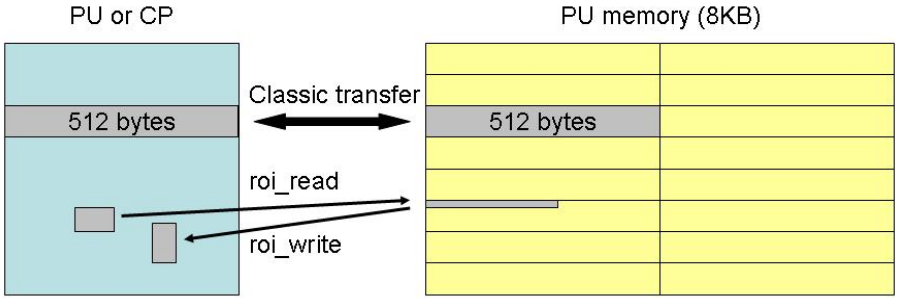
```
#include <stdimap.h>
void IxEmem2emem(
outside sep void *addr0,      /* source external memory line address */
outside sep void *addr1,      /* destination external memory line address */
uint lines                    /* number of lines to transfer */
)
```

Description IxEmem2emem() transfers data between EMEM memory areas. Any number of lines can be transferred.

3.5 Functions to transfer data within DMEM (ROI transfers)

A ROI transfer is used to transfer a user-specified rectangular area of data, from external memory to a data cache in a CP or PU.

Using the ix_roi_read and ix_roi_write functions reduce the transfer time when reading or writing a small amount of data, as well as for a specific rectangular area of data, because each line in a PU data cache is large (512 bytes).



3.5.1 ix_roi_read

Synopsis

```
#include <stdimap.h>
void ix_roi_read(void* RectAddr, void* ContAddr, uint width, uint offset, uint lines);
```

Description This function transfers a rectangular area of data starting at RectAddr, to the continuous DMEM area starting at ContAddr.

- width defines the rectangle width
- offset defines the address increment from the end of the current line to the beginning of the next line
- lines defines the number of lines

Except for lines, the units for all parameters are bytes.

3.5.2 ix_roi_write

Synopsis

```
#include <stdimap.h>
void ix_roi_write(void* RectAddr, void* ContAddr, uint width, uint offset, uint lines);
```

Description This function transfers data from the continuous DMEM area starting at ContAddr to a rectangular area starting at RectAddr.

- width defines the rectangle width
- offset defines the address increment from the end of the current line to the beginning of the next line
- lines defines the number of lines

Except for lines, the units for all parameters are bytes.

3.6 Data cache manipulation functions

3.6.1 ix_civd_all

Synopsis #include <stdimap.h>

void ix_civd_all()

Description This function invalidates all valid cache lines.

3.6.2 ix_civda

Synopsis #include <stdimap.h>

void ix_civda(void *addr)

Description This function invalidates cache lines hit at addr.

3.6.3 ix_civda2

Synopsis #include <stdimap.h>

void ix_civda2 (void *area_p, uint size);

Description This function disables cache lines in the range from area_p to area_p + size – 1. However, size must be 128 or less.

3.6.4 ix_cwb_all

Synopsis #include <stdimap.h>

void ix_cwb_all()

Description This function writes back all dirty cache lines.

3.6.5 ix_cwba

Synopsis #include <stdimap.h>

void ix_cwba(void *addr)

Description This function writes back dirty cache lines that are hit at the address addr.

3.6.6 ix_cwba2

Synopsis #include <stdimap.h>

void ix_cwba2(void *area_p, uint size)

Description This function writes back cache lines in the range from area_p to area_p + size – 1. However, size must be 128 or less.

3.6.7 ix_cwbivda

Synopsis #include <stdimap.h>

void ix_cwbivda(void *addr)

Description This function writes back and then invalidates cache lines hit at addr.

3.7 Dynamic Memory Allocation

3.7.1 ix_malloc, ix_calloc, ix_free, ix_realloc, ix_heap_overflow

Synopsis

```
#include <stdimap.h>
void  *ix_malloc(ulong size);
void  *ix_calloc(ulong num,ulong size);
void  ix_free(void* data_p);
void  *ix_realloc(void* data_p,ulong new_size);
void  ix_heap_overflow();
```

Description

ix_malloc returns the starting address of a size-byte memory block aligned at a 1,024 byte boundary.

The 16 bytes immediately preceding the returned address are used as memory block header.

ix_calloc returns the starting address of a num × size -byte memory block aligned at a 1,024 byte boundary.

The 16 bytes immediately preceding the returned address are used as memory block header.

When using these functions from a PU, memory is allocated from the local heap for that PU.

However, because these functions must store the starting address of an empty memory block chain in a static variable, to use these functions from a thread in a PU, the threads must be started after allocating memory for static variables using the Pthreads function library or Ix_cp_start_pu and Ix_cp_start_multi_pu.

If only Ix_forkinit and Ix_fork are used to start a thread in a PU, memory is not allocated for static variables and these functions cannot be used.

Also note that these functions use the value stored in the special register dstklm, so that this register value and its related values and must not be changed.

ix_malloc and ix_calloc use an empty memory chain to manage released memory.

This chain is initially empty, and, when memory is allocated, a memory block of the required size is removed from the chain and returned, if such a block exists. If such a block does not exist, the memory is taken from the heap.

When the ix_free function is used to release memory, that memory block is released for the heap if it is adjacent to the heap, or added to the empty memory block chain (in ascending order of memory size) if not.

If the block was released for the heap, any memory in the chain that can be released for the heap is also released for the heap.

ix_realloc changes the size of a memory block previously allocated using ix_malloc or ix_calloc.

ix_malloc, ix_calloc, ix_free, and ix_realloc cannot be re-executed at the same time.

Therefore, if an interrupt routine might call these functions, enclose these functions in ix_idis() and ix_iena() on the interrupted side to prevent the issue of interrupts when calling them.

ix_heap_overflow() simply calls ix_halt(1). When this function is called, the program stops.

By calling ix_heap_overflow whenever functions such as ix_malloc return NULL, the user can determine when an invalid program operation is performed due to insufficient memory in the DMEM heap, just by checking which function the program stopped within:

```
if ( !(ip = ix_malloc(...) ) Ix_heap_overflow();
```

3.7.2 Ix_eheap, Ix_eheap_available, Ix_eheap_overflow

Synopsis

```
#include <stdimap.h>
outside sep void* Ix_eheap(uint lines);
ulong    Ix_eheap_available();
void     Ix_eheap_overflow();
```

Description

Ix_eheap() extracts lines bytes from the EMEM heap for each PE, and then returns the corresponding transfer address.

If lines is not a multiple of 4, it uses the nearest multiple of 4 that is greater than lines.

If memory cannot be allocated, the return value is 0 (= NULL).

Ix_eheap() only reduces the EMEM heap pointer by the defined number of lines.

After you finish using the allocated EMEM heap area, be sure to use Ix_eheap_return(), which is described below, to release the area.

Ix_eheap_available() returns the amount of usable memory in the EMEM heap when called.

Ix_eheap_overflow() simply calls ix_halt(1). When this function is called, the program stops.

By calling ix_eheap_overflow whenever Ix_eheap() returns 0, the user can determine when an invalid program operation is performed due to insufficient memory in the EMEM heap, just by checking which function the program stopped within:

```
if(!(ep=Ix_eheap(...)) Ix_eheap_overflow();
```

3.7.3 Ix_eheap_return

Synopsis

```
#include <stdimap.h>
ulong Ix_eheap_return(uint lines);
```

Description This function releases the area allocated using Ix_eheap() for the EMEM heap.

Similarly to how Ix_eheap() only reduces the EMEM heap pointer by the defined number of lines, Ix_eheap_return() only increases the EMEM heap pointer by the defined number of lines.

However, if lines is not a multiple of 4, it uses the nearest multiple of 4 that is greater than lines.

Therefore, in the same way as performing push and pop operations for a stack, Ix_eheap_return must be called using the same number of lines defined for Ix_eheap().

Ix_eheap_return() returns the current EMEM heap pointer value.

3.7.4 Ix_iheap, Ix_iheap_available, Ix_iheap_overflow

Synopsis

```
#include <stdimap.h>
sep void* Ix_iheap(uint lines);
uint Ix_iheap_available();
void Ix_iheap_overflow();
```

Description Ix_iheap() extracts lines bytes from the IMEM heap for each PE, and then returns the corresponding transfer address.

If lines is not a multiple of 4, it uses the nearest multiple of 4 that is greater than lines.

If memory cannot be allocated, the return value is 0 (= NULL).

Ix_iheap() only reduces the IMEM heap pointer by the defined number of lines.

After you finish using the allocated IMEM heap area, be sure to use Ix_iheap_return(), which is described below, to release the area.

Ix_iheap_available() returns the amount usable memory in the IMEM heap when called.

Ix_iheap_overflow() simply calls ix_halt(1). When this function is called, the program stops.

By calling the ix_iheap_overflow function whenever Ix_iheap() returns 0, the user can determine when an invalid program operation is performed due to insufficient memory in the IMEM heap, just by checking which function the program stopped within:

```
if ( !(ip = Ix_iheap(...) ) Ix_iheap_overflow();
```

3.7.5 Ix_iheap_return

Synopsis #include <stdimap.h>

uint Ix_iheap_return(uint lines);

Description This function returns the area allocated by Ix_iheap() to the IMEM heap.

Similarly to how Ix_iheap() only reduces the IMEM heap pointer by the defined number of lines, Ix_iheap_return() only increases the IMEM heap pointer by the defined number of lines.

However, if lines is not a multiple of 4, it uses the nearest multiple of 4 that is greater than lines.

Therefore, in the same way as performing push and pop operations for a stack, Ix_iheap_return must be called using the same number of lines defined for Ix_iheap().

Ix_iheap_return() returns the current IMEM heap pointer value.

3.8 Sep Data Manipulation Functions

3.8.1 IxAddsep, IxAddsepl, IxAddsepll

Synopsis

```
#include <stdimap.h>
long IxAddsep(separate int a)
long IxAddsepl(separate long a)
long long IxAddsepll(separate long a)
```

Description These functions return the sum of all elements of a in active PEs.

3.8.2 IxCountsep

Synopsis

```
#include <stdimap.h>
int IxCountsep(separate uchar a);
```

Description This function counts PEs for which the value of a is not 0, and returns the result.

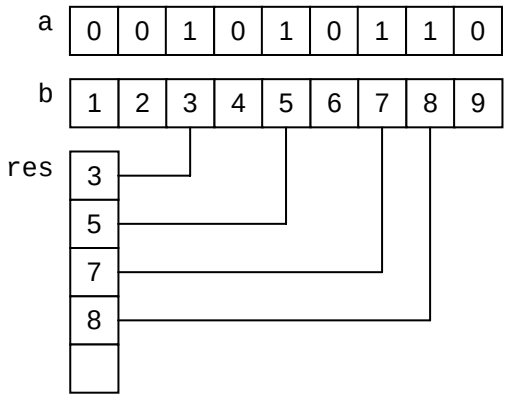
If a condition is defined for a, for example, PEs that satisfy that condition can be counted.

3.8.3 IxGathersep

Synopsis

```
#include <stdimap.h>
int IxGathersep(separate int a,
                separate long b,
                uint void *res,
                int bytes);
```

Description This function fills the uint array res with the value of b for PEs for which the value of a is not 0, and then returns how many times b values were placed in the array.



bytes defines how many bytes (either 2 or 4) are placed in res for each value placed in the array. If bytes is 2, res is regarded as a pointer to short data, and the lower two bytes of b are placed in res for each applicable PE.

If bytes is 4, res is regarded as a pointer to long data, and all 4 bytes of b are placed in res for each applicable PE.

Remark: res is assumed to have enough space to store values.
For example, if a variable that stores a PE number is defined for b, res is filled with the numbers of matching PEs.

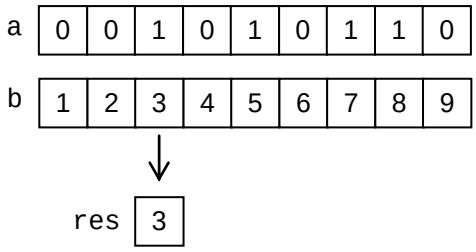
```
sep b = _PENUM;
int res[PENO];
ret = IxGathersep(a,b,res);
```

3.8.4 IxGetleftsep

Synopsis

```
#include <stdimap.h>
int IxGetleftsep(separate uchar a,
                 separate uint b,
                 uint *res);
```

Description This function places into res the leftmost b value of PEs for which a is not 0, and then returns 1 if the value could be obtained or 0 if not.

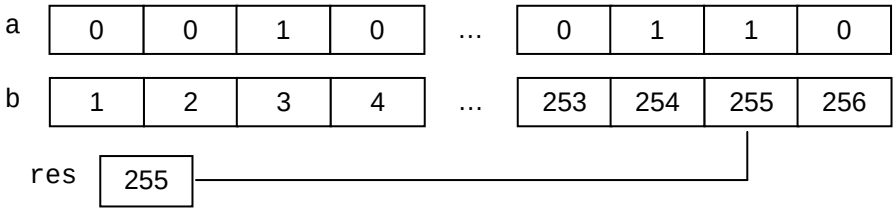


3.8.5 IxGetrightsep

Synopsis

```
#include <stdimap.h>
int IxGetrightsep(separate uchar a,
                  separate uint b,
                  uint *res);
```

Description This function places into res the rightmost b value of PEs for which a is not 0, and then returns 1 if the value could be obtained or 0 if not.



3.8.6 IxMaxsep

Synopsis #include <stdimap.h>

uint IxMaxsep(separate uint a)

Description This function returns the maximum value of all elements of a (in active PEs).

3.8.7 IxMinsep

Synopsis #include <stdimap.h>

uint IxMinsep(separate uint a)

Description This function returns the minimum value of all elements of a (in active PEs).

