Storing Application Parameters in BGDSPM Shared Memory and Non-Volatile Memory

December 16, 2003

Craig Drennan

Storing Application Parameters in BGDSPM Shared Memory and Non-Volatile Memory

December 16, 2003

Craig Drennan

TABLE OF CONTENTS

TABLE OF LISTINGS

I. Introduction

The Booster Generic DSP Module (BGDSPM) is a VXI module on which several Booster Low Level RF instrumentation and control applications have been built. Programmable parameters for each application that are read and written from ACNET use a block of shared memory on the BGDSPM. The communications path includes an Ethernet connection to a Power PC crate controller in slot 1 of the VXI crate. Parameter data is exchanged over the VXI backplane.

This note attempts to document the way the parameter memory was structured in the development of the Phase Control Module application. It lists the specific C language structures used and the procedures or methods used in storing and retrieving data with these structures. This discussion is limited to the procedures and methods used with the modules in the test stand where a direct terminal connection is made with the Power PC crate controller. Not included here are the methods used to transfer data with ACNET.

Also described in this note are the structures and methods used for storing and retrieving parameter data with the non-volatile memory on the BGDSPM. Both code downloaded to the module and the current values of the parameters are stored in the non-volatile memory. This allows the modules to startup and run autonomously after a power cycle or other reset event.

II. The Shared Memory Data Structure

Except for *long double* data format variables, all C/C++ data types compiled for the SHARC DSP used on the BGDSPMs are 32-bits in size. This simplifies a lot of things when defining how the shared memory is laid out. A C *struct* type data structure is defined and located at the beginning address of the shared memory. By knowing the definition of the C data structure, the base address of the shared memory and the local address of the VXI module, programs running on the Power PC crate controller can address any variable in the shared memory.

```
struct dualportmem
{
       int    msgsem;          /* semaphore for message buffer*/
       int    nChar;           /* number of characters in the buffer*/
       char   msg[512]; /* Message buffer for communicating with PPC terminal*/

       int    nint_params;    /* Number of integer parameters and curves.*/
       int    nflt_params;     /* Number of floating point parameters and curves*/

       int    PARAM_MARKER;
```

```
        struct int_params ip[NINTPARAMS]; /*An array of integer parameter structures*/
        struct flt_params fp[NFLTPARAMS]; /*An array of float parameter structures*/

        int    DATA_MARKER;

};
```

Listing II.1 The Dual-Port (Shared) Memory Structure.

The dual-port, shared memory data structure is given in Listing II.1. The first three elements in the structure are used in passing diagnostic messages between the code running on the DSP and the terminal connected to the Power PC. The BGDSPM itself lacks a serial interface or other communications path directly from the board. This message buffer is very useful during code development and can provide a means of reporting errors to a terminal when in service.

The next two elements report the number of parameter data structures contained in the shared memory. The PARAM_MARKER is simply a bookend of sorts that marks the beginning of the collection of parameter structures. The other bookend coming immediately after the last parameter structure, and immediately before the first data value is the DATA_MARKER. The use of these markers will become clearer as we continue. Go ahead and note that knowing only the definition of the *dualportmem struct* , the shared memory base address and the VXI module local address, the memory address of the PARAM_MARKER is known. The memory address of the DATA_MARKER depends on the number of parameter structures defined.

Listing II.2 gives the *int_params* structure and Listing II.3 gives the *flt_params* structure. These are the structures for integer value parameters and floating value parameter, respectively. Note that the two structures are identical except for the variable type in several instances. Yet, since all variable types for the DSP are 32-bit, the two structures are identical in size.

