# SSS-PC プログラマーズ ガイド第1.0版

株式会社情報科学研究所
http://www.isll.co.jp/

2003年12月1日

# 目 次

# 1 MBCF 通信同期機構

## 1.1 概要

本節では *SSS–PC* に実装された *MBCF(*メモリベース通信同期機構*)* について述べる。メモリベース通信同期機構は操作対象タスクの操作対象論理アドレス上のデータを直接操作する通信同期手法である。通信機構が直接対象タスクのメモリを操作するように実装されるため、カーネル空間内の無駄なコピーが発生せず、また操作対象のデータをアドレスで指定できるため、操作内容自体にもバラエティを加えることができる点が特徴である。

*SSS–PC* の *MBCF* は以下のように実装されている。

- ユーザが *MBCF* 要求を行うことにより、*MBCF* パケットが対象ノードに送信される。
- *MBCF* パケットを受信したノードの *MBCF* 通信同期機構によって操作対象タスクの操作対象アドレスに対して指定された操作を行う。

## 1.2 MBCF パケットフォーマット

### 1.2.1 Ethernet Header

| 31 24 | 23 16 | 15 8 | 7 0 |
|---|---|---|---|
| destination address[0–3] | | | |
| source address[0–1] | | destination address[4–5] | |
| source address[2–4] | | | |
| length | | protocol (0x6666) | |

図 1: Eathernet Header for MBCF

**destination address** *dest*

**source address** *source*

**protocol 0x6666** *for MBCF*

**length** *len*

### 1.2.2 MBCF Ether パケットフォーマット

| 31 24 | 23 16 | 15 8 | 7 0 |
|---|---|---|---|
| source node | | | |
| received sequence number | full flag | sequence number | status (unused?) |
| source task | | | |
| source task key | | | |
| destination task | | | |
| destination task key | | | |
| MBCF command code | | | |
| MBCF packet payload (command specific) | | | |

図 2: MBCF Ether パケットフォーマット

**source node** *pnid of sender node*

**status** *unused?*

**sequence number** *sequence number of the packet*

**full flag** *need ack ?*

**received sequence number** *sequence number of packet received at destination node.*

**source task** *physical task number of source task.*

**source task key** *access key of source task.*

**destination task** *physical task number of destination task.*

**destination task key** *access key of destination task.*

**MBCF command code** *command*

**MBCF packet payload** *command data*

### 1.2.3 MBCF パケットペイロード

**1.2.3.1 基本型** 基本的にはコマンドごとに最適なフォーマットを採用する方針である。ただし、コマンド毎にソースタスクから送信する情報が異なる、コマンドによってはパディングが挿入されるなど、微妙な変化がある。したがって以下のフォーマットに出現する項目は、全てのパケットに含まれるという意味ではない。が、パケット内の並び順などの方針はあらわしている。ちなみに、ペイロードは *Ether* ヘッダの先頭から *44* バイト目の位置 *(12* 個目の *32bit* ワード*)* から開始する。*data* フィールドはコマンドとオプションの組み合わせ *(*現状ではカーネルの `MBCF_USE_WRITE8` コンパイル*)* によっては *8, 64* バイトアラインするために *padding* が挿入されることがある。

以下では *pbox* 構造体 *(2.1.1* 節参照*)* のメンバ変数の名前を使って解説する。ただし *pad* は *pbox* 構造体には含まれない。なお、*addr2, addr3* にはコマンドによっては、メモリアドレス以外の情報が格納されることがある。

| 31 24 | 23 16 | 15 8 | 7 0 | |
|---|---|---|---|---|
| addr | | | | 44 |
| len | | | | 48 |
| addr2(xaddr) | | | | 52 |
| addr3(xdata) | | | | 56 |
| stat_addr | | | | 60 |
| data<br>... | | | | 64 |
| (pad) | | | | |

図 3: MBCF ペイロードフォーマット

**addr** 操作対象アドレス

**len** 操作対象データ長 *(1* パケットに乗せられるデータ量は最大で `MBCF_MAX_DATA_LEN` バイト*)*

**addr2** 追加アドレス *2*。現状以下で利用されている。

- *MBCF_WRITE_F* および *MBCF_WRITE_F_STAT* のフラグアドレス
- *MBCF_FETCH_ADD* および *MBCF_FETCH_ADD* の *fetch* データ書き戻しアドレス
- *MBCF_SIGNAL* での *MB_SIGNAL_MEMW_OFFSET* データ格納タイプの場合のオフセット

- *MBCF_ZBUF_PACK* および *MBCF_ZBUF_PACK24* の *Z* アドレス

**addr3** 追加アドレス *3*。現状以下で利用されている。データが入ることもある。

- *MBCF_WRITE_F* および *MBCF_WRITE_F_STAT* のフラグデータ

**stat_addr** *Stat* 付きコマンドのフラグ書き戻しアドレス。

**data** 送信データ。

**pad** 最後のワードは *4* バイト境界まで *padding* される。*MBCF_USE_WRITE8* でコンパイルされたカーネルは *8* バイト境界まで *padding* する。

#### 1.2.3.2 コマンド毎のペイロードフォーマット

**1.2.3.2.1 ペイロードフォーマット一覧** 表 *1* に、*MBCF* ペイロードのフォーマットを一覧する。図 *4* から図 *22* にて各フォーマットタイプを示す。フォーマットタイプは新しいコマンドが定義されると追加されることがある。

表 1: MBCF payload format 一覧

| | | pbox fields | | | | | |
|---|---|---|---|---|---|---|---|
| コマンド名 | タイプ | len | addr | addr2 | addr3 | data | 備考 |
| MBCF_NOP | 0 | | | | | | |
| MBCF_CLOSE | 0 | | | | | | |
| MBCF_TASK_END | 0 | | | | | | |
| MBCF_WTASKQ | 1 | ○ | | | | ○ | |
| MBCF_WRITE | 2 | ○ | ○ | | | ○ | |
| MBCF_MIGRATION | 2 | ○ | ○ | | | ○ | |
| MBCF_BLKWRITE | 2B | ○ | ○ | | | ○ | |
| MBCF_BLKWRITE_CTX | 2B | ○ | ○ | ○ | | ○ | |
| MBCF_WRITE_CTX | 2C | ○ | ○ | ○ | | ○ | |
| MBCF_READ | 3n | ○ | ○ | | | ○ | |
| MBCF_WRITE_F | 4 | ○ | ○ | ○ | as data | ○ | |
| MBCF_FIFO | 2 | ○ | ○ | | | ○ | |
| MBCF_STREAM | 2 | ○ | ○ | | | ○ | |
| MBCF_STREAM_READ | 2 | ○ | ○ | | | ○ | |
| MBCF_FIFOe_NORMAL | 2 | ○ | ○ | | | ○ | |
| MBCF_FIFOe_RETRY | 2 | ○ | ○ | | | ○ | |
| MBCF_STREAMe_NORMAL | 2 | ○ | ○ | | | ○ | |
| MBCF_STREAMe_RETRY | 2 | ○ | ○ | | | ○ | |
| MBCF_SIGNAL | 3 | ○ | ○ | ○[1] | | ○ | |
| MBCF_SWAP | 3 | ○ | ○ | | | ○ | |
| MBCF_CMP_SWAP | 3 | ○ | ○ | | | ○ | |
| MBCF_LOGICALop | 3 | ○ | ○ | opcode | | ○ | |
| MBCF_K_CMP_SWAP | 3 | ○ | ○ | | | ○ | |
| MBCF_FETCH_ADD | 3 | ○ | ○ | ○ | | ○ | |
| MBCF_ZBUF_PACK24 | 3 | ○ | ○ | ○ | | ○ | |
| MBCF_ZBUF_PACK | 3 | ○ | ○ | ○ | | ○ | |
| MBCF_USER_RESUME | 2n | dummy | dummy | | | | |
| MBCF_KERNEL_RESUME | 2n | dummy | dummy | | | | |
| MBCF_USER_STOP | 2n | dummy | dummy | | | | |
| MBCF_KERNEL_STOP | 2n | dummy | dummy | | | | |

表 2: MBCF コマンド一覧 (その二)

| コマンド名 | タイプ | len | addr | addr2 | addr3 | data | 備考 |
|---|---|---|---|---|---|---|---|
| MBCF_WTASKQ_STAT | 1-s | ○ | | | | ○ | |
| MBCF_WRITE_STAT | 2-s | ○ | ○ | | | ○ | |
| MBCF_BLKWRITE_STAT | 2B-s | ○ | ○ | | | ○ | |
| MBCF_BLKWRITE_CTX_STAT | 2B-s | ○ | ○ | ○ | ○ | ○ | |
| MBCF_WRITE_CTX_STAT | 2C-s | ○ | ○ | ○ | ○ | ○ | |
| MBCF_READ_STAT | 3n-s | ○ | ○ | | | ○ | |
| MBCF_WRITE_F_STAT | 4-s | ○ | ○ | ○ | as data | ○ | |
| MBCF_FIFO_STAT | 2-s | ○ | ○ | | | ○ | |
| MBCF_STREAM_STAT | 2-s | ○ | ○ | | | ○ | |
| MBCF_STREAM_READ_STAT | 2-s | ○ | ○ | | | ○ | |
| MBCF_SIGNAL_STAT | 3-s | ○ | ○ | ○ | | ○ | |
| MBCF_SWAP_STAT | 3-s | ○ | ○ | | | ○ | |
| MBCF_CMP_SWAP_STAT | 3-s | ○ | ○ | | | ○ | |
| MBCF_LOGICALop_STAT | 3-s | ○ | ○ | opcode | | ○ | |
| MBCF_K_CMP_SWAP_STAT | 3-s | ○ | ○ | | | ○ | |
| MBCF_FETCH_ADD_STAT | 3-s | ○ | ○ | ○ | | ○ | |
| MBCF_ZBUF_PACK24_STAT | 3-s | ○ | ○ | ○ | | ○ | |
| MBCF_ZBUF_PACK_STAT | 3-s | ○ | ○ | ○ | | ○ | |
| MBCF_USER_RESUME_STAT | 2n-s | dummy | dummy | | | | |
| MBCF_KERNEL_RESUME_STAT | 2n-s | dummy | dummy | | | | |
| MBCF_USER_STOP_STAT | 2n-s | dummy | dummy | | | | |
| MBCF_KERNEL_STOP_STAT | 2n-s | dummy | dummy | | | | |

#### 1.2.3.2.2 タイプ 0

*MBCF_TASK_NOP, MBCF_TASK_CLOSE*
*MBCF_TASK_END*

*MBCF packet payload* なし。

#### 1.2.3.2.3 タイプ 1 *len, data*

*MBCF_WTASKQ*

| 31 24 | 23 16 | 15 8 | 7 0 |
|---|---|---|---|
| len | | | |
| data<br>... | | | |

図 4: MBCF ペイロードフォーマット (タイプ 1)

#### 1.2.3.2.4 タイプ 1-s *len, stat_addr, data*

*MBCF_WTASKQ_STAT*

| 31 24 | 23 16 | 15 8 | 7 0 |
|---|---|---|---|
| len | | | |
| stat_addr | | | |
| data<br>... | | | |

図 5: MBCF ペイロードフォーマット (タイプ 1-s)

#### 1.2.3.2.5 タイプ 2 *addr, len, data*

*MBCF_WRITE*
*MBCF_FIFO*
*MBCF_STREAM*

| 31 24 | 23 16 | 15 8 | 7 0 |
|---|---|---|---|
| addr | | | |
| len | | | |
| data<br>... | | | |

図 6: MBCF ペイロードフォーマット (タイプ 2)

| 31 24 | 23 16 | 15 8 | 7 0 |
|---|---|---|---|
| addr | | | |
| len | | | |
| padding0 (data の 8 byte align) | | | |
| data<br>... | | | |

図 7: MBCF ペイロードフォーマット (タイプ 2, data の 8 バイトアライン)

#### 1.2.3.2.6 タイプ 2-s *addr, len, stat_addr, data*

*MBCF_WRITE_STAT*
*MBCF_FIFO_STAT*
*MBCF_STREAM_STAT*

| 31 24 | 23 16 | 15 8 | 7 0 |
|---|---|---|---|
| addr | | | |
| len | | | |
| stat_addr | | | |
| data<br>... | | | |

図 8: MBCF ペイロードフォーマット (タイプ 2-s)

#### 1.2.3.2.7 タイプ 2n *addr, len*

*MBCF_USER_RESUME, MBCF_USER_STOP*
*MBCF_KERNEL_RESUME, MBCF_KERNEL_STOP*

| 31 24 | 23 16 | 15 8 | 7 0 |
|---|---|---|---|
| addr | | | |
| len | | | |

図 9: MBCF ペイロードフォーマット (タイプ 2n)

#### 1.2.3.2.8 タイプ 2n-s *addr, len, stat_addr*

*MBCF_USER_RESUME_STAT, MBCF_USER_STOP_STAT*
*MBCF_KERNEL_RESUME_STAT, MBCF_KERNEL_STOP_STAT*

| 31 24 | 23 16 | 15 8 | 7 0 |
|---|---|---|---|
| addr | | | |
| len | | | |
| stat_addr | | | |

図 10: MBCF ペイロードフォーマット (タイプ 2n-s)

#### 1.2.3.2.9 タイプ 2C *addr, len, dst_ctx, data*

*MBCF_WRITE_CTX*

| 31 24 | 23 16 | 15 8 | 7 0 |
|---|---|---|---|
| addr | | | |
| len | | | |
| dst_ctx | | | |
| data | | | |
| ... | | | |

図 11: MBCF ペイロードフォーマット (タイプ 2C)

#### 1.2.3.2.10 タイプ 2C-s *addr, len, stat_addr, dst_ctx, data*

*MBCF_WRITE_CTX_STAT*

| 31 24 | 23 16 | 15 8 | 7 0 |
|---|---|---|---|
| addr | | | |
| len | | | |
| stat_addr | | | |
| dst_ctx | | | |
| data | | | |
| ... | | | |

図 12: MBCF ペイロードフォーマット (タイプ 2C-s)

#### 1.2.3.2.11 タイプ 2B *addr, len, [dst_ctx], data; Blocked MBCF_BLKWRITE, MBCF_BLKWRITE_CTX*

| 31 24 | 23 16 | 15 8 | 7 0 |
|---|---|---|---|
| addr | | | |
| len | | | |
| dst_ctx/padding0 | | | |
| padding1 | | | |
| padding2 | | | |
| data<br>... | | | |

図 13: MBCF ペイロードフォーマット (タイプ 2B)

*data* は *64 byte align* される。

#### 1.2.3.2.12 タイプ 2B-s *addr, len, stat_addr, [dst_ctx], data; Blocked MBCF_BLKWRITE_STAT, MBCF_BLKWRITE_CTX_STAT*

| 31 24 | 23 16 | 15 8 | 7 0 |
|---|---|---|---|
| addr | | | |
| len | | | |
| stat_addr | | | |
| dst_ctx/padding0 | | | |
| padding1 | | | |
| data<br>... | | | |

図 14: MBCF ペイロードフォーマット (タイプ 2B-s)

*data* は *64 byte align* される。

#### 1.2.3.2.13 タイプ 3 *addr, len, xaddr, data*

*MBCF_FETCH_ADD, MBCF_SIGNAL,*
*MBCF_SWAP, MBCF_CMP_SWAP, MBCF_K_CMP_SWAP,*
*MBCF_LOGICALop_STAT,*
*MBCF_ZBUF_PACK, MBCF_ZBUF_PACK24*

| 31 24 | 23 16 | 15 8 | 7 0 |
|---|---|---|---|
| addr | | | |
| len | | | |
| xaddr | | | |
| data<br>... | | | |

図 15: MBCF ペイロードフォーマット (タイプ 3)

#### 1.2.3.2.14 タイプ 3-s *addr, len, xaddr, stat_addr, data*

*MBCF_FETCH_ADD_STAT, MBCF_SIGNAL_STAT,*
*MBCF_SWAP_STAT, MBCF_CMP_SWAP_STAT, MBCF_K_CMP_SWAP_STAT,*
*MBCF_LOGICALop_STAT,*
*MBCF_ZBUF_PACK_STAT, MBCF_ZBUF_PACK24_STAT*

| 31 24 | 23 16 | 15 8 | 7 0 |
|---|---|---|---|
| addr | | | |
| len | | | |
| xaddr | | | |
| stat_addr | | | |
| data<br>... | | | |

図 16: MBCF ペイロードフォーマット (タイプ 3-s)

| 31 24 | 23 16 | 15 8 | 7 0 |
|---|---|---|---|
| addr | | | |
| len | | | |
| xaddr | | | |
| stat_addr | | | |
| padding0 (data の 8 byte align) | | | |
| data<br>... | | | |

図 17: MBCF ペイロードフォーマット (タイプ 3-s, data の 8 バイトアライン)

#### 1.2.3.2.15 タイプ 3n *addr, len, xaddr*

*MBCF_READ_STAT*

| 31 24 | 23 16 | 15 8 | 7 0 |
|---|---|---|---|
| addr | | | |
| len | | | |
| xaddr | | | |

図 18: MBCF ペイロードフォーマット (タイプ 3n)

#### 1.2.3.2.16 タイプ 3n-s *addr, len, xaddr, stat_addr*

*MBCF_READ_STAT*

| 31 24 | 23 16 | 15 8 | 7 0 |
|---|---|---|---|
| addr | | | |
| len | | | |
| xaddr | | | |
| stat_addr | | | |

図 19: MBCF ペイロードフォーマット (タイプ 3n-s)

#### 1.2.3.2.17 タイプ 4 *addr, len, xaddr, xdata, data*

*MBCF_WRITE_F*

| 31 24 | 23 16 | 15 8 | 7 0 |
|---|---|---|---|
| addr | | | |
| len | | | |
| xaddr | | | |
| xdata | | | |
| data<br>... | | | |

図 20: MBCF ペイロードフォーマット (タイプ 4)

| 31 24 | 23 16 | 15 8 | 7 0 |
|---|---|---|---|
| addr | | | |
| len | | | |
| xaddr | | | |
| xdata | | | |
| padding0 (data の 8 byte align) | | | |
| data<br>... | | | |

図 21: MBCF ペイロードフォーマット (タイプ 4, data の 8 バイトアライン)

#### 1.2.3.2.18 タイプ 4-s *addr, len, xaddr, xdata, stat_addr, data*

*MBCF_WRITE_F_STAT*

| 31 24 | 23 16 | 15 8 | 7 0 |
|---|---|---|---|
| addr | | | |
| len | | | |
| xaddr | | | |
| xdata | | | |
| stat_addr | | | |
| data<br>... | | | |

