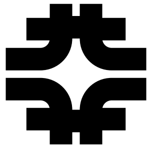

Fermi National Accelerator Laboratory

FERMILAB-Conf-95/141-E
D0

Using Modern Software Tools to Design, Simulate and Test a Level 1 Trigger Sub-System for the DZero Detector

R. Angstadt, F. Borcherding and M.E. Johnson

Fermi National Accelerator Laboratory
P.O. Box 500, Batavia, Illinois 60510

L. Moreira

CBPF-LAFEX/CEFET-EN
Rio de Janeiro, Brazil

June 1995

Presented at the *Real Time 95 Conference*,
East Lansing, Michigan, May 22-26, 1995

Operated by Universities Research Association Inc. under Contract No. DE-AC02-76CHO3000 with the United States Department of Energy

Disclaimer

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

Using Modern Software Tools to Design, Simulate and Test a Level 1 Trigger Sub-System for the D Zero Detector

R. Angstadt, F. Borcherding, M.E. Johnson
Fermilab[1] P.O. Box 500 Batavia, IL 60510
L. Moreira
CBPF-LAFEX/CEFET-EN Rio de Janeiro, Brazil

Abstract

This paper describes a system which uses a commercial spreadsheet program and commercial hardware on an IBM PC to develop and test a track finding system for the D Zero Level 1 scintillating Fiber Trigger. The trigger system resides in a VME crate. This system allows the user to generate test input, write the pattern to the hardware, simulate the results in software, read the hardware result, compare the results and inform the user of any differences.

I. Introduction

The D Zero (D0) Detector is located at Fermilab near Batavia, Illinois. It is undergoing a major upgrade for higher luminosity running with the new Main Injector. Most new systems require software for engineering development, board testing and repair, and system debugging. An important requirement is that the engineering tools are flexible and easy to modify so that prototypes can quickly be converted to finished designs. We have developed a test system which uses an Excel[2] spreadsheet to do all the testing for a level one trigger system for D0.

II. Overview

Most of the digital electronics in D0 reside in VME crates. A VME crate bus may be connected to a Personal Computer with a commercial interface board set.[3] This allows the PC to be used to test the VME hardware.

Microsoft's, MS, Excel V5.x includes Visual Basic for Applications, VB. It does not support "peek" and "poke" operations to physical PC memory directly. Unfortunately the PC Bit3 board requires this feature. The 16 low address bits are output directly to a 64 Kbytes address space and the higher address byte(s) are output through Input Output PC "Ports".

One way to access physical memory under the MS Windows operating system is to use their DOS Protected Mode Interface or DPMI[1]. This is a low level interface requiring the use of assembler or the ability to use Assembler code Inline. Borland "C" is what we used to write a "Library" of Bit3 interface routines made into a Window's Dynamic Link Library, DLL. Any of the functions in this Library may then be called from VB, and even directly from an Excel cell once the function is "registered". Registering tells Excel and VB where to find the function and to load it into memory. Although Excel Version 4.x macro language also allows "registering", Version 5.x's VB provides a more useful and relatively robust wrapper as well as a real programming language.

Once the DLL is created and the Functions are "Registered", a worksheet may be constructed so that user entered data is written directly to the hardware under test.[4] Excel provides a user interface for input. Excel, VB and the DLL transfer the information to the Device Under Test, DUT, read the results from the VME crate and present the results on the display. In parallel VB and Excel simulate the DUT and display any differences to the user. See Fig. 1.

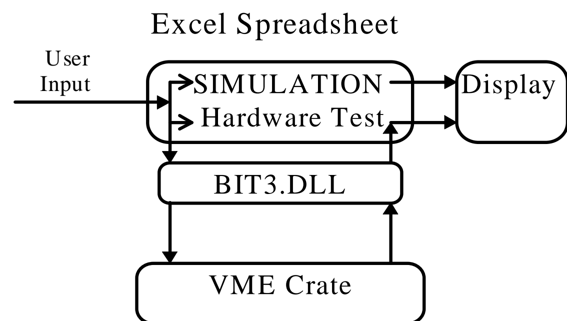

Fig. 1. Functional diagram of Test System. Simulation and hardware test occur in the Excel Spreadsheet, the BIT3.DLL drives the Bit3 hardware. The VME crate contains the system under test.

[1] Work supported by the U.S. Department of Energy under contract No. DE-AC02-76CHO3000

[2] Excel is a commercial spreadsheet Manufactured by Microsoft Corp. We are using it with Microsoft Windows on a PC.

[3] We are using a commercial board set made by Bit3 Computer Corporation Models 403 or 406. The model 403 only supports 24 bit addressing while the model 406 supports 32 bit addressing. The PC boards of both model are for Industry Standard Architecture Bus or ISA, bus. They do not support Direct Memory Access, DMA on the ISA bus.