Of particular interest are the *rdg* and *set* elements of the structures. The parameter "setting" is the value written from the Power PC to the VXI modules shared memory. The parameter "reading" is the value written into shared memory by the DSP processor that is currently being used in the application. This provides a check back to the higher level system that the parameter values being set have or have not been applied.

```
struct int_params
{
        char   desc[128];    /*description of the parameter (string)*/
        int    dim;          /*array size of the parameter or curve */
        int    ul;           /*upper limit on value*/
        int    ll;           /*lower limit on value*/
        int    dflt;         /*default value */
        int    flag;         /*update flag*/
        int    rdg;  /*location of parameter reading (offset from "PARAM_MARKER")*/
        int    set;  /*location of parameter setting (offset from "PARAM_MARKER")*/
        int    *data;      /*not used. only makes ppc and dsp struct sizes the same*/
};
```

Listing II.2 The structure used for integer parameters.

```
struct flt_params
{
       char   desc[128];    /*description of the parameter (string)*/
       int    dim;          /*array size of the parameter or curve */
       float  ul;           /*upper limit on value*/
       float  ll;           /*lower limit on value*/
       float  dflt;         /*default value */
       int    flag;         /*update flag*/
       int    rdg;  /*location of parameter reading (offset from "PARAM_MARKER")*/
       int    set;   /*location of parameter setting (offset from "PARAM_MARKER")*/
       float  *data;      /*not used. only makes ppc and dsp struct sizes the same*/
};
```

Listing II.3 The structure used for floating point parameters.

Note that the actual value, setting or reading, is not located in any of the structures mentioned. The actual values of the settings and readings are stored in the shared memory starting just after the DATA_MARKER. The *rdg* and *set* elements in the parameter structure store the address offset from the location of the PARAM_MARKER that the parameters reading and setting, respectively, are stored. The particular offset is computed in a routine that initializes a parameter structure for every parameter in the application. A fragment of code for doing this is given in Listing II.4.

The "flag" for a particular parameter is set by the crate processor whenever the value of the "setting" has been changed as an indication to the application code that the new value has been set.

```
Struct dualportmem  *sm;
sm = (struct dualportmem *)  SHARED_MEM_BASE;  // this is 0x00460000
OFFSET_1  =  &sm->DATAMARKER - &sm->PARAM_MARKER;
Offset_2  =  1;
for ( k = 0; k < sm->nint_params; k++)
  {
       sm->ip[k].rdg = OFFSET_1 + Offset_2;
       Offset_2      = Offset_2 + sm->ip[k].dim;
       sm->ip[k].set = OFFSET_1 + Offset_2;
       Offset_2      = Offset_2 + sm->ip[k].dim;
   }
```

Listing II.4 Code fragment illustrating how the data offsets are determined

One benefit of handling the parameters in this manner is that the methods used for managing scalar parameters (dim = 1) are the same as for arrays or lists (dim > 1). A second benefit of arranging the parameter information this way is that all the parameters can be setup and initialized in the DSP code and the code written to run from the Power PC to read, write, list, sort, etc. can be written in a more generic way. This is because all that needs to be known about a parameter is established in the DSP code.

III. DSP Data Memory Parameter Data Structure

The SHARC DSP processor executes instruction quicker when operating with data stored in its internal data memory. For each parameter we also define a data structure in the DSP data memory. This structure holds the parameter properties (eg dimension, limits, default values) and pointers to the other memory locations we want to read or write the parameter value. Listing III.1 gives the structure definition for the integer and floating point versions.

```
struct dm_int_params
{
       int    dim;          /*array size of the parameter or curve */
       int    ll;           /*lower limit on value*/
       int    ul;           /*upper limit on value*/
       int    dflt;         /*default value */
       int    *sm_rdg;      /*pointer to parameter reading in shared memory*/
       int    *sm_set;      /*pointer to parameter setting in shared memory*/
       struct nvshort2 *bram_val; /*pointer to the battery backed ram storage*/
       int    *param;       /*pointer to FPGA register or DSP local variable*/
};

struct dm_flt_params
{
       int    dim;          /*array size of the parameter or curve */
       float  ll;           /*lower limit on value*/
       float  ul;           /*upper limit on value*/
       float  dflt;         /*default value */
       float  *sm_rdg;      /*pointer to parameter reading in shared memory*/
       float  *sm_set;      /*pointer to parameter setting in shared memory*/
       struct nvlong *bram_val;  /*pointer to the battery backed ram storage*/
       float  *param;       /*pointer to FPGA register or DSP local variable*/
};
```

Listing III.1 DSP internal data memory parameter data structure.

