TO: MSPM Distribution
FROM: Michael J. Spier
DATE: 12/09/68
SUBJECT: IPC documentation

The attached sections BJ.10.02-BJ.10.05 describe the new Interprocess Communication facility which is currently being implemented. They supersede sections BQ.6.03 through BQ.6.09 inclusive, which describe the old IPC.

Section BJ.10.01 is re-issued to reflect the following changes in IPC design:

1. The calling sequence of ipc$block has no longer the "interaction" argument.

2. The structure of the event message has been modified to include information about the origin of the message.

3. Three new calls have been added, ipc$read_ev_chn, ipc$chn_1 and ipc$chn_2.

Published: 12/09/68
(Supersedes: BJ.10.01 10/02/68)

Identification

IPC reference manual
Michael J. Spier

Purpose

This section is intended to serve as a reference manual for the users of the interprocess communication facility (IPC); it lists all the entry points which are available to the facility's users, giving the full calling sequence (return arguments are underlined) as well as the arguments' PL/I declarations. Associated with each paragraph is an MSPM section number to be used as reference.

The following is subdivided into three paragraphs corresponding to the IPC's three major modules, the user-ipc, the hardcore-ipc and the device-signal table manager.

User-IPC entry points (See BJ.10.03)

The user-ipc is a collection of procedures in segment <ipc>; this segment resides in all non-hardcore rings (1->63) and its entry points are available to all users.

1) To create an event channel in the caller's validation ring associated ECT,

```
call ipc$create_ev_chn(chname, code);

dcl  chname fixed bin(71), code fixed.
```

2) To destroy an event channel,

```
call ipc$delete_ev_chn(chname, code);
```

3) To make an event channel into an event-call type channel

```
call ipc$decl_ev_call_chn(chname, procptr,
     dataptr, prior, code);

dcl  (procptr, dataptr) pointer, prior fixed;
```

4) To make an event channel into an event-wait type channel (a newly created channel has the event-wait type by default),

```
call ipc$decl_ev_wait_chn(chname, code);
```

5) To empty a channel of all pending signals (in other words, in order to reset an event channel),

```
call ipc$drain_chn(chname, code);
```

6) To inhibit a channel for reading purposes (this turns the channel `off' and causes ipc$block to completely ignore it) without affecting any pending signals,

```
call ipc$cutoff(chname, code);
```

7) To undo the previous call (turn channel `on' again),

```
call ipc$reconnect(chname, code);
```

8) To get the name of this validation level's associated event channel 1,

```
call ipc$chn_1(chname, code);
```

9) To get the name of this validation level's associated event channel 2,

```
call ipc$chn_2(chname, code);
```

10) To give, in procedure ipc$block, event-wait channel interrogation precedence over event-call channel interrogation,

```
call ipc$set_wait_prior (code)
```

11) To give, in procedure ipc$block, event-call channel interrogation precedence over event-wait channel interrogation,

```
call ipc$set_call_prior (code);
```

12) To cause procedure ipc$block to completely ignore event call channels (in the caller's ring),

```
call ipc$mask_ev_calls (code);
```

13) To undo the above and make ipc$block re-interrogate event call channels,

```
call ipc$unmask_ev_calls (code);
```

Note: The last four calls (numbers 10->13) take effect in the caller's validation-level associated ECT only, and do not affect the remaining (potential) 62 ECTs.

14) To block one's process until some event of current interest has occurred,

```
call ipc$block(argptr, msgptr*, code);

dcl  (argptr, msgptr) pointer;
```

argptr is a pointer to the base of an argument-structure (named the `wait-list') which is declared as follows:

```
dcl  1 wait_list,
     2 number_of_channels fixed,  /*current size of
                                    following array*/
     2 channels(n) fixed bin(71); /*array of event
                                    channel names*/
```

msgptr is a pointer to the base of a return argument structure into which ipc$block puts the received event-signal message and which has the following declaration:

```
dcl  1 message,
     2 chname fixed bin(71),      /*channel over which
                                    message arrived*/
     2 message fixed bin(71),     /*2-word event
                                    message*/
     2 sender bit(36),            /*sending process*/
     2 origin,                    /*origin of event
                                    message*/
       3 devsignal bit(18),       /*1 = device signal*/
       3 ring bit(18),            /*sender's ring
                                    number*/

* /* Note: the preceding two items are right-adjusted
         (packed fixed bin(18) variables)*/

       2 channel_index fixed;     /*channel's index in
                                    `wait-list'*/
```

