SLAC-PUB-2995
October 1982
(M)

SOFTWARE FOR MANAGING MULTICRATE FASTBUS SYSTEMS*

S.R. Deiss and D.B. Gustavson
Stanford Linear Accelerator Center
Stanford University, Stanford, California 94305

Abstract

The FASTBUS System Manager software that was designed and implemented on an LSI-11 system using PASCAL is described. Particular attention is given to the file structures, file access mechanisms, and basic routing algorithms. Portability to other machines and languages is described.

Introduction

FASTBUS systems can involve multiple processors operating in parallel on interconnected segments.[8] FASTBUS architecture allows the efficient configuration and operation of such large parallel systems. However, powerful software is required to assist the experimenter in configuring, initializing, controlling, and maintaining these systems. For example, modules that are programmed via downloading must be given the addresses of all those other modules that they talk to. But these addresses themselves change as the system evolves. Furthermore, each Segment Interconnect has a table of address groups for each direction of transfer through it. These tables must be initialized properly for the system to allow any coherent intersegment communication.

The FASTBUS SYSTEM MANAGER is a software system written in transportable UCSD PASCAL II.0 for the purpose of helping the experimenter manage large, complex systems.[1,7] It has a data base that describes the overall configuration and pertinent details about each module. It includes a data base editor with an English-like command language. It can generate route maps that allow all modules to talk to all others. With improvements it would be able to initialize the system's SI's and downloadable modules and provide configuration management tools for running experiments.

Although this software has many 'big system' algorithms, it was designed with small systems in mind too. While its most likely home would be in a large mainframe such as a VAX the FASTBUS SYSTEM MANAGER was developed on and currently runs on an LSI-11 with a modest amount of memory and dual 8" floppy disks. The software would be a very good match for such a device as the SLAC FASTBUS CONTROLLER.[6] However, even in the very small environment of an LSI-11 enough code and data can be stored to support a system of several dozen FASTBUS crates.

Management Tasks

It was recognized early that there are many tasks involved in monitoring and controlling a FASTBUS system.[2]

SYSGEN

There has to be a DATA BASE which describes all the segments and their contents. This data base could contain a minimum of information and a minimum of structure if one merely sought to turn the system on, let it run a few years, and then turn it off. However, since getting the system turned on and adjusted right consumes the bulk of the expenditure of human effort, it makes sense to include in the data base any information that would assist in the turn on process and in the maintenance process.

Using this data, the routing algorithms described below can generate SI route maps and allocate address space in order to free the experimenter for other tasks. However, it is possible that the experimenter might desire to completely specify the routes and the BROADCAST TREE and assign address space by hand. Or, perhaps, there is only a need to specify these things manually in one critical part of the system. The programs that do the sysgen should be sophisticated enough to allow this direct specification. It would serve as a set of baseline constraints from which everything else could then be computed automatically.

DEADSTART

As the name implies, when a FASTBUS system or subsystem is deadstarted every module, SI, and segment has to be set to a known initial state which is consistent across all components. Another way of saying this is that everywhere there is a bit of uncertainty in the system, that bit must be specified at deadstart. Considering the number and variety of modules in a large system it becomes a substantial task just to organize the information needed to initialize each one while maintaining some common format that an automatic initializer can follow.

For example, it is easily possible for a FASTBUS crate to contain 25 intelligent modules each of which has to be told what to do and how to do it. This introduces a related problem of software development cross products. If each of those 25 modules is a microprocessor, it is (understatement) a challenge to think of a way that the software development process could be unified around a common set of cross products. However, even if one does not set their sights that high, it is still a challenge just to figure out a common format for all load modules and linkage information so that one program could load the 25 modules with the happy result that they can all talk to and understand each other.

VERIFICATION

In large systems during developmental stages, it would be very helpful if the software could interrogate the system and verify that it matches the description in the data base. Often two installa-

----

* Work supported by the Department of Energy, contract DE-AC03-76SF00515.

(Presented at the Nuclear Science Symposium, Washington, D.C., October 20-22, 1982.)

tion teams might find themselves at cross purposes unless each can quickly find out how the other group left the system configured. Likewise shift changes during development can create havoc without a way of finding out the configuration.