IV. Setup and Initialization of the Parameter Structures

Several tasks must be performed at reset or startup. The shared memory parameter data structures must be initialized. Offsets in the shared memory where the parameter readings and settings are located must be determined. Pointers to the shared memory parameter readings and settings must be set in the DSP data memory structures. Beyond these, the parameter values previously stored in the non-volatile battery backed memory must be retrieved. Interfacing with the battery backed memory is covered in a following section.

The DSP internal data memory structure is declared and initialized in a header file. Listing IV.1 gives an example of this initialization. Listing IV.2 gives the setup procedure used in the Paraphase Curve Generator application. Table V.1 provides another view of where the data ends up.

```
/* ========= Initialization of the DSP data memory structure */
struct dm_int_params iparams[NINTPARAMS] =
       {
                             //{dimension | L. L.| U. L.| Dflt |       <pointers >        }
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
/* 0. Curve Start Count     */ {1        , 0    , 511  , 500  , 0, 0, 0, (int*) WR_SEQ_CNT  },
/* 1. Initial Curve Offset  */ {1        , -8191, 8191 , 0    , 0, 0, 0, (int*) WR_C1_OFFSET},
/* 2. Post Transition Offset*/ {1        , -8191, 8191 , 0    , 0, 0, 0, (int*) WR_C2_OFFSET},
/* 3. Beam Rotation Offset  */ {1        , -8191, 8191 , 0    , 0, 0, 0, (int*) WR_BR_OFFSET},
/* 4. Trans Offset Start Cnt*/ {1        , 0    , 32760, 15000, 0, 0, 0, (int*) WR_C2_COUNTS},
/* 5. Beam Rot Start Cnt    */ {1        , 0    , 32760, 24000, 0, 0, 0, (int*) WR_BR_COUNTS},
/* 6. The Paraphase Curve   */ {MAXCRVLEN, -8191, 8191 , -1   , 0, 0, 0, (int*) DACTBL_BASE }
       };

/*--------------------------------------------------------------------------------------*/
float ftemp;
struct dm_flt_params fparams[NFLTPARAMS] =
       {
                            //{dimension| Lower Lim| Upper Lim| Default Val|<pointers >}
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
/* 0. Undefined float param */{1        , 0.       , 1.       , 1.         , 0, 0, 0, &ftemp}
       };
```