3.8.8 IxPack8, IxPack8i, IxPack8all

Synopsis #include <stdimap.h>

sep uchar IxPack8(sep uchar * addr)

sep uchar IxPack8i(sep uint * addr)

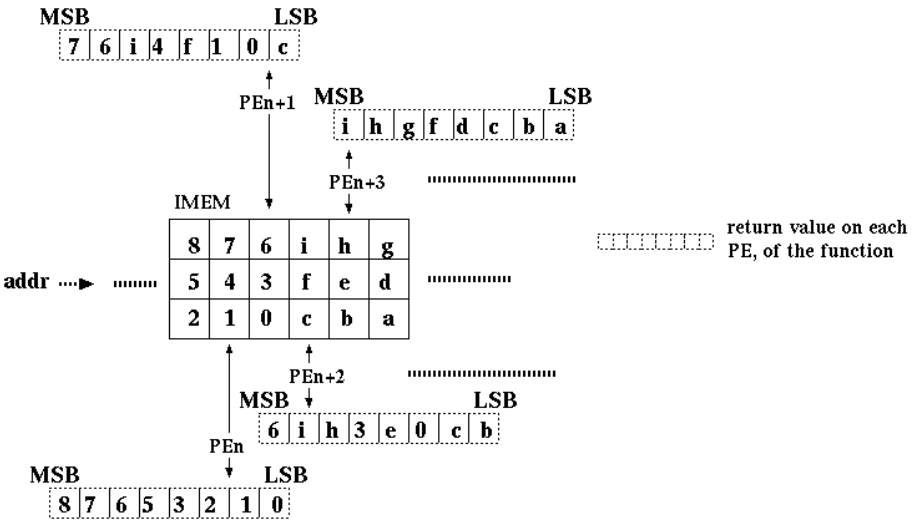
void IxPack8all(sep uchar* src, sep uchar *dst, uint lines)

Description IxPack8 and IxPack8i pack the 8 pixels adjacent to the pixel at addr, by converting non-zero bits to 1 and the rest to 0. They return the resulting 8-bit sep value.

For example, the bit pattern that results from packing the 8 adjacent pixels shown in the table below is “01001000”, so that these functions return 0x48.

Note that the return values are undefined for the PEs at the left and right edges, and are also undefined if the IMEM address indicated by *(addr – 1) or *(addr + 1) is not in the range from 0 to IX_IMEM_MAX – 1 (== 4,095).

0	65531	0
0	…	66
0	0	0



IxPack8all applies IxPack8 to the lines lines of the sep array starting at src, and writes the results to the lines lines of the sep array starting at dst.

3.8.9 IxPack8s, IxPack8si

Synopsis

```
#include <stdimap.h>
sep uchar IxPack8s(sep uchar * sep addr)
sep uchar IxPack8si(sep uint * sep addr)
```

Description These functions are the same as the `IxPack8` and `IxPack8i` functions, except the `addr` values are sep-type pointers to sep-type uchar and uint data (that is, different values are obtained for each PE).

3.8.10 IxOrdersep

Synopsis

```
#include <stdimap.h>
sep ulong IxOrdersep(sep ulong val,uint flag,uint bytes)
```

Description This function sorts the elements of the sep variable val in ascending or descending order. The meanings of the flag value are as follows:

flag = 0: ascending order (The maximum value is stored in the PE at the right edge (the PE with the largest PE number).)

flag = 1: descending order (The maximum value is stored in the PE at the left edge PE0)

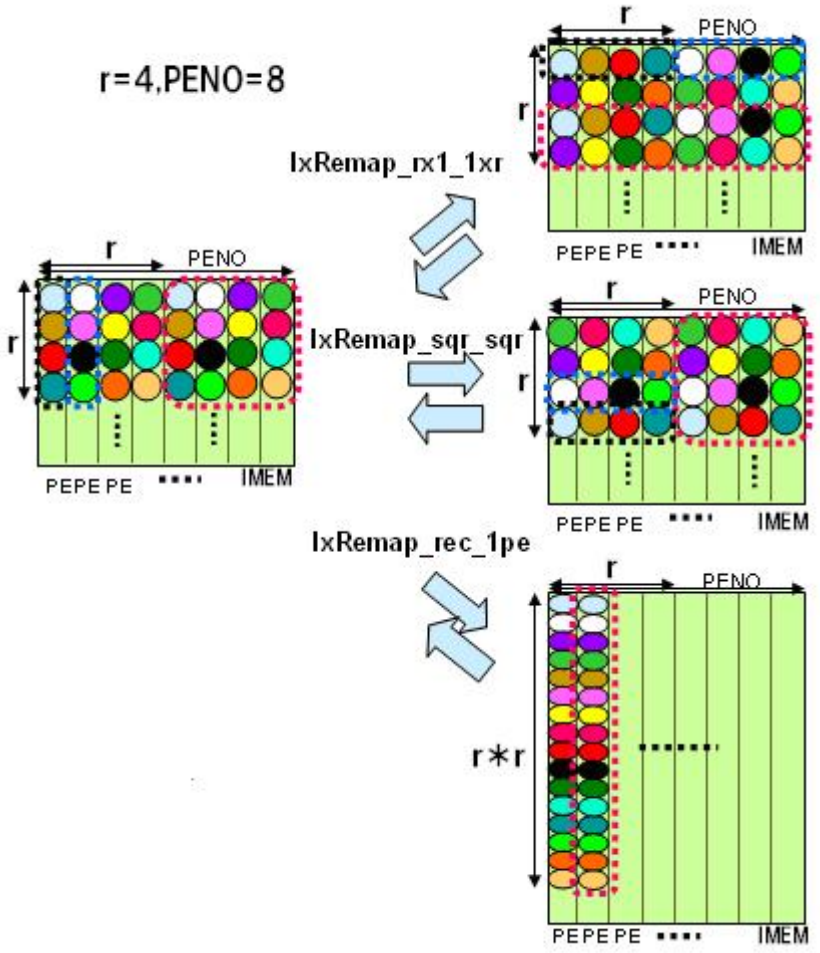
bytes specifies the actual number of bytes in each element of val. In other words, each element specifies a uint (2 bytes) if bytes is 2 or a ulong (4 bytes) if bytes is 4.

3.9 Sep Data Transposition Functions

Immediately after being transferred from EMEM to IMEM, data are not necessarily arranged as expected by algorithm designers. The functions below are used to efficiently sort such data in IMEM.

The following figure shows the operation of these sep-type data transposition functions.

Function name	Description
IxRemap_rx1_1xr	Transposes data in IMEM using the defined width r (where r is a power of 2 equal to PENO or less)
IxRemap_sqr_sqr	Uses the defined width r to rotate by 90° an r x r block in IMEM (where r is a power of 2 equal to PENO or less)
IxRemap_rec_1pe	Expands all the data within an r x r block in IMEM to one PE or vice versa (where r is a power of 2 equal to PENO or less)
IxRot90	Same as IxRemap_sqr_sqr when the defined width r equals PENO but faster



3.9.1 IxRemap_rx1_1xr

Synopsis #include <stdimap.h>

void IxRemap_rx1_1xr(sep void *src, uint bytes, uint width, uint dir, uint lines);

Description This function transposes lines lines of the IMEM data in src according to dir, and overwrites src with the result:

- If dir is `CLOCK_WISE (==1)`: width × 1 → 1 × width
- If dir is `ANTI_CLOCK_WISE (==0)`: 1 × width → width × 1

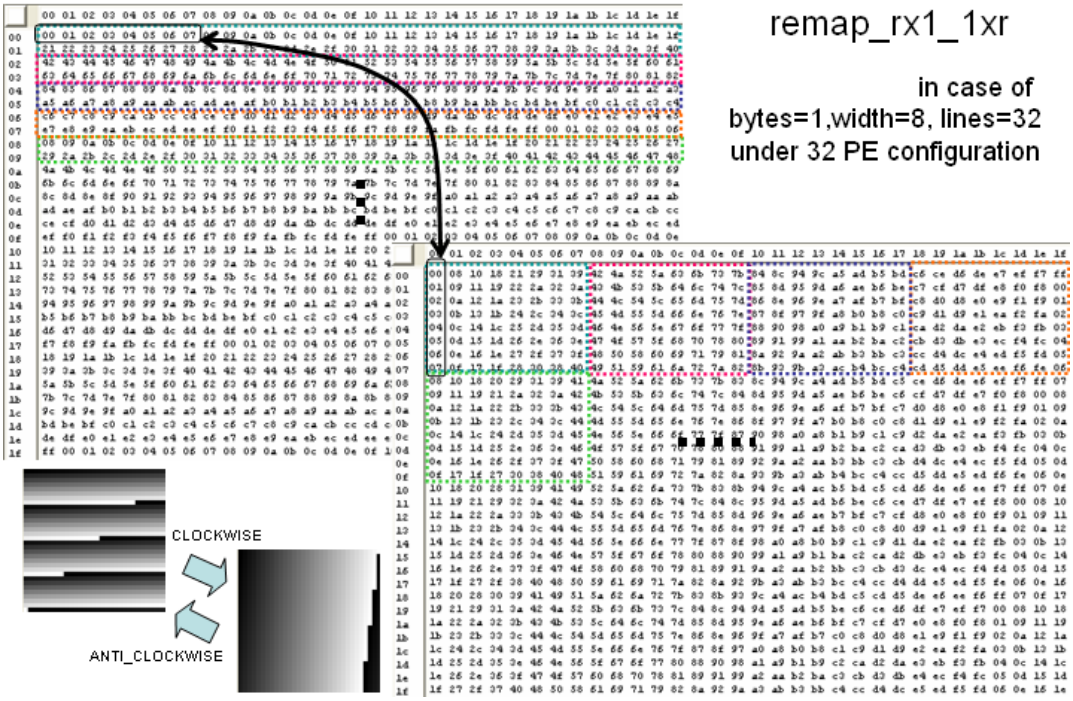
bytes specifies the units to use, which must be 1, 2, or 4 bytes.

width must be a power of 2 that is greater or equal to 2 and less or equal to PENO, and width × bytes must be greater or equal to 4.

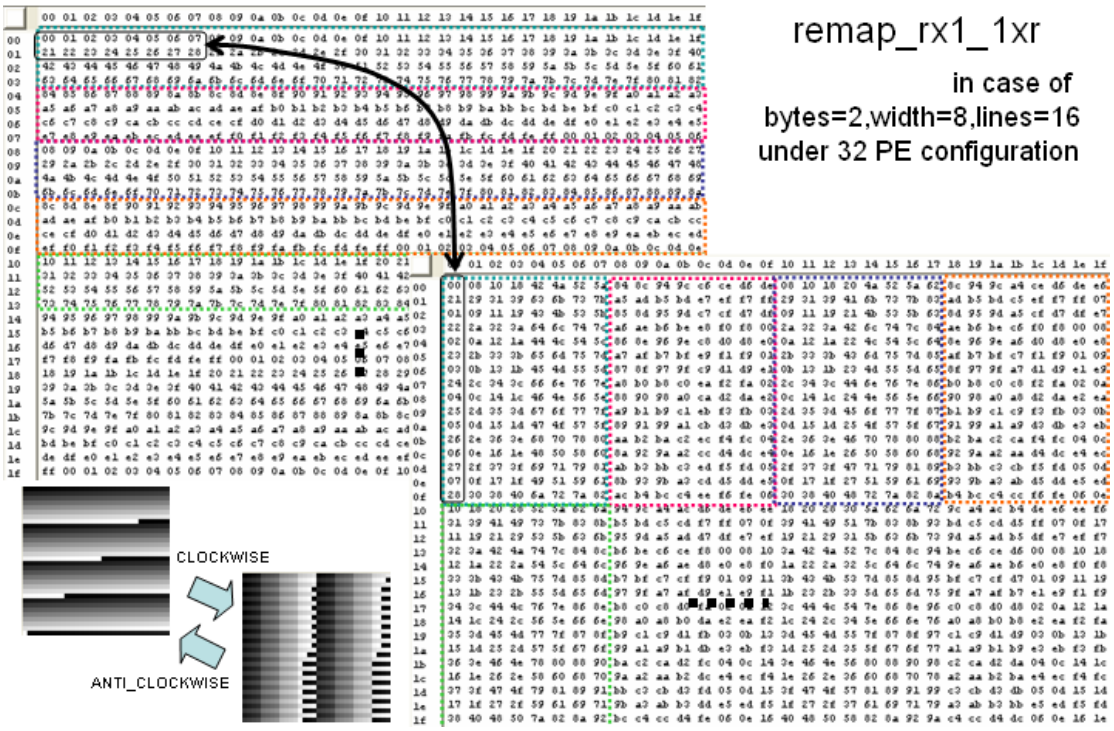
lines must be an integer multiple of width.

This function consumes approximately bytes × width lines of IMEM.

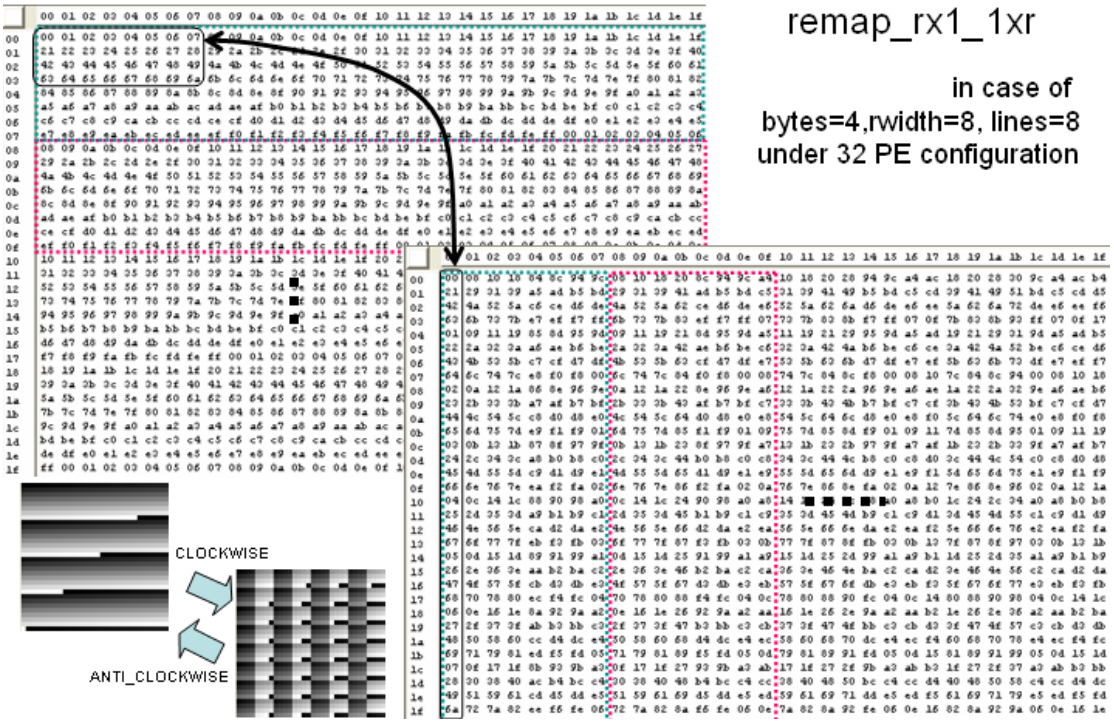
The following figure shows an operational overview of how each line is processed by IxRemap_rx1_1xr() when bytes = 1, width = 8, lines = 32, and PENO = 32.



Case when bytes = 2, width = 8, lines = 16, and PENO = 32.



Case when bytes = 4, width = 8, lines = 8, and PENO = 32.