Even after the system is running it often helps if one has a way to tell if the replacement module plugged in yesterday is really the latest revision that it was thought to be. This is especially true if the modules are scattered over large distances that preclude a hands-on check.

SYSTEM PAUSE

If you have subsystems that require some kind of synchronization in time, it may be necessary to start and stop them in unison. Alternatively, a maintenance procedure might be required which involves a health hazard unless some part of the system is momentarily ´paused.´

RECONFIGURATION

In a very large system one might want the flexibility to move interchangeable components around without requiring a whole new sysgen. The problem here is the meaning of the term ´interchangeable.´

For example, can a large memory card be plugged in in place of a failing small memory? Does anything in the data base have to be updated? Do any of the modules that talk to the memory have to have their algorithms adjusted? And so on. Again these things are not the sort of problems one encounters in a running experiment, rather they crop up when the experiment is being installed and brought on line.

ERROR RECOVERY

If a system has many intelligent modules, these might be capable of some self-test or of subsystem monitoring and testing. If one of these watchdog modules sees an error or a pending disaster, it might inform the system management software in expectation of some kind of intelligent diagnostic action.

All extremes are imaginable, ranging from a report of a failing memory module detected by too many parity errors to a request that the system dynamically reroute itself around one particular segment that is suffering from a traffic overload.

Then there is always the usual startup maintenance and running maintenance tasks that require test procedures to be run on system components. If nothing else, one might wish to log errors somewhere for adjustment of experimental results even when no automatic recovery can be performed.

These are some of the major tasks one would hope to be anticipated in the design of an on-line FASTBUS management system.

The FASTBUS SYSTEM MANAGER

In 1980 the committment was made to develop a prototype of this kind of software system. The purpose of the prototype would be primarily to serve as an existence proof for some of the routing algorithms required by FASTBUS. However, secondary purposes included gaining experience with different ways of partitioning and structuring the data base required to serve all of the above tasks.

PASCAL had been chosen as the unofficial ´publication language´ of the FASTBUS Software Working Group. That plus the availability of UCSD PASCAL for LSI-11´s and familiarity with it made it a natural choice for a pilot prototype. It had the added advantage of being a transportable system requiring minimal hardware support. Thus, it was thought that others might be able to transport the software to other environments for further prototype development.

The FASTBUS SYSTEM MANAGER, or FSM, described herein was written during July through October of 1980. It was subject of discussion at the 1980 NSS meeting in Orlando, FL.[3] Based on that discussion

