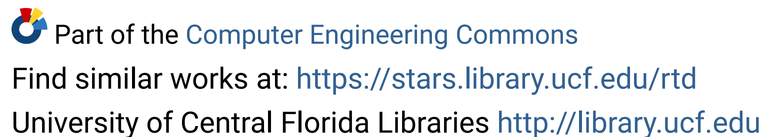
University of Central Florida
STARS

Retrospective Theses and Dissertations

1986

A Task Manager for a Multiprocessor Computer System

Peter J. Ingraham
University of Central Florida



Part of the Computer Engineering Commons
Find similar works at: https://stars.library.ucf.edu/rtd
University of Central Florida Libraries http://library.ucf.edu

This Masters Thesis (Open Access) is brought to you for free and open access by STARS. It has been accepted for inclusion in Retrospective Theses and Dissertations by an authorized administrator of STARS. For more information, please contact STARS@ucf.edu.

STARS Citation
Ingraham, Peter J., "A Task Manager for a Multiprocessor Computer System" (1986). *Retrospective Theses and Dissertations*. 4903.
https://stars.library.ucf.edu/rtd/4903

A TASK MANAGER
FOR A MULTIPROCESSOR COMPUTER SYSTEM

BY

PETER J. INGRAHAM
B.S.E.E., Northeastern University, 1983

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Engineering
in the Graduate Studies Program of the
College of Engineering
University of Central Florida
Orlando, Florida

Fall Term
1986

ABSTRACT

The advent of the 32-bit microprocessor has sparked the manufacture of a number 32-bit single-board computers (SBCs). These SBCs rival performance of minicomputers for a fraction of the cost, and a number of them conform to standard 32-bit bus structures. A 32-bit multiprocessor computer system can be created by connecting two or more SBCs on a 32-bit bus. This computer system has the potential of providing many times the power as a single-processor computer system as well as a significant reduction in cost.

The Multiprocessor Task Manager is designed to efficiently distribute software tasks, schedule their execution, and manage data flow between them, such that an effective and reliable multiprocessor is realized.

ACKNOWLEDGEMENTS

I wish to thank Paul Hardin and Ken Lambert of General Electric Company in Daytona Beach, FL for their contributions in the development of the Task Manager. Paul Hardin is also the author of the multiprocessor's operating system software.

TABLE OF CONTENTS

LIST OF FIGURES v

INTRODUCTION 1
Background 1
The Cost of High Performance Systems 5
The Frame 1 Multiprocessor 9
Task Management 10

CHAPTER I, TASK MANAGER OVERVIEW 13
Data-Flow Architecture 13
Program Abstraction 13
Levels of Abstraction, Levels of Precedence 14

CHAPTER II, TASK MANAGER SETUP 15
Data-Flow Graph 15
Lex and Yacc 19
Other Setup Functions 20

CHAPTER III, TASK CONTROL GRAPH AND TASK SEQUENCER . . 23
TCG Traversal 23
Effect of Tokens on Traversal 23
TCG Nodes 23
Pointers to Lists of Pointers 24
Method of Traversal 28
Tasks and Data 29
Off-Board Data Handling 31
Sending Data Off Board 32
Receiving Off-Board Data 33
Queues for Context Switching and More about TCG Traversal 34

CHAPTER IV, SYSTEM EVENT MONITOR 37
Interrupts (Events and Exceptions) 37
Processor Interrupts 37
Virtual Interrupts 38
SEM ISRs and Their Operation 39

CHAPTER V, CONCLUSIONS 44

APPENDIX, LEX AND YACC SPECIFICATIONS 46

REFERENCES 51

LIST OF FIGURES

Figure 1. Block Diagram of Typical CIG System 4

Figure 2. A Typical Multiprocessor Computer System 8

Figure 3. System Software Description 17

Figure 4. Symbolic Representation of DFG 18

Figure 5. Task I/O Information Embedded in Source Code 22

Figure 6. Task Control Graph Node Types 25

Figure 7. Task Control Graph Linkage 26

Figure 8. Task Control Graph Tables 27

Figure 9. System Event Monitor - Master SBC 42

Figure 10. System Event Monitor - Typical Slave SBC. 43

INTRODUCTION

Background

Simulators provide a relatively inexpensive means of training pilots for a variety of vehicles, ranging from tanks to jet fighters. In most modern simulators, Computer Image Generation (CIG) is used for the visual portion of the simulator. A typical CIG system consists of a general-purpose front-end processor, and special-purpose Image-Generator (IG) hardware.

