The New Beam Position Monitoring System
Architecture, Algorithms, and Interface

Jeffrey R. Moffitt

The Department of Physics, The College of Wooster
Wooster, Ohio, 44691

Abstract

With the addition of several new hardware components such as a Digital Signal Processor, the new Beam Position Monitoring system will be able to offer a wider range of functionality than the old system. However, this new hardware requires a certain amount of software to provide these new abilities. The aim of this project was to develop the C algorithms, memory and control architecture, and the FORTRAN interface that will direct the hardware in providing this new functionality. To date preliminary versions of each of these have been completed.

Introduction

The fundamental physics behind beam position monitoring involves the collection and monitoring of signals induced on the beam pipe by the passing charged particle beam. In the **C**ornell **E**lectron-positron **S**torage **R**ing the induced signals are picked up by sets of capacitor pickups known as beam buttons. In each **B**eam **P**osition **M**onitor four beam buttons are located radially around the beam pipe. As the beam is displaced from the center of the pipe, the signals increase/decrease linearly for small perturbations. By monitoring the relative intensities of each of these signals, the amount of displacement from the center in both the horizontal and vertical directions can be calculated. Also, by summing the intensities of all of the signals the current of the passing bunch can be calculated. [1]

In the old **BPM** system the signal from each beam button was transferred to a **BPM** processor by a system of relays. This processor recorded the signals from each button and then passed these signals to the control system for position calculation. Unfortunately, in this system the signal from only one beam button at a time could be collected, removing the possibility of single turn monitoring. Also, by transferring the signal data back to the control system, the burden of processing that data fell onto the control system.

The new system, however, will not necessarily relay the sampled button signals back to the control system. The new system has the hardware, i.e. the **DSP**, to allow the signals to be processed onboard each individual **BPM**. [2] Not only will this take the processing burden off of the control system, but it will allow each **BPM** to perform several new tasks. The first of which is a series of self-calibrations, which will account for inherent differences in the digital hardware of each channel, such as pedestal values, slight signal delays between individual channels, and slight differences in the amplification produced by each gain setting. See figure 1. Also, onboard processing will not only allow the **BPM** to calculate the position and current of a bunch for each turn, but it will allow it to do calculations with these positions.

The hardware alone is not enough to create this new functionality in the **BPM**. Software is needed to take advantage of the capabilities of the hardware. The aim of this project was to develop the software needed to run the new **BPM**s. For clarity the development of this software and the final code produced will be presented here in three main sections: *Architecture*, which discusses the basic memory framework and control system access framework, *Algorithms*, which discusses the code that implements these functions on the **DSP**, and *Interface*, which presents the work that has been done to create a user interface for the system and an interface between external data files and the **BPM**.

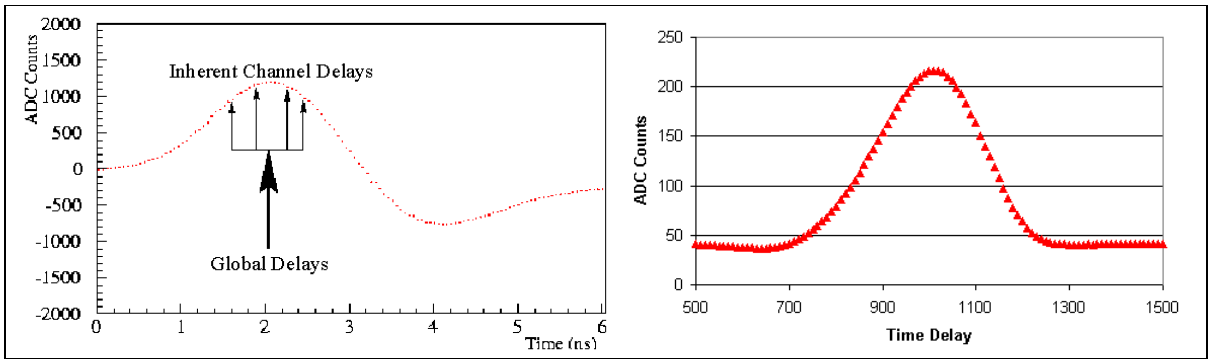

FIGURE 1. These figures illustrate some of the aspects of the hardware that must be calibrated. The figure on the left illustrates the effects of the inherent delays of each channel. If these delays were not accounted for, then even if the global delays were positioned on the peak of the signal, each channel might not be. The figure on the right is data collected by the **BPM** when attached to a pulse generator. The shoulders of the pulse should register zero; however, the pedestal value of the channel gives the pulse a DC offset.