Listing IV.1 Declaration and initialization of the DSP internal memory structures.

```
Struct dualportmem  *sm;
sm = (struct dualportmem *)  SHARED_MEM_BASE;  // this is 0x00460000


/*========================================================================== */
// setup_params(void)
//
// This routine establishes the parameter settings for the application.
// It sets up values for the shared memory and DSP memory

int setup_params(void)
{
       int OFFSET_1, offset, bram_ndx;
       int j;
       offset = 1;
       bram_ndx = 0;

       sm->nint_params = (int) NINTPARAMS;
       sm->nflt_params = (int) NFLTPARAMS;

       // Compute the offset between the PARAM_MARKER and DATA_MARKER
       OFFSET_1 = &sm->DATA_MARKER - &sm->PARAM_MARKER;

       // The ip[].rdg and ip[].set variables will be the offset from the
       // PARAM_MARKER and the shared memory location storing the parameters
       // reading and setting, respectively.  That is the offset to the first
       // value if the dimension of the parameter is more than one.

       for (j=0; j<NINTPARAMS; j++)
       {
              sm->ip[j].dim = iparams[j].dim ;  // this is the parameter dimension.
              sm->ip[j].ul  = iparams[j].ul  ;  // upper limit of value
              sm->ip[j].ll  = iparams[j].ll  ;  // lower limit of value
              sm->ip[j].dflt= iparams[j].dflt;  // default value
              sm->ip[j].flag= 0;                   // update flag

              sm->ip[j].rdg = OFFSET_1 + offset; // pointer to reading (marker + 1)
              offset += sm->ip[j].dim;

              sm->ip[j].set = OFFSET_1 + offset; // pointer to setting (marker + 2)
              offset += sm->ip[j].dim;

              iparams[j].sm_rdg = (int *) (sm->ip[j].rdg + &sm->PARAM_MARKER);
              iparams[j].sm_set = (int *) (sm->ip[j].set + &sm->PARAM_MARKER);

              /*ptr to the battery backed ram storage*/
              iparams[j].bram_val = &(bramshortparam->bshort[bram_ndx]);
              bram_ndx += iparams[j].dim;

       // Note that the iparams[].param pointer is setup in the declaration or elsewhere
       }

       // Parameter Description for shared memory`~`~`~`~`~`~`~`~`~`~`~`~`~`~`~`~`~`~`~`~`~`~`~
       strcpy(sm->ip[0].desc, "0. Curve Start Count: Start Cnt for the paraphase curve player\n");
       strcpy(sm->ip[1].desc, "1. C1 Offset: Baseline offset for the paraphase curve\n");
       strcpy(sm->ip[2].desc, "2. C2 Offset: Post-Transition offset for the paraphase curve\n");
       strcpy(sm->ip[3].desc, "3. BR Offset: Beam Rotation offset for the paraphase curve\n");
       strcpy(sm->ip[4].desc, "4. C2 Count: Apply Offset C2 at C2_Count DSP Clocks from start\n");
       strcpy(sm->ip[5].desc, "5. BR Count: Apply Offset BR at BR_Count DSP Clocks from start\n");
       strcpy(sm->ip[6].desc, "6. Curve 1: The Paraphase Curve\n");
```

```
        // Setup float type parameters ~`~`~`~`~`~`~`~`~`~`~`~`~`~`~`~`~`~`~`~`~`~`~`~`~`~`~`~`~`~`~`~`~`~`~`~`
        for (j=0; j<NFLTPARAMS; j++)
        {
                sm->fp[j].dim = fparams[j].dim ;  // this is the parameter dimension.
                sm->fp[j].ul  = fparams[j].ul  ;  // upper limit of value
                sm->fp[j].ll  = fparams[j].ll  ;  // lower limit of value
                sm->fp[j].dflt= fparams[j].dflt;  // default value
                sm->fp[j].flag= 0;                   // update flag

                sm->fp[j].rdg = OFFSET_1 + offset; // pointer to reading (marker + 1)
                offset += sm->fp[j].dim;

                sm->fp[j].set = OFFSET_1 + offset; // pointer to setting (marker + 2)
                offset += sm->fp[j].dim;

                fparams[j].sm_rdg = (float *) (sm->fp[j].rdg + &sm->PARAM_MARKER);
                fparams[j].sm_set = (float *) (sm->fp[j].set + &sm->PARAM_MARKER);

                /*ptr to the battery backed ram storage*/
                fparams[j].bram_val = &(bramlongparam->blong[bram_ndx]);
                bram_ndx += fparams[j].dim;

                // Note that the iparams[].params pointer is setup in the declaration or elsewhere.

        }

        // Prototype float value~`~`~`~`~`~`~`~`~`~`~`~`~`~`~`~`~`~`~`~`~`~`~`~`~`~`~`~`~`~`~`~`~`~`~`~`~`~
        strcpy(sm->fp[0].desc, "F0. Prototype float value\n");

        // Set a known value in DATA_MARKER
        sm->DATA_MARKER = 0x0A5A5A5A;
        sm->PARAM_MARKER = 0x0FEDCBA9;

        return 0;
}
```

Listing IV.2 The parameter structure initialization procedure.

Listing VI.3 Absolute addresses in the shared memory for the Paraphase Curve Generator application.

