ORGANISATION EUROPÉENNE POUR LA RECHERCHE NUCLÉAIRE
CERN EUROPEAN ORGANIZATION FOR NUCLEAR RESEARCH

FASTBUS SOFTWARE WORKSHOP
Geneva, 23 and 24 September 1985

PROCEEDINGS
© Copyright CERN, Genève, 1985

Propriété littéraire et scientifique réservée pour tous les pays du monde. Ce document ne peut être reproduit ou traduit en tout ou en partie sans l'autorisation écrite du Directeur général du CERN, titulaire du droit d'auteur. Dans les cas appropriés, et s'il s'agit d'utiliser le document à des fins non commerciales, cette autorisation sera volontiers accordée.

Le CERN ne revendique pas la propriété des inventions brevetables et dessins ou modèles susceptibles de dépôt qui pourraient être décrits dans le présent document; ceux-ci peuvent être librement utilisés par les instituts de recherche, les industriels et autres intéressés. Cependant, le CERN se réserve le droit de s'opposer à toute revendication qu'un usager pourrait faire de la propriété scientifique ou industrielle de toute invention et tout dessin ou modèle décrits dans le présent document.

Literary and scientific copyrights reserved in all countries of the world. This report, or any part of it, may not be reprinted or translated without written permission of the copyright holder, the Director-General of CERN. However, permission will be freely granted for appropriate non-commercial use.

If any patentable invention or registrable design is described in the report, CERN makes no claim to property rights in it but offers it for the free use of research institutions, manufacturers and others. CERN, however, may oppose any attempt by a user to claim any proprietary or patent rights in such inventions or designs as may be described in the present document.
ABSTRACT

FASTBUS is a standard for modular high-speed data-acquisition, data-processing and control, developed for use in high-energy physics experiments incorporating different types of computers and microprocessors. This Workshop brought together users from different laboratories for a review of current software activities, using the standard both in experiments and for test equipment. There are also papers on interfacing and the present state of systems being developed for use in future LEP experiments. Also included is a discussion on the proposed revision of FASTBUS Standard Routines.
FOREWORD

All participants at the CERN FASTBUS Software Workshop are most warmly thanked for contributing to its success. The speakers are to be especially thanked for their oral presentations and written papers; session chairmen did an excellent job in keeping more or less to the agreed schedule and in animating the discussions; those people 'behind the scenes' at any such meeting were, as usual, so efficient and discreet that, as usual, we hardly noticed them at all; nevertheless their efforts were fully appreciated.

E.M. Rimmer (Organiser)
# CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Author(s)</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Foreword</td>
<td></td>
<td>v</td>
</tr>
<tr>
<td><strong>Review of FASTBUS Software Activities</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Use of FASTBUS Software at CERN</td>
<td>T.Berners-Lee</td>
<td>1</td>
</tr>
<tr>
<td>Status of FASTBUS Software in the FERMILAB Computing Department</td>
<td>A.Cohen, S.Gannon, R.Pordes</td>
<td>5</td>
</tr>
<tr>
<td>FASTBUS Software Activity at SLAC</td>
<td>D.Sherden</td>
<td>9</td>
</tr>
<tr>
<td>Work at NIKHEF on FASTBUS</td>
<td>G.J.Evers, D.Gosman, H.L.Groenstege</td>
<td>15</td>
</tr>
<tr>
<td><strong>FASTBUS in Experiments</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FASTBUS at the Colliding Detector Facility</td>
<td>C. Van Ingen</td>
<td>25</td>
</tr>
<tr>
<td>FASTBUS Software in the NA31 Data Acquisition Environment</td>
<td>The NA31 Collaboration (K.Peach)</td>
<td>28</td>
</tr>
<tr>
<td>Software Environment of FASTBUS ADC's Crate for NA31 Experiment</td>
<td>The NA31 Collaboration (C.Arnault)</td>
<td>49</td>
</tr>
<tr>
<td><strong>FASTBUS Test Software</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FASTBUS Monitor Program</td>
<td>Y.Bertsch</td>
<td>52</td>
</tr>
<tr>
<td>FASTBUS Slave Diagnostic Software</td>
<td>C.M.Story</td>
<td>58</td>
</tr>
<tr>
<td>A System for Test of Slave Functions in FASTBUS Modules</td>
<td>P.Werner</td>
<td>66</td>
</tr>
<tr>
<td><strong>FASTBUS Interfacing Software</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Software for a FASTBUS Segment Manager/Interface Module</td>
<td>B.Yost</td>
<td>75</td>
</tr>
<tr>
<td>A FASTBUS to VME Interface</td>
<td>J.Oliver, J.Rohlf, A.Schwartz</td>
<td>87</td>
</tr>
<tr>
<td>Plans for a CERN FASTBUS Database and Associated Software</td>
<td>T.Kozlowski, K.Osen, C.M.Story</td>
<td>92</td>
</tr>
</tbody>
</table>

vii
CERN Supported FASTBUS Interface Routines
E.M.Rimmer

Proposed Revision of the FASTBUS Standard Routines
J.Marsh, C.Van Ingen, D.Sherden, E.J.Siskind

FASTBUS 'Equipment'

The XOP Processor in FASTBUS
J.Lecoq, M.Moynot, G.Perrot, P.Bahler, C.Liuslin

The Readout Controller of the ALEPH Time Projection Chamber
S.R.Amendolia, F.Fidecaro, S.Galeotti, B.Löfstedt, A.Lusiani, P.S.Marrocchesi, D.Passuello

The ALEPH Event Builder
R.Benetta, A.Marchioro, G.McPherson, W.von Rüden

General Purpose Masters GPMs in the DELPHI Online System
H.Müller

The FASTBUS Segment Extension
D.Machen

The Crate Clustering Card
S.R.Amendolia, F.Fidecaro, S.Galeotti, B.Löfstedt, P.S.Marrocchesi, D.Passuello

******************************************************************************

List of Participants

******************************************************************************
Use of FASTBUS Software in CERN

T.J. Berners-Lee
CERN /DD, 1211 Geneva 23, Switzerland

Introduction

This paper serves to review the use of FASTBUS software in CERN. It does not describe any aspect in great detail: many systems are detailed elsewhere in these proceedings or existing documents.

CERN FASTBUS software [1] is based on the original set of routines proposed by the FASTBUS Software Working Group, and is implemented on a variety of processors and interfaces. It is in widespread use both for hardware development and data acquisition.

Of the application software which has been produced, we may distinguish between that being used for testing FASTBUS hardware and software, and that being used for actual data acquisition. Firstly, though, let us put it in perspective with a look at the hardware configurations on which software is running.

Paths to FASTBUS

Below is an attempt to show on one diagram all the methods currently used at CERN of connecting a computer to FASTBUS:

FASTBUS via CAMAC

At CERN, CAMAC is the most established method of connecting a minicomputer or VME based system to FASTBUS. CAMAC Input/Output registers are used to drive one of the interface modules which reside in the FASTBUS crate. This is largely for historical reasons: there is a lot of CAMAC equipment at CERN, as well as CAMAC-related software.

Direct Interfacing

An alternative to CAMAC interfacing is direct interface from a parallel I/O register to the FASTBUS interface module. The CERN FASTBUS routines have been extended to optionally run through a DRE11 interface from the DEC UNIBUS, or from a VME crate via a CERN-designed “SuperVIOR” Input/Output Register.

The VIRTUS project [2], which uses a farm of General Purpose Master (GPM) modules [3] for on- or off-line processing, used a direct parallel interface between a VAX UNIBUS register and the auxiliary parallel interface of a GPM to form another variety of host—FASTBUS interconnection.

The VALET-Plus Small test system

VME is in extensive use in CERN, one use being as the “VALET-Plus” [4] small test system. This is based on an M68000 processor running in a VME crate, but uses a personal computer as a terminal, and to provide disk and printer support. The personal computer runs a small “bridge” program, which communicates with the VME crate over a 19.2 kbaud RS232 serial line. Bridge software has been produced for the Macintosh, Hewlett Packard and IBM personal computers, and also for the VAX. The Macintosh and IBM-PC implementations are the most commonly used.

The MacVEE approach

An alternative test system along similar lines is the “MacVEE” [5] designed by CERN/EP, and in use by UA1 and Aleph. This uses a direct bus connection between a Macintosh bus and the VME backplane. The software runs on the processor in the Macintosh. The CERN
FASTBUS routines have been ported to the MacVEE.

**FASTBUS Interface devices**

| CFI | SM 1821 | FSEQ | FIORI |

Three interface devices have been designed at CERN, and the LeCroy Research Systems 1821 Segment Manager [6] is also used. The CERN interfaces are

- **The FIORI** (FASTBUS Input-Output Register Interface), a European version of the IORFI module. This is basically a collection of multiplexers and latches allowing a connected computer to do virtually anything with the FASTBUS bus lines, under program control. It is slow, but flexible, and therefore useful for testing slave modules and crates.

- **The Fast Sequencer** [7], by contrast, runs very fast. An ECL sequencer acts as FASTBUS interface, and executes a pre-loaded list of FASTBUS operations. There are several constraints on the list length and content which make the fast sequencer inappropriate for general data acquisition, but the fact that it can drive a slave to and past the speed limits of the FASTBUS specification makes it invaluable for hardware testing.

- **The CFI** (CERN FASTBUS Interface) is the current tool for general data acquisition. It uses a 150ns cycle sequencer for FASTBUS access and data transfer to the host, and has an on-board buffer memory and 68000 processor. The CFI can store a number of lists of general FASTBUS operations, and invoke them on demand. Its principal limitation is the lack of close coupling between the M68000 and the sequencer, making it inefficient to mix list operations with related computation.

**Embedded Processors**

| GPM1 | EB |

Of the various autonomous processors which are being designed to run within a FASTBUS system, two of the most advanced at CERN are the Delphi General Purpose Master [3], the Aleph Event Builder [8], and the Aleph Time Projection Processor [9]. In summary, these are M68000 or M68020 based processor modules, with a large amount of memory and a fast, closely coupled, FASTBUS interface, a number of peripheral interfaces and the option of extension via an M68020 bus connector.

**The “Mac-GPM”**

A recent development is the porting of the VALET-Plus test software to the GPM, making it into a single-board FASTBUS test system. It can be used with the same selection of personal computers as the Valet-plus.

**Software portability**

Of the various processors mentioned which are designed for running application software, the CERN FASTBUS routine library is available for the VAX, NORD, VALET-Plus, MacVEE and GPM. It is worth pointing out that this FASTBUS access software has had to be made portable, avoiding the snags of varying byte ordering and compiler and system facilities on different machines. The library will also provide access through a variety of interfaces and interconnection paths, selectable at run time.

This work, and standardisation, at the system level now allows user application programs to run interchangeably on all any host: a feature much exploited, especially for module testing.

**Graphics Emulation**

Graphics emulation is a feature of the FASTBUS library provided originally for training purposes. At the point when an application program selects the interface it wishes to use for access to FASTBUS, it may instead select the Graphics pseudo-interface. The result is that when a FASTBUS operation is performed, the waveforms which ought to occur on the bus lines are represented on the user’s graphic terminal. This mimics the display of a logic state analyser attached to FASTBUS.

**Direct compilation**

The CERN FASTBUS routines operate by building up a “list” of FASTBUS operations in a common internal format (F-Code), and then translating that into machine instructions for the particular interface involved.

The Aleph Event Builder team have taken an alternative approach to FASTBUS software for
their embedded processor. They use a modified FORTRAN compiler which will recognise a call to a FASTBUS access routine, and replace it with the equivalent in-line code to drive the interface. This method suffers from lack of portability: the compiler must be personalised for every device. It does, however, allow the intimate mixing of FASTBUS I/O operations and normal computation.

The difference, discussed in [10], between these two approaches is a significant design choice (and standardisation problem) for the next generation of FASTBUS software.

Module Testing Software

Of the FASTBUS application programs written at CERN, the greatest profusion is among test programs. This is to be expected, considering that much work is being directed toward LEP, for which many FASTBUS modules are still being prepared. The test programs vary in their areas of speciality, although there is some overlap. Some ideas about the provision of general database-driven test programs will be presented at this conference [11].

The PILS language

The PILS language is dominant in this area, as it makes developing a small program easy. PILS is a BASIC-like language [13]. It runs on VAX and NORD minicomputers, and on the VALET-Plus and "Mac-GPM" systems. PILS is more structured and modular than BASIC, and may be either interpreted or compiled. It is linked to the same runtime libraries as used by FORTRAN and Pascal programs, so PILS programs use the same CERN FASTBUS library.

<table>
<thead>
<tr>
<th>Contact</th>
<th>Emphasis of tests</th>
<th>Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>K.J. Peach</td>
<td>CFI</td>
<td>FORTRAN</td>
</tr>
<tr>
<td>C. Bizeau</td>
<td>CFI internals</td>
<td>PILS</td>
</tr>
<tr>
<td>T. Berners-Lee</td>
<td>General Purpose Master</td>
<td>Pascal</td>
</tr>
<tr>
<td>J. Schiavi</td>
<td>LRS modules</td>
<td>PILS</td>
</tr>
<tr>
<td>Y. Bertsch</td>
<td>General FASTBUS operations</td>
<td>PILS</td>
</tr>
<tr>
<td>E.M. Rimmer</td>
<td>Bit read/write</td>
<td>PILS</td>
</tr>
<tr>
<td></td>
<td>Block operations</td>
<td>PILS</td>
</tr>
<tr>
<td></td>
<td>Single word tests</td>
<td>PILS</td>
</tr>
<tr>
<td>C. Story</td>
<td>FASTBUS Display Module</td>
<td>FORTRAN</td>
</tr>
<tr>
<td></td>
<td>General slave</td>
<td>PILS</td>
</tr>
<tr>
<td>P. Werner</td>
<td>General Slave (individual line level)</td>
<td>PILS</td>
</tr>
<tr>
<td>U. Gruhn</td>
<td>Fast Sequencer</td>
<td>PILS</td>
</tr>
<tr>
<td></td>
<td>Slave, high speed test</td>
<td>PILS</td>
</tr>
<tr>
<td></td>
<td>(\text{translated from Bambi originals})</td>
<td></td>
</tr>
<tr>
<td>B. Heck</td>
<td>Time Projection Digitiser</td>
<td>FORTRAN</td>
</tr>
<tr>
<td>E.M. Rimmer</td>
<td>Segment Interconnect</td>
<td>PILS</td>
</tr>
<tr>
<td></td>
<td>(\text{translated from FDL originals})</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: A selection of Module test programs

Data Acquisition Systems

Much of the work on FASTBUS at CERN is being directed toward the LEP accelerator, as yet incomplete. Here, however, are three examples of the use of FASTBUS for on-line acquisition.

The NA31 experiment

The data acquisition for this experiment is described elsewhere in these proceedings [14]. The NA31 team have taken data using a VAX and a CFI, and also through emulators. They have used their own data acquisition software, and the CERN FASTBUS routines for list building, test and diagnostics.
Time Projection test beams

The Aleph TPC and Delphi HPC detectors are running in test beams. Delphi are using a NORD, and a data acquisition program which calls the FASTBUS routines directly. Aleph are using the CERN VAX data acquisition software, which involves the construction of CAMAC lists to invoke FASTBUS lists.

Useful experience has been gained, highlighting the problems of making a flexible data read-out system using list processing techniques. It was found, for instance, that a small amount of calculation was required at list execution time in order to read out the Time Projection Digitiser slave. This could not be provided by the FASTBUS list stored in the CFI, as the necessary FASTBUS list operations are not defined. It was in fact provided by the CAMAC list, which could include condition jump operations and simple arithmetic, which could be programmed to dynamically create suitable FASTBUS lists. This method has allowed a workable but non-optimal solution.

The UA2 upgrade

The UA2 experiment is planned to have an upgraded read-out system which will use techniques similar to those to be used for LEP. As it will be running a year or two in advance of LEP, the experiment has been seen by many as a pilot system and possible testbed for new ideas.

Many previously unsolved problems will have to be resolved in the construction of the read-out software. Decisions already taken include those to use the CFI and Event Builder devices. The Event Builder, as well as building events, will be used as a read-out controller in the front-frames. It will run Motorola’s RMS68K real-time kernel, with the CERN MoniCa monitor.

Acknowledgements

I am indebted to the many people at CERN who have provided me with details of their work, and particularly to E. M. Rimmer, who introduced me to FASTBUS, and on whose detailed knowledge of FASTBUS systems I have continually relied.

References

Status of FASTBUS Software
In the Computing Department at Fermilab

Andrew Cohen, Stephen Gannon (*), Ruth Pordes
Fermilab (1), P.O.Box 500,
Batavia, Illinois 60510, USA

This paper presents the current status of FASTBUS software projects in the Data Acquisition Group at Fermilab. The Computing Department, which includes this group, supports software running under the PDP-11 RT-11, RSX and VAX/VMS operating systems. Fortran callable subroutine libraries are available to access FASTBUS through the IORFI-II, UNIBUS Processor Interface (UPI) and Lecroy 1821 Host Interface. The FASTBUS Diagnostic Language, implemented and supported at the University of Illinois, is used at Fermilab, where it has been converted to run under the VAX/VMS and RSX-11M operating systems. A first version of a FASTBUS database has been implemented which provides data to a program which automatic generates logical addresses and Segment Interconnect route tables.

(*) Deceased
(1) Operated by the University Research Association under contract to the U.S. Department of Energy
Introduction

The Data Acquisition group of the Fermilab Computing Department has responsibility for supporting the online computing requirements of experiments at Fermilab.

Historically, the group was involved in the specification of the Unibus (Fastbus) Processor Interface (UPI), the debugging and use of the prototype versions and support of the CDF Fastbus test stands before CDF was able to provide their own software support.

At present the group supports a few single Fastbus crate systems in experiments. Experiment 653, with 11 Fastbus crates of Lecroy hardware which is read out through 1821s into an PDP-11/23, provides its own FASTBUS software support; as does CDF for its multi-crate Fastbus system.

The group supports the goal of publishing software routine standards for FASTBUS. To this end, it is an active member of the FASTBUS Software working group; provided the editor of the first standard routine specification; has been active in the Revision Subcommittee; and is now collaborating in editing of the new Standard Routine Specifications.

Software for PDP-11s

An implementation of the April 1983 draft Specification for Standard Routines for Fastbus is provided for the PDP-11 RT-11 single job monitor, interfaced through the UPI running V5 of the microcode.

Dave Lesny of the University of Illinois has written an emulation of UPI Microcode V5 for the I/O Register to FASTBUS Interface (IORFI) interfaced through the DRV-11J to Q-Bus. The University of Illinois has also written and supports an interactive, diagnostic tool for Fastbus (the Fastbus or Device Diagnostic Language - DDL) which uses the 1983 Standard Routine Library. The PDP-11 test stands at Fermilab in general use the Fastbus Diagnostic Language, or Fortran programs which call the standard routine library directly.

An addition has been made to the Fermilab RT-11 Data acquisition program, RTMULTI, to allow specification of Fastbus operations in the list driven event readout. This also calls the standard routine library. This implementation has been used by CDF in a FASTBUS test stand. Experiment 400, an experiment in the Proton area at
Fermilab, read data from four 2-Mbyte FASTBUS memories using the routines directly from their data acquisition program.

The Standard Routine Library and the FASTBUS Diagnostic Language have been converted to run under the RSX-11M operating system — mapping directly to the interface I/O page registers, without a software driver.

**VAX/VMS Support**

The computing department provides a VAX/VMS device driver for the UPI. The driver supports both the UPIs Master Interface - the FASTBUS Segment Driver - and the slave FASTBUS Interrupt Receiver. The driver reads unsolicited FASTBUS Interrupt Messages and distributes them to connected processes.

A subroutine library implementing the April 1983 specification for standard routines has been implemented for VAX/VMS and the Device Diagnostic Language converted to run on the VAX. Extensions have also been made to the program for VMS and RT-11 to support access to CAMAC through the Jorway 411 CAMAC interface used at Fermilab.

**FASTBUS Database/Route Table Generation**

A database containing information on location, addresses, implemented Control Status Registers etc of FASTBUS devices has been implemented in DEC Datatrieve. A Fortran program has been written to generate the Segment Interconnect route tables and logical address assignments for an arbitrary FASTBUS system topology. VAX/VMS command files are available to provide an interface between the database and this program. The program reads the segment topology, address requirements, slot and segment interconnect information for the FASTBUS system, and generates logical addresses for any modules requiring them. It constructs the route tables for the segment interconnects. Some parts of this system are used in the CDF online system.

**Support for Lecroy FASTBUS Hardware**

Several experiments at Fermilab are using Lecroy 1879 TDCs and 1885 ADCs, using the Lecroy Segment Manager/Interface to read out the data. At present the computing department supports control of the 1821 through the CAMAC 2891 interface. Data has been read out into an RTMULTI system, and into a VAX. Versions of a full diagnostic program for the 1821 are provided to run under
RT-11, RSX and VMS. A Diagnostic tool program to provide extended functionality testing for 1821 based systems is also available to run on the three systems.

Support for Microcode development is provided through access to a Microtec Metassembler, a definition file for the 1821 microcode word, and sample microcode programs.

E653 in the Neutrino area at Fermilab, does data acquisition from a FASTBUS system of 11 crates containing 50,000 channels of Lecroy ADCs and TDCs. The Physics Department at Ohio State University, a collaborator on that experiment, provided the microcode definition file. The experiment has implemented a device driver for using the Lecroy 1821 through a DR-11W to 1821/DEC interface on an RSX-11M system.

Future Plans

The Computing Department is committed to providing support for the Revised Standard Routines for FASTBUS. To this end we are converting the CDF VMS based implementation to run on our PDP-11s. This work is in the debugging stage. We are in the final test phase of interfacing the Device Diagnostic Language to the revised standard routines. We will install and support the SLD implementation for the IORFI interface on a Microvax.

In the next year we hope to write a VAX/VMS device driver for the Lecroy 1821 accessed through a DR-11W and intend to extend the standard routine implementation to support this interface to FASTBUS.

Acknowledgements

The FASTBUS effort of the Fermilab computing department has involved many other people besides the authors of this paper. Al Brenner, head of the department up to a year ago, was involved in the original work to propose a new data acquisition bus standard, and encouraged the departments role and work in FASTBUS. Jeff Appel, the associate head, has allocated software and hardware resources to FASTBUS development. The Computing Department is continuing with its commitment for the support and development of FASTBUS. We acknowledge the contributions and collaboration with Dave Lesny at the University of Illinois, Ed Barsotti and Cathy van Ingen at CDF, for work on the IORFI and UPI software; Ron Sidwell and Toby Burnett for their work and consultation on the Lecroy 1821; and the summer and student employees who have worked on FASTBUS projects.
FASTBUS Software Activity at SLAC

D. J. Sherden
SLAC, P.O.Box 4349, STANFORD, USA — CA 94305

ABSTRACT

FASTBUS software at SLAC centers around three sources: the upgraded Mark II detector for SLC, the SLD detector for SLC, and FASTBUS maintenance and development in the electronics department. The Mark II system utilizes FASTBUS for readout of drift chamber flash ADCs and TDCs. Central to the system are the SLAC Scanner Processor (SSP), used for readout, control, and formatting of the data, and a cluster of 3081/E processors, used for more complicated analysis of the data prior to transmission to host computer. FASTBUS software is implemented using a local dialect of the proposed FASTBUS Standard Routines. The SLD detector will use FASTBUS for the readout of all components of the system. Intelligence in the system is provided by SSPs and a cluster of FASTBUS uVAXes. FASTBUS software will use the latest proposed FASTBUS Standard Routines, an implementation of which for an IORFI interfaced work-station is nearly complete. FASTBUS maintenance and development in the electronics department is supported by an LSI-11 based system using an IORFI interface and FORTH as an interactive programming language.

MARK II DETECTOR

The upgraded Mark II detector is scheduled to begin operation at PEP this fall and at SLC at startup in late 1986. Although major portions of the data acquisition will use a pre-existing CAMAC system, the new drift chamber system and the control and monitoring system for the liquid argon calorimetry cryogenics will be implemented in FASTBUS (1). The FASTBUS system is shown schematically in fig. 1 and will use 4 crates of LeCroy 1879 TDCs and 18 crates of SLAC-built 6-bit 100 MHz flash ADCs. Each FADC board contains 16 channels and stores 128 time samples per channel.

Local intelligence in the FASTBUS crates is provided by the SLAC Scanner Processor (SSP) (2). The SSP contains a 32-bit bit-slice processor, emulating a subset (fixed point only) of the IBM System/370 instruction set as well as a set of FASTBUS instructions. It can communicate as either master or slave on either the crate or cable segment. Thus the SSP typically operates as an intelligent crate master, with the capability of acting as a "poor-man's" Segment Interconnect (or a "rich man's" Buffered Interconnect) for communication with other FASTBUS levels.

In the acquisition module crates, the SSPs read each event, perform pedestal subtractions and data sorting, and reformat the data. During calibration, the SSPs read data generated by calibration pulses and maintain tallies of averages and variances. In addition to acting as crate master for each of the acquisition module crates, SSPs are used to manage a cluster of 3081/E processors and to act as traffic managers for the FASTBUS system as a whole, and are used for FASTBUS system verification and initialization.

The computing power of the host VAX 8600 is augmented by a cluster of 3081/E processors (3,4). The 3081/E emulates an IBM main-frame computer, providing the equivalent computing power of 4-5 VAX 11/780s, full support for floating point operations, and support of up to 28 Mbytes of memory. The processor acts as a FASTBUS slave through a dual-ported interface, allowing separate FASTBUS paths for loading and unloading of memory. The 3081/E cluster will be used to provide the final merging of TDC and ADC data, detector monitoring, and full event reconstruction.

For FASTBUS operations from the host computer, a variation of the Standard Routines (5) is used. The major differences arise in part from historical reasons and in part due to unique aspects of the DDI/DR780 interface used. For FASTBUS operations from SSPs a similar but different and more compact set of FORTRAN routines is used in order to maximize efficiency. For truly maximum efficiency, a set of assembly language macros is also available. Source code for the SSPs and 3081/E is written and, where possible, debugged on an IBM 3081K. IBM object modules are then translated to a form suitable for the processors, and linked into an executable module. The executable modules are then transferred to the VAX (or, for 3081/E diagnostic work, to an IBM PC), from which they may be downloaded into the processors. The processors may be downloaded in either a debugging or a batch mode. Debugging features allow one to symbolically ex-
Fig. 1 MARK II FASTBUS System
amine/deposit individual memory locations or registers, read/write common blocks, or dump or disassemble memory. Debugging commands can be entered interactively or through the use of command files.

Additional software facilities include:

- Routing facilities for transferring FASTBUS data through the "system" SSPs to or from those in lower level crates. An interrupt message facility is included, allowing command/response sequences through SSPs.
- Diagnostic routines for SSPs, 3081/Es, acquisition modules, and FASTBUS system initialization and verification.
- Interactive routines for performing FASTBUS operations through SSPs to devices in remote crates.
- Calibration and data acquisition routines for the data acquisition modules.

The full TDC system and 1 module of flash ADCs were used for cosmic ray running this summer. Approximately one third of the flash ADC system (6 crates) and two 3081/Es are being installed for beam running during the fall cycle.

**THE SLD DETECTOR**

The SLD detector, scheduled to begin operation at SLC in 1989, will implement the entire data acquisition system in FASTBUS. The principal elements of the system include:

- A vertex detector consisting of 200 CCD chips producing a total of 50 million pixels with 22 um spacing;
- A 14,000-channel drift chamber system;
- A 33,000-channel Cerenkov Ring Imaging system;
- A 46,000-channel liquid argon calorimeter system;
- An iron streamer tube calorimeter system with 10,000 channels of analog pad and 111,000 channels of digital strip readouts.

The FASTBUS readout and trigger system (6), shown schematically in fig. 2, is arranged in three levels:

- The lowest level consists of data acquisition modules, with each FASTBUS crate controlled by an SSP. For each beam crossing, fast trigger data are read by the acquisition modules; this data is compacted by the lower level SSPs and transmitted to the higher level.
- The trigger decision is made at the second level, which consists of several SSPs. The low 180 Hz repetition rate of SLC allows sufficient time that this function can be implemented entirely in SSP software. Single wire latch data from the drift chamber and a combination of tower energy sums and single bit hit data from the calorimeter are used to form the trigger using fast look-up table techniques.
- At the third level a farm of approximately 20 FASTBUS uVAXen (7) is used to separate physics events from the mostly background triggers. Because the processing already performed at previous trigger level is capable of reducing the 180 Hz beam crossings to an acceptable 1–2 Hz trigger rate, all events reaching the uVAX level will be transmitted to the host and logged. Thus the function of this level is simply to tag events for which full reconstruction is required, without losing the ability to recover events if the filtering algorithms are later changed. This approach significantly reduces the amount of offline processing required and also provides the online analysis, which must operate on a sampling basis, with flags to identify events of interest.

FASTBUS software development at SLD is only beginning. A standardized FASTBUS workstation is being developed using a Micro-VAX computer and an IORFI interface. An IORFI driver for the CDF base version of the revised Standard Routines (8) is nearly complete. A VAX-based IBM cross-assembler has been developed for use with SSPs, and the Mark II SSP translator/linker has been adapted to run on a VAX.

The FASTBUS Micro-VAX (9) is being developed by NYCB Real-Time Computing, Inc., and consists of two FASTBUS boards, onto one of which a DEC Micro-VAX board is grafted. Communication between the Micro-VAX and FASTBUS is managed by an Intel 80186, which also emulates a DEC DEQNA ethernet interface and MSCP disk interface, allowing the Micro-VAX to run the Micro-VMS operating system. The disk emulation operates through FASTBUS to physical disk storage on a host VAX. The FASTBUS list format is designed to be compatible with that developed for the CDF implementation of the revised Standard Rou-
Figure 2  SLD FASTBUS system
times (8). The hardware and major portions of the software and firmware are complete. Completion of two prototypes is expected in less than one year.

Two custom sample and hold chips have been developed for the drift chamber (and Cerenkov ring imaging) and calorimeter modules. The Analog Memory Unit (AMU) chip (9,10) for the drift chamber stores 256 time samples of a single input channel at rates up to 200 MHz, which can then be read out at slower rates (1 MHz) by a single ADC. The Calorimetry Data Unit (CDU) (11) stores up to 4 samples of 32 separate input channels. AMU chips are currently in production, and a prototype hybrid package incorporating 16 AMU chips and associated clocking logic is under test. A prototype CAMAC package has been tested under beam conditions, and work is in progress on calibration and temperature stability problems required to obtain the desired 10-bit accuracy. Prototype CDU chips are currently being tested and will be run under beam conditions this fall.

**ELECTRONICS DEVELOPMENT AND MAINTENANCE**

In the electronics department at SLAC, an LSI-11 based system with FORTH as an interactive programming language is used for prototype development, maintenance, and production testing of instrumentation. Programming is done by people with all levels of programming skill. FORTH has been in use at SLAC since 1976, and has been found to have several characteristics making it highly appropriate to the electronics maintenance and development environment.

- FORTH provides an interactive programming environment which gives immediate feedback to the user in case of error.