The front-end processor is referred to as the Frame 1 subsystem. The IG consists of two subsystems, called Frame 2 and Frame 3, respectively. The subsystems bear these names because their processing time is based on video frame time, or one thirtieth of a second (33.33 milliseconds). This is the rate at which a video projector or display must be updated in order for the human eye not to detect flickering. A frame time consists of two field times, each one sixtieth of a second (16.67 milliseconds). At the end of the first field time, the even raster lines (0, 2, 4, ..., 1022) of the display are updated, and at the end of the second field time the odd lines (1, 3, 5, ..., 1023) are updated. Figure 1 shows a typical CIG system.

Basically, Frame 1's job is to track the three-dimensional position of objects moving through a geometric database. The database consists of the environment (fields, hills, trees, etc.) that is being flown or driven in, and resides on a disk pack whose drive is connected to Frame 1. In the case of tracking a pilot's position, Frame 1 must calculate where the pilot is in relation to the environment according to velocity, roll, pitch, and yaw inputs (usually from a joystick). From velocity and acceleration calculations, it must also predict where the pilot is heading, and where he will be by the time the information is displayed at the output of Frame 3 (approximately 80 milliseconds later). Frame 1 loads the database memory of Frame 2 from disk with the environmental data corresponding to the pilot's current position, along with data corresponding to where he might be in some short time, based on his current velocity and acceleration. At that same time, Frame 1 also passes to Frame 2 the flight dynamics (roll, pitch and yaw) data and the three-dimensional position of the pilot in relation to the environment.

Frame 2 and Frame 3 are special-purpose parallel-pipeline processors. Frame 2 does all the coordinate transformations required to map three-dimensional space into two-dimensional projector space. It also orders the faces (polygons which the database and moving models are composed

of) in the pilots field of view. The ordering is done in relation to the pilot's position, and the direction he's facing. This data is passed on to Frame 3, which adds color and texture to the environment and moving-model data, and then displays the picture. All the action described above must be performed on a per field (16.67 millisecond) basis in each of the three subsystems.

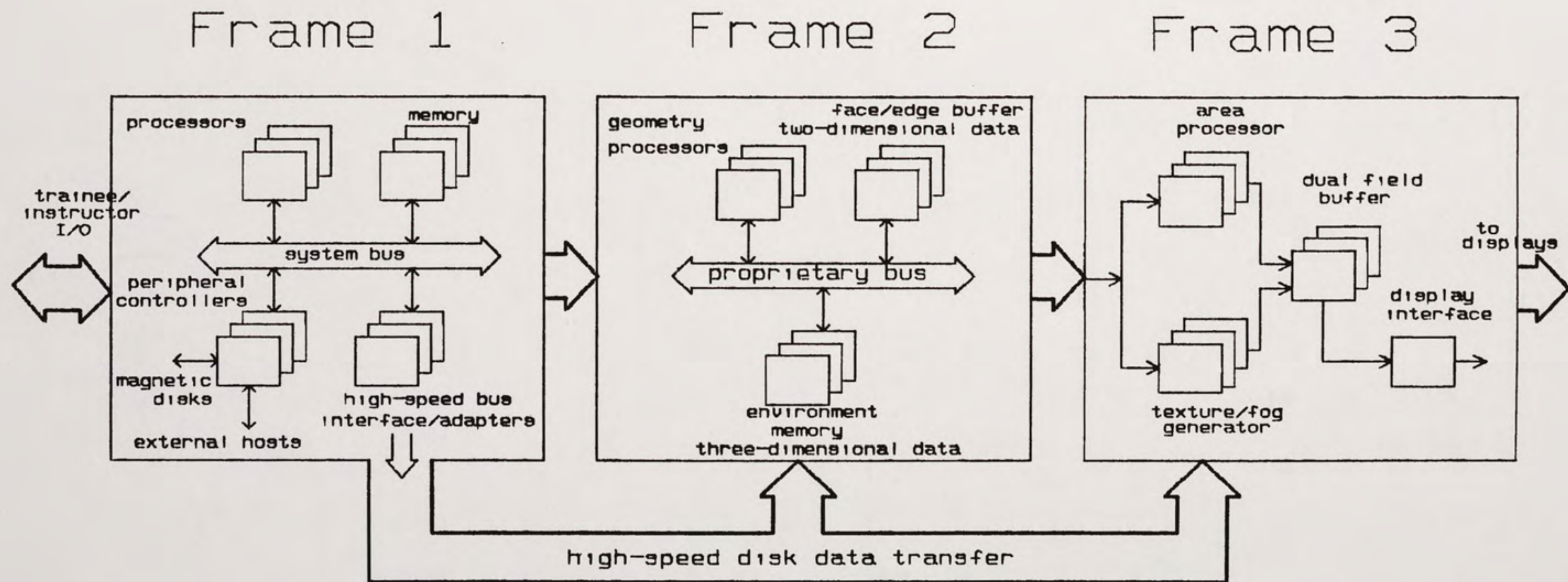


Figure 1. A Typical CIG Block Diagram.