図 22: MBCF ペイロードフォーマット (タイプ 4-s)

## 1.3 MBCF コマンド体系

### 1.3.1 MBCF コマンドフィールド・フォーマット

| *Bit* | *Field* | *:ACTION_MASK2<br>+:ATTRIBUTE_MASK | =:SEND_MASK<br>@:PROT_REPLY_MASK | *Comment* |
|---|---|---|---|---|
| *00* | | * | = | |
| *01* | | * | = | |
| *02* | | * | = | |
| *03* | | * | = | |
| *04* | *STAT* 付きコマンド | * | = | |
| *05* | MBCF_REPLY_ON_FAILURE | * | = | |
| *06* | | * | = | |
| *07* | *Kernel Task* コマンド | * | = | |
| *08* | | * | = | |
| *09* | | * | = | |
| *10* | | * | = | |
| *11* | | * | = | |
| *12* | MBCF_DEBUG_DISP | * | = | |
| *13* | MBCF_ENROLL_ON_DEMAND | | = | |
| *14* | MBCF_TASK_WAKEUP | | = | |
| *15* | MBCF_NO_FLOW_CONTROL | | = | |
| *16* | *EMB level* | + | = | |
| *17* | *EMB level* | + | = | |
| *18* | *EMB level* | + | = | |
| *19* | MBCF_PROT_ACK_MASK | + | = | |
| *20* | *Proto Reply* | | @ | *1:*W_REPLY |
| *21* | *Proto Reply* | | @ | *2:*R_REPLY, *3:*A_REPLY |
| *22* | MBCF_STADDR_IN_PBOX | | | *(for pbox)* |
| *23* | MBCF_PBOX_LINK_FLAG | | | *(for pbox)* |
| *24* | MBCF_MULTI_PACKET_FLAG | | | *(for pbox)* |
| *25* | MBCF_SB_CONT | | | |
| *26* | MBCF_SB_RESET | | | |
| *27* | MBCF_RETRY_WAIT_CANCEL | | | |
| *28* | MBCF_SB_ACK | | | |
| *29* | MBCF_SB_RETRY | | | |
| *30* | MBCF_RETRY_MASK | + | | |
| *31* | MBCF_STICKY_MASK | + | | |

```
#define MBCF_MAX_DATA_LEN 1440 /* 1408 packet max length */
#define MBCF_MAX_DATA_LEN_DIRECT 1472 /* 1440 packet max length */
#define MBCF_MULTI_PACKET_LIMIT 16 /* multi packet max iteration */

#define MBCF_PROT_W_REPLY       0x00100000 /* WRITE op.  */
```

```
#define MBCF_PROT_R_REPLY       0x00200000 /* READ op.   */
#define MBCF_PROT_A_REPLY       0x00300000 /* ATOMIC op. */
#define MBCF_EMB_LEVEL_MASK     0x00070000
#define MBCF_EMB_LEVEL_SHIFT 20

#define MBCF_ACTION_MASK 0x0000ffff
#define MBCF_ACTION_MASK2 0x00001fff
#define MBCF_ACTION_MASK3 0x00000fff
#define MBCF_ACTION_MASK2SEND 0x00000fdf
#define MBCF_USER_SEND_MASK 0x000fff7f
#define MBCF_SEND_MASK         0x000fffff
#define MBCF_ATTRIBUTE_MASK 0xc00f0000
```

### 1.3.2 MBCF コマンド一覧

*SSS–PC* では表 *3* のコマンドが利用可能である。

表 3: MBCF コマンド一覧 (その一)

| コマンド名 | コード | 説明 |
|---|---|---|
| MBCF_NOP | 0x01 | 対象タスクに対して何もしない?? |
| MBCF_CLOSE | 0x02 | |
| MBCF_TASK_END | 0x82 | 対象タスクにタスク登録終了を通知する |
| MBCF_MIGRATION | 0x83 | |
| MBCF_WTASKQ | 0x81 | send_taskq() システムコール用 |
| MBCF_BLKWRITE | 0x04 | 対象タスクの対象アドレスに書き込む (パケット上のデータがアラインされる) |
| MBCF_BLKWRITE_CTX | 0x84 | /* only for kernel task */ |
| MBCF_WRITE | 0x05 | 対象タスクの対象アドレスに書き込む |
| MBCF_WRITE_CTX | 0x85 | /* only for kernel task */ |
| MBCF_READ | 0x06 | 対象タスクの対象アドレスから読み出す |
| MBCF_WRITE_F | 0x07 | 対象タスクの対象アドレスにデータを書き込み対象フラグアドレスにフラグ値を書き込む |
| MBCF_FIFO | 0x08 | 対象タスクの FIFO にデータを送る |
| MBCF_FIFOe_NORMAL | 0x48 | MBCF_FIFO で FIFO がフルのときには失敗する |
| MBCF_FIFOe_RETRY | 0x49 | MBCF_FIFO で FIFO がフルのマークを取り消す |
| MBCF_STREAM_READ | 0x46 | 対象タスクの STREAM からデータを読み出す |
| MBCF_STREAM | 0x47 | 対象タスクの STREAM にデータを送る |
| MBCF_STREAMe_NORMAL | 0x4a | MBCF_STREAM で STREAM がフルのときには失敗する |
| MBCF_STREAMe_RETRY | 0x4b | MBCF_STREAM で STREAM がフルのマークを取り消す |
| MBCF_LOGICALop | 0x4f | 対象アドレスにて送信データと論理演算を行う |
| MBCF_SIGNAL | 0x09 | 対象タスクにて SIGNAL を実行する |
| MBCF_SWAP | 0x0a | 対象タスクの対象アドレスに書き込み、元のデータを読み出す |
| MBCF_CMP_SWAP | 0x0b | 対象タスクの対象アドレスの値がゼロなら SWAP し、そうでなければ対象アドレスの値を読み出す |
| MBCF_K_CMP_SWAP | 0x8b | カーネルタスク用の CMP_SWAP |
| MBCF_FETCH_ADD | 0x0c | 対象タスクの対象アドレスに加算し古い値を読み出す |

表 4: MBCF コマンド一覧 (その二)

| コマンド名 | コード | 説明 |
|---|---|---|
| MBCF_ZBUF_PACK24 | 0x0d | 対象タスクの対象アドレスと送信データを z バッファ値と色値とし対象アドレスの色値を更新する (24 ビットカラー) |
| MBCF_ZBUF_PACK | 0x0e | 対象タスクの対象アドレスと送信データを z バッファ値と色値とし対象アドレスの色値を更新する (8 ビットカラー) |
| MBCF_USER_RESUME | 0x41 | |
| MBCF_KERNEL_RESUME | 0xc1 | |
| MBCF_USER_STOP | 0x42 | |
| MBCF_KERNEL_STOP | 0xc2 | |
| MBCF_UPDATE | 0x43 | |
| MBCF_INVALIDATE | 0x44 | |

表 5: MBCF コマンド一覧 (その三)

| コマンド名 | コード | 説明 |
|---|---|---|
| MBCF_WTASKQ_STAT | 0x91 | 状態返答付き MBCF_WTASKQ |
| MBCF_BLKWRITE_STAT | 0x14 | 状態返答付き MBCF_BLKWRITE |
| MBCF_BLKWRITE_CTX_STAT | 0x94 | 状態返答付き MBCF_BLKWRITE_CTX |
| MBCF_WRITE_STAT | 0x15 | 状態返答付き MBCF_WRITE |
| MBCF_WRITE_CTX_STAT | 0x95 | 状態返答付き MBCF_WRITE_CTX |
| MBCF_READ_STAT | 0x16 | 状態返答付き MBCF_READ |
| MBCF_WRITE_F_STAT | 0x17 | 状態返答付き MBCF_WIRTE_F |
| MBCF_FIFO_STAT | 0x18 | 状態返答付き MBCF_FIFO |
| MBCF_FIFOe_NORMAL_STAT | 0x58 | 状態返答付き MBCF_FIFOe_NORMAL |
| MBCF_FIFOe_RETRY_STAT | 0x59 | 状態返答付き MBCF_FIFOe_RETRY |
| MBCF_STREAM_READ_STAT | 0x56 | 状態返答付き MBCF_STREAM |
| MBCF_STREAM_STAT | 0x57 | 状態返答付き MBCF_STREAM |
| MBCF_STREAMe_NORMAL_STAT | 0x5a | 状態返答付き MBCF_STREAMe_NORMAL |
| MBCF_STREAMe_RETRY_STAT | 0x5b | 状態返答付き MBCF_STREAMe_RETRY |
| MBCF_LOGICALop_STAT | 0x5f | 状態返答付き MBCF_LOGICALop |
| MBCF_SIGNAL_STAT | 0x19 | 状態返答付き MBCF_SIGNAL |
| MBCF_SWAP_STAT | 0x1a | 状態返答付き MBCF_SWAP |
| MBCF_CMP_SWAP_STAT | 0x1b | 状態返答付き MBCF_CMP_SWAP |
| MBCF_K_CMP_SWAP_STAT | 0x9b | 状態返答付き MBCF_K_CMP_SWAP |
| MBCF_FETCH_ADD_STAT | 0x1c | 状態返答付き MBCF_FETCH_ADD |
| MBCF_ZBUF_PACK24_STAT | 0x1d | 状態返答付き MBCF_ZBUF_PACK24 |
| MBCF_ZBUF_PACK_STAT | 0x1e | 状態返答付き MBCF_ZBUF_PACK |
| MBCF_ZBUF_PACK8_STAT | 0x1e | 状態返答付き MBCF_ZBUF_PACK8 |
| MBCF_USER_RESUME_STAT | 0x51 | 状態返答付き MBCF_USER_RESUME |
| MBCF_KERNEL_RESUME_STAT | 0xd1 | 状態返答付き MBCF_KERNEL_RESUME |
| MBCF_USER_STOP_STAT | 0x52 | 状態返答付き MBCF_USER_STOP |
| MBCF_KERNEL_STOP_STAT | 0xd2 | 状態返答付き MBCF_KERNEL_STOP |

### 1.3.3 MBCF コマンド捕捉

1. *STAT* 返答と不正ターゲットタスク
   *WTASKQ_STAT, K_CMP_SWAP_STAT, USER_STOP_STAT, USER_RESUME_STAT* の各コマンドの場合は、パケットに指定されたターゲットタスクがターゲットノードに存在しない場合は、*STAT* 返答がソースタスクに返信される。それ以外のコマンドの場合は何も返信されない。

#### 1.3.3.1 WTASKQ_STAT の STAT 返答

| | |
|---|---|
| *1* | 成功 |
| *-1* | タスクキューがフルで失敗 |
| *-2* | 存在しないタスクが指定された |

2. *STAT* 返答とゼロバイトデータ
   *WRITE_STAT, READ_STAT* などの *STAT* 付きコマンドで、データ長としてゼロバイトを指定したパケットを使用した場合、*STAT* の値をチェックするループをまわるプログラムは永久にループから抜けない。これらコマンドは成功した場合に処理したデータ長 *(*送信したデータ長と等しい：基本的に *MBCF* ではデータの一部のみのアクセスでの処理を成功として扱うことはしない予定 *(MBCF* によるメモリアクセスのアトミック性が現実的には保証できないため*))* を返すが、*STAT* は *MBCF* 送信システムコールによりゼロで初期化されるため、値ゼロの *STAT* 返答を区別できない。

   原則的なプログラミング作法として、*MBCF* では長さゼロのデータを *MBCF* として送信しないことになっている。

3. *MBCF* パケットのコンバイニングおよび一括送信におけつ *STAT* 返答アドレス
   *MBCF* では *MBCF_PBOX* のコマンドで、*MBCF_PBOX_LINK_FLAG* を指定すると、複数のパケットをコンバイニングして一度に送信できる。*MBCF_MULTI_PACKET_FLAG* の指定がある場合は、複数の宛先に一度のシステムコールで送信できる。これらは組み合わせて送信処理を行うこともできる。

   *MBCF_PBOX_LINK_FLAG* が指定されているときは、*(MBCF_MULTI_PACKET_FLAG* の指定のパターンによらず*)* 二つ目以降のパケットではデフォルトで *flagaddr = 0* で、*MBCF_STADDR_IN_PBOX* が指定されていれば各パケットごとに *flagaddr = pbox->stat_addr* となる *(MBCF_STADDR_IN_PBOX* の扱いは最初のパケットも同様*)*。

   したがって、*MBCF_STADDR_IN_PBOX* がどの *pbox* にも指定されていない場合は、最初のパケットは *mbcf_send()* システムコールに渡された引数が直接 *flagaddr* として採用され、二つ目以降のパケットは *flagaddr == 0* となる。

   *MBCF* で *STAT* 付きコマンドをコンバインニングして利用する場合は、*mbcf_send()* の第二引数を使うよりは、各コマンドの *pbox.stat_addr* にフラグアドレスを設定して送信することが推奨されている。

### 1.3.4 MBCF_WRITE(_STAT), MBCF_BLKWRITE(_STAT)

#### 1.3.4.1 コマンドコード

*0x05* *MBCF_WRITE*
*0x15* *MBCF_WRITE_STAT*
*0x04* *MBCF_BLKWRITE*
*0x14* *MBCF_BLKWRITE_STAT*

#### 1.3.4.2 機能 ターゲットタスクにデータを書き込む

#### 1.3.4.3 入力 *pbox*

表 6: PBox 構造体 (WRITE, BLKWRITE)

| 名前 | 説明 |
|---|---|
| ltask | 対象論理 TASK 番号 |
| key | 対象 TASK のアクセスキー |
| comm | 0x05, 0x15 |
| len | 送信データのバイト長 |
| addr | 送信データの書き込み先論理アドレス |
| addr2 | – |
| addr3 | – |
| data | 送信データの先頭論理アドレス |
| status_addr | status 返答アドレス |

#### 1.3.4.4 オペレーション

*for i=0 to len-1*
*addr@ltask[i] = data[i]*

#### 1.3.4.5 出力 なし

#### 1.3.4.6 STAT 返答

*1* 成功 *(obsolete)*
*-1* 失敗 *(*ターゲットノードでアクセス違反*)*
*>0* 成功 *(*書き込みバイト数*)*

**1.3.4.7 説明** *MBCF_WRITE(_STAT), MBCF_BLKWRITE(_STAT)* コマンドは対象タスクにデータを書き込む。

*MBCF_BLKWRITE(_STAT)* はデータを *Ether* パケットの先頭から *16* バイトオフセット・アラインしたパケットフォーマットを利用する。

### 1.3.5 MBCF_WRITE_F(_STAT)

#### 1.3.5.1 コマンドコード

*0x07* *MBCF_WRITE_F*
*0x17* *MBCF_WRITE__FSTAT*

#### 1.3.5.2 機能

ターゲットタスクにデータおよびフラグを書き込む。

#### 1.3.5.3 入力 *pbox*

表 7: PBox 構造体 (WRITE_F)

| 名前 | 説明 |
|---|---|
| ltask | 対象論理 TASK 番号 |
| key | 対象 TASK のアクセスキー |
| comm | 0x07, 0x17 |
| len | 送信データのバイト長 |
| addr | 送信データの書き込み先論理アドレス |
| addr2 | 対象フラグ論理アドレス |
| addr3 | フラグデータ値 |
| data | 送信データの先頭論理アドレス |
| status_addr | status 返答アドレス |

#### 1.3.5.4 オペレーション

*for i=0 to len-1*
*addr@ltask[i] = data[i]*
$*addr2@ltask = addr3$

#### 1.3.5.5 出力

なし

#### 1.3.5.6 STAT 返答

*1* 成功 *(obsolete)*
*-1* 失敗 *(ターゲットノードでアクセス違反)*
*>0* 成功 *(書き込みバイト数 (FLAG 分は含まない))*

#### 1.3.5.7 説明

*MBCF_WRITE_F(_STAT)* コマンドは対象タスクにデータを書き込むと同時に、対象フラグに書き込む。*STAT* 返答には *FLAG* 書き込みのための *xaddr* で指定された領域への書き込み分は含まれない。

### 1.3.6 MBCF_READ(_STAT)

#### 1.3.6.1 コマンドコード

*0x06* *MBCF_READ*
*0x16* *MBCF_READ_STAT*

#### 1.3.6.2 機能 ターゲットタスクからデータを読み出す。

#### 1.3.6.3 入力 *pbox*

表 8: PBox 構造体 (READ)

| 名前 | 説明 |
|---|---|
| ltask | 対象論理 TASK 番号 |
| key | 対象 TASK のアクセスキー |
| comm | 0x06, 0x16 |
| len | 読み出すデータのバイト長 |
| addr | 読み出すデータの論理アドレス |
| addr2 | – |
| addr3 | – |
| data | 読み出したデータの格納先論理アドレス |
| status_addr | status 返答アドレス |

#### 1.3.6.4 オペレーション

*for i=0 to len-1*
*data[i] = addr@ltask[i]*

#### 1.3.6.5 出力 読み出したデータが *pbox.data* 領域に格納される。

#### 1.3.6.6 STAT 返答

*1* 成功

#### 1.3.6.7 説明 *MBCF_READ* は対象タスクから自タスクにデータを読み出す。

### 1.3.7 MBCF_SWAP(_STAT)

#### 1.3.7.1 コマンドコード

*0x0a* *MBCF_SWAP*
*0x1a* *MBCF_SWAP_STAT*

#### 1.3.7.2 機能

ターゲットタスクのターゲットアドレスに書き込み、元のデータを読み出す。

#### 1.3.7.3 入力

*pbox*

表 9: PBox 構造体 (SWAP)

| 名前 | 説明 |
|---|---|
| ltask | 対象論理 TASK 番号 |
| key | 対象 TASK のアクセスキー |
| comm | 0x0a, 0x1a |
| len | 送信データのバイト長 |
| addr | 送信データの書き込み先 (SWAP 対象) 論理アドレス |
| addr2 | – |
| addr3 | – |
| data | 送信データ (元データ格納先) の先頭論理アドレス |
| status_addr | status 返答アドレス |

#### 1.3.7.4 オペレーション

*for i=0 to len-1*
*swap(data[i], addr@ltask[i])*

#### 1.3.7.5 出力

ターゲットタスクのターゲット論理アドレスにあった元のデータが *data* 領域に格納される。

#### 1.3.7.6 STAT 返答

*1* 成功

#### 1.3.7.7 説明

*MBCF_SWAP(_STAT)* コマンドは対象タスクにデータを書き込み、ターゲットタスクの元データを *data* 領域に読み出す、*swap* 動作を実行する。