Architecture

The design process of the memory and control system architecture was unique from the creation of the function algorithms in that there were less constraints on final form of the memory and control structures than the code of the algorithms; however, there were still two major concerns that were considered when these two systems were designed. The first was external access to variable values by the control system for the purpose of collecting processed data and for controlling the function of the **DSP**. The second was efficient use of the limited storage space for not only the actual values of the variables, but the limited memory space for the actual program.

Memory Structure

In the current version of the C code of the **DSP**, *CoreV7*, the internal variables are all contained within 7 separate memory structures, which are defined in a sole header file, *Bpm_config.h*. The variables are grouped into structures depending on their role in the code and the level of access that will be needed from the control system. Table 1 contains all of the memory structures, the times in which the control system will access these memory structures, and the functions of the member variables of these structures. Out of these 7 memory structures, only 2 structures contain variables that need to be accessed by the control system during normal operation. The other 5 structures contain variables that will most likely never be changed after initialization. For example the *Host* structure holds the name of the specific **BPM** element, the names of its vector, packet, and timing nodes, and values such as its data base index. These variables will be set once during initialization of the **DSP** and then will never be changed during the running of the **BPM**. The other 4 structures which will only be accessed during initialization contain variables that control the operation of most of the algorithms. Once optimized, these variables will most likely remain unchanged; however, instead of *#define*ing these variables in *CoreV7*, they were left as variables so that the code will not have to be recompiled and reloaded every time a change is desired.

TABLE 1. Memory structures used in *CoreV7*

Structure Name	Control Access	Function of Member Variables
Host	During Init Only	Identify the **BPM** to the control system
Local	During Init and Calibrations	Contain all variables specific to the individual **BPM**, such as calibrations constants
Control	During Manual Function Operation	Control functions whose exact operation must be controlled by user
RunTime	During Init Only	Control functionality of data collection functions
Pedestal	During Init Only	Control functionality of pedestal calibration functions
Delay	During Init Only	Control functionality of delay calibration functions
Beta	During Init Only	Control functionality of betatron phase measurement functions

The other 2 memory structures were created for variables that will need to be readily accessible to the control system. The *Local* structure contains the calibration tables, i.e the table of global delay register values that will place the **BPM** on each bunch in the accelerator, the table of pedestal values for each channel, the table of amplifications produced by each gain register value, etc. Since all of these tables can be changed by self-calibration algorithms, they need to be accessible to the control system. The other multiple access structure, *Control*, contains variables that will control a set of manual functions, which allow the user to have a wide range of control over the operation of these functions. Every time these functions are accessed, only certain variables will change, thus only these variables are located in *Control*.

The use of memory structures in *CoreV7* has helped to eliminate the bulky I/O functions required for variables that are located separately. An advantage to structures is that once the address of that structure is known, an algorithm can access any number of variables without having to treat them distinctly. Before the variables were declared as structure members in *CoreV7*, the I/O functions constituted approximately 17-18% of the available program memory. After the structure definitions, this was reduced to 8-9%, leaving more room for more desirable algorithms.

The other concern in the memory architecture was how to store the large amounts of data that will need to be passed to the control system. This problem is solved through the use of a section of memory, the *pkt_segment*, that is completely accessible by the control system without any intervention by the **DSP**; however, a problem did arise due to the fact that the control system uses the VAX encoding scheme for floating point numbers while the **DSP** uses the IEEE standard encoding scheme. This problem was addressed through the use of specific bit shifting algorithms and will be discussed more thoroughly in the *Algorithms* section.

The DSP CSR Control Structure

While the use of multiple access structures such as *Control*, is appropriate when entering several variables at a time. Navigation through the various function levels within *CoreV7* required a slightly different scheme. This is the role of the DSP CSR Control structure. The difference between this structure and the structures mentioned above is that the DSP CSR has been assigned to a specific region of memory that is accessible from the control system in the same fashion as the *pkt_segment*; thus no I/O functions are needed to transfer the value from the control system to memory or vice versa.