[4] Within four milliseconds of data entry depending on processor. Further timings available in Tables 1 through 3 of this paper.

III. ABOUT THE D0 TEST

Figure 2 is a worksheet representing one wedge of the proposed Detector Upgrade. The small squares with numbers on them are VB “Textboxes” which represent ~480 round scintillating fibers arranged in sixteen layers logically grouped into four Superlayers. Superlayer “A” has four layers with 16 fibers per layer; “B” has four layers of 24 fibers per layer; “C” has four layers of 32 fibers per layer; and “D” has four layers of 40 fibers per layer.

The present prototype D0 Level 1.0 Trigger Electronics consists of a VME board with several Altera “Flex” Field Programmable Gate Arrays, FPGA’s. This hardware is described in greater detail in another paper[2].

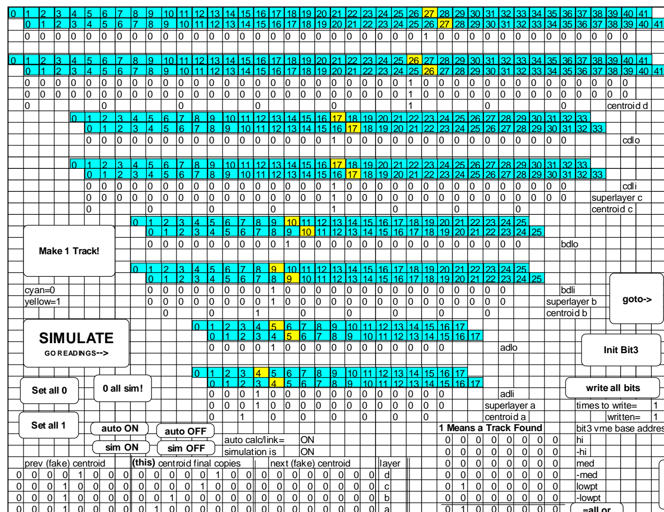

Fig. 2. An Excel Worksheet set up to represent the fibers of one wedge of a Detector upgrade study.

A particle hitting a fiber is represented as a yellow “Textbox” and represented as a “1” in the underlying cell and the VME hardware. Similarly a blue “Textbox” means a fiber was not “hit” and is represented everywhere by a “0”. Adjacent fiber layers overlap. Each overlapping pair is called a “doublet”. The logical “or” operation is carried out on the individual fibers of a doublet pair and is stored in the first row of cells immediately below the dark “Textboxes” The next lower row of cells show the “or”ed results of adjacent doublets *in the same layer.* The third lower row displays a logical “and” with the other doublet in the Superlayer. Eight “centroids” can be formed in this Superlayer. The number of doublets “or”ed together increases from two in the “A” Superlayer to five in the “D” Superlayer. This provides data reduction and holds the number of “centroids” constant: eight per Superlayer[5].

The binary values representing each fiber are packed into words and mapped into VME address space which the FPGA’s use in place of real detector hits. Results of the hardware computation are read back from VME and compared to the Excel simulation results. Differences are shown as a “1” in the spreadsheet.

User input is by 1) “clicking” on a single “Textbox” to toggle a single fiber, 2) have the spreadsheet calculate a track and turn on the fibers that the track goes through, or 3) input Particle “Events” from an externally generated file. Other options exist to set all of the fibers to “1” or all to “0” and for explicit control of the Bit3 card and the VME crate.

IV. EXCEL AS A DEVELOPMENT ENVIRONMENT

Much has been written about the speed of Excel and VB as a Rapid Windows application Development environment[6]. It is not an exaggeration to realize a “hundred times improvement in productivity.”[5]

Excluding doing the ground work of writing the DLL in “C” and learning how to use it, a spreadsheet with “Buttons” was first done in about one week. Users really wanted color and “Buttons” did not have a color attribute so a new sheet was made in a second week using another object which does allow color, “Textboxes”. Automatic track generation took a third week. Reading in a file and adding two side wedges took a final week. Saying that each stage took a week is slightly understating just how rapid development actually is compared to traditional methods. One week includes, the conception of what is desired, algorithm understanding and design, implementation (in a day or two), testing, debugging, and slight refinements added or changes made and some tuning.

The name “Visual Basic” is misleading in that all that is left of the traditional line numberd Basic is some of the “for”, “next” syntax. Line numbers are gone as are the redundant “let” statement. Enhancements include both Subroutine and Function calls. It has scope rules similar to Pascal and “C” as well as support for dynamic variables. The programmer has the option of forcing variables to be declared or not to be declared with the “option explicit” directive similar to some FORTRAN implementations. Block structured syntax is used including Pascal “with” statement syntax and “case” statements.