3.9.2 IxRemap_rec_1pe

Synopsis

```
#include <stdimap.h>
void IxRemap_ rec_1pe(sep void *src,
        uint bytes,
        uint width,
        uint height,
        uint dir,
        uint hori,
        uint lines);
```

Description This function rearranges lines lines of the IMEM data in src according to dir, collecting the data from a rectangular area extending across width into one PE (in the case of CLOCK_WISE), or expanding the data in one PE into the rectangular area width–wide and height-high extending across width (in the case of ANTI_CLOCK_WISE), and then overwrites the result into src.

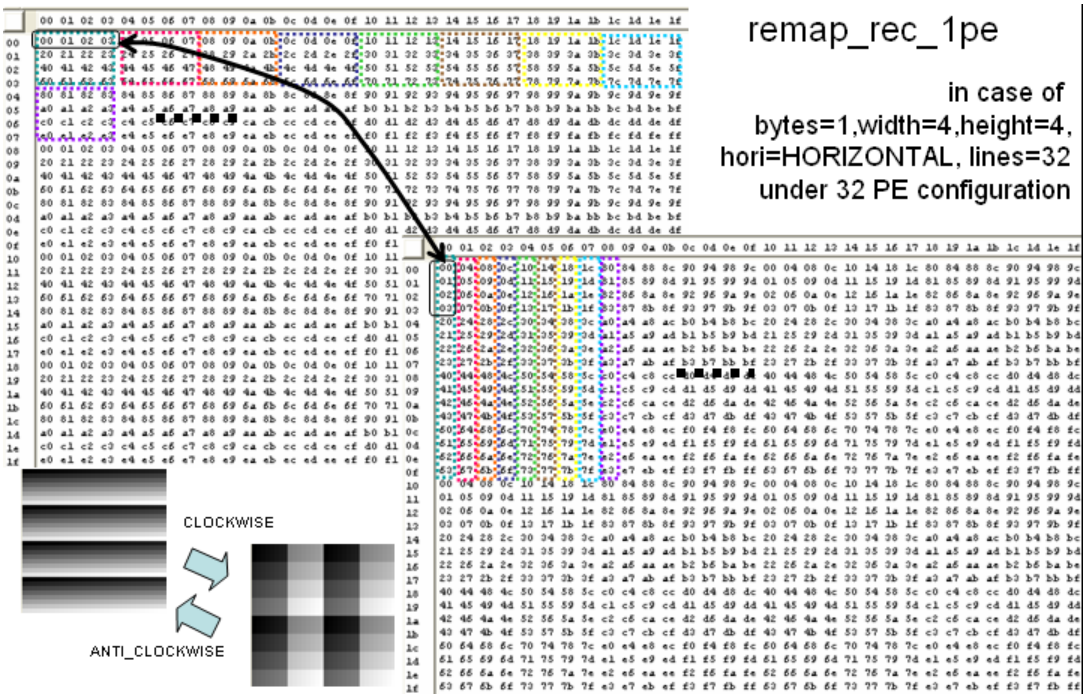
- If dir is CLOCK_WISE (==1):
  - each row of data in the first width-wide and height-high rectangular area is extracted and then placed over width x height data in one PE
  - each row of data in the next width-wide and height-high rectangular area is extracted and placed over width x height data in the next PE (immediately to the right of the first PE), etc...
- If dir is ANTI_CLOCK_WISE (==0):
  - width x height data in the first PE are arranged in the first width-wide and height-high rectangular area, one row at a time
  - width x height data in the next PE are arranged in the next width-wide and height-high rectangular area (immediately to the right of the first rectangular area), etc...

The data in the rectangular area is sequentially used in the PE direction (if hori is 1) or in the IMEM address direction (if hori is 0).

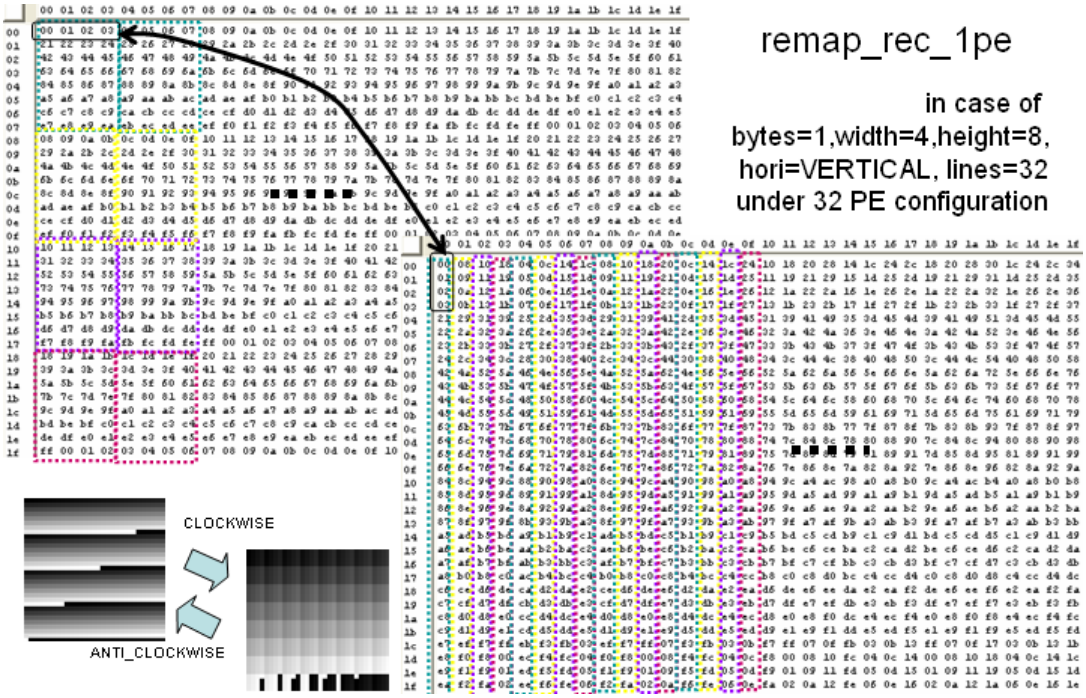
- bytes specifies the units to use, which must be 1, 2, or 4 bytes.
- width must be a power of 2 that is greater than or equal to 2 and less than or equal to the number of PEs, and width x bytes must be greater than or equal to 4.
- height must be a power of 2 that is greater than or equal to both 2 and width.
- dir must be 0 (ANTI_CLOCK_WISE) or 1 (CLOCK_WISE).
- hori must be 0 (VERTICAL) or 1 (HORIZONTAL).
- lines must be an integer multiple of width x height.

This function consumes approximately bytes x width x height lines of IMEM.

If bytes = 1, width = height = 4, lines = 32, hori = HORIZONTAL, PENO = 32.



If bytes = 1, width = 4, height = 8, lines = 32, hori = VERTICAL, and PENO = 32.



3.9.3 IxRemap_sqr_sqr

Synopsis #include <stdimap.h>

void IxRemap_sqr_sqr(sep void *src, uint bytes, uint width, uint dir, uint lines);

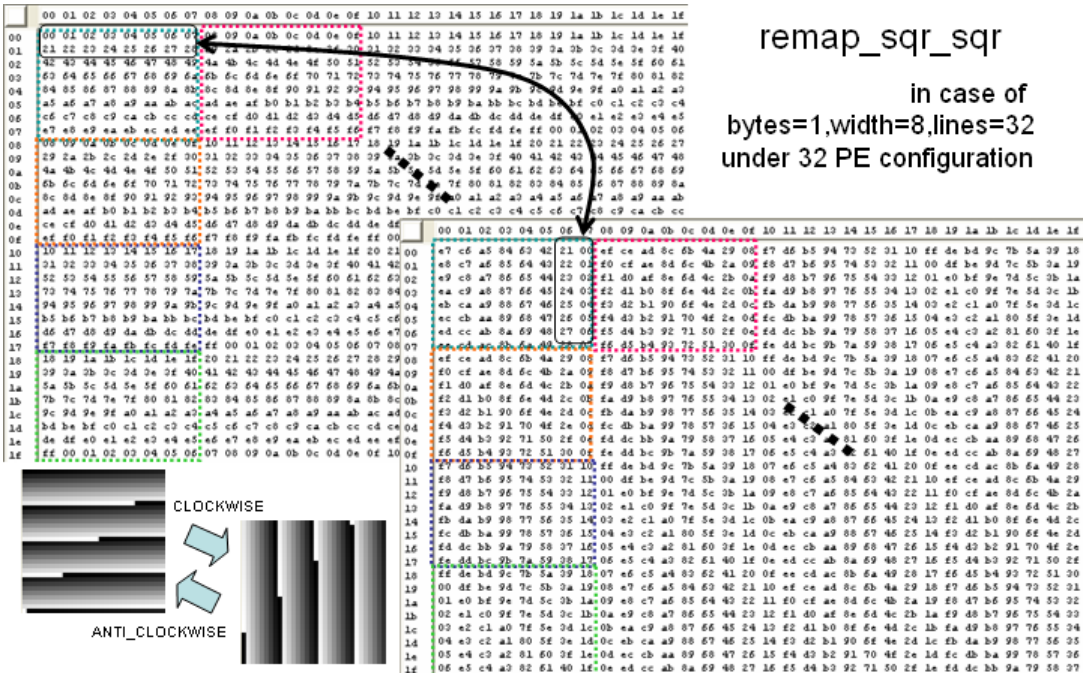
Description This function generates into src the image that results by rotating a width x width block by 90 degrees in the input image src.

For rotation direction, counterclockwise (ANTI_CLOCK_WISE), clockwise (CLOCK_WISE), or transposed (TRANSPOSE) can be defined.

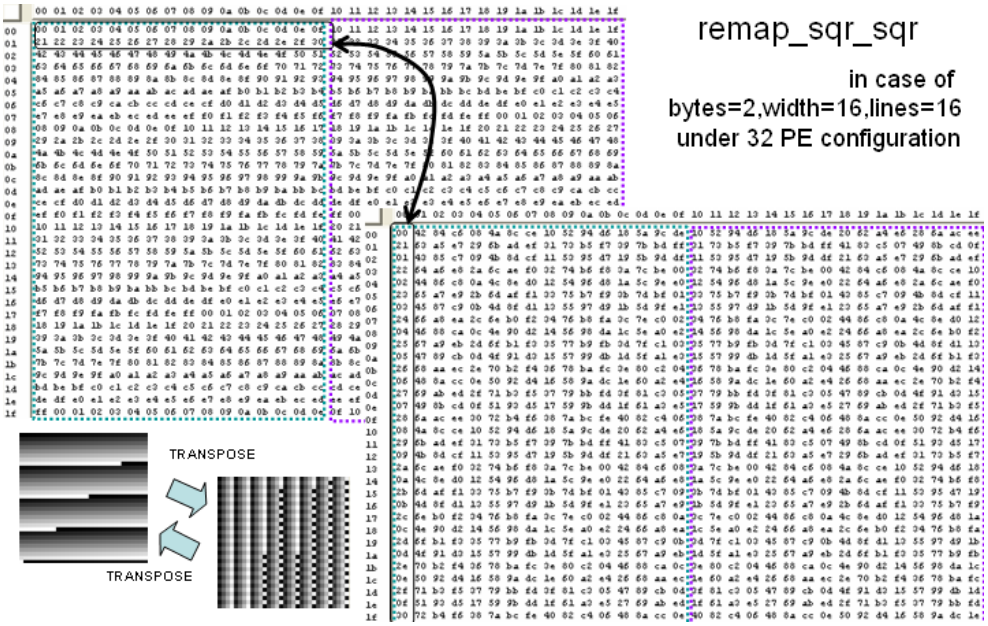
- bytes defines the units to use, which must be 1, 2, or 4 bytes.
- width must be a power of 2 that is greater or equal to 2 and less or equal to PENO, and width x bytes must be greater or equal to 4.
- dir must be 0 (ANTI_CLOCK_WISE), 1 (CLOCK_WISE), or 2 (TRANSPOSE).
- lines must be an integer multiple of width.

This function consumes approximately bytes x width lines of IMEM.

Case when bytes = 1, width = 8, lines = 32, dir = CLOCK_WISE or ANTI_CLOCK_WISE, and PENO = 32.



If bytes = 2, width = 16, lines = 316, dir = TRANSPOSE, PENO = 32.



3.9.4 IxRot90

Synopsis #include <stdimap.h>

void IxRot90(sep void *src,uint dir,uint bytes,uint lines);

Description This function rotates the entire input image src by 90 degrees and overwrites the result into src.

For rotation direction, counterclockwise (ANTI_CLOCK_WISE) or clockwise (CLOCK_WISE) can be defined.

Note that, if lines is less or equal to PENO, src is overwritten with the result of rotating that number of lines, and, if lines exceeds PENO, PENO is considered.

bytes defines the units to use, which must be 1, 2, or 4 bytes.

3.10 Basic functions for MP mode

3.10.1 ix_cp_id

Synopsis

```
#include <stdimap.h>
int ix_cp_id();
```

Description This function returns the ID number of the CP (same value as ix_total_pu_number(), described below).

3.10.2 Ix_start_mode

Synopsis

```
#include <stdimap.h>
void Ix_start_mode(ulong mode);
```

Description This function defines the hardware operating mode:

- by specifying 0xffffffff, which means that all PUs have finished, for the special register mpjoin, which indicates that each PU has finished
- and by writing the PU allocation information defined by mode to the special register mpasn, which indicates the defined PU operation.

In addition, when a different operating mode is entered from either the SIMD or MIXED mode, the data in the IMEM area starting at the IMEM zero address and ending at istkp (the IMEM stack pointer) is saved. And when the SIMD or MIXED mode is re-entered, this saved data is restored.

The mode can be specified as one of the following values:

- 0L for the SIMD mode.
- 0xaaaaaaaaL for the MIXED mode (in which the number of operating PUs is 1/8th of the PEs)
- 0xffffffffL for the MP mode (in which the number of operating PUs is 1/4th of the PEs).

Alternatively, the mode can be specified by using one of the following definitions in `1DC/cc1dc/lib/include/imap/stdimap.h`:

```
#define IX_SIMD_MODE  0L
#define IX_MP_MODE    0xffffffffL
#define IX_MIXED_MODE 0xaaaaaaaaL
```

3.10.3 lx_cp_start_pu

Synopsis

```
#include <stdimap.h>
ulong lx_cp_start_pu(int puid,
        void(*pufunc)(),
        ulong dstklm,
        ulong dstkp,
        ulong hp,
        ulong *param);
```

Description

This function starts the PU whose number is defined by puid at the starting address using pufunc. Each started PU uses the area from dstklm to hp – 4 as local data memory, and the area from dstkp to hp – 4 as stack.

dstklm must be less or equal to dstkp and a multiple of 512.

The capacity of dstklm to hp must be a multiple of 512.

param defines the only parameter passed to the function void pufunc(void*), which is used to operate the specified PU.