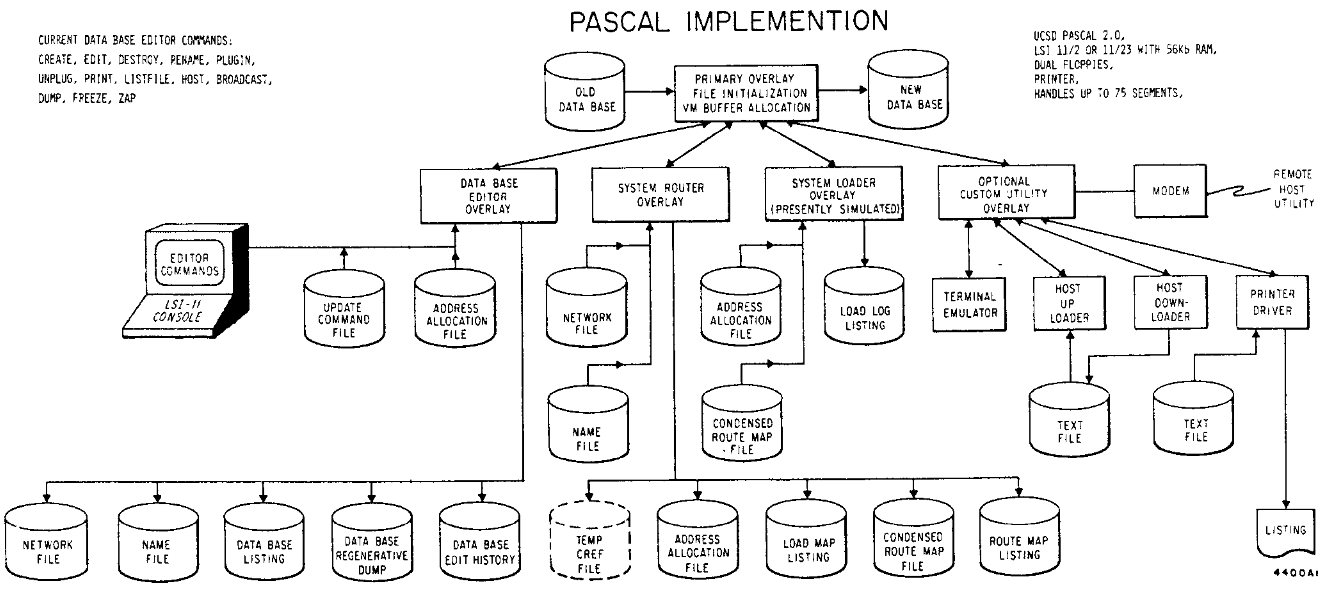

Figure 1

and other considerations a few new features were added and a live demonstration was given at SLAC in January of 1981.[1] This demonstration was further fine tuned and presented as part of a larger FASTBUS demonstration at the 1981 NSS in San Francisco, CA.[4]

Figure 1 provides an overview of the entire FSM as it presently exists. The primary overlay provides global procedures and common data structures for all the secondary overlays of which there are four.

The data base EDITOR overlay provides a simple means of making changes to the data base that is TTY compatible. It can accept interactive commands or input can be directed to it from one or more files that were prepared with some other editor. Likewise, its output can be sent to the terminal or it can be sent to a log file so that the user has a record of what has been done to the data base.

The ROUTER overlay takes the latest data base information and 1) makes address space assignments, 2) generates compacted route maps, 3) makes a broadcast tree, and 4) generates several optional listings.

The LOADER overlay performs a simulation of the deadstart process. At the time the code was written no SI hardware was available to make the actual device drivers needed to do a real system initialization. Also additional work is needed to design a universal load module format. This is desirable so that one loader overlay or program can initialize any kind of device using a device independent representation of the sequence of operations needed to load or otherwise initialize it.

The fourth and final overlay is an optional site-specific utility overlay. At SLAC this overlay contains utilities for uploading and downloading files to and from WYLBUR, for emulating a terminal on WYLBUR, and for making listings on a small local printer.

Data Base

Not counting listing files and editor input files, the data base logically consists of four major files: 1) The NETWORK file, 2) the NAME file, 3) the ROUTE MAP file, and 4) the LINKAGE file.

The network file, NET, contains all the system interconnection information and everything needed by the routing algorithms (Fig. 2). This file is a random file of fixed length records of 5 variant types. The directory variant contains global information telling which record describes the host segment, which is for the broadcast source segment, where the first record is in the linked list of segments, the head of a list of free records, and the head of a list of modules or SI´s which exist, but have not been plugged in. The free variant is just a blank record containing a pointer to the next in that chain. The segment variant incudes: optional segment address and amount of address space required, pointer to first SI in the list of same on that segment, pointer to a list of modules for the segment, pointer to the next segment following this one. An SI variant in the NET file contains: the SI group field size, a type of transforming or non-transforming, arbitration vectors for both sides of the SI, segment slots for both sides, segment numbers for both sides, next segment numbers for both