### 1.3.8 MBCF_CMP_SWAP(_STAT), MBCF_K_CMP_SWAP(_STAT)

#### 1.3.8.1 コマンドコード

*0x0b* *MBCF_CMP_SWAP*
*0x1b* *MBCF_CMP_SWAP_STAT*
*0x8b* *MBCF_K_CMP_SWAP*
*0x9b* *MBCF_K_CMP_SWAP_STAT*

**1.3.8.2 機能** *compare and swap* 動作を実行する。ターゲットタスクのターゲットアドレスの値が *0* なら送信データを書き込み、元のデータを読み出す。そうでなければ単にターゲットタスクのターゲットアドレスのデータを読み出す。

**1.3.8.3 入力** *pbox*

表 10: PBox 構造体 (CMP_SWAP, K_CMP_SWAP)

| 名前 | 説明 |
|---|---|
| ltask | 対象論理 TASK 番号 |
| key | 対象 TASK のアクセスキー |
| comm | 0x0b, 0x1b, 0x8b, 0x9b |
| len | 送信データのバイト長 (1,2,4,8 バイトのみ有効) |
| addr | 送信データの書き込み先 (SWAP 対象) 論理アドレス |
| addr2 | – |
| addr3 | – |
| data | 送信データ (元データ格納先) の先頭論理アドレス |
| status_addr | status 返答アドレス |

#### 1.3.8.4 オペレーション

```
if (*addr@ltask == 0)
        swap(data, addr@ltask)
else
        *data = *addr@ltask
```

**1.3.8.5 出力** ターゲットタスクのターゲット論理アドレスにあった元のデータが *data* 領域に格納される。

#### 1.3.8.6 STAT 返答

*1* 成功
*-1* *len* が不正
*-2* ターゲットタスクが存在しない *(K_CMP_SWAP_STAT のみ)*

**1.3.8.7　説明**　*MBCF_CMP_SWAP(_STAT), MBCF_K_CMP_SWAP(_STAT)* コマンドは対象タスクの対象データが *0* なら送信データとターゲットデータを *swap* し、そうでなければ対象データを *data* 領域に読み出す、*compare-and-swap* 動作を実行する。*len* は *1, 2, 4, 8 (*バイト数*)* のうちのどれかでなければならない。

*MBCF_K_CMP_SWAP(_STAT)* コマンドはカーネル *(*タスク？*)* のみ実行できる。*STAT* で *-2* が返答されるのは、*K_CMP_STAP_STAT* コマンドのみ。これは、カーネルが *K_CMP_SWAP_STAT* コマンドをタスクのマイグレーション処理の同期に利用しているため。

### 1.3.9 MBCF_FETCH_ADD(_STAT)

#### 1.3.9.1 コマンドコード

*0x0c* *MBCF_FETCH_ADD*
*0x1c* *MBCF_FETCH_ADD_STAT*

**1.3.9.2 機能** ターゲットタスクのターゲットアドレスを送信データの値で加算し、元のデータを読み出す。

**1.3.9.3 入力** *pbox*

表 11: PBox 構造体 (FETCH_ADD)

| 名前 | 説明 |
|---|---|
| ltask | 対象論理 TASK 番号 |
| key | 対象 TASK のアクセスキー |
| comm | 0x0c, 0x1c |
| len | 送信データのバイト長 |
| addr | 送信データの書き込み先 (ADD 対象) 論理アドレス |
| addr2 | 元データ格納先の先頭論理アドレス |
| addr3 | – |
| data | 送信 (ADD) データの先頭論理アドレス |
| status_addr | status 返答アドレス |

#### 1.3.9.4 オペレーション

```
for i=0 to len-1
        tmp[i] = addr@ltask[i]
        addr@ltask[i] += data[i]
        if (addr2 != 0)
                addr2[i] = tmp[i]
```

**1.3.9.5 出力** ターゲットタスクのターゲット論理アドレスにあった元のデータが *data* に格納される。

#### 1.3.9.6 STAT 返答

*1* 成功

**1.3.9.7 説明** *MBCF_FETCH_ADD(_STAT)* コマンドは、対象タスクの対象データを送信データの値で加算し、ターゲットタスクの元データを *data* 領域に読み出す、*fetch_and_add* 動作を実行する。

### 1.3.10 MBCF_STREAM_READ(_STAT)

#### 1.3.10.1 コマンドコード

*0x47* *MBCF_STREAM_READ*
*0x57* *MBCF_STREAM_READ_STAT*

#### 1.3.10.2 機能 ターゲットタスクの対象 *STREAM* からデータを読み出す

#### 1.3.10.3 入力 *pbox*

表 12: PBox 構造体 (STREAM_READ)

| 名前 | 説明 |
|---|---|
| ltask | 対象論理 TASK 番号 |
| key | 対象 TASK のアクセスキー |
| comm | 0x46, 0x56(STAT) |
| len | 受信バッファのバイト長 |
| addr | 読み出し先対象 MB_STREAM 構造体の論理アドレス |
| addr2 | – |
| addr3 | – |
| data | 受信バッファの論理アドレス |
| status_addr | status 返答アドレス |

#### 1.3.10.4 オペレーション

*mbstream = addr@ltask*
*ll = min(len, length(mbstream)) for i=0 to ll-1*
*data[i] = GetQue(mbstream)*

#### 1.3.10.5 出力 なし

#### 1.3.10.6 STAT 返答

*>0* *STREAM* バッファから読み出されたデータのバイト数
*-1* *STREAM* バッファが空であった
*-2* *STREAM* が初期化されていない *(top==-1 or top == NULL)*
*-2* *len* が不正 *(len¡=0)*

**1.3.10.7 説明** *MBCF_STREAM_READ* は対象タスクの対象 *STREAM* からデータを読み出す。*MBCF_STREAM_READ* を受信したノードでは対象タスクの *MB_STREAM* 構造体を読み出し、ストリームデータの読み出しおよび *MB_STREAM* 構造体の更新を実行する。一連の処理はローカルにそしてアトミックに行われる。

### 1.3.11 MBCF_STREAM(_STAT)

#### 1.3.11.1 コマンドコード

*0x47* *MBCF_STREAM*
*0x57* *MBCF_STREAM_STAT*

#### 1.3.11.2 機能 ターゲットタスクに *1* バイト *STREAM* データを書き込む

#### 1.3.11.3 入力 *pbox*

表 13: PBox 構造体 (STREAM)

| 名前 | 説明 |
|---|---|
| ltask | 対象論理 TASK 番号 |
| key | 対象 TASK のアクセスキー |
| comm | 0x47, 0x57(STAT) |
| len | 送信データのバイト長 |
| addr | 送信データの書き込み先 MB_STREAM 構造体の論理アドレス |
| addr2 | – |
| addr3 | – |
| data | 送信データの先頭論理アドレス |
| status_addr | status 返答アドレス |

#### 1.3.11.4 オペレーション

*mbstream = addr@ltask*
*for i=0 to len-1*
*PutQue(mbstream, data[i])*

#### 1.3.11.5 出力 なし

#### 1.3.11.6 STAT 返答

*MBCF_STREAMe_NORMAL_STAT, MBCF_STREAMe_RETRY_STAT* コマンド *(1.3.12* 節*)* と同様。

**1.3.11.7 説明** *MBCF_STREAM* は対象タスクに *1* バイト *STREAM* データを書き込む。*MBCF_STREAM* を受信したノードでは対象タスクの *MB_STREAM* 構造体を読み出し、受信データの格納および *MB_STREAM* 構造体の更新を実行する。一連の処理はローカルにそしてアトミックに行われる。

対象タスクは *STREAM* 構造体および *STREAM* 領域の確保ならびに、*mb_stream_init()* システムコールによる *STREAM* 構造体の初期化を実行しておかなければならない。*STREAM* 領域に受信されたデータは、対象タスクが *MB_STREAM* 構造体のポインタを利用して読み出し、自己責任でポインタを更新するか、任意のタスクが *MBCF_STREAM_READ* コマンドを使って読み出すことができる。*(*なお、*June.2003* 現在、*STREAM* 構造体は *FIFO* 構造体と等価な構造をしているが、将来的には変更される予定である。*mb_stream_init()* システムコールも未実装であり、*mb_fifo_init()* が代用される。*)*

*MB_STREAM* 構造体によって指定された領域がフルであった場合は、コマンド実行はキャンセルされる。*STREAM* 領域に保護が必要である場合には、*MB_STREAM* 構造体および *STREAM* データ領域はユーザからは *read–only* にマップされるべきである。

転送データの *FIFO* 性を管理する必要がある場合には、*MBCF_STREAM_STAT* コマンドを利用してハンドシェークを行うか、*MBCF_STREAMe* コマンドセットを利用する。送信者と受信者が一対一で通信することが分かっている場合は、送信サイズを管理することによるフローコントロールも可能である。

なお、*MBCF_FIFO* コマンドセットと違い、長さ *0* バイトの *STREAM* データを送信することはできるが、対象 *STREM* 領域に対しては何も効果がない。

### 1.3.12 MBCF_STREAMe_NORMAL(_STAT), MBCF_STREAMe_RETRY(_STAT)

#### 1.3.12.1 コマンドコード

*0x48* *MBCF_STREAMe_NORMAL*
*0x49* *MBCF_STREAMe_RETRY*
*0x58* *MBCF_STREAMe_RETRY_STAT*
*0x59* *MBCF_STREAMe_NORMAL_STAT*

#### 1.3.12.2 機能 ターゲットタスクに *STREAM* データを書き込む

#### 1.3.12.3 入力 *pbox*

表 14: PBox 構造体 (STREAMe_NORMAL, STREAMe_RETRY)

| 名前 | 説明 |
|---|---|
| ltask | 対象論理 TASK 番号 |
| key | 対象 TASK のアクセスキー |
| comm | 0x48, 0x49, 0x58, 0x59 |
| len | 送信データのバイト長 |
| addr | 送信データの書き込み先 MB_STREAM 構造体の論理アドレス |
| addr2 | – |
| addr3 | – |
| data | 送信データの先頭論理アドレス |
| status_addr | status 返答アドレス |

#### 1.3.12.4 オペレーション

```
mbstream = addr@ltask
for i=0 to len-1
        PutQue(stream, data[i])
```

#### 1.3.12.5 出力 なし

**1.3.12.6 STAT 返答** *MBCF_STREAMe_NORMAL_STAT* および *MBCF_STREAMe_RETRY_STAT* コマンドの結果返される状態は以下のような値になる。*MBCF_REPLY_ON_FAILURE* が指定されている場合は負の値に対応した結果の場合のみが返答される。

*>0* *STREAM* バッファに格納されたデータのバイト数
*-1* *STREAM* バッファへのデータ格納処理中にバッファがフルになった
*-2* *MB_STREAM* 構造体へのポインタが不正アラインしていた
*-2* *STREAM* が初期化されていない *(top==-1 or top == NULL)*
*-2* *len* が不正 *(len¡=0)*
*-3* *STREAM* がキャンセル状態のときに *MBCF_STREAMe_NORMAL* コマンドを実行しようとした

**1.3.12.7　説明**　*MBCF_STREAMe* は対象タスクに *STREAM* データを *'eager'* に書き込む。*STREAM* データの管理方法は *1.3.11* 節を参照せよ。

*MBCF_STREAM* コマンドとの違いは、*STREAM* バッファがフルの場合の処理方法にある。*MBCF_STREAMe* コマンドで *STREAM* バッファがフルになった場合、対象 *STREAM* は別の特別なコマンド *(MBCF_STREAMe_RETRY)* を受信するまでは使用を禁止される。*STREAM* データの *FIFO* 性順序管理および再送処理のためのデータコピーの管理は送信タスクの責任である。

*STREAM* 状態の遷移を通知する状態応答を受信した場合、送信タスクは *MBCF_STREAMe_RETRY* コマンドまたは *MBCF_STREAMe_RETRY_STAT* コマンドを使って、キャンセルされたパケットを再送する。*MBCF_STREAMe_RETRY* コマンドおよび *MBCF_STREAMe_RETRY_STAT* コマンドだけが対象 *MB_STREAM* 構造の再有効化を指示できる。

**MBCF_STREAMe_NORMAL** 通常の *STREAM* コマンドと同様である。*STREAM* バッファ領域がフルになりキャンセル状態に入ると、それ以降の　*MBCF_STREAMe_NORMAL* コマンドはすべてキャンセルされる。

**MBCF_STREAMe_RETRY** キャンセル状態に突入している *MB_STREAM* を再有効化する。ただし *STREAM* バッファ領域に空きが存在する場合に限る。*MB_STREAM* が有効であればこのコマンドは *MBCF_STREAMe_NORMA* コマンドと同様の処理が実行される。

**MBCF_STREAMe_NORMAL_STAT** 状態応答つきの *MBCF_STREAMe_NORMAL* コマンドである。*MBCF_REPLY_ON_FAILURE* が指定されている場合は、対象 *STREAM* に対する処理が成功しなかった場合にのみ状態を返す。*MBCF_REPLY_ON_FAILURE* が指定されていない場合は必ず状態が返される。

**MBCF_STREAMe_RETRY_STAT** 状態応答つきの *MBCF_STREAMe_RETRY* コマンドである。*MBCF_REPLY_ON* が指定されている場合は、対象 *STREAM* に対する処理が成功しなかった場合にのみ状態を返す。*MBCF_REPLY_ON_FA* が指定されていない場合は必ず状態が返される。

### 1.3.13 MBCF_FIFO(_STAT)

#### 1.3.13.1 コマンドコード

*0x08* *MBCF_FIFO*
*0x18* *MBCF_FIFO_STAT*

#### 1.3.13.2 機能 ターゲットタスクに *FIFO* データを書き込む

#### 1.3.13.3 入力 *pbox*

表 15: PBox 構造体 (FIFO)

| 名前 | 説明 |
|---|---|
| ltask | 対象論理 TASK 番号 |
| key | 対象 TASK のアクセスキー |
| comm | 0x08, 0x18 |
| len | 送信データのバイト長 |
| addr | 送信データの書き込み先 MB_FIFO 構造体の論理アドレス |
| addr2 | – |
| addr3 | – |
| data | 送信データの先頭論理アドレス |
| status_addr | status 返答アドレス |

#### 1.3.13.4 オペレーション

*mbfifo = addr@ltask*
*PutQue(mbfifo, ltask)*
*PutQue(mbfifo, pnode)*
*PutQue(mbfifo, ptask)*
*PutQue(mbfifo, len)*
*for i=0 to len-1*
    *PutQue(mbfifo, data[i])*

#### 1.3.13.5 出力 なし

#### 1.3.13.6 STAT 返答

*MBCF_FIFOe_NORMAL_STAT, MBCF_FIFOe_RETRY_STAT* コマンド *(1.3.14* 節*)* と同様。

**1.3.13.7 説明** *MBCF_FIFO* は対象タスクに *FIFO* データを書き込む。*MBCF_FIFO* を受信したノードでは対象タスクの *MB_FIFO* 構造体を読み出し、受信データの格納および *MB_FIFO* 構造体の更新を実行する。一連の処理はローカルにそしてアトミックに行われる。

対象タスクは *FIFO* 構造体および *FIFO* 領域の確保ならびに、*mb_fifo_init()* による *FIFO* 構造体の初期化を実行しておかなければならない。*FIFO* 領域に受信されたデータは、対象タスクが *mb_fifo_read()* システムコールを利用して読み出す。

*MB_FIFO* 構造体によって指定された領域がフルであった場合は、コマンド実行はキャンセルされる。*FIFO* に保護が必要である場合には、*MB_FIFO* 構造体および *FIFO* データ領域はユーザからは *read–only* にマップされるべきである。

転送データの *FIFO* 性を管理する必要がある場合には、*MBCF_FIFO_STAT* コマンドを利用してハンドシェークを行うか、*MBCF_FIFOe* コマンドセットを利用する。送信者と受信者が一対一で通信することが分かっている場合は、送信サイズを管理することによるフローコントロールも可能である。

*Enroll on Demand* フラグが *top* ポインタにセットされている *(mb_fifo_init_with_type()* で設定しておくことができる*)* *FIFO* 構造体を指定して、コマンドに *MBCF_ENROLL_ON_DEMAND* をセットしたパケットを受信した場合、ターゲットタスクの論理タスク表に送信元タスクが登録されていない場合でも、送信元タスクを論理タスク表に登録した上で、通常通り受信処理を継続する。事前に *pstart* 起動や *send_taskque()* などでタスクの登録をしておかなくても、*MBCF_FIFO* コマンドを起点にしてタスク同士を通信できる状態に移行させることができる。

**1.3.13.8　バグ**　*STAT* 返答について *MBCF_FIFOe_NORMAL_STAT, MBCF_FIFOe_RETRY_STAT* コマンド *(1.3.14* 節*)* と同様のバグが存在する。

### 1.3.14 MBCF_FIFOe_NORMAL(_STAT), MBCF_FIFOe_RETRY(_STAT)

#### 1.3.14.1 コマンドコード

*0x48* *MBCF_FIFOe_NORMAL*
*0x49* *MBCF_FIFOe_RETRY*
*0x58* *MBCF_FIFOe_RETRY_STAT*
*0x59* *MBCF_FIFOe_NORMAL_STAT*

**1.3.14.2 機能** ターゲットタスクに *FIFO* データを書き込む

**1.3.14.3 入力** *pbox*

表 16: PBox 構造体 (FIFOe_NORMAL, FIFOe_RETRY)

| 名前 | 説明 |
|---|---|
| ltask | 対象論理 TASK 番号 |
| key | 対象 TASK のアクセスキー |
| comm | 0x48, 0x49, 0x58, 0x59 |
| len | 送信データのバイト長 |
| addr | 送信データの書き込み先 MB_FIFO 構造体の論理アドレス |
| addr2 | – |
| addr3 | – |
| data | 送信データの先頭論理アドレス |
| status_addr | status 返答アドレス |

**1.3.14.4 オペレーション**

*mbfifo = addr@ltask*
*PutQue(mbfifo, ltask)*
*PutQue(mbfifo, pnode)*
*PutQue(mbfifo, ptask)*
*PutQue(mbfifo, len)*
*for i=0 to len-1*
    *PutQue(mbfifo, data[i])*