The return value is -1 if an invalid value is assigned to dstkp or dstklm. In other cases, the bit at the position corresponding to the PU that was successfully started is set.

3.10.4 lx_cp_start_multi_pu

Synopsis

```
#include <stdimap.h>
ulong lx_cp_start_multi_pu(ulong puno,
        void (*pufunc)(),
        void *faddr,
        ulong fnum,
        void *daddr,
        ulong dnum,
        ulong dstklm_base,
        ulong dstkp_base,
        ulong size,
        void *param);
```

Description This function starts the multiple PUs defined by puno at the starting address using pufunc. Each started PU uses the following areas (in bytes):

- The area from dstklm_base + *A* x size to dstklm_base + (*A* + 1) x size - 4 is used as local data memory.
- The area from dstkp_base + *A* x size to dstkp_base + (*A* + 1) x size - 4 is used as stack.

Here, *A* is equal to 0 for the first PU, 1 for the second, 2 for the third, and N-1 for the N^{th} PU, assuming N is equal to the number of started PUs.

dstklm_base must be less or equal to dstkp_base and a multiple of 512.

size must also be a multiple of 512.

param defines the only parameter passed to the function void pufunc(void*), which is used to operate specified PUs.

Before starting the defined PUs, this function performs the following operations to place the appropriate instruction cache lines and data cache lines in arrays:

1) If daddr is not NULL, the function processes the first dnum elements in the daddr array, and sends (broadcasts) the hit data cache lines to the PUs.

2) If faddr is not NULL, the function processes the first fnum elements in the faddr array, and sends (broadcasts) the hit instruction cache lines to the PUs.

3) Finally, the function sends to the PUs (broadcasts) instruction cache lines that result in hits at the pufunc() address.

The return value is -1 if an invalid value is assigned to dstkp or dstklm. In other cases, the bit at the position corresponding to any PU that was successfully started is set.

3.10.5 Ix_fork

Synopsis

```
#include <stdimap.h>
inline ulong    Ix_fork(ulong puno, ulong pc);
```

Description This function determines the pc starting address for the PU or Pus defined by the puno bit positions to start those PUs.

This function returns the result of the operation (-mprun) & mpasn & puno.

Note that mprun and mpasn are special registers.

3.10.6 Ix_fork_again

Synopsis

```
#include <stdimap.h>
void  Ix_fork_again(ulong puno,
                    ulong pc,
                    ulong *pca,
                    uint pcatime,
                    ulong *dca,
                    uint dctime,
                    uint waittime)
```

Description This function waits for the PU or PUs whose specific bit positions in puno are 1 to stop, and then performs steps 1) to 3) below.

If 0 is assigned to puno, PUs are currently in use. In other words, all PUs for which the corresponding mpasn bit is 1 are defined.

Note that the system waits for PUs determined by puno to stop by polling the special register mpjoin, up to 2^{32} times if waittime is 0, or up to waittime times if waittime is not 0.

1) If pcatimes is not 0, this function processes the pcatimes elements in the pca array and repeatedly calls Ix_forkp() to transfer (broadcast) the hit instruction cache lines (512 bytes each) to the PUs.

However, note that, if 3 or more elements hit the same instruction cache line on the PU side, only the first two instructions are transferred.

2) If dctimes is not 0, this function processes the dctimes elements in the dca array and repeatedly calls Ix_forkp() to transfer (broadcast) the hit data cache lines (512 bytes each) to the PUs.

However, note that, if 3 or more elements hit the same data cache line on the PU side, only the first two data items are transferred.

3) If pc is 0, the mpjoin bits corresponding to the PUs determined by puno are cleared, and these PUs restart at the position where they stopped.

If pc is not 0, the PU program counter is set to the position of the instruction code defined by pc, and the PUs restart.

3.10.7 lx_forkd

Synopsis #include <stdimap.h>

void lx_forkd(ulong puno, ulong dc);

Description This function transfers (broadcasts) 1 cache line (512 bytes) that includes the data starting at dc to the PU or PUs whose specific bit positions in puno are 1.

3.10.8 lx_forkinit

Synopsis #include <stdimap.h>

void lx_forkinit(ulong puno, ulong param)

Description This function uses the values from param and puno to initialize the special registers mpjoin, mpasn, mparea, hp, dstkp, and dstklm (shown in the table below). These registers must be set up before using a PU or PUs whose specific bit positions in puno are 1.

This function also sets other special registers and cache tags to their values immediately following a reset.

In the following table, the number of bits refers to the number of bits for each PU.

Name (number of bits)	Setting	Purpose
mpjoin (1 bit)	mpjoin | puno	Indicates whether the PU has stopped (1: stopped, 0: other status)
mpasn (1 bit	mpasn | puno	Indicates whether the PU is allocated (1: allocated, 0: not allocated)
mparea (4 bits)	param[31..28]	The fixed 4 higher bits of the 28-bit data memory address used when the PU accesses external memory
hp (24 bits)	(param[27..14]<<10)-4	PU heap pointer
dstkp (24 bits)	param[13..0]<<10	PU stack pointer
dstklm (24 bits)	param[13..0]<<10	Minimum PU stack address

3.10.9 lx_forkp

Synopsis #include <stdimap.h>

void lx_forkp(ulong puno, ulong pc)

Description This function transfers one cache line (512 bytes) that includes the program data starting at pc to the PU or PUs whose specific bit positions in puno are 1.

3.10.10 lx_initm

Synopsis #include <stdimap.h>

void lx_initm()

Description This function clears the message channel (invalidating all messages).

3.10.11 ix_is_MIXED, ix_is_MP, ix_is_SIMD

Synopsis #include <stdimap.h>

void ix_is_MIXED()

void ix_is_MP()

void ix_is_SIMD()

Description ix_is_SIMD returns a non-zero value if the current operating mode is SIMD, or a zero otherwise.

ix_is_MIXED returns a non-zero value if the current operating mode is MIXED, or a zero otherwise.

ix_is_MP returns a non-zero value if the current operating mode is MP, or a zero otherwise.

3.10.12 lx_join

Synopsis #include <stdimap.h>

int lx_join(ulong puno,ulong time)

Description This function waits for the PU or PUs whose specific bit positions in puno are 1 to stop[Note 1], or waits for the usage of all PUs[Note 2] to stop[Note 1] if 0 is assigned to puno.

This function polls the value of the special register mpjoin up to 2^{32} times if time is 0 or up to time times if time is not 0.

Note 1. Indicates that the corresponding bit in the special register mpjoin is 1

Note 2. Indicates PUs for which the corresponding bit in the special register mpasn is 1

The return value is 1 if the execution of the specified PU did not finish, or 0 otherwise.

3.10.13 Ix_mutex_init, Ix_mutex_destroy

Synopsis

```
#include <stdimap.h>
uint Ix_mutex_init(ix_mutex_t * mtx)
uint Ix_mutex_destroy(ix_mutex_t * mtx)
```

Description

The Ix_mutex_init function generates the mutex variable defined by mtx. The function returns a non-zero value if the variable is successfully generated, zero otherwise.

Note that, when calling Ix_mutex_init in the SIMD mode, the mutex variable is not correctly generated and the return value is zero.

Ix_mutex_destroy discards the mutex variable defined by mtx. The function returns a non-zero value if the variable is successfully discarded, zero otherwise.

These functions can only be executed on a CP, and their operation is not guaranteed when executed in a PU.

Usage example:

```
#include <stdimap.h>
ix_mutex_t mtx;
outside unsigned int mutx;
#pragma Kmp=1
func(){
 ix_mutex_lock(&mtx);
    ~Exclusively access variables here.~
 ix_mutex_unlock(&mtx);
}

#pragma Kmp=0
main(){
Ix_start_mode(IX_MP_MODE);  // Switch to the MP mode.
Ix_mutex_init(&mtx); // Call this function to generate the mutex variable mtx.
start_multiple_pu_to_execute_func();// Generate multiple PUs that execute func().
Ix_mutex_destroy(&mtx); // Discard the mutex variable mtx.
}
```

3.10.14 ix_mutex_lock, ix_mutex_unlock

Synopsis #include <stdimap.h>

uint ix_mutex_lock(ix_mutex_t * mtx)

uint ix_mutex_unlock(ix_mutex_t * mtx)

Description The ix_mutex_lock function locks the mutex variable defined by mtx (causing the system to wait until the variable can be used), and returns a non-zero value if lock was successful, zero otherwise.

Note that any request to lock a locked mutex variable is ignored, and the lock is considered as successful.

The ix_mutex_unlock function unlocks the mutex variable defined by mtx (and returns it to the system). The function returns a non-zero value if unlock was successful, zero otherwise.

Note that any request to unlock an unlocked mutex variable is ignored, and the unlock is considered as successful.

For an example of using these functions, see the section describing the Ix_mutex_init function.

3.10.15 ix_modify_dstkp

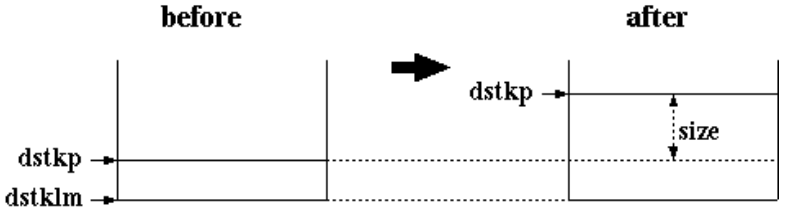
Synopsis #include <stdimap.h>

void* ix_modify_dstkp(ulong size)

Description This function takes the data in the area from the minimum value of the stack pointer (dstklm) to the stack pointer (dstkp). It copies this data to the area from dstklm + size to dstkp + size, and adds size to dstkp.

size must be a positive integer that is a multiple of 4. If size is not a multiple of 4, the nearest multiple of 4 that is less than size is used.

This function returns the dstklm value.



3.10.16 ix_pu_id

Synopsis #include <stdimap.h>

int ix_pu_id();

Description This function returns the currently executing PU or CP ID number. When this function is called from a program running on a CP, the same value as ix_total_pu_number() is returned.

3.10.17 ix_pu_local_area

Synopsis

```
#include <stdimap.h>
void* ix_pu_local_area(int size)
```

Description This function increases or decreases the values of the stack pointer (dstkp) and minimum value of the stack pointer (dstklm) by size bytes, and moves data in the range from dstklm to dstkp.

However, size must be a multiple of 1,024. If size is not a multiple of 1,024, the nearest multiple of 1,024 that is less than size is used.

If size is a positive value, using this function allocates a local PU area of size size, at a position before the stack that can be used in common by functions executed in PUs, and then returns a function pointer to the beginning of the area.

If size is a negative value, the function returns size bytes of the most recently allocated local PU area to the system. The values of dstkp and dstklm are considered as dstkp + size and dstklm + size, and data is moved in the range from dstklm to dstkp.

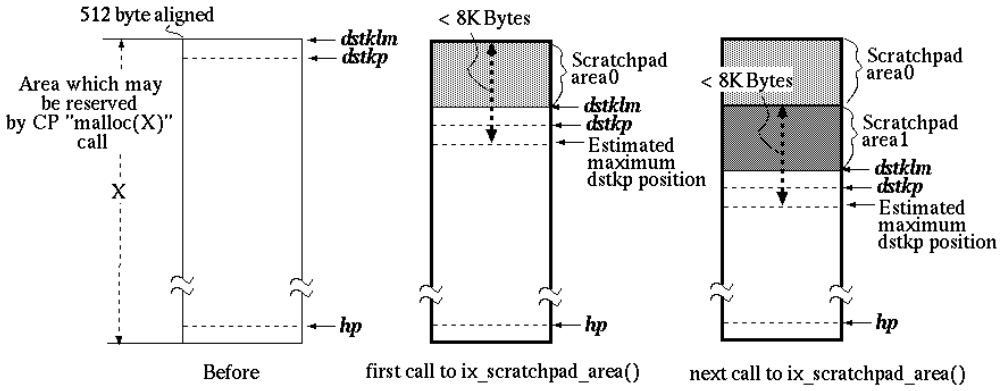
Any signed short integer can be assigned to size, but, if dstkp + size exceeds the area pointed to by the heap pointer (hp), the local PU area is not allocated, and the function returns 0.

The local PU area obtained by calling this function with a positive size value is always arranged continuously within the stack (dstklm to dstkp).

This characteristic can be used to minimize the frequency of data cache misses under user control:

- by specifying the size of the local PU area such that it is within 8 KB, which is the capacity in the data cache for each PU, and is equal to the “size of the local PU area + the assumed maximum capacity in the stack that is used”
- by limiting reference definitions of explicit data to the local PU area[Note] only. This refers to a case, for example, in which data reference definitions are required by other areas, and the use of references, definitions, or NC (non-cache access) after performing an ROI transfer between a local PU area and another area has been set up on the user side.

The following figure shows the memory map when this function is called twice with a positive value defined for size.



Note that the operation is not guaranteed when this function is used to return an area to the system that is larger than the allocated data area.

3.10.18 ix_TLSread_{u1, u2, u4, s1}, ix_TLSwrite{u1, u2, u4}

Synopsis

```
#include <stdimap.h>
inline ulong ix_TLSread_u4(long a);
inline uint  ix_TLSread_u2(long a);
inline uchar ix_TLSread_u1(long a);
inline char  ix_TLSread_s1(long a);
inline void  ix_TLSwrite_u4(long a, ulong v);
inline void  ix_TLSwrite_u2(long a, uint v);
inline void  ix_TLSwrite_u1(long a, uchar v);
```

Description These inline functions provide access to the data area allocated using ix_pu_local_area at the relative position indicated by dstklm.

The parameter a defines the access index, and v defines the value to write during write access. As shown in the following usage example, each access index value can be used as the identifier for a PU-specific data area (which can be read even if it extends over a function). In other words, these index values can be used as thread-local storage (TLS).

```
#define TLSr_u4(idx)   ix_TLSread_u4(-4-4*(idx))
#define TLSw_u4(idx,v) ix_TLSwrite_u4(-4-4*(idx),v)

#define A 0  // TLS variable A
#define B 1  // TLS variable B

#pragma Kmp=1   // Following code can be executed on CP or PU
void func() {
  // read from TLS area
  printf("a=0x%lx,b=0x%lx\n",TLSr_u4(A),TLSr_u4(B));
}
void pumain(void * params) {
  ulong puid = ix_pu_id();

  ix_pu_local_area(4096); // reserve 4KB of stack bottom area
  TLSw_u4(A,0xabcd+(puid<<24));  // write to TLS area
  TLSw_u4(B,0x1234+(puid<<24));  // write to TLS area
  func();
}
```

3.10.19 ix_total_pu_number, ix_max_total_pu_number

Synopsis

```
#include <stdimap.h>
int ix_total_pu_number();
int ix_max_total_pu_number();
```

Description

ix_total_pu_number() returns the actual number of usable PUs.

ix_max_total_pu_number() returns the maximum number of PUs that can be used by the program from which it is called.