- A broad range of programming techniques, from assembly language through prepacaged higher level sequences, can be used within the same programming environment. This allows one to make reasonable compromises between execution efficiency and the convenience of a higher level interactive language. It also allows a matching to the skills of the individuals involved.

- The natural evolution of FORTH constructs as sequences of sequences of operations encourages the modularization of functions and facilitates the debugging of such modules in step by step fashion. The ability to modularize sequences of operations which can easily be combined to form other functional units provides a flexibility which is extremely valuable in diagnostic work.

Support facilities include packages for code editing, up- and down-loading to or from SLAC IBM and VAX computers, a menu system, printing facilities, a primitive file system, CAMAC, FASTBUS operations through an IORFI interface, and FORTH enhancements. A variety of application packages exist for specific modules, e.g. downloading of programs and interactive communication with SSPs.

The LSI-11 systems used to date have run a stand-alone version of FORTH which functions without a higher level operating system. This has been driven by the desire to have direct access to the hardware without the overprotection incurred through an operating system. The lack of a more sophisticated operating system has been compensated for by ready access to the SLAC central computer, whose facilities far outweigh the simple floppy-disk storage capabilities of the LSI work-stations. As the power and storage capabilities of micro-computers have increased, however, this approach is being re-evaluated. A symbiotic version of FORTH which will run under VAX or Micro-VAX VMS is under development.

**REFERENCES**


WORK AT NIKHEF–H ON FASTBUS

G.J. Evers, D.Gosman, H.L. Groenstege,
NIKHEF–H, P.O. Box 41882, NL – 1009 DB Amsterdam

Abstract:

This paper summarizes the present and future activities on FASTBUS at NIKHEF. Since in this workshop 'well-known' hardware modules will already be treated and a review will be given on FASTBUS routines, emphasis will be given on the FASTBUS modules developed at NIKHEF and the software implemented or to be implemented for those modules. A description will be given of the hardware and software test environment for FASTBUS modules. Finally a description of the activities concerning the L3 and DELPHI experiments will be given.
Test environment

NIKHEF has two VMEbus test systems from Compcontrol BV with the following hardware configuration:

• 8 MHz 68000 CPU, 16 Kbyte on-board RAM, occupying one slot;
• 2 x 512 Kbyte memory, occupying two slots;
• 1 floppy disk/SASI interface, occupying one slot;
• 1 8-inch double-sided floppy disk drive;
• 1 20-Mbyte Winchester disk;
• 20 slots VME crate.

The Operating System is CP/M 68K. An assembler and C is supported. We installed RTF68K (1), as a native compiler.

The VMEbus test systems are connected via a serial link to our network of APOLLO workstations.

On the APOLLO workstations all CERN cross-software has been installed, including pushers and linkers, except for the cross FORTRAN compiler, which will be installed as soon as the CERN version, written in PASCAL, is available.

On the VMEbus based testsystems version 2.0 of the PILS (2) interpreter has been installed. The compiler version will be installed later.

Some remarks concerning the system dependent PILS - Host interface:

• Terminal I/O is made via MoniCa (a RAM version of MoniCa is linked with PILS);
• Disk I/O makes use of the TRAP 2 calls of the BDOS (the Basic Disk Operating System of CP/M 68K);
• The external interfaces to FORTRAN libraries, like FASTBUS, HMINI, MINIGD3, are compiled on another host, from which the CUFOM output is sent by a terminal line to the APOLLO's, to be linked to PILS.
Concerning FASTBUS equipment we have:

- 1 FB crate
- 1 FB monitor module
- 1 FB module FIORI
- 1 VME – FB interface complete (FB module CF1 + VME dual I/O register VFIVC)
- 1 FB diagnostic module FDM
- 2 FB active extenders
- 1 FB kluge card
- 1 FB kluge card
- 4 TDC's
- 1 Calibration and Timer module
- 1 Segment/manager interface

So we are able to test the whole chain of CERN cross-software and can use part of the FB routines (3), by implementing a modified version of FBMON, an interactive Fastbus testprogram (4), or FDMTST(5).

Present FASTBUS activities

As NIKHEF is responsible for the readout of trigger data of the muon chambers of L3, a project has been started to construct a muon-trigger interface (fig. 1). The muon trigger interface feeds the first and second level trigger. The data transferred by the interface contains 'hit-cells' only. A cell in the muon detector is defined by a group of 24 wires for the middle muon chambers and 16 wires for the inner and outer chambers (we only treat here "p-measurements", though we also make an interface for the "z-chambers").

Number of electronic channels:
<table>
<thead>
<tr>
<th>Chamber</th>
<th>z-wires</th>
<th>p-wires</th>
</tr>
</thead>
<tbody>
<tr>
<td>MO (outer layer)</td>
<td>3,456</td>
<td>5,536</td>
</tr>
<tr>
<td>MM (middle layer)</td>
<td>-</td>
<td>6,000</td>
</tr>
<tr>
<td>MI (inner layer)</td>
<td>3,520</td>
<td>3,672</td>
</tr>
</tbody>
</table>

---Total---: 6,976 14,208

Twelve FB crates are needed for the total read-out of the muon chambers.

The interface consists of three parts:

1) - The PC
The Personality Card is connected to the TDC via its auxiliary connector and the FB auxiliary backplane.
Its function is to collect hit information of the wires during drifttime. It logically OR-s the information of two adjacent wires, and sets a memory when one of them is hit.
There are 48 of these memories on one PC.

2) - The PCC
The Personality Card Controller, is a FASTBUS module located in the same crate as the TDC's, that deliver their hit information to it via the PC's. One PCC can handle all PC's of one crate.
Its function is to read the PC's in the proper sequence, gather the data of each cell and compare the data with the threshold (number of hit wire pairs) set for that cell.
It then sends its data to the "hit array" of the first level trigger and to the Multi Port Multi Event Buffer of the second level trigger.
3) - The MTC
The Muon Trigger Controller might be a VME module. Its function is to
distribute signals such as:
'start read-out' and 'reset' PC's to all PCC's.

Furthermore there is the PC bus.
The PC bus is a flat cable, connecting all the PC's in one crate to the
PCC.
It consists roughly of three parts:
The data bus (48 bits).
The control bus, which contains signals to control the PC's during data
taking.
A daisy chain, to control the read-out sequence of the PC's.

A prototype of the PC is now available and will be used in the test set-
up.

As the PCC is not yet designed, a 'PCT' (Personality Card Test Module)
is under construction. The PCT simulates roughly a PCC, and a TDC. The
PCT must send commands to the PC's and read data back via the PC bus.

In a further stage one has to do with the grouping of cells over the
TDC's. They must be stored in RAM space of the PCC.

The order of read-out of the PC's by the PCC is a daisy chain type. The
PC's are read in the sequence of their physical location.

Foreseen is in test possibilities, like writing six times 8-bit datawords into
the PC hit-memories and then reading them back.

External control must be implemented, for example:
as input to the PCC:
enable collect, start encode, clear hit memory, stop encoding, and as
output from the PCC:
encode busy, FASTBUS busy, start error, PC error.
For a more detailed description, see reference 6.
Future FASTBUS activities

A small prototype of the middle layer of the muon chambers, for momentum measurement, has been constructed (fig. 2). It consists of $5 \times 24$ wires, from which at least in the beginning, 24 will be connected via a pre-amplifier and a discriminator to a few TDC's. These TDC's will be read out by the 1821 LRS segment/manager interface.

Only programs for the read-out of the TDC's are foreseen at the moment — just to test the prototype in the magnet field in the test bundle at CERN next year.

Later on data has to go, via an ECLine and an LRS 1891 memory on the sub-detector crate to the event builder, a combination of the General Purpose Master and the Block Mover (7).

The inner detector of DELPHI is under construction at NIKHEF (8).

The vertex chamber has 24 sections of 24 wires on which accurate drift-times need to be measured. A number of 576 channels has to be read out by LTD's.

The number of channels of the outer trigger part is substantially higher — $5 \times 160$ anode wires and $5 \times 200$ cathode strips. The wires and strips will be treated in the same way: a pre-amplifier with limited band width sends the data to a 7(8)-bit FADC.

So 1,800 channels amplitude measurement by FADC's is necessary. As discussions on the design of the LTD's and FADC's are still going on, it is difficult to give a detailed read-out scheme. Also the third-level trigger is not 'fixed'. It is clear that at the moment we are in the design phase, but we hope to start hard- and software activities on FASTBUS within 1 year.
Conclusions

A test environment for FASTBUS has been created. CERN FB routines have been implemented. The writing of test programs will start this year. In the beginning they will not be too complicated, due to the fact that part of the hardware is still in the design phase. Concerning the read-out, discussions are still going on, and one is still in the stage of proposals.

Figure Captions

(1) - The L3 Muon Trigger Interface.

(2) - The prototype of a middle-layer muon chamber.

REFERENCES


-- -- --
FASTBUS at CDF

C. van Ingen
CDF Computing Group
Fermilab M.S. 318
Batavia, Ill. 60510 USA

The CDF data acquisition represents one of the first large scale uses of FASTBUS. This year, a reduced, single partition system, consisting of thirteen crate segments and three cable segments, was commissioned for the 1985 engineering test of the CDF detector. The one partition system contains many of the features of the planned multi-partition system to be used in late 1986.

General Programming Environment

The CDF online system is VAX based and the UNIBUS Processor Interface or UPI provides the necessary access to FASTBUS. The FASTBUS master portion of the UPI is a microcoded list processing engine with DMA hardware on the UNIBUS and an ECL sequencer capable of entire block mode transactions (arbitration, primary address cycle, secondary address cycle, data cycles, address release, bus release) on the FASTBUS. The UPI also includes a FASTBUS Interrupt Receiver and can be enabled to interrupt the host processor upon detection of a FASTBUS Service Request.

All online CDF FASTBUS software uses the Revised or Revisionist Standard Routines for FASTBUS. The initial baseline package was released for use on October 1, 1984. The implementation has appeared stable to the users since February 15, 1985, although internal improvements for execution efficiency and addition of seldom used functions are still being made at a low level. The implementation required the redesign of the UPI microcode; the UPI now uses the Revisionist "generic" list structure.

The Revised Standard Routines mandate much functionality within the implementation such as explicit buffer access modes and the automatic error reporter. These functions greatly simplify the application code; the resulting code is also more uniform, hence more maintainable. The routines also emphasize tools for list processing. This has also tended to
increase the execution speed of user code and reduce the I/O load on the
host machine in the CDF environment, since users can make use of delayed
mode execution with relative ease.

All CDF system resources are accessed via the Resource Manager and all
FASTBUS devices are considered to be resources. Devices must be booked
by name (RABBIT_SCANNER_2) prior to any access. The Resource Manager
also supplies all relevant information from the FASTBUS database, cabling
data base, or other system configuration database. Thus, a user must book
a FASTBUS device and inquire as to the device primary address prior to
attempting to address the device. At present, no protection is actually
implemented, only the information retrieval functionality is present.

Major System Components

The Buffer Manager directs all data flow within the CDF system and keeps
summary statistics on the event stream. All communication between the
various hardware modules is in the form of FASTBUS interrupt messages.
The hardware version of the Buffer Manager will be based on a µVAX II and
VAX ELN with a modified Qbus UPI. The present software Buffer Manager is
running on the host VAX and was coded in a combination of ELN and Pascal.
This allows much of the code for the eventual hardware version to be used.
Also, the ELN compiler has proven to be extremely efficient, for example it
offers an INLINE switch which forces inline code generation circumventing
the usual CALL instruction overhead and yielding very fast code.

The Event Builder is responsible for the accumulation of all data from the
front end scanners. It must read out each scanner and perform a rapid
scatter-gather of the data inorder to reorder it in a more optimal fashion
for later analysis and monitoring. At the present time, only the software
Event Builder exists. This version also runs on the host VAX filling a
shared global event buffer. The hardware version is scheduled for
installation in Fall 1986, although it seems likely that some intermediate
solution will be necessary due to time constraints.

The Level 1 Trigger System is totally packaged in FASTBUS. The system
must be downloaded and all DACs calibrated via FASTBUS prior to the
initiation of any physics run. Charge injection pedestal runs are also with
simultaneous trigger readout are required for verification of the trigger
activity. The combined Level 1/Level 2 system will fill twenty-six
FASTBUS crates; only three crates are presently in use.
The Trigger Supervisor or TS is responsible for coordinating the activities of the front end scanners and the trigger system. The TS acts on the basis of dedicated lines to the Trigger System and the master clock system. The readout by the front end scanners is initiated by the TS via a global Start-Scan broadcast. After the detection of the scanner DONE from each, the TS sends a FASTBUS interrupt message to the Buffer Manager denoting the acceptance of a new event. If at least one of the enabled front end buffers is free, the TS then re-enables triggering.

Two types of front end scanners are in use. The calorimetry readout scanners, the MXen, are responsible for packaging the data from the front end RABBIT system. The MX is not packaged in FASTBUS, but is accessed from FASTBUS via a simple slave, the MEP. The MX must be downloaded with the readout algorithm, channel readout lists, channel gains, channel pedestals, and channel correction coefficients. The MX algorithm switches readout list in response to distinct special purpose start scans. The tracking data readout is performed by the second scanner type, the SLAC Scanner Processor or SSP. The SSPs scan LeCroy 1879s and other data compaction modules. The SSP was intended to be accessed as a slave on a FASTBUS cable segment and act as a master on a FASTBUS crate segment. At present, three SSPs are in use; two are presently accessed only through an SI. The SSP algorithms are written in standard FORTRAN; a current project is the development of a post processor for the VAX linker to enable code to be developed in two stages on a VAX, rather than in five stages on an IBM as presently done.

**Future Work**

In the next year, much work is left to be done. The CDF multiple partition system must become operational and the hardware versions of the Buffer Manager and the Event Builder are to be integrated into the FASTBUS network. The work to date has been primarily directed towards achieving the functionality required, work to increase the throughput is clearly in order. Finally, the first prototype Level 3 Trigger System will be attempted. The primary Level 3 system processor environment will be provided by the FERMILAB Advanced Computer Project. The necessary system integration will be performed by both ACP and CDF personnel.
FASTBUS Software in the NA31 Data Acquisition Environment


Abstract

The NA31 experiment on CP-violation by the CERN-Dortmund-Edinburgh-Orsay-Pisa-Siegen collaboration is the first at CERN to take data using a full implementation of the FASTBUS Standard. All of the event-associated data is read via FASTBUS, either directly through a CFI or indirectly through a FASTBUS Block Mover/168E/Buffer Memory into a VAX 11/750. The features and performance of data acquisition using this configuration are discussed.

(presented by K.J. Peach, University of Edinburgh)
1. Introduction

The CERN-Dortmund-Edinburgh-Orsay-Pisa-Siegen [1] experiment on CP-violation at the CERN SPS is the first experiment at CERN to use a full implementation of FASTBUS [2] for data acquisition. The experiment had a short run in July and August 1984, with only part of the FASTBUS system installed, and had a run in April-August 1985 with the complete system. This paper describes briefly the software environment for the 1985 run.

2. FASTBUS configuration

The configuration of the FASTBUS and data acquisition system is shown in Figure 1.

Figure 1: Configuration of FASTBUS and Data Acquisition System
The main components of the system are:-

1. The FASTBUS ADC system under the control of the PDP 11/23*. This is described in the paper by C Arnault [3] submitted to the conference.

2. A VAX 11/750 to control the experiment (via CAMAC) and to collect the data, write tape, monitor etc. Two CFI's [4] and a FIORI [5] allowed the VAX to access the FASTBUS system.

3. A twin 168E system [6] accessed from the VAX through a CFICC [7] and CAMAC. This system is described in reference 6. The 168E's were used to filter the data, based on a partial reconstruction of the events, and accepted events were placed into a 2MByte buffer memory, for subsequent transfer to the VAX.

4. Three FASTBUS Crate segments and one FASTBUS Cable segment.

One of the FASTBUS crate segments (called the DAQ Crate) contained all of the modules to be read for a single event; these modules have been designed and built within the collaboration, and are briefly described in Appendix A. One special feature of all of these modules is that they had a single-event internal buffer, thus creating a one-event pipeline in the readout chain, and hence reducing the deadtime. The DAQ Crate also contained a CFI for direct event readout by the VAX, a Segment Interconnect [8] to access the cable segment and the second (Auxiliary) crate, and the FASTBUS Block Mover [9] (BM) which transferred data from the DAQ crate through the Switch Coupler Card [10] (SCC) to the 168E system. The auxiliary Crate (which was used mainly for testing purposes) contained the Cable Segment Terminator [11] (CST), a Segment Interconnect, CFI, FIORI and a Fastbus Display Module [12] (FDM).
3. FASTBUS Software Components

The software for this rather simple system must accomplish the following tasks.

1. System Initialization
2. System Testing and Diagnosis
3. Module Testing and Diagnosis
4. Data Acquisition

In fact, only a few programs were required for these tasks. Most problems could be diagnosed without the need for special programs, once the modules had been commissioned. The various software components are described below.

The CERN Standard software routines [13] were used in all applications. However, for some programs, only the List Compiler was used, the compiled set of instructions being stored on disk for later execution. A standard program (BUILDLIST) was written to compile a set of FASTBUS actions (see Appendix B) from a simple command sequence. The execution of these lists did not use the CERN List processor, but a set of CFI-specific routines which handled the CFI Command-Reply sequence and associated DMA using optimized CAMAC QIO's; this reduced the CPU overhead considerably, and was also about a factor of two faster in real time as compared with the standard routines using delayed list execution. Programs which used this second set of routines are indicated with an asterisk. All programs were written in VAX FORTRAN, and many assumed that the terminal was compatible with a VT100; these programs are not therefore easily transportable to other systems.
3.1 System Initialization

A single program FBRESET initialized the three-segment FASTBUS system from data stored in a simple database. Because the system was small, the complete database could be held in memory. The initialization proceeded in three stages.

1. The interfaces were initialized

2. All segment interconnect route tables and registers were initialized and loaded, and the ancillary logic registers were set appropriately.

3. All segments were scanned for modules in each physical station on the segment (0:25 for a crate, 0:31 for a cable). The results of this scan were compared with the previous contents in the database, and changes were indicated; if there had been a change, the database could be updated on disk if required.

It should be noted that even for this small system, the time taken to initialize was about one minute. This was partly because the responses and loading of registers were checked at each stage during the initialization by the VAX, and partly because the initialization was controlled directly by the VAX, instead of (as would be better) from some intelligence in the FASTBUS system itself, which could perform the initialization after being loaded with the database.

3.2 System Test and Diagnosis

Much of the system was tested by the FBRESET program. However, a more thorough test was required before the multi-segment system was in use, and to test those parts of the system (particularly the AD lines between the GA and GP fields) which were not used in system initialization. The programs developed for these purposes were:

1. CFITEST: This was a general program which used an auxiliary memory (usually the FDM) to write, read and compare test patterns in single word and block transfer geographical and logical addressing; it was a combined test of the CFI, FDM, ancillary logic and (if used over the cable segment) the SI's and cable segment.
2. FASTCOMP (*): The CFITEST program was too slow for high data volume testing of the cable segment and the SI's. A more simple program, which used only block transfers and geographical addressing with a more limited range of test patterns (only random integers) was used to write, read and compare over the cable. The random numbers were changed by one location, and one new random number was created, for each loop.

3. TESTGAC: This program tested the responses of the ancilliary logic (essentially CSR3) on a directly connected crate segment using an auxiliary FDM. Because it used the FASTBUS primitive routines, this program could only use the FIORI interface.

4. SITEST: This semi-interactive program allowed the loading and testing of Segment Interconnects.

3.3 Module Testing and Diagnosis

There were two types of test programs for modules. Each module had its own test program, usually written by, of interest to and often only understood by, the design team; these are not discussed here. However, once the module had been commissioned, it was often necessary to check from time to time, or when there were problems, simple module-independent functions, for example, to check whether CSR0 was readable, or to discover the settings of the CSR0 bits. These programs are described below, in roughly the order of increasing complexity.

1. FBSTUDY [14]: This program usually used the FIORI interface, and allowed essentially static or 'slow-motion' testing of modules. It had access to all of the CERN standard access mode routines, including the primitives. It also had a simple facility for storing useful test sequences, for executing sequences of operations and for looping. However, its main use was as a very low level diagnostic program, for stepping slowly though a FASTBUS sequence, or for setting the system into a static state for hardware investigation.

2. CFiloop: This program used the CFI to execute singly or repeatedly the eight functions corresponding to the routines FRD, FWD, FRDB ,FWDB ,FRC ,FWC ,FRCB ,FWCB.
For output, there were several options available - user defined, alternating, all 0, all 1, incremental, bit-shift and random. This program was most useful when testing individual functions of a module and (by using the CFI EXLOOP facility) to allow engineers to fault trace at high speed without using the VAX CPU.

3. EXECLIST (*): This program had similar facilities to CFILLOOP, except that a more complex sequence of operations could be executed, including full data acquisition execution with a simulated trigger.

4. BMTALK [15]: This program was used to control and thus to test the Block Mover, but could also be used with an auxiliary memory to test source modules at faster data rates than was possible with the CFI.

5. TALK [6]: The purpose of TALK was to study, test and debug the 168E hardware, to perform timing measurements, and to develop a library of utility routines from which the final data acquisition task was constructed. It was possible to write, read, compare and dump all six memories in the system, to run programs on the 68000 processor on the Microprocessor Controlled Interface (MCI) and the 168E's, and to perform test loops etc. Together with BMTALK, it allowed a check of the complete data route through the BM/168E system using events downloaded through a CFI.

Several more specific programs have been developed to test particular functions by modifying the EXECLIST program - for example the FASTCOMP program to gain speed, or perhaps to add some CAMAC action for specific modules. The development of these programs from EXECLIST required rather little effort.

3.4 Data Acquisition Software

The data acquisition system was written by the NA31 collaboration to control the experiment, and to acquire the data via the FASTBUS, either directly through the CFI or indirectly through the BM/168E system. The software was written in VAX FORTRAN, and consisted of several independent programs, communicating mainly through a 2MByte shared memory area.
(the DAS Buffer). The main components of the system are described briefly below.

1. A suite of routines (The Micro-controller [16] ) to control access to the DAS Buffer.

2. A supervising STATUS_MONITOR program. This program took no part in the management of the data acquisition system, but checked continuously for pathological conditions such as a program crash while locking access to the DAS Buffer, and tidied up the tables to allow the system to continue. This program also maintained a display of the current state of all data acquisition tasks, the state of the DAS Buffer etc.

3. A Master Control program which allowed the operator to control the experiment and the data acquisition tasks (via the DAS Buffer). This was the only program with which the operator communicated directly.

4. An Event Display program [16]

5. A Histogramming and Plotting system [17] to accumulate and display information produced by the data acquisition tasks, and controlled from the master program.

6. Four producer tasks to place data into the DAS Buffer. These task were:-

   a. ZFASTCAM (*) to control the experiment through CAMAC IO registers, to handle all real-time interrupts and to acquire data through the CFI. This program also exercised direct control over the functioning of the Z168E program.

   b. Z168E to initialize and setup the 168E system, and to empty the 168E system buffer memory at the end of each data-taking burst.

   c. PLAYBACK to read data back from tape.

   d. ZMCDATA to read Monte Carlo data from disk.
PLAYBACK and ZMCDATA have no connection with FASTBUS, and serve in any case obvious functions; the other producer tasks are discussed in more detail below.

7. Up to eight consumer tasks. Three tasks ran permanently, the other five were allocated and run as necessary, for detailed checking of particular parts of the data and equipment. The three principal tasks were;

   a. ZMAG to write the data to tape. This program accepted all the data.

   b. RUNSUM to monitor a sample (1/n) of the events

   c. FORMAT to check the structure of a sample (1/m) of the events.

Typically, RUNSUM analysed 1/20th of the data and FORMAT 1/100th.

At saturation, the CPU time was roughly 30% for ZMAG, 15% for Z168E and RUNSUM, 5% in FORMAT and the remainder distributed amongst the other tasks in the system.

3.4.1 ZFASTCAM

This program was responsible for the direct control of the experiment through CAMAC, and for servicing all real-time interrupts associated with the SPS machine, as well as taking data through the CFI.

The SPS machine super-cycle of 14.8 secs was divided into five cycles.

1. Cycle 1, of length 6 secs, during which time

   a. Z168E emptied the 168E system buffer memory

   b. the PDP11/23+ had control of the FASTBUS ADC system
2. Cycle 2 of length 3.7 secs, to allow the PDP11/23 to complete its task, and to prepare for the next cycle.

3. Cycle 3 of length 0.3 secs, during which two or three calibration events were read through the CFI by ZFASTCAM. (There was a small amount of CAMAC read in association with these calibration events.)

4. Cycle 4 of length 2 secs to prepare for the next cycle.

5. Cycle 5 of length 2.8 secs, during which data was taken either directly through the CFI by ZFASTCAM, or indirectly through the BM into the 168E system, in which case ZFASTCAM was quiescent.

The only interrupts serviced by the VAX were associated with changes of cycle. For data taking during cycle 5, of necessity for the BM/168E system and by choice for the CFI, the individual event 'interrupts' were fielded directly by the interfaces and associated electronics. This had a particular advantage for data acquisition through the CFI, which allowed several events to be read into the VAX during a single DMA, thus reducing considerably the system overhead.

The other tasks performed by ZFASTCAM were principally:

1. Before each run, the DAQ crate was scanned to find the modules in the crate, and the result compared with the contents of the crate stored in the database. From this scan, the appropriate Fastbus lists were constructed for execution at the start of run (SOR), start of burst (SOB) and for data acquisition through the CFI (CYCLE5D). The scan list for loading into the BM was also constructed.

2. At the start of run and start of burst (end of Cycle 1) the appropriate list was executed by the CFI to initialize and reset the Fastbus modules.

3. Various experiment modules were set or reset through CAMAC as appropriate at changes of Cycle.
4. At the end of each burst, a set of CAMAC scalers were read, to give information, for example, on beam intensities etc.

5. Monitoring functions were provided for subsidiary tasks as required.

The FASTBUS lists for execution were constructed from prototype lists stored in the database (requiring only the updating of the primary address of each module) for each of the three lists SOR, SOB and CYCLE5D. There was no CERN FASTBUS software directly used by ZFASTCAM.

ZFASTCAM also controlled directly Z168E as necessary. At the beginning of Cycle 1, Z168E was launched to empty the 168E system buffer memory, and ZFASTCAM waited for this to complete before continuing to Cycle 2.

3.4.2 Z168E

In contrast to the multiple tasks performed by ZFASTCAM, the program to control the 168E system was dedicated only to this task. However, the loading and initialization sequence for the 168E system was complicated, requiring the loading of the control program into the Micro-Processor Controlled Interface (MCI), and then the appropriate Filter program and data constants into the two 168E processors.

To increase the reliability of the system, many tests were built into the software, for example the downloading of all programs was verified by reading them back and comparing. Also, when the buffer memory was emptied, the integrity of the data structure was checked before events were released to the DAS buffer.

During each burst, the Z168E checked the status of the 168E system, and tried automatic recovery of the system in case of fault.

To facilitate diagnosis of problems, as well as monitoring the system performance, error reporting included a complete traceback of detected error
conditions; error messages and accounting information were also written into a journal file.

4. Performance details

Some brief details of the performance achieved with this configuration are given in this section, to allow evaluation of the potential of the FASTBUS system through the two interfaces used in this experiment. The actual performance depended on many factors, such as event length and trigger rate, and within this experiment the actual data rates depended strongly on the type of trigger and the beam. These performance figures should then be considered as being 'typical' only for this application. It should also be stressed that there are several ways in which these figures could be, and will be, improved before the next running period in 1986.

4.1 Data Acquisition through the CFI

The data rate through the CFI was limited by two factors

1. The CAMAC transfer time of 1.9\mu s/16-bit word.

2. The need to format the data from the 12 modules to allow fast decoding in the VAX; this took about 200\mu s/module.

With these constraints, the real time to read one event was about 4msecs. In addition, the overhead on launching a preloaded list, and starting the DMA was about 30msecs; however, this was reduced to about 4msecs/event by taking eight events in a single DMA operation, leading to an average time per 2kbyte event of about 8msecs, or 300 events in a burst of 2.8 secs (600kbytes of data).
4.2 Data Acquisition through the BM/168E system

The limitations on the BM/168E system are more difficult to define precisely. The time taken to read a typical event was less than 100μsecs, including formatting the data in the same way as the CFI. However, the filter program took a minimum of 800μsecs to process an event, and if the full filter was run, this rose to more than 1.5msecs. With the trigger rates in the experiment, and with the particular filter algorithm and philosophy used in the 1985 running, the deadtime was typically 30%. At this figure, the data rate was 1000 events/burst into the 168E system, and (after filtering) 500 events/burst into the VAX and on to tape. (The filter program rejected about 70% of the data, but 25% of the data was written to tape independently of the filter program result, in order to monitor the performance of the system.) This data rate in fact saturated the tape-writing capacity available in 1985.

5. Future Developments

There are some changes required in the data acquisition system before the next run, in order to meet the requirements of the experiment; apart from general improvements in the trigger and the filter algorithms, the changes which affect FASTBUS and the associated software are:

1. Change the CFI interface from CAMAC to Unibus, using the UFI [18] system.

2. Since the CFI and the 168E system both use the same type of CAMAC interface, a similar change from CAMAC to UNIBUS could also be made for the 168E system as for the CAMAC.

3. Install a fast FIFO memory on a second cable segment between the BM and the SCC, in order to reduce the system deadtime

With these improvements, the data rate into the VAX directly should improve by perhaps a factor two, and into the 168E system by about the same.
6. Summary and Conclusions

A brief description of the main software components for the NA31 data acquisition system has been given. The system has performed reliably during the 1985 run, and some 40 million triggers have been written to tape. The performance of the system requires further enhancement for the 1986 run.

Acknowledgements

It is impossible to acknowledge all the contributions to the work described in this paper. We would therefore like to thank all of those people who have helped in setting up this experiment, and who have offered advice and assistance.
APPENDIX A

BRIEF DESCRIPTION OF THE SOURCE MODULES

A brief description of each of the source modules in the DAQ crate for the data in this experiment is given here; further details are obtainable from the module designers.

A.1 Pattern Unit (F680E)

This module contained 128-bits of information, strobed in 16 groups of 8 bits, and read as four 32-bit words. The module was designed and built at Pisa.

A.2 Wire Chamber Buffer Card (F6813)

This module read the data from the wire chamber encoders, and delivered the data in a maximum of 17 words (one status word, and then a maximum of four hits for each of the 8 wire chamber planes, packed two into each 32-bit FASTBUS word). This module was designed and built at Siegen.

A.3 Time to Digital Converters (F680B)

This module contained 32 6-bit TDC's and a 5-bit Vernier on the common start. The data were packed into seven 32-bit FASTBUS words. The module was designed and built at CERN.

A.4 ADC System (F682A)

This module in the DAQ Crate served only to connect the DAQ crate to the memory in the ADC Crate, and is described in ref 3.
A.5 Scaler (F680C)

This module contained four 16-bit fast scalers, packed into two 32-bit FASTBUS words. The module was designed and built at CERN.