Having VB and Excel so tightly coupled is an empowering technology. It gives an implementor a choice of doing things in either the traditional linear programming method in a procedural language VB, “C” via a DLL and FORTRAN via DLL with restrictions, or using Excel to hold data in cells that would normally be held in internal arrays. By using a combination of both techniques one can literally build applications in a day. The VB syntax

```
Cells(iRow,iCol).Value = myVariable
```

---

[5] This design is a frozen “proof of principle” design. Work is under way to optimize this idea. See reference [3].

[6] pg. 16 of Reference [4] came up with an acronym, “MARVEL” for Modular programming, Automated interfaces, Rethinking, Visual development environments, Extensible programming languages, Linkage among functions and between applications.

puts whatever is in myVarible in the cell specified by "iRow" and "iCol" of the "Active" worksheet. (The converse is also true.) Thus data may be collected and inserted directly into a "Cell" with a procedural language. Once the data is in the Cells the user may view and further manipulate it with any Excel or user written function.

Alternatively one may read and write directly from and to a cell by placing the VME peek or poke function into the Cell on the sheet. However in this method one relinquishes some control as to when the data is actually obtained. After the first or second time the sheet is re-calculated Excel stops accessing the VME crate because it does not know that it is External! An individual cell may be explicitly updated by clicking on the edit box and then clicking on the little check box of the active cell which is like re-entering the formula and pressing the "Enter" key. This is tedious.

The work around to force the VME bus access to take place is to change either the cell containing the address and/or value of the read or write function(s). Thus if all of the cells containing VME addresses are referenced to a cell containing a "base address", the VME I/O will take place when the base address is changed. A macro may read and temporarily store the base address, turn off automatic re-calculation, write a dummy VME address in the cell that all of the VME functions are referenced to; restore the original address, and finally turns on automatic re-calculation. Excel's recalculation algorithm recalculates all of the dependent cells so that the desired actual external VME bus cycle(s) take place.

VB syntax is *object.method* format which also may be chained: *object1.object2...objectn.method.* An object may be a single object or a collection of like objects indicated by plural tense: "s". Thus *objects* refers to a collection and *object* refers to only one object while *objects("an_object_name")* or *objects(indexNumber)* refers to one member of a collection. The programming paradigm for the most part becomes "find the right object and method for what is desired" and then use it. There are a lot of objects and methods to try to sort through but a very useful "record macro" feature is provided that allows one to record a segment and then edit it.

V. VB NOT "TRULY" OOP AND OTHER SHORTCOMINGS

In Booch's terminology[6], VB is "Object-Based" and not truly OOP, or Object Oriented Programming because VB objects lack "inheritance". VB does not interactively allow objects to "inherit attributes from" other objects ("supertypes [superclasses]"). For example if an VB object ("type", "class"), does not have a needed feature (attribute) then a different object must be used. No provision is provided in VB to extend the object by giving it a new attribute or method. Thus when the "button" object did not have the "color" attribute a new "ButtonWithColors" object can not be made by starting with the previous object and adding new attributes to it. A whole new object had to be found: in this case a "Textbox" was used. Fortunately there are a rather large number of pre-defined and useful objects included. Also other libraries of VB "Custom Controls", "Controls" or "VBX" file(s) which may be purchased from other vendors.

VI. PERFORMANCE CONSIDERATIONS

All timing was obtained visually by observing a Light Emitting Diode, LED, on the VME hardware. The LED showed when the module was being accessed by the PC. The PC used was a Gateway 2000 Model "4DX2-66E" with eight Mbytes Random Access Memory.[7] All timing was with a manual electronic stop watch displaying seconds in hundredths.

The following VB fragment or its equivalent was used as the basic timing algorithm:

```
For longintCount= 1 To longintMax
   integerPlaceVal = VB_Readi(longintAdd)
Next longintCount
```

Three runs were averaged after the extremes were discarded. Values used for longintMax were varied from 3000 Excel cells to 100,000 for the VB and "C" loops to keep the loops long enough to minimize human timing error (5 to 13 seconds).

Initially all "C" code was functionally equivalent to Reference 1. This produced (unpublished) Times uniformly slower by 22 to 25 µs than Tables 1 and 2. Performance gains were made by not calling two Windows Functions", "AllocSelector" and "FreeSelector", each time a VME read or write operation was performed. Instead the address "Selector" was obtained at DLL and program initialization and released at termination. This is contrary to recommended Windows programming practice as there are a relatively small number of Windows "Selectors". However tying up only 1 Selector provides a rather large performance gain and the DLL seems to work as advertised.

All Table timings are in µs. Table 1 is with the Bit3 Base Address in Protected Address Space. Table 2 is with the Bit3 set in Real Address Space. The difference between Table 1 and Table 2 is that ~ 25 µs worth of DPMI code is not executed each VME read or write operation.

	Using DPMI Bit3 > 1Meg=Protected Address		
	call from	read	write
1	VB called in an Excel Cell	3616	3996
2	indirect VB call	64	91
3	direct VB call	46	46
4	all "C" in one "C" module	35	36

Table 1 Bit3 Base Address at $800000. Timings in µs.