The DSP CSR Control structure contains several different types of variables, which are listed in Table 2, along with the number of members which share the similar functions, and the functions of those members. The variables that control the navigation of the **DSP** through the various functions and their levels are the MODE1 and MODE2 variables. The value of a *#defined* function mask is entered into MODE1 each time that function is entered. MODE2 is primarily used for handshaking with the control system and passing inputed values to their locations in the internal memory. During output, the values being passed to the control system are passed to the member, OUTPUT. The DSP CSR structure also contains a variable, ERROR, where all of the error flags are raised.

TABLE 2. DSP CSR Control Structure Members and their functions.

Member Name	Number	Member Functions
TIMER	5	Record values from the **DSP** C commands, *timer_on* and *timer_off*, which return the number of clock cycles between function calls
DEBUG	5	Indicate the current position of the **DSP** in the running algorithm and store the version number of the core code
MODE	2	Instruct the **DSP** to enter a certain algorithm (MODE1) or handle any handshaking with the control system (MODE2)
OUTPUT	1	Pass values of internal variables to the control system
ERROR	1	Contains any error flags that are raised during function execution

Along with variables that are directly related to the functionality of the core code, the DSP CSR contains members that were used primarily for analysis of the function of the **DSP**. The most used of these variables, DEBUG1 through DEBUG5, indicate the current function and level in which the **DSP** resides and indicate the current version of the core code which is loaded. Along with these are several variables that were set aside to hold different timing values, TIMER1 through TIMER5. These variables take advantage of the **DSP** internal timer by storing the number of clock cycles between two points in an algorithm. This not only allows the user to determine the functioning of the algorithm, but may prove to be a useful internal check of data integrity.

Algorithms

While the development of the architecture described here was certainly done with several appeals to simplicity and size, the ultimate guiding factor was the demands that the algorithms providing the function of the **BPM** placed on the architecture. These algorithms, described below, were the central role of this project. Their purpose is to provide all of the functionality promised by the new **BPM** system.

The types of algorithms written can be divided into three categories: algorithms to provide self-calibration, algorithms to provide data collection, and the I/O algorithm.

Self-Calibrations

Despite the relatively straight forward physics involved in measuring the position of a beam of moving charged particles, the actual collection of data has many hurdles that must be overcome. As mentioned above, a few of these hurdles come from features of the digital hardware. See Figure 1. The self-calibration algorithms were designed to account for these behaviors.

One such behavior is the offset of the **A**nalog to **D**igital **C**onverter output from zero when there is no induced signal on the beam buttons. The **ADC**s are the hardware component that digitizes the signals from the beam buttons. Each channel has its own **ADC** which has its own characteristic count when there is no signal from the beam button. These counts are known as pedestal values and are unique for each channel and each board. If these pedestals were not accounted for, then the **BPM** would measure an incorrect value for the position and current of the beam. In order to measure these values, the *Pedestal* algorithm instructs the **DSP** to set the global delays to 14 ns before a reference bunch, selected specifically for pedestal calibration. This places the **BPM** in a region with no signal. The algorithm then instructs the **DSP** to take N samples for each channel at each gain setting. The average value and the RMS for each channel at each gain setting are recorded in tables in the *Local* structure. The average value can then be subtracted to correct the signals during data collection.

Another inherent hardware feature that must be accounted for is the possibility that the amplification of the signal may not exactly follow the expression $G = 2^{r-2}$, where r is the register value and G is the amplification factor. Because of possible discrepancies in the actual amplification produced, *CoreV7* does not rely on this equation to calculate the actual signal seen by the buttons. Instead, a table of actual amplifications produced at each gain setting for each channel is contained in *Local*. Currently, the only function in which this table may be modified, is the data processing algorithm, *CalibrateGain*. Whenever the counts from the **ADC**s fall outside a defined range, then a gain adjustment function is called. The purpose of this function is to set the gain register so that the **ADC** counts stay within the defined range without producing a discontinuity in the position and current measurement. To prevent a discontinuity the signal produced under the old gain setting must match the signal produced by the new gain setting. To ensure this the function changes the amplification factor of the given channel for the new gain register by taking N samples at the old gain setting and N samples at the new gain setting. The new gain setting amplification factor is calculated by requiring that the average signal with the old gain setting and the average signal with the new gain signal be equal. In order to keep this algorithm from slowly moving away from the actual amplification factors, there is one gain setting for which the algorithm will never change the gain table. The amplification factor for this gain setting will have to be calibrated by hand.