Address	**Variable**	**Offset from PARAM_MARKER**
	struct dualportmem	
0	msgsem	
1	nChar	
2 to 513	msg[512]	
514	nint_params	
515	nflt_params	
516	PARAM_MARKER	
517 to 652	ip[0] struct int_params	
653 to 788	ip[1] struct int_params	
789 to 924	ip[2] struct int_params	
925 to 1060	ip[3] struct int_params	
1061 to 1196	ip[4] struct int_params	
1197 to 1332	ip[5] struct int_params	
1333 to 1468	ip[6] struct int_params	
1469 to 1604	fp[0] struct flt_params	
1605	DATA_MARKER	
	Readings and Settings	***Offset from PARAM_MARKER***
1606	reading ip[0]	1090
1607	setting ip[0]	1091
1608	reading ip[2]	1092
1609	setting ip[1]	1093
1610	reading ip[2]	1094
1611	setting ip[2]	1095
1612	reading ip[3]	1096
1613	setting ip[3]	1097
1614	reading ip[4]	1098
1615	setting ip[4]	1099
1616	reading ip[5]	1100
1617	setting ip[5]	1101
1618 to 2129	reading ip[6]	1102
2130 to 2641	setting ip[6]	1614
2642	reading fp[0]	2126
2643	setting fp[0]	2127

V. Reading and Writing the Battery Backed Non-Volatile RAM

Both the DSP code and the parameter data settings are stored in battery backed non-volatile memory on the BGDSPM. The abbreviation *bram* is used in the code as a prefix to refer to this memory. The width of the battery backed memory is 8 bits. The DSP parameters we wish to store are, however, 32 bits wide. In order to make efficient use of the battery backed memory we store only 16 bits of the parameter where the application does not need more resolution that 16 bits. Still we need some procedures to store these values 8 bits at a time.

To this end we define the structures given in Listing V.1. By creating arrays of 4 byte and 2 byte structures it makes it a lot easier to index through the parameters that are stored in the battery backed memory.

```
/*======== Define structures for data storage in battery backed boot RAM =======*/

// this is a virtual variable structure used to map names to 4-byte BRAM segments
struct nvlong { int nvbyte[4]; };

// this is a structure containing 64, 4 byte parameters
struct nvlongstorage { struct nvlong blong[64]; };

// define type nvlongPtr as pointer to an nvlong structure
typedef struct nvlong *t_nvlongPtr;
typedef struct nvlongstorage *t_nvlongstorePtr;

/*=== Define structure for 2 byte (16 bit) values in battery backed boot RAM ===*/

// this is a virtual variable structure used to map names to 2-byte BRAM segments
struct nvshort2 { int nvbyte[2]; };

//this is a structure containing 4096 (2 byte) values
struct nvshortstorage { struct nvshort2 bshort[4096]; };

typedef struct nvshort2 *t_nvshortPtr;
typedef struct nvshortstorage *t_nvshortstorePtr;
```

Listing V.1 Definition of battery backed memory structures.

Listing V.2 Shows the definition of pointers to these *bram* structures and the base addresses of these structures in the *bram* memory. Note that large application written in C that also use lots of parameters and curves could easily use up the available memory. We need to keep track of how large the code becomes.

Specific procedures were written to write the long and short *bram* parameters. These routines are located in “bgdspm_bramreadwrite.c” which includes the header file “bgdspminit.h”. These files are found in the “./BooLibrary” directory.

Note that currently the short *bram* Read routines do not sign extend the 16 bit data word taken from the battery backed memory to the 32 bit result. This is currently done by the routine that called the short *bram* Read..

```
/*== Define space for parameters.  This leaves 0x1F000 126,976 for program memory =*/

#define BRAMLONGPARAM 0x41F000   // 0x41F000 to 0x41F3FF, 255 4 byte long params
                                 //  1023 bytes of Boot Ram address space
#define BRAMSHORTPARAM 0x41F400   // 0x41F400 to 0x41FFFF, 1535 2 byte short params
                                 //  3071 bytes of Boot Ram address space

/* == Pointers to Non-volatile parameter storage in battery backed boot ram */
t_nvlongstorePtr  bramlongparam;
t_nvshortstorePtr bramshortparam;

       bramlongparam  = (t_nvlongstorePtr)  BRAMLONGPARAM;
       bramshortparam = (t_nvshortstorePtr) BRAMSHORTPARAM;
```

Listing IV.2 Declarations and assignments for the battery backed memory structures.