The Cost of High Performance Systems

Traditionally, Frame 1 functions have been performed by a single-processor von Neumann machine, typically a Gould SEL 32/9705 or a Digital Equipment Corporation VAX 11/780. The SEL is rated at approximately 3.2 million instructions per second (MIPS), 3.7 MIPS with a floating-point accelerator board, and the VAX is rated at approximately 1 MIPS. The SEL costs about $300,000 and the VAX costs about $200,000.

The advent of the 32-bit microprocessor has brought about the development of 32-bit single-board computers (SBCs) that rival, and even exceed in some cases, the processing power of the superminis described above. For example, the Motorola 32-bit 68020 microprocessor is rated at approximately 2.5 MIPS*.

Motorola has also developed an extremely fast floating-point coprocessor (FPC) chip, the 68881, to be used with the 68020. The 68881 can operate concurrently with the 68020, and provides double-precision floating-point multiplication in approximately 5 microseconds. The 68881 also supports simultaneous sine and cosine calculations in approximately 30 microseconds*.

---

* 16.67 MHz operation.

A third chip in this set is the 68851 memory-management unit coprocessor (MMUC), which enables a 68020-based board to operate as a 32-bit virtual machine (Beimes 1985).

Single-board computers that incorporate the 68020, the 68881, and 68851, along with 1 megabyte of main memory, have been developed at a customer cost of less than $5,000. Imagine a single-board computer with twice the processing power of the mighty VAX 11/780 for a fraction of the cost! These boards can support the UNIX[1] (System V or bsd4.2) operating system, for development applications, and a variety of real-time operating systems (VRTX, pSOS, and MTOS, to name a few) for real-time applications.

The 32-bit SBCs described above have been, and are being, developed to interface with the prevailing 32-bit standard bus structures, Motorola's VMEbus and Intel's Multibus II.[2] These buses are designed with multiprocessor systems in mind. Both buses can support up to 20 SBCs. Each single-processor board communicates with the others over the main 32-bit bus (parallel system bus, or system bus). The philosophy is that processing power can be increased by plugging more of these cheap SBCs into the system bus, and having them process in parallel. For

---

[1] UNIX is a trademark of Bell Laboratories.
[2] Multibus is a trademark of Intel Corporation.

instance, by plugging 10 boards into a system, with each board rated at 2 MIPS and costing $5,000, a 20-MIPS machine can be obtained for less than $100,000. (Currently, a 20-MIP IBM costs about $16,000,000.) Of course, it is impossible for all SBCs to process concurrently at all times, and program distribution and communication over the system bus are not trivial. In fact, the development of multiprocessor systems is still in its infancy. However, because of the ever-increasing availability of inexpensive 32-bit hardware, the trend toward multiprocessor machines is gaining a great deal of popularity from both producers and consumers.

Figure 2 shows a typical multiprocessor system.