[7] This is an Intel 486 externally clocked at 33 MHz with the CPU doubled to 66 MHz. on internal 8K byte cache hits The "E" indicates this is an EISA bus. Recall that the Bit3 Models 403 and 406 are ISA cards.

	Using DPMI Bit3 < 1Meg=Real Address		
	call from	read	write
1	VB called in an Excel Cell	3570	3670
2	indirect VB call	34	52
3	direct VB call	19	19
4	all "C" in one "C" module	14	13

Table 2 Bit3 Base Address at $D0000. Timings in µs

.

Table 3 Timings are included as baselines. Item 4 of this table is from the Bit3 manual. Table 3, Item 4 is to dual ported memory without going through VME bus arbitration. All other times were obtained using the Bit3 as VME bus arbitrator.[8]

Timing 1 of Table 3 is included as a reference point. It cannot be directly compared to Table 2, Line 4 because it is implemented in Turbo Pascal rather than “C”. It might be used to estimate similar DOS based “C” code without some Windows overhead. Both Timings contain similar error handling logic and both Compilers are by the same vendor with optimization off.

	Turbo Pascal DOS Code		
	MS Windows NOT Running		
	Bit Base Address < 1 Meg=Real Address		
		read	write
1	Similar to "C" code above	9	6
2	486 no error checking	3	2
3	Pentium no error checking	2	2
4	Manual without arbitration	1.5-2.0	0.65

Table 3 Bit3 Base Address at $D0000. Timings in µs.

Bit3 writes are two stage pipelined and are intrinsically faster than the one stage read as Line 4 of Table 3 shows. This advantage falls out and even degrades differently on different call stacks. Subsequent timings reflect that Read functions have only one long word argument: the VME address. Write functions have an additional argument: the value to be written to VME. Both functions return an integer used as error reporting.

The first two tables indicate that Excel’s overhead for cell recalculation is on the order of approximately 3.5 ms. One sheet has an estimated 2,520 cells with data and/or functions in them. Recalculation time varied between 8 and 9 seconds. Doing the division gives a cell time of 3.17 to 3.57 ms per cell for the 486 described above. The large variance in time can be attributed to the non-deterministic Excel recalculation algorithm. Using the overlapping region of the two measurements gives about 3.5 ms per cell.

[8] This Turbo Pascal code has the form:
“ival:=memw[$D000:8300]” {ival read from VME memory }
“memw[$D000:8300]=ival” {ival written to VME memory.}
Turbo “C” has similar built in intrinsic arrays and would presumably produce similar instructions. Language and compiler differences at this level running on fast processors are overshadowed by the relatively slow ISA bus making this useful comparison for estimating purposes.

VII. Conclusion

Excel and Visual Basic for Applications is an enabling technology providing excellent and inexpensive tools for Rapid Application Development. Besides being relatively easy to use they are programmable and extensible. If Interface Bandwidth is not paramount then it can be a complete solution.

Windows Performance Penalties to External Interfaces via DLL calls can be mitigated with care as Tables 1 through 3 show. In cases where bandwidth is paramount they may provide a quick intermediate solution as a prototype until a faster application using other methods can be developed.

Increased throughput could be obtained by using Direct Memory Accesses, DMA. Other Windows schemes including Dynamic Data Exchange, DDE, and Object Linking and Embedding, OLE, are expected to be slower and were not investigated.

Special thanks go to D. Huffman and L. Rasmussen, both of Fermilab, for invaluable assistance with the DPMI code.

VIII. References

[1] D.L. Huffman, “Reaching Physical Board Addresses In a PC From Windows Protected Mode” *IEEE Nuclear Science Symposium*, (to be published 1994 proceedings)

[2] paper to be presented at CHEP-Computing in High Energy Physics September 18-22, 1995 Laboratorio de Cosmologia e Fisica, Experimental de Altas Energias (LAFEX/CBPF), Rio de Janeiro, Brazil by L. Moreira.

[3] F. Borcherding, “Level 1 Trigger Design for the D0 Upgrade Central Fiber Detector”, (Internal) D0 Note 2369, Nov. 21,1994.

[4] D.E.Y. Sarna and G.J. Febish, Windows Rapid Application Development, Ziff-Davis Press, 1993.

[5] Electronic Engineering Times CMP Publication, March 13,1995 Issue 839, pp. 1.

[6] G. Booch, Object-Oriented Analysis and Design , The Benjamin/Cummings Publishing Company, Inc., 1991, pp. 38.

[7] J.Webb, Using Visual Basic for Applications, Que Corporation, 1994.

[8] Microsoft Press, Microsoft Excel Software Development Kit Version 4, (this is a book with a disk.) Microsoft, 1993.