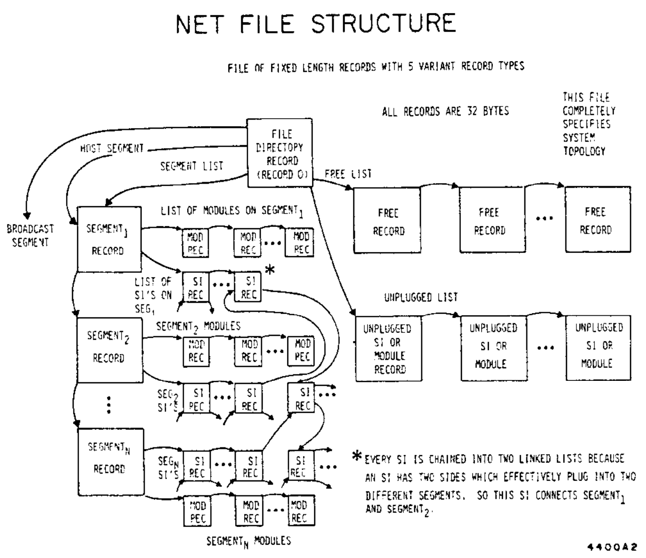
NET FILE STRUCTURE
FILE OF FIXED LENGTH RECORDS WITH 5 VARIANT RECORD TYPES
ALL RECORDS ARE 32 BYTES
THIS FILE COMPLETELY SPECIFIES SYSTEM TOPOLOGY
FILE DIRECTORY RECORD (RECORD 0)
HOST SEGMENT
SEGMENT LIST
FREE LIST
BROADCAST SEGMENT
SEGMENT$_1$ RECORD
LIST OF MODULES ON SEGMENT$_1$
MOD REC
FREE RECORD
LIST OF SI'S ON SEG$_1$
SI REC
UNPLUGGED LIST
UNPLUGGED SI OR MODULE RECORD
UNPLUGGED SI OR MODULE
SEGMENT$_2$ RECORD
SEGMENT$_2$ MODULES
SEG$_2$ SI'S
SEGMENT$_N$ RECORD
SEG$_N$ SI'S
SEGMENT$_N$ MODULES
* EVERY SI IS CHAINED INTO TWO LINKED LISTS BECAUSE AN SI HAS TWO SIDES WHICH EFFECTIVELY PLUG INTO TWO DIFFERENT SEGMENTS. SO THIS SI CONNECTS SEGMENT$_1$ AND SEGMENT$_2$.
4400A2

Figure 2

sides (each side of the SI is in a list of SI´s for the segment it is plugged into), pointers to the route map tables in the MAP file for the two sides of the SI, and a pointer to the next SI in the chain of SI´s which all have the same size of group field. A module variant contains an optional address (relative to the segment address) and address space allocation, a segment number, a slot, a pointer to the next module on that segment, and a code number for an initialization routine.

All of the above variants contain a common area that holds a search heuristic. This consists of the first, middle, and last characters of the object´s full name and the length of the name. When an object is referred to by name during editing, it is this template that is used to match against. Once a match is found the object´s full name is retrieved from the NAME file to confirm the match. This was done so that little paging would be required for NAME file records which are large. More will be said about paging below.

As can be seen the NET file contains multiple linked lists. The routing algorithms are linked list oriented, and a special list processing subsystem was written to provide uniform linked list processing, no matter what the list. The list processing functions are 1) get the head of the list, 2) get the next record in the list, 3) insert a record in the middle of a list, 4) remove a record from a list, and 5) find a record in a list. The list routines all require the list type as a parameter in order to know where to look in the record for the list pointers.

The NAME file, NAM, contains for each object the information which is used infrequently or not at all during routing. This file is referenced most heavily by the EDITOR (Figure 3). As before, there is a directory variant and a free variant. The directory only contains pointers to the heads of lists of free records and used records. The free record is blank other than the chaining pointer. The rest of the records in the name file are of one variant that is used for segments, modules and SI´s. This record variant contains: a chain pointer to the next, a pointer to the corresponding record for it in the NET file, a pointer to a diagnostic (unused), a pointer to an initializer (unused), a pointer to an error recovery routine (unused), a 32 byte name string, and 4 12-character strings for manufacturer's part number, serial number, inventory number, and requisition number.

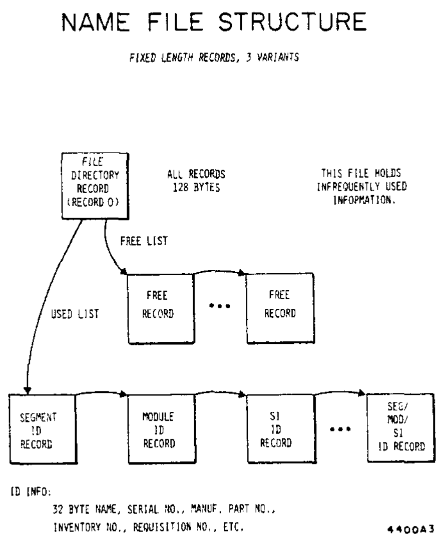
NAME FILE STRUCTURE
FIXED LENGTH RECORDS, 3 VARIANTS
FILE DIRECTORY RECORD (RECORD 0)
ALL RECORDS 128 BYTES
THIS FILE HOLDS INFREQUENTLY USED INFORMATION.
FREE LIST
USED LIST
FREE RECORD
FREE RECORD
SEGMENT ID RECORD
MODULE ID RECORD
SI ID RECORD
SEG/ MOD/ SI ID RECORD
ID INFO:
32 BYTE NAME, SERIAL NO., MANUF. PART NO.,
INVENTORY NO., REQUISITION NO., ETC.
4400A3