A.6 Time History Module (F680C)

This module provided 48 channels of Logical State Analysis of various signal in the trigger system and on various discriminated levels in the detector, spanning a range of up to 255 steps about the trigger time. This module was designed and built at Pisa.
APPENDIX B

BUILDLIST PROGRAM

The input file for this program consisted of

1. One line specifying the mnemonic name for the list
2. a BEGIN statement
3. a sequence of FASTBUS instructions
4. an END statement

There were three groups of FASTBUS instructions; the structure of these was:-

1. 
   \texttt{<READ|WRITE> <WORD|BLOCK> <DATA|CSR> PA SA (<VALUE|LENGTH>}
   These instructions compiled into the corresponding sequences for single word or block transfers to or from data and CSR space.
   PA and SA are the primary and secondary addresses
   The VALUE was only required for WRITE WORD commands
   The LENGTH was only required for BLOCK transfer commands

2. 
   \texttt{INSERT <FCODE|MARKER|WC|MW> (VALUE1 (VALUE2}}
   where WC inserts a wordcount into the data stream, and MW inserts both a marker word and a word count
   This command entered specific instructions into the FASTBUS list.
   VALUE1 was not required for INSERT WC
   VALUE2 was only required for INSERT FCODE

3. 
   \texttt{<SET|CLEAR> CONTROL BIT}
   BIT is a bit-mnemonic, as defined in ref 13.
   This did not directly compile into the FASTBUS list, but controlled the setting of the option bits in the control array.

The values could be entered in decimal, Hexadecimal or Octal form.
Example

TESTLIST
BEGIN
SET CONTROL FOASUP
WRITE WORD CSR 0 0 FFFF0000H
SET CONTROL FONOPA
WRITE WORD CSR 0 0 80000000H
WRITE WORD CSR 0 0 00001080H
WRITE WORD CSR 0 0 80000000H
CLEAR CONTROL FOASUP
WRITE WORD CSR 0 16 00004021H
CLEAR CONTROL FONOPA
INSERT MARKER 0000FFFFFFH
END
**Listing of the list built from Example above**

***LISTDUMP*** of FASTBUS list - length 2048 Words (32-bit) options 10012

<table>
<thead>
<tr>
<th>Element</th>
<th>OPCODE</th>
<th>Data</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>101006F</td>
<td>0</td>
<td>Start List, List Identifier</td>
</tr>
<tr>
<td>2</td>
<td>80</td>
<td>1</td>
<td>Trigger/Execute Control</td>
</tr>
<tr>
<td>3</td>
<td>22</td>
<td>0</td>
<td>Embedded Primary CSR Space Address Write</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>0</td>
<td>SA Embedded Write</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>FFFF0000</td>
<td>Embedded Random Data Write</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>0</td>
<td>SA Embedded Write</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>80000000</td>
<td>Embedded Random Data Write</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>0</td>
<td>SA Embedded Write</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>1080</td>
<td>Embedded Random Data Write</td>
</tr>
<tr>
<td>10</td>
<td>4</td>
<td>0</td>
<td>SA Embedded Write</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>80000000</td>
<td>Embedded Random Data Write</td>
</tr>
<tr>
<td>12</td>
<td>4</td>
<td>10</td>
<td>SA Embedded Write</td>
</tr>
<tr>
<td>13</td>
<td>0</td>
<td>4021</td>
<td>Embedded Random Data Write</td>
</tr>
<tr>
<td>14</td>
<td>36</td>
<td>0</td>
<td>AS/AK disconnect</td>
</tr>
<tr>
<td>15</td>
<td>70</td>
<td>FFFF</td>
<td>Insert Embedded Marker into IPB</td>
</tr>
<tr>
<td>16</td>
<td>61</td>
<td>0</td>
<td>End List</td>
</tr>
</tbody>
</table>
References

CERN, Geneva, Switzerland

H Bluemer, G Dietrich, F Eisele, W Heiren, M Kasemann, K Kleinknecht, B Panzer, B Renk
University of Dortmund, Dortmund, Germany

D J Candlin, P McInnes, J Muir, K J Peach, B J Pijlgroms, I P J Shipsey, W Stephenson
University of Edinburgh, Edinburgh, UK

C Arnault, E Auge, R L Chase, M Corti, J-P Coulon, J-C Druilot, D Fournier, P Heusse, A-M Lutz, J-P Richer
Linear Accelerator Laboratory, Orsay, France

A Bigi, M Calvetti, R Carosi, R Casali, C Cerri, R Fantechi, S Galeoti, A Nappi, D Passuello, G Pierazzini
University of Pisa, Pisa, Italy

C Becker, D Heyland, M Holder, G Quast, M Rost, W Weihs, G Zech
University of Siegen, Siegen, Germany


[3] C Arnault et al., these proceedings.


[7] CAMAC FASTBUS Interface Camac Coupler (CFICC), B Struck (Hamburg), STR300.
[8] These were the University of Illinois designed Segment Interconnects.
[9] L Pregernig, The Block Mover (F6809), (CERN) to be submitted to the IEEE Nuclear Science Symposium, October 1985 (San Fransisco).
[10] M Rost, Switch Coupler Card (F6812), (Siegen)
[11] This was the University of Illinois Cable Segment Terminator.
[14] see A Lacourt, CERN, for details of FBSTUDY.
[15] see P Clarke, CERN, for details of BMTALK.
[16] See M Corti, Orsay, for details of the Micro-controller and Event Display.
[17] see C Youngman, CERN, for details of the histogramming and plotting system.
[18] Unibus-FASTBUS-Interface, B Struck (Hamburg). This uses the standard CFI FASTBUS Coupler, and interfaces to a custom-built personality card on a standard Digital DRE11 unibus interface.
Title: Software environment of Fastbus ADC's Crate for NA-31 experiment

Authors: Collaboration NA-31 — Presented by C. ARNAULT

Abstract:

The 1628 analog outputs coming from electromagnetic and hadronic calorimeters of the NA-31 experiment are fed into 18 Fastbus ADC modules (F6829) of 96 channels each. The ADC modules are read by the Fast Acquisition Sequencer (FAS — F6827) and data are stored during the same transaction in the FAS (spying purposes) and in the Fast-Buffering and Arithmetic Processor module (FBAP, F6828) either as raw data or corrected data. In a subsequent step, data are transferred to the main Fastbus crate through the Memory Segment Link module (MSL — F682A) during the main acquisition phase.

The Fastbus ADC Crate is under control of a PDP 11-23 PLUS through Camac interface of the FAS module. The whole software system has been developed at home laboratory and concerns tools needed by the FAS and other Fastbus modules, acquisition and monitoring tasks.
Software tools of the FAS

From a firmware point of view the FAS executes instruction over 52 bits. Successive instructions are stored in a 256 words fast access memory. Due to the firmware design, subfields of bits in an instruction have actions timed by an internal clock.

Two levels of software were developed: first an editor explicits the meaning of the instruction word subfields into keywords and thus enables to write/read easily any type of Fastbus transaction and to link transactions between them. Second a "super-editor" is used to build sets of instructions as macro-like code. After editing stage the micro-code is built with an absolute start address (both symbolic and absolute codes are saved on disk). Finally the editor includes tools to load/read programs specifying a starting address and to load/read interactively internal FAS 32 bits data memories (2x4 K - 32 bits mem) at any address.

Those tools are stored as Fortran library and can be used in "user-programs" for loading, reading, initialising the FAS and to read/write/verify access to memory banks. The size of micro-code memory of the FAS allows to load many independent programs which are selected only by pointing to their starting address. In such way, without any loading procedure, one can run the acquisition procedure, the automatic initialisation of the modules in the crate (based on ID), the loading of calibration parameters in the FBAP, the error recovery program ...

Acquisition software

An acquisition system for the PDP 11.23+ has been completely written that looks like a "micro-DAS". The single producer, the FAS, is read through the Camac interface either for spying events, or calibration events (electromagnetic or hadronic). Monitoring tasks are served as consumers by the system to get control of events or to perform a complete calibration to refresh the constants used by the FBAP module.

A Data Base contains all informations needed to configure the Fastbus module (CSR's, ID ...) and other CAMAC modules. The parameters needed by the acquisition task or other user tasks are also stored. The data base is accessible in interactive mode (read/write) and a Fortran library gives access to the informations of the database by any user (read only mode).
Since the main acquisition of the experiment is done with a VAX-750, a specific message and file transfer protocol between VAX and PDP has been developed through a RS-232 line. The main goal is to have complete command of the ADC Fastbus Crate from the VAX. The link is transparent to the user, and the PDP is under VAX control.

Main advantages

- The ADC Fast bus crate is an entity which can be either totally independent or absolutely slave. The independance is essential during the installation phase and for calibration purposes that cannot be achieved by the main computer due to overloading. The slave behaviour has to be used during physics data acquisition; the spying function of the FAS is then welcome.

- The DEC compatibility between computers helps to solve many problems (communication protocol) and all the tools developed work on both machines.

- The FAS micro-code tools are sets of complete Fastbus transactions. That preserves the speed (essential concept) and the usage of all FAS specific functionalities in a very light way.
FBMON :A FASTBUS MONITOR PROGRAM

Y. Bertsch

L.A.P.P., Annecy-le-Vieux, F - 74019 CEDEX

1. INTRODUCTION

When FASTBUS started in our lab, at the beginning of 83, systems were not reliable. The hardware (FDM and FIORI) was still uncertain as prototypes can be. The software was not fixed and, except CATY on PDP's, the CERN package in FORTRAN had never been really tested.

After some tests with CATY on our LSI11-23 equipped with a CC11 camac, people soon asked for help for the manipulation of the many subroutines, the hundred of F codes, of error codes, the decoding of the many registers ...etc.

Now it seems normal to get an FDM or a CFI which, when plugged into a crate, starts working perfectly. In 1983 we spent our first months between hardware and software for fixing, with CERN people and other users the tools which were necessary for our development.

The necessity for a general purpose test program appeared rapidly as a strong request from people who wished to work independently of software experts.

FBMON had to satisfy certain requirements:

- Immediate access for the uninitiated user.
- Access to all the features of the CERN FB package.
- An easy dialogue mode, with as many default values as possible.
- A flexible operating mode allowing immediate or non-immediate execution (access to FB lists).
- Easily interpreted FB status and results after execution.

After a one year of development a PDP version was debugged, and a VAX version followed a few months later. During Spring 84 a wider distribution of FBMON was impeded by the problems with availability of standard FORTRAN at CERN. It was decided to rewrite the program in full F77, also using MZCEDEX, the new portable dialogue/menu package.

Three months of collective work was necessary for the implementation of FBMON on the VAX, NORD and VALET, involving people working at CERN on FASTBUS. This version has been the standard version for a year. We have collected suggestions and a new version is being prepared for the end of 1985.

As a detailed manual is available from CERN manuals we'll just summarise the features of the program.

Users should remember that two working options are offered:

- Immediate execution mode: List processor is activated after list compiler. It is possible to modify the default sequence (arbitrate,primary address cycle,...) which is internally provided by the CERN package. After execution a precise error report is displayed with results of the operation.

- Delayed execution mode: For personalised tests, a particular sequence may be required. Options related to list manipulations allow a flexible handling of the current list, with the possibility of saving it on disc. At any moment a user's list can be recalled from disc and initiated by activating the current list processor.
2. MENU DESCRIPTION

2.1 Top and First Level Menus

FASTBUS TEST MENU
A : INITIALISATION
B : ARBITRATION ROUTINES
C : INDIVIDUAL SIGNAL HANDLING ROUTINES
D : SINGLE WORD TRANSFER ROUTINES
E : BLOCK TRANSFER ROUTINES
F : *
G : LIST MANAGEMENT
H : HELP INFORMATION
I : *
J : SYSTEM CONFIGURATION
K : ***
L : CHANGE EXECUTE MODE: 1 = IMMED 0 = LIST [ D 1 ]
M : LONG DIALOGUE MODE: 1 = SET 0 = RESET [ D 1 ]

ALSO AVAILABLE: 1 = TOP; 2 = BACK; 3 = LINE MODE

SELECT:

Options A, B, C, D and E exploit all the features of the CERN FB package except for the Segment Inter-connect Routines. Option L changes the execution mode.

2.2 Second Level Menu

2.2.1 OPTION A: Initialisation

INITIALISATION
A : FMNI: INITIALISE MASTER
B : FSINI: INITIALISE SLAVE
C : INTERFACE SWITCH: [ D 2 ]
D : CAMAC IGOR BRANCH: [ D 1 ]
E : CAMAC IGOR CRATE: [ D 2 ]
F : CAMAC IGOR STATION: [ D 10 ]

On entering this page, the CAMAC connection to the FASTBUS interface is set up and checked. For a VME interface branch, crate, station must equal to 1.

2.2.2 OPTION B: Arbitration Routines

ARBITRATION
A : SET ARBITRATION LEVEL: [ D 1 ]
B : SET TIME OUT VALUE: [ D 1 ]
C : FREQ: SET LEVEL AND ARBITRATE: [ D 1 ]
D : FREL: RELEASE BUS MASTERSHIP
E : SET TIME OUT AND WAIT: [ D 1 ]

Options A and B only change parameters (the arbitration level and the timeout) kept by the FASTBUS package. Options C, D and E actually perform bus operations.

2.2.3 OPTION C: Individual Signal Handling Routines

INDIVIDUAL-SIGNAL-HANDLING
A : READ INDIVIDUAL SIGNAL
B : WRITE INDIVIDUAL SIGNAL
C : DISPLAY OPTIONS

This option allows the manipulation of FB lines, but is currently only supported by the FIORI interface.
2.2.4 OPTION D: Single Word Transfer Routines

SINGLE-WORD-TRANSFER-ROUTINES
A: FRD: READ SINGLE WORD DATA SPACE
B: FRC: READ SINGLE WORD CNTRL SPACE
C: FRDSA: READ NTA DATA SPACE
D: FRCSA: READ NTA CNTRL SPACE
E: FRDM: BROADCAST READ DATA SPACE
F: FRCM: BROADCAST READ CNTRL SPACE
G: ***
H: FWD: WRITE SINGLE WORD DATA SPACE
I: FWC: WRITE SINGLE WORD CNTRL SPACE
J: FWDSA: WRITE NTA DATA SPACE
K: FWCSA: WRITE NTA CNTRL SPACE
L: FWDM: BROADCAST WRITE DATA SPACE
M: FWCM: BROADCAST WRITE CNTRL SPACE

2.2.5 OPTION E: Block Transfer Routines

BLOCK-TRANSFER-ROUTINES
A: FRDB: BLOCK READ DATA SPACE
B: FRCB: BLOCK READ CNTRL SPACE
C: FRDMB: BLOCK BROADCAST READ DATA SPACE
D: FRCMB: BLOCK BROADCAST CNTRL SPACE
E: ***
F: *****
G: ******
H: FWDB: BLOCK WRITE DATA SPACE
I: FWCB: BLOCK WRITE CNTRL SPACE
J: FWDMB: BLOCK BROADCAST WRITE DATA SPACE
K: FWCMB: BLOCK BROADCAST WRITE CNTRL SPACE

These options allow single word or block transfer data exchange between the master and the CSR or Data space of a slave. Broadcast operations are also available.

For these calls, a default sequence is provided to perform the normal data transfer (arbitration, address cycle, data cycle etc...). This standard sequence can be modified by setting appropriate bits in the hardware control word. For example, the following dialogue could occur:

```
HW-CONTROL WORD [ H 0 ]: 1=CLEAR ; 2=SAME ; 3=CHANGE
SELECT : [ D 2 ] 1
= = > PRIMARY ADDRESS [ H 0 ] 14
= = > SECONDARY ADDRESS [ H 0 ] H10
= = > DATA [ H 0 ] HABCDE

DO YOU AGREE?[Y]

LOOP CYCLES [ D 1 ]
```

The first question allows the control word to be cleared, modified or left unchanged. The current state of the control word is displayed. If you chose to change the options in this word, you will be asked such questions as:

SIGNAL DEASSERTED ON ERR......YES = 1 [ D 1 ] 0

54
2.2.6 OPTION G: List Management

LIST-MANAGEMENT
A: DISPLAY MODIFY PRESENT LIST
B: DISPLAY STATUS AND CONTROL
C: RESET LIST AND/OR STATUS
D: **************
E: READ FROM DISK LIST NAME
F: WRITE ON DISK CURRENT LIST
G: * ************** *
H: EXECUTE LIST

In list execution mode all the questions are the same as in immediate execution mode, the difference occurring at the moment of execution of the FASTBUS operations. In list mode, the request is compiled into a form suitable for FASTBUS, (a sequence of F-Codes), and stored in the control array. In this menu page there are some tools for dealing with a list.

INFO-HELP
A: TRICK AND PROBLEM
B: MENU DESCRIPTION
C: PRESENT SOFTWARE STATUS
D: ******
E: DISPLAY ERROR CODE
F: DISPLAY FCODE DESCRIPTION

Options A, B and C explain the program, and give useful information for beginners. Options E and F display the error descriptions used for reporting FASTBUS errors and describing F-Codes. It may be useful to refer to the proper document [1] for more information.

3. EXECUTION OF FASTBUS OPERATIONS

The number of execution cycles can be set. Its default value is always 1 in list execution mode, but can be changed in immediate execution mode by updating the value in SYSTEM CONFIGURATION. Following the execution the program produces an error report, an initial condition display and a results report.

1. 'error report': This is displayed for each FB-operation, including the list filling, which reports a software error, e.g. buffers full. A typical example of an error report follows:

```
LOOP CYCLES[D 1]
STATUS WORD [ H80008008 ]

BIT / ERROR DESCRIPTION

SS RESPONSE : 0 — VALID DATA
3 FAILED TO OBTAIN BUS MASTERSHIP
15 IF ALONE NON FASTBUS ERROR
31 ERROR DETECT, IF ALONE INCORRECT INTERFACE
```

The sw/hw status word and the decoded word are shown. The first figure is the ss response i.e. the coding of the SS0, SS1 and SS2 bits. Next is a list of other bits set in the word, with their meanings.

2. 'initial conditions': This is displayed if the FB-operation was a transfer cycle. An example:
INITIAL CONDITIONS

ARBITRATON AT LEVEL 1

CONTROL WORD [ H 0 ]
PRIMARY ADDRESS [ H E ]
SECONDARY ADDRESS [ H 10 ]

REDISPLAY? —Y/N— [N]

Any hardware options in use are displayed together with the sw/hw control word, and the addresses if not inhibited.

3. 'result report': This is displayed if the FB-operation requires a data result. The data are always displayed in hex notation. An example of a list’s execution:

DISPLAY THE DATA? [ Y ]
SECOND ADDR EMB WR [ H 0 ]
EMB RAND DATA WRITE [ H12345678 ]
SECOND ADDR EMB WR [ H 1 ]
WRITE WORDS BUFFERED : 10
1 : [ H 10 ] 2 : [ H 18 ] 3 : [ H 20 ] 4 : [ H 28 ]
5 : [ H 30 ] 6 : [ H 38 ] 7 : [ H 40 ] 8 : [ H 48 ]
9 : [ H 50 ] 10 : [ H 58 ]
SECOND ADDR EMB WR [ H 0 ]
READ WORDS BUFFERED : 11
1 : [ H876114 ] 2 : [ H 10 ] 3 : [ H 18 ] 4 : [ H 20 ]
5 : [ H 28 ] 6 : [ H 30 ] 7 : [ H 38 ] 8 : [ H 40 ]
9 : [ H 48 ] 10 : [ H 50 ] 11 : [ H 58 ]

N. LIST ELEMENTS : 20

The list itself is scanned, and all the data, both embedded and buffered, are displayed, as are the secondary addresses. For a block transfer, the number of words transferred is displayed, the number before the data being the position within the buffer.

4. MENU : I/O AND LINE MODE

4.1 I/O Rules

When there is a question to a user, a default value is always displayed. This value is used if just <CR> is typed. Integers are displayed in a number base most appropriate to the context: addresses and data are displayed in hex, word counts in decimal. Note that whatever the base used for display of the default value, the number typed in is considered to be:

1. decimal if there is no prefix or the prefix is "D".

2. hexadecimal if the prefix is "H".
4.2 Line Mode

As an alternative to menu mode, the program may work in "line mode". Line mode is reached by typing "3" in response to a menu. In line mode there is a prompt ">". A selection from any page of the menu system may now be typed in as a single command.

5. POSSIBLE IMPROVEMENTS

Users’ remarks and suggestions can be summarised as follows:

- Many options which are currently coded as 0 and 1 should be expressed as YES/NO questions.
- The menu organisation should allow modification or execution of a list immediately after building.
- Using VT100 screen facilities would be a solution to the slowness of MZCDEX, but would require a large amount of work. We have also to consider that ZCDEX is widely distributed and is an argument for using the compatible MZCDEX.

6. CONCLUSION AND ACKNOWLEDGEMENTS

FBMON is a program which has been developed for two purposes. Firstly it is a very useful tool for rapidly getting familiar with FASTBUS. Secondly, as it allows any FASTBUS transaction between a master and a slave, it is a basic software tool for development.

Maintaining more than 4000 lines of code to be accepted by many F77 different compilers is not straightforward, but is the price to pay for a large distribution.

I would like to thank those who contributed to the FBMON program, and particularly Paolo Baggia, Roberto Diviá and Doris Burekhart for help with the software, and Guy Perrot and Michel Moynot for comments on the hardware.
Fastbus Slave Diagnostic Software and System Verification

C.M. Story
Nuclear Physics Lab., Oxford University, UK.
currently at CERN, Geneva, Switzerland.

Geneva,

Abstract: Ideas for a Diagnostic and verification package for Fastbus segments and certain slave modules will be presented, together with a brief description of some existing software. In the existing program, the test(s) to be performed are selected by the user via a menu, and error producing cycles may be repeated as required. An extension of the software utilising a database of module characteristics will automatically test the elements of a Fastbus configuration, and may be used for stand-alone testing and as part of a standard startup procedure.

1. Introduction

The new generation of experiments at CERN involve a heterogeneous assembly of many detectors, each with a complex hardware configuration for data acquisition, which is capable of running either stand-alone or as a partition within the overall experiment. Many of the experiments plan to use Fastbus for all or part of their Acquisition System. The configurations range from small (c. 10 crates) to large (c. 100 crates), each containing up to 26 Fastbus modules ranging from simple slaves to complex master modules containing Motorola 68020 CPUs and co-processors.

During experimental running, it is important to be able to verify that the Fastbus system is fully operational, isolate potential problem areas, and perform increasingly extensive tests on individual modules or subsystems. In addition, during module development and hardware setting-up, extensive module and configuration testing is required. The diagnostics will be run either via an interactive interface to allow the investigation of any aspect of a module, system or full configuration, or automatically using a predefined procedure. This paper addresses the specific class of problems concerning FASTBus diagnostics which is one aspect of the overall monitoring of the performance of a readout system. It presents some ideas for a general package to test Fastbus slave modules, and the overall system configuration, and briefly describes some existing software.
2. Module Diagnostics and System Verification

There are three distinct phases during the building up and running of an experiment:

- during module development and Hardware setting up
- during final testing of the hardware partition
- during experimental running periods.

Each phase has differing requirements for Fastbus system diagnostics and verification which may be grouped into three different categories:

- Module tests.
  This consists of performing tests at various levels ranging from minimal (i.e. does a particular module respond to geographical addressing) to an exhaustive set of tests which will check all characteristics of a module; the full scope of tests must be available for each of the many different types of modules in a Fastbus system.

- Checking that the configuration/connections are as expected (i.e. as described in the Fastbus database). This includes:
  - checking whether Module in each slot is as described in the database
  - checking the connections of the interconnect modules
  - checking that the Software versions are correct

- Checking that the modules respond to other modules in the system as expected (for example, that the master responsible for handling SRs takes the correct action for an SR from every possible source.

The requirements described above lead to the concept of different diagnostic levels, defining a hierarchy of tests, which may be invoked with minimal effort (possibly by a non-expert user) The following categories are envisaged:

- Detailed (automatic or user-selected) tests for individual modules. These will be a mixture of standard test sequences generated when the module characteristics are entered in the database, and additional interactive tests defined at run time. The tests defined at run time will themselves fall into two categories; 'once off' tests for use only during the design debugging phase, and tests which will be required during later phases of the module's lifetime. It is envisaged that 'once off' tests will normally be performed using PILS programs or FBMON and that they will be module specific; therefore they will not be considered in detail here.
• Detailed automatic tests for a hardware partition, where a "partition" is any connected subset of the overall configuration. These will consist of the full standard tests for each module in the partition and the necessary tests to verify the connectivity of the subsystem, possibly carried out on a section of the hardware independently of the rest of the Fastbus system.

• Full verification of the system, its connections and each module. This will consist of the tests of each partition in the system, and any additional ones necessary to check the global connectivity. It is the most exhaustive test of the system and will be carried out when the system has been powered down or unused for some time.

• Fast (minimal) verification of the system or a partition. This will consist of a subset of the tests performed for the full verification, (e.g. connectivity and module identity checking), and will normally be carried out during experimental running, for example, after a particular crate has been switched off, or when a fault is suspected (because of incorrect monitoring data, unexpected error messages, etc.).

In addition, various diagnostic tools will be necessary; for example:

• enabling route tracing for a selected path through the system.
• using a recording module (e.g. an SFDM [6]) to record and playback sequences of Fastbus cycles for fault tracing and reproduction.
• Using a programmable Logic Analyser to trigger on selected signal-pattern sequences and compare the previous Fastbus cycles with a reference set.
• etc.

These tools, while not being part of the diagnostic package, may need to be interfaced into it in a well defined way.

Clearly within any given Fastbus configuration there will be many different types of modules. Each will have its own characteristics, and in addition, each will have a core of functions common to some or all module types. A small generalised diagnostic and verification package is being developed which will perform predefined sets of tests to check common features of different modules, and to allow the designer or user to develop additional test sequences at any time which may be incorporated into the general Diagnostic system. The package will answer the question "Is the configuration working", and if it is not, will attempt to isolate the actual fault, giving as many details as possible to the user. Using this, hardware faults (e.g. non-working features of modules), and software faults (for example incorrect route tables) may be identified by a non-expert.
There are several advantages to this approach of a provision of a general test package:

- Whilst requiring initial software effort, once the package is available, standard features of new modules may be tested immediately without the need to write and debug new software.

- During module and configuration development, when a fault occurs, it is the hardware which is being tested and not the diagnostic software being debugged.

- In the final setup, there will be a small amount of software to be maintained, with a consistent user interface and error handling.

A natural extension of this concept is to develop a formalism to describe the test sequences or test primitives, and to drive the software from a database. This allows module-specific tests, which may be automatically invoked at a later stage during some or all of the verification tests, to be added to the database as required.

There are two possible philosophies for the implementation of this:

- Putting all possible test sequences into high level code and generating test-masks either automatically or manually for the different verification levels. The test-masks may be stored in a database and used by a steering routine to control the flow of the verification software at run time.

- Using sequences of primitives to describe the tests. The primitives would be stored on the database and executed by a minimal piece of software; expected results would be stored in the database and compared against the results obtained. An example of such an implementation would be based on the F-codes generated by the CERN Fastbus list compiler, with some extensions (for example, for control) to define the test sequence primitives. The designer would use PILS or FBMON to debug the module, and to develop additional module specific tests.

In each case the information defining the tests for the standard module characteristics may be generated automatically when the module information is entered into the Fastbus database, leaving the module designer to add any relevant module-specific test sequences. The expected responses for each test must be obtainable from the database; either by using the module description information, or by using specific diagnostic response data.

The second possibility, while being more flexible in that new tests may be generated at any time (using PILS or FBMON), and existing test sequences may be modified (using FBMON), has certain disadvantages. For example, the amount of information stored in the database will be large (a test sequence primitive will typically correspond to one Fastbus cycle) and the lists of operations will be difficult to understand unless very well docu-
mented. Also, a large amount of information on the expected results of each action must be stored (for example, the results of block transfer write-read sequences) using some well defined 'language'.

These problems may be largely overcome by combining the two approaches; i.e. by retaining the high level code for the standard tests, and allowing additional module specific test sequences to be added into the database by means of lists of F-code primitives generated by PILS or FBMON. Sequences which are initially applicable to one module and are later used for several, may be converted to high level (hopefully self documenting!) code within the 'standard' software diagnostic package.

These ideas are preliminary, and will be investigated in more detail during the coming months.

3. Fastbus Configuration and Slave Test Program

A first attempt to put some of these ideas in practice is a program called FBSTEST (Fastbus Slave Test program), which is a high-level program, written in portable Fortran-77, to test standard features of Fastbus slave modules and the overall connectivity of the system. It evolved from a specification [2] by D. Burckhart/DD for a program to test the FDM (Fastbus Diagnostic Module), and using the FDM, to test the crate itself. It is complementary to the other existing diagnostic tools at CERN (PILS and FBMON) in that it uses predefined sequences of Fastbus operations, grouped together to form tests which may be invoked individually or in groups. The program attempts to answer the question "is this Fastbus configuration working?" rather than providing detailed hardware debug facilities for the user to invoke at will, although for convenience, the single line handling routines are available, and there is a mode which allows the user to perform single Fastbus cycles in any order, any number of times.

The tests are grouped into families; each family tests one aspect of a module (e.g. the SS responses, data space memory tests, etc.). The program is menu driven, and execution is governed by a steering routine which sets up test-masks and calls the relevant lower level modules. It is envisaged that, in future, lists of required tests for various applications and diagnostic levels will be stored in the database, thus defining the standard test sequences for various levels of diagnostics. A steering routine will convert this information to appropriate test-masks and invoke the lower level software modules in FBSTEST.

Some of the current test facilities available are:

- execute exhaustive tests on a Fastbus slave module to verify that all appropriate standard Fastbus functions are working correctly. This includes memory testing on all or a subset of available memo-
ry, and checking of SS responses by using illegal Fastbus cycle sequences).

- perform all tests quickly (all appropriate tests are performed, but, for example, the time-consuming memory tests are curtailed).

- perform a subset of tests within one or more families (this is used when a fault is suspected, to test one or more of the module characteristics).

- check the connectivity of the system by addressing any module in the system.

- perform single Fastbus cycles in any order.

- using the FIORI interface, execute the Fastbus single line handling routines to read or modify lines as required.

- different modules may be selected, and the characteristics modified by menu; this gives the facility to perform appropriate tests on a module not yet known to the program. (At the moment, the characteristics are loaded via a routine, but in the future this will be replaced by calls to the Fastbus database. However the user will still be allowed to define a 'special' module, to allow tests on modules not yet entered in the database).

- A stand-alone set of routines to interpret the Fastbus errors and either to display the messages as the errors occur, output the errors to disk, or to log them and display a summary at program termination, has been produced and used for the error logging. Program detected errors (for example, when data read is not the same as the data written) are also handled by the package. FBSTEST appends details of the test being performed, and the Fastbus cycle causing the error, to the error details output by the stand alone error routines.

- The program will repeat each part of a test n times, where n is user defined (by default n is set to one). When an error occurs, the program will repeat the cycle n times and then ask the user what action should be taken. The options are:
  - abort the complete set of tests
  - continue with the next stage of the test
  - repeat a user specified number of times

  This allows an individual cycle to be repeated many times while a fault is investigated.