**1.3.14.5 出力** なし

**1.3.14.6 STAT 返答** *MBCF_FIFOe_NORMAL_STAT* および *MBCF_FIFOe_RETRY_STAT* コマンドの結果返される状態は以下のような値になる。*MBCF_REPLY_ON_FAILURE* が指定されている場合は負の値に対応した結果の場合のみが返答される。

| | |
|---|---|
| *>0* | *FIFO* バッファに格納されたデータのバイト数 |
| *1?* | ？コマンドは完全に処理されデータはすべて *FIFO* バッファに格納された？ |
| *2?* | ？コマンドはキャンセルされデータは *FIFO* バッファに格納されなかった？ |
| *3?* | ？*FIFO* バッファがキャンセル状態であったのでデータは *FIFO* バッファ領域に格納されなかった？ |
| *-1* | *FIFO* バッファへのデータ格納処理中にバッファがフルになった |
| *-2* | *MB_FIFO* 構造体へのポインタが不正アラインしていた |
| *-2* | 対象タスクが見つからない |
| *-2* | *FIFO* が初期化されていない *(top==-1 or top == NULL)* |
| *-3* | *FIFO* がキャンセル状態のときに *MBCF_FIFOe_NORMAL* コマンドを実行しようとした |

**1.3.14.7　説明**　*MBCF_FIFOe* は対象タスクに *FIFO* データを *'eager'* に書き込む。*FIFO* データの管理方法は *1.3.13* 節を参照せよ。

*MBCF_FIFO* コマンドとの違いは、*FIFO* バッファがフルの場合の処理方法にある。*MBCF_FIFOe* コマンドで *FIFO* バッファがフルになった場合、対象 *FIFO* は別の特別なコマンド *(MBCF_FIFOe_RETRY)* を受信するまでは使用を禁止される。*FIFO* データの *FIFO* 性順序管理および再送処理のためのデータコピーの管理は送信タスクの責任である。

*FIFO* 状態の遷移を通知する状態応答を受信した場合、送信タスクは *MBCF_FIFOe_RETRY* コマンドまたは *MBCF_FIFOe_RETRY_STAT* コマンドを使って、キャンセルされたパケットを再送する。*MBCF_FIFOe_RETRY* コマンドおよび *MBCF_FIFOe_RETRY_STAT* コマンドだけが対象 *MB_FIFO* 構造の再有効化を指示できる。

**MBCF_FIFOe_NORMAL** 通常の *FIFO* コマンドと同様である。*FIFO* バッファ領域がフルになりキャンセル状態に入ると、それ以降の　*MBCF_FIFOe_NORMAL* コマンドはすべてキャンセルされる。

**MBCF_FIFOe_RETRY** キャンセル状態に突入している *MB_FIFO* を再有効化する。ただし *FIFO* バッファ領域に空きが存在する場合に限る。*MB_FIFO* が有効であればこのコマンドは *MBCF_FIFOe_NORMAL* コマンドと同様の処理が実行される。

**MBCF_FIFOe_NORMAL_STAT** 状態応答つきの *MBCF_FIFOe_NORMAL* コマンドである。*MBCF_REPLY_ON_FAI* が指定されている場合は、対象 *FIFO* に対する処理が成功しなかった場合にのみ状態を返す。*MBCF_REPLY_ON_FAILU* が指定されていない場合は必ず状態が返される。

**MBCF_FIFOe_RETRY_STAT** 状態応答つきの *MBCF_FIFOe_RETRY* コマンドである。*MBCF_REPLY_ON_FAILURE* が指定されている場合は、対象 *FIFO* に対する処理が成功しなかった場合にのみ状態を返す。*MBCF_REPLY_ON_FAILU* が指定されていない場合は必ず状態が返される。

**1.3.14.8　バグ**　データなし *(len == 0)* の *FIFO_STAT, MBCF_FIFOe_NORMAL_STAT* および *MBCF_FIFOe_RETRY_S* コマンドの結果返される状態値は *0(*ゼロ*)* であるため、*2003.June* 時点での *SSS–PC* では送信側が *STAT* 値の変化を検知できない。そのため送信側が *STAT* をチェックするループを回っていると、ループから抜けられない。

### 1.3.15 MBCF_SIGNAL(_STAT)

#### 1.3.15.1 コマンドコード

*0x09* *MBCF_SIGNAL*
*0x19* *MBCF_SIGNAL_STAT*

#### 1.3.15.2 機能

ターゲットタスクの権限で指定されたプログラムを呼び出す。

#### 1.3.15.3 入力 *pbox*

表 17: PBox 構造体 (SIGNAL)

| 名前 | 説明 |
|---|---|
| ltask | 対象論理 TASK 番号 |
| key | 対象 TASK のアクセスキー |
| comm | 0x09, 0x19 |
| len | 送信データのバイト長 |
| addr | 呼び出すシグナルに対応した MB_SIGNAL 構造体の論理アドレス |
| addr2 | MB_SIGNAL_MEMW_OFFSET を使う場合のオフセット値<br>それ以外では未使用 |
| addr3 | – |
| data | DSP, ULS)<br>データ領域への送信データの先頭論理アドレス<br>SSP)<br>data[0]:呼び出すプログラムの論理アドレス<br>data[1–]:データ領域への送信データ |
| status_addr | status 返答アドレス |

#### 1.3.15.4 オペレーション

プログラムを呼び出す。

#### 1.3.15.5 出力

なし

#### 1.3.15.6 STAT 返答

*MBCF_SIGNAL_STAT* コマンドの結果返される状態は以下のような値になる。*MBCF_REPLY_ON*
が指定されている場合は負の値に対応した結果の場合のみが返答される。

| | |
|---|---|
| *2* | 受信は成功したが、ターゲットタスクがシグナルをブロックしていた。 |
| *1* | ターゲットノードでシグナルがアンブロックされペンディングしていた要求が登録された *(*以前に *STAT* として *2* を返答していたコマンドに対応する分として*)* |
| *1* | シグナル要求の登録処理が対象ノードにて成功した |
| *-1* | シグナル要求のキューがフル |
| *-1* | 呼び出し条件が *EVERY_TIME_BLOCK* の場合に、以前のシグナルが実行中であった |
| *-2* | シグナルのタイプが不正 |
| *-2* | 対象タスクが見つからない |
| *-1,-2,-3* | *FIFO* タイプのデータ格納処理が失敗した |

**1.3.15.7　説明**　*MBCF_SIGNAL* はターゲットタスクの権限で指定されたプログラムを呼び出す。プログラムはタスクが再スケジュールされる時点で呼び出される。ターゲットタスクはメモリベースシグナル構造体を準備する。これは保護およびセキュリティの観点から重要。したがってメモリベースシグナル構造体はユーザレベルからは *read–only* に設定されていることが望ましい。*MBCF_SIGNAL_STAT* コマンドは、*MBCF_REPLY_ON_FAILURE* が指定されている場合は、対象 *SIGNAL* に対する処理が成功しなかった場合にのみ状態を返す。*MBCF_REPLY_ON_FAILURE* が指定されていない場合は必ず状態が返される。

送信タスクは対象タスクのメモリベースシグナル構造体を指定して *MBCF_SIGNAL* コマンドを発行する。呼び出されるプログラムへのデータの供給はバッファ領域への *MBCF_WRITE* コマンドまたは *MBCF_FIFO* コマンドと同様の機構で行うことができる。メモリベースシグナル構造体は以下の情報を持つ。

**ENROLL フラグ**　`Mb_siganl.type & MB_SIGNAL_ENROLL_on_DEMAND`

**呼び出しタイプ**　`Mb_siganl.type & MB_SIGNAL_INVOKE_TYPE_MASK`

**データ領域タイプ**　`Mb_siganl.type & MB_SIGNAL_DATAOP_TYPE_MASK`

**呼び出し条件**　`Mb_siganl.freq`

**対象タスク指定プログラム**　`Mb_siganl.dsp`[2]

**ユーザレベルスケジューラ**　`Mb_siganl.uls` *(User Level Scheduler)*

**データ格納領域アドレス**　`Mb_siganl.address`

**1.3.15.7.1　ENROLL フラグ**　*ENROLL* フラグがセットされている *(mb_signal_init()* 時点に設定しておくことができる*)* と、ターゲットタスクに送信元タスクが論理タスク表に登録されていない場合でも、送信元タスクを論理タスク表に登録した上で、通常通り受信処理を継続する。事前に *pstart* 起動や *send_taskque()* などでタスクの登録をしておかなくても、*MBCF_SIGNAL* コマンドを起点にしてタスク同士を通信できる状態に移行させることができる。ただし、送信タスクがコマンドに *MBCF_ENROLL_ON_DEMAND* をセットして送信している場合に限定される。

[2] Destination Specified Program; ターゲットタスクの通常のユーザレベル関数のこと

#### 1.3.15.7.2 呼び出しタイプ

呼び出しタイプには以下のものがある。

**MB_SIGNAL_NONE** メモリベースシグナル構造体の動作を一時的に中断する。

**MB_SIGNAL_POLLING** *MBCF_SIGNAL* コマンドの受信時にはプログラムの呼び出しはせずに受信データ等の書き込みのみを行う。ターゲットタスクはデータを *polling* してチェックし、必要に応じてプログラムを起動する。データ領域タイプが指定されていない場合ターゲット側がシグナルを検知する方法がないので、*MBCF_SIGNAL* は失敗する。

**MB_SIGNAL_ULS_INVOKE** 対象タスクのユーザレベルスケジューラ *(ULS)* を起動する。*ULS* は *OS* が管理する。

**MB_SIGNAL_DSP_BY_ULS** 対象タスクの *ULS* を起動する。*ULS* は保護とセキュリティのチェックをしたうえでメモリベースシグナル構造体に指定されたターゲットタスクの関数 *(DSP)* を呼び出す。

**MB_SIGNAL_DSP_INVOKE** 対象タスクの関数 *(DSP)* を直接呼び出す。

**MB_SIGNAL_SSP_INVOKE** 対象タスクに対して任意の命令アドレスを指定して関数 *(SSP)* を呼び出す。呼び出す関数の命令アドレスはソースタスクが *MBCF_SIGNAL* パケットに入れて指定する。

**MB_SIGNAL_SSP_MEMW_SSP** 動作は `MB_SIGNAL_SSP_INVOKE` と同じ。ただしデータの格納領域は、メモリベースシグナル構造体に指定されたアドレスに対してオフセットを指定できる。ソースタスクの情報はまったく残らない。

#### 1.3.15.7.3 データ領域タイプ

データ領域タイプには以下のものがある。

**MB_SIGNAL_NONW** 対象タスクに対してデータは一切格納されない。

**MB_SIGNAL_QUEW** データ領域として *FIFO* を使う。*FIFO* 領域はメモリベースシグナル構造体により指定される。受信パケットの情報およびデータが *FIFO* に格納される。*MBCF_SIGNAL* コマンド受信時点で *FIFO* がフルであった場合は、コマンドパケットはキャンセルされる。

**MB_SIGNAL_MEMW** データ領域として一つのバッファを使う。バッファ領域はメモリベースシグナル構造体により指定される。受信パケットの情報およびデータが格納される。一つの固定アドレスがバッファ領域へのポインタとして使用されるので、続けて *MBCF_SIGNAL* コマンドを受信した場合は、データは上書きされる。

**MB_SIGNAL_MEMW_OFFSET** データ領域として、メモリベースシグナル構造体に指定されたベースアドレスおよび *MBCF_SIGNAL* コマンドパケットに指定されたオフセットから計算されるバッファが使われる。*ULS_INVOKE, DSP_BY_ULS* では無効。*OFFSET* は *address2 in packet*。

#### 1.3.15.7.4 呼び出し条件タイプ

呼び出し条件タイプには以下のものがある。

**MBCF_SIGNAL_EVERY_TIME_BLK** プログラムの呼び出しはアトミックに行われる。以前に受信した *MBCF_SIGNAL* コマンドに呼び出されたプログラムがまだ実行している場合はキャンセルされる。言い替えれば、それ以降のメッセージを最初のメッセージの処理が済むまで受け取らないという意味。

**MBCF_SIGNAL_FIRST_ONLY_BLK** *MBCF_SIGNAL_FIRST_ONLY_BLK* と同じ動作。

**MBCF_SIGNAL_EVERY_TIME_NONBLK** *MBCF_SIGNAL* コマンドを受信するたびにプログラムを呼び出す。アトミックではないので、プログラムの実行中に同じプログラムを指定した *MBCF_SIGNAL* コマンドにより割り込まれる可能性がある。

**MBCF_SIGNAL_FIRST_ONLY_NONBLK** 以前に受信した *MBCF_SIGNAL* コマンドに呼び出されたプログラムがまだ実行している場合は、データ格納領域にパケット情報およびデータが格納され、プログラムは呼び出されない。この場合受信した *MBCF_SIGNAL* コマンドは成功したことになる。プログラムが実行中でない場合のみ *MBCF_SIGNAL* コマンド受信時点でプログラムが呼び出される。プログラム実行中に到着した *MBCF_SIGNAL* コマンドを検知するために、*MBCF* 割り込みハンドラはフラグを管理する。このフラグはプログラム実行中に到着した *MBCF_SIGNAL* コマンドをの処理時にセットされる。このフラグをリセットするには、*mb_signal_flag_reset()* システムコール *(2.2.7* 節*)* を利用する。通常このシステムコールは呼び出されるプログラムの最後にターゲットタスクにて実行される。このシステムコールは *FIFO* に新たなエントリがあればその情報を返し、何もなければフラグをリセットする。

**1.3.15.7.5　呼び出しプログラムへのパラメータ**　シグナルから呼び出されるプログラム関数は、*ULS*、*DSP*、*SSP* いずれの場合も次のようなプロトタイプを持つ。

```
static int user_invoke_program(dataopp, sigp, arg0)
char*           dataopp;
MB_SIGNAL*      sigp;
int             arg0;
```

**dataopp** *MB_SIGNAL* 構造体に *mb_init_signal() (2.2.6* 節参照*)* への *address* 引数からセットされているアドレス。*FIFO* を使う場合なら *MB_FIFO* 構造体へのポインタ、バッファ領域を使うならその領域へのポインタである。

**sigp** *mb_init_signal() (2.2.6* 節参照*)* を使ってターゲットタスクが準備している *MB_SIGNAL* 構造体へのポインタである。ソースタスクは *pbox.addr* にセットし指定する。

**arg0** ソースタスクデータの先頭。ソースタスクが *pbox.data* に指定したデータは *FIFO* またはデータバッファにコピーされる。その先頭 *4* バイトは呼び出された関数からは *arg0* 引数として知ることができるようになっている。ただし *SSP* を使用する場合、*pbox.data* の先頭 *4* バイトは起動するプログラム情報であるため、*arg0* にセットされるのはその次の *4* バイトである起動プログラムへの引数データとなる。

以下に *MBCF_SIGNAL* パケット受信時に決定されるパラメータについてまとめる。*func* は呼び出されることになるユーザ関数である。*datap* は送信タスクが *MBCF_SIGNAL* を送信する再に *pbox.data* に指定したデータを意味する。送信タスクから起動するプログラムへ引数を渡せる仕組みになっている。

表 18: UIP invoke parameters

| type & MB_SIGNAL_INVOKE_TYPE_MASK | func | arg0 |
|---|---|---|
| MB_SIGNAL_POLLING | NULL | 0 |
| MB_SIGNAL_ULS_INVOKE | sigp->uls | 0 |
| MB_SIGNAL_DSP_BY_ULS | sigp->uls | 0 |
| MB_SIGNAL_DSP_INVOKE | sigp->dsp | datap[0] |
| MB_SIGNAL_SSP_INVOKE | datap[0] | datap[1] |
| MB_SIGNAL_MEMW_SSP | datap[0] | datap[1] |

表 19: UIP dataop parameters

| type & MB_SIGNAL_DATAOP_TYPE_MASK | dataopp | sigp |
|---|---|---|
| MB_SIGNAL_QUEW | (MB_FIFO*)sigp->address | sigp |
| MB_SIGNAL_MEMW | (char *)sigp->address | sigp |
| MB_SIGNAL_MEMW_OFFSET | (char *)sigp->address + addr2 | sigp |

### 1.3.16 MBCF_LOGICALop(_STAT)

#### 1.3.16.1 コマンドコード

*0x0c* *MBCF_LOGICALop*
*0x1c* *MBCF_LOGICALop_STAT*

#### 1.3.16.2 機能 ターゲットタスクの対象アドレスで送信データと論理演算を実行する。

#### 1.3.16.3 入力 *pbox*

表 20: PBox 構造体 (LOGICALop)

| 名前 | 説明 |
|---|---|
| ltask | 対象論理 TASK 番号 |
| key | 対象 TASK のアクセスキー |
| comm | 0x4f, 0x5f(STAT) |
| len | 送信データのバイト長 |
| addr | 送信データの書き込み先 (論理演算対象) 論理アドレス |
| addr2 | 論理演算オペコード |
| addr3 | – |
| data | 送信 (論理演算) データの先頭論理アドレス |
| status_addr | status 返答アドレス |

#### 1.3.16.4 オペレーション

*OPCODE = addr2*
*for i=0 to len-1*
*addr@ltask[i] = OPCODE(addr@ltask[i], data[i])*

#### 1.3.16.5 出力 なし

#### 1.3.16.6 STAT 返答

*>0* 対象データの先頭論理アドレス
*-1* 不正な論理演算オペコードが指定された
*-2* 対象アドレスへのアクセスでページフォールトが発生した

**1.3.16.7 説明** *MBCF_LOGICALop(_STAT)* コマンドは、対象タスクの対象データと送信データの値で論理演算を実行し、対象データを更新する。論理演算オペコードは *addr2* で指定する。有効な論理演算オペコードは表 *21* の通り。

*MBCF_LOGICAL_NOP* は意味のある論理演算は行われない。受信側で対象アドレスの値をメモリから読み出す動作のみを実行するため、論理アドレスの有効性のチェックに利用することが可能である。

表 21: LOGICALop opecodes

| opcode | 値 | 説明 |
|---|---|---|
| MBCF_LOGICAL_NOP | 0 | 演算なし |
| MBCF_LOGICAL_AND | 1 | and 演算 |
| MBCF_LOGICAL_OR | 2 | or 演算 |
| MBCF_LOGICAL_XOR | 3 | xor 演算 |
| MBCF_LOGICAL_NOT | 4 | not 演算 |
| MBCF_LOGICAL_NAND | 5 | nand 演算 |
| MBCF_LOGICAL_NOR | 6 | nor 演算 |
| MBCF_LOGICAL_XNOR | 7 | xnor 演算 |

