Charlie Strohman
4/19/05
Version 1

CESR DSP Beam Instrumentation API

Introduction

This document attempts to define the API (Application Programmer's Interface) that must be available to people writing applications that use the CESR DSP-based Beam instrumentation. It starts with an outline of a typical application, followed by a description of the functions that are available to users. It also contains a description of the supporting data structures that are in the CESR MPM.

Eventually all of the functions available for writing applications should be documented. That includes the functions listed here, the functions for communicating with the devices, and the functions for dealing with data.

All function names will start with the name 'cbi_' (for Cesr Beam Instrumentation). The function names will consist of words separated by underscores. All text is lowercase.

All functions will be specified for calling from a 'c' program.

Application Pseudo-code

The following description illustrates the structure of an application. It is not rigorous 'c' code. As we refine the API, several of the functions called should probably be merged.

```
my_app(argc, argv) {

        int prog_num;           /* program number from 2 to 254 */
        int pid;                /* VMS process ID */
        char prog_name[];       /* The name of this program */

        /* Call cbi_dev_alloc_setup to get a program number. This number will be unique to each
        instance of any program that uses the DSPs. */
        prog_num = cbi_dev_alloc_setup();

        /* Read in a file that contains a list of the devices that this instance of this program uses.
        The 'device_filename' can come from command line arguments, a global symbol, or a
        logical name. Supply the program number. A compact 'device request mask' will be
        constructed and saved in the MPM. The order of devices in this file could control the order
        in which data is read from the devices. */
        cbi_read_devlist(device_filename, prog_num, &dev_req_struct);

        /* Read in configuration data for all of the devices that this program uses. The data
        structures are kept in the local memory of this program; other programs do not need to see
        the data. */
        cbi_read_config(config_file_list);
```

```
/* Set a flag to indicate that all structures need to be downloaded the first time. This will
prevent problems with recycled program numbers. This should be set by the
cbi_read_config function. */
first_time_flag = TRUE;

/* Enter the main program loop and follow a sequence of operations */
done = FALSE;
while (done == FALSE) {

    /* Make an allocation request for all of the devices that are needed. Always request
    the full set of required devices at once to avoid deadlock. Get back a timing trigger
    code.  The request mask and program number may be maintained by the API
    functions. */
    tim_trig_code = cbi_dev_alloc_req(dev_req_mask, prog_num);

    /* Enter the setup loop where constant control structures are sent from the copy in
    local memory to the devices as they are allocated. */
    while (waiting_for_another == TRUE) {

        /* Wait for notice that another device has been allocated. Maybe a timer is
        used here and some action is performed if we have to wait too long?
        Possibly call cbi_dev_alloc_cancel() and work with a smaller set of
        devices? */
        while(alloc_info.dev_num == 0) {
            alloc_info = cbi_dev_alloc_ready();
        }

        /* Setup the device that was just allocated. If the ‘first_time_flag’ is true,
        then we have to send all setup and configuration data. Otherwise, look at the
        program number of the previous owner that is in the ‘alloc_info’. If it was
        our program, then we only need to send structures that have to be sent every
        time. If it was not our program, we need to examine who modified each
        structure and decide if we need to update it. Some structures may have been
        modified that we don’t care about, so we can skip them */

        /* If we’ve been granted ownership of all the devices that we asked for and
        have finished setting up the devices, clear the ‘waiting_for_another’ flag to
        exit this loop. */
        if (setup_count == num_devices) {
            waiting_for_another = FALSE;
        }
    }

    /* Enter the acquisition loop where we control devices and read data without giving
    up ownership of the devices. There may be several passes of control and
    acquisition. Only after the last pass do we de-allocate devices. */

    while(keep_devices == TRUE) {

        /* Setup parameters unique to this pass. Send commands to start our data
        acquisition in all of the devices. */
```

```
                /* If a timing trigger is needed, have the trigger manager put our trigger
                code into the command byte on the global clock. We have configured our
                devices to look for this byte from the timing system. */
                if (need_trig) {
                        cbi_dev_alloc_trig(tim_trig_code);
                }

                /* Wait for acquisition to complete. */

                /* Loop over all devices and collect the data. Maybe try to process them in
                order from the one with the least data to the one with the most data. This
                keeps us from tying up devices that other applications may be waiting for.
                */
                for (device = 1; device < num_devices; device++) {

                        /* Collect data */
                }

                /* If we are done with all commands and are ready to give up device
                ownership, break out of the loop. Otherwise, issue more commands and
                gather more data */
                if (finished) {
                        keep_devices = FALSE;
                }
        }

        /* If we are done with the devices for now, deallocate them. Do it all over again
        when commanded to or when some timer elapses. */
        for ( loop over devices) {
                cbi_dev_alloc_free(device);
        }
        if (final_pass) {
                done = TRUE;
        }
    }

    /* We are finished and ready to exit. Release the 'prog_num' that was allocated to us. */
    cbi_dev_alloc_close(prog_num);

    exit();
}
```

Vector Access to Interlocked Data

The DSPs will be collecting and processing data in a background mode. The control system will want to read the data. However, we do not want part of the data from an instrument to change during the time when the control system is reading the data.