In addition, the user may control the way in which the program runs:

- various program parameters, for example the default number of times to repeat each Fastbus cycle, may be changed.
3.1 Future Developments

A stable version of FBSTEST is available; however, various additions are planned, some of which are mentioned below:

- A short term development will be the provision of SI initialisation, and the ability to define a subset of the configuration to be tested. A steering routine will then loop over the specified hardware interrogating the database and performing appropriate tests according to the module types found and the level of diagnostics required.

- An additional steering routine to test automatically the connectivity of the initialised system.

- A facility to test more module specific characteristics without writing large quantities of module-specific code. This will initially be implemented by creating F-code lists using a PILS program or FBMON, storing the generated list in the database, and then executing it as part of the standard test procedure for the module.

- Normally FBSTEST will only output error messages (unless special Debug printout is switched on). For known faults on a particular module, it may be useful to store the resulting messages on the database and automatically compare the current results 'reference output', and unnecessary error messages suppressed. This is clearly only useful for 'benign' known faults (i.e. those occurring when exercising features of a module which are not used during readout).

Longer developments include the upgrading of the package to perform full configuration and connectivity tests, the inclusion of tests for Fastbus masters and the provision of some protection against interference in a multi-user environment.

3.2 Implementation

FBSTEST is written in the subset of Fortran-77 available for a PDP11. It uses the CERN Fastbus library to access the hardware, and is available on all machines for which the Fastbus software is available (currently VAX, NORD, M68k, PDP11).

An official release of the current version is planned for the end of September.
References


[6] The SFDM, the Snoop FDM; FDM upgrade specification H. Muller, D. Burckhart, April 1985


[8] Information in a Fastbus Database. T. Kozlowski, C.M. Story, July 1985


[14] FBSTTEST, a general Fastbus slave Test Program. C.M. Story, August 1985
A SYSTEM FOR THE TEST OF SLAVE FUNCTIONS
IN FASTBUS DEVICES

P. Werner
EP Division, CERN, Geneva, Switzerland

Geneva, 25 September 1985

Abstract: The multitude of FASTBUS Devices to be designed and produced will require a considerable quantity of test software that will be used frequently on stand alone test systems by industry and laboratories. These Devices will vary widely in user functionality but the FASTBUS properties will be a subset of a common domain.
This paper proposes a way to describe FASTBUS Slave Devices so that general test programs can test a Device by interpreting its FSDD, 'FASTBUS Slave Device Descriptor'. This Descriptor is defined interactively, by specifying information about implemented address and data cycles as well as word range, bit range and type for implemented registers/memory in CSR and DATA space.

1. INTRODUCTION

Electronic Devices are usually designed, prototyped and produced in a small series at CERN. If many units are required, production and testing are normally done by a contracted firm. The procedure consists of:

1. Device design
2. building prototype(s)
3. writing test software
4. debugging/improving the Device and the test software
5. series production and testing
6. testing and verification of potentially faulty Devices in experiments and workshops.

The FASTBUS specification makes it possible to have FASTBUS relevant information for a Device described in a standard fashion.
This 'FASTBUS Slave Device Descriptor', FSDD, could then be, among other things, interpreted by general test programs to test FASTBUS functionality of a Device.
This would mean that the Device specific software, if any, only has to test the non-FASTBUS related functions of the Device, with the Device's address— and data— cycle responses, data paths and memory/registers already tested, completely or to a large extent.
2. IMPLEMENTATION

The FASTBUS test environment for EP Electronics is a DD supported VME based system, the VALET PLUS [1]. Programming languages are primarily PILS [2], which is both interactive and compiled, and FORTRAN-77 with CERN extensions. As file handling is not yet supported in FORTRAN-77 on the VALET PLUS, the FSDD software is first implemented in PILS for later conversion to FORTRAN-77 when necessary. To ensure description portability between small systems a simple ASCII file is used to store the FSDD information on disk.

FSDD is implemented as a set of functions and subroutines which can be divided into four parts:

1. INITIALIZER module, that takes a Device name, e.g. F6808, and initializes internal memory buffers from the FSDD file.

2. ACCESS module, the routines the test program uses to get Device characteristics.

3. DEFINITION module, that is used to define/look at/modify a FSDD.

4. TEST module, by which a user interactively can use the ACCESS routines on a FSDD.

Figure 1 illustrates how FSDD modules and FSDD application programs access FASTBUS Device characteristics.

Figure 1: FSDD overview
2.1 INITIALIZER module

The INITIALIZER initializes the necessary memory buffers from the FSDD disk file and verifies the structural consistency of the description. The INITIALIZER must be called before any other FSDD routines are called.

2.2 ACCESS module

To use the ACCESS module the internal data structure must be set up by the INITIALIZER. The ACCESS routines are all implemented as functions where the function value is a STATUS indicating the result of the function. These functions are implemented in such a way that the user of them accesses the Device characteristics in way similar to FILE access, i.e. either randomly or sequentially. To access information about the FASTBUS Device in a "random" way the "GET..." functions should be used while "NXT..." functions should be used for "sequential" access. The "RST..." functions are used to reset the "sequential" access to the start of the "..." context for the Device description, e.g., RSTAC resets the sequential access to the beginning of defined address cycle definitions while RSTDC resets the sequential access of data cycle definitions WITHIN the CURRENT address cycle.

The implemented functions are described in appendix A.

2.3 DEFINITION module

This module uses the INITIALIZE and ACCESS modules. The DEFINITION module is used to create a new, modify an existing, or simply look at the definitions in a FSDD.

The user interacts with the routines through a tree-like traverse of menus where current information is displayed where relevant. Appendix B contains an example of a menu that is shown when DISPLAY of memory/register information previously has been selected. Except for the "action" choice the CREATE, MODIFY, DELETE and DISPLAY menus are the same. Other menus have a similar structure.

The definition is divided into two parts, the memory/register definitions and the address/data cycle definitions. The address/data cycle definition contains information about responses for each possible FASTBUS address mode (logic address to data space, general broadcast to CSR space, etc.) and for each address mode the related data cycle responses. Figure 2 shows the internal organisation of the address/data cycle information.

One memory/register definition defines a block of one or more consecutive registers in CSR or data space by specifying information like:

- FASTBUS address range and space.
- which bits are of what memory type (ram, rom, set etc.).
- which data cycles are allowed for single Device connect and what SS responses can they give.
- which data cycles are allowed for broadcast connect and what SS responses can they give.
- optionally, "expansion data" can be defined for the block.

Figure 3 shows the internal organisation of the memory/register information where the memory/register definitions are linked together forming the memory/register specification for the FASTBUS Device. A program to define a FSDD needs only to contain the line

CALL DEFINE
Figure 2: Internal Organisation Of The Address/Data Cycle Information

Figure 3: Internal Organisation Of The Memory/Register Information
2.4 TEST module

This module uses the INITIALIZE and ACCESS modules. The TEST module gives the possibility for interactive execution of ACCESS routines on a FSDD.

The user interacts with the routines through menus where current information is displayed where relevant. Appendix C contains some example menus that show how the user interacts with the access TEST module. A program to test a FSDD needs only to contain the line

CALL TEST

3. APPLICATIONS of FSDD

Once a FASTBUS Device is defined as an FSDD, more use of it can be made than just for test programs. Two examples of this are a DOCUMENTATION GENERATOR program and a FASTBUS consistency CHECK program.

The DOCUMENTATION GENERATOR accesses information about the FASTBUS Device through the ACCESS module and generates a text file for direct printing or in a format suitable as input to a word processor.

A CHECK program checks a FSDD for consistency with the FASTBUS specifications, or with local standards.

Some mandatory features that could be checked are

1. Geographic addressing to CSR space
2. secondary addressing to CSR0.
3. CSR0 with rom in upper 16 bits and data read cycle allowed.
4. a valid FASTBUS Identifier defined for the Device.

as it is mandatory that the FASTBUS Identifier can be read.

Similarly, if CSR 3 exists:

1. CSR3 memory type is RAM for bits 31–IA, where IA is the Internal Address field length
2. logical addressing is defined for either CSR or DATA space.
3. CSR 0 <1> is implemented and is, set by writing a one into the bit, cleared by writing a one to bit CSR0 <17>.
4. etc ...

If CSR3 is not implemented corresponding "negative" tests could be performed.

These and similar CHECKs would, among other things, improve the consistency of modules being designed by different persons at different locations.
4. STATUS and PLANS

A preliminary PILS version has implemented:

1. the INITIALIZE and DEFINE modules
2. the ACCESS module for the memory/registers and partly for the address-/data- cycles.
3. Access TEST module for memory/register routines.

The current source code is 2000 lines and the complete version is expected to be 3000 lines with about 500 lines for INITIALIZE and ACCESS modules.

A complete PILS version of the INITIALIZE, ACCESS, DEFINE and access TEST modules is expected to be ready in November 1985. A FORTRAN-77 version of the INITIALIZE and ACCESS modules for test program use could be ready at the same time if requested.

As the FSDD internal data structure links definition objects within a simple vector of 32 bit words, it is feasible to add other Device information if necessary, either as a sub or super structure. One possible extension is to describe FASTBUS master characteristics.

5. CONCLUSION

The FSDD offers several advantages for FASTBUS Slave Device design and testing.

- The designer can interactively describe the Device being developed.
- Test programs are made available as the Device is specified.
- The Device can be verified for consistency with the FASTBUS specifications.
- Standardized documentation for the Device would be available.
- The FSDD program modules and the description files are portable by using PILS/FORTRAN-77 program languages and simple ASCII files for disk storage.

This concept of describing bus devices is valid for any well specified bus system.

6. Acknowledgements

Acknowledgements are due to P.Ponting for the original suggestion and active encouragement, C.Story for fruitful discussions and F.Formenti for acting as a guinea pig to demonstrate the effectiveness of the approach.

References

[1], VALET PLUS overview, DD–OC GA/NOTE17, CERN, Geneva, Switzerland
[2], PILS, A Portable Interactive Language System,
     R.Russell, D.Williams, CERN, Geneva, Switzerland
The following ACCESS routines are defined

GETAC (requested_AC, SS_word) is requested address cycle implemented

NXTAC (AC, SS_word) get next address cycle implemented

RSTAC reset sequential AC access

GETDC (any/current AC, requested_DC, SS_word) is requested data cycle implemented

NXTDC (DC, SS_word) get next data cycle for current connect

RSTDC reset sequential DC access

for information about Device address cycles and related data cycles.

GETBLK (flag, space, FBA(2), #memply, sd_type, bc_type) is requested memory/register block implemented

NXTBLK (space, FBA(2), #memply, sd_type, bc_type) get next memory/register block implemented

RSTBLK reset sequential memory/register access

GETMTP (memply, mask, flag) is requested memory type implemented in current block

NXTMTP (memply, mask, flag) get next memory type implemented in current block

RSTMTP reset sequential type access in current block

EXPDAT (fba, data) get predefined data for FB address, fba

EXPBLK (fba(dim), data(dim)) get predefined data for the address range fba(1) < − > fba(2)

GTCMSK (bitno, mask) get bit mask for the bits cleared by writing a one into the clear bit, bitno.

GTSMSK (bitno, mask) get bit mask for the bits set by writing a one into the set bit, bitno.

MESSAG (message_number) convert message number into a message string

for information about the Device’s memory/registers, their types, accessibility etc.
APPENDIX B
EXAMPLE OF A DEFINITION MENU

(user responses are inside double quotes, "")

*** DISPLAY MEMORY/REGISTER BLOCK ***

- 1: ABORT and leave DISPLAY
  1: DISPLAY next definition
  2: SPACE [DATA]
  3: ADDRESSES [0 - FFH]
  4: MEMORY TYPES [1]
  5: TRANSFER TYPES (single device) [6]
  6: SS - responses (single device) [3]
  7: TRANSFER TYPES (broadcast ) [2]
  8: SS - responses (broadcast ) [1]
  9: MEMORY EXPANSION [NO]

choose? "4"
The following MEMORY TYPES are DEFINED for this definition:
  RAM with hex bit mask FF

choose? "5"
transfer types for SINGLE DEVICE connect
  single word write
  single word read
  handshaked BT write
  handshaked BT read
  pipelined BT write
  pipelined BT read

choose? "6"
SS responses for SINGLE DEVICE connect
  SS0
  SS2
  SS7

choose? "7"
transfer types for BROADCAST connect
  single word write
  handshaked BT write

choose? "8"
SS responses for BROADCAST connect
  SS0

choose? "- 1"
return to the top level menu
APPENDIX C

TEST MENU SAMPLES

After initializing a FSDD an example sequence can be: (user responses are inside double quotes, "")

- 1 : leave FSDD_TESTS
  1 : list definition raw data
  2 : FIND memory/register (getblk)
  3 : NEXT memory/register (nxtblk)

choose? "2"
CSR or DATA space? "CSR"
give blocks low FB address[0] "< CR >"
give blocks high FB address[0] "< CR >"
give flag (0 - 3)? 0
2 memory types with transfer control word for
SINGLE DEVICE = 410003, BROADCAST = 410001
in CSR space hex address range 0 - 0

- 1 : leave FSDD_TESTS
  1 : list definition raw data
  2 : FIND memory/register (getblk)
  3 : NEXT memory/register (nxtblk)
  4 : FIND memory/register type in current block(gettmp)
  5 : NEXT memory/register type in current block(nxttmp)

choose? "3"
1 memory types with transfer control word for
SINGLE DEVICE = 8500F3, BROADCAST = 10011
in DATA space hex address range 0 - FF

- 1 : leave FSDD_TESTS
  1 : list definition raw data
  2 : FIND memory/register (getblk)
  3 : NEXT memory/register (nxtblk)
  4 : FIND memory/register type in current block(gettmp)
  5 : NEXT memory/register type in current block(nxttmp)

choose? "5"
bit pattern FFH valid for RAM

choose? "5"
end of types in block
Software for a FASTBUS Segment Manager/Interface Module

B. T. Yost, J. Anderson, J. Bennett, R. Roush
LeCroy Corporation, Spring Valley, NY 10977, U.S.A.

ABSTRACT

An implementation of the FASTBUS standard subroutines in software and firmware for the LeCroy Model 1821 Segment Manager/Interface Module is described. These routines permit complete management of a standard FASTBUS Segment at full FASTBUS speed and support arbitration within the Segment. Data acquisition and transfer rates are presented.

Support for interfaces to CAMAC, ECL links to Event Buffers, and the DEC PDP-11/VAX UNIBUS is discussed. Implications for present and planned data acquisition architectures using multiple Segments are reviewed. A special hardware readout mode for LeCroy high density TDC and ADC data acquisition modules is also discussed.

Introduction

The Model 1821 FASTBUS Segment Manager/Interface (SM/I) is a programmable FASTBUS Master. Its original design intent was to provide a readout module and tester for the LeCroy FASTBUS 1800 Series of data acquisition modules. As more FASTBUS experience was gained, the 1821 SM/I's programmability provided users with a great deal of flexibility in designing and implementing FASTBUS data acquisition systems. The intent of this paper is to describe the current status of the software and also to describe various architectures that have already been implemented (Figures 1 and 2).
Figure 1
FASTBUS Architectures Implemented at LeCroy
Figure 2

FASTBUS Architectures Implemented by Experiment 653 at Fermilab
1821 SM/I Hardware

The 1821 SM/I has been described as a previous paper\(^1\). However, a short description may provide insight into the software implementation. The 1821 SM/I is a double width FASTBUS module consisting of two boards, the 1821-1 and the 1821-2.

The 1821-1 provides the FASTBUS interface and control. It consists of a high speed ECL sequencer capable of fetching and executing over 30 million instructions per second. The sequencer micro-instruction word is 64 bits wide and its instruction memory is 256 words deep. Each word is divided into 8 nearly independent fields, each capable of a micro-operation. Because of its high speed and the width of the instruction word, the sequencer can execute over 100 million operations per second in certain applications. Figure 3 shows on the following page a block diagram of the sequencer.

The second board, the 1821-2, provides the HOST interface system. It consists of 8 I/O registers, sequencer program memories (PROM and RAM), a 4K X 32-bit data memory, a 8K 10-bit pedestal memory, and the pedestal subtraction hardware. It allows the user to bypass the on-board memory and route data directly to the HOST, or indirectly to the HOST via DMA transfer from the on-board data memory.

1821 Instruction Word

The sequencer instruction word is 64 bits wide. It is divided into 8 fields, 7 that are currently used, and the 8th field which is reserved for future hardware upgrades. Table 1 describes the fields.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OP-CODE</td>
<td>Defines the instruction to be executed. There are 11 instructions currently defined.</td>
</tr>
<tr>
<td>CONDITION CODE</td>
<td>Defines the Condition Code to be tested.</td>
</tr>
<tr>
<td>MULTIPLEXER</td>
<td></td>
</tr>
<tr>
<td>BUS DEFINITION</td>
<td>Defines HSDATA and IAD Bus sources.</td>
</tr>
<tr>
<td>HSDATA</td>
<td>Is an 8-bit data field that can be loaded onto the HSDATA Bus.</td>
</tr>
<tr>
<td>STROBES</td>
<td>Defines the strobes that latch or set different conditions within the Sequencer.</td>
</tr>
<tr>
<td>DATA CONTROL</td>
<td>Defines the mode of the 32-bit Register (either BYTE or WORD), whether data is piped to other subsystems.</td>
</tr>
<tr>
<td>FASTBUS PROTOCOL</td>
<td>Defines the FASTBUS lines to be SET/RESET, and the mode (SLAVE or MASTER).</td>
</tr>
</tbody>
</table>

Table 1
Figure 3
Block Diagram of the Sequencer on the 1821-1 Board
Because of the bit slice design of the sequencer, operations within each field are executed simultaneously. The sequencer instruction set consists of 11 instructions. Although small, it does provide the user with the ability to execute all standard and non-standard FASTBUS protocol. Of the 11 instructions, only 6 have been used in the firmware that will be described in this paper. Table 2 describes the instructions and their functions.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>START (STR)</td>
<td>Fetch address on INITIAL WORD ADDRESS lines (IWA).</td>
</tr>
<tr>
<td>RETURN (RET)</td>
<td>Fetch address in RETURN ADDRESS REGISTER (RAR).</td>
</tr>
<tr>
<td>NEXT (NXT)</td>
<td>Fetch address in NEXT SEQUENTIAL ADDRESS REGISTER (NSAR).</td>
</tr>
<tr>
<td>JUMP (JMP)</td>
<td>Fetch address on HSDATA Bus.</td>
</tr>
<tr>
<td>CJMP</td>
<td>Fetch address on HSDATA Bus if CC bit is TRUE, otherwise fetch address in NSAR.</td>
</tr>
<tr>
<td>CALL</td>
<td>Fetch address on HSDATA Bus and latch NSAR address into RAR.</td>
</tr>
<tr>
<td>CCAL</td>
<td>Fetch address on HSDATA BUS if CC bit is TRUE, otherwise fetch address in NSAR.</td>
</tr>
</tbody>
</table>

**NOTE:** The following OP-CODES are of limited use.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>NCAL</td>
<td>Fetch address in NSAR, and latch it into RAR.</td>
</tr>
<tr>
<td>NRET</td>
<td>Fetch address in RAR, and latch NSAR into RAR.</td>
</tr>
<tr>
<td>LSTR</td>
<td>Fetch IWA address, and latch NSAR into RAR.</td>
</tr>
<tr>
<td>CRET</td>
<td>Fetch address in RAR if CC bit is TRUE, otherwise fetch IWA address.</td>
</tr>
</tbody>
</table>

A useful feature of the micro-instruction word is the ability to dynamically test the state of over 170 unique conditions, including all FASTBUS Master, Slave and bus management signals. Other conditions which may be tested include internal timers, Host interface lines, Personality Card signals, and 8 user defined conditions. The appropriate test condition must be selected with the condition code multiplexer on the preceding micro-instruction. The user may then use the CJMP micro-instruction to branch to the address specified on the HSDATA bus if the Condition Code (CC) is true.

The High Speed Data bus (HSDATA) can be driven by internal 1821 registers, the instruction word, or FASTBUS depending on the state of the bus definition field. Similarly the Internal Address bus (IAD) can be driven by internal 1821 registers or FASTBUS. When the HSDATA bus is driven by instruction word, the data is derived from the HSDATA field immediately following the bus definition field.

The 8-bit strobe field allows the user to control the function of the TCNT/TPOLL register, reset internal timers, load the PDREG or load the 32-bit register. The data control field
allows selection of the operational modes of the 32-bit register (either byte or word). It also provides control of the sequencer pipeline.

The FASTBUS PROTOCOL field allows the user to set or clear FASTBUS signals. Different signals are set or cleared depending on the mode (MASTER or SLAVE).

High Level Language Implementation

A group of primitives has been written in micro-code that allow the user to perform all but the most complicated operations. Table 3 lists all the primitives that have been implemented. These micro-codes routines conform to the FASTBUS software specification. All of the routines can be executed via the HOST, while some can be called from within a micro-coded program executed via the HOST.

Table 3

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPAC/FPAD</td>
<td>Primary address cycle - control/data space</td>
</tr>
<tr>
<td>FPACM/FPADM</td>
<td>Broadcast address cycle - control/data space</td>
</tr>
<tr>
<td>FPWNTA/FPRNTA</td>
<td>Secondary address cycle - write/read</td>
</tr>
<tr>
<td>FPRW/FPR</td>
<td>Data cycle - write/read</td>
</tr>
<tr>
<td>FBABE</td>
<td>Set arbitration level</td>
</tr>
<tr>
<td>FBBREW</td>
<td>Initiate arbitration cycle for bus mastership</td>
</tr>
<tr>
<td>FPREFL</td>
<td>Release bus mastership</td>
</tr>
<tr>
<td>FPRL</td>
<td>Pulse RB signal line</td>
</tr>
<tr>
<td>FPRL</td>
<td>Read a block of data from an addressed slave</td>
</tr>
</tbody>
</table>

Table 4 lists a group of micro-code routines that have been specifically implemented for LeCroy data acquisition modules, but are generally applicable to any slaves.

Table 4

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPSCAN</td>
<td>Initiate a T-Pin scan of the segment</td>
</tr>
<tr>
<td>TPINRD</td>
<td>Initiate a broadcast (type specified by the HOST), scans the Segment, and readout all slaves that have asserted their T-Pin.</td>
</tr>
<tr>
<td>HDRBL</td>
<td>Uses a special hardware feature of the 1821 to readout a slave at a higher speed than the normal FASTBUS block transfer - FPRBL</td>
</tr>
<tr>
<td>READTDC</td>
<td>Read only a specified channel of a TDC until SS=2.</td>
</tr>
</tbody>
</table>

The micro-coded routines described in the previous tables are currently called by FORTRAN routines with the same name. The FORTRAN routines are responsible for downloading parameters to or unloading parameters from the 1821. The procedure for doing this is simple and straightforward. The source of the PDATA bus must be selected by writing the appropriate data in Register 0. If data is to be passed to the sequencer, then the appropriate data must be loaded in Register 2. Once this has been accomplished, the micro-code must be executed. Figure 4 shows the procedure for executing the micro-code.
As one can see, in order to execute the different micro-coded routines, the user need only modify the value written to Register 1, the PADDR bus.

The standard PROM 0 Version 6 Update 3 has been implemented with a transfer address table. This allows modifications within the micro-coded routines without any modifications to the user source code. Preprogrammed timeouts along with a subset of the standard FASTBUS error codes have also been implemented. In most cases, it is expected that PROM 0 will provide the user with all the firmware necessary to communicate with a FASTBUS Segment.

For the more experienced user, a cross assembler is available. Figure 5 shows a subroutine written in the source code for the cross assembler. The cross assembler is written in FORTRAN and adaptable to almost any HOST. The implementation referenced throughout this paper is for a DEC PDP-11/34A. A definition file, source listing of the standard PROM 0, flowcharts and runtime documentation will be provided to any interested user.
ARBIT; Arbitrate for FASTBUS Mastership

;This routine will cause the 1821 to raise its AGK line. This will cause the hardware to
;arbitrate for mastership of the FASTBUS. The routine will wait for a GK response
;signifying a successful request. If no GK is returned from the arbitration hardware the
;route will time out and return to the idle loop with an error code=20 decimal in
;the PDREG.

ARBIT: NOP D #20 & CLKPD & SETAGK
      NOP 0 & TIMER & IFT IGK
      CJMP ARWON & IFF T4098
      CJMP ARLOOP & IFT IGK
ARLOOP: JMP IDLE & CLRAGK
ARL0ST: JMP IDLE & CLRAGK
ARW0N: JMP IDLE & CLKPD & CLRAGK

;Initialize PDREG and start
;arbitration
;Reset timer, test; GK=1?
;if GK=1, won arbitration.
;Also, test: timeout?
;if no timeout, loop back
;and test; GK=1?
;Timed out, clear AGK and go
;back to idle loop
;Won arbitration, clear PDREG
;and go back to idle loop

Figure 5
Source Code for META29M Cross Assembler

The 1821 SM/I has been interfaced to CAMAC via the LeCroy
Model 2891 and to the DEC UNIBUS™ via the DR11W and
the 1821/DEC. DMA transfers of up to 4 Megabytes/sec have
been measured when the DR11W has been configured for
N-cycle burst mode. The software driver for the DR11W is
available upon request. DR11W compatible interfaces are also
available commercially from other manufacturers

Interfacing to bus structures and other LeCroy modules is
accomplished by specific Personality Cards. Table 5 shows the
LeCroy Personality Cards that are currently available or will
be available in the near future.

Table 5

<table>
<thead>
<tr>
<th>Personality Card</th>
<th>Bus/Module</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1821/DEC</td>
<td>UNIBUS™</td>
<td>Available</td>
</tr>
<tr>
<td>1821/ECL</td>
<td>LeCroy 1821 Memory</td>
<td>Available</td>
</tr>
<tr>
<td>1821/QBUS</td>
<td>QBUS™</td>
<td>Near Future</td>
</tr>
<tr>
<td>1821/VME</td>
<td>VME/VMX</td>
<td>Near Future</td>
</tr>
</tbody>
</table>

These Personality Cards have been used to implement the
architectures shown in Figure 1. A FASTBUS to VME
Interface for the Model 1821 SM/I has been designed and built
by Harvard for UA1. It will be presented at this conference.
For debugging user written micro-code, the 1821/MON is also available. The 1821/MON Monitor Accessory allows the user of the 1821 SM/I to monitor execution of 1821 micro-code with single-step and halt-on-address capability. The monitor also allows the user to change the clock frequency of the 1821, and displays the address and state of the HSDATA Bus when the 1821 is halted. The simple control panel and small size of the 1821/MON make it an invaluable aid to programming and interfacing the 1821 SM/I. This provides the user with the ability to:

1. Display the micro-code
2. Single step micro-coded routines
3. Stop execution at a predefined address

**Current Status - FASTBUS Data Rates and Architectures**

As can be seen from Figure 1, many different architectures have been implemented and tested. Table 6 displays some measured Master/Slave data rates obtained with LeCroy FASTBUS modules.

<table>
<thead>
<tr>
<th>Master</th>
<th>Slave</th>
<th>Rate</th>
<th>Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>1821</td>
<td>96-Channel TDC</td>
<td>9 MHz</td>
<td>Block</td>
</tr>
<tr>
<td>1821</td>
<td>96-Channel TDC</td>
<td>5 MHz</td>
<td>Random</td>
</tr>
<tr>
<td>1821</td>
<td>96-Channel ADC</td>
<td>7 MHz</td>
<td>Random</td>
</tr>
<tr>
<td>1821</td>
<td>1 Megabyte Memory</td>
<td>1.3 MHz</td>
<td>Block</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(5.2 MHz)*</td>
<td></td>
</tr>
</tbody>
</table>

*Predicted data rate using Texas Instruments ALS232 FIFO no: yet available.

**NOTE:** The preceding rates do not include the digitization times or the time required to establish the AS-AK lock.

Of the FASTBUS architectures that have been implemented, the architectures depicted by Figures 1A and 1C are used routinely at LeCroy for the production testing of the 1800 Series FASTBUS modules. The architecture depicted in Figure 1A employs the standard PROM 0, while the architecture depicted in 1C employs firmware that is still under development. This firmware allows the Master in the primary Segment to communicate via the Slave (LeCroy 1892 Memory) with a maximum of 16 secondary Segments.

The architectures depicted in Figures 1B and 1D have only been used to demonstrate that the architectures are feasible. No serious measurements have been made. DMA's were tested from each Segment shown in Figure 1B and bus arbitration between different Hosts was tested as shown in Figure 1D. Both architectures employed PROM 0.
Figure 2 depicts the FASTBUS architecture that has been employed by Experiment 653 at Fermi National Accelerator Laboratory\(^6\). The system consisted of 11 FASTBUS Segments, 1 primary Segment containing 11 LeCroy 1891\(^6\) FASTBUS memories, and 10 secondary Segments each containing a LeCroy 1821 SM/I, a LeCroy 1810 CAT, and varying numbers of LeCroy 1885 ADC's and 1879 TDC's. The experiment took data this summer. The results are shown on the figure.

The experimenters used the standard PROM 0 as a basis for implementing their own micro-code. They were also responsible for the first implementation of the Microtec cross assembler.

**Conclusion**

The 1821 SM/I is possibly the most versatile and highest speed Host interface available commercially today. The programmability of the 1821 SM/I allows the user to communicate with any Slave using FASTBUS standard software, with the ability to support non-standard protocol. The variety of commercially available Personality Cards provides the user with direct connection to CAMAC, UNIBUS, VME, or any computer that supports a parallel I/O interface. Thus, the 1821 SM/I bridges the gap between today's high speed data acquisition modules and existing control and acquisition systems.

**Acknowledgements**

The authors would like to acknowledge many fruitful discussions with our colleagues at LeCroy, in particular, George Bleier, Werner Farr and Peter Martin. We would also like to thank Judie Johnston for her help in preparing this manuscript.
References


3. Microtec™ META29M, Microtec Research Inc. P.O.Box 60337, Sunnyvale, CA 90488

4. The XT-170 Peacemaker is a high speed bi-directional interface that fits into any full size slot on an IBM PC/XT. It permits DMA transfers between the PC and any DR11W compatible peripherals. It is available from IGC, Inc., 1290 Motor Parkway, Hauppauge, NY 11788, (516) 582-8828

The SAIVME-DR11W VME bus to DR11W Interface provides bi-directional transfers between VME bus and any DR11W compatible peripherals. It is available from Science Applications International Corporation, 2109 W. Clinton Avenue, Huntsville, AL 35805, (205) 533-5900

In principle, both of these devices should provide an interface between the 1821 SM/I using the 1821/DEC Personality Card and their respective buses. Since we have not had a chance to test these devices at LeCroy, we can assume no responsibility for their operation.


6. The 1891 FASTBUS Memory has been discontinued. Its replacement is the 1892 FASTBUS Memory.
A FASTBUS TO VME/VMX INTERFACE

J. Oliver, J. Rohlf, and A. Schwartz

High Energy Physics Laboratory
Harvard University
Cambridge, Massachusetts 02138
U.S.A.