Figure 3

The thinking behind the three unused pointers is that someday the data base could be extended to include additional files of initialization, diagnostic, and error recovery procedures. The only assumption made about these files is that the information in them could be readily accessed with a 16 bit pointer. For example if each of those files was organised as random with fixed length records, the pointer might tell where the desired record sequence starts. Much additional space is reserved in these records for future additions such as pointers to maintenance manuals, user notes, and other things not yet thought of.

The ROUTE MAP file, MAP, contains condensed representations of all the route maps. In a large system the NET and NAM files quickly expand to many hundred K bytes of data. If the raw route maps were stored, they could waste a lot of space. Consider 12 bit SI's. One hundred of them requires 200 X 4096 X 3 bits at least. Clearly compaction is necessary.

Maps are stored as a boolean array with one boolean for each segment in the system. For each segment the boolean value tells whether or not that segment's addresses are passed through this SI. This way 100 SI's now requires 200 X N bits where N is the number of segments. This is roughly a 120 fold space savings in a 100 segment system. The cost is that the system initializer has to expand these maps at load time using the LINKAGE information that tells where things were put in address space.

Finally, the LINKAGE file is output by the router along with the route maps. This file tells the address space allocation for each segment, and within each segment, for the modules. This file is written to disk in a peculiar order that directly corresponds to the order in which system components would be initialized. The route maps are likewise ordered. As a result these two files can be read efficiently at initialization time in order to expedite a fast load of the system even with floppy disks.

Data Structures

At execution time several additional data structures come into play. Space does not permit mention of all of them, but a couple of them stand out as unique and powerful.

In this implementation the compromise took the form of a simulation of virtual memory built into the file access mechanisms for the NET file and the NAM file. Whenever a record from either of these files is accessed, the record number is first passed through a function which returns a revised number - not the record number in the file - but the number of the record in a HEAP resident cache array of records. In other words the funtion has deliberate and predictable side effects. The function scans the cache for the record. If not there, it looks for a free cache slot to put it into. If none, it pages out the oldest record in the cache using a LRU algorithm (assuming the oldest record is flagged as having been dirtied). Then the record being referenced is paged into that free cache slot, and the slot number is returned. The code making the reference has to be written to indirectly reference file records through the cache array using record numbers that are converted by the cache management funtions. The other requirement is that the referencing code must flag the pages that it will dirty when it references them by using a negative record number.

The result of this approach is to make record access expressions slightly more cumbersome to read while eliminating the need for the program code to optimize its disk accesses with some kind of buffering. Without this simplification the code could never have been written in such a short time frame.

ROUTER

The task of the router is to find the shortest reversible unique route between every possible pair of segments. This is done so that every segment can send messages to every other and vice versa. When two routes are available that are of equal length, the router chooses the one which has the largest minimum size SI along the path, ie., the smallest window. The reason for using the smallest window is that when systems contain more than one size of SI, they naturally partition themselves into clumps which have to be allocated address space in quanta that correspond to the size of next smaller SI that forms the partition boundary. By always choosing the paths that avoid coarser (smaller) SI's where possible, one avoids address space fragmentation later. Further algorithm work remains to find a way to use the clump partitioning information in the allocation of address space.

The route map generator overlay uses three square matrices allocated from the HEAP. The 3 matrices include a PATH matrix which shows at each row/column intersection which segment to go through first on the route connecting the segments represented by the row and the column. When initialized PATH shows which segments are directly connected by an SI. Path also shows what size the smallest SI is on the route. During each iteration the router spreads out one further level using the new segments that it found after the last iteration. Each time it updates a NEWS matrix that tells it what new paths were found if any. It then checks a running total matrix, RTM, that tells it if any segments remain unconnected. The routing is successful when RTM is all filled in. There is a failure if an iteration results in no news and RTM still has blanks.

The routing algorithm serves two more purposes. It can optionally generate a broadcast tree after the routes are generated. This is done by using a backtrack algorithm which works back from all segments toward the broadcast source segment until either it is reached or some other segment already a part of the tree is crossed.