15) To read an event message out of an event channel

```
call ipc$read_ev_chn(chname, readmark, msgptr*,
     code);

declare  readmark fixed;          /*0 = no event message
                                      returned
                                    1 = event message
                                      returned*/
```

(*) Even though msgptr is provided by the caller, it points to a structure into which ipc$block puts return information.

All of the above-mentioned calls return one of the following values for `code':

code=0 -> No error

code=1 -> Ring access violation (event channel resides in ring which is protected from caller's validation level).

code=2 -> ECT not found (special case of "event channel not found").

code=3 -> Event channel not found in ECT.

code=4 -> Logical error in using IPC (e.g. waiting for event-call chn).

code=5 -> Erroneous argument. (e.g. zero-value event channel name).

Calls to an associated procedure

A call to ipc$block may result in the diversion of the process' execution into an event-call-channel's associated procedure.

An associated procedure is always called by ipc$block with the following standard calling sequence:

```
call [associated-procedure] (msgptr);

declare    msgptr pointer;
```

where [associated-procedure] is an entry point pointed to by `procptr' (see ipc$decl_ev_call_chn)

and where `msgptr' points to the base of the following structure:

```
dcl  1 ev_message,
     2 event_channel fixed bin(71),
     2 message fixed bin(71),
     2 sending_process bit(36),
     2 origin,
       3 dev_signal bit(18),        /*right adjusted*/
       3 ring bit(18),              /*right adjusted*/
     2 dataptr pointer;
```

Hardcore-IPC entry points

This is a collection of procedures in segment <hc_ipc> which, as its name implies, resides in ring-0. These entry points are accessible through the gate <hcs_>. Normally, only entry point hcs_$wakeup is called by the user, and the remaining two entry points are internal to the IPC. However, the user may call hcs_$ipc_init which is foolproof and ineffective if unnecessarily invoked, or hcs_$block (at his peril) which will block his process until the next wakeup is received (if ever).

As implied above, all these entry points are available from all non-hardcore rings.

1) To send an IPC signal to some other (or perhaps one's own) process,

```
call hcs_$wakeup(processid, chname, message, code);

dcl  processid bit(36), message fixed bin(71);
```

The error code returned by this call differs from the above-mentioned and can assume one of the following values:

code=0 -> No error (signalling correctly done)

code=1 -> Signalling correctly done but target process was found to be in the 'stopped' state.

code=2 -> Erroneous call argument, signalling aborted (e.g. zero-value process-ID, zero-value channel name).

code=3 -> Target process not found, signalling aborted. (e.g. process-ID is wrong, or target process has been destroyed).

2) To block one's process until the occurrence of the next wakeup,

```
call hcs_$block
```

3) To inform ring_0 of an ECT in one's validation-level ring.

```
call hcs_$ipc_init(ectptr);

dcl  ectptr pointer;
```

Device Signal Table Manager entry points (See BJ.10.05)

The Device Signal Table Manager is a collection of procedures in segment <dstm> to provide an interface between (hardware) processor interrupts and the Traffic Controller's entry `wakeup'. The DSTM resides in wired-down hardcore and can be invoked by hardcore procedures only.

1) To attach a device to one's process and associate it with some event channel,

```
call dstm$attach(devindex, mode, chname, code);

dcl  (devindex, mode) fixed;
                              /*mode=0 -> binary,
                                mode=1 -> count*/
```

`code' returns one of the following values:

code=0 -> No error (device successfully attached).

code=1 -> Erroneous device index.

code=2 -> Device already attached; call aborted.

2) To detach a device from its present owner,

```
call dstm$detach(devindex);
```

3) To find out to whom a specific device is currently attached,

```
call dstm$check_auth(devindex, processid);
```