ABSTRACT

A module is described which interfaces Fastbus to VME/VMXbus. Connection to Fastbus is made via the front panel I/O port of a LeCroy 1821 Segment Manager, providing VMEbus access to eight read/write Fastbus control registers. The main purpose of the interface is to provide DMA data transfer from Fastbus memory to a VME/VMXbus dual-port memory module via VMXbus. The data transfer rate is about 8 megabytes per second.
INTRODUCTION

The Fastbus/VME/VMX interface allows connection of a LeCroy 1821 Fastbus Segment Manager\(^1\) to a VME/VMX\(^2\) CPU module and a VME/VMX dual-ported memory module. The interface has two distinct modes of operation: a direct access mode which permits the CPU to read and write to the LeCroy 1821 control registers R0-R7 via VMEbus, and a DMA mode which performs block transfers from the 1821 data memory to the dual-port memory using the VMXbus. In the direct access mode, the eight registers of the LeCroy 1821 appear as VME memory addresses. The DMA mode requires simply that the VMX starting address and the number of words to transfer be previously loaded into appropriate interface registers prior to initiating the block transfer.

HARDWARE DESCRIPTION

The interface is a standard-size VME module with auxiliary VMX connector and operates as a secondary master. Its four-layer printed circuit board has buried ground and power planes and communicates with the LeCroy 1821 front panel I/O port. Data access through this port is limited to 16-bit words but does not require the use of a separate personality card on the 1821. To permit this port to drive bus lines over an appreciable distance (a few meters) at high speed, a small board was built which converts the pinouts to twisted-pair cable with each line having its own ground and series damping resistor. The board is positioned directly in front of the 1821 and is connected to it by flat ribbon cable. It is purely passive, requiring no power supply connections.

The direct access mode permits the CPU to read and write 16-bit words to 1821 registers R0-R7. These registers are mapped onto VME addresses and are accessed using standard VMEbus handshake protocol. The base address is jumper selectable on the interface module. The leading edges of DS0* and DS1* are used to produce a 150 nanosecond strobe S1 to the 1821, and the leading edge of the 1821 ACK* generates a DTACK* on the VMEbus. The DTACK* is held low until DS0* and DS1* are released. Failure of the 1821 to generate an ACK* within the 150 nanosecond strobe window indicates a fault condition and is reported in a status register.

The block transfer mode is controlled by a sequencer in the interface module which alternately reads 16-bit words from R4 and R5, stores them in intermediate registers, and then writes them as one 32-bit longword into the dual-port memory using the VMXbus (see Figure 1). The process continues for the total number of cycles specified in a word count register.

88
The operation is pipelined such that two new words are extracted from the 1821 while the sequencer is writing to the memory module. This mode utilizes the sequential word access capability of the dual-port memory: as each longword is written, the memory's address counter is automatically incremented until the sequential access is terminated. Only the VMXbus starting address must be given to the module. This is done at the beginning of the DMA by the sequencer, which pulls the address from two previously loaded registers. The data transfer rate for this operation is about 500 nanoseconds per 32-bit longword, or 8 megabytes per second. A jumper is provided on the interface connecting SMACK* with SMRQ*, permitting the DMA to be implemented without any VMXbus arbitration.

The interface has 13 registers which are accessible to the VMEbus (see Table 1). These registers support only 16-bit (i.e. WORD) read and write operations. The lower eight (0-7) are the 1821 read/write registers R0-R7 which are used to control the Fastbus crate. Registers 8 and 9 contain the 24-bit VMXbus starting address used in the DMA mode, and register A contains the number of 32-bit words to transfer in this mode. Register B is used to initiate the VMXbus DMA and to reset the interface module. Register C reports various status conditions: bit 0 for DMA operation, bits 1 and 2 for SMACK* and ACK* (VMX) respectively, bit 3 for DMA completed, bit 4 for a DERR* received during DMA (block transfer terminated) and bit 5 for the 1821 not responding. Since the DMA mode uses only the VMXbus for data transfer, this status register can be read via VMEbus during the block transfer.

SOFTWARE DESCRIPTION

The software for controlling the Fastbus readout is written in MC68000 assembly language for execution by a Data Sud CPUA1 which resides in the same VME crate as the interface module. The CPUA1 accesses the 1821 Segment Manager with MOVE.W instructions to and from the lower eight registers of the interface. This enables the assembly program to download Fastbus microcode from CPU memory into the 1821 sequencer, program the Fastbus data acquisition modules, and read-out Fastbus data after an event trigger.

The readout procedure first executes Fastbus sequencer microcode to block transfer data from the data acquisition modules into the 1821 data memory, and then implements a VMXbus DMA to transfer the data from the 1821 to the dual-port memory. The program knows when the Fastbus block transfer is finished by checking a status bit in R7 which is set by the microcode. The program similarly determines when the VMXbus DMA is complete by checking the interface status register, which is always accessible via the VMEbus.
### TABLE 1 Interface Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Read/Write</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>R/W</td>
<td>Register R0 on LeCroy 1821</td>
</tr>
<tr>
<td>1</td>
<td>R/W</td>
<td>Register R1 on LeCroy 1821</td>
</tr>
<tr>
<td>2</td>
<td>R/W</td>
<td>Register R2 on LeCroy 1821</td>
</tr>
<tr>
<td>3</td>
<td>R/W</td>
<td>Register R3 on LeCroy 1821</td>
</tr>
<tr>
<td>4</td>
<td>R/W</td>
<td>Register R4 on LeCroy 1821</td>
</tr>
<tr>
<td>5</td>
<td>R/W</td>
<td>Register R5 on LeCroy 1821</td>
</tr>
<tr>
<td>6</td>
<td>R/W</td>
<td>Register R6 on LeCroy 1821</td>
</tr>
<tr>
<td>7</td>
<td>R/W</td>
<td>Register R7 on LeCroy 1821</td>
</tr>
<tr>
<td>8</td>
<td>W</td>
<td>VMXbus starting address for DMA (lower 12 bits)</td>
</tr>
<tr>
<td>9</td>
<td>W</td>
<td>VMXbus starting address (upper 12 bits)</td>
</tr>
<tr>
<td>A</td>
<td>W</td>
<td>Word count for DMA (12-bit one's complement)</td>
</tr>
</tbody>
</table>
| B        | W          | Control: bit 0 = VMXbus DMA  
|          |            | bit 1 = reset  
| C        | R          | Status: bit 0 = VMXbus DMA  
|          |            | bit 1 = SMACK* (status of VMX request)  
|          |            | bit 2 = ACK* (VMX acknowledge)  
|          |            | bit 3 = DMA complete  
|          |            | bit 4 = DERR *(VMX)  
|          |            | bit 5 = 1821 not responding |

### NOTES


5. The interface is used to read-out data from 512 channels of Fastbus multihit TDC's (LeCroy 1879) for a microvertex detector in the UA1 experiment. The interface and dual-port memory, which are connected by VMXbus, reside in separate VME crates. This allows another primary master to have VME access to Fastbus data.
Fastbus/VME/VMX Interface

VMX Add. (R8)
VMX Add. (R9)
Word count (RA)
Control (RB)
Status (RC)

Sequencer

Address (24 bits)

Requester (bus jumper)

SMRQ*

SMACK*

Data (32 bits)

32 bit register

To Fastbus
(LeCroy 1821 Front Panel)

Figure 1
PLANS FOR A CERN FASTBUS DATABASE AND ASSOCIATED APPLICATION SOFTWARE

T. Kozlowski
DD Division, CERN, CH-1211 Geneva 23, and
Los Alamos National Laboratory, Los Alamos, N.M. 87545, U.S.A.

K. Osen
DD Division, CERN, CH-1211 Geneva 23

C. M. Story
Nuclear Physics Laboratory, University of Oxford, GB - OXFORD OX1 3NP

1. INTRODUCTION

The Fastbus support section in DD division at CERN plans a database system organising the large amount of information necessary to describe large multi-segment Fastbus systems, such as those being planned for LEP experiments. This database system is intended to drive automated initialisation of complex Fastbus systems, including down-loading software into intelligent Fastbus masters, and to drive general purpose diagnostic programs. In addition, the existence of such a database facility will provide a powerful bookkeeping facility for Fastbus hardware and software. Two separate but interrelated systems are envisaged: a CERN wide archival database containing functional descriptions of Fastbus devices used at CERN, and a system to support a database specific to an experiment, containing descriptions of the physical and logical aspects of a Fastbus data acquisition system. Building upon these databases, general purpose tools will be provided for specifying a Fastbus system, as well as application software for automatically initialising a system, and for running diagnostics on a system. The exact software that is implemented and its capabilities will be determined by user needs and the available resources.
A subsidiary purpose of the initial phases of the proposed project is to evaluate the use of a commercial relational database system in the environment of a high energy physics experiment. Based on studies and experiences at CERN and elsewhere, and its availability on CERN VAX and IBM systems, the ORACLE relational database system will be used for at least the initial phases of implementation. One reason for choosing ORACLE is that its user interface is based on SQL (Standard Query Language for relational database systems) which is a de facto standard for many relational database systems. It also supports a full FORTRAN interface.

This paper briefly presents the functional and implementation requirements for the proposed databases and associated application software, and the proposed implementation plan at CERN. Although many desirable facilities have been foreseen, only the basic features necessary to support Fastbus Applications software (in particular basic system initialisation and diagnostics packages) will be implemented initially. More features will be added as and when manpower is available.

2. FUNCTIONAL REQUIREMENTS

This section discusses the perceived requirements for the databases necessary to describe both individual Fastbus modules and a complete Fastbus data acquisition system. It also discusses the requirements arising from the general purpose software that is necessary for automated Fastbus system initialisation and automated Fastbus module and system diagnostics.

2.1 Descriptions of Fastbus Modules

The architectural and functional characteristics of individual Fastbus modules will be stored in a database, providing a resource for driving general purpose diagnostic programs, and also for configuring and initialising systems.
2.1.1 Module database

Archived descriptions of all known Fastbus modules used at CERN will reside in a CERN Fastbus module database. This resource will also include a mechanism to allocate new Fastbus module identification numbers within Europe. The database will eventually include the following features and information:

- Fastbus identification number allocation
- module version identifier, if more than one version exists
- full functional descriptions of modules, for example:
  - number of crate slots occupied
  - power consumption
  - number of Fastbus ports
  - Fastbus type (master or slave)
  - width of internal address for logical addressing
  - assertion/handling of Service Requests
  - assertion/handling of interrupt messages
  - support of parity
  - support of block transfers
  - support of pipe-lined transfers
  - types of broadcasts supported
  - CSR registers implemented and type of access
  - address, size and type of access for each contiguous block of data space
  - SS responses implemented
  - functional information to drive diagnostics

- operational and maintenance manual
- modification history
- hardware/software problem history

2.1.2 Module database support software

Entry of data into and maintenance of the CERN archival database will be controlled by EP Division, but an easy to use interactive interface is intended so that a designer may enter descriptive information directly into a development subset of the database. When a description has been finalised it can then be merged into the archival database.
Two basic forms of output are planned: "catalogs" of the contents of the module data base in various appropriate formats (still to be determined), and subsets of descriptive information for selected module(s) (in computer readable form), for incorporation into the description database of a particular Fastbus system.

2.2 Description of a Fastbus System

In order to implement system wide Fastbus initialisation and diagnostic software, a description of a particular system and a way for a user to generate it must exist. In this context, a "system" is the Fastbus data acquisition system for a particular experiment or test set-up, which may have from one to over a hundred segments. This section discusses the goals and requirements for this facility.

2.2.1 System database

The description of a particular system will reside in a Fastbus system database. This database will include data extracted from the CERN Fastbus module database for all those modules that are used in the system. The following additional information will be necessary in the database:

- physical location (geographical address) of all modules
- correspondance to detector components
- logical names for components (modules, segments, partitions)
- group addresses allocated for a segment
- physical location of inter-connections
- status and logical configuration of inter-connections
- arbitration levels for masters
- specification of "partitions" (sub-systems)
- access control for partitions
- owner and access rights to a system, partition, segment or module
- location of software to be down-loaded
- logical addresses (if used)
- modification history for a system, partition, segment, module
- maintenance and problem log for a system, partition, segment, module
It is likely that for indicating the correspondance of a Fastbus system element to a detector component the use of logical names for all Fastbus elements is sufficient.

A "partition" as used in the present context has a specific meaning: a Fastbus sub-system which is a subset of segments of a Fastbus system (or of a larger partition) that has no more than one logical connection (via Fastbus interconnects) to the rest of the system. It is possible that a partition may have no logical connection to the rest of the Fastbus system, for example, in the case in which it is desirable to completely isolate a sub-detector from the rest of the detector for testing. The requirement that there be no more than one connection between any one of the segments in a partition and any one of the segments in the rest of the system implies that any partition forms a complete "sub-tree" of the larger system. It may often be desirable to limit access to or from some partitions, for example, to prohibit one sub-detector system from affecting another, or to isolate a sub-system from the main system. To provide for this it is proposed to permit the definition of access attributes for a partition, which in fact can be implemented "in hardware" by appropriately setting up the routing tables and the operation passing and SR passing bits in a segment interconnect. They are: any access or no access into the partition from the outside, or any or no access or only access to specified segments from the partition to the outside. The latter case, for example, would allow a master in a partition to send an interrupt message to a specific external segment.

The concept of owner and access rights for any Fastbus component (for example, read or write access for anyone, for the owner, or for only a specially privileged user to a particular module) permits the implementation of protection mechanisms at the software level, so that a mandatory protocol for any program that accesses Fastbus might be to check its access rights first. This facility is in addition to the normal access protection provided by the Fastbus arbitration protocols.

2.2.2 System specification input and output

An important part of the system description facility will be the user interface for specifying a Fastbus system. Besides its main purpose of defining an existing system, it can also be a useful tool to aid in the design of a new system or of a system reconfiguration. The following capabilities are envisaged:
The proposed facility will provide general purpose software to carry out automated initialisation driven by the Fastbus system and module databases. It will allow initialisation of a complete system, partition or segment, or even re-initialisation of an individual module. An important requirement is that the initialisation process be fast (a few minutes for an entire system) to keep the time to start a run within reasonable limits. Because of the large amount of components involved in proposed Fastbus systems, achieving this goal will not be trivial. As part of the solution the initialisation software ought to be designed in such a way that intelligent masters (or hosts) in the Fastbus system could carry out the initialisation tasks in parallel. Throughout the initialisation process user "hooks" will be provided at logical points, so that user defined procedures can be invoked at almost any time.

Some detailed planned features of the initialisation facility are:

- loading of routing tables
- resolution of master/slave symbolic cross-references
- down-loading of user specified software and data into the appropriate modules
- initialising specified CSRs with specified bit patterns
- protection against accidental or unauthorised initialisation of a sub-system
- verification of the physical presence and type of all modules
- verification of specified master-slave communication paths
- optional invocation of user specified procedures

In order to provide a flexible system, it will be possible for users with appropriate privileges to reconfigure the system (partitions, segments, interconnects, etc.) at any appropriate time. It will be possible for the user to interactively update the system database, automatically verify the hardware based on the update, and optionally initialise the reconfigured subsystem (route tables, software down-loading, etc.).

2.3.2 Fastbus Module and System Diagnostics

As in the case of system initialisation, with the existence of a database containing diagnostic control information and a detailed description of a Fastbus system, including functional descriptions of Fastbus modules, it is possible to implement a facility to automatically run diagnostic programs on the system. Again, the complexity of future Fastbus systems really requires some kind of
automated diagnostic procedures. The availability of an automated diagnostic facility makes it much easier, and therefore more likely, to run diagnostics on the Fastbus system or a subsystem whenever free time is available, or if there is some small suspicion there is a Fastbus problem. With a less automatic system, requiring more effort to run, it is more likely that running diagnostics will be delayed until absolutely necessary.

The facility to be provided will be as general purpose as possible, being driven by the functional descriptions of modules in the database. For those cases where more detailed or application or software dependent diagnostics are necessary, provision will be made for user "hooks" at appropriate points. It will be possible to run diagnostics on the entire system, or to limit them to a sub-system (partition, segment, module). Some additional intended features are:

- optional exclusion of specific components as diagnostic targets
  (partitions, segments, modules, segment inter-connects)
- a range of diagnostics options
  (verification of ID, read access, write access, exhaustive functional tests)
- localisation of communication path failures using segment interconnect route-tracing

3. IMPLEMENTATION RELATED REQUIREMENTS

Some topics not directly related to functional aspects of the Fastbus databases and associated application software that must be considered are constraints on the interfaces between users and the database system, between application software and the database system, and the hardware environment in which the proposed software must run.

3.1 Interfaces for Users and Software

Initial implementations of the database software will be on the ORACLE commercial relational database system. However, it is not obvious at this time that ORACLE will be adequate for all intended applications (an evaluation of ORACLE
is a subsidiary goal of this project). Also, some groups are using or considering using database systems other than ORACLE. Therefore it is important that the software for the proposed facilities be written to be independent of any particular database system, but still achieve adequate performance. Therefore, software routines that access the database will be implemented in levels:

- an application dependent level
  (e.g., "retrieve logical address for module A")
- a database dependent level in SQL format
  (e.g., "from table=MODULE retrieve LOGICAL-ADDRESS for NAME=A")

ORACLE already includes a set of FORTRAN routines that implements the SQL level. To provide access to another database system it will then be necessary to implement for the new database, either the application level routines, or (if it is a relational database) possibly only the database level routines.

In order to facilitate portability across different computer systems, standard DD-OC support software will be used wherever possible, for example, standard message reporting protocols, standard graphics and interactive terminal software.

A set of user and programmer manuals will be written describing the structures of the databases, and how to use the provided software interactively and at the software level.

3.2 Target Systems

At present ORACLE is supported on VAXs, IBM mainframes, IBM PCs and soon on Norsk Data systems. This covers many, but not all, of the systems commonly used at CERN in the data acquisition environment. However, other processors (Motorola 68000s, for example) that do not support ORACLE are used both as part of large systems or as small standalone Fastbus systems. Some LEP experiments (and UA2) will employ several processors (VAXs) in networks. The databases and associated application software must be used in these multiprocessor and "non-ORACLE" processor environments. It is not known at present exactly how database systems will be used and accessed, and what the performance requirements will be on these diverse systems. The different environments must be accommodated by whatever software systems are designed.
As discussed in the previous section associated software will be written to be as portable and database independent as possible. In addition, the database structures themselves will be designed with relatively simple structures so that it is possible to "port" them with a minimum of effort to non-ORACLE environments. In some cases it may be necessary to port only portions or "summaries" of databases in sequential file format, or as simple blocks of data in special cases in which no mass storage device is available on a system. The application level database routines discussed in the previous section would have to be modified accordingly to support such differently structured "databases".

Data acquisition systems distributed across computer networks will probably require implementing database access, and the associated application software across these networks. ORACLE apparently is already supported across DECNET. Non-DECNET network support (if necessary) will probably require some development effort.

4. IMPLEMENTATION PLANS

At CERN, plans are being defined for implementing the proposed Fastbus databases and associated application software. Although relatively complete designs will be developed at the outset, the actual implementation will be done in stages, as user needs and available resources dictate. The proposed projects fall into four areas, each of which may be developed somewhat independently:

- the Fastbus module database
- the Fastbus system database and associated user interface
- automated module and configuration diagnostics
- automated initialisation

These projects are interrelated, and all system development must be coordinated. This is especially true in implementing the subroutines for accessing the databases from application software. Each of these areas is discussed in more detail below.
4.1 Fastbus module description

A project is currently underway to implement a module database and associated interactive tools, primarily for the purposes of allocating Fastbus identification numbers within Europe and generation of catalogs of known Fastbus modules. This database will be the basis for an expanded one that incorporates full functional module descriptions for driving diagnostics and initialisation.

Although provision should be made at the outset for including additional information in the module database (operational manual, modification history, problem history) the actual implementation of these items should be done at a later time.

4.2 Fastbus system database and the user interface

The initial task will be to define the database structure for a full physical and logical description of a Fastbus system. In parallel, the user interface for entering data into and for displaying data from the system database will be defined. The initial implementation of the user interface will allow the user to interactively specify the physical and logical structure of the Fastbus systems. The user will be able to specify the module contents of all segments, and physical and logical interconnections (by directly specifying routes), partition structures. Segment group addresses, logical addresses, and arbitration levels ought to be specified "manually" in the first version. A graphical interactive interface is desirable; however, so that a useable product is available quickly, the initial implementation will probably use a simple conversational or menu interface. Experience with the initial system should help determine what is needed in an improved version.

Additional features will be added in later versions as needed, for example, error and consistency checking, protection mechanisms, support for simultaneous versions, automatic allocation of group and logical addresses, arbitration vectors. A facility to generate database summaries for incorporating into user reports and logging with data should probably be delayed until needs are more clearly understood, i.e., closer to the time when this feature will actually be used.
4.3 Module and Configuration Diagnostics

There exists a program (FBSTEST) that is a general purpose Fastbus diagnostic to exercise memory and common Fastbus operations. The characteristics of Fastbus modules that it supports are presently stored internally and modified via a menu.

The first version of the diagnostic facility will be built upon a version of the module database that contains module functional descriptions equivalent to those incorporated in the existing FBSTEST program. Therefore the first priority is to define this database structure and to modify FBSTEST to access the database. The first version will perform tests on standard Fastbus slave functions; a subsequent version will include a mechanism for invoking user supplied tests.

A later version of the module description stored in the database will provide a more complete functional description. Corresponding additions to the previous diagnostics program will provide a full general purpose diagnostics facility at the Fastbus module level. When the full configuration data is available in the Fastbus system description database, full configuration testing will be added for segments, complete systems and partitions. Features will also be added to prevent specified system components from being affected by testing.

4.3.1 Initialisation implementation

The automated initialisation facility is essential for the large LEP Fastbus systems, and it is also of use for much smaller systems of only a few segments. As a first step a skeletal high level facility will be implemented that provides a framework that automatically traverses the system in an orderly manner, initialising the segment hardware and loading routing tables, but invoking user software to do the detailed module level initialisation. This first version will also verify module existence. At the same time, it is probably wise to specify the means for implementing "parallel initialisation" (the low level initialisation being done by processors other than the host computer).

Later versions of the initialisation software will allow specification of sub-systems for initialisation, and down-loading of user software including resolving symbolic references. Later versions will also provide protection mechanisms to help prevent unauthorised initialisation, and the ability to reference different versions of the system description.
CERN SUPPORTED FASTBUS INTERFACE Routines

E.M. Rimmer

DD Division, CERN, 1211 Geneve 23, Switzerland

INTRODUCTION

This paper describes the CERN implementation of the original Draft Proposal for Standard FASTBUS Interfacing Routines. Methods used to maximise portability of both the implementation and application code are discussed. The advantages and disadvantages of using a common Library and List Processing techniques are presented, with an appraisal of the experience gained over the past two or three years using the Routines with a variety of Host/FASTBUS Interfaces.

SCOPE OF THE IMPLEMENTATION

The CERN FASTBUS Routines package is based on a version of the Draft Proposal for Standard FASTBUS Interfacing Routines which was available in 1982/83 [1]. The Draft underwent the natural evolution of any such document, and it proved either undesirable or difficult or both for the CERN implementation, amongst others, to follow the detailed changes. In-house additions and modifications were made which inevitably led to divergences. The CERN package will be updated as soon as an official Specification is published.

The CERN package differs from the Draft Proposal mainly by omission, for no good reason except lack of effort. The main differences are:

• Access Routines to set up control structures and decode returned errors are not available. The usual technique of INCLUDE files to define system parameters is used instead, to minimise portability problems in application code. Error decoding has recently been implemented and is in the process of being released.

• Only a sub-set of the Cycle Primitives is available. This is a great pity, as experience has shown that for complex FASTBUS sequences, code written in terms of Cycle Primitives is more understandable and therefore more debuggable than option and parameter manipulations used to suppress sub-cycles in the full FASTBUS Transaction Routines.

• Of the Compound Routines, only Segment Interconnect Route table access is supported. Block Moving Routines are relatively easy to add, but Sparse Data Scans and Service Request Handling algorithms pose problems which need more careful study.

• A super-set of the Line Access Primitives is supported. This CERN extension, to allow read/write access to all Lines, was included for basic Crate testing and debugging, and has proved invaluable for tracing backplane defects, connector short circuits, and so on.

As with any implementation, a particular Interface is supported either by the full set of Routines or a sub-set matched to its FASTBUS capability.

SUPPORTED INTERFACES AND HOSTS

Most of the Interfaces supported consist of two separate modules: one, hereinafter referred to as 'the Interface', is in a FASTBUS backplane segment, the other is an I/O module in a Host-connected bus. The two modules are joined by a parallel cable Interconnect, driven by I/O Library or Driver Routines in the Host. The Interconnect is used either to generate the FASTBUS protocol sequences in the case of simple Interfaces, or to carry messages in the case of intelligent Interfaces.

The simple FASTBUS-to-I/O-Register Interface F|ORI [2], the intelligent CERN FASTBUS Interface CFI [3], and the Fast Sequencer Interface FSEQ [4] are supported with CAMAC or VME Interconnects. The CFI can also be used with UNIBUS or Q-bus Interconnects. The LeCroy 1821 Segment Manager is supported only via a CAMAC Interconnect.

A FASTBUS Interface Emulator which generates FASTBUS Timing Diagrams on a Graphics Terminal as calls are made to the Routines is available for educational purposes.

The package is in use on VAX, M68000-VME, Norsk Data 100 and PDP-11 systems and has been ported with ease to several installations where the CAMAC or VME Interfaces differ from those used in the original implementation.
The package has also been implemented for a FASTBUS General Purpose Master GPM [5] built as a single-board FASTBUS module with a M68000 CPU and memory. This implementation is not yet completely tested.

**DESIGN OF THE IMPLEMENTATION**

To minimize software development and maintenance, as much Host and Interface independence as possible is desirable. The implementation, mainly written in FORTRAN-77, uses List Processing techniques. In areas where portability is irrelevant, namely, the VAX UNIBUS/Q-bus Interconnect Routines and the VME I/O Library, it was written in Pascal and M68000 Assembler respectively.

**List Processing**

All user calls to the FASTBUS Routines access a unique List Handler, which initializes a List array, fills it with intermediate code, named F-Code, and calls a common List Execution Routine. F-Code Lists are Interface independent and define sequences of FASTBUS operations which are interpreted and executed by List Processor software or firmware.

Run-time switches configure the Routines to use a particular combination of Interface and Interconnect. The common List Execution Routine calls the List Processor appropriate to the selected Interface. The List Processor performs the necessary Interconnect I/O to control the Interface and generate the required FASTBUS operations. List Processors call common Interconnect I/O Routines, based on the NIM/ESONE CAMAC Software Standard [6]. The common I/O Routines then call the appropriate Library Routines for the selected Interconnect.

In Immediate Mode, where a single call to a FASTBUS Routine proceeds through the execution phase, the List is internal to the package and of no concern to the user. In User List Processing mode, the user provides the workspace and is responsible for calling the List Initialization and Execution Routines. Calls to FASTBUS Routines after Initialization and before Execution result in List filling only.

User List Processing allows complex sequences of FASTBUS operations to be built up before execution. However, all non-FASTBUS code between List Initialization and List Execution is executed when encountered and not during List Execution, because the present List Processors are only capable of FASTBUS I/O. A mixture of FASTBUS I/O and computation can only be achieved by Immediate Mode calls embedded in non-FASTBUS code. Although adequate for test and diagnostic purposes, this can be a severe disadvantage for some applications.

With this software architecture, the addition of a new Interface or a new Interconnect is achieved with relatively little effort and with no disturbance to existing code. Word-size and longword/addressing differences in the Host machines are accommodated in a single software module which essentially defines a few global parameters.

**APPLICATION ENVIRONMENTS**

**Test and Development Systems**

The homogeneity of the test and development environment at CERN has improved over the past few years with the introduction of PILS [7], a Portable Interactive Language System, written in PASCAL and installed on VAX, M68000-VALET-Plus [8], M68000-GPM and ND-100 machines. Given the availability of most CERN Libraries, including FASTBUS, on all of these machines, test programs in PILS can be ported from one system to another without modification. Run-time selection of the FASTBUS Interface and Interconnect also enhances portability. A user program can run with a VAX/UNIBUS/CFI system then with a M68000/VME/TOURI system with no changes. The recent introduction of an in-memory compiler for the PILS language, which combines full interactivity for program development with the efficiency of compiled code, has almost entirely removed the need for test programs to be written in FORTRAN. This has considerably simplified the work of designers and maintenance engineers.

**The CFI in Data Acquisition Systems**

The CFI has been used, with CAMAC Interconnects, for Data Acquisition at the NA31 experiment [9] and for LEP equipment testing by the ALEPH and DELPHI Collaborations. The programming language used was FORTRAN.

The CFI has an on-board M68000 and a fast memory buffer and is coupled to FASTBUS by a micro-coded sequencer, which essentially executes F-Code instructions. FASTBUS Lists are stored in the CFI prior to execution, and data are transferred between FASTBUS and the CFI buffer at speeds approaching those of FASTBUS after a List has been triggered for execution by a 32-bit message sent over the Interconnect. Write data are loaded before List execution and read data removed after execution at the speed of the Interconnect. If the on-board memory is insufficient for the total data transfer, data in one or other of the directions can be moved directly between FASTBUS and the Interconnect, when the bandwidth is determined by the slower of the two operations, the FASTBUS cycles and the Interconnect DMA cycles.
NA31 used the CERN Routines for List Generation and down-loading into the CFI. List Execution was triggered in real-time by CAMAC QIO calls following receipt of a CAMAC LAM. ALEPH used the standard CERN VAX Data Acquisition package, in which the CAMAC driver executes CAMAC Lists in response to a LAM. DELPHI used the Routines directly from a background program, to read-out FASTBUS with the CERN ND-100 Data Acquisition system.

As stated earlier, the CFI FASTBUS Sequencer executes only FASTBUS I/O and does not at present support conditional or computational procedures. The ALEPH group had a FASTBUS Time Projection Digitiser Module to read, which required dynamic modification of the read-out sequence. The restrictions of the CFI were overcome thanks to a certain flexibility in the CERN VAX CAMAC List interpreter used for Data Acquisition. A part-empty list was held in Host memory, filled with parameters read from the TPD via a List already in the CFI. Then the completed List was down-loaded into the CFI and executed. This was clumsy but at least possible.

COMMENTS ON THE IMPLEMENTATION

Having implemented and used the Routines for a diverse range of Host systems, Interfaces and applications, a non-exhaustive list of comments are noted, covering the main aspects of the software and hardware designs.

ROUTINE LIBRARY

Advantages

- No special language or special compiler is required. With the advent of FASTBUS, CERN decided to abandon previous methods used for supporting CAMAC, where CAMAC I/O actions were embedded in the syntax of a special interactive language such as CATY. Not only does such a language have to be implemented on several Hosts, but Routines must be provided in addition for use with ‘normal’ languages such as FORTRAN. Use of a portable language or widely available high-level languages and a portable FASTBUS Library is overwhelmingly more economic.

- The support and maintenance load is reduced. A single package can be called from any set of languages which have compatible calling sequences and only one set of documentation is required. Language translators are easier to implement if required.