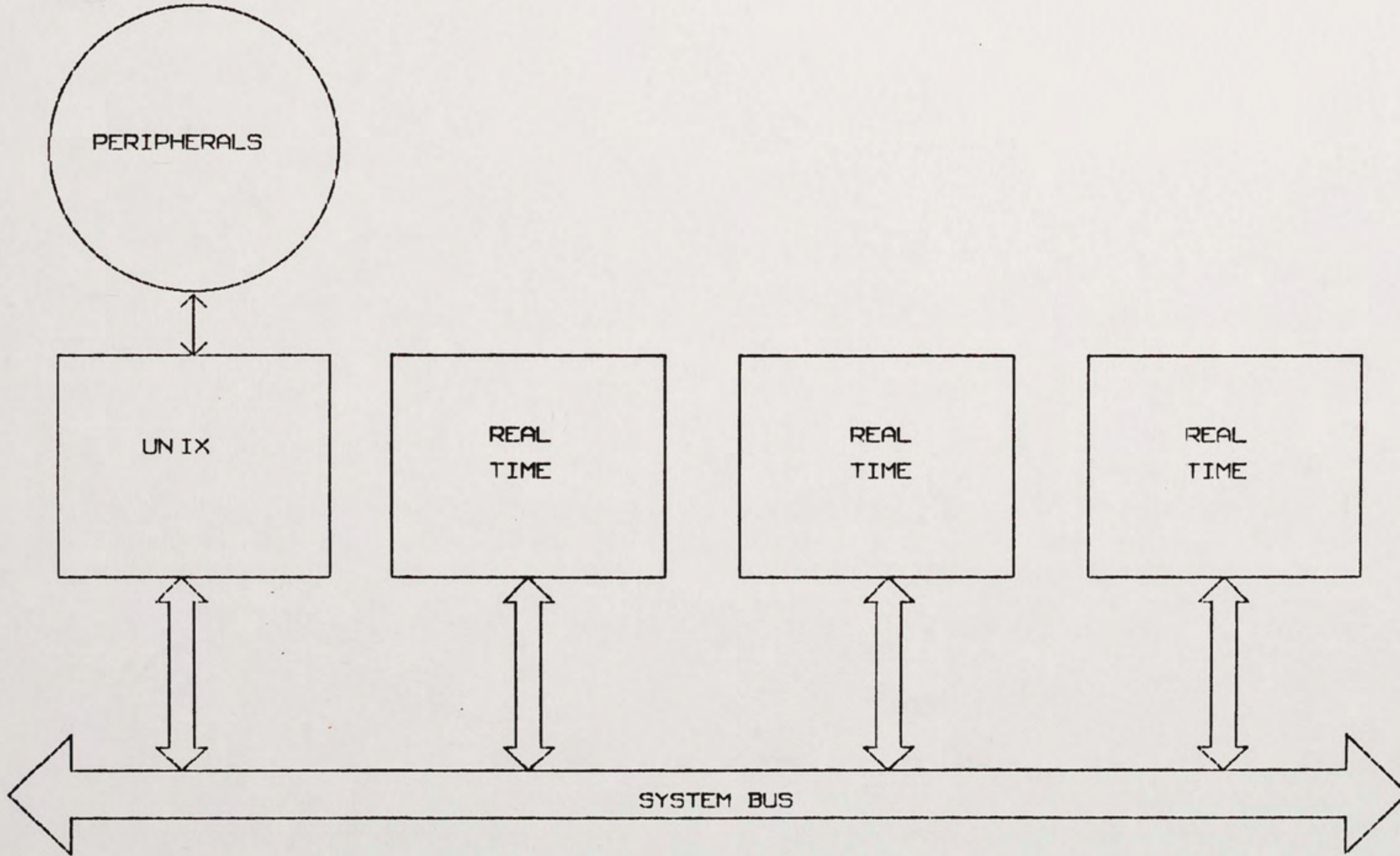


Figure 2. A Typical Multiprocessor Computer System.

The Frame 1 Multiprocessor

The Frame 1 subsystem performs many tasks that could ideally be done in parallel. Because of the potential parallelism of these tasks, it was decided that the development of a multiprocessor general-purpose computer system could yield a high-performance low-cost Frame 1 subsystem. This system could be used over the entire performance spectrum, with more power achieved by plugging in more boards. In order to develop the system in a timely fashion, it was decided that an off-the-shelf standard bus structure and a 32-bit SBC should be chosen.

After many months of research and debate, the standard bus structure chosen was Intel's Multibus II (Ingraham and Bloomer 1985). The heart of Multibus II multiprocessing capability is the Message Passing Coprocessor (MPC), a 149-pin grid array that resides on each SBC. The MPC is a hardware interface between the SBC and the system bus. It allows synchronous, high-speed 32-bit transfers of data (messages) over the system bus from one SBC to another, or one SBC to many, thus making global memory unnecessary for multiprocessing (Intel 1984).

The 32-bit Motorola chips (68020, 68881, and 68851) where chosen as the basis for the SBC.

The Frame 1 multiprocessor is configured with at least one UNIX SBC, to act as an operator interface, and to perform other non-realtime functions, such as system initialization. All real-time processing is distributed over those SBCs running a real-time operating system. Figure 2 shows a typical Frame 1 multiprocessor configuration.

Task Management

In order to achieve a multiprocessor that operates efficiently and reliably, there must be a facility for distributing software program modules (tasks), scheduling their execution, and managing data-flow between them, whether they are on the same SBC or not. The Multiprocessor Task Manager is such a facility.

The Frame 1 operating system software provides the means for transferring data from one SBC to one or more other SBCs over the system bus, for dynamically allocating memory on an SBC, and for generating various interrupts. Much of this software, especially that handling communication over the bus, is hardware specific. The Task Manager uses the facilities provided by the operating system software to get its job done. Theoretically, since it's shielded from hardware specifics, the Task Manager can be ported to another multiprocessor having entirely different

hardware, provided that the operating system facilities are similar.

The Task Manager