The third major calibration need, is the need to account for any difference in the inherent delays in each channel. These inherent delays place each channel at slightly different places on the signal waveform; thus, the channels will not be reading out the peak value of the induced signal. Again see Figure 1. To correct for this the new **BPM** system has 20 ps delays for each of the channels. The algorithm *DelayCalibration* sets these delays by scanning the peak of the signal over several turns and then fitting a quadratic to the peak. The proper delay is calculated by the parameters produced by the curve fit, which is done by minimizing Chi squared. The resulting matrix is solved by LU decomposition because this allows the resulting parameters to be improved through iteration. [3] Local delays produced by *DelayCalibration* are not stored for each bunch because it is believed that they will not be stable over long times.

Data Collection

With the hardware on the new **BPM**s, the possibility of collecting and processing the signal for each turn during that turn seemed possible. The majority of the data collection algorithms this summer were written with this style of data collection in mind. The algorithm would read the **ADC** counts, check them for integrity, adjust the sign of the stored 32 bit integer to match the 12 bit **ADC** values, remove the pedestal value, scale the values by the appropriate gain factor, and then calculate and store the horizontal and vertical displacements along with the current of the bunch. These calculations were written with the hope that they could be completed in the time span of one turn, 2.56 μs, so that data could be collected from a contiguous series of turns. However, timing of these algorithms has revealed that this style of data collection appears to not be feasible due to the amount of time it takes to perform these calculations.

The new style of data collection that was revealed in *CoreV7* is based on quick collection and storage of slightly processed **ADC** counts. This data is then processed in a separate algorithm and the results are recorded in a circular buffer to which the control system can request access. Timing studies have indicated that there is enough time to remove the pedestals and scale the signals with the amplification factors, if care is taken to ensure that the algorithms are effiecent. In the current algorithm, *CollectADC*, the **ADC** values are loaded immediately out of the **ADC** registers and into variables on the stack to ensure that all of the **ADC** values come from the same turn. These values are then masked to ensure integrity by removing the possibility of influence by the upper 16 bits. The resulting 32 bit numbers are then adjusted to account for the sign of the original 12 bit **ADC** integers. These values are converted to floats and the pedestals are removed from each channel. These signals are then scaled with the amplification factor and made available to other algorithms for further processing by writing them to a large memory buffer.

In **H**igh **E**nergy **P**hysics conditions, the normal monitoring quantities desired will most likely be an average horizontal and vertical position, an average current, and the corresponding RMS values for each of these for N turns. The *RunTime* function provides this, by calling a function to record N turns of raw data and then calling a processing function to return the desired **HEP** monitoring quantities. These values are then recorded to a circular buffer, which contains 254 words of data and two flagging words. One of these words indicates the position in the buffer of the most recent data, allowing the control system to unwrap the data into the correct order. The other word contains the position in memory of the set of data that was collected after the most recent delay calibration. In this function data collection will be a continuous process of alternating data collection and processing, with every N cycles interrupted to recalibrate the local delays. When data is desired the control system will instruct the **DSP** that it is going to read the circular buffer. The control system then waits until the **DSP** has finished collecting and processing any data that it was collecting. The **DSP** then halts data collection until the control system lowers the DSP_WAIT flag, indicating that it has finished reading the circular buffer.

Unlike monitoring during **HEP** conditions, monitoring during the injection process requires position and current information for each turn. The *InjectionMonitor* function accounts for this by processing the raw data that it collects in a different fashion from the *RunTime* function. While data is collected by calling the same function, the data is now processed by calculating the position and current information for each set of 4 signals and writing this information into the same circular buffer that was used in **HEP** monitoring, thus this buffer now holds information on a factor of N less turns. The *InjectionMonitor* will also check the data for any transients that are worth noting, and will stop data collection to preserve the data of these transients.