- The user quickly learns the functionality of the Library whatever the language environment, and can more easily move between languages, for example, from PILS to FORTRAN.

Disadvantages

- If the overhead for a subroutine call is excessive, then use of a Library can reduce run-time efficiency.

- Library Routines are not necessarily suited to real-time Data Acquisition, where it may be necessary to embed read-out protocols in a Device Driver for efficiency or because of system constraints.

LIST PROCESSING

Advantages

- As the List Handler can be written to be largely Host- and Interface-independent, List Processing eases the job of producing multiple implementations. A large amount of common code simplifies code management and debugging and ensures that all implementations keep in step.

- Once a List Processor is available which can handle all F-Codes, new Routines are easily added at the common List generation level.

- List Generation can be debugged for conformity to the FASTBUS protocol specification, thereby safeguarding the uninitiated user from programming errors.

Disadvantages

- If the List Codes are defined primarily for I/O, as is the case for F-Codes, then conditional and computational procedures whilst executing a FASTBUS sequence can be extremely difficult to handle. The only solution, if List processing is preserved, is to extend the F-Code set, with the obvious risk of inventing yet another machine language.

- If the intermediate code is interpreted at run-time, the method can be inefficient compared to executing embedded code. This depends on the hardware architecture of the Interface. In the case of intelligent Interfaces with a tightly-coupled connection to FASTBUS via address mapping or co-processors, the Lists must be translated to in-line code to retain efficiency.

HOST INTERCONNECTS

Advantages
• The use of simple Interconnects allows the support and maintenance problem to be separated into two distinct areas; that of the Host I/O module, which is often supported by the Host manufacturer, and that of the FASTBUS Interface, which is more likely to be the responsibility of the application experts.
• Having one FASTBUS Interface which can be connected to a variety of Hosts and Interconnects makes economic sense to the Interface manufacturer and reduces effort required for training, trouble-shooting, spares pooling, etc.
• The simplicity of the Host I/O connection eases error diagnosis, since the interfacing complexity is more or less isolated in the FASTBUS system itself.

Disadvantages

• Use of an Interconnect may limit bandwidth. However, FASTBUS is designed to house data processing elements, and the connected Host is essentially used only for data recording. In this case, the Interconnect bandwidth may be more than adequate, as long as data can be removed from FASTBUS into an intermediate buffer at high speed. High-density recording devices directly interfaced to FASTBUS are already being designed, which will make the Interconnect bandwidth of even less importance.

MESSAGE EXCHANGE

Advantages

• Message Exchange is increasingly used for communicating in Multi-Processor systems [10]. Modern FASTBUS Interfaces have on-board processors and can be treated as any other processor in the system. Therefore, the use of messages to communicate across the Interconnect increases the software homogeneity. With layered communication software, details of the I/O port used to exchange messages can be implemented at a very low level and much of the code can be common to all embedded processors.

Disadvantages

• With an intelligent buffered Interface there are no disadvantages obvious to the author.

THE 1983 DRAFT PROPOSAL

The following comments cover only the overall aspects of the 1983 Draft Proposal. Account has been taken of some features included in the proposed Revision of that Draft [11].

Inadequacies

• In general, the 1983 Proposal contained most features required of a basic set of FASTBUS access Routines.
• The ability to work without any knowledge of the implementation method, in this case List Processing, is of considerable importance to non-programmers.
• The ability for sophisticated users to manipulate Lists has proved useful. This is to be compared with the flexibility of Assembler code programming and carries similar risks and portability penalties.
• The fact the workspace for generating complex Lists was provided by the user, avoided the need for an implementation language which supports dynamic memory allocation.

• The basic Standard must be entirely Host, Interconnect and Interface independent. If necessary, Appendices can define specifics.
• The Standard must define conformity levels for implementations. A basic mandatory set of Routines must match identically the mandatory hardware functionality of any FASTBUS Master.
• Protection mechanisms and resource sharing techniques should be defined to allow use of the Routines in multi-taking, multi-user systems.
• A complete set of Line Access Routines is needed for basic debugging. This cannot be part of the basic mandatory Routines, as it is not mandatory for FASTBUS Masters to have individual line access capability.
• The use of Cycle Primitives should be given more ‘respectability’ as it eases code writing and debugging. With increased exposure to FASTBUS, users are now becoming sufficiently mature to generate efficient and correct sequences. The number of control flags could than be considerably reduced, which would also simplify coding.
• The supported data buffering modes are somewhat restricted. Gathered writes and scattered reads maybe necessary, although they may not codeable in all implementations.

ACKNOWLEDGMENTS AND CONCLUSIONS

The present CERN FASTBUS and associated Routines have been incrementally implemented over the past two or three years by the following
people; T.J.Berners-Lee, C.Bizeau, D.Burckhart, R.Divia, E.M.Rimmer, J.Schiavi and C.M.Story. Of the many users who contributed comments and suggestions, the most vocal was K.Peach, to whom we extend our special thanks.

We are in no doubt as to the utility of an International Software Standard for FASTBUS Interfacing Routines. We strongly encourage the ESONE and NIM Committees to ensure that the present Revision discussions are rapidly brought to a fruitful conclusion. With the publication of an Official Specification, all implementations can stabilise and the real benefits of standardisation can be guaranteed to all FASTBUS users.

REFERENCES


[6] Subroutines for CAMAC, ESONE/SR/01; DOE/EV-0016; IEEE Std.758


[9] K.Peach and members of NA31, FASTBUS Software in the NA31 Data Acquisition Environment; these Proceedings

See also, T.J.Berners-Lee, Use of FASTBUS Software at CERN; these Proceedings


PROPOSED REVISION OF THE FASTBUS STANDARD ROUTINES

S. Gannon, D. Hansson, J. Marsh, C. Van Ingen, FERMILAB
D. Sherden, SLAC
E. Siskind, NYCB Real—Time Computing, Inc

ABSTRACT

A proposal for major revisions to the Fastbus Standard Routines has been made to and accepted in principle by the U.S. FASTBUS Software Working Group. To enhance portability of application code, the control structures used to define and control Fastbus operations have been made internal to the Standard Routines, rather than directly accessible to application code. A generic control structure is proposed as an optional sub—standard to be used within the Standard Routines. Additional modes for the specification of data buffers have been added for increased flexibility. Specifications for default automatic reporting of errors, which may be overridden by application code, have been added to the standard. Further modifications of and additions to the Standard Routines have been made, principally in the areas of list construction and manipulation and enhanced multi—user capability. The rationale for and a brief description of the major changes are presented.

OVERVIEW

In the revised standard (and, to a large extent, the existing standard as well), the central concept is that of the “list”. A FASTBUS interface is assumed to be a list processing device capable of executing a series of FASTBUS operations, either through its own onboard intelligence or through software emulation within the driver portion of the Standard Routines. From the point of view of application routines a list is composed of “list elements”, in which a call to a large class of FASTBUS routines generates a single element, e.g. a block read from data space or modification of an interface characteristic. The Standard Routines for FASTBUS transaction processing can then be divided into three categories: those used for list construction, list execution, and status decoding. In a fashion similar to the existing CERN implementation (2) of the Standard Routines, list construction can be thought of as the compiling of application routine requests into FASTBUS “code”. Ideally, the code produced would be interface—indepedent, although this is not mandated by the standard. List execution performs the interface—specific actions required to carry out the operations encoded into the list. List execution is independent of list construction, although it may be triggered by list construction (immediate execution mode). Thus a list need be constructed only once, and can then be executed many times. During execution, status information is transferred from the interface to the list, and the final stage of processing consists of decoding and error reporting of the status information.

In the remaining sections we describe the rationale for and a brief description of the major proposed changes to the standard. Because we are attempting to justify the need for these changes, we are of necessity critical of the existing standard. This should not be interpreted as a blanket indictment of the original work. While the proposed changes are major, the present proposal is intended as a revision of the original, and as such owes a great deal to the existing standard in its concepts.
Open vs Hidden Control Structures
Modularity and Portability

The existing Standard provides for four separate user-defined control structures: the Operation Control Block, the Status Block, the Operation Parameter Block, and the List Storage Buffer. Each of these structures is specifically designated as being implementation dependent, although a recommended format is offered in most cases. In practice, existing implementations differ dramatically in their control structure formats. While the standard does define subroutines which allow application routines to manipulate the structures in an implementation independent fashion, the rationale for requiring application routines to provide and manage such structures is missing. Any direct use of the structures by application routines would destroy portability and defeat the purpose of standardization. In practice, the situation is even worse. The CERN implementation, for example, fails to provide the implementation-independent support routines, so that application routines not only can, but must, manipulate implementation dependent parameters. On more general grounds, modular coding practices dictate that, to the extent possible, higher level (application) routines should not be required to know the details of the structures used by routines at lower levels. While the knowledge of the existence of control parameters might be required, the detailed locations in which these parameters are stored is usually not. This allows for future changes in lower level structures which do not propagate to higher levels. It also permits modularity in the standard as well as modularity in the code.

In the revised standard, all control structures are dynamically allocated and managed by the Standard Routines themselves. Application code requests allocation of one or more FASTBUS "lists" through one or more calls to FB_LIST_ALLOC. This routine returns as an argument a "list ID", which the application routines use to identify the list in subsequent calls to Standard Routines. A list may be used to store a sequence of FASTBUS operations for later (repeated) execution, or, when operating in immediate execution mode, may be reused for varying FASTBUS operations. During list execution status information from the operations is stored internally and associated with the list. Application routines access the status information in implementation independent fashion using the list ID and calls to Standard Routines.

The fact that the list structure is submerged into the Standard Routines eliminates the requirement of mandatory control structure specifications in the Standard Routines, allowing greater flexibility in their implementation. This does not, however, imply that further standardization of the control structures themselves is undesirable. In the CDF implementation (3) of the revised Standard Routines, an attempt was made to define a generic (interface independent) list structure which was sufficiently general to allow for future expansion and for interface specific features. Furthermore, the interface-specific and operating-system-specific portions of the code were separated from that of more general applicability. This provides transportability for the bulk of the implementation, requiring re-coding of only a limited subset of routines for adaptation to different interfaces or operating systems. We advocate that the generic list structure developed into a secondary standard, but emphasize that it is not a mandatory feature of the revised Standard Routines.

Operation Parameters
Multi-User Execution Environment

In addition to providing for FASTBUS data transfer operations, the Standard Routines must provide for the control of the execution environment through operation parameters, e.g. waiting/non-waiting execution, hold bus, timeouts, error responses. In both the existing and revised standards, a distinction is made between single bit, or flag, parameters and numeric parameters. Because flags occupy very little storage space, it is practical to embed them in each element of the list to be processed. For numeric parameters, however, the increased storage required makes this impractical. In the existing standard, such parameters are defined in a single Operation Parameter Block. A naive interpretation of this standard would imply that changing an Operation Parameter in a multi-user environment would affect all processes. Even with a more liberal interpretation of the Operation Parameter Block as being defined on a per process basis, similar problems arise in a multi-threaded (e.g. interrupt/non-waiting execution) environment. On a more mundane level, an application routine which modifies a parameter for a particular operation but fails to restore the default value can cause the failure of correctly coded FASTBUS operations in other parts of the same program. For multi-element lists, the list is frequently constructed in one part of the program and executed in another. Since the Operation Parameters must be changed at execution time, the definitions of the FASTBUS operations to be performed and the environment under which they will be executed are separated in an awkward fashion. Furthermore, in the execution of a given list, the Operation Parameters must be the same for all individual elements of the list. Thus, for example, element by element
specification of error responses to non-zero SS codes is prohibited.

In the revised standard the specification of Operation Parameters is explicitly integrated into the list to be executed. A set of default parameter values, applicable to all FASTBUS lists unless explicitly modified within the list itself, is defined. In addition to FASTBUS transaction list elements, which perform normal data transfer operations, parameter modification list elements, which modify the execution environment, are defined. Since list elements are executed sequentially, the environment can be changed dynamically during the execution of the list. The storage problem is alleviated, since the parameter modification list elements need to be included only when one wishes to alter parameters from their value for the previous list element.

Data Buffer Access Modes

The existing standard has two data access modes: Data can be transferred to or from either the List Storage Buffer itself or separate user specified input and output buffers. The input and output buffers are specified when the list is executed.

The revised standard defines four data access modes which can be independently specified on an element by element basis:

- Inline—Data are embedded into the list itself. Inline mode is specifically restricted to single word transfers. For inline writes, the value of the data is specified at list construction time. For inline reads, a supporting routine is provided to retrieve the read datum from the list after execution.

- Random — Data are transferred to/from a buffer whose address is specified at list construction time for an individual list element. Thus data can be transferred to or from different locations on an element by element basis.

- Indexed sequential — Data are transferred to/from a single list-wide data buffer (see below). The index within the buffer at which the transfer should begin is specified at list construction time.

- Offset sequential — Data are transferred to/from a single list-wide data buffer. An offset is specified at list construction time. When the list is executed, this offset, relative to the end of previously transferred data, determines the starting transfer address for the list element. (The tightly packed case referred to above would correspond to an offset of zero.)

The sequential buffers referred to above are specified using a special list element invoked through the routine FB_DECLARE_SEQ_DATA_BUFFER. Sequential buffer may be declared more than once in a list, so that more than one sequential buffer can be used. During execution, however, such re-definition of the sequential buffer will cause the "memory" of the current location in the previous buffer to be lost (i.e. you can't go home again).

Status and Error Handling

In the existing standard all routines include as an argument a status array into which status and error information can be encoded. Standard Routines are provided which allow application routines to extract error or word count information from the array in an implementation independent fashion, or to report all errors in the status array. With the exception of the single error reporting routine, the Standard Routines themselves do not report errors (other than by returning information in the status arrays), so that the responsibility of checking for, decoding, and reporting errors, is left to the application routines. From a purist's point of view, this is a correct approach, since the calling routine has available to it contextual information which allows it to better manage the error handling. Experience in the high energy physics environment, however, has shown that it is far more effective to provide default automatic error reporting with provision for suppression of reporting for particular situations or particular errors.

Default automatic error reporting is provided in the revised standard. When a list is constructed, several mechanisms for controlling the reporting at list execution time are provided. Error reporting can be suppressed entirely, or specific anticipated errors (e.g. specific non-zero SS codes) can be suppressed on an element by element basis. For groups of elements within a list, the error reporting routines provided by the standard may be replaced by user-specified dispatch routines. Additional routines are provided to allow application programs to decode, sort, or report errors.

In the existing standard, status information from subroutine calls and status information from executed lists are treated in the same fashion. In the revised routines they are treated slightly differently. In a manner patterned after that used by the VAX/VMS operating system, all routines include as an argument a single word return code, which summarizes the success, failure, or status of the requested function. Application routines may then check the return code against a standard success code (FENORM), a severity level (success, informational, warning, error, or fatal), or a particular error code. For every return code there is
an associated line of text which can be used for reporting purposes. Because the return codes can be handled in implementation independent fashion, the application code to call a Standard Routine and check for success usually requires only two lines of code, compared to three in the existing standard. Coupled with the automatic error reporting, this results in simplification of application code. The status information from list execution is stored internally in implementation—dependent form by the Standard Routines. This information may contain indications of multiple errors, which can be decoded into individual status codes of the same form as return codes. While Standard Routines which perform list execution may return a single return code which summarizes the success or failure of the operation, application routines must use the decoding routines to access the full status information.

**Immediate Mode Execution**

**Permanent Parameters**

As has already been mentioned, the concept of the FASTBUS list is more central to the revised standard than in the existing standard. Naively one would like to think of immediate execution mode as simply the construction and immediate execution of a single element list. However, because it is expected that application code will make extensive use of immediate execution mode, it is important that its use be kept as simple as possible. A number of features have been added to the revised standard to make this possible.

The most obvious example of a potential complication is the total life cycle of a list: allocation, construction, execution, and deletion. It would clearly be excessive to require application routines to call four separate Standard Routines for every FASTBUS operation. Consequently, provision is made for reuse of allocated lists. At program initialization, the application routine allocates a single list. In immediate execution mode a call to a FASTBUS transaction routine causes the requested list element to be contracted and the list to be executed. If a second call is made to a transaction routine, the list is first reset (i.e. the previously added list element is deleted), the new element constructed, and the list executed. Thus successive immediate execution mode FASTBUS transactions can be accomplished through single subroutine calls. (Note that the list is not cleared immediately after execution; this is to allow application routines to interrogate status information after list execution.)

A second complication arises from the association of Operation Parameters with specific lists (although the complication is small compared to those which arise from not doing so). Within the Standard Routines a distinction is made between transaction list elements and parameter modification list elements. In immediate execution mode, the addition of a transaction element, but not the addition of parameter modification elements, causes the list to be executed. Thus application routines may freely modify Operation Parameters (and, in so doing, construct a multi—element list) which will apply to the eventual FASTBUS transaction which they wish to execute. In specifying Operation Parameters one can envision two somewhat different situations. In one case, an application routine may wish to define an operating environment which will apply to a number of different FASTBUS transactions. In the second case the routine may simply wish to alter the default values for a single transaction. To simplify the handling of these two cases, the concept of a "permanent" Operation Parameter is introduced. Permanent parameters are added to a list before non—permanent parameters or transactions. When a list is reset, the non—permanent parameters are reset (i.e. their list elements are deleted), but the permanent parameters are not. Thus an application routine may define an execution environment which becomes permanently associated with a given list. This feature can be used in delayed execution as well as immediate execution mode.

**Acknowledgement**

This work was supported by the Department of Energy, including contract DE—AC03—76SF00515 and SBIR contract DE—AC01—83ER80078.

**REFERENCES**


ABSTRACT

Dans le cadre de l'expérience LEP 3, un interface FASTBUS Maître a été conçu pour être associé au microprocesseur XOP.

Cet article décrit la philosophie de l'interface, son fonctionnement interne et la programmation de FASTBUS par XOP. Quelques exemples sont donnés dans les différents modes de fonctionnement ainsi que les performances de l'ensemble.

A FASTBUS MASTER interface has been designed for the XOP microprocessor used in the L3 experiment.

This paper describes its design philosophy, its internal sequences and the related FASTBUS XOP software. Examples are given in the different modes and the corresponding performances are quoted.

* CERN - DIV.DD.
XOP is a fast microprogrammable processor developed at CERN. (1) Its reliability and its modularity allow special user extension. To implement XOP as a trigger processor in the L3 experiment we have designed a fastbus master interface which operates as an internal XOP module, running in parallel with the other modules.

This module is an XOP dedicated interface designed as an XOP extension. It consists of an XOP card linked to one Fastbus card per master (Maximum: 2 masters per interface). (2)

The Fastbus instruction is an horizontal extension of the XOP micro-instruction which is increased from 160 to 190 bits.

This new field is as independant as possible of the old ones to allow simultaneous processings.

Fast execution is obtained by the use of two internal sequencers. One is used to share the 4 typical fastbus operation on a single XOP microinstruction (i.e. arbitrate, primary address, secondary address, data cycle).

The other is used to manage a complete pipe line transfer at a programmable speed (up to 125 ns/32 bit word).

All these operating modes, timings and busses source/destination are defined in the 29 bits of the microcode described below.

Fastbus addresses, data, Control and Status are pipelined into 12 16-bits registers to allow a good synchronization between XOP and Fastbus.

Two extra registers are used to program the word count and the speed of transfers executed in pipe line mode.

Synchronous and asynchronous modes are both available to avoid the dead time due to cycle by cycle synchronization. The synchronization of the module on XOP master clock is accessible by software via a special bit in the fastbus field. (hold)
A flag, named fastbus flag has been added to the old ones (carry, zero, overflow, sign, counter zero). This flag, testable during any other XOP action, allows a constant check of the fastbus operations.

**FASTBUS OPERATIONS AND SEQUENCES**

The module is activated on the fastbus side by a non zero configuration on its "Do Something" field. (Cf table 1 micro instruction XOP, fastbus field.) Fastbus operation is then executed at the fastbus speed.

On the XOP side, it is activated by a non zero configuration on its XOP connection field, (bits 28, 29) Data transfer is then executed at the XOP speed (50 ns/16 bit data).

Possible Fastbus actions are:

Arbitrate, Do primary address cycle, Do secondary address cycle, Do data cycle, Do release AS/AK, Do release master ship.

They can be activated one by one or in the same XOP instruction and then shared by the sequencer.

Because Fastbus and XOP CPU are running at the same speed and simultaneously, DMA mode is not implemented, i.e. each transfer is to be programmed.

The error checking is managed via the control and the status registers. (Cf. table 2).

**PROGRAMMATION OF THE INTERFACE**

In the XOP microinstruction, the programmation of fastbus is independent of the other XOP fields.

In the example given, the programmation of the non fastbus XOP field will be ignored to simplify the corresponding instructions. 

**ARBIRTRATION**: the "Do arbitrate" order can be given at any time, if the sequencer is not busy.
If Master is enabled and Running (CSR 0) and Arbitration not inhibited (CSR 8, A I), the arbitration sequence is started.

XOP Program : Do Arbitrate, hold % wait to the end of ARB

JMP Error if FBFLAG Set

Note that as arbitrate is a "slow" fastbus action, it is recommended to execute it in the asynchronous mode (no hold).

**PRIMARY ADDRESS CONNECTION**

This sequence is started by the corresponding bit in the sequencer.

If it is possible (mastership three and primary address connection false), it generates the effective sequence at the address contained in the primary address registers FBPAH and FBPAL.

XOP PROGRAM IS :

LD, FBPAH load primary address registers (high and low)
LD, FBPAL
DO PAC, HOLD Do primary address cycle
CJMP , FBFLAG Jump error if fastbus flag set.

As the fastbus flag test can be done in parallel with the next instruction, the execution time can be estimated at 250 ns.

As the arbitrate cycle and the memory management of XOP are independent they can be executed at the same time. For example: a pipelined concatenation of the two previous examples can be executed in only 3 XOP instructions.

**SECONDARY ADDRESS CYCLE**

If possible, this sequence generates the corresponding fastbus signals according to the contents of the secondary address registers FBSAH - FB SAL. An error generates fastbus flag.
Possible errors are:

- No primary address connection,
- time-out,
- parity error,
- S.S. response ≠ 0.

In case of error the corresponding bits are set in the status register FBS.

The programming sequence is:

- Load secondary address registers
- Start the cycle (DO SAC) with hold
- Test the fastbus flag.

This sequence can be pipelined with the two sequences seen above to spend only one instruction more.

**DATA CYCLE**

The sequence is similar.

The data registers are to be loaded before a write operation or read after a read cycle.

A sequence error, a time-out, a parity error, or an S.S. response ≠ 0 will set the flag and the corresponding status.

All these sequences can be started at the same time. In this case only one hold is necessary to resynchronize the interface and all the parameters can be fetched in XOP memory during the arbitration and the other fastbus sequences.

In an example of a random data read we can perform the complete program in 4 XOP instructions.

This program includes:

- The loading of the four address registers with value fetched in XOP data memory,
- The execution of arbitration - primary address cycle secondary address cycle - data cycle,
- The storage of the 32 bits data read,
- The check of error.

The corresponding execution time can be estimated at only 200 ns more than the fastbus execution time itself.

In block transfer mode the software loop can be reduce to one single instruction including the two memory cycle to load or store the 32 bits data and the word count management.

The execution time is then reduced up to 150 ns/data.

In pipe line mode the transfer speed and the word count are both managed by hardware.

These parameters are programmed via two registers from 100 ns to 800 ns for the data rate, up to 64 K for the word count register. In this way the maximal fastbus speed is possible, only depending on the slave speed.

In the L3 experiment this interface will be used with the multiport multievent buffer designed at L.A.P.P. (3).

This slave module uses the fastbus state data transfer protocol chip "DATPRO" designed at L.A.P.P. (4) and now running.

With this fast coupler associated with fast ECL memories we hope to obtain transfer speed in pipe line mode up to 125 ns/word.
### MICRO INSTRUCTION XOP

#### Fastbus Instruction

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Ms code : Primary address</td>
</tr>
<tr>
<td>3</td>
<td>Ms code : Data cycle</td>
</tr>
<tr>
<td>6</td>
<td>R/W Secondary address cycle</td>
</tr>
<tr>
<td>8</td>
<td>Hold</td>
</tr>
<tr>
<td>9</td>
<td>Do arbitrate</td>
</tr>
<tr>
<td>10</td>
<td>Do primary address</td>
</tr>
<tr>
<td>11</td>
<td>Do secondary address</td>
</tr>
<tr>
<td>12</td>
<td>Do data</td>
</tr>
<tr>
<td>13</td>
<td>Do release AS/AK</td>
</tr>
<tr>
<td>14</td>
<td>Release Mastership code</td>
</tr>
<tr>
<td>16</td>
<td><strong>Source destination register--busses</strong></td>
</tr>
<tr>
<td>17</td>
<td>Register code during first 50 ns</td>
</tr>
<tr>
<td>20</td>
<td>Register code during past 50 ns</td>
</tr>
<tr>
<td>24</td>
<td>R/W first 50 ns</td>
</tr>
<tr>
<td>26</td>
<td>Bus code first 50 ns</td>
</tr>
<tr>
<td>28</td>
<td>Bus code past 50 ns</td>
</tr>
<tr>
<td>30</td>
<td><strong>Table 1</strong></td>
</tr>
</tbody>
</table>

- **Table 1:***
  - 00: No connection
  - 01: RD bus
  - 10: WR bus
  - 11: No connection
### XOP Status Register

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SS code or arbitration status</td>
</tr>
<tr>
<td>1</td>
<td>or sequence status</td>
</tr>
<tr>
<td>2</td>
<td>Sequence error</td>
</tr>
<tr>
<td>3</td>
<td>Time out error</td>
</tr>
<tr>
<td>4</td>
<td>Time out wait state</td>
</tr>
<tr>
<td>5</td>
<td>Parity error</td>
</tr>
<tr>
<td>6</td>
<td>Sequencer state</td>
</tr>
<tr>
<td>7</td>
<td>SR</td>
</tr>
<tr>
<td>8</td>
<td>RB</td>
</tr>
<tr>
<td>9</td>
<td>BUSY</td>
</tr>
<tr>
<td>10</td>
<td>ERROR</td>
</tr>
</tbody>
</table>

### XOP Control Register

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Master 1/2</td>
</tr>
<tr>
<td>1</td>
<td>Reset Master Interface</td>
</tr>
<tr>
<td>2</td>
<td>Reset Bus</td>
</tr>
<tr>
<td>3</td>
<td>Parity enable</td>
</tr>
<tr>
<td>4</td>
<td>non stop on parity error on data cycle</td>
</tr>
<tr>
<td>5</td>
<td>non stop if ss=2 on data cycle</td>
</tr>
<tr>
<td>6</td>
<td>&quot; &quot; ss=3 &quot; &quot;</td>
</tr>
<tr>
<td>7</td>
<td>&quot; &quot; ss=6 &quot; &quot;</td>
</tr>
<tr>
<td>8</td>
<td>&quot; &quot; ss=7 &quot; &quot;</td>
</tr>
<tr>
<td>9</td>
<td>Set EG</td>
</tr>
</tbody>
</table>

### PIPELINE Counter Register

16 bits $\Rightarrow$ 65 K words

### PIPELINE Transfer Speed Register

5 bits $\Rightarrow$ 100 ns to 800 ns in 25 ns steps.

**TABLE 2**
REFERENCES

1 - Tor LINGJAERDE, C.LJUSLIN - XOP, main documentation internal report - 1983,

2 - J.LECOQ, M.MOYNOT, G.PERROT - XOP - Fastbus master interface F 682 C internal report - 1985,

3 - J.LECOQ, M.MOYNOT, G.PERROT - Multiport Multievent Buffer - F 682 B internal report - 1985,

The Readout Controller of the Aleph Time Projection Chamber

S.R. Amendolia\textsuperscript{1)}, F. Fidecaro\textsuperscript{1)}, S. Galeotti\textsuperscript{1)}, B. Läfstedt\textsuperscript{2)}, A. Lusiani\textsuperscript{1)}, P. S. Marrocchesi\textsuperscript{1)} and D. Passuello\textsuperscript{1)}

Geneva, 26 September 1985

Abstract: The readout system of the Aleph TPC is described. Each of the 72 processors has to perform data formatting and reduction, chamber monitoring and calibration, as essential tasks to ensure the quality of the data coming from the detector. The implementation of the processor as an intelligent Fastbus module based on the Motorola 68020 microprocessor is described. Solutions for Fastbus interfacing and for data processing are discussed. Operational problems raised by the complexity of the system are also briefly discussed.

1. Introduction

The Aleph Time Projection Chamber is an imaging detector providing 3-dimensional position measurements of particle tracks. The information is obtained by letting the ionization electrons drift towards the chamber endplates, where these are detected by a system of 6,000 proportional sense wires and 44,000 cathode pads. The $r$-$\phi$ measurement is obtained by analyzing the relative pulse height on neighbouring pads, while the position along the drift direction is obtained from the time of the pulse.

Each analog channel is connected to its own circuit of fast sampling electronics, so that the 44 $\mu$s long signal is digitized into about 500 8-bit samples. This result in 25 Mbytes of raw data being stored in the read-out system at each event. The suppression of non-significant data is achieved in the digitizing module, by considering as valid data only those samples belonging to a succession of samples above a given threshold. The useful data are then formatted and processed by a dedicated intelligent Fastbus module, the Time Projection Processor, to be then assembled as part of a full Aleph event. This paper describes the functions this module has to perform, not only to read out the data but also to analyze these in order to monitor the performance of the detector and of its electronics.

2. Structure of the TPC readout

Each of the two endplates of the Aleph TPC is subdivided into 18 sectors (see fig. 1), to optimize the tracking efficiency still having an acceptable dead space. The sensitive part of each sector is made out of typically about 180 wires and 1500 pads. While the preamplifiers reside on the sector, the rest of the sector's electronics is housed in the Time Projection Digitizers (TPD), residing in 3 Fastbus crates and coping each one with 64 channels. For each sector there are 3 TPD for the wire channels and from 18 to 24 TPD for the pad channels, residing in 3 Fastbus crates (see fig. 2 for the structure of the TPC readout from the Fastbus point of view). These modules, together with the calibration system, are controlled in each sector by 2 Time Projection Processors, one for the wires, the other for the pads. The TPD are connected by cable segments to the TPC Event Builder, where the data from all sectors are assembled to build the TPC event. The TPD are Fastbus masters on the Fastbus crate segment,

\textsuperscript{1)} INFN, Dipartimento di Fisica, Scuola Normale Superiore, Pisa, Italy.
\textsuperscript{2)} CERN, EP Division, Geneva, Switzerland
the latter being extended to a set of 3 crates by means of Crate Clusterizing Cards [1]. The TPD are slaves on the same segment. On the other hand, the TPP are slaves on the Fastbus cable segment that communicates with the Event Builder.

When an event is accepted by the first level of the trigger logic, the sampling electronics records the pulses from the channels. This is foreseen to happen at a rate of at most 500 Hz. The second level trigger (at a rate of at most 10 Hz) enables then the readout of these data. As a first step, a simple zero-suppression procedure is performed inside each TPD. The output of this procedure is a collection of pulses, characterized by a channel number, a beginning time and a pulse length, together with the relevant samples. The data are collected by the TPP, where they receive appropriate treatment, as described in the next chapter. Data from the TPP are then collected by the TPC event builder.