Be careful when using ix_total_pu_number() and ix_max_total_pu_number() for purposes such as declaring the sizes of arrays, because their return values (including the attributes of these values, that is, whether they are constants) differ depending on the values defined for the compiler options PENO and PEMAX as shown in the following table.

	ix_total_pu_number()	ix_max_total_pu_number()
0 is assigned to PENO when compiling	Determined at execution (variable)	-
A value other than 0 is defined to PENO when compiling	Constant (PENO/4)	-
PENO is not defined when compiling	Constant (128/4 = 32)	-
PEMAX is defined when compiling	-	Constant (PEMAX/4)
PEMAX is not defined when compiling	-	Constant (128/4 = 32)

3.11 Functions to send and receive messages in MP mode

In the MP or MIXED mode, messages can be exchanged between PUs and CPs or between PUs, using a C-ring. This section provides a simple description of basic background information, and describes the basic specifications of functions used to send and receive messages in the MP or MIXED mode.

3.11.1 Message format

Three types of messages (data messages, response messages, and interrupt messages) can be sent. The message format is as follows:

Format of data messages and response messages

	MSB ---------- LSB						Total
Field name	V	ID	BC	TARGET	SOURCE	TID	DATA	
Number of bits	1	2	1	6	6	16	32 × 3	128 bits

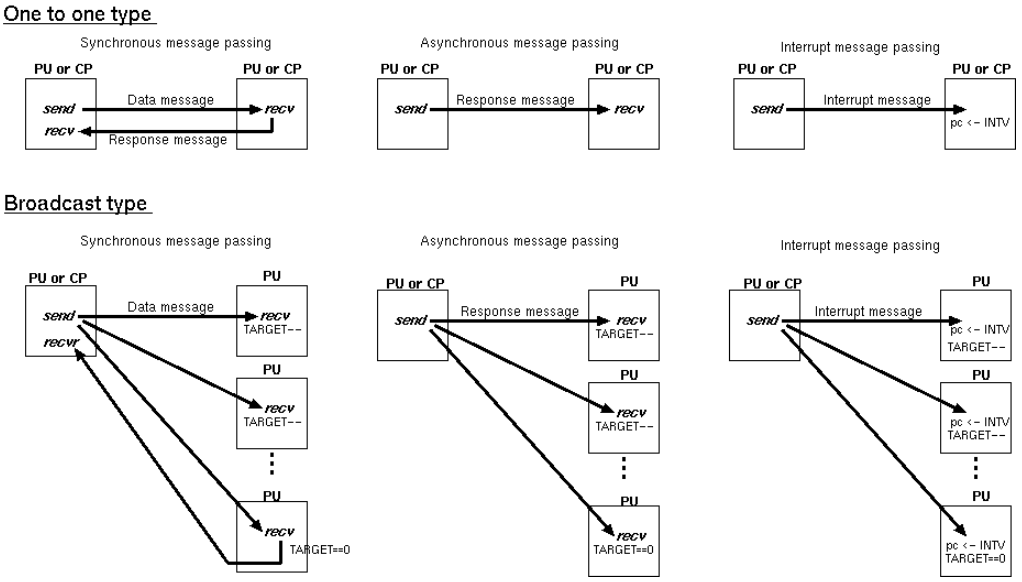
Format of interrupt messages

	MSB ---------- LSB						Total
Field name	V	ID	BC	TARGET	INTV		DATA	
Number of bits	1	2	1	6	22		32 × 3	128 bits

Name	Number of bits	Description
V	1	Indicates whether the message is valid. 0:invalid, 1:valid
ID	2	Data message: 00 (has an automatic response) Data message: 01 (has en automatic response) Response message: 10 Interrupt message: 11
BC	1	Indicates whether the message is a broadcast-type message. 1: broadcast type, 0: not a broadcast type
TARGET	6	The destination PU number (from 0 to the number of PUs - 1). CP is the number of PEs divided by 4. For a broadcast-type message, this can also be used to count the remaining destinations
SOURCE	6	The source PU number (from 0 to the number of PUs – 1). CP is the number of PEs divided by 4.
TID	16	Message ID
DATA	32 x 3	Message body (ignored for interrupt messages)
INTV	22	Interrupt vector (starting address of the interrupt routine)

3.11.2 Patterns to send and receive messages, and coding examples

When using the three types of messages, the following models for sending and receiving are available.



The table below shows several coding examples. Note that TID represents any 16-bit data (and is used as the message ID), and Xa to Xf represent any 32-bit data.

- Example for one-to-one synchronous message passing

Example code on the sender side to send message with defined ID (=TID) to a specific PU or CP, and wait for a response	Example code on the receiver side to receive message with defined ID (=TID) from any transmission source, and attach the data portion to the response message
#include <stdimap.h>	#include <stdimap.h>
ix_MSG msg;	ix_MSG msg;
int target = ix_cp_id(); // if sending to CP	int target = ix_pu_id(); // own ID
ulong received_data[3];	ulong received_data[3];
msg.hd = *ix_msg_header_sync*(target,TID,0);	msg.hd = *ix_msg_header*(target,TID,0);
msg.d2 = … // Data body to send (=Xa)	msg.d2 = … // Response data body (=Xd)
msg.d1 = … // Data body to send (=Xb)	msg.d1 = … // Response data body (=Xe)
msg.d0 = … // Data body to send (=Xc)	msg.d0 = … // Response data body (=Xf)
ix_send_msg(&msg,0);	
ix_recv_msg(&msg,0);	*ix_recv_msg*(&msg,0);
received_data[2] = msg.d2; // (=Xd)	received_data[2] = msg.d2; // (=Xa)
received_data[1] = msg.d1; // (=Xe)	received_data[1] = msg.d1; // (=Xb)
received_data[0] = msg.d0; // (=Xf)	received_data[0] = msg.d0; // (=Xc)

- Example for one-to-one asynchronous message passing (1)

Example code on the sender side to send response message with defined ID (=TID) to a specific PU or CP (to synchronize operation)	Example code on the receiver side to receive message with defined ID (=TID) from a specific PU or CP (to synchronize operation)

#include <stdimap.h>
int target = ix_cp_id(); // if sending to CP
ix_send(*ix_msg_header*(target,TID,0),0);
	#include <stdimap.h>
	int target = 3; // if receive from PU3
	ix_recv(target,TID,0);

- Example for one-to-one asynchronous message passing (2)

Example code on the sender side to send response message with defined ID (=TID) to a specific PU or CP	Example code on the receiver side to receive message with defined ID (=TID) from any transmission source

#include <stdimap.h>
ix_MSG msg;
int target = ix_cp_id(); // if sending to CP
ulong received_data[3];
msg.hd = *ix_msg_header*(target,TID,0);
msg.d2 = … // Data body to send (=Xa)
msg.d1 = … // Data body to send (=Xb)
msg.d0 = … // Data body to send (=Xc)
ix_send_msg(&msg,0);
	#include <stdimap.h>
	ix_MSG msg;
	int target = ix_pu_id; // own ID
	ulong received_data[3];
	msg.hd = *ix_msg_header*(target,TID,0);
	ix_recv_msg(&msg,0);
	received_data[2] = msg.d2; // (=Xa)
	received_data[1] = msg.d1; // (=Xb)
	received_data[0] = msg.d0; // (=Xc)

- Example for one-to-one interrupt message passing

Example code on the sender side to send an interrupt message to a specific PU or CP	Example interrupt routine code on the receiver side to receive an interrupt message from any transmission source

#include <stdimap.h>
int target = ix_cp_id(); // if sending to CP
extern void ifunc(); // Interrupt routine
ulong hd = *ix_msg_header_intr*(target,ifunc,0);
ix_send(hd,0);
	#include <stdimap.h>
	#pragma Kinterrupt = 1
	void ifunc(){
	// define interrupt routine
	// processing here
	}
	#pragma Kinterrupt = 0

- Example for braodcast-type synchronous message passing

Example code on the sender side to send N+1 response or messages with defined ID (=TID) to any destination	Example code on the receiver side to receive message with defined ID (=TID) from any transmission source

#include <stdimap.h>
// send N+1 response messages
ix_send_any(N,TID);
// Below, when all N+1 messages are received, a // response message is sent back
ix_send_any_sync(N,TID);
ix_recv_any(TID,0); // Receive response messages
	#include <stdimap.h>
	ix_recv_any(TID,0);

3.11.3 Restriction on the number of message buffers

A circular network called a C-ring is used to connect PUs to exchange messages.

However, care must be taken to avoid deadlocks when designing patterns to send and receive messages, because only "(number of PEs / 4) / 2" (or "number of PUs / 2") physical buffers to store messages can exist in a C-ring.

For example, consider the following pattern to send and receive messages:

1) After each interrupt message to start the interrupt handler running on the CP (for which multiple interrupts are prohibited) is sent from any PU, the system waits until one message sent by a CP arrives.

2) When the CP receives an interrupt message from any PU, the interrupt routine sends one response message to the PU.

First, when half the PUs in 1) finish sending their interrupt messages, the buffers in the C-ring are full, and the remaining half of the PUs must wait to send their messages.

In 2), the CP empties one message buffer each time it receives an interrupt message from a PU, but that buffer is immediately used by one of the PUs waiting to send its interrupt message.

Therefore, the CP is unable to send response messages to PUs, and, because multiple interrupts are prohibited for the interrupt handler running on the CP, interrupt messages from other PUs are not received and a deadlock occurs.

To work around the deadlocks caused by a pattern similar to the above, the following two approaches seem feasible:

a) Create a design in which multiple interrupts are allowed for the interrupt handler on the CP.

b) Limit the number of PUs that can send messages to the CP at once.

Of these methods, a) is not currently supported, so the following describes how to use method b):

1) The CP sends a message allowing interrupt issue to a small number of PUs at any destinations.

2) Only the PUs that received the above message send interrupt messages to the (interrupt handler running for the PU on the) CP.

3) The CP uses its PU interrupt handler to send response messages to the appropriate PUs, sends a message allowing interrupt issue to new PUs, and then stops.

Although performing the above 3 steps results in successive operations between PUs, the C-ring traffic is reduced and the reverse effects on the performance are minimal.

3.11.4 ix_msg_header

Synopsis

```
#include <stdimap.h>
#define ix_msg_header(target_or_count, tid, bc) \
 ( ( (ulong) ( ( 1<<15 ) | ( 2<<13 ) | ( ((bc)&1)<<12 ) | \
 ( ((target_or_count)&0x3f)<<6 ) | ix_pu_id() )<<16 ) + ((tid)&0xffff))
```

Description This macro returns the 32-bit header information for a normal message (a data or response message). This macro can be used to generate headers for both sending and receiving messages.

To generate a header for sending a message, assign 1 to bc if the message is of broadcast type, or 0 if the message is of one-to-one type. When generating a header for receiving a message, the value assigned to bc is ignored.

Define the message ID for tid.

Define one of the following values for target_or_count:

If 0 is assigned to bc:

- If generating a header to send a message, define the number of the destination PU or CP.
- For a header to receive a one-to-one type message, define the number of the PU or CP that sent the message.
- For a header to receive a broadcast-type message, define the number of the receiving PU or CP.

If 1 is assigned to bc:

- Define the value of “number of receptions – 1” for a broadcast-type message used for sending. Note that, if the number of receptions for a broadcast-type message is N, that message automatically disappears after being received N times by a PU[Note].

For the one-to-one type, the message disappears when the destination PU or CP (defined by target_or_count) that is waiting to receive message with tid ID receives a message that has a header returned by this macro and is in the C-ring.

For the broadcast type, the message automatically disappears after any PU[Note], that is waiting to receive the message with tid ID, receives a message that has a header returned by this macro and is in the C-ring target_or_count + 1 times.

Note: The CP does not receive broadcast-type messages.

3.11.5 ix_msg_header_intr

Synopsis

```
#include <stdimap.h>
#define ix_msg_header_intr(target_or_count,ifunc,bc)\
(((ulong)((1<<15)|(3<<13)|(((bc)&1)<<12)|(((target_or_count)&0x3f)<<6))<<16)\
+(ulong)(ifunc))
```

Description This macro returns the 32-bit header information for an interrupt message used for sending.

Define either the number of the PU or CP to receive the message or the number of receptions for target_or_count, define the starting address of the interrupt routine for ifunct, and define either 1 (for the broadcast type) or 0 (for the non-broadcast type) for bc.

For the non-broadcast type, only one interrupt message, that has a header returned by this macro, is sent until the destination PU or CP indicated by target_or_count is interrupted, and then the interrupted PU or CP jumps to the interrupt routine starting at the address indicated by ifunc.

For the broadcast type, messages that have headers generated by this macro are sent to PUs that can be interrupted or the CP target_or_count + 1 times, and then the messages disappear. Each interrupted PU or CP jumps to the corresponding interrupt routine starting at the address indicated by ifunc.

3.11.6 ix_msg_header_sync

Synopsis

```
#include <stdimap.h>
#define ix_msg_header_sync(target_or_count,tid,bc)\
(((ulong)((1<<15)|(1<<13)|(((bc)&1)<<12)|(((target_or_count)&0x3f)<<6)|\
ix_pu_id())<<16)+((tid)&0xffff))
```

Description This macro returns the 32-bit header information for a message that has an automatic response and is used for sending.

For bc, define either 1 for the broadcast type or 0 for the one-to-one type.

For target_or_count, define either the number of the PU or CP to receive the message if bc is 0 or the value of “number of receptions – 1” if bc is 1.

Define the message ID for tid.

For the one-to-one type, the message disappears when the destination PU or CP defined by target_or_count, that is waiting to receive the message with the tid ID, receives a message that has a header returned by this macro, and a response message is automatically sent to the sender of the message.

For the broadcast type, the message disappears after any PU[Note], that is waiting to receive the message with tid ID, receives a message that has a header returned by this macro target_or_count + 1 times, and a response message is sent from the last receiver to the sender of the message.

Note: The CP does not receive broadcast-type messages.

3.11.7 ix_msg_id

Synopsis

```
#include <stdimap.h>
#define ix_msg_id(hd)      ((hd) & 0xffff)
```

Description This macro returns the ID information in the message header hd.

3.11.8 ix_msg_receiver

Synopsis

```
#include <stdimap.h>
#define ix_msg_receiver(hd) (((hd)>>22) & 0x3f)
```

Description This macro returns the destination information in the message header hd.

3.11.9 ix_msg_sender

Synopsis

```
#include <stdimap.h>
#define ix_msg_sender(hd)   (((hd)>>16) & 0x3f)
```

Description This macro returns the sender information in the message header hd.

3.11.10 ix_msg_type

Synopsis

```
#include <stdimap.h>
#define ix_msg_type(hd)     (((hd)>>29) & 0x3)
```

Description This macro returns the message type information in the message header hd.