The other purpose of the router is to assign address space to segments and to the modules within segments. This algorithm works for systems in which all SI´s are the same size (all systems to date), but it needs refinement to work in the general case of systems with many SI sizes. The algorithm uses the same basic scheme to assign space for segments and for modules. The same subroutine is used for both. As the data base is scanned to initialize the data structures needed in routing, a temporary file is written out that has address space allocated for the modules within a segment. At the same time a dynamic array is built that contains 2 linked lists of segments: a list of those fixed at addresses by user request, and a list of segments free to move about. As soon as all the data has been read in the mobile segments are merged in with the fixed ones taking the largest mobile segments first. When this process is complete the temporary file is read back in and updated with the segment address space allocations. This new file is then written out as the linkage file. It is used later to expand route maps. It is the basis of the deadstart (LOADER) routines. It also is the source of the load map which is produced for both the user, and for use by any cross software development packages that need external references resolved.

The load map tells for segments their base address and their high address. For modules it gives their logical address and their geographical address. For SI´s the load map only specifies a geographical address for each side of the SI.

The EDITOR

The data base editor allows the user to perform several functions: create segments, modules, and SI´s, edit their descriptions later, delete them when necessary, plug modules and SI´s in, and unplug them, specify a HOST segment and a BROADCAST source segment, make listings of selected data base items, and dump out data base contents in a ready made format that can be used to recreate it later if something goes wrong.

The editor command parser is a simple state transition system driven by the first 3 letters of command keywords. A list of noise words is recognized and ignored. A list of ignorable characters is also dropped from the input. There is also a list of context words which are used as search heuristics. For example when the user says ´PRINT SEGMENT A´, the system sets a context variable to indicate that the referent is a segment. When the data base is scanned for the information to print, only segments will be checked to save search time. Even then the previously mentioned heuristic is used to speed up the search further.

The LOADER

The loader is simplified because the MAP and LINKAGE files were previously written out in an order conducive to system initialization. That is, the route maps and module references are stored in exactly the order needed at load time. This order consists of a depth first algorithm starting from the HOST.

When a segment is first encountered, it is reset. Then all of its modules are loaded using the initialization code in the NET file. Then the loader loops through the SI´s on the segment. For each it loads the outgoing map and looks at the far side segment to see if it has been initialized. If it has not and if the reverse map contains the HOST, the loader then recursively starts loading the far side segment. When the above test fails, the loader will continue to the next SI in the list on the segment until all are finished. Then it can back up to the segment it was working on when it was sent on to this one. The recursive approach requires two 16 bit words to be pushed down for each level of recursion.

Current Work

FNAL was supplied with copies of source code and documentation for assistance in applying FSM techniques to the CDF FASTBUS software. As a result routing and address space allocation algorithms are being revised and the structure of the data base is being carefully examined.[5] In this system it is proposed to integrate the System Management data base with the experimental data base in a unified system, perhaps using a commercial product.

Summary

The FASTBUS SYSTEM MANAGER software has been described up to its current state. While it only begins to tackle the larger tasks of system management, it has demonstrated that they fall within reach of even modest computational environments.

Acknowledgements

The authors wish to thank members of the FASTBUS Software Working Group for many useful suggestions, particularly, Connie Logg for her ideas on Routing.

References

1. ´FASTBUS SYSTEM MANAGER SOURCE CODE´, S. Deiss, SLAC, 1/1/81.
2. ´FASTBUS SYSTEM SOFTWARE´, S. Deiss, SLAC, 6/28/80.
3. ´FASTBUS SYSTEM MANAGER DEMONSTRATION´, S. Deiss, SLAC, 11/1/80.
4. ´FASTBUS DEMONSTRATION SYSTEMS´, Paffrath et. al., SLAC-PUB-2835, Oct. 81.
5. Comments from R. Pordes, S. Gannon, and D. Hanssen at FNAL, 1982.
6. ´A FASTBUS Controller Module Using a MULTIBUS MPU´, S. Deiss, NSS, 1982.
7. ´UCSD Pascal User´s Manual´, SOFTECH Microsystems, San Diego, CA, 1980.
8. ´FASTBUS...Tentative Specification´, U.S. NIM Committee, 6/7/82.