The typical data volume from the TPC for a $Z^0$ event amounts to 60 kbytes, 80 % of these coming from the wires and only 20 % from the pads. These data are not uniformly distributed over all sectors: on average a sector contributes with 1.8 kbytes, but it can have up to 20'000 good samples. These fluctuations are dealt with by a scheme that includes 4 front end buffers in the TPD and 2 output buffer in the TPP. The effectiveness of this setup to reduce dead time has been studied by simulating the whole behaviour of the whole Aleph read out system [2].

3. Operations to be performed by the TPP

3.1 Operations during read-out

The minimal set of operations to be performed under TPP control during data acquisition are:

- data read-out from TPD modules.
- data formatting

The first operation consists in performing Fastbus transactions with the concerned TPD, then to store the data inside the TPP memory in order to avoid unnecessary memory-to-memory copies. Block identifiers and wordcounts have to be added, to build the required data structure and to allow subsequently an efficient processing. In addition, channel numbers become changed to the detector numbering scheme, the latter being closely related to the geometrical position inside the chamber.

3.2 Beyond read-out

An online data reduction can be performed safely on the wire data, which contribute to most of the data volume, but are carrying information less critical than the one coming from the pads. The reduction consists in replacing the samples by a pulse height and time computed with a simple algorithm. The expected volume of data from the TPC after reduction is about 60 % of the original one. This reduction has to be performed at a speed of 4 $\mu$s/sample, requiring therefore specialized hardware.

If allowed by the trigger rate, the installed computing power can be used to perform parallel tasks that do not require communication between processors. An example of this is the recognition of pulses that are at the same time on neighbouring pads and the subsequent computation of the position. Another possibility is the recovery of samples near a good pulse that were lost by the zero-suppression procedure. If performed, these tasks would ease the amount of offline computing needed to reconstruct a TPC event.
3.3 Other TPP tasks

A number of tasks have to be performed asynchronously and sometimes concurrently to the data acquisition. These include the management of the system itself, the calibration of the analog chain and monitoring the TPC.

The purpose of the electronics calibration procedure is to adjust all channel's gains within 1 % before data taking, so that the data do not require further offline corrections. This calibration of the TPC is expected to take only a few minutes, using the parallelism achieved by the TPP.

To monitor the TPC and its readout system, several programs will run inside the TPP. As a basic requirement, checks on the read-out functionality will be provided, flagging time-outs and errors and ensuring system recovery in case of faults. Statistical information with and without distinction of channel can be collected to monitor the performance of the detector with respect to noise, discharges, etc. These distributions on a channel by channel basis, together with hit frequencies will then help detecting faulty channels. These tasks will be performed during TPP idle time, keeping an event in memory until it is fully analyzed. Results will then be sent to the host computer, to be analyzed and archived by an appropriate software.

4. TPP hardware

The functional block structure of the TPP is shown in fig. 3. It consists of a CPU and a bus which sees memory and peripherals, the most important among them being the Fastbus coprocessor, the slave port on the cable segment and the local area network interface.

4.1 CPU and bus

The CPU is a Motorola 68020. The reasons for the choice of the 68000 family are mainly the flexibility achieved by the powerful set of instructions and the availability of software both inside and outside CERN. In addition it is the supported family of microprocessors inside CERN. The choice of the 68020 is then due to the fact that it supports external 32-bit architecture, the one of our data acquisition and offline computers, that it has a high instruction execution speed, adequate to our data volume, and also because it features the concept of co-processor, that allows an easy interface to special hardware (like FB or signal processing unit) or to a floating point unit.

4.2 Memories

The memory accessible from the CPU is subdivided as follows: there is an EPROM to contain startup and some system code, the main RAM where programs reside, the output buffers that can also be accessed from the slave port and a section of the main RAM which allows duplication at writing time of an output buffer.

The ROM is 64 kbytes, and it contains the resident monitor. The RAM is 512 kbytes, with an access time of 55 ns, requiring thus no wait states.

The two output buffers have a size of 32 kbytes each. However only one of these buffers is accessible from the CPU, the other appearing instead as Fastbus data space on the cable segment. The buffers are swapped by a switch that can be set by writing in a special location of the 68020 address space. The monitoring buffer, also with a size of 32 kbyte, can contain a copy of an event, to be analyzed during between the arrival of events. To avoid active copying by the CPU, the monitoring buffer is written at each time the bus makes a write cycle addressing the output buffer. This is enabled by a switch at some place in the 68000 address space.
4.3 The slave port

The slave port allows access and control of the TPP via FB through the CSR space and allows the transfer of the TPC data that are present in the Data space. In addition, it must be able to communicate with the event builder, by asserting a Service Request and by responding to FB operations, either while being addressed and during Broadcasts operations.

4.4 Fastbus master port

The cluster of crates containing the TPD is not connected via an SI to the Aleph Fastbus configuration. Since there are only 2 TPP that can act as master on the extended backplane, and they don’t need to communicate, the TPP has only a master port on the backplane, without slave. Thus, the master port on the backplane is in normal operation the only way to access the TPD’s, for read-out, calibration, testing, debugging. An analysis of the operations involved in these segments shows that there is no need of implementing “elementary primitive” Fastbus commands: only complete operations are at disposal of the user (e.g. no primary address cycle alone, without secondary address and data cycles). Implementing separately, for higher flexibility, the various components of a FB instruction would require a more complex machine instruction and a greater number of hardware components. On the other hand, the TPD transmit part of its data in 8-bit format, without packing these in 32-bit words. Thus the TPP needs to implement a block transfer capability for 8-bit or 32-bit data read/write. The 8-bit data will be packed into 32-bit longwords in the TPD before writing into memory.

The following FB operations are available at the master FB port:

i. random read/write - 32 bit options:
   • keep bus at end of transaction
   • release bus

ii. block transfer read/write options:
   • DMA or NON DMA mode
   • 8 or 32 bit field

During block transfer read the following options are available:

   • START : starts block xfer read on memory longword boundary
   • CONTINUE : resumes block xfer read from previous address (inside a longword if necessary)

For random r/w, the primary address (PA), the secondary address (SA) and the datum (D) can be given in the instruction stream, or can be found in a block in memory addressed by the instruction; for block transfers, a block is always created in memory.

In case of a random r/w FB command issued by the VAX and to be executed on a TPD (as a partial software emulation of an SI), the following steps are followed:

   • a block with PA,SA,D is written to TPP’s CSR space
   • the 68020 is appropriately interrupted
   • a command is issued from the 68020 to the FB coprocessor to execute the “list” of PA,SA,D described in the block.
An alternative way, since the TPP slave port can access the whole address space, is that any master, emulating the 68020 coprocessor protocol (e.g. the VAX), can access directly the master port in the TPP.

To implement the communication between the 68020 and the FB coprocessor, only General Type instructions are used [3]. One 16-bit Command Word is enough to accommodate all the options (see fig. 4). This Command Word is fetched by the coprocessor immediately after the General Type (GT) instruction. The 68020 waits for the end of the Fastbus transaction before executing the next instruction. There are therefore less problems in case of errors in a multitasking environment.

The events that cause the coprocessor to generate a trap are the following:

- invalid command word
- parity error
- timeout on AR, AS, DS
- SS not 0 during random R/W
- SS not 0 or 2 during block transfer

A 32-bit register is available, reporting the state of the Fastbus and of the interface at the moment of the error.

The implementation is made with a set of 3 finite state machines, which transitions are caused by the various signals of the FB and 68020 coprocessor protocols. All signals are sampled and latched by the 68020 clock. With a 60 ns clock for the CPU, the coprocessor can transfer a 32-bit word in block mode and DMA every 90 ns, if the slave responds to DS in less than 50 ns, and every 120 ns if the slave responds in 50—80 ns.

4.5 Auxiliary Arithmetic Unit

This other 68020 coprocessor will perform the wire data reduction. It will consists of a dedicated signal processing unit, to perform the reduction algorithms at the required speed. The design of this unit will start at the end of this year.

4.6 Interface to the Local Area Network

The foreseen LAN is, at the moment, Cheapermet. An interface, possibly run by a 6809 microprocessor, is required to transfer data from the network to the main memory. This interface should be able to recognize specific messages, like "reset TPP", to be used in case the FB hangs up. The design of this interface, possibly using existing projects, will also start at the end of this year.

5. Software Considerations

5.1 TPP General Software

There are many programs expected to run concurrently in the TPP. Together with a high priority data acquisition and formatting process, there will be one or more monitoring programs, that will run during idle time. All these programs require to communicate with the TPC VAX, for error logging and to store the accumulated statistics for further processing. Dialogue with the host is also required for downloading and to specify what the TPP should do. We expect that these programs will run under supervision from a real time kernel (RMS68K is for the moment our candidate) to exploit priority schemes and resource allocations provided by such a software.
5.2 TPP Fastbus software

To allow migration of programs, the standard CERN Fastbus Fortran calls are implemented, as an FBPACK, written in assembly language. These routines fill the extension word of the FB coprocessor instruction according to the values of their parameters and then execute it. The exception handling routine returns an hardware and software status stored in word 2 of the STATUS 1*4 array, and an "F-code" status stored in word 9, to give access to the state of the finite machines and of the Fastbus signals at the moment of the error.

In addition to the standard calls there is:

• a KEEP BUS option in single listener routines
• a BLOCK TRANSFER CONTINUE option
• a RELEASE BUS option when secondary address is transferred

5.3 VAX software

The TPC VAX will manage the whole read-out system, by down-loading the programs that run in the TPP, triggering checks and calibrations, collecting results and preparing the system for data acquisition. There will be various running modes of the system (TPC normal data acquisition in experiment of stand-alone, self-triggering, calibrations, ...). In particular, there will be on the VAX a management program that can access all TPPs, down-loading the appropriate code for each running condition and ensuring to the operator access to the single TPP in the debugging phase.

This requires a careful design of the dialogue procedures between the host computer and the TPP and also of the way the system is seen by the user. The latter point will determine the manageability of the TPC read-out in absence of the experts.

The collection and analysis of the calibration constants and of the results of monitoring will give the ultimate checks of the performances of the system. Due to the large number of channels, at least semi-automatic analysis should be provided in order to detect faulty channels. This includes availability of calibration constants in an interactive database management system, statistical analysis to pick up channels out of tolerances, histogram comparison facilities to evidentiate breaking or smooth deterioration of elements, etc.

5.4 Production of the TPP software

The programs will be adapted from running programs on the TPC VAX where we expect to have a very similar environment (for monitoring tasks) to the one in the TPP. These programs will then migrate most of the calibration and test software will migrate from the TPC VAX to the TPP as we expect not to switch from FORTRAN 77 to another language (Pascal or 68020 Assembler). We expect to write only the data acquisition part of the software (including the drivers for the peripherals) in 68000 assembler, relying on the goodness of the code produced by the available compilers for the remaining software. This software is available (real time KERNEL, VAX cross compilers, linkers and pushers) and is being adapted to CERN needs by current microprocessor users. We expect this software to be sufficiently performant at the time the first TPP is tested.

The Auxiliary Arithmetic Unit, if programmable, requires an ad-hoc assembler that can generate the microcode. The design of the microcode and of the assembler goes in parallel with the design of the unit. The task of producing a compiler is then a well mastered exercise.
References


Fig.1 Aleph TPC endplate
Fig. 2: TPC read-out scheme
Fig. 3: TPP functional structure
Fig. 4: Extension word to the General Type Coprocessor Instruction in the TPP Fastbus Master Port
The ALEPH Event Builder
A Fastbus based CPU

R. Benetta, A. Marchioro, G. McPherson, W. von Rüden
CERN/EF

Introduction

The ALEPH detector consists of a number of sub-detectors such as the Inner Tracking Chamber, the Time Projection Chamber, the Calorimeters etc., each having several Fastbus crates filled with digitising electronics. One or more such Fastbus crates are controlled by a Readout Controller (ROC) and several ROCs are under the supervision of an Event Builder (EB). The task of the EB is to collect and format the data from these ROCs and to pass them either to the Main Event Builder and/or to the sub-detector computer (Ref. 1). For simple sub-detectors, like the Inner Tracking Chamber or the Trigger, one EB will combine both the ROC and the EB functions.

General considerations

The functions required for the ALEPH data acquisition system led us to the design of a new Fastbus master, known as the ALEPH Event Builder, which implements the following features:

- a powerful general purpose CPU for data handling, offering the possibility to run large programs written in a high level language;

- a fast interface to the data acquisition bus.

When designing the Event Builder for the ALEPH experiment we believed that these two problems must be given equal weight. In particular, the second requirement must be satisfied by having a machine executing Fastbus bus instructions in a way which completely integrates the Fastbus environment and the data acquisition software in a manner transparent to the programmer.

Until now, list processing has been used as a means of reducing the I/O overhead introduced by complex operating systems. As our front-end microprocessors do not need to run a 'heavy' system, the arguments for list processing are no longer valid if one can efficiently implement access to Fastbus. A good example is the SLAC Scanner Processor (Ref. 2), which implements a new kind of Fastbus CPU. However, this machine is too limited for our needs.

We have chosen the Motorola MC68020 microprocessor because it satisfies our needs regarding computing power (Ref. 3) and it offers the unique potential for the designer to add his own machine instructions through special coprocessors. In our case, a coprocessor based on a micro-programmed sequencer has been designed to handle the complex processor-coprocessor protocol and Fastbus access.
The block diagram of the Event Builder can be seen in Fig. 1. A dual bus structure has been adopted to allow simultaneous coprocessor access to the event memories and MC68020 access to program store and I/O. The LAN interface provides an independent channel to the module and will serve as the control and data path in systems where no direct Fastbus interface is available on the host computer. Here we refer to situations where the home laboratory computer has no connection to CAMAC or Fastbus, since it serves as the computer centre, where such connections are not acceptable.

Fig. 1
Software aspects

The MC68020 and sequencer configuration is capable of executing all "standard" Fastbus cycles such as single word moves, block transfers, broadcasting and even a few more (like indivisible read-modify-write cycles) as native MC68020 assembly instructions, making the programming of such a machine extremely simple.

As can be seen from the software model in Fig. 2, the machine is supplied with a set of new registers which, from the programmers point of view, are seen as additional MC68020 registers.

![Fig. 2]

A Primary Address Register (PAR) and a Secondary Address Register (SAR) contain the Fastbus addresses of the current instruction. Similarly a Memory Address Register (MAR) holds the address of the Event Memory location during a block move and the Block Length Register (BLR) holds the length (in bytes) of the transfer. The Fastbus control functions normally scattered in various CSR registers (such as the arbitration level, parity enable etc.) are grouped in a general CONTROL Register accessible via the FBMOVE instructions. The STATUS Register contains all bits of information (like SS codes, bus error flags etc.) related to the status of the coprocessor unit and Fastbus. Finally the IADDR Register contains a copy of the MC68020 program counter for the last executed Fastbus instruction. This is particularly useful for program debugging and error recovery, since block transfers run in parallel to MC68020 program execution.

To illustrate the features of this machine let us take an example out of a Fortran program:

CALL FBSWRD ( Prim_Add, Sec_Add, Data)
This is a single word read (Data) from data space in module Prim_Add and from internal address Sec_Add.

It can be translated into a few assembly language instructions:

\[
\begin{align*}
\text{MOVEA.L} & \quad \text{Prim}\_\text{Add}, \text{A0} \\
\text{MOVEA.L} & \quad \text{Sec}\_\text{Add}, \text{A1} \\
\text{FBSWRD} & \quad \text{A0, A1, D0} \\
\text{MOVE.L} & \quad \text{D0, Data}
\end{align*}
\]

As shown, a Fastbus word can be transferred to or from a data register in a single MC68020 instruction. Furthermore, the coprocessor concept offers very elegant and simple ways for handling errors, if its capabilities are fully exploited. As an example, assume that an error occurs in the above call and the module responds with an SS=6. As single word transfers are synchronized (our choice), the MC68020 waits for completion of the Fastbus cycle. Therefore, the coprocessor can return an exception code to the MC68020 instead of the data word, which will make the processor behave as if a machine trap had occurred.

In general, error conditions can be trapped on any MC68020 exception vector. In our case, we decided to use only user vectors. The Fastbus error handler is then under complete software control and can be written to suit a particular system in the same fashion as the MC68020 exception handler.

Fastbus block moves run directly between the Event Memory and the Fastbus master port. Once these instructions are started the MC68020 continues execution of its program without waiting for the end of the block transfer, which is possible due to the double bus architecture of the machine. However, the MC68020 can gain access to Fastbus only after completion of the block transfer. An option is foreseen by which the coprocessor can interrupt the MC68020 after completion of a block transfer; this capability is particularly useful in a multi-tasking environment.

In principle the sequencer can be micro-programmed to accept additional addressing modes for the Fastbus instructions. The addressing modes for the instructions we have chosen are believed to offer the best compromise between programming flexibility and execution speed (a manual describing our instructions is in preparation).

The CERN MC68020 cross-assembler has been updated to recognize the newly defined instructions while, at the same time, a popular Fortran compiler widely used at CERN and elsewhere has been modified to treat Fastbus calls in the same manner shown above (as in-line macros). The Event Builder can therefore be easily programmed in Fortran without the need for any Fastbus library.
Event Builder hardware

To make efficient use of the coprocessor concept and also have fast Fastbus cycles, we have chosen to design the coprocessor part using a very fast ECL sequencer. A detailed diagram of the coprocessor unit is shown below.
The coprocessor consists of a microcoded machine based on an MC10904 sequencer running at a cycle time of 25 ns with a 96 bit wide microprogram memory. This machine can execute 'test and branch' instructions within the same cycle. It controls Fastbus normally as a master, but it is also able to receive interrupt messages as a slave, which then will be passed to the MC68020. The sequencer also controls the DMA circuitry of the Event Memory for block transfer instructions.

The coprocessor machine is micro-programmed with the aid of a commercially available micro-assembler. The microstore memory is fast ECL Ram (7 ns access time) and is loaded directly from the MC68020 at start up. This permits easy changes of the micro-program software at the development stage and will ultimately allow several suites of microprograms, such as diagnostics for Fastbus, special Fastbus instructions and normal data acquisition instructions to co-exist. The sequencer also performs checks on each instruction coming from the MC68020 to ensure their integrity.

The Fastbus coprocessor and the MC68020 share the memory bus when addressing the Event Memory; access is controlled by a bus arbiter giving priority to the microprocessor in cases of simultaneous access. Typical memory write cycles from the coprocessor DMA circuitry are of the order of 75 ns plus the overhead of arbitration (~ 20 ns). This time is completely overlapped, on a word-by-word basis, with the Fastbus block transfer time. In principle, therefore, block transfer rates of the order of 100 ns per word can be achieved.

Present status of the Event Builder

We have constructed a wire-wrap prototype unit which includes the MC68020, the floating point coprocessor (MC68881), 128Kb of Ram and 512Kb of Eprom for program memory and I/O for terminal and host communication. The Fastbus coprocessor is operational and most of the required instructions are implemented. A dual ported Event Memory module of 128 Kb has also been constructed and is operating correctly.

An Ethernet interface and the second Event Memory (spy) are in preparation.

ACKNOWLEDGMENTS

We would like to thank Hans von der Schmitt for the implementation of the specific Event Builder Fastbus calls in his Fortran Compiler. Horst von Eicken kindly provided the necessary extensions in the cross assembler.
REFERENCES
1. G. McPherson et al., The ALEPH Event Builder, Conference on Computing in High Energy Physics, Amsterdam, June 1985
2. H. Brafman et al., The SLAC Scanner Processor: a Fastbus module for data collection and processing, SLAC-PUB-3471, Oct. 1984
3. A. Marchioro, H. von der Schmitt, Benchmarking a 68020+68881 system, to be published.
General Purpose Masters GPMs in the DELPHI Online System

H. Muller 1)

Abstract

Important functional entities of the DELPHI readout system, such as the event- and trigger supervisors, the 3rd level trigger and multi-event buffer stages can be implemented using single-widt master/slave modules (GPMs). They provide the programmability of any Fastbus operation by direct execution of 32 bit 68000 instructions in Fastbus. Other features like high-speed Fastbus I/O via the master or the slave ports, mutual interrupts or source code exchange via Fastbus, and self-addressing capability allow the GPM to be used in different areas of the online system for either data-flow control functions, programmable decisions or diagnostic functions. The need to integrate computing power into the DELPHI online system for parallel treatment of subdetector data can be accomplished by the GPM with an option to boost their processing power by using 68020/68881 processors, or an optional DMA controller to gain time for internal transfers.

1) CERN, CH-1211 Geneva 23, Switzerland, Division EP
1. THE DELPHI READOUT SYSTEM

The DELPHI Fastbus online architecture [1] consists of Fastbus readout branches (order of 10) which are connected to a common multi-event buffer (fig. 1). Each branch can be individually controlled via μVAX-type equipment computers. Event- and trigger-supervisor functions need to be locally integrated in each branch, and synchronised with central supervisors in the main control crate. [2] During the full readout, the contents of the front-end buffers are transferred in parallel into the multi-event buffers. This process takes only place after a 2nd level trigger and is controlled by local event supervisors. The event-data blocklets can be sent in selected order to either the common buffer in the multievent crate, or to the equipment—computer crate’s spy buffer. In data-taking periods, the DSM [3] buffer in the equipment―crate can, on demand, get event data copies of its detector part. The local event supervision is performed by processors, which define the readout-list of the fast block-mover [5] BM.

The full event which is transferred in parallel to DSM buffers [6] can be accessed by processors to perform the 3rd level trigger.

Supervisory masters are connected to the control segment for communication with the central supervisors. The masters can also communicate via Fastbus interprocessor messages amongst themselves, to resolve correlations of the trigger, or to control subordinated masters.

The 3rd level trigger decision needs to be ready in 20-30 ms after the start of the full readout. The 3rd level processes can start after a selected part of the event has been sent under control of the event supervisors and the block movers. For this purpose, the 3rd level processors can be interrupted to start processing when the selective readout part of the transfer is complete whilst the rest of the event transfer continues. A rejection is performed by resetting the pointers in the buffers. The events residing in the output buffer of the central crate are read out into one of the 3081 emulators under control of a supervisory master and a block-mover type I/O interface. A dedicated master in the output segment can accumulate diagnostic or event-related information to present it to an online graphic display device.

The programmable Fastbus masters described in this system need to have the following attributes:

- Programmability in a standard high level language (F77, C, PILS)
- I/O efficiency for both passive and active data movements
- Interprocessor message and interrupt support
- the computing power of at least a 68000 microprocessor
- sufficient program and data capacity (min 1/2 Mbyte)
- source code distribution via Fastbus (Slave and Master)
- interface(s) to special logic (i.e. NIM)
- local memory extension bus
- Possibility of autonomous Fastbus I/O in parallel with processing

The logical functions as described above can be performed by GPM processors. The GPM [4] can map the internal addresses of DSM buffers into its address range, via the auxiliary bus connection. In several cases, it is possible to share functions, in particular if they are not exclusive in time. For this purpose, multitasking can be used on individual processors.
2. GPM ARCHITECTURE

The GPM processor has been designed in the first place as programmable Fastbus master for general purpose applications. This fact allows other online architectures than DELPHI to use GPMs. [7] In particular, the use of the GPM as processor component in the VIRTUS multiprocessor project [8] has introduced a set of features which are required for multiprocessor systems. The GPM's main characteristics can be summarised as follows:

• **Modularity and general purpose:** The GPM consists of 16 modular functions (and two options). The slave and the master-ports are on the same (Crate-)segment and the slave port provides access to 6 CSR registers and a 32 Kbyte, two port data buffer. The Slave port supports a large number of primary address protocols including logical addressing, class-N broadcasts and T-pin broadcasts. The master-port is general purpose in the sense that it allows to generate any standard Fastbus operation either via list processing, or direct calls into 68000 language or execution of autonomous block transfers.

• **68000 or 68020 operation-code execution in Fastbus:** The master-port can be directly driven by 32 bit instructions of the microprocessor. Instructions which are directly executed in Fastbus can be mixed with normal 68000 instructions.

• **Simultaneous processing and buffered I/O operations:** High speed, autonomous block transfers can be executed using the data-space memory as I/O buffer. Both slave input (passive) or master block transfers (active) use the 32 kbyte buffer independently of the processor's activity. A DMA controller-type NTA with several control and stop modes defines the word-counter- or end-address conditions.

• **Self address capability:** The master port can address it's own slave port via Fastbus providing a stand-alone host/target diagnostic environment with only one Fastbus module.

• **Fastbus I/O buffers:** The 68000 can access the slave's or master's 32 Kbyte I/O buffer via an internal port. The access can be competitive (alternating access) or mutually exclusive via a programmable busy response or a CSR(0) lock-bit. Additionally, the slave can also access the 68000 address space in blocks of 64 Kbyte.

• **Source code exchange via Fastbus:** The slave can bypass it's data buffer and read or write directly into the 68000 address space. This function allows source-code distribution via Fastbus (single or multiple destinations) and is enabled via a CSR(0) user bit. The possibility exists to send executable code (i.e. assembly-lists) to a GPM and start it's execution via an interprocessor interrupt using the GPM's CSR(1) register.

• **Registers for set-up and information:** Dedicated registers allow to configure both the slave's and the master's static configuration (software flags, the timeout settings, or interrupt masks, etc). The Fastbus status is always readable via a register with bit correspondences of Fastbus signals. For example, the GPM can read it's own geographical position from the backplane.

• **Interprocessor and event interrupts:** Event- and protection interrupts such as service request are handled by separate interrupt controllers to allow different priority scheduling and individual masking. The CSR(1) provides an interprocessor interrupt mechanism which is based on exchange of Fastbus interrupt vectors within normal write-cycles.

• **Processor subsystem:** A 68000-family microprocessor can be used, optionally a 68020/68881 processor combination, and/or a 68440 DMA controller is available. There is room for 1/2 Mbyte of RAM and 1/4 Mbyte of ROM. The Fastbus I/O buffer is mapped into the processor's address space. Serial I/O is provided on two RS232 channels which support also multidrop configurations. Parallel I/O is provided via general purpose interface and timer chips. A NIM pulse interface can be directly addressed to read patterns, generate pulses or levels. Event type interrupts can be generated via NIM inputs.

• **Local memory extension:** The Fastbus auxiliary connector is used for memory extension via a cable memory-extension bus defined for the GPM and the DSM. This provides the possibility to map independently filled Fastbus I/O buffer modules into the processor's 16 Mbyte address range.
3. 68K OPERATION CODE TO DRIVE FASTBUS

The GPM can directly execute 68000 or 68020 operations code in Fastbus. For this purpose there are twice 8 address windows in the 68000 address space which correspond to twice 8 MS-codes for Fastbus (primary address and data-cycles). The Fastbus arbitration is automatically embedded in primary address cycles. 32 bit cycles which use one of these address-windows in their operation fields, automatically access Fastbus with proper MS code and handshake detection. SS-responses are vectored into different 68000 interrupts. Complicated Fastbus operations can be efficiently assembled by mapping Fastbus list-code [9] into corresponding 68000 operation code. This allows to generate executable source code lists with the possibility to include user defined 68000-code in the list. For diagnostics, single stepping of Fastbus/68000 code list-execution is supported by the GPM's debugger software. Since Fastbus data cycles can be generated by any 32 bit instruction, Fastbus operations can be rationalized: An ADD instruction for example replaces a random read into a buffer, modification in the buffer and random write by an indivisible operation without buffering. As an example, a complete Fastbus operation can be defined in 2 to 3 lines of assembly code to arbitrate, address a slave, test a bit of the slave's NTA and disconnect:

MOVE.L PRIMAD, A-WINDOW arbitrate and primary address
BTST.L #BIT, SEC-WINDOW DISCON test the NTA-bit
CLR.B DISCON disconnect optionally

4. AUTONOMOUS, BUFFERED TRANSFERS

The I/O efficiency for online systems like DELPHI is mainly important for block-transfers, since these make up a large percentage of the bus occupancy. Autonomous transfers of data can be triggered via three 68000-addresses, BHW, BHR and BPW, after defining the pointers and/or wordcounts of the NTA/DMA controller. The CPU can continue to process data in parallel. By using the same buffer for slave input as for master output (with access possibility from the processor), no internal transfers to separate I/O buffers are required. A second, optional autonomous I/O mechanism transfers data between any addressable buffer and Fastbus or other addressable parts by using a 68K-amily DMA controller. In this case, parallelism of CPU activity and I/O is excluded.

5. PROCESSOR OPTIONS AND PERFORMANCE

The GPM can be equipped with 68000, 68010 or 68020/68881 processors. In the latter case, an additional plug-in board (DD5367) is required to replace the 68000 chip by a 68020/68881 processor combination. The standard GPM with 10 MHz 68000 chips can execute Fastbus operations at an intrinsic 1 to 2 μs Fastbus cycle time. These directly executed microprocessor operations can be mixed with autonomous block transfers of 10-20 Mbyte speed capacity. The use of the current 68020 option increases the computational power of the GPM by a factor 2.3 if it executes 68000 code. The use of the floating point coprocessor and the use of 68020-specific instructions allow for a further increase to achieve an average 2.5 MIPS [10] and 0.4 MFLOPS for the coprocessor. The Fastbus I/O speed is increased by 30 to 50 percent. By using increased clock rates (i.e. 16.6 MHz) and a 32 bit I/O adapter we expect to perform complete Fastbus operations (i.e. arbitration, primary address, secondary address, data cycle and disconnection) in less than two μs. By using autonomous block transfers, 32 Kbyte of data can be moved from the GPM to a Fastbus destination within an operation of 1.6 ms duration and with parallel CPU activity in other parts of the GPM's address range.

141
6. APPLICATION AREAS FOR THE GPM

Several areas of application can be identified, covering single user diagnostic test systems, control and interface functions as well as online processors and components for multiprocessor systems. Multi-tasking allows to share different types of functions on one physical GPM.

6.1 Stand-alone test system

Fastbus masters or slaves are subject to programmable tests. In the simplest case, the GPM's self-address capability allows to run self test via its home segment. A minimal configuration consists of a GPM and a Personal computer like the Apple Macintosh, or IBM PC. The PIls [11] language provides a convenient interpreter-environment to develop Fastbus test programs.

6.2 Diagnostic master

Segment-tests, access to backplane information for activity monitoring and event-interrupts to detect or monitor bus conditions like an idle, blocked bus or disrupted connections can be implemented with GPM masters. These diagnostic functions can be executed on one single GPM in a multi-tasking context with other functions. When detecting serious segment problems (via event interrupts), the GPM can take action as exception handler in which function it may reset its home segment and generate a messages to its next-priority supervisor, or by using its serial I/O port as message port, to a connected host.

6.3 Interface applications

The GPM's interface ports provide connection to parallel and asynchronous serial buses. The parallel interface is based on two programmable I/O chips which, via specific software drivers and external I/O registers, can provide a link to other buses like the Unibus or the SCSI-bus. The I/O rate for this connection can reach 4 Mbytes/s if the DMA-controller option is used in the pulsed mode.