The following three types of messages exist:

0 or 1: Messages that have automatic responses (also called data messages)

2: Normal messages (also called response messages)

3: Interrupt messages

3.11.11 ix_recv

Synopsis

```
#include <stdimap.h>
inline ulong     ix_recv(uint source, uint id, uint max_retry);
```

Description This function waits to receive the message with the ID id sent by the PU with the number source. If the number of the receiving PU is defined for source, a message can be received without defining a sender.

This function attempts to receive a 32-bit message up to max_retry times, but, if max_retry is 0, the function retries until successful. The return value is received data for which the MSB is 1 if reception was successful or 0 if reception failed.

Received messages actually contain 96 bits of data, but this data is discarded if this function is used to receive a message. To also receive the data, use ix_recv_msg().

3.11.12 ix_recv_any

Synopsis

```
#include <stdimap.h>
ulong ix_recv_any(uint id)
```

Description This function waits to receive the message with the ID id from any sender.

Received messages actually contain 96 bits of data, but this data is discarded if this function is used to receive a message. To also receive the data, use ix_recv_msg().

The return value is 1 if reception was successful or 0 if reception failed.

This function is defined as follows in stdimap.h:

```
#define ix_recv_any(id)    ix_recv(ix_pu_id(),(id),0)
```

3.11.13 ix_recv_msg

Synopsis

```
#include <stdimap.h>
uint ix_recv_msg(ix_MSG* msg_p, uint retry_cnt);
```

Description The ix_MSG structure is defined as follows in:

`1DC/cc1dc/lib/include/imap/ stdimap.h`

```
typedef struct {
  ulong d0;
  ulong d1;
  ulong d2;
  ulong hd;
} ix_MSG;
```

msg_p->hd is a message header.

The PU that has the PU number represented by the 16 higher bits of the header (= *X*) is the sender. The message that has the ID represented by the 16 lower bits of the header (=*Y*) is awaited. However, by defining the number of the receiving PU for *X*, a message can be received without defining a sender.

This function attempts to receive a message up to max_retry times, but, if max_retry is 0, the function retries until successful. The received message is the 128-bit area starting at the msg_p address. In other words, the message is stored in the area from msg_p->d0 (the lower 32 bits) to msg_p->hd (the higher 32 bits).

The return value is not 0 if reception is successful or 0 if reception failed.

Note that, when a data message is received, the response message whose body was originally stored in the area from msg_p->d0 (the lower 32 bits) to msg_p->hd (the higher 32 bits) is automatically returned to the sender.

3.11.14 ix_recv_msg_blk

Synopsis

```
#include <stdimap.h>
void ix_recv_msg_blk(void *buf, uint size_in_long, uint id);
```

Description This function repeatedly receives messages with the ID id and writes the received data to buf at the offset position defined by the received messages. This function continuously receives messages until the number of words (where 1 word is 32 bits) defined by size_in_long are received.

This function is designed to receive an array of messages sent by the ix_send_msg_blk function. Therefore, define the same transfer size (size_in_long) for ix_recv_msg_blk that is defined for ix_send_msg_blk. Operation is not guaranteed if a different size is defined.

3.11.15 ix_send

Synopsis #include <stdimap.h>

```
inline uint     ix_send(ulong message, uint max_retry);
```

Description This function attempts to send a 32-bit message up to max_retry times, but, if max_retry is 0, the function retries until successful. The return value is 0 if the sending failed or not 0 if the sending was successful.

Messages that are up to 128 bits can be sent, but this function sends only the higher 32 bits of a message corresponding to its header, and the remaining 96 bits are undefined. To send 128 bits of message data, use ix_send_msg().

3.11.16 ix_send_any

Synopsis #include <stdimap.h>

```
uint ix_send_any(uint num, uint id)
```

Description This function waits until a broadcast-type response message with the ID id is successfully sent, and automatically disappears after being received by a PU num + 1 times.

The return value is 0 if the sending failed or not 0 if the sending was successful.

This function is defined as follows in stdimap.h:

```
#define ix_send_any(num,id) ix_send(ix_msg_header(num,(id),1),0)
```

3.11.17 ix_send_any_sync

Synopsis #include <stdimap.h>

```
uint  ix_send_any_sync(uint num, uint id)
```

Description This function waits until a data message with the ID id is successfully sent, and automatically disappears after being received by a PU num + 1 times. Note that, after the message is sent and received num + 1 times, a response message arrives in reply that must be received, such as by using ix_recv_any.

The return value is 0 if the sending failed or not 0 if the sending was successful.

This function is defined as follows in stdimap.h:

```
#define ix_send_any_sync(num,id) ix_send(ix_msg_header_sync(num,(id),1),0)
```

3.11.18 ix_send_msg

Synopsis

```
#include <stdimap.h>
uint    ix_send_msg(ix_MSG* msg_p, uint retry_cnt);
```

Description The ix_MSG structure is defined as follows in `1DC/cc1dc/lib/include/imap/ stdimap.h`:

```
typedef struct {
  ulong d0;
  ulong d1;
  ulong d2;
  ulong hd;
} ix_MSG;
```

This function sends 128-bit messages for which the 32 bits of msg_p->hd are the header and d0 to d2 are the body.

ix_send_msg() attempts to send a 128-bit message starting at the msg_p address up to max_retry times, but, if max_retry is 0, the function retries until successful. The return value is 0 if the sending failed or not 0 if the sending was successful.

If a data message for which the value of the ID field (2 bits of information that determine the message type and differ from the message ID) in the message header (the hd field of ix_MSG) is sent (that is, if a message that has a header generated by the ix_msg_header_sync function is sent), a response message is automatically returned when the data message is received, so make sure to receive response messages, by using ix_recv(), at the appropriate times.

3.11.19 ix_send_msg_blk

Synopsis

```
#include <stdimap.h>
void ix_send_msg_blk(void *buf, uint size_in_long, uint id);
```

Description This function repeatedly reads data from the area whose starting address is determined by buf until size_in_long words of data (where 1 word is 32 bits) have been read. Then it sends each read data set as the body of a message that has the message ID id. These messages can be received by executing ix_recv_msg_blk with the same message ID defined in any PU that is waiting to receive a message that has that ID.

This function holds order information about each sent message. Therefore, note that, if there are multiple receivers, messages are stored in an array on the receiver side at offset positions that depend on the order in which they were received.

This function is designed for the array of messages it sends to be received by ix_recv_msg_blk. Therefore, define the same transfer size (size) for ix_send_msg_blk that is defined for the ix_recv_msg_blk function. Operation is not guaranteed if a different size is defined.

3.12 Inline Operation Functions

3.12.1 Absolute difference (ix_asub)

Synopsis

```
#include <stdimap.h>
inline sep uchar Ix_asub(sep uchar p1,sep uchar p2);// For compatibility with IMAPCAR
inline sep uint  Ix_asub_u2(sep uint p1, sep uint p2); // For a PE array
inline sep uint  Ix_asub_s2(sep int p1, sep int p2); // For a PE array
inline uint  ix_asub_u2(uint p1, uint p2); // For a CP/PU
inline uint  ix_asub_s2(int p1, int p2); // For a CP/PU
```

Description These functions return the result of |p1 - p2|.

3.12.2 Addition of an absolute difference to an accumulator (ix_acc_asub)

Synopsis

```
#include <stdimap.h>
inline void Ix_acc_asub_u2(sep uint p1, sep uint p2); // For a PE array
inline void Ix_acc_asub_s2(sep int p1, sep int p2); // For a PE array
inline void ix_acc_asub_u2(uint p1, uint p2); // For a CP/PU
inline void ix_acc_asub_s2(int p1, int p2); // For a CP/PU
```

Description These functions add the result of |p1 - p2| to the ra10 or cra10 accumulation register.

3.12.3 Addition of a product to an accumulator (ix_acc_mul)

Synopsis

```
#include <stdimap.h>
inline void Ix_acc_mul8(sep uchar p1, sep uchar p2); // For compatibility with IMAPCAR
inline void Ix_acc_mul_u2_u4(sep uint p1, sep uint p2); // For a PE array
inline void Ix_acc_mul_s2_s4(sep int p1, sep int p2); // For a PE array
inline void ix_acc_mul_u2_u4(uint p1, uint p2); // For a CP/PU
inline void ix_acc_mul_s2_s4(int p1, int p2); // For a CP/PU

inline void Ix_acc_mul_u2_u8(sep uint p1, sep uint p2); // For a PE array
inline void Ix_acc_mul_s2_s8(sep int p1, sep int p2); // For a PE array
inline void ix_acc_mul_u2_u8(uint p1, uint p2); // For a CP/PU
inline void ix_acc_mul_s2_s8(int p1, int p2);   // For a CP/PU
```

Description These functions either add the 16 lower bits of the product of p1 x p2 to the 32-bit accumulation register ra10 or cra10 (if the function name ends with a *4*), or add the 32-bit product of p1 x p2 to the 64-bit accumulation register ra3210 or cra3210 (if the function name ends with an *8*, except for Ix_acc_mul8).

Note that the accumulation register for the Ix_acc_mul8 function, has 24 bits in IMAPCAR but is expanded to 32 bits in the XC core, which is to maintain compatibility with IMAPCAR. Therefore, unlike IMAPCAR, when 24 bits are exceeded in the XC core, up to 32 bits can be used for accumulation.

3.12.4 Reading the result of adding to an accumulator (ix_acc_read)

Synopsis

```
#include <stdimap.h>
inline sep ulong      Ix_acc_read(); // For compatibility with IMAPCAR
inline sep ulong      Ix_acc_read_u4();// For a PE array
inline sep ulong long Ix_acc_read_u8();// For a PE array

inline ulong          ix_acc_read_u4();// For a CP/PU
inline ulong long     ix_acc_read_u8();// For a CP/PU
```

Description Ix_acc_read and Ix_acc_read_u4 return the value of the 32-bit accumulation register ra10 for each PE and simultaneously clear the register. Similarly, Ix_acc_read_u8 returns the value of the 64-bit accumulation register ra3210 for each PE and simultaneously clears the register.

On the other side, ix_acc_read_u4 returns the value of the 32-bit accumulation register cra10 for each CP or PU and simultaneously clears the register. Similarly, ix_acc_read_u8 returns the value of the 64-bit accumulation register cra3210 for each CP or PU and simultaneously clears the register.

3.12.5 Adding to an accumulator (ix_acc)

Synopsis

```
#include <stdimap.h>
inline void Ix_acc_s2_s4(sep int p1); // For a PE array
inline void Ix_acc_s4_s8(sep long p1); // For a PE array
inline void ix_acc_s2_s4(int p1); // For a CP/PU
inline void ix_acc_s4_s8(long p1); // For a CP/PU
```

Description These functions add a defined 16 or 32-bit value to the ra10, cra10, ra3210, or cra3210 accumulation register.

3.12.6 Saturation arithmetic (ix_sadd, ix_ssub)

Synopsis

```
#include <stdimap.h>
inline sep uchar Ix_sadd(sep uchar p1, sep uchar p2);          // Saturated value: 255, for
                                                               //compatibility with IMAPCAR
inline sep uint Ix_sadd_u2_255(sep uint p1, sep uint p2);      // Saturated value: 255, for
                                                               //a PE array
inline sep uint Ix_sadd_u2_65535(sep uint p1, sep uint p2);    // Saturated value: 65535, for
                                                               //a PE array
inline uint   ix_sadd_u2_255(uint p1, uint p2);                // Saturated value: 255, for a CP/PU
inline uint   ix_sadd_u2_65535(uint p1, uint p2);              // Saturated value: 65535, for a CP/PU

inline sep uchar Ix_ssub(sep uchar p1,sep uchar p2);           //Saturated value: 0, for
                                                               //compatibility with IMAPCAR
inline sep uint Ix_ssub_u2(sep uint p1, sep uint p2);          // Saturated value: 0, for a PE array
inline uint ix_ssub_u2(uint p1, uint p2);                      // Saturated value: 0, for a CP/PU
```

Description The ix_sadd functions return either the result of p1 + p2, or the saturated value if the result is greater than that value.

The ix_ssub functions return either the result of p1 - p2, or the saturated value if the result is less than that value.

3.12.7 Maximization and minimization operations (ix_max, ix_min)

Synopsis

```
#include <stdimap.h>
inline sep uchar   Ix_max3(sep uchar p1,sep uchar p2,sep uchar p3);   // For compatibility with
                                                                      //IMAPCAR
inline sep uint    Ix_max_u2(sep uint p1,sep uint p2);                // For a PE array
inline uint        ix_max_u2(uint p1,uint p2);                        // For a CP/PU

inline sep uchar   Ix_min3(sep uchar p1,sep uchar p2,sep uchar p3);   // For a PE array
inline sep uint    Ix_min_u2(sep uint p1,sep uint p2);                // For a PE array
inline uint        ix_min_u2(uint p1,uint p2);                        // For a CP/PU
```

Description These functions return the maximum or minimum of p1, p2 (and p3, if applicable).

3.12.8 Bit manipulation operations (ix_bitr, ix_bts, ix_skz)

Synopsis

```
#include <stdimap.h>
inline sep uchar    Ix_bitr(sep uchar a, sep uchar b);          //For compatibility with IMAPCAR
inline sep uint     Ix_bitr_u2(sep uint a, sep uint b);         // For a PE array
inline uint   ix_bitr_u2(uint a, uint b); // For a CP/PU

inline sep uint     Ix_bts_u2(sep uint a, sep uint b, sep uint c); // For a PE array
inline uint         ix_bts_u2(uint a, uint b, uint c);          // For a CP/PU

inline sep uint     Ix_skz_u2(sep uint a);                      // For a PE array
inline uint         ix_skz_u2(uint a);                          // For a CP/PU
```

Description ix_bitr: these functions return the value of a at the bit position defined by b&15.

ix_bts: these functions assigns bit "7 – (c&7)" of b to the LSB of (a << 1) and return the result.

ix_skz: these functions return the number of 0s in a up to the first bit position where there is a 1, starting with the MSB.

3.12.9 Packing operations (Ix_pack)

Synopsis

```
#include <stdimap.h>
inline sep uint Ix_pack (sep uchar p1, sep uchar p2, sep uchar p3); // For compatibility with IMAPCAR
inline sep uint  Ix_pack_u2(sep uint p1, sep uint p2, sep uint p3);    // For a PE array
```

Description These functions check for 0s in the total of 9 words stored in the p1, p2, and p3 of the PE on the left, the p1, p2, and p3 of the defined PE, and the p1, p2, and p3 of the PE on the right, and then store 0s in the 8 lower bits of the return value that correspond to locations where 0s were found, or 1s in the 8 lower bits that correspond to locations where non-0 values were found. The 8 higher bits of the return value are cleared.