### 1.3.17 MBCF_ZBUF_PACK24(_STAT), MBCF_ZBUF_PACK(_STAT)

#### 1.3.17.1 コマンドコード

*0x0d* *MBCF_ZBUF_PACK24*
*0x1d* *MBCF_ZBUF_PACK24_STAT*
*0x0e* *MBCF_ZBUF_PACK*
*0x1e* *MBCF_ZBUF_PACK_STAT*

#### 1.3.17.2 機能 ターゲットタスクに Z バッファデータを書き込む。

#### 1.3.17.3 入力 *pbox*

表 22: PBox 構造体 (ZBUF_PACK24, ZBUF_PACK)

| 名前 | 説明 |
|---|---|
| ltask | 対象論理 TASK 番号 |
| key | 対象 TASK のアクセスキー |
| comm | 0x07, 0x17 |
| len | 送信データのバイト長 |
| addr | 送信データの書き込み先論理アドレス |
| addr2 | 対象 Z バッファ論理アドレス |
| addr3 | – |
| data | 送信データの先頭論理アドレス |
| status_addr | status 返答アドレス |

#### 1.3.17.4 オペレーション

意味上のオペレーション：

```
for i=0 to len-1
    if (data[i].z ≥ addr2@ltask[i])
        addr@ltask[i] = data[i].color
        addr2@ltask[i] = data[i].z
```

*MBCF_ZBUF_PACK24(_STAT)* コマンドの場合：

```
for i=0 to len-1
    if (data[2*i] ≥ addr2@ltask[i])
        addr@ltask[i] = data[2*i+1] /* color */
        addr2@ltask[i] = data[2*i] /* z-value */
```

*MBCF_ZBUF_PACK(_STAT)* コマンドの場合：

*for i=0 to len-1*

*if (data[i] & 0x00ffffff* $\geq$ *addr2@ltask[i])*

*addr@ltask[i] = (data[i] >> 24) /* color */*

*addr2@ltask[i] = (data[i] & 0x00ffffff) /* z-value */*

#### 1.3.17.5 出力 なし

#### 1.3.17.6 STAT 返答

*1* 成功

*2* 成功 *(*更新されない値が存在した*; (*`data[i].z < addr2@ltask[i]`*))*

**1.3.17.7 説明** *MBCF_ZBUF_PACK24(_STAT), MBCF_ZBUF_PACK(_STAT)* コマンドは対象タスクに *Z* バッファデータを書き込む。*MBCF_ZBUF_PACK24(_STAT)* コマンドは *24* ビットカラー、*32* ビット *Z* バッファ値データを、*MBCF_ZBUF_PACK(_STAT)* コマンドは *8* ビットカラー、*24* ビット *Z* バッファ値データを扱う。*MBCF_ZBUF_PACK24(_STAT)* コマンドでは、

```
pbox.data[2*i] = z[i]
pbox.data[2*i+1] = color[i]
```

の形式をとる。*MBCF_ZBUF_PACK(_STAT)* コマンドでは、

```
pbox.data[i] = (z[i] & 0x00ffffff)
pbox.data[i] = ((color[i] & 0x000000ff) << 24)
```

の形式をとる。

## 1.4 コマンドのオプションフラグ

本節では *MBCF* コマンドに付与できる幾つかのフラグについて説明する。これらフラグは *MBCF_PBOX.command* に *OR* 演算して設定することで利用できる機能である。

### 1.4.1 MBCF_TASK_WAKEUP

ターゲットタスクが *sleep* 状態であった場合に、ターゲットタスクを *wakeup* することを指示する。ターゲットタスクが `conditional_sleep(100, TASK_WAKEUP_MBCF);` のようにしてスリープしていた場合に適用される。第二引数に *_TASK_WAKEUP_MBCF* をセットして *conditional_sleep()* をしたタスクは、*sleep* 時間 *(*第一引数*)* が経過していなくても、*MBCF* コマンドを受信した時点でおこされる。相手からの次の応答がいつ起こるか見積もれない場面でも効率をおとさずにタスクは待機することができるようになる。

### 1.4.2 MBCF_MULTI_PACKET_FLAG

複数のノードへ向けた *MBCF* コマンドを一度の *MBCF* 送信システムコールで送信処理を実行することを指示する。送信のたびに発生するシステムコール呼び出しのオーバヘッドを軽減できる。*MBCF_PBOX_LINK_FLAG* と組み合わせて使うことができる。

### 1.4.3 MBCF_PBOX_LINK_FLAG

同一ノードへ向けた複数のコマンドを一つのパケットにコンバイニングして送信することを指示する。あわせて *MBCF_PBOX.next* ポインタを適切に設定し、コンバイニングするコマンドに対応した *MBCF_PBOX* 構造体をリンクしておく。*MBCF_PBOX_LINK_FLAG* と組み合わせて使うことができる。

### 1.4.4 MBCF_STADDR_IN_PBOX

*MBCF_MULTI_PACKET_FLAG* や *MBCF_PBOX_LINK_FLAG* を利用して、*STAT* 応答つきコマンドを使用する場合に、返答アドレスを *MBCF* 送信システムコールの第二引数ではなく、*MBCF_PBOX.stat_addr* の値を使用することを指示する。一度に送信処理を実行するケースでもコマンドごとに異なるアドレスで *STAT* 応答を待つことができるようになる。

### 1.4.5 MBCF_NO_FLOW_CONTROL

このフラグはユーザプログラムは普通使わない。*MBCF* のシークエンス番号管理を行わないことを指示する。そのためターゲットノードに到着することも、到着順も保証されない。

### 1.4.6 MBCF_ENROLL_ON_DEMAND

自タスクが登録されていないタスクに *MBCF* コマンドを送信した場合に、ターゲットタスクの論理タスク表に自タスクを登録するように指示する。*send_taskque()* を使わなくても、独立したタスク同士が *MBCF* 通信できる状態になれる。*MB_FIFO* 構造体と *MBCF_FIFO* コマンドおよび、*MB_SIGNAL* 構造体と *MBCF_SIGNAL* コマンドと組み合わせて利用する。

一旦論理タスク表に登録したタスクとは、それ以降は安全に *MBCF* 通信が可能である。

### 1.4.7 MBCF_DEBUG_DISP

通常は表示されない *MBCF* 送信処理や *MBCF* 受信処理にで出力するデバッグ用のメッセージを *ON* にする。

# 2 MBCF プログラミングインタフェース

## 2.1 構造体

### 2.1.1 PBox 構造体

*MBCF* コマンドは *PBox* 構造体にパラメータをセットして送信システムコールを呼び出すことでを実行する。*PBox* 構造体は表 *23* のような情報を持つ。

表 23: PBox 構造体の定義

| 型 | 名前 | 説明 |
|---|---|---|
| int | ltask | 対象論理 TASK 番号 |
| unsigned int | key | 対象 TASK のアクセスキー |
| unsigned int | comm | 操作コマンド (Light Only) |
| unsigned int | len | 操作対象のバイト長さ |
| unsigned int | addr | 操作対象論理アドレス |
| unsigned int | addr2 | 操作対象論理アドレスまたはデータ（予備 1） |
| unsigned int | addr3 | 操作対象論理アドレスまたはデータ（予備 2） |
| unsigned char * | data | データのポインタ (先頭アドレス)<br>又は返答アドレス<br>status 返答アドレスは syscall 第二引数にセットする |
| struct Mbcf_pbox* | next | 多重メッセージ時の pbox のリンクリスト |
| unsigned int | status_addr | MBCF_STADDR_IN_PBOX 設定時の<br>status 返答アドレス |
| int | src_ctx | for MBCF_WRITE_CTX, MBCF_BLKWRITE_CTX |
| int | dst_ctx | for MBCF_WRITE_CTX, MBCF_BLKWRITE_CTX |

*PBox* は mbcf.h にて以下のように宣言される。

```
struct Mbcf_pbox {
        int             ltask;          /* 対象論理 TASK 番号 */
        unsigned int    key;            /* 対象 TASK のアクセスキー */
        unsigned int    comm;           /* 操作コマンド (Light Only) */
        unsigned int    len;            /* 操作対象のバイト長さ */
        unsigned int    addr;           /* 操作対象論理アドレス */
        unsigned int    addr2;          /* 操作対象論理アドレス（予備 1） */
        unsigned int    addr3;          /* 操作対象論理アドレス（予備 2） */
        unsigned char   *data;          /* データのポインタ (先頭アドレス)
                                           又は返答アドレス */
                                        /* status 返答アドレスは syscall 第二引数 */
        struct Mbcf_pbox *next;         /* 多重メッセージ時のリンク */
        unsigned int    status_addr;    /* MBCF_STADDR_IN_PBOX 設定時の
                                           status 返答アドレス */
        int             src_ctx;        /* for MBCF_WRITE_CTX, MBCF_BLKWRITE_CTX */
```

```
        int             dst_ctx;        /* for MBCF_WRITE_CTX, MBCF_BLKWRITE_CTX */
};
#define MBCF_PBOX       struct Mbcf_pbox

#ifdef ultra
/*
 * for 32bit tasks
 */
struct Mbcf_pbox32 {
        int             ltask;          /* 対象論理 TASK 番号 */
        unsigned int    key;            /* 対象 TASK のアクセスキー */
        unsigned int    comm;           /* 操作コマンド (Light Only) */
        unsigned int    len;            /* 操作対象のバイト長さ */
        unsigned int    addr;           /* 操作対象論理アドレス */
        unsigned int    addr2;          /* 操作対象論理アドレス（予備 1） */
        unsigned int    addr3;          /* 操作対象論理アドレス（予備 2） */
        unsigned int    data;           /* データのポインタ (先頭アドレス)
                                           又は返答アドレス */
                                        /* status 返答アドレスは syscall 第二引数 */
        unsigned int    next;           /* 多重メッセージ時のリンク */
        unsigned int    status_addr;    /* MBCF_STADDR_IN_PBOX  設定時の
                                           status 返答アドレス */
};
#define MBCF_PBOX32     struct Mbcf_pbox32
#endif
```

### 2.1.2 EMB_STRUCT 構造体

*EMB_STRUCT* 構造体は表 *24* の情報を持つ。

表 24: EMB_STRUCT 構造体

| 型 | 名前 | 説明 |
|---|---|---|
| unsigned int | w_req_count | ライトリクエスト数 |
| unsigned int | w_rpl_count | ライトリプライ受信数 |
| unsigned int | r_req_count | リードリクエスト数 |
| unsigned int | r_rpl_count | リードリプライ受信数 |
| unsigned int | a_req_count | 不可分操作リクエスト数 |
| unsigned int | a_rpl_count | 不可分操作リプライ受信数 |

*EMB_STRUCT* 構造体 は `mbcf.h` にて以下のように宣言される。

```
#define LEVEL_OF_EMB    8
struct Emb_struct{                      /* elastic memory barrier */
    unsigned int        w_req_count;    /* ライトリクエスト数 */
    unsigned int        w_rpl_count;    /* ライトリプライ受信数 */
    unsigned int        r_req_count;    /* リードリクエスト数 */
    unsigned int        r_rpl_count;    /* リードリプライ受信数 */
    unsigned int        a_req_count;    /* 不可分操作リクエスト数 */
    unsigned int        a_rpl_count;    /* 不可分操作リプライ受信数 */
};
#define EMB_STRUCT      struct Emb_struct
```

### 2.1.3 MB_FIFO 構造体

*EMB_FIFO* 構造体は表 *25* の情報を持つ。*FIFO* 構造体の初期化は *mb_fifo_init_(),mb_fifo_init_with_type()* シ

表 25: MB_FIFO 構造体

| 型 | 名前 | 説明 |
|---|---|---|
| unsigned int * | top | バッファ領域の境界下限論理アドレス |
| unsigned int * | bottom | バッファ領域の境界上限論理アドレス |
| unsigned int * | head | バッファに挿入されたデータの先頭論理アドレス |
| unsigned int * | tail | バッファに挿入されたデータの最後尾論理アドレス |

ステムコールを使う。*FIFO* バッファはリングバッファとして管理される。*FIFO* バッファからのデータの読みだしは *mb_read_fifo()* システムコールを使うが、*FIFO* 構造体を *STREAM* として利用している場合は、*head, tail* ポインタを適切に操作すれば、直接 *FIFO* バッファのデータを読み出すことも可能である。

*top* ポインターの下 *2* ビットを使って *FIFO* へのアクセス制御を行っている。*FIFO* バッファがフルになると `top & 0x1` ビットがセットされ、*FIFO* からデータが読み出されるまで新しいデータは受信されない。*MBCF_FIFO_STAT* コマンドの送信者にはエラーが戻される。

*FIFO* 構造体の `top & 0x2` *(Enroll on Demand)* ビットがセットされていると、*MBCF_ENROLL_ON_DEMAND* が付与された *MBCF_FIFO, MBCF_FIFO_STAT* コマンドを受信すると、受信タスクの論理タスク表に送信元タスクが未登録であれば、登録をした上でコマンドの処理が実行される。ビットのセットは *mb_init_fifo_with_type()* システムコールを使うか、任意の時点でユーザが直接 *top* ポインタにセットする。*Enroll on Demand* の処理は *STREAM* 関連処理では認められていない。

*MB_FIFO* 構造体 は `mbcf.h` にて以下のように宣言される。

```
struct Mb_fifo{
    unsigned int*       top; /* 0 or 0xffffffff : 使用中断
top & 0x1 == 1 で中断 at STREAM
top & 0x2 == 2 で Enroll on Demand
 */
    unsigned int*       bottom;
    unsigned int*       head;
    unsigned int*       tail; /* tail & 0x3 == 1: 書き込み中断中、
                                 MB_FIFO_RETRY で再開 */
};
#define MB_FIFO         struct Mb_fifo
#ifdef ultra
struct Mb_fifo32{
    unsigned int        top; /* 0 or 0xffffffff : 使用中断 */
    unsigned int        bottom;
    unsigned int        head;
    unsigned int        tail; /* tail & 0x3 == 1: 書き込み中断中、
                                 MB_FIFO_RETRY で再開 */
};
#define MB_FIFO32               struct Mb_fifo32
```

```
#endif
```

### 2.1.4 MB_SIGNAL 構造体

*EMB_SIGNAL* 構造体は表 *26* の情報を持つ。*SIGNAL* 構造体の初期化は *mb_signal_init_()* システムコールを

表 26: MB_SIGNAL 構造体

| 型 | 名前 | 説明 |
|---|---|---|
| unsigned int | type | シグナルタイプ、データ領域タイプ |
| unsigned int | freq | 呼び出し条件 |
| int (*dsp)() | dsp | Destination Specified Program |
| int (*uls)() | uls | User Level Scheduler |
| unsigned | address | 呼び出すプログラムへの引数データ (MB_FIFO へのポインタかバッファ領域の論理アドレス) |

使う。

*MB_SIGNAL* 構造体 は `mbcf.h` にて以下のように宣言される。

```
struct Mb_signal{
    unsigned int        type;
    unsigned int        freq;
    int                 (*dsp)();       /* Destination Specified Program */
    int                 (*uls)();
    unsigned int        address;
    int                 count;          /* for DEBUG */
    int                 last_node;      /* for DEBUG */
    unsigned int        last_task;      /* for DEBUG */
};
#define MB_SIGNAL       struct Mb_signal

struct Mb_signal32{
    unsigned int        type;
    unsigned int        freq;
    unsigned int        dsp;    /* Destination Specified Program */
    unsigned int        uls;
    unsigned int        address;
    int                 count;          /* for DEBUG */
    int                 last_node;      /* for DEBUG */
    unsigned int        last_task;      /* for DEBUG */
};
#define MB_SIGNAL32     struct Mb_signal32
```

## 2.2 システムコール等

### 2.2.1 mbcf 送信システムコール (mbcf_send)

*PBox* 構造体を使って *MBCF* コマンドを送信するシステムコール。このシステムコールは表 *27* のように使われる。

表 27: mbcf_send() システムコール

| 項目 | Align | 説明 |
|---|---|---|
| 宣言 | | `sss_mbsend_pbox(MBCF_PBOX *pbox,int *flagp)` |
| 引数 | | |
| `MBCF_PBOX* pbox` | | 送信する *PBox* 構造体へのポインタ |
| `int *flagp` | 4 | STAT 付きコマンドを使う場合の状態返答の受信アドレス |
| 戻り値 | | |
| 正 | | 送信成功; 送信したパケット長が返る |
| -1 | | 送信処理の失敗; ユーザは再度送信を実行する |
| -2 | | 送信相手先が停止中 |
| -3 | | combine packet 処理中のエラー |
| -4 | | pbox の len が負 |
| -5 | | 自タスクへのアクセス設定が write が不許可であるのに STAT 返答 (flagp) が指示された |
| -6 | | pinfo に存在しない ltask が指定された |
| -7 | | 自ノード (物理ノード番号が負) への送信を自タスク (ltask が-1) 以外のタスクに対して実行しようとした |
| -8 | | 送信デバイス (dtlnk) が見つからない |
| -9 | | 送信デバイスの MAC アドレスが未設定 |
| -10 | | 送信デバイスのポート (Port) が見つからない |
| -11 | | エラー (今のところ使われていない) |
| -12 | | 送信処理中の異常エラー (バグ?) |
| -13 | | カーネル用コマンドが非カーネルタスクから送信しようとした |
| -14 | | 不正 MBCF コマンドが指定された |

*SSS–PC (UltraSparc 版)* ではこのシステムコールを使わなくても、次のアセンブラーコードでユーザプログラムは *PBox* 構造体を使って *MBCF* コマンドを送信できる。

`MBCF_WRITE` コマンドなど、`STAT` 付きコマンドをつかわない場合。

```
asm volatile ("mov %1, %%o0; mov %2, %%o1; ta 0x21; mov %%o0, %0"
              : "=r" (result)
              : "r" (&pbox), "r" (0) : "%o0", "%o1");
```

`MBCF_WRITE_STAT` コマンドなど、`STAT` 付きコマンドをつかって
`flags` にコマンド実行結果を受信する場合。

```
asm volatile ("mov %1, %%o0; mov %2, %%o1; ta 0x21; mov %%o0, %0"
```

```
: "=r" (result)
: "r" (&pbox), "r" (&flags) : "%o0", "%o1");
```

### 2.2.2 mb_fifo_init() システムコール

*MBCF_FIFO* 系コマンドで利用する *FIFO* 構造体を初期化する。このシステムコールは表 *29* のように使われる。

表 28: mb_fifo_init() システムコール

| 項目 | Align | 説明 |
|---|---|---|
| 宣言 | | `int mb_fifo_init(fifop,top,size)` |
| 引数 | | |
| `MB_FIFO* fifop` | 16 | MB_FIFO へのポインタ |
| `ubyte4* top` | 4 | FIFO データエリアの先頭 |
| `ubyte4 size` | 4 | FIFO データエリアの大きさ |
| 戻り値 | | |
| 0 | | 正常終了 |
| -1 | | 異常終了、アドレスまたはサイズが正しくない |