6.4 Supervisor

As programmable part of decision logic in Fastbus, the GPM provides a large number of functions and interface connections required to build supervisors for timing and/or to control the transfer of data. A GPM event supervisor may transfer data from source to destination using its own I/O capabilities, or by controlling hardware block-movers for higher speed. GPM trigger supervisors require to be interfaced to fast timing logic and external synchronisation signals. Synchronised arbitration to prepare backplane access for Fastbus timing operations as well as T-pin polling for the ready-status of a segment are supervisor tasks in the DELPHI system.

6.5 Trigger processor

The processing power of 68000-family microprocessors which can be higher than the power of a VAX 780, can be very cheaply integrated into the online systems of modern experiments. The DELPHI experiment plans to use 68000-based Fastbus masters for third level triggering in the 20-30 ms time range. A number of parallel working processors are active during the readout time of the full event to improve the selectivity beyond that of the 2nd level and to utilize correlations between different detector parts. The option to use a 68020/68881 processor combination in the GPM increases significantly the GPM's computing power to more than a VAX 780 equivalent. The local bus connection to
a DSM Fastbus memory allows to access event data without Fastbus I/O overhead. Several DSMs can be mapped into the GPM's address space to group data from different parts of the detector into one logical buffer.

6.6 Multiprocessor systems

The main requirements for multiprocessor systems such as distributed arbitration with fairness protocol, good message mechanism, interprocessor interrupts and access functions are implemented in the GPM via normal Fastbus protocols. Farm-like multiprocessor arrangements like VIRTUS, make use of Fastbus enquiry protocols (sparse data-scan) on two different conditions and the GPM's two-port data buffer for independent I/O.

7. SOFTWARE FOR GPM PROCESSORS

The GPM software provides transportability on the user level, via compatibility with the general software framework for the 68000-family which exists at CERN for different physical host computers and interfaces to Fastbus. In view of applications outside CERN, also non-standard software (monitors, interpreters and multiprocessing kernels) has been used, or ported to the GPM. Software in four categories exists, or has been implemented on the GPM:

- **Debugging Monitor and Run-time system:** The MoniCa symbolic debugging monitor is the standard, minimal software which is distributed with the GPM (inEPROM). MoniCa provides compatibility with the CERN supported languages and libraries.
- **Cross-languages:** Fortran, Pascal, C, and Modula-2 can be cross-compiled on CERN computers and linked with the MoniCa run time libraries. The generated code is loaded, in S-record format, via the GPM's host port, and debugged with MoniCa.
- **Resident language:** PILS is available as EPROM-resident language on the GPM. The PILS interpreter allows calls to user procedures which are compatible with the Fortran calling conventions. For file storage, a host computer, typically a personal computer is required.
- **Libraries:** Fastbus, Multitasking, Floating point, user procedures etc. can be linked with cross software, or with PILS. Libraries can be burnt into EPROMs on the GPM.

7.1 The GPM-plus environment

Similar to the VALET test-system [12] approach, the GPM can be combined with a host, typically a personal computer, to generate an inexpensive, standard test environment (fig.2) with file storage. The PILS language, the MoniCa monitor, the Fastbus libraries and communication software, called BRIDGE have been implemented on a Macintosh/GPM combination, called the MAC-GPM. A similar environment can be established on IBM-PC, HP-9000/200 or the VAX. In a development phase, an auxiliary host was required to download cross assembled libraries, using the MoniCa download command. An optional EPROM programmer box (DELPHI), which can be connected to the GPM's parallel port is used for convenient development of a GPM-resident software configuration.
REFERENCES

[12] C.Parkman et al., VALET-PLUS overview, CERN-DD, OC-group-GA/Note 17
THE FASTBUS SEGMENT EXTENSION*

Donald R. Machen
Scientific Systems International**
Los Alamos, New Mexico USA

Summary

The Segment Extension is a recent development in FASTBUS hardware that will require changes in the controlling software now being written for physics data acquisition systems. Applications involving the Segment Extension include the logical expansion of the front-end instrumentation crates holding electronics for the hundreds of particle detector channels planned for LEP experiments at CERN, the SLC experiments at SLAC, and CDF experiments at FERMILAB. This paper discusses the characteristics of the Segment Extension and the FASTBUS system-software requirements and hardware requirements imposed by the device.

Introduction

The FASTBUS (Modular, High Speed Data Acquisition) system specification (1) for high energy physics and other data system applications defines a Segment Interconnect (2) that allows a user to expand a FASTBUS system beyond the 26-slot basic Segment. The development of a different technique for coupling FASTBUS Segments, called the Segment Extension (SE) concept (3), has recently been completed under contract to the U.S. Department of Energy. The SE allows the user to logically expand a single FASTBUS Segment to 7 physical Segments in a cost-effective way.

The new segment coupling technique will be of particular interest to FASTBUS system users faced with the task of instrumenting a large number of particle detector channels in the front end of the next generation of high energy physics experiments. The SE is considered less complex than the Segment Interconnect; however, the system software responsible for the front-end instrumentation in a large FASTBUS-implemented

*Work performed under contract to the U.S. Department of Energy, Division of Energy Research #DE-AC03-83ER80077
**Scientific Systems International
3491B Trinity Drive
Los Alamos, NM 87544 USA
(505) 662-6712
experiment will need to address the SE-specific requirements in a system configured with SEs.

The Segment Extension restricts the placement of Master devices to a Primary, or Master, Segment, but Slave devices may be placed in any physical Segment in the expansion. The technique uses the extended addressing bits AD<07:05> in addition to AD<04:00> during the Geographic Address Operation. Broadcast cases 3 and 3a also use AD<07:05> to direct the Sparse Data Scan Read Operation to the specific Segment in the expansion (4). Once modules in an expansion have been initialized with a Logical Address, etc., most FASTBUS operations proceed normally, albeit with additional delay and thus a longer cycle time.

System Configuration

A FASTBUS system expanded by Segment Extension devices contains one primary, or Master, Segment (with Master modules, SIs, etc.) that is equipped with normal Ancillary Logic (5). Expansion Segments may "star" from the Primary Segment, using one SE Bus per leg of the star, or they may proceed from the Primary Segment on a single SE Bus. Figure 1 shows a star configuration with two SE Bus cables and a total of 3 segments and the FASTBUS interface connected to the Primary Segment, while Figure 2 shows a single SE Bus configuration. A limit to the number of arms of a star-configured expansion is imposed by termination requirements. A set of rules for SE expansion configurations may be set down as follows:

1. One SE per Segment is required in an SE expansion.
2. An SE expansion contains only one set of ancillary logic modules (GAC, ATC) and the Segment equipped with the GAC/ATC is termed the Primary or Master Segment.
3. The Primary Segment may contain Master modules, Slave modules, or other controlling devices, such as a Segment Interconnect or front-end processor module.
4. Segments other than the Primary Segment must contain only Slave FASTBUS modules and must be terminated by "terminator-only" ancillary logic modules, capable of also supplying the Geographic Address voltage source.
5. Each Segment Extension module is assigned a unique address in the range <0-6> (i.e., GA + 32, 48, ... 192) from the Extended Address field defined in Section 4.3.2 of the FASTBUS Specification. Address <7> is available (i.e., GA +224), however, a restriction has been placed on this address owing to a possible conflict in addressing the GAC ancillary logic in a 32-slot Segment expanded to eight full Segments of 32 slots each.
6. Each crate in an expanded FASTBUS system may connect its Run/Halt switch to the terminal provided on the SE or to (parallel) terminals on the SE Bus connector board. Connection to the Run/Halt terminals on the expansion ATC ancillary logic module (located in the Primary Segment) can be made from the SE or the SE Bus connector.
associated with that Segment.

7. The SE modules at the physical ends of an SE Bus must have 100-ohm SE Bus terminator packages inserted to properly terminate electrically the SE Bus transmission lines.

8. The recommended maximum length of the SE Bus between SEs is 1 meter. Lengths greater than 1 meter may be used; however, there will be no guarantee that the system will operate properly owing to possible timing conflicts which may cause time-out logic to be activated.
Fig. 1 Star Multidrop Extension
FASTBUS SEGMENT EXTENSION

- MASTERS ALLOWED ON THE SOURCE SEGMENT ONLY
- GAC AND ATC ON THE SOURCE SEGMENT AND TERMINATORS ON OTHER SEGMENTS

Fig. 2 Linear Multidrop Extension
Hardware

The list below shows the FASTBUS signals that are actively or passively extended and those that are not extended:

Signal Lines Extended - Active
AD<31:00>
AS, AK, DS, DK, EG
MS<03:00>, SS<03:00>
WT, RD, SR, RB, BH
PE, DA
RX, TX (optional)

Signal Lines Extended - Passive
FP<02:00>, UR<01:00>
Ground (SE Bus - 1 per single line)

Segment Lines Not Extended
GA<04:00>
TR<07:00>
T - Pin
Reserved Lines
Daisy Chain Lines
Power Bus-Lines
Arbitration Lines
(AG, AI, AR, GK, and AL<05:00>).

Each FASTBUS signal line that is actively extended is provided with a bidirectional buffer implemented in ECL technology. The Segment driver gate, A, shown in Figure 3, and the SE Bus driver gate, C, are controlled (i.e., enabled or disabled) by a directional control signal developed on the SE. In some cases, this control signal is a static level dependent upon the function of the SE module (i.e., Primary or Slave). In other cases the control signal combines the latter with the state of the RD signal line to change dynamically the direction of the buffer during a FASTBUS cycle (EG, AD, PE, PA, etc.).

The Segment driver gate, A, and the Segment receiver gate, B, are physically located adjacent to the Segment connector to minimize capacitive loading of the Segment bus lines. The SE Bus driver gate, C, and the SE Bus receiver gate, D, are physically located adjacent to the SE Bus connector (i.e., the module Auxiliary connector). Printed circuit traces between the two halves of a buffer are arranged to introduce minimum signal coupling consistent with the circuit density requirements of the printed circuit design (traces are 10 mils in width and as close together as 10 mils when required). A photo of the prototype Segment Extension without front panel is shown below.

The FASTBUS timing signals (AS, AK, etc.) are electrically delayed with respect to the information lines. The delays are applied to both positive- and negative-going edges of the signal. In this way, both SE control decoding and signal differential delay are covered by a skew delay.
ACTIVE BIDIRECTIONAL BUFFER

Fig. 3 Active Bidirectional Buffer
Prototype Segment Extension

Signal Timing

Table 1 below indicates the timing relationships for a single SE.

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
<th>Value(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SE Bus</td>
<td>1 m nominal cable (flat ribbon cable)</td>
<td>Tcd = 5 ns</td>
</tr>
<tr>
<td>Bus loading</td>
<td>SE loading of SE Bus</td>
<td>T_l = 1.5 ns/SE</td>
</tr>
<tr>
<td>Delay</td>
<td>Control and Information</td>
<td>T(u) = 20 ns</td>
</tr>
<tr>
<td></td>
<td></td>
<td>T(d) = 15 ns</td>
</tr>
<tr>
<td>Delay</td>
<td>Timing Signals</td>
<td>TAS(u) = 30 ns</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TAS(d) = 20 ns</td>
</tr>
<tr>
<td></td>
<td>TAK(u), TDS(u), TDK(u), TEG(u)</td>
<td>TEG(u) = 35 ns</td>
</tr>
<tr>
<td></td>
<td>TAK(d), TDS(d), TDK(d), TEG(d)</td>
<td>TEG(d) = 25 ns</td>
</tr>
<tr>
<td>Skew</td>
<td></td>
<td>&lt;= 5% of Tcd</td>
</tr>
</tbody>
</table>

Table 1. SE Timing Delays
The increase in the random-data cycle time in an SE-expanded FASTBUS system with two or more Segments is indicated in Figure 4.
The relationship

\[ T_{cy} = 48(#\text{SEG}-1) + 550 \text{ ns} \]

has been developed from

\[
T_{cy} = 6[(T_{cd} + 2T_{l})(#\text{SEG}-1) + T_s] \\
+ 2T_{AS}(u) + 2T_{AS}(d) + 2T_{AK}(u) \\
+ 2T_{AK}(d) + 2T_{DS}(u) + 2T_{DK}(u)
\]

where Ts is the nominal Segment signal transit time for a fully loaded Segment (about 15 ns).

SE Control

The SE must either block EG and AS during a Geographic Address cycle or pass EG and AS with AD<07:05> set to logic 0, depending upon the match between the state of AD<07:05> when it arrives on the SE Bus and the state of the three address switches located on the SE. A nonmatch indicates that the GA address cycle was not directed at that SE, while a match indicates that the cycle was directed at that SE.

Other extremely important FASTBUS functions are the Case 3 or 3a Broadcast Read Operations (i.e., the Sparse Data Scan functions). These functions are also accommodated by testing for a match between AD<07:05> and the three address switches. In this last situation, the Case 3 or 3a Broadcast Operations are specifically decoded.

During a Geographic Address, the actual address test occurs when EG.AK=0.AS(u). In a Broadcast Case 3/3a Operation, the test occurs when MS = 2 or 3. AK = 0.AS(u). The direction of the AK buffer is reversed in a slave SE if a Broadcast Operation is decoded. The AK response signal serves to terminate the SE control- and select-period and latch the correct state for the remainder of the FASTBUS Operation. The use of the AK response signal in this way is important in either normal or Broadcast FASTBUS Operations.

The SE relays Segment signals during a Logical Address Operation or Broadcast Write Operations, imposing only the delays noted above.

Finally, the SE requires 750 ma from the -2.0 Volt rail if SE Bus terminators are in place, and 550 ma if the terminators are out. The SE draws 2200 ma from the -5.2 Volt rail.

Software Considerations

The software used for either overall system control or for local Segment control during the operation of a FASTBUS data system must be fully aware of the existence of SE-expanded Segments and their location in the system. A major consideration in the development of the SE was minimum cost, minimum hardware, and operational simplicity. These constraints are reflected in the fact that the SE is totally transparent to user software, once slave modules are initialized and Logical Addresses are
enabled. The SE provides no identifying response to Geographically Addressed search operations other than the response of slave modules on extended Segments. A FASTBUS Read Operation directed to the slot containing the SE, if AD<07:05> are set to logic 0, will produce an Address Response timeout, as if no module were there at all. Additionally, an improperly configured SE will cause an Address Response timeout when a Master attempts to access a module off the Primary Segment. Thus, system initialization software that scans a FASTBUS system Segment by Segment, searching for a response to a CSR#0 Read Operation, will find no response, and, unless told otherwise, will assume that no module exists in the slot occupied by the SE (the Primary SE in this case).

The controlling software must be provided with an extended addressing table that is associated with the module address-and-location table. Provided expanded Segments are present, the Geographic Address of a Slave module will be the sum of its actual Geographic Address and the Address set in the SE that couples the Segment to the SE Bus. It is not possible to read the SE address back from the SEs expanding a Primary Segment, again owing to the necessity of providing a crate-coupling device at the lowest possible cost. However, address timeout errors occurring during Geographic Addressing of extended Segments will most probably be caused by improper address switch settings or improper address table entries. The communications between the systems engineers, the software engineers, and the experiment physicists must be complete if the overall system is to function. Users and system programmers must also be aware that one and only one SE must have Master status set. This status is displayed in the green front-panel "MASTER" LED. The SE displaying Master status must be inserted in the primary crate. All other SEs must have their yellow LED or "SLAVE" indication active and be inserted in the extension Segments.

The reliability of data transfer in an SE-expanded system is related to SE Bus termination, and inversely to the overall length of the SE Bus. Although the systems engineer has the ultimate responsibility for the SE Bus, the systems programmers can easily verify that the bus terminators are in place in the proper position. The "TERM" LED will indicate which SE has bus terminators installed. The terminators must be at either end of a single SE Bus or in a two-leg star configuration.

System operation with SNOOP modules is possible. The SNOOP diagnostic module developed at the Stanford Linear Accelerator Center (SLAC) in California uses the RX and TX Segment lines for communications. If an SE-expanded system is equipped with SNOOP modules in each Segment, the RX and TX buffer circuits must be disconnected from the SE Bus at each SE. Jumper connections are provided near the upper position of the SE Auxiliary connector to effect this configuration change.
Conclusions

An inexpensive, easy-to-use Segment expansion device has been developed for the FASTBUS data system. The Segment Extension complements the existing Segment Interconnect, which allows multimaster operation on each Segment and structured message routing; the SE logically expands a Master or Primary Segment into as many as 7 physical Segments, allowing only Slave modules in the extension Segments. The delays associated with an SE-expanded system have been determined for several modes of FASTBUS operation.

The Segment Extension was developed under contract to the U.S. Department of Energy by Scientific Systems International and will be manufactured and marketed by KineticSystems Corporation as a FASTBUS standard product.

References


2. ibid, chapter 10, The Interconnection of Segments


4. op. cit., chapter 4, FASTBUS Operations: Addressing

5. ibid, chapter 6, Bus Arbitration
The Crate Clustering Card

S.R.Amendolia\(^1\), F.Fidecaro\(^1\), S.Galeotti\(^1\), B.Lofstedt\(^2\), P.S.Marrocchesi\(^1\) and D.Passuello\(^1\)

Geneva, 25 September 1985

Abstract: A simple and cheap device for the extension of the Fastbus Crate Segment is described. It allows the use of master modules in any one of the crates forming the crate cluster. Up to four crates can be connected together.

1. Introduction

The idea of a simple and flexible Fastbus backplane extension was originated in the Aleph experiment by the needs of the Time Projection Chamber read-out system.

This is made of 108 Fastbus crates, divided in 36 racks. The 3 crates of each rack house the read-out electronics of a chamber's sector, which is controlled by two Fastbus master modules, linked to the rest of the system by a Fastbus cable segment.

These 3 crates can be seen as one backplane.

2. Requirements

The design had to obey the following constraints:

- to be standard Fastbus

\(^1\) INFN, Dipartimento di Fisica, Scuola Normale Superiore, Pisa, Italy
\(^2\) CERN – EP Division
• have the possibility of forming a cluster of up to 4 crates
• be able to use any standard Fastbus slave module in any of the crates connected together
• be able to use any standard Fastbus master module in any of the crates, allowing it to per-
  form any Fastbus operation on the whole system
• have the possibility to gain access to the cluster of crates from a Segment Interconnect
  placed in any of the connected crates
• no slots in the crate backplane have to be occupied
• have a low cost

3. Implementation

We worked out a solution, called Crate Clustering Card (CCC), which satisfies the above require-
ments.

The two to four crates forming the cluster are identified by the two least significant bits of the
group field: these constitute the "cluster address", ranging from 0 to 3.

This choice allows us to put a master in any of the crates. In facts, any slot in the "extended back-
plane" has a geographical address made up by its geographical position plus its cluster address, inde-
pendently of where it is addressed from.

The one crate in the cluster which contains the Ancillary Logic Card must have a cluster address equal
to 0.

The connection between any two crates in the cluster is made up by a pair of CCCs, one in each
crate.

The Fastbus signals carried through the connection are:

- AL < 00:05 >, SS < 00:02 >, MS < 00:02 >, AK, AS, DK, DS, AG, AI, AR,
- GK, RD, PA, PE, RB, BH, WT, SR, AD < 00:31 >
These signals are repeated on the two cards via the bidirectional circuit shown in Fig. 1: per each signal, there are two "differential" lines and a ground line.

One of the two CCC in a pair includes an EG generation logic circuit, shown in Fig. 2.

For a crate with cluster address "n", this circuit performs the following logic operation:

\[
\text{EG} = \text{.NOT. AK} \ast \text{.NOT. MS1} \ast \text{.NOT. MS2} \ast \\
\{\text{AD < 24:25} \ast \text{.EQ. n)} \ast \{\text{AD < 08:23} \ast \text{.EQ. 0)} \ast \\
\{[\text{AD < 26:31} \ast \text{.EQ. N}] + \{\text{AD < 26:31} \ast \text{.EQ. 0)]
\]

The value of "N" and "n" (which form the Group Number GPN) are switch selectable on the CCC which contains the EG logic generation.

The CCCs in a crate occupy the first and last slot of the backplane (on the rear side). They provide for a correct termination of the bus.

A typical 3 crates cluster is shown in Fig. 3, where 4 CCCs are used together with an Ancillary Logic Card, a Bus Terminator and a Segment Interconnect.

4. Broadcast handling

Special attention was paid to the handling of the broadcast messages: i.e. we require that a master in one of the crates can send a broadcast to another crate in the cluster without affecting the remaining ones.

One limitation arises since, due to the linear structure of the crate cluster (ref. Fig. 3), as opposed to the tree structure of Fastbus interconnections, it is not possible to have a broadcast effective on "one of the clustered crates" (identified by a group number Ni) "and on the subtree below Ni", being this subtree non defined in our cluster structure.

In a crate cluster made via the CCCs, any broadcast to one of the crates which is made with the Global Bit ON, is effective on the whole cluster.
In order to allow the local broadcast, the AD00 signal (Local Bit) is not just transmitted from crate to crate (as for the other signals, ref. Fig. 1), but is regenerated by the circuit shown in Fig. 4, which performs the following logic:

\[
AD00 = MS1 \cdot \text{NOT. AK} \cdot ([AD < 24:31] \cdot \text{EQ. GPN}) + AD01
\]

\[
AD00 = (\text{NOT. MS1 + AK})
\]

Table 1 shows the behaviour of this circuit, when the broadcasting master is in crate 3 (as shown in Fig. 3).

<table>
<thead>
<tr>
<th></th>
<th>crate</th>
<th>crate</th>
<th>crate</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>N3</td>
<td>N0</td>
<td>N1</td>
</tr>
<tr>
<td>GP</td>
<td>G</td>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>N3</td>
<td>0</td>
<td>X</td>
<td>1</td>
</tr>
<tr>
<td>N1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>N1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>N0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>N0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>N0</td>
<td>1</td>
<td>X</td>
<td>1</td>
</tr>
<tr>
<td>N3</td>
<td>1</td>
<td>X</td>
<td>1</td>
</tr>
<tr>
<td>N1</td>
<td>1</td>
<td>X</td>
<td>1</td>
</tr>
<tr>
<td>M</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>M</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
5. Status of the project

The CCC prototypes have been successfully tested.

The CCCs are now in the engineering phase for pre-series production.

The CCCs will be plugged in the rear side of the backplane: several mechanical solutions are being tested to accommodate the different Fastbus crate structures now on the market.

The power consumption of a CCC is about 11 Watts on -5.2 Volts and 2 Watts on -2 Volts.

Fig. 1: Bidirectional line transmission circuit
Fig. 2 EG generation logic
Fig. 3: 3-crates cluster

Fig. 4: Modification to the AD00 line (Broadcast Local Bit)
PARTICIPANTS

AMENDOLIA, S.R.
INFN Sezione di Pisa
v.Vecchia Livornese 582/A
I — 56010 S.Piero a Grado (PI)

ARNAULT, C.
LAL
Bat.200
Université PARIS XI
F — 91405 ORSAY

BAHLER, P.
DD Division, CERN

BAL, F.
EP Division, CERN

BARLOW, J.
RAL
Chilton
GB — DlDcot OX11 0QX

BERNERS-LEE, T.J.
DD Division, CERN

BERTL, W.
SIN
5234 VILLIGEN

BERTSCH, Y.
L.A.P.P.
IN2P3
B.P.909
F — 74019 ANNECY—LE—VIEUX CEDEX

BIBBY, J.
Nuclear Physics Dept.
University of Oxford
Keble Road
GB — OXFORD OX1 3NP

BIZEAU, C.
DD Division, CERN

BLUME, H.
DESY — F36
Notkestrasse 85
D — 2000 HAMBURG 52

BOGAERTS, A.
DD Division, CERN

BOLOGNA, G.
EP Division, CERN

BOUQUET, B.
LAL
Bat.200
Université PARIS XI
F — 91405 ORSAY

BOUROTT, J.
LPNHE
Ecole Polytechnique
Route de Saclay
F — 91128 PALAISEAU

BOZZOLI, W.
DD Division, CERN

BURCKHART, D.
DD Division, CERN

CAMPBELL, A.
DESY, F14
Notkestrasse, 85
D — 2000 HAMBURG 52

CAMPBELL, J.R.
Blackett Lab. (HENP)
Imperial College
Prince Consort Road
GB — LONDON SW7 2AZ

CANDLIN, D.J.
Physics Department
University of Edinburgh
King’s Buildings
GB — EDINBURGH EH9 3JZ

CHERRY, M.
EP Division, CERN

CHOLLET, F.
L.A.P.P.
IN2P3
B.P.909
F — 74019 ANNECY—LE—VIEUX CEDEX

CITTOLIN, S.
DD Division, CERN

CLARKE, P.
EP Division, CERN

DARBO, G.
EF Division, CERN

DE CLERCQ, C.
Vrije Universiteit Brussel
IIHE
Pleinlaan 2
B — 1050 BRUSSELS
DEGRE, A.
L.A.P.P.
IN2P3
B.P.909
F — 74019 ANNECY—LE—VIEUX CEDEX

DOBINSON, R.W.
TRIUMF
4004 Wesbrook Mall
UBC Campus
VANCOUVER B.C.
Canada V6T 2A3

ECK, C.
DD Division, CERN

EINSWEILER, K.
EP Division, CERN

EVERS, G.J.
NIKHEF — H
P.O.Box 41882
NL — 1009 DB AMSTERDAM

FALCIANO, S.
Istituto di Fisica G.Marconi
Piazzale A.Moro, 2
I — 00185 ROMA

FIDECARO, F.
INFN Sezione di Pisa
v.Vecchia Livornese 582/A
I — 56010 S.Piero a Grado (PI)

FREMONT, G.
EP Division, CERN

GAGLIARDI, F.
DD Division, CERN

GALEOTTI, S.
INFN Sezione di Pisa
v.Vecchia Livornese 582/A
I — 56010 S.Piero a Grado (PI)

GAVILLET, Ph.
EF Division, CERN

GOPAL, G.
RAL
Chilton
GB — DIOCOT OX11 0QX

GOSMAN, D.
NIKHEF — H
P.O.Box 41882
NL — 1009 DB AMSTERDAM

GRUHN, C.R.
EF Division, CERN

GRUHN, U.
EP Division, CERN

GUSTAFSSON, L.
DD Division, CERN

HARRIS, J.F.
Nuclear Physics Dept.
University of Oxford
Keble Road
GB — OXFORD OX1 3NP

HARVEY, J.
EP Division, CERN

HECK, B.
EF Division, CERN

JACOBS, D.
DD Division, CERN

KALKANIS, G.
EF Division, CERN

KATSANEVAS, S.
EP Division, CERN

KOZLOWSKI, T.
DD Division, CERN

LANDON, M.P.J.
Physics Department,
Royal Holloway College
Egham Hill, EGHAM,
GB — SURREY TW20 0EX

LECOQ, J.
L.A.P.P.
IN2P3
B.P.909
F — 74019 ANNECY—LE—VIEUX CEDEX

LOFSTEDT, B.
EP Division, CERN

LOKEN, J.G.
Nuclear Physics Dept.
University of Oxford
Keble Road
GB — OXFORD OX1 3NP

MACHEN, D.R.
Scientific Systems International
3491 B Trinity Drive
LOS ALAMOS
U.S.A. — NM 87544

MAPELLI, L.
EP Division, CERN

MARCHIORO, A.
EF Division, CERN
MARROCCHESI, P.S.
INFN Sezione di Pisa
v.Vecchia Livornese 582/A
I — 56010 S.Piero a Grado (PI)

MASONI, A.
Dipartimento di Scienze Fisiche
Universita degli Studi,
via Ospedale, 72
I — 09100 CAGLIARI

MCLAREN, R.
DD Division, CERN

MCPHERSON, G.
EF Division, CERN

MORNACCHI, G.
DD Division, CERN

MULLER, Hans
EP Division, CERN

OLSCHLAGER, H.
Company Dr.Struck
Hauptstrasse 95
D — 2000 TANGSTEDT

OSEN, K.
DD Division, CERN

PATRICELLI, S.
Dipartimento di Fisica
Padiglione 20
Mostra d’Oltremare
I — 80125 NAPOLI

PEACH, K.
EP Division, CERN

PERRIN, Y.
DD Division, CERN

PERROT, G.
L.A.P.P.
IN2P3
B.P.909
F — 74019 ANNECY—LE—VIEUX CEDEX

PICHEROT, G.
DHFN/HE
CEA — SACLAy
F — 91191 GIF/YVETTE CEDEX

PIZZOCARO, M.
LAL
Bat.200
Universite PARIS XI
F — 91405 ORSAY

PONTING, P.
EP Division, CERN

PORDES, R.
MS 120
FERMILAB, P.O.BOX 500
BATAVIA
U.S.A. — IL 60510

PRUYS, H.S.
S.I.N.
CH — 5234 VILLAGEN

PYLGRIMS, B.
EP Division, CERN

RANIERI, A.
INFN
via AMENDOLA, 173
I — 70125 BARI

RENAUDY, J—F.
SEPH — CEN
SACLAy
F — 91191 GIF/Yvette CEDEX

RIMMER, E.M.
DD Division, CERN

RINANDO, G.
Istituto di Fisica
Corso M.D’Azeqlio, 46
I — 10125 TORINO

ROMAYA, A.M.
Nuclear Physics Dept.
University of Oxford
Keble Road
GB — OXFORD OX1 3NP

RUGGIERI, F.
INFN
via Amendola, 173
I — 70126 BARI

SCHIAVI, J.
EP Division, CERN

SCHWARTZ, A.
High Energy Physics Lab.
University of Harvard
CAMBRIDGE
U.S.A. — MA 02138

SEDBEE, J.
Blackett Lab. (HENP)
Imperial College
Prince Consort Road
GB — LONDON SW7 2AZ

SENDALL, D.M.
DD Division, CERN

167
SHERDEN, D.
SLAC
P.O.Box 4349
STANFORD
U.S.A. — CA 94305

SIMONET, L.
EF Division, CERN

SMITH, L.S.
Oliver Lodge Laboratory
Department of Physics
University of Liverpool
GB — LIVERPOOL L69 3BX

STORY, C.M.
DD Division, CERN

TEMPESTA, P.
INFN
via Amendola, 173
I — 70126 BARI

THENARD, J—M.
L.A.P.P.
IN2P3
B.P.909
F — 74019 ANNECY—LE—VIEUX CEDEX

TREMBLET, L.
DD Division, CERN

VAN DE VYVRE, P.
DD Division, CERN

VAN DER VLUGT, C.R.
EP Division, CERN

VAN INGEN, C.
FERMILAB
P.O.BOX 500
BATAVIA
U.S.A. — IL 60510

VON RUDEN, W.
EF Division, CERN

VINCELLI, M.
LeCroy Research Systems SA
Route Nant D'Avril, 101
1217 MEYRIN

WERNER, P.
EP Division, CERN

YAMDAGNI, N.
Institute of Physics
University of Stockholm
Vanadisvägen 9
S — 11346 STOCKHOLM

YOST, B.T.
LeCroy Research Systems Corp.
700 South Main Street
SPRING VALLEY
U.S.A. — NY 10977

YOUNGMAN, Ch.
EP Division, CERN