Bit position in return value	Value checked for 0
7	Left p1
6	Defined p1
5	Right p1
4	Left p2
3	Right p2
2	Left p3
1	Defined p3
0	Right p3

3.13 Inline functions to transfer data between PEs

3.13.1 Ix_mvlc, Ix_mvrc, Ix_mvlc_u4, Ix_mvrc_u4

Synopsis

```
#include <stdimap.h>
inline sep uint   Ix_mvlc(sep uint a, sep uint b, const uint dist);
inline sep uint   Ix_mvrc(sep uint a, sep uint b, const uint dist);
inline sep ulong  Ix_mvlc_u4(sep ulong a, sep ulong b, const uint dist);
inline sep ulong  Ix_mvrc_u4(sep ulong a, sep ulong b, const uint dist);
```

Description Ix_mvlc specifies that the PE elements, that are dist to the right of sep data (twice the length of the number of PEs, obtained by connecting b to the right of a), are used as the PE elements of the sep data, and then returns the obtained sep data.

Ix_mvrc specifies that the PE elements, that are dist to the left of sep data (twice the length of the number of PEs, obtained by connecting b to the left of a), are used as the PE elements of the sep data, and then returns the obtained sep data.

Except for the fact that they accept sep ulong data for a and b and return sep ulong data, Ix_mvlc_u4 and Ix_mvrc_u4 perform the same operations as Ix_mvlc and Ix_mvrc.

3.13.2 Ix_mvlz, Ix_mvrz, Ix_mvlz_u4, Ix_mvrz_u4

Synopsis

```
#include <stdimap.h>
inline sep uint   Ix_mvlz(sep uint a, const uint dist);
inline sep uint   Ix_mvrz(sep uint a, const uint dist);
inline sep ulong  Ix_mvlz_u4(sep ulong a, const uint dist);
inline sep ulong  Ix_mvrz_u4 (sep ulong a, const uint dist);
```

Description Ix_mvlz specifies that the PE elements, that are dist to the right of a, are used as the current PE elements, and then returns the obtained sep data.

Ix_mvrz specifies that the PE elements, that are dist to the left of a, are used as the current PE elements, and then returns the obtained sep data.

Except for the fact that they accept sep ulong data for a and returns sep ulong data, Ix_mvlz_u4 and Ix_mvrz_u4 performs the same operations as Ix_mvlz and Ix_mvrz.

3.14 Integer Square Root Functions

3.14.1 ixSqrt, ixSqrtl, ixSqrtll

Synopsis

```
#include <stdimap.h>
uint  ixSqrt(uint n);
uint  ixSqrtl(ulong n);
ulong ixSqrtll(ulong long n);
```

Description These functions return the square root of the integer `n`. The decimal portion is truncated.

3.14.2 lxSqrtsep, lxSqrtsepl, lxSqrtsepll

Synopsis

```
#include <stdimap.h>
separate uint  lxSqrtsep(separate uint n);
separate uint  lxSqrtsepl(separate ulong n);
separate ulong lxSqrtsepll(separate ulong long n);
```

Description These functions return the square root of the sep integer `n`. The decimal portion is truncated.

3.15 Functions of obtaining the number of PEs and PE numbers

3.15.1 Ix_max_total_pe_number

Ix_max_total_pe_number() returns the PEMAX constant, which is defined as a compiler option, as the maximum number of usable PEs. Note that the value returned by Ix_max_total_pe_number() is as follows if the PEMAX compiler option is not defined, and is always a constant.

If PEMAX is defined when compiling	If PEMAX is not defined when compiling		
	If a non-zero value is defined for PENO when compiling	If zero is defined for PENO when compiling	If PENO is not defined when compiling
Constant (PEMAX)	Constant (PENO)	Constant (128)	Constant (128)

3.15.2 Ix_pe_id

Ix_pe_id() returns the ID number for the PE in which processing is executing.

This macro is defined in stdimap.h as follows:

```
#define Ix_pe_id()  (_PENUM)
```

_PENUM is a global sep constant defined as standard in IMEM, and the value is corrected when Ix_start_mode is used to switch the operating mode.

3.15.3 Ix_real_pe_number

Ix_real_pe_number() returns the number of usable PEs when it is called.

Because this macro is defined in stdimap.h as follows, the return value differs according to the operating mode:

```
#define Ix_real_pe_number()     (ix_spec_read(SPEC_MPASN)==0 ? PENO : (PENO/2))
```

3.15.4 Ix_total_pe_number

Ix_total_pe_number() returns the number of physical PEs that do not depend on the operating mode.

This macro is defined in stdimap.h as follows:

```
#define Ix_total_pe_number()     (PENO)
```

Be careful when using Ix_total_pe_number() (PENO) to declare the size of an array, because the value (including its attribute, that is, whether it is a constant) differs depending on the values specified for the PENO and PEMAX compiler options as shown in the following table.

If zero is defined for PENO or PENO is not defined when compiling	If a non-zero value is defined for PENO when compiling
If zero is defined for PENO or PENO is not defined when compiling	If a non-zero value is defined for PENO when compiling
Determined at execution (a variable)	Constant (equal to the value defined for PENO when compiling)

3.15.5 PEMAX, PEMIN

The PEMAX constant indicates the maximum number of PEs, and the PEMIN constant indicates the minimum number of PEs. Both are defined in stddef.h as follows:

```
#ifndef PEMAX
#define PEMAX 128
#endif
#ifndef PEMIN
#define PEMIN 16
#endif
```

3.15.6 PENO

PENO is either a constant, if the number of PEs is defined when compiling, or a variable, if the number of PEs is not defined, that indicates the number of PEs

3.15.7 PENUM, _PENUM

PENUM and _PENUM both indicate the ID number for each PE, but, while _PENUM represents the ID number for each PE even in the MIXED mode, PENUM is handled as a constant and does not represent the correct PE ID number unless in the SIMD mode.

However, unlike _PENUM, if PENUM is used for an operation such as the following, constant folding, an optimization technique, is used to compile the result of the addition instead of performing the addition at run-time:

PENUM + *constant*

Therefore, using this constant can generate code that runs more quickly.

3.16 Image Processing Functions

3.16.1 IxPk8tlup

Synopsis

```
#include <stdimap.h>
void IxPk8tlup(sep uchar* src,
        sep uchar *tbl,
        uchar hitmask,
        uchar missmask,
        uint oddeven,
        uint lines)
```

Description If oddeven is 0, this function calculates a 1 byte value *X* by examining each data position in the lines lines of 2-face data that starts at the IMEM address indicated by the sep array src.

For the 8 pixels adjacent to each data position, if the value of the pixel is not 0, this function defines a 1 for the corresponding bit of the byte *X* (shown in the table below), and, if the value of the pixel is 0, this function defines a 0 for the corresponding bit of *X*.

Next, this function uses *X* as an index to reference the sep array tbl. It finds the logical conjunction of hitmask and each bit stored in src if the reference result is not 0, or the logical conjunction of missmask and each bit stored in src if the reference result is 0. Then it overwrites the value of src.

Bit position in X	Position of pixel checked for 0
7	Upper left
6	Upper
5	Upper right
4	Left
3	Right
2	Lower left
1	Lower
0	Lower right

Remark: here, *upper* refers to the direction in which IMEM addresses decrease from src, and *lower* refers to the direction in which IMEM addresses increase from src.

If oddeven is not 0, IxPk8tlup first performs the above processing on PEs that have odd numbers and then performs the processing on PEs that have even numbers.

This function is written in assembly language but performs processing equivalent to the following 1DC code:

```
#define proc()(*src=(*src)&(*(tbl+(Ix_pack(*(src-1),*(src),*(src+1))))? hitmask:missmask))
  if (!oddeven)
    while(lines--) {proc();src++;}
  else
    while(lines--) {
      mif(_PENUM&1) proc();
      melse proc();
      src++;
    }
```

3.16.2 IxVote_u1u2

Synopsis

```
#include <stdimap.h>
uint IxVote_u1u2 (sep uchar* src,
                  sep uint *wrk,
                  uint lines,
                  uint off)
```

Description Each PE uses the values of lines lines of elements in the sep array src as indexes into the wrk array. Indexed elements of the wrk array are incremented.

If off is 1, the processing is performed for all the lines in src, but, if off is greater than 1, off – 1 lines are skipped each time an element of wrk is incremented.

This function performs processing equivalent to the following 1DC code:

```
int i;
for(i=0; i<lines;i+=off) wrk[src[i]]++; // column-wise voting
return 0;
```

This function always returns 0.

3.16.3 IxVoteCollect_u2u4

Synopsis

```
#include <stdimap.h>
int  IxVoteCollect_u2u4 (sep uint *wrk,
                         sep ulong *hist,
                         uint levels)
```

Description This function totals levels lines of data in the sep array wrk. Next, assuming PEN0 is the actual number of PEs, it stores the resulting elements on line *X*, *X* + PEN0, *X* + PEN0 × 2 and so on for PEs that have numbers in the range *X* (where *X* is 0 to PEN0 – 1) as the first, second, third, and subsequent elements of the hist array. Therefore, hist must be a pointer to a ulong area greater or equal to (levels + PEN0 – 1)/PEN0.

This function always returns 0.

3.17 Interrupt functions

3.17.1 ix_idis, ix_iena

Synopsis

```
#include <stdimap.h>
int     ix_idis();
int     ix_iena();
```

Description ix_idis prohibits subsequent interrupts.

ix_iena cancels the specification of prohibited interrupts by the most recent call to the ix_idis function.

In addition to prohibition of subsequent interrupts, ix_idis increments the number of nested levels of prohibited interrupts. This function returns the current nested level for which interrupts are prohibited.

ix_iena decrements the number of nested levels of prohibited interrupts. If this level reaches 0, interrupts are allowed. This function returns the current nested level for which interrupts are prohibited. If the value is 0 or less, it means that interrupts are allowed by the ix_iena function.

3.17.2 ix_interrupt_return

Synopsis

```
#include <stdimap.h>
void  ix_interrupt_return();
```

Description The ix_interrupt_return function enables the subsequent execution of other interrupt functions. If an interrupt function is executing, other interrupts are automatically prohibited.

Calling ix_interrupt_return once from within an interrupt function cancels the interrupt prohibited status and enables the reception of other interrupts. If ix_interrupt_return is not called, however, interrupts are not allowed even after processing returns from the current interrupt function.

Therefore, this function must normally be called once at an appropriate position in interrupt functions, such as at the end.

Note that the pragma statements below are normally used when defining an interrupt function in a 1DC program. For example, to use the myInterrupt_function function as an interrupt routine, define the following:

```
#pragma Kinterrupt=1
void myInterrupt_function()
{
    int a=1;
    otherfunc(a);
    ...
}
#pragma Kinterrupt=0
```

If the above code is used, ix_interrupt_return does not normally have to be explicitly called because CP registers are backed up at the beginning of interrupt functions, then restored at the end of the functions, and calls to ix_interrupt_return are automatically generated by the compiler.

However, to enable the reception of interrupts before the end of a function such as myInterrupt_function, ix_interrupt_return has to be explicitly called.

3.17.3 ix_push_creg_all, ix_pop_creg_all

Synopsis

```
#include <stdimap.h>
void  ix_push_creg_all();
void  ix_pop_creg_all();
```

Description

ix_push_creg_all backs up the CP register values below to the stack.

ix_pop_creg_all restores these values from the stack.

cr0,cr1,...,cr23,

ctmp0,ctmp1,cmd,cmh,cmuld,cmulh,

cflgpenum,cra01,cra23,crt01,

wst,wpara

Note that, if the following pragma statements are used, a call to ix_push_creg_all is generated at the beginning of myInterrupt_function, and a call to ix_pop_creg_all is generated at the end:

```
#pragma Kinterrupt=1
void myInterrupt_function()
{
  ...
}
#pragma Kinterrupt=0
```

3.17.4 Ix_push_ireg_all, Ix_pop_ireg_all

Synopsis

```
#include <stdimap.h>
void Ix_push_ireg_all();
void Ix_pop_ireg_all();
```

Description

Ix_push_ireg_all backs up the PE register values below to the stack.

Ix_pop_ireg_all restores these values from the stack.

r0,r1,r2,...,r14,

md,mh,muld,mulh,tmp0,tmp1,flgpenum,

ra01,ra23,rt01,rt23

Note that the value of the IMEM stack pointer changes when calling the Ix_push_ireg_all or Ix_pop_ireg_all, because the IMEM stack is used as the backup location for register values. Therefore, functions from which Ix_push_ireg_all or Ix_pop_ireg_all is called must not use the IMEM stack.

In particular, in an interrupt function that uses a PE array to perform arithmetic operations, back up the PE registers by calling Ix_push_ireg_all immediately after the processing starts, and restore the PE register values by calling Ix_pop_ireg_all immediately before the processing finishes and processing returns from the interrupt. Note that Ix_push_ireg_all and Ix_pop_ireg_all do not have to be called in an interrupt function that does not use a PE array to perform arithmetic operations.

3.18 Other functions

3.18.1 ix_halt

Synopsis

```
#include <stdimap.h>
inline void     ix_halt(uint completion_code);
```

Description The ix_halt function halts execution.

If this function is called from a PU, the LSB of completion_code is assigned to the value of the bit at the position corresponding to that PU in the special CP register mpjoin. To restart a PU stopped using this function, an operation using the host or CP is required, and, to restart a CP stopped using this function, an operation using the host is required.

For example, when ix_halt(1) is called to stop a PU, the corresponding bit in the special register mpjoin on the CP is set, and the PU is then regarded as stopped. After that, unless the host or CP is used to clear that mpjoin bit, the PU does not run even if mprun or mpstep is modified using the host or CP. In contrast, calling ix_halt(0) only stops a PU, and the PU can be started by writing to mprun or mpstep using the host or CP.

In contrast to the above, if a (program on a) CP calls ix_halt(0) or ix_halt(1), the operation is the same, and the CP runs if run or step is modified using the host.

3.18.2 ix_exec

Synopsis

```
#include <stdio.h>
int ix_exec(char *str);
```

Description This function executes the debugger command defined for str. str must be a character string for a command supported by the debugger. The return value is 0 if the defined command executed successfully, or a non-0 value if the command terminated abnormally.

Examples:

```
ix_exec("timerint on\n");              // Use a debugger command to enable the interrupt timer.
ix_exec("timerint 6650 15\n");         // Specify the interval and the number of the interrupt vector // to use for the interrupt timer.
ix_exec("write iv15 'Fint_func\n"); // Specify the interrupt function.


char cmd[128];
#define LINES 480L
#define WIDTH 640L
outside separate unsigned char ein[LINES*WIDTH/PENO]
sprintf(cmd,"loadpeblk “test.pgm” %lu %u w %u\n",fname,ein,LINES,WIDTH);
ix_exec(cmd);  // Read an image from the hard disk to an outside sep array.
```

3.18.3 ix_exec2

Synopsis

```
#include <stdio.h>
int ix_exec2(char *str, void* buff, int size);
```

Description

This function executes the debugger command defined for str.

buff stores size bytes of the data that results from executing the defined command, which differs depending on the type of command.

str must be a character string for a command supported by the debugger.