VI. Restoring and Updating the Application Parameters

It has been described in the previous sections how the application parameters' settings and readings are maintained in the shared memory, and that the current values of the parameters are stored in the battery backed memory. We also declare variables in the internal data memory of the DSP which also contain the values of the parameters. These variables are the ones that are employed in the application's computations since the internal DSP memory can be read and written more quickly than the *bram* memory or even the shared memory. Besides the values of the parameters, we also have DSP data memory variables with the values of the limits and defaults of each parameter. This allows the parameter limits to be check more quickly when new values are set.

The parameter values are often need to be written into an FPGA register where some function is performed. The values of the parameters written to the FPGA's must be updated and confirmed.

The standard procedure is to restore the parameter values stored in the battery backed memory to the parameter settings in the shared memory, and then set the flag for the parameter setting indicating the value has been updated. Following the restore operation at the startup, and then at regular intervals (every booster cycle), the update parameter routine is called. When this routine sees the parameter setting flag set it reads the setting and tests the value against the limits for the parameter. Then the value is written to three locations; the battery backed memory, the parameter reading in shared memory, and the internal data memory variable or FPGA register pointed to by the **param* pointer of the parameters internal data memory structure. The parameter setting flag is cleared.

Listing VI.1 gives the restore parameter routine. Listing VI.2 gives the update parameter routine.

```
//-------------------------------------------------------------------------------
//     Restore parameters from nonvolatile memory at program startup
int restoreparams()
{

       int j,k;

       float fval;
       int   ival;

       for (j=0; j<NINTPARAMS ;j++)
       {
              for (k=0;k < iparams[j].dim; k++)
              {
                     // Read the value stored in the bram
                     bramshortRead((void *)(iparams[j].bram_val + k), &ival);

                     //Sign extend the 2 bytes from bram to 32 bits
                     if ((ival & 0x8000) > 0) ival = ival | 0xFFFF0000;
                     else                     ival = ival & 0x0000FFFF;

                     // Test the value read from the bram
                     if ( ival > iparams[j].ul || ival < iparams[j].ll)
                     {
                            ival = iparams[j].dflt;
                     }
                     *(iparams[j].sm_set + k) = ival; // Write the sm setting
              }
              sm->ip[j].flag = 1;
       }


       for (j=0; j<NFLTPARAMS ;j++)
       {
              for (k=0;k < fparams[j].dim; k++)
              {
                     bramparamRead((void *)(fparams[j].bram_val + k), &fval);
                     if ( fval > fparams[j].ul || fval < fparams[j].ll)
                     {
                            fval = fparams[j].dflt;
                     }
                     *(fparams[j].sm_set + k) = fval; // Write the sm setting
                     *(fparams[j].param + k) = fval; // Write the local parameter
              }
              sm->fp[j].flag = 1;
       }


       return 0;
}
```

Listing VI. Routine for restoring parameter values from the battery backed memory.

```
//     Read and check new parameters at 15Hz; use if within limits, otherwise don't
change
int updateparams()
{
       int   j,k;
       float fval;
       int   ival;

       for (j=0; j<NINTPARAMS ;j++)
       {
              if ( sm->ip[j].flag > 0)
              {

                     for (k=0; k < iparams[j].dim; k++)
                     {
                            ival = *(iparams[j].sm_set + k);

                            if( ival <= iparams[j].ul && ival >= iparams[j].ll )
                            {
                             // Bits are shifted up into LD(31..16) for local parameter
                                   *(iparams[j].param + k) = ival << 16;
                              // Set the sm reading
                                   *(iparams[j].sm_rdg + k) = ival;
                              // Store in bram
                               bramshortWrite((void *)(iparams[j].bram_val + k), &ival);
                            }
                     }
                     sm->ip[j].flag = 0;
              }
       }

       for (j=0; j<NFLTPARAMS ;j++)
       {
              if (   sm->fp[j].flag > 0)
              {
                     for (k=0; k < fparams[j].dim; k++)
                     {
                       fval = *(fparams[j].sm_set + k);

                       if( fval <= fparams[j].ul && fval >= fparams[j].ll )
                       {
                            *fparams[j].param = fval;        // Set the local parameter
                            *(fparams[j].sm_rdg + k) = fval; // Set the sm reading
                            bramparamWrite((void *)(fparams[j].bram_val + k), &fval);
                            }
                     }
                     sm->fp[j].flag = 0;
              }
       }
       return 0;
}
```