We must provide:

> An interlock semaphore in the instrument
> DSP code that respects the interlock
> Xbus processor code that knows how to use the interlock
> Data in a format acceptable to the control system
> > (ints or VAX floating point)

1. The Xbus processors know from the Xbus operation specified in the database that they must obtain ownership of the interlock semaphore in each instrument. The DSP must also get ownership of the semaphore in order to update the information.

2. Once the semaphore is obtained, the Xbus processor gets the data from each instrument. The DSP is locked out from updating any of the data, guaranteeing that the data does not span multiple measurements.

3. The final step of the Xbus processor is to release all owned semaphores.

Notes:

1. If the Xbus processor cannot get the semaphore for one or more devices after a given number of tries, it will skip that device and leave the database unchanged. Status information will be returned to the calling program indicating that there was an error for one or more elements.

2. Vector operations are serialized by the Xbus processors, so we do not need to be concerned about conflicts that might arise from two different applications accessing vectors on the same device.

3. The maximum data in a database node should be limited to around 500 elements to minimize the impact on all the other control system programs.

Allocation function library

cbi_dev_alloc_setup

```
int prog_num = cbi_dev_alloc_setup();
```

Register with the allocation server.
Returns a unique program number or error code.

prog_num < 0 means error
prog_num > 0 is the program number to use

cbi_dev_alloc_req

```
int trig_code = cbi_dev_alloc_reg(struce dev_mask, int prog_num);
```

Request the set of devices that are needed.
Returns a timing trigger code or error code.

trig_code < 0 means error
trig_code > 0 is the trig_code to use

cbi_dev_alloc_ready

```
struct *alloc_info = cbi_dev_alloc_ready();
```

See if another device has been allocated.
Returns a pointer to an information structure.

```
struct alloc_info{
        int status;
        int dev_num;
        int last_owner;
}
```

status == 0 means no additional devices allocated
status < 0 means error
status > 0 means that a device has been allocated

cbi_dev_alloc_free

```
int status = cbi_dev_alloc_free(int dev_num);
```

Deallocate a single device.
Returns success or failure.

status < 0 means error
status = 1 means success

cbi_dev_alloc_cancel

int status = cbi_dev_alloc_cancel(int dev_num);

Cancel the allocation request for a single device.
Returns success or failure.

status < 0 means error
status = 1 means success

The calling program must remove the specified device from its list of active devices.

cbi_dev_alloc_trig

int status = cbi_dev_alloc_trig(int trig_code);

Transmit specified trigger code on timing system cable.
Returns success or failure.

status < 0 means error
status = 1 means success

cbi_dev_alloc_close

int status = cbi_dev_alloc_close(int prog_num);

Clean up.
Returns success or failure.

status < 0 means error
status = 1 means success

This function must de-allocate any devices that are still allocated.

cbi_read_devlist();

cbi_read_config();

<u>**Allocation Server Tables in MPM**</u>

cbi_prog_tab

This table has an entry associated with each ‘program number’. When an application calls ‘cbi_dev_alloc_setup’, the function code will step through the semaphores associated with each element. When it finds a free semaphore, the element number that it is at will become the ‘program number’ that is returned to the caller. The process ID and program name of the caller will be put in the table. Later on, the device mask that shows which devices this program is using will be updated. This may be useful for management purposes when we are interested in seeing what programs are using which devices. The ‘cbi_dev_alloc_close’ function will clear the semaphore. It may or may not reset the element data.

node CBI_PROG_TAB

ele	pid	name	dev_mask	
1	2680011F	XYPLOT	00000000 00000400	!using dev #10
2	2640658E	CONTROL;13	00000000 00006000	!using dev #13,#14
3	00000000	null	00000000 00000000	!available
4	00000000	null	00000000 00000000	!available
5	26202975	XRAY	00000000 10000000	!using dev #28
.				
.				
40	00000000	null	00000000 00000000	!available

Allocation Server Tables in MPM

cbi_dev_tab

This table has an entry associated with each device that may be used by applications. The ‘dev_name’ fields contain the names of all of the devices (like ‘BPM 6W’ or ‘BSM 23E’). These are the same names that will be used in the files that are read by ‘cbi_read_devlist’. The element number associated with each name is used to identify devices by number. As devices are allocated to various applications, the allocation server will read the program number of the previous owner and update it with the current owner. The program number of the previous owner will be returned to the calling program so that it can determine if the device configuration may have been altered.

node CBI_DEV_TAB

ele	name	last_owner	
1	BPM_0W	0	!not used since refresh
2	BPM_1W	24	!
.			
.			
10	BPM_9W	2	!
.			
.			

Allocation Server Tables in MPM

cbi_hist_tab

This table is a 2-dimensional history array with an entry for every writable data structure in every device.

When an application writes to a device, the function that does the writing needs to update the program number for that structure in that device.

When an application allocates a device and it is informed that some other application previously owned the device, it can use the information in this table to figure out what structures it may need to download.

node CBI_HIST_TAB

ele	device	struct	last_writer	
1	1	1	22	
2	1	2	8	
.				
50	1	50	22	
51	2	1	0	
52	2	2	0	
.				
.				
100	2	50	0	
.				
.				
??	last_dev	50	0	!total size = num dev * num write structs