実際の宣言は `mbcf.h` にて次のように行われる。

```
int mb_fifo_init();
/*
  int   mb_fifo_init(fifop,top,size)

  MB_FIFO*      fifop;  MB_FIFO へのポインタ
  ubyte4*       top;    FIFO データエリアの先頭
  ubyte4        size;   FIFO データエリアの大きさ

  返り値:       0; 正常終了
                -1; 異常終了、アドレスまたはサイズが正しくない
*/
```

### 2.2.3 mb_fifo_init_with_type() システムコール

*MBCF_FIFO* 系コマンドで利用する *FIFO* 構造体を初期化する。このシステムコールは表 *29* のように使われる。*mb_fifo_init()* システムコールに一つタイプ引数が追加されている。*MBCF_ENROLL_ON_DEMAND* を実現するのにあたり追加されたシステムコール。

表 29: mb_fifo_init() システムコール

| 項目 | Align | 説明 |
|---|---|---|
| 宣言 | | `int mb_fifo_init(fifop,top,size)` |
| 引数 | | |
| `MB_FIFO* fifop` | 16 | MB_FIFO へのポインタ |
| `ubyte4* top` | 4 | FIFO データエリアの先頭 |
| `ubyte4 size` | 4 | FIFO データエリアの大きさ |
| `ubyte4 type` | 4 | 2:ENROLL on DEMAND 0:NORMAL |
| 戻り値 | | |
| 0 | | 正常終了 |
| -1 | | 異常終了、アドレスまたはサイズが正しくない |

実際の宣言は `mbcf.h` にて次のように行われる。

```
nt mb_fifo_init_with_type();
/*
  int mb_fifo_init_with_type(fifop,top,size,type)

  MB_FIFO* fifop;  MB_FIFO へのポインタ
  ubyte4* top; FIFO データエリアの先頭
  ubyte4 size; FIFO データエリアの大きさ
  ubyte4 type; type=2: ENROLL on DEMAND, type=0: NORMAL

  返り値: 0; 正常終了
               -1; 異常終了、アドレスまたはサイズが正しくない
*/
```

### 2.2.4 mb_fifo_read() システムコール

*MBCF_FIFO* 系コマンドで受信した *FIFO* データを読み出す。このシステムコールは表 *30* のように使われる。

表 30: mb_fifo_read() システムコール

| 項目 | Align | 説明 |
|---|---|---|
| 宣言 | | `int mb_fifo_read`<br>`(fifop, ltask_p, src_node_p, src_task_p, buf, size)` |
| 引数 | | |
| `MB_FIFO* fifop` | | MB_FIFO へのポインタ |
| `int* ltask_p` | | writer の ltask を書き戻すポインタ |
| `int* src_node_p` | | writer の node_id を書き戻すポインタ |
| `OBJECT_NAME* src_task` | | writer の task 物理 name を書き戻すポインタ |
| `ubyte4* buf` | | データを受け取るバッファ |
| `ubyte4 size` | | データを受け取るバッファのサイズ |
| 戻り値 | | |
| 正, 0 | | データ長、正常終了 |
| -1 | | FIFO が空である |
| -2 | | FIFO が使用可能になっていない |
| -3 | | 読み出しバッファのサイズが足りない（データは FIFO に残る） |

実際の宣言は `mbcf.h` にて次のように行われる。

```
int mb_fifo_read();
/*
  int mb_fifo_read(fifop, ltask_p, src_node_p, src_task_p, buf, size)

  MB_FIFO*      fifop;  MB_FIFO へのポインタ
  int*          ltask_p; writer の ltask を書き戻すポインタ
  int*          src_node_p; writer の node_id を書き戻すポインタ
  OBJECT_NAME*  src_task; writer の task 物理 name を書き戻すポインタ
  ubyte4*       buf; データを受け取るバッファ
  ubyte4        size; データを受け取るバッファのサイズ

  返り値:       データ長; 正常終了
                -1; FIFO が空である
                -2; FIFO が使用可能になっていない
                -3; 読み出しバッファのサイズが足りない（データは FIFO に残る）
*/
```

### 2.2.5 mb_fifo_read_offset() システムコール

*MBCF_FIFO* 系コマンドで受信した *FIFO* データを、*FIFO* 構造体の *FIFO* データ領域からオフセット加算したアドレスから読み出す。このシステムコールは表 *31* のように使われる。

表 31: mb_fifo_read_offset() システムコール

| 項目 | Align | 説明 |
|---|---|---|
| 宣言 | | `int mb_fifo_read_offset`<br>`(fifop, buf, size, offset)` |
| 引数<br>`MB_FIFO* fifop`<br>`ubyte4* buf`<br>`ubyte4 size`<br>`int offset` | | <br>MB_FIFO へのポインタ<br>データを受け取るバッファ<br>データを受け取るバッファのサイズ<br>FIFO ポインタを修正する値 (バイト数)<br>FIFO 構造体を異なった論理アドレスで共有できる。 |
| 戻り値<br>正, 0<br>-1<br>-2<br>-3 | | <br>データ長; 正常終了<br>FIFO が空である<br>FIFO が使用可能になっていない<br>読み出しバッファのサイズが足りない（データは FIFO に残る） |

実際の宣言は `mbcf.h` にて次のように行われる。

```
int mb_fifo_read_offset();
/*
  int mb_fifo_read_offset(fifop, buf, size, offset)

  MB_FIFO*      fifop;  MB_FIFO へのポインタ
  unsigned int* buf; データを受け取るバッファ
  unsigned int  size; データを受け取るバッファのサイズ
  int           offset; FIFO ポインタを修正する値
                        FIFO 構造体を異なった論理アドレスで共有できる。

  返り値:       データ長; 正常終了
                -1; FIFO が空である
                -2; FIFO が使用可能になっていない
                -3; 読み出しバッファのサイズが足りない（データは FIFO に残る）
*/
```

### 2.2.6 mb_signal_init() システムコール

*MBCF_SIGNAL* 系コマンドで使用する *MB_SIGNAL* 構造体を初期化するシステムコール。このシステムコールは表 *32* のように使われる。

表 32: mb_signal_init() システムコール

| 項目 | Align | 説明 |
|---|---|---|
| 宣言 | | `int mb_signal_init(sigp,type,freq,dsp,uls,address)` |
| 引数 | | |
| `MB_SIGNAL * sigp` | 32 | MB_SIGNAL へのポインタ |
| `ubyte4 type` | | MB_SIGNAL の type 項 |
| `ubyte4 freq` | | MB_SIGNAL の freq 項 |
| `int (*dsp)()` | 4 | MB_SIGNAL の dsp 項<br>Destination Specified Program のエントリアドレス |
| `int (*uls)()` | 4 | MB_SIGNAL の uls 項<br>User Level Scheduler のエントリアドレス |
| `ubyte4 address` | 4 | パケットのパラメータデータを格納するキューもしくは<br>メモリのアドレス（少なくとも 4byte 境界） |
| 戻り値 | | |
| 0 | | 正常終了 |
| -1 | | 異常終了、アドレスまたは type/freq の値が正しくない |

実際の宣言は `mbcf.h` にて次のように行われる。

```
int mb_signal_init();
/*
  int mb_signal_init(sigp,type,freq,dsp,uls,address)

  MB_SIGNAL*    sigp; MB_SIGNAL へのポインタ
  ubyte4        type; MB_SIGNAL の type 項
  ubyte4        freq; MB_SIGNAL の freq 項
  int           (*dsp)(); MB_SIGNAL の dsp 項
                          Destination Specified Program のエントリアドレス
  int           (*uls)(); MB_SIGNAL の uls 項
                          User Level Scheduler のエントリアドレス
  ubyte4        address; パケットのパラメータデータを格納するキューもしくは
                         メモリのアドレス（少なくとも 4byte 境界）

  返り値:       0; 正常終了
                -1; 異常終了、アドレスまたは type/freq の値が正しくない
*/
```

### 2.2.7 mb_signal_flag_reset() システムコール

*MBCF_SIGNAL* 系コマンドで使用する *MB_SIGNAL* 構造体のパラメータデータの格納アドレスを変更するシステムコール。このシステムコールは表 *33* のように使われる。

表 33: mb_signal_flag_reset() システムコール

| 項目 | 説明 |
|---|---|
| 宣言 | `int mb_signal_flag_reset(sigp,offset)` |
| 引数 | |
| `MB_SIGNAL*` `sigp`<br>`ubyte4` `offset` | MB_SIGNAL へのポインタ<br>パケットのパラメータデータを格納するメモリアドレス<br>のオフセット（キューでは使用しない、少なくとも 4byte 境界） |
| 戻り値<br>0<br>-1<br>-2 | <br>正常終了<br>reset 前に新たなパラメータエントリが追加された<br>異常終了、アドレスまたは type/freq の値が正しくない |

実際の宣言は `mbcf.h` にて次のように行われる。

```
int mb_signal_flag_reset();
/*
  int mb_signal_flag_reset(sigp,offset)

  MB_SIGNAL*    sigp; MB_SIGNAL へのポインタ
  ubyte4        offset; パケットのパラメータデータを格納するメモリアドレス
                        のオフセット（キューでは使用しない、少なくとも 4byte 境界）

  返り値:       0; 正常終了
                -1; reset 前に新たなパラメータエントリが追加された
                -2; 異常終了、アドレスまたは type/freq の値が正しくない
*/
```

### 2.2.8 mbcf_sigblock() システムコール

実際の宣言は mbcf.h にて次のように行われる。

```
int
mbcf_sigblock(mask)
unsigned int mask;
{
        _trap_syscall_1( TASK_SELF_MBSIGBLOCK,ul mask );
}

int
mbcf_sigunblock(mask)
unsigned int mask;
{
        _trap_syscall_1( TASK_SELF_MBSIGUNBLOCK,ul mask);
}

int
mbcf_sigunblock2(mask,name)
unsigned int mask;
OBJECT_NAME name;
{
        _trap_syscall_2( TASK_MBSIGUNBLOCK,ul mask, ul name);
}
```

### 2.2.9 Target task Specified Program (DSP)

*MB_Signal* で呼び出される関数。*mb_signal_init()* システムコール *(2.2.6* 節参照*)* で *MB_SIGNAL* 構造体に設定する。次のような仕様の関数に定義されている。

```
static int user_invoke_program(dataopp, sigp, arg0)
char*        dataopp;
MB_SIGNAL*       sigp;
int              arg0;
```

**2.2.9.1 MBCF SIGNAL ハンドラのサンプル** 以下が *FIFO* バッファと *DSP* を使う *MBCF SIGNAL* ハンドラの例である。

```
static int rmtdsp(quep, sigp, arg0)
MB_FIFO*         quep;
MB_SIGNAL*       sigp;
int              arg0;
{
int rc;
    int ltask, pnode, ptask;
    int reqbuf[16];

 more_req:
    rc = mb_fifo_read(quep, &ltask, &pnode, &ptask, &reqbufp[0], sizeof(reqbuf));

    switch (arg0) {
    default:
        printf(``arg0 の値や rc(受信サイズ) の値にしたがって処理を分岐する\n'');
    }

    /* さらに次の MBCF_SIGNAL が受信されていないかチェックする. */
    rc = mb_signal_flag_reset(sigp, 0, &arg0);
    if (rc == -1) goto more_req;

    /* カーネルでは帰り値は無視される. */
    return rc;
}
```

# 3 MBCF の使用例

本節では *SSS–PC* の *MBCF* の使用例として *MBCF_WRITE* コマンドの例を示す。次にあげるのは並列実行レイトレーシングプログラムプログラムコードからの抜粋である。一つのレイトレーシング結果画像の画面表示タスクがその他のレイトレーシング計算タスクを使ってレイトレーシングを並列計算する工程を、コマンドラインで指定された回数だけループするプログラムである。ループ毎に全計算タスクを同期させ、次のレイトレーシング計算パラメータを送信してから計算が実行される。全計算タスクの計算が終了し、表示タスクがモニタ画面にレイトレーシングの結果画像を表示おわると次のループに継続する。

```
/*
 *  MBCF で処理されるグローバル変数。
 */
volatile int mynode;
volatile int    rtbuf_f = 0;
volatile RT_ARG rtbuf; /* レイトレーシングの入力パラメータ */

int
main(ac,av)
int     ac;
char**  av;
{
        int     i, j, k, n, id;
        int     loop_count;
        int     loop_count0;
        int     wait_time;
        int     loop_wait;
        char    *tmp_cp,*endp;
        unsigned mykey;
        int     disp_task,disp_node;
        volatile L_TASKINFO     *tinfop;
        RT_ARG  rt;
        volatile RT_ARG *t = &rtbuf;
        int     buf[32];
        int     flg[MAX_HOST];
        volatile int    end_flg[MAX_HOST];
        MBCF_PBOX       pbox;
        VSCRN   vs, *v = &vs;
        int     endflag = 0;
        int     stime, etime;

        /*
         *  [ 略 ]
         *    コマンドラインオプションの解析、変数の初期化、メモリの確保、
```

```
 *     などプログラム開始時点のコードがここに入る。
 */

if (id == 0) { /* display node */

    /*
     * 表示ノードの場合はこちらを実行する。
     */

    /* 変数の初期化 */
    for(i = 0; i <= n; ++i){
        flg[i] = -2;
        end_flg[i] = 0;
    }

    /* モニタ登録 */
    if((i = mon_setapp(" V-Ray-Trace ")) == -1 || i == -2) {
        printf("Can't get Mon3-Scrn\n");
    } else {
        /* printf("Get Mon3-Scrn No.%d\n",i); */
    }

    /* モニタへの問い合わせ */
    if ((i = mon_get_vscrndata(RAY_TRACE,buf,sizeof(buf))) < 0) {
        printf("%s: error mon_getparam('D')\n",av[0]);
        return 2;
    }
    self_sleep(10); /* mon3 による画面クリアのための時間 */

    /*
     * リモートへの計算開始要求パケットの作成
     */
    rt.r_task   = buf[0];
    rt.r_node   = get_pnid();
    rt.r_key    = buf[1];
    rt.s_fb     = buf[3];
    rt.s_type   = buf[11];
    rt.s_linebytes  = buf[4];
    rt.s_width  = buf[10];
    rt.s_height = buf[5];
    rt.v_xpos   = buf[6];
    rt.v_ypos   = buf[7];
    rt.v_width  = buf[8];
```

```
    rt.v_height = buf[9];
    rt.seg_siz  = buf[RAY_PACKSIZ];
    rt.model    = buf[RAY_MODEL];
    rt.shadow   = buf[RAY_SHADOW];

    /* リモートへパケットを送る */
    stime = sys_gettime();

    /*
     *  自タスクのキーを取得する。
     */
    mykey = get_accesskey();

    /*
     *  計算ノードに計算内容(rtbuf)を送信するために、
     *  pbox にパラメータを設定する。
     */
    pbox.key = mykey;
    pbox.comm   = MBCF_WRITE_F;
    pbox.len    = sizeof(rt);
    pbox.addr   = (unsigned)&rtbuf;
    pbox.addr2  = (unsigned)&rtbuf_f;
    pbox.addr3  = 1;  /* flag value */
    pbox.data   = &rt;

/*
 *  レイトレーシングをループ実行する。
 */
for (;;) {

    for(i = 1; i <= n; ++i){
        rt.peid     = i - 1;
        if (buf[2] != 0)
            rt.r_stat = buf[2]/* + sizeof(int)*i */;
        else
            rt.r_stat = 0;
        rt.mig_id = (loop_count % n)+1;
        rt.endf_addr = (int) &end_flg[i];
        /*
         *  論理タスク番号[i]の計算タスクに
         *  計算パラメータを送信する。
         */
        pbox.ltask = i;
```

```
    do {
      int ret;
      /*
       *  計算内容の送信。失敗したら１秒待ってから再送する。
       */
      ret = mbcf_send(&pbox,0);
      if (ret != -1) break;
      self_sleep(1);
    } while (1);
}

/*
 *  全計算タスクで計算が完了するまで待つ。
 *
 */
for(j=0;j<10000;j++) {
    k = 1;
    for(i = 1; i <= n; ++i){
        /*
         *  一つでも計算終了していない、すなわち
         *  計算終了通知が届いていないタスクがあれば
         *  最初からチェックを実行しなおす。
         */
        if (end_flg[i] != 1) k = 0;
    }
    if (k == 1) break;
    else self_sleep(10);
}
loop_count--;
if (loop_count <= 0) break;
/*
 *  ループして再度レイトレーシングを実行するため、
 *  全計算タスクの計算完了ステータスをクリアする。
 */
for(i = 1; i <= n; ++i) end_flg[i] = 0;
{
    int  wait_tick,start_tick,end_tick;
    wait_tick = wait_time * 1000 / SSSMC_TIMER_UNIT;
    start_tick = sys_gettime();
    for(;;) {
        end_tick = sys_gettime();
        if ((end_tick - start_tick) >= wait_tick) break;
        self_sleep(0);
```

```
        }
    }

    /*
     *  次のループではレイトレーシングする三次元モデルを変更する。
     */
    rt.model++;
    if (rt.model == 2) rt.model = 3;
    if (rt.model == RT_MAX_MODEL) rt.model = 0;
 }
    etime = sys_gettime();
    printf("%s: time=%d (%d-%d)\n",av[0],etime-stime,etime,stime);
    if (k == 1) return 0;
    return -1;
} else { /* id != 0 */
    /*
     * 計算ノードの場合はこちらを実行する。
     */

    init_suntime();
    mynode = get_pnid();
    printf("vrt@node%d id=%d start.\n",mynode,id);
    self_sleep(1);
    self_setslice(1);   /* コンテキストはゆっくりと切り変える */
    color_init();

    /*
     *  ループ回数までレイトレを実行する。
     */
    for(;;){
        int  wait_tick,start_tick,end_tick;
        wait_tick = RT_MAX_WAIT_TIME * 1000 / SSSMC_TIMER_UNIT;
        start_tick = sys_gettime();

        /*
         *  画面表示ノードの準備が完了するのを待つ。
         *  rtbuf_f は画面表示ノードからの MBCF_WRITE_F コマンド実行
         *  により更新される。
         */
        for(;;) {
            if(rtbuf_f == 1) break;
            end_tick = sys_gettime();
            if ((end_tick - start_tick) >= wait_tick) self_end(-1);
```

```
    self_sleep(0);
}
rtbuf_f = 0;

/*
 *  自タスクのキーを取得する。
 */
mykey = get_accesskey();
/*
 *  表示ノードの論理タスク番号を設定しなおす。
 */
disp_node = t->r_node;
disp_task = set_newtask(disp_node,t->r_task);
vs_init(v,0,0,t->v_width,t->v_height,0);

/*
 * レイトレーシングの計算実行。
 * mon3 モニタープログラムに自タスクの状態を通知する。
 */
if (loop_count == loop_count0)
    /* RT_STAT_RUN: 実行中 */
    et_rt_mon2(t,RT_STAT_RUN,disp_task);
else
    et_rt_sub2(t,v,t->shadow,t->model,disp_task,0);
if (loop_count == 1)
    /* RT_STAT_IDLE: 休止中 */
    et_rt_mon2(t,RT_STAT_IDLE,disp_task);

/*
 *  表示ノードに計算終了を通知する。
 *  表示ノードの終了通知フラグ t->endf_addr へむけて
 *  endflag (==1) の内容を書き込む。
 */
endflag = 1;
if (t->endf_addr != 0) {
    pbox.ltask = 0;
    pbox.key = mykey;
    pbox.addr = (ubyte4)t->endf_addr;
    pbox.data = (void*)&endflag;
    pbox.len  = sizeof(endflag);
    for(;;){
        int r;
        /*
```

```
                        *  計算終了通知の送信。失敗したら１秒待って
                        *  から再送する。
                        */
                       r = write_taskdata(&pbox,0);
                       if(r != -1) break;
                       self_sleep(1);
                   }
               }
               /*
                *  表示ノード論理タスクをこのタスクのタスクリストから削除する。
                */
               del_task(disp_task);
               loop_count--;
               if (loop_wait > 0) loop_wait--;
               if (loop_count <= 0) break;
           }
           return 0;
       }
}
```