Another one of the important functions of the **BPM** system is to measure the phase advance of the betatron oscillation produced by a horizontal and vertical shaker around the ring. This phase advance can be a good measure of the magnetic lattice parameters of the ring. The new **BPM** system can accumulate the proper data to calculate the phase advance. The *Betatron* function receives the phase of the vertical wiggler frequency and the horizontal frequency from the turn marker. It then finds the position of the beam and uses the phase to accumulate the amplitude of the Fourier components at these phases for N points. To ensure that no precision is lost these values are stored in several accumulators, which are then summed after the algorithm has finished taking data. These values are stored in the memory structure *Beta*.

Along with the automated data collection functions, there are a few manual functions. The first of these is a manual scanning function. This function uses scanning parameters from the *Control* structure. It simply increments the delays, takes N samples of **ADC** counts without subtracting the pedestals or scaling with the amplification factor. These values are stored in the *pkt_segment* for access by the control system. The second manual function is the *RawData* function, which simply takes N_1 points with N_2 samples each and stores the values in the *pkt_segment*.

I/O Functions

Values are placed into the internal memory of the **DSP** and handed to the control system from the internal memory by the algorithm *IOFunction*. This function consists of a loop for each of the memory structures. When values need to be input into the internal memory, *IOFunction* is called with an input flag. When output is desired, *IOFunction* is called with an output flag. In both cases the specific structure is then indicated by placing the mask of that structure into MODE1. Handshaking with the control system is contained to the variable MODE2. This variable is set to a value defined as DSP_WAIT. As long as MODE2 contains DSP_WAIT, the algorithm will remain paused. During input, the value of the first member of the structure is placed into MODE2, clearing the DSP_WAIT flag. The algorithm then assigns this value to the first structure member and then reassigns MODE2 to DSP_WAIT. This process continues until all of the structure members have been assigned a value. The **DSP** will remain in the input mode of *IOFunction* until MODE2 is assigned the value of INPUT_EXIT.

The handshaking process by which values are handed from the **DSP** to the control system is very similar to the input process; however, during output mode, the first member of the given structure is handed to OUTPUT once a value other then DSP_WAIT has been assigned to MODE2. The algorithm then reassigns DSP_WAIT to MODE2 and waits until this value is changed to assign the value of the next structure member to OUTPUT. Once all of the members of the given structure have been transferred, the **DSP** will remain in the output mode of *IOFunction* until MODE2 is assigned the value of OUTPUT_EXIT.

The input and output modes of *IOFunction* transfer integers and characters to and from the control system without a problem; however, the transfer of floating-point numbers is complicated by the fact that the control system stores its floating-point numbers in the VAX format while the **DSP** stores its in the IEEE standard format. Thus these numbers require algorithms to shift convert one format to the other.

The VAX format and the IEEE formats only differ in three ways. The first difference is that the leftmost 16 bits in the IEEE standard contains the sign bit, exponent bits, and the first 9 bits of the mantissa. In the VAX format this information is contained in the rightmost 16 bits. The second difference is that in the IEEE format the most significant bit is the 1 before the decimal point; thus, the mantissa is normalized to a value between 1 and 2. In the VAX format the most significant bit is the first 1 after the decimal point, which means that the mantissa is normalized to a value between 0.5 and 1. Finally, the third difference is that the IEEE standard biases its exponent by 127 while the VAX exponents are biased by 128. The end result is that the VAX exponent is 2 higher than the IEEE exponent and the 1st 16 bits and the last 16 bits are flipped. In order to compensate for this, *DSPFloat* takes the bit pattern of the Real number placed in MODE2 and subtracts 256, which lowers the bits that will be the exponent by 2. It then flips the first and last 16 bits. *VMSFloat*, on the other hand, multiplies the float it will be passing to the control system by 4, raising the exponent 2, and then swaps the lower and upper 16 bits. Since the bit pattern for 0.0 is all 0s in both formats, *DSPFloat* must check to see if the value is 0 before it subtracts 256. Also, since the VAX format does not recognize -0.0, then *VMSFloat* must convert all values of -0.0 to 0.0 before passing them to the control system.

Interface

While the DSP CSR Control structure and the layout of the algorithms in *CoreV7* make movement into and out of functions fairly easy, there are still some task that need to be done quickly and done accurately. These are the roles of the FORTRAN interfaces.