Listing VI.2 Routine for updating the parameter values from the setting in shared memory

VII. Definition of Memory Mapped Devices and Registers

For parameters that are written to components external to the DSP chip, a memory map locates the parameters in the address space. Programmable logic will need to have been implemented to decode the address lines to produce the necessary enables and / or chip selects. Listing VII.1 shows the defined constants for the DSP Module in general. Listing VII.2 shows the constants for the C code of the Phase Control application.

```
/* bgdspminit.h - general header file for Booster VXI board initialization*/
/* Bob Webber */


#ifndef _BGDSPMINIT
#define _BGDSPMINIT

/*  specify addresses for board resources  */
#define ADC_BASE 0x00420000
#define DAC_BASE 0x00420003
#define BRAM_BASE 0x00400000
#define SHMEM_BASE 0x00460000
#define FPALTERA_BASE 0x00470000
#define USER_IP_ALTERA_BASE 0x00440000
#define IP0_BASE 0x00440000
#define IP1_BASE 0x00450000

#define BRAMPARAM 0x41F000  // 0x41F000 to 0x41f100, 256 bytes of Boot Ram address space
#define BRAMCURVE 0x41F100  // 0x41F100 to 0x41f500, 1024 bytes of Boot Ram address space

/* specify parameters for board resources ----------------------------------------*/
#define DACBITS 14
#define ADCBITS 14
#define DACMAXV 10.
#define DACMINV -10.

/* (float) DAC bits per volt  == (2^DACBITS / full span), e.g. 4096/20.52.     */
#define DACBPV 798.44

/* (float) ADC volts per bit == (full scale / 2^31), e.g. 10./2147483648.    */
#define ADCVPB 4.6566E-9

/* (float) ADC volts per bit == (full scale / 2^31), e.g. 5./2147483648.    */
#define ADCM10VPB 2.3283E-9

/* bit mask for 32 bit adc input word  == 0Xfffc0000 for 14 bit ADC    */
#define ADCMASK 0Xfffc0000

/* adc output for front panel zero volts input    */
#define ADCZEROV 0X7fffffff
```

Listing VII.1 Memory map constants for the BGDSPM VXI module.

```
//==============================================================
// FILENAME: curvplay.h
//
// This is the header file that defines the constants and functions
// for the Curve Player logic implemented in the ADC/DAC FPGA.
//
//
//  Craig Drennan July 24, 2003
//



/* CONSTANT POINTERs */

//The Curve Player Memory
#define DACTBL_BASE 0x00422000
#define DACTBL_MAX  512         // Max number of curve points

// Enables The Curve Player when a value of 1 is written
// to this address
#define ENA_PTR_SEQ 0x00420016

//The initial memory pointer count from which the pointer
//decrements through the memory to zero
#define WR_SEQ_CNT  0x00420017  //pointer to count register
                                                          // Count can be 0 to 511

//Manual Control Commands.  Writing to these addresses
// performs the described function.
#define START_SEQ_CNT  0x00420018  //Manually start sequence
#define STOP_SEQ_CNT   0x00420019  //Manually start sequence


//Programmable Phase Offset Registers
#define WR_C1_OFFSET          0x0042001A  //Base Offset
#define WR_C2_OFFSET          0x0042001B  //Post Transition Offset
#define WR_BR_OFFSET          0x0042001C  //Bunch Rotation Offset

// Programmable time offset (counts) from start of cycle
// Conversion = 1.28 us / count
#define WR_C2_COUNTS    0x0042001D  //Counts until Post Transition Offset
#define WR_BR_COUNTS    0x0042001E  //Counts until Bunch Rotation Offset

 int update_fpga_params();
 int update_curve_table();
```

Listing VII.2 Memory map constants for the Phase Controller application.