# 4 MDIR:メモリベース・ディレクトリ機構

## 4.1 概要

本節では *SSS–PC* に実装された *MDIR(*メモリベースディレクトリ機構*)* について述べる。メモリベースディレクトリ機構はクラスタ全体で一意な情報を管理する仕組みを提供します。

**4.1.0.2 IDM(情報開示機構) と経緯** 情報開示機構は元々、システム（メタ）情報開示機構とタスク情報開示機構の二種類が考えられていました。しかし、*SSS–PC* では論理タスク *ID* によってタスク間通信が自由に行われるため、アプリケーション固有の情報開示は *OS* が特別に用意しなくてもよく、システム情報開示機構のみが実装されています。*pstart* によって起動される並列タスクはまさにシステム情報開示機構のみで十分です。

しかし、*pstart* せずに起動されるタスク間で論理タスク *ID* を割り振るためには、*send_taskque* によって少なくとも一方が他方のタスクキューにエントリ（通信）する必要があります。この時のタスクを指定するためにはどうしてもクラスタ全体で一意な指定方法が必要となります。クラスタレベルの *port mapper* や *NIS* に相当するものと言えるでしょう。*MBCF* 考案当初はこれを *TCP/IP* か *UDP/IP* 上に作れば良いと考えていましたが、*SSS-PC* との相性や性能、プログラミング容易性と言った点を考えるとやはり *MBCF* を使った機構として実装する方がよいといえます。そこで実装されたのが、*MDIR(Memory-based directory)* 機構です。

**4.1.0.3 MDIR とは何か** *MDIR(Memory-based directory)* 機構は、*4byte* および *8byte* のクラスタ *ID*（ただし、従来のオブジェクト名との互換性は *4byte* のクラスタ *ID* のみ取られる）と任意の *255* 文字以下の文字列をディレクトリのキーとして使用可能です。クラスタ *ID* は各ノードレベルで一部情報（現在の本体が存在するノードと物理タスク名および状態）をキャッシュ可能です。このキャッシュ機能を使ってクラスタ *ID* と *MBCF* もしくは *MDIR* 機構によって高速に通信やクラスタレベルのシステムコールを実現することができます。

クラスタ *ID* は生成時のオーバヘッドコストを下げるために、*ID* 内に生成された物理ノード番号を含んでいます。マイグレーションによってオーナー（生成タスク）が移動するとクラスタ *ID* の本籍も移動しますが、*ID* 自体は不変です。よって、マシンが再起動してきた（もしくは交換された）場合には、最初に自分の物理ノード番号を持つクラスタ *ID* をクラスタ全体から収集します。そして、後で生成（もしくは登録）されるクラスタ *ID* の一意性を保証します。文字列キーに関しては、生成時にクラスタ全体に問い合わせて、同じキーが存在しないことを確認してから登録することにより一意性を保証しています。なお、文字列キーの文字情報自体はカーネル内ではなく生成タスク内に保持されています。クラスタ *ID* および文字列キーのディレクトリ内のエントリは生成（もしくは登録）タスクが消滅した時には消去されます。このコンベンションはタスクによって生成されたオブジェクト（子タスクやキュー）と同じです。

クラスタ *ID* および文字列キーは登録時に生成元（登録元）タスク内のメモリと関連づけることができます。生成元タスクはこのメモリに自由に公開したい関連情報を記録しておくことができます。この方式にちなんで *Memory-based directory* と命名しました。

**4.1.0.4 MDIR の特徴** 簡単にまとめると、*SSS–PC* の *MDIR* には以下のような特徴があります。

- クラスタ内ユニークな *ID* を管理できます
- *4* バイト値、*8* バイト値または *255* 文字以下の文字列をキーとして *128KB* 未満の情報と関連ずけられます
- クラスタ *ID* の場合、各ノードレベルで一部情報（現在の本体が存在するノードと物理タスク名および状態）をキャッシュ可能です
- 原則として、*PBOX(MDIR_PBOX)* を利用したシステムコールで操作を行います

## 4.2 MDIR コマンド体系

### 4.2.1 MDIR コマンド一覧

*SSS–PC* では表 *34* のコマンドが利用可能である。

表 34: MDIR コマンド一覧 (その一)

| コマンド名 | コード | 説明 |
|---|---|---|
| MDIR_NOP | 0 | |
| MDIR_SET_U4 | 1 | |
| MDIR_SET_U8 | 2 | |
| MDIR_SET_STR | 3 | |
| MDIR_DEL_ALL | 4 | |
| MDIR_DEL_U4 | 5 | |
| MDIR_DEL_U8 | 6 | |
| MDIR_DEL_STR | 7 | |
| MDIR_GET_U4 | 9 | |
| MDIR_GET_U8 | 10 | |
| MDIR_GET_STR | 11 | |
| MDIR_QUERY_U4 | 13 | |
| MDIR_QUERY_U8 | 14 | |
| MDIR_QUERY_STR | 15 | |
| MDIR_OPEN_U4 | 17 | |
| MDIR_OPEN_U8 | 18 | |
| MDIR_OPEN_STR | 19 | |
| MDIR_CLOSE_ALL | 20 | |
| MDIR_CLOSE_U4 | 21 | |
| MDIR_CLOSE_U8 | 22 | |
| MDIR_CLOSE_STR | 23 | |
| MDIR_STOP_ALL | 24 | |
| MDIR_STOP_U4 | 25 | |
| MDIR_STOP_U8 | 26 | |
| MDIR_STOP_STR | 27 | |
| MDIR_RESUME_ALL | 28 | |
| MDIR_RESUME_U4 | 29 | |
| MDIR_RESUME_U8 | 30 | |
| MDIR_RESUME_STR | 31 | |

表 35: MDIR コマンド一覧 (その二)

| コマンド名 | コード | 説明 |
|---|---|---|
| MDIR_GOPUBLIC_U4 | 33 | |
| MDIR_GOPUBLIC_U8 | 34 | |
| MDIR_RmtQUERY_U4 | 37 | |
| MDIR_RmtQUERY_U8 | 38 | |
| MDIR_UPDATE_U4 | 41 | |
| MDIR_UPDATE_U8 | 42 | |
| MDIR_UPDATE_STR | 43 | |
| MDIR_RmtGET_U4 | 45 | |
| MDIR_RmtGET_U8 | 46 | |
| MDIR_ALLOC_MEM | 100 | |
| MDIR_FREE_MEM | 101 | |

### 4.2.2 MDIR_SET_U4, MDIR_SET_U8, MDIR_SET_STR

#### 4.2.2.1 コマンドコード

*1* *MDIR_SET_U4*
*2* *MDIR_SET_U8*
*3* *MDIR_SET_STR*

**4.2.2.2 機能** ディレクトリのエントリを新規に作成する。

**4.2.2.3 入力** *pbox*

表 36: MDIR_PBOX 構造体 (SET)

| 名前 | 説明 |
|---|---|
| comm | 1,2,3 |
| subcomm | 0x100:**open_flag**, 0x200:**stop_flag** |
| arg[2] | U4,U8 での CLID、0 なら自動生成される |
| namep | STR の場合の文字列キーの先頭論理アドレス |
| len | キーに関連づけるメモリ領域のバイト数 |
| data | キーに関連づけるメモリ領域の先頭論理アドレス |

**4.2.2.4 オペレーション**

**4.2.2.5 出力** なし

**4.2.2.6 sss_mdir_request() の戻り値**

*-1* メモリ空き領域不足
*-2* 指定された *ID* は登録済
*-3* エントリの登録領域不足
*>0* 成功 *(*送信バイト数*)*

**4.2.2.7 説明** ディレクトリのエントリを新規に作成する。*U4,U8* は *arg[0], arg[1]* に *0* を指定すると、適当なクラスタ *ID* を *OS* が生成する。その場合は *sss_mdir_request()* の第二引数に生成された *ID* を受け取る領域へのポインタを指定する。*data* と *len* で *MDIR_ALLOC_MEM* によって取られた領域を渡すことでクラスタ *ID* （もしくは文字列キー）とメモリ領域を関連づけられる。*data=0* で生成後、*MDIR_UPDATE_**でメモリ領域を後から指定することが可能。また、*subcomm* で生成時のエントリの *type* の **open_flag***(0x100)* および **stop_flag***(0x200)* の状態を指定できる。

### 4.2.3 MDIR_DEL_ALL,MDIR_DEL_U4, MDIR_DEL_U8, MDIR_DEL_STR

#### 4.2.3.1 コマンドコード

*4* *MDIR_DEL_ALL*
*5* *MDIR_DEL_U4*
*6* *MDIR_DEL_U8*
*7* *MDIR_DEL_STR*

**4.2.3.2 機能** 指定された *ID* およびディレクトリのエントリを消去する

**4.2.3.3 入力** *pbox*

表 37: MDIR_PBOX 構造体 (DEL)

| 名前 | 説明 |
|---|---|
| comm | 4,5,6,7 |
| subcomm | – |
| arg[2] | U4, U8 の CLID |
| namep | STR の場合の文字列キーの先頭論理アドレス |
| len | – |
| data | – |

**4.2.3.4 オペレーション**

**4.2.3.5 出力** なし

**4.2.3.6 sss_mdir_request() の戻り値**

*0* 成功
*<0* 失敗

**4.2.3.7 説明** *MDIR_DEL_ALL* はタスクがそれまでに登録したディレクトリのエントリおよびクラスタ *ID* と文字列キーをすべて消去する。なお、エントリ内に指定された内容（*contents_ptr* が指す領域）メモリも解放される。

*MDIR_DEL_U4*、*MDIR_DEL_U8*、*MDIR_DEL_STR* はそれぞれ *arg[0], arg[0]+arg[1], namep* で指定された *ID* およびエントリを消去する。他人が登録したものは消去できない。なお、エントリ内に指定された内容（*contents_ptr* が指す領域）メモリも解放される。

#### 4.2.4 MDIR_GET_U4, MDIR_GET_U8, MDIR_GET_STR

##### 4.2.4.1 コマンドコード

*8* *MDIR_GET_U4*
*9* *MDIR_GET_U8*
*10* *MDIR_GET_STR*

**4.2.4.2 機能** クラスタ *ID*（またはキー）と関連づけられたディレクトリの内容を読み出す

**4.2.4.3 入力** *pbox*

表 38: MDIR_PBOX 構造体 (GET)

| 名前 | 説明 |
|---|---|
| comm | 9,10,11 |
| subcomm | – |
| arg[2] | U4, U8 の CLID |
| namep | STR の場合の文字列キーの先頭論理アドレス |
| len | 受信パラメータのバイト長さ |
| data | 受信領域へのポインタ (先頭アドレス) |

**4.2.4.4 オペレーション**

**4.2.4.5 出力** クラスタ *ID*（またはキー）と関連づけられたディレクトリの内容を読み出す

**4.2.4.6 sss_mdir_request() の戻り値**

*>=0* 成功 *(*受信したバイト数*)*
*-1* エラー
*-2* **open_flag** が設定されていない
*-3* *(*未使用？*)*
*-4* *(*未使用？*)*
*-5* *(*エントリを保持しているタスクが存在していない？*)*
*-6* *(*アクセス違反？*)*
*-7* **stop_flag** が設定されいた
*-100* 指定された *ID* は未登録
*-200* **GOPUBLIC** されていなくてノードが異なる

**4.2.4.7 説明** *MDIR_GET_U4*、*MDIR_GET_U8*、*MDIR_GET_STR* は、*data* と *len* で指定されたタスク内のメモリ領域にクラスタ *ID*（またはキー）と関連づけられたディレクトリの内容を読み出す。ただし、クラスタ *ID* の場合は **GOPUBLIC** されていない場合はノードが異なる場合はエラーになる。その場合は *MDIR_RmtGET_** を使用する。*U4* または *U8* で **GOPUBLIC** されている場合は、リモートにアクセスして認証に問題がなければ内容を読み出す。

### 4.2.5 MDIR_QUERY_U4, MDIR_QUERY_U8, MDIR_QUERY_STR

#### 4.2.5.1 コマンドコード

*13* *MDIR_QUERY_U4*
*14* *MDIR_QUERY_U8*
*15* *MDIR_QUERY_STR*

#### 4.2.5.2 機能 クラスタ *ID*（または文字列キー）の情報を読み出す

#### 4.2.5.3 入力 *pbox*

表 39: MDIR_PBOX 構造体 (QUERY)

| 名前 | 説明 |
|---|---|
| comm | 13,14,15 |
| subcomm | – |
| arg[2] | U4, U8 の CLID |
| namep | STR の場合の文字列キーの先頭論理アドレス |
| len | CL_QUERY_ENTRY 構造体のバイト長さ |
| data | CL_QUERY_ENTRY 構造体へのポインタ (先頭アドレス) |

#### 4.2.5.4 オペレーション

#### 4.2.5.5 出力 *CL_QUERY_ENTRY* 構造体の形式でクラスタ *ID*（または文字列キー）の情報

#### 4.2.5.6 sss_mdir_request() の戻り値

*>0* 成功 *(*受信したバイト数*)*
その他 失敗

**4.2.5.7 説明** *MDIR_QUERY_U4*、*MDIR_QUERY_U8*、*MDIR_QUERY_STR* は、*CL_QUERY_ENTRY* 構造体としてクラスタ *ID*（または文字列キー）の情報を *data* と *len* で指定された領域に読み出す。ただし、クラスタ *ID* の場合は **GOPUBLIC** されていない場合はノードが異なる場合はエラーになる。その場合は *MDIR_RmtQUERY_**を使用する。*U4* または *U8* で **GOPUBLIC** されている場合はキャッシュの内容が読み出される（リモートアクセスはない)。

#### 4.2.6 MDIR_OPEN_U4, MDIR_OPEN_U8, MDIR_OPEN_STR

##### 4.2.6.1 コマンドコード

*17* *MDIR_OPEN_U4*
*18* *MDIR_OPEN_U8*
*19* *MDIR_OPEN_STR*

**4.2.6.2 機能** 指定したクラスタ *ID*（または文字列キー）のエントリの *type* フィールドの **open_flag** をセットする

**4.2.6.3 入力** *pbox*

表 40: MDIR_PBOX 構造体 (OPEN)

| 名前 | 説明 |
|---|---|
| comm | 17,18,19 |
| subcomm | – |
| arg[2] | U4, U8 の CLID |
| namep | STR の場合の文字列キーの先頭論理アドレス |
| len | – |
| data | – |

**4.2.6.4 オペレーション**

**4.2.6.5 出力** なし

**4.2.6.6 sss_mdir_request() の戻り値**

*0* 成功
その他 失敗

**4.2.6.7 説明** *MDIR_OPEN_U4*、*MDIR_OPEN_U8*、*MDIR_OPEN_STR* は、指定したクラスタ *ID*（または文字列キー）のエントリの *type* フィールドの **open_flag** をセットする。自分が登録したエントリ以外は操作できない。**Public** なエントリに関しては操作内容が各ノードの *Cache* に伝搬する。

#### 4.2.7 MDIR_CLOSE_ALL,MDIR_CLOSE_U4, MDIR_CLOSE_U8, MDIR_CLOSE_STR

##### 4.2.7.1 コマンドコード

*20* *MDIR_CLOSE_ALL*
*21* *MDIR_CLOSE_U4*
*22* *MDIR_CLOSE_U8*
*23* *MDIR_CLOSE_STR*

**4.2.7.2 機能** 指定したクラスタ *ID*（または文字列キー）のエントリの *type* フィールドの **open_flag** をリセットする

**4.2.7.3 入力** *pbox*

表 41: MDIR_PBOX 構造体 (CLOSE)

| 名前 | 説明 |
|---|---|
| comm | 20,21,22,23 |
| subcomm | – |
| arg[2] | U4, U8 の CLID |
| namep | STR の場合の文字列キーの先頭論理アドレス |
| len | – |
| data | – |

**4.2.7.4 オペレーション**

**4.2.7.5 出力** なし

**4.2.7.6 sss_mdir_request() の戻り値**

*0* 成功
その他 失敗

**4.2.7.7 説明** *MDIR_CLOSE_ALL* は全てのエントリについて、*MDIR_CLOSE_U4*、*MDIR_CLOSE_U8*、*MDIR_CLOSE_STR* の場合は、指定したクラスタ *ID*（または文字列キー）のエントリについて、*type* フィールドの **open_flag** をリセットする。自分が登録したエントリ以外は操作できない。**Public** なエントリに関しては操作内容が各ノードの *Cache* に伝搬する。