The return value is 0 if the defined command executed successfully, or a non-zero value if the command terminated abnormally.

Examples:

```
char cmd[64];
char fname[32] = "myimagefile";
osep uchar *ein=(osep uchar*)Ix_eheap((MAXWIDTH/PENO)*MAXLINES*sizeof(char));
ulong wl;
sprintf(cmd,"loadpeblk %s.pgm %lu\n",fname,ein);
ix_exec2(cmd,&wl,4); // execute debugger/simulator command
printf("wl=%u,loaded image(width x lines) = (%u x %u) \n",wl,wl&0xffff,wl>>16);
```

The following table lists the debugger commands that return result data and provides information about the format of this data:

Command name	Maximum assignable size	buff format	Remark
Loadpi Loadpiblk Loadpe Loadpeblk	4	First 2 bytes: width Last 2 bytes: length	For details, see the following structure, which is defined in stdio.h: ix_exec2_loadpe_result
Loadpirgb Loadpergb Loadpeblkrgb	10	First 2 bytes: width Next 2 bytes: length Next 2 bytes: addr_r Next 2 bytes: addr_g Last 2 bytes: addr_b	For details see the following structure which is defined in stdio.h: ix_exec2_loadpergb_result
Liadpiyc Loadpeyc Loadpeblkyc	8	First 2 bytes: width Next 2 bytes: length Next 2 bytes: addr_y Last 2 bytes: addr_c	For details, see the following structure, which is defined in stdio.h: ix_exec2_loadpeyc_result

3.18.4 lx_in, lx_ou

Synopsis

```
#include <stdimap.h>
inline ulong    lx_in(uint portno);
inline void     lx_ou(uint addr, ulong data, uint wmask);
```

Description lx_ou writes data to the port address addr. The 4 lower bits of wmask can be used to define whether each byte of data is written (by defining 0 to write the data, else 1).

The lx_in function returns the data read from the port address addr.

3.18.5 ix_nopwait

Synopsis

```
#include <stdimap.h>
void ix_nopwait();
```

Description This function triggers a determined number of cycles during which no operations are performed. This function is not moved even if compiler optimization is performed.

This function is intended to be defined in a loop that must wait, such as for an external interrupt or message interrupt. Interrupts are prohibited while an empty loop is repeated, but this function can be added to the body of a loop to trigger cycles during which no operations are performed (and during which interrupts are allowed).

3.18.6 ix_spec_read, ix_spec_write, lx_spec_read, lx_spec_write

Synopsis

```
#include <stdimap.h>
inline ulong            ix_spec_read(uint spreg);
inline sep ulong        lx_spec_read(sep uint spreg);
inline void ix_spec_write(uint spreg, ulong v);
inline void lx_spec_write(sep uint spreg, sep ulong v);
```

Description These functions read from (or write to) the special register defined for spreg, which is one of the character strings below defined in:
`1DC/cc1dc/lib/include/inline.h`

If reading is performed, the read value is returned.

```
#define SPEC_RA01          0
#define SPEC_RA23          1
#define SPEC_RT01          2
#define SPEC_RT23          3
#define SPEC_FLGPENUM      4
#define SPEC_RT23A         5
#define SPEC_RT23B         6
#define SPEC_RT23C         7
#define SPEC_PC            8
#define SPEC_WST           9
#define SPEC_WPARAM        10
#define SPEC_DSTKLM        11
#define SPEC_DSTKP         12
#define SPEC_HP            13
#define SPEC_STATUS        14
#define SPEC_ESTKP         16
#define SPEC_EHP           17
#define SPEC_BER0          18
#define SPEC_BER1          19
#define SPEC_MPAREA0       20
#define SPEC_MPAREA1       21
#define SPEC_MPAREA2       22
#define SPEC_MPAREA3       23
#define SPEC_MPRUN         24
#define SPEC_MPBREAK       25
#define SPEC_MPSTEP        26
#define SPEC_MPRESET       27
#define SPEC_MPJOIN        28
#define SPEC_MPASN         29
#define SPEC_MPERR         30
#define SPEC_ISTKPHP       31
```

3.18.7 IxPtime

Synopsis

```
#include <stdimap.h>
ulong IxPtime();
```

Description This function returns how many clock cycles have elapsed since the last time this function was called. The return value is always 0 when this function is executed in a PU.

3.18.8 IxPtimeAcc

Synopsis

```
#include <stdimap.h>
ulong IxPtimeAcc(char *msg, int CLK);
```

Description This function returns how much processing time has elapsed (in μs) since the first time IxPtime or IxPtimeAcc was called. This processing time is calculated using the value of the clock counter for the computer and the CLK parameter, which is in MHz. The return value is always 0 when this function is executed in a PU.

If a pointer to a character string other than NULL is assigned to msg, that character string, the returned processing time, and the processing time that has elapsed since the last time IxPtime or IxPtimeAcc was called are displayed using printf.

Usage example:

```
main()
{
  IxPtime();
       funcA();
  IxPtimeAcc("After calling funcA()",133); // To display the processing time
       funcB();
  IxPtimeAcc("After calling funcB()",133); // To display the processing time
       funcC();
  printf("Total processing time: %lu us\n",IxPtimeAcc(NULL,133));//To obtain the //processing time
}
```

3.19 Imported Standard C Functions

The table below shows standard C functions that have been imported into the standard function library described in this document. Unless otherwise specified, the source files (respectively header files) for the standard C functions are stored in `1DC\imap\imap\src\lib1dc` (respectively `1DC\cc1dc\lib\include`).

Function name	Functional classification	Source file	Header file
exit	Closing programs	exit.lc	stdlib.h
free	Dynamic memory allocation	ix_malloc.lc	stdlib.h
malloc	Dynamic memory allocation	ix_malloc.lc	stdlib.h
rand	Random number generation	rand.c	stdlib.h
realloc	Dynamic memory allocation	ix_malloc.lc	stdlib.h
srand	Random number generation (changes the seed to change the series of randomly generated number)	rand.c	stdlib.h
fclose	File manipulation	stdio.lc	stdio.h
feof	File manipulation	stdio.lc	stdio.h
ferror	File manipulation	stdio.lc	stdio.h
fflush	File manipulation	stdio.lc	stdio.h
fgetc	File manipulation	stdio.lc	stdio.h
fgetpos	File manipulation	stdio.lc	stdio.h
fgets	File manipulation	stdio.lc	stdio.h
fopen	File manipulation	stdio.lc	stdio.h
fprintf	File manipulation	stdio.lc	stdio.h
fputc	File manipulation	stdio.lc	stdio.h
fputs	File manipulation	stdio.lc	stdio.h
freopen	File manipulation	stdio.lc	stdio.h
fscanf	File manipulation	stdio.lc	stdio.h
fseek	File manipulation	stdio.lc	stdio.h
fsetpos	File manipulation	stdio.lc	stdio.h
ftell	File manipulation	stdio.lc	stdio.h
fwrite	File manipulation	stdio.lc	stdio.h
printf	Standard I/O (formatted character strings)	stdio.lc	stdio.h
putchar	Standard I/O (characters)	stdio.lc	stdio.h
puts	Standart I/O (character strings)	stdio.lc	stdio.h
sprintf	Outputs a formatted character string	stdio.lc	stdio.h
memcpy	Copies memory	ix_memcpy.lc	string.h
memset	Clears memory	ix_memset.lc	string.h
strcmp	Compares character strings	string.c	string.h

strlen	Returns the length of a character string	string.c	string.h

3.20 Imported C Functions for Mathematical Operations

A list of C functions for mathematical operations that have been imported into the library described in this document is provided below. Unless otherwise specified, the source files for these functions are stored in `1DC\imap\imap\src\libm`, and the header files for these functions are stored in `1DC\cc1dc\lib\include`.

```
/*math.h */
#ifndef _MATH_H
#define _MATH_H

void sincos(double,double *,double *);
double sin(double);
double cos(double);
double tan(double);
double atan(double);
double asin(double);
double acos(double);

void sincosf(float,float *,float *);
float sinf(float);
float cosf(float);
float tanf(float);
float atanf(float);
float asinf(float);
float acosf(float);

void lx_sincos(separate double,separate double *,separate double *);
separate double lx_sin(separate double);
separate double lx_cos(separate double);
separate double lx_tan(separate double);
separate double lx_atan(separate double);
separate double lx_asin(separate double);
separate double lx_acos(separate double);
```

```
void lx_sincosf(separate float,separate float *,separate float *);
separate float lx_sinf(separate float);
separate float lx_cosf(separate float);
separate float lx_tanf(separate float);
separate float lx_atanf(separate float);
separate float lx_asinf(separate float);
separate float lx_acosf(separate float);

double sinhcosh(double,double *,double *);
double sinh(double);
double cosh(double);
double tanh(double);
double exp(double);
double log(double);
double atanh(double);
double sqrt(double);
double asinh(double);
double acosh(double);

float sinhcoshf(float,float *,float *);
float sinhf(float);
float coshf(float);
float tanhf(float);
float expf(float);
float atanhf(float);
float logf(float);
float sqrtf(float);
float asinhf(float);
float acoshf(float);
```

```
separate double lx_sinhcosh(separate double,separate double *,separate double *);
separate double lx_sinh(separate double);
separate double lx_cosh(separate double);
separate double lx_tanh(separate double);
separate double lx_exp(separate double);
separate double lx_atanh(separate double);
separate double lx_log(separate double);
separate double lx_sqrt(separate double);
separate double lx_asinh(separate double);
separate double lx_acosh(separate double);

separate float lx_sinhcoshf(separate float,separate float *,separate float *);
separate float lx_sinhf(separate float);
separate float lx_coshf(separate float);
separate float lx_tanhf(separate float);
separate float lx_expf(separate float);
separate float lx_atanhf(separate float);
separate float lx_logf(separate float);
separate float lx_sqrtf(separate float);
separate float lx_asinhf(separate float);
separate float lx_acoshf(separate float);

double atan2(double,double);
double ceil(double);
double fabs(double);
double frexp(double,int*);
double floor(double);
double fmod(double,double);
double log10(double);
double modf(double,double*);
double pow(double,double);
```

```
float atan2f(float,float);
float ceilf(float);
float fabsf(float);
float frexpf(float,int*);
float floorf(float);
float fmodf(float,float);
float log10f(float);
float modff(float,float*);
float powf(float,float);

separate double lx_fmod(separate double,separate double);
separate double lx_atan2(separate double,separate double);
separate double lx_ceil(separate double);
separate double lx_fabs(separate double);
separate double lx_frexp(separate double,separate int*);
separate double lx_floor(separate double);
separate double lx_fmod(separate double,separate double);
separate double lx_log10(separate double);
separate double lx_modf(separate double,separate double*);
separate double lx_pow(separate double,separate double);

separate float lx_fmodf(separate float,separate float);
separate float lx_atan2f(separate float,separate float);
separate float lx_ceilf(separate float);
separate float lx_fabsf(separate float);
separate float lx_frexpf(separate float,separate int*);
separate float lx_floorf(separate float);
separate float lx_fmodf(separate float,separate float);
separate float lx_log10f(separate float);
separate float lx_modff(separate float,separate float*);
separate float lx_powf(separate float,separate float);

#endif /* _MATH_H */
```

4 Performance

The tables below list information about the performance of basic mathematical operations and certain standard 1DC functions (including the required number of cycles and how much more efficient the XC core is than IMAPCAR). In these tables, performance values that are at least twice as large as those of IMAPCAR are shown in bold.

4.1 Mathematical Operations

CP operations:

Operation	Number of cycles	XC core (128 PEs) versus IMAPCAR (cycle base)	Remarks
16-/32-/64-bit add/sub	1 / 1 / 2	1.0 / 1.0 / 1.0	-
16-/32-/64-bit multiply	1 / 4 / 16	1.0 / 1.0 / 1.0	-
16-/32-/64-bit div/mod Note	54 / 104 / 371	2.3 / 3.0 / 9.0	-
signed/unsigned 4-bit shift	1 / 1	1.0	-
float add/sub	1	2980	-
float multiply	1	26123	-
float division	26	2103	-
double add/sub	339	9.1	-
double multiply	383	41.5	-
double division	557	75.6	-

PE array operations :

Operation	Number of cycles	XC core (128 PEs) versus IMAPCAR (cycle base)	Remarks
16-/32-/64-bit add/sub	1 / 2 / 4	2.0 / 2.0 / 2.0	-
16-/32-/64-bit multiply	1 / 4 / 16	4.0 / 4.0 / 6.7	-
16-/32-/64-bit div/mod Note	54 / 104 / 371	1.2 / 2.0 / 10.9	-
signed/unsigned 4-bit shift	1 / 1	8.0 / 4.0	-
float add/sub	282	12.7	The number of cycles is proportional to the number of PEs because the CP performs mathematical operations in place of each PE
float multiply	311	40.0	
float division	367	148.8	
double add/sub	410	12.2	-
double multiply	478	29.7	-
double division	689	72.5	-

Note:

The number of required cycles for integer division (and remainders) changes depending on the size of the numerator, and the values in the table indicate the required number of cycles when the numerator is the maximum value.

4.2 Sep Data Manipulation Functions

Function name	Number of cycles	XC core (128 PEs) versus IMAPCAR (cycle base)	Remark and used IMAPCAR lib1dc function name
IxAddsep	49	21.5	addsepi()
IxAddsepl	89	•	
IxCountsep Note	171	13.5	countsep()
IxGathersep Note	171	7.5	gathersep()
IxGetleftsep	7	3.1	getleftsep()
IxGetrightsep	7	3.1	getrightsep()
IxMaxsep	39	9.1	maxsepi()
IxMinsep	39	10.0	minsepi()
IxPack8	8	2.0	pack8()
IxPack8i	8	•	•
IxPack8s	44	1.6	pack8s()
IxPack8si	42	1.9	pack8si()

Note: This is the number of cycles if there is valid data in all PEs. The actual required number of cycles is somewhat proportional to how many valid data items there are in the PE array.

4.3 Functions to transfer data between sep-type data and scalar data

Function name	Number of cycles	XC core (128 PEs) versus IMAPCAR (cycle base)	Remark and used IMAPCAR lib1dc function name
IxDmem2imem	138	14.2	1DC description used
IxImem2dmem	190	7.6	1DC description used
IxDmem2PE	324	5.5	1DC description used

4.4 Function to transpose sep-type data

The following comparisons were made based on how many cycles were necessary to transpose a 128 × 127 byte area of data in IMEM using each function.

Function name	Number of cycles	XC core (128 PEs) versus IMAPCAR (cycle base)	Remark and used IMAPCAR lib1dc function name
IxRot90	1434	2.5	rot90
		4.7	rot90i
		8.5	rot90l