Initialization of DSP Internal Variables

Before the **DSP** can operate under *CoreV7*, the values of all of the variables in all 7 memory structures need to be initialized. Technically, it would be possible to put all of the values into the memory of the **DSP** by hand using the *vxputn* command; however, this would be incredibly time consuming and tedious especially for all of the **BPM**s around the ring. The interface program, *dsp_init*, is the first version of a final program that will take care of this. *Dsp_init* reads the initialization values out of the file *dsp_init.inp* and then calls all of the necessary interface functions, *vxputn* and *vxgetn*, to place these values into the memory of the **DSP**. The values put into the first 2 spaces of each line indicate to the program whether the following value or values comprise an integer or integer array, a real number or real number array, or a character string. For example, if the line was: $i4, T1B4 =$ 72 5 3 24, then *dsp_init* would understand that it is to read 4 integers off of this line with the first value to be loaded being 72, then 5, then 3, then 24. In this example the global delay table for Train 1 Bunch 4 of one of the two species would be loaded. The program is capable of recognizing several different text delimiters; however, the space seems to be the delimiter that allows the *.inp* file to be the most readable by the user. Also the hyphen is reserved to represent negative real numbers.

The major downfall to *dsp_init* is that the order in which the program is set up must match the order in which the variables appear in each structure. Also, the *.inp* must reflect this same order as well. This is a result of placing all of the **DSP**s internal variables into structures. Inorder to help alleviate the problems that could occur if a version of *dsp_init* was being used to initialize a slightly different version of *CoreV7*, *CoreV7* assigns a version number to the DSP CSR member, DEBUG4. When *dsp_init* begins execution, it checks this version number with its own internal version number and the version number listed in *dsp_init.inp*. As long as these version numbers are changed each time a modification is made to one of the programs, this will ensure that the proper initialization code will be used with the current *Core* version.

User Interface

The primary role of the user interface will be to allow the user to enter the function that is desired and the specific **BPM**s around the ring that the user wishes to perform this function. This interface will also be responsible for watching the ERROR value for each of the **BPM**s and indicating to the user any time an error flag has been raised. It will also be responsible for uploading processed data from the the proper **BPM**s, storing that data in the proper database, and handshaking with the appropriate **BPM**s once the data is received.

A secondary role of this user interface will be to control the manual algorithms. This will allow the user to use these functions to collect data at one set of timing delays and gain values and to also scan through the timing delays. These manual interfaces will be crucial in developing the first version of the global delay table for all of the **BPM**s. They will also be crucial for establishing that the automated algorithms are working correctly.

Future Work

While the preliminary work done here has established a system that is almost completely operational, there are still several things that need to be done. The first of these is the completion of the *Core* code for the **DSP**. This primarily involves the completion of a few more algorithms, such as a precision measurement system which calculates the peak value using the curve fitting algorithms. This will also involve the optimization of memory allocation for program and data memory regions in future versions of the code.

Also, a good portion of the work on the interfaces still remains. The largest of these tasks is to write the actual user interface for the new **BPM** system. Some of the algorithms written here may be of some use in the final user interface, especially *dsp_init*; though, these algorithms will most likely be rewritten from FORTRAN to C before they are used in the final user interface.

Finally, although a good portion of the algorithms have been somewhat finalized, the system still needs to be tested with beam in the storage ring. Solving the problems that arise during these tests will most likely be the last few steps towards the full operation of the new **BPM** system.

Acknowledgments

I would like to acknowledge the following individuals from Cornell University: Mark Palmer and Charlie Strohman for their guidance on this project, John Dobbins and Don Hartill for their work in connection to this project, and Rich Galik and Gerry Dugan for their leadership of this REU program.

Also, it should be noted that this work was supported by the National Science Foundation REU grant PHY-0097595 and research grant PHY-9809799.

References

1. For more information on the physics of beam position monitoring see: *Beam Instrumentation*, Littauer, Raphael. **AIP** Conference Proceedings No. 105. “Physics of High Energy Accelerators”, Ed. Melvin Month.
2. For more information on the hardware and setup of the new **BPM** system see: “An Upgrade for the Beam Position Monitoring System at the Cornell Electron Storage Ring”, M.A. Palmer, J.A. Dobbins, D.L. Hartill, C.R. Strohman, to be published in the Proceedings of the 2001 Particle Accelerator Conference, Chicago, Illinois.
3. W. Press, S. Teukolsky, W. Vetterling, B. Flannery, **Numerical Recipes in C**. Second Edition. Cambridge University Press, 1999.