#### 4.2.8 MDIR_STOP_ALL,MDIR_STOP_U4, MDIR_STOP_U8, MDIR_STOP_STR

##### 4.2.8.1 コマンドコード

*24* *MDIR_STOP_ALL*
*25* *MDIR_STOP_U4*
*26* *MDIR_STOP_U8*
*27* *MDIR_STOP_STR*

**4.2.8.2 機能** 指定したクラスタ *ID*（または文字列キー）のエントリの *type* フィールドの **stop_flag** をセットする

**4.2.8.3 入力** *pbox*

表 42: MDIR_PBOX 構造体 (STOP)

| 名前 | 説明 |
|---|---|
| comm | 24,25,26,27 |
| subcomm | – |
| arg[2] | U4, U8 の CLID |
| namep | STR の場合の文字列キーの先頭論理アドレス |
| len | – |
| data | – |

**4.2.8.4 オペレーション**

**4.2.8.5 出力** なし

**4.2.8.6 sss_mdir_request() の戻り値**

*0* 成功
その他 失敗

**4.2.8.7 説明** *MDIR_STOP_ALL* は全てのエントリについて、*MDIR_STOP_U4*、*MDIR_STOP_U8*、*MDIR_STOP_STR* の場合は、指定したクラスタ *ID*（または文字列キー）のエントリについて、*type* フィールドの **stop_flag** をセットする。自分が登録したエントリ以外は操作できない。**Public** なエントリに関しては操作内容が各ノードの *Cache* に伝搬する。

タスクがマイグレーションする際には必ず *STOP_ALL* し、マイグレーションを完了後ノードを移って実行を再開後に、*RESUME_ALL* しなければならない。

#### 4.2.9 MDIR_RESUME_ALL,MDIR_RESUME_U4, MDIR_RESUME_U8, MDIR_RESUME_STR

##### 4.2.9.1 コマンドコード

*28* *MDIR_RESUME_ALL*
*29* *MDIR_RESUME_U4*
*30* *MDIR_RESUME_U8*
*31* *MDIR_RESUME_STR*

**4.2.9.2 機能** 指定したクラスタ *ID*（または文字列キー）のエントリの *type* フィールドの **open_flag** をリセットする

**4.2.9.3 入力** *pbox*

表 43: MDIR_PBOX 構造体 (RESUME)

| 名前 | 説明 |
|---|---|
| comm | 28,29,30,31 |
| subcomm | – |
| arg[2] | U4, U8 の CLID |
| namep | STR の場合の文字列キーの先頭論理アドレス |
| len | – |
| data | – |

**4.2.9.4 オペレーション**

**4.2.9.5 出力** なし

**4.2.9.6 sss_mdir_request() の戻り値**

*0* 成功
その他 失敗

**4.2.9.7 説明** *MDIR_RESUME_ALL* はすべてのエントリについて、*MDIR_RESUME_U4*、*MDIR_RESUME_U8*、*MDIR_RESUME_STR* の場合は、指定したクラスタ *ID*（または文字列キー）のエントリについて、*type* フィールドの **open_flag** をリセットする。自分が登録したエントリ以外は操作できない。**Public** なエントリに関しては操作内容が各ノードの *Cache* に伝搬する。

タスクがマイグレーションする際には必ず *STOP_ALL* し、マイグレーションを完了後ノードを移って実行を再開後に、*RESUME_ALL* しなければならない。

#### 4.2.10 MDIR_GOPUBLIC_U4, MDIR_GOPUBLIC_U8

##### 4.2.10.1 コマンドコード

*33* *MDIR_GOPUBLIC_U4*
*34* *MDIR_GOPUBLIC_U8*

**4.2.10.2 機能** ディレクトリエントリを **Public** に指定して、各ノードにキャッシュを生成する

**4.2.10.3 入力** *pbox*

表 44: MDIR_PBOX 構造体 (GOPUBLIC)

| 名前 | 説明 |
|---|---|
| comm | 33,34 |
| subcomm | – |
| arg[2] | U4, U8 の CLID |
| namep | – |
| len | – |
| data | – |

**4.2.10.4 オペレーション**

**4.2.10.5 出力** なし

**4.2.10.6 sss_mdir_request() の戻り値**

*0* 成功
その他 失敗

**4.2.10.7 説明** *MDIR_GOPUBLIC_U4*、*MDIR_GOPUBLIC_U8* は、*U4* および *U8* のディレクトリエントリを **Public** に指定して、各ノードにキャッシュを生成する。ただし、ディレクトリの本文は *cur_node, cur_task* にのみ存在する。

#### 4.2.11 MDIR_RmtQUERY_U4, MDIR_RmtQUERY_U8

##### 4.2.11.1 コマンドコード

*37* *MDIR_RmtQUERY_U4*
*38* *MDIR_RmtQUERY_U8*

**4.2.11.2 機能** *CL_QUERY_ENTRY* 構造体としてクラスタ *ID*（または文字列キー）の情報を必ず *cur_node* から読み出す

**4.2.11.3 入力** *pbox*

表 45: MDIR_PBOX 構造体 (RmtQUERY)

| 名前 | 説明 |
|---|---|
| comm | 37,38 |
| subcomm | – |
| arg[2] | U4, U8 の CLID |
| namep | – |
| len | CL_QUERY_ENTRY 構造体のバイト長さ |
| data | CL_QUERY_ENTRY 構造体へのポインタ (先頭アドレス) |

**4.2.11.4 オペレーション**

**4.2.11.5 出力** なし

**4.2.11.6 sss_mdir_request() の戻り値**

*>0* 成功 *(*受信バイト数*)*
その他 失敗

**4.2.11.7 説明** *MDIR_RmtQUERY_U4*、*MDIR_RmtQUERY_U8* は、*CL_QUERY_ENTRY* 構造体としてクラスタ *ID*（または文字列キー）の情報を *data* と *len* で指定された領域に必ず *cur_node* から読み出す。要求元のノードにキャッシュがある場合は、キャッシュの情報を更新する。*MDIR_QUERY_** やクラスタ *ID* による *MBCF* にキャッシュエントリ無しで失敗した場合に実行する。

### 4.2.12 MDIR_UPDATE_U4, MDIR_UPDATE_U8, MDIR_UPDATE_STR

#### 4.2.12.1 コマンドコード

*41* *MDIR_UPDATE_U4*
*42* *MDIR_UPDATE_U8*
*43* *MDIR_UPDATE_STR*

#### 4.2.12.2 機能 *MDIR_SET_** 実行後に *contents_ptr* の内容を変更（設定）する

#### 4.2.12.3 入力 *pbox*

表 46: MDIR_PBOX 構造体 (UPDATE)

| 名前 | 説明 |
|---|---|
| comm | 41,42,43 |
| subcomm | 0x100:**open_flag**, 0x200:**stop_flag** |
| arg[2] | U4, U8 の CLID |
| namep | STR の場合の文字列キーの先頭論理アドレス |
| len | 送信パラメータのバイト長さ |
| data | 送信領域へのポインタ (先頭アドレス) |

#### 4.2.12.4 オペレーション

#### 4.2.12.5 出力 なし

#### 4.2.12.6 sss_mdir_request() の戻り値

*0* 成功
その他 失敗

**4.2.12.7 説明** *MDIR_UPDATE_U4*、*MDIR_UPDATE_U8*、*MDIR_UPDATE_STR* は、*MDIR_SET_** 実行後に *contents_ptr* の内容を変更（設定）したい場合に使用する。*subcomm* によって **open_flag** と **stop_flag** を立てることも可能。ただし、この操作はキャッシュには反映されない。*contents_ptr* の変更の場合は古い領域は解放される。

#### 4.2.13 MDIR_RmtGET_U4, MDIR_RmtGET_U8

##### 4.2.13.1 コマンドコード

*45* *MDIR_RmtGET_U4*
*46* *MDIR_RmtGET_U8*

**4.2.13.2 機能** クラスタ *ID*（またはキー）と関連づけられたディレクトリの内容（本文）を必ず *cur_node* から読み出す

**4.2.13.3 入力** *pbox*

表 47: MDIR_PBOX 構造体 (RmtGET)

| 名前 | 説明 |
|---|---|
| comm | 45,46 |
| subcomm | – |
| arg[2] | U4, U8 の CLID |
| namep | – |
| len | 受信パラメータのバイト長さ |
| data | 受信領域へのポインタ (先頭論理アドレス) |

**4.2.13.4 オペレーション**

**4.2.13.5 出力** なし

**4.2.13.6 sss_mdir_request() の戻り値**

*>=0* 成功 *(*受信したバイト数*)*
その他 *GET* コマンドに準じる？

**4.2.13.7 説明** *MDIR_RmtGET_U4*、*MDIR_RmtGET_U8* は、*data* と *len* で指定されたタスク内のメモリ領域にクラスタ *ID*（またはキー）と関連づけられたディレクトリの内容（本文）を必ず *cur_node* から読み出す。

#### 4.2.14 MDIR_ALLOC_MEM

##### 4.2.14.1 コマンドコード

*100* *MDIR_ALLOC_MEM*

##### 4.2.14.2 機能 メモリ領域をタスク内の情報開示機構用メモリ領域に確保する

##### 4.2.14.3 入力 *pbox*

表 48: MDIR_PBOX 構造体 (ALLOC_MEM)

| 名前 | 説明 |
|---|---|
| comm | 100 |
| subcomm | – |
| arg[2] | – |
| namep | – |
| len | 確保する領域のバイト長さ |
| data | – |

##### 4.2.14.4 オペレーション

##### 4.2.14.5 出力 なし

##### 4.2.14.6 sss_mdir_request() の戻り値

正 確保した領域の先頭論理アドレス

*0* 失敗

**4.2.14.7 説明** *MDIR_ALLOC_MEM* は、*len* で指定された大きさのメモリ領域をタスク内の情報開示機構用メモリ領域に確保する。この領域は *execve()* をしても影響されない。

#### 4.2.15 MDIR_FREE_MEM

##### 4.2.15.1 コマンドコード

*101*　　*MDIR_FREE_MEM*

##### 4.2.15.2 機能　情報開示機構用メモリ領域を解放する

##### 4.2.15.3 入力　*pbox*

表 49: MDIR_PBOX 構造体 (FREE_MEM)

| 名前 | 説明 |
|---|---|
| comm | 101 |
| subcomm | – |
| arg[2] | – |
| namep | – |
| len | – |
| data | 解放する領域の先頭論理アドレス |

##### 4.2.15.4 オペレーション

##### 4.2.15.5 出力　なし

##### 4.2.15.6 sss_mdir_request() の戻り値

*0*　　成功
その他　　失敗

**4.2.15.7 説明**　*MDIR_FREE_MEM* は、*data* で示された情報開示機構用メモリ領域を解放する。ただし、*MDIR_ALLOC_MEM* で獲得された領域である必要がある。

# 5 MDIR プログラミングインタフェース

## 5.1 構造体

### 5.1.1 MDIR_PBOX 構造体

*MDIR* で管理される情報の操作は、原則として *PBOX (MDIR_PBOX)* を通して行います。現在のところ例外として、クラスタ *ID* を操作するシステムコール
*(*get_gnameU4()、get_gpnameU4()、set_gnameU4()、set_gpnameU4()*)* があります。
*Mdir_pbox* 構造体は表 *50* のような情報を持つ。

表 50: Mdir_pbox 構造体の定義

| 型 | 名前 | 説明 |
|---|---|---|
| unsigned int | comm | MemDir 操作コマンド |
| unsigned int | subcomm | MemDir 操作副コマンド |
| unsigned int | arg[2] | 操作対象 ID(UBYTE4,UBYTE8) |
| unsigned char* | namep | 文字列キー CLID へのポインタ (先頭アドレス) |
| int | len | 送信受信パラメータのバイト長さ |
| unsigned char * | data | 送信受信領域へのポインタ (先頭アドレス) |

*MDIR_PBOX* 構造体は sssdir.h にて以下のように宣言されます。

```
struct Mdir_pbox {
    unsigned int comm; /* MemDir 操作コマンド */
    unsigned int subcomm; /* MemDir 操作副コマンド */
    unsigned int arg[2]; /* 操作対象 ID(UBYTE4,UBYTE8) */
    unsigned char *namep; /* string CLID へのポインタ (先頭アドレス)*/
    int     len; /* 送信受信パラメータのバイト長さ */
    unsigned char *data; /* 送信受信領域へのポインタ (先頭アドレス)*/
};
#define MDIR_PBOX struct Mdir_pbox

#ifdef ultra
/*
 * for 32bit tasks
 */
struct Mdir_pbox32 {
    unsigned int comm; /* MemDir 操作コマンド */
    unsigned int subcomm; /* MemDir 操作コマンド */
    unsigned int arg[2]; /* 操作対象 ID(UBYTE4,UBYTE8) */
    unsigned int namep; /* string CLID へのポインタ (先頭アドレス)*/
    int len; /* 送信受信領域のバイト長さ */
    unsigned int data; /* 送信受信領域へのポインタ (先頭アドレス)*/
};
```

```
#define MDIR_PBOX32 struct Mdir_pbox32
#endif
```

### 5.1.2 CL_QUERY_ENTRY 構造体

この構造体は *MDIR_GET_** 、*MDIR_RmtGET_** コマンドの結果の情報として利用される。
*Mdir_pbox* 構造体は表 *51* のような情報を持つ。

表 51: Mdir_pbox 構造体の定義

| 型 | 名前 | 説明 |
|---|---|---|
| OBJECT_NAME | name[2] | |
| int | type | 下記参照 |
| int | dummy | |
| int | cur_node | 所有タスクの現在のノード |
| OBJECT_NAME | cur_task | 所有タスクの現在の物理タスク名 |
| char * | contenst_ptr | 情報の実態がある場合に有効 |

*CL_QUERY_ENTRY* 構造体は `sssdir.h` にて以下のように宣言されます。

```
struct clname_query_block{
OBJECT_NAME name[2];
int     type; /* 0:unused, 1:32bitID, 2:64bitID, 4:stringID
   0x1000: Cache entry (32bitID and 64bitID only)
   0x2000: Public entry (32bitID and 64bitID only)
   0x8000: Temporary entry (reserve flag)
   0x0100: open_flag, 0x0200: stop_flag */
int     dummy;
int     cur_node;
OBJECT_NAME cur_task;
        char     *contents_ptr;  /* 情報の実体がある場合のみ使用
       0: no entity
    */
};
#define CL_QUERY_ENTRY struct clname_query_block
```

## 5.2 システムコール等

### 5.2.1 sss_mdir_request() システムコール

*Mdir_pbox* 構造体を使って *MDIR* コマンド要求を発行するシステムコール。原則的に、ディレクトリ情報へのアクセスが完了するまでブロックする。このシステムコールは表 *52* のように使われる。

表 52: sss_mdir_request() システムコール

| 項目 | Align | 説明 |
|---|---|---|
| 宣言 | | `sss_mdir_request(MDIR_PBOX *pbox,int *flagp)` |
| 引数 | | |
| `MDIR_PBOX* pbox` | | 送信する *Mdir_pbox* 構造体へのポインタ |
| `int *flagp` | 4 | システム自動生成 CLID 値の受信アドレス |
| 戻り値 | | |
| | | (CPU のポインタサイズのデータを戻り値として返す) |
| 正 | | 処理成功; 送信したパケット長が返る |
| -1 | | 処理失敗; ユーザは再度送信を実行する |

実際の宣言は `ssscore.h` にて次のように行われる。

```
unsigned int sss_mdir_request();
/* unsigned long long sss_mdir_request(); for 64bit */
/*
        unsigned long long sss_mdir_request(pbox,flagaddr);
        MDIR_PBOX      *pbox;
        char   *flagaddr;
*/
```

### 5.2.2 get_gnameU4() システムコール

実際の宣言は ssscore.h にて次のように行われる。

```
unsigned int get_gnameU4();
/*
        クラスタグローバルなタスク名を返す
        get_gnameU4() と get_gnameU8() はどちらか一方のみが使用可能であり、
        get_gnameUx() もしくは set_gnameU4() が初めて実行されたタイミングで
        gnameUx がタスク構造体にアサインされる。
        getpid (U4 version)、 エラーは (UBYTE4)-1
*/
```

### 5.2.3 get_gpnameU4() システムコール

実際の宣言は ssscore.h にて次のように行われる。

```
unsigned int get_gpnameU4();
/*
        クラスタグローバルな親タスク名を返す
        fork 前に親に gnameU4 がアサインされていなければ、
        子タスクには情報が伝搬されない。
        getppid (U4 version)、エラーは (UBYTE4)-1、親タスク名未定義は 0
*/
```

### 5.2.4 set_gnameU4() システムコール

実際の宣言は ssscore.h にて次のように行われる。

```
unsigned int set_gnameU4();
/*
         unsigned int set_gnameU4(taskname);
         OBJECT_NAME    taskname; クラスタ拡張されたオブジェクト名
        クラスタグローバルなタスク名をセットする。
        get_gnameU4() を使用する前にシステムコールを呼ぶ必要がある。
        taskname=NIL で呼び出した場合は OS が適当な名前を生成して割り当てる。
        返り値はセットされたクラスタグローバルなタスク名。
        getpid (U4 version)、 エラーは (UBYTE4)-1
*/
```

### 5.2.5 get_gnameU8() システムコール

実際の宣言は ssscore.h にて次のように行われる。

```
unsigned long long get_gnameU8();
/*
        !!!!!!!Ver2.3or2.4 では使用を禁止し、必ずエラーとする。!!!!!!!!!!!!!
        クラスタグローバルなタスク名を返す
        get_gnameU4() と get_gnameU8() はどちらか一方のみが使用可能であり、
        get_gnameUx() もしくは set_gnameU4() が初めて実行されたタイミングで
        gnameUx がタスク構造体にアサインされる。
        getpid (U8 version)、エラーは (UBYTE8)-1
*/
```

### 5.2.6 get_gpnameU8() システムコール

実際の宣言は ssscore.h にて次のように行われる。

```
unsigned long long get_gpnameU8();
/*
        !!!!!!!Ver2.3or2.4 では使用を禁止し、必ずエラーとする。!!!!!!!!!!!!!
        クラスタグローバルな親タスク名を返す
        fork 前に親に gnameU8 がアサインされていなければ、
        子タスクには情報が伝搬されない。
        getppid (U8 version)、エラーは (UBYTE8)-1、親タスク名未定義は 0
*/
```

# 開発元


| | |
|---|---|
| 株式会社情報科学研究所 | *http://www.isll.co.jp/* |
| *SSS-PC* プロジェクトチーム | *http://www.ssspc.org/* |
| 国立情報学研究所　松本　尚 | *http://www.nii.ac.jp/* |
| | |
| お問い合わせ先 | *info@isll.co.jp* |



本ソフトは、*IPA* (情報処理振興事業協会) の 『*2003* 年度重点領域情報技術開発事業』の支援をうけて開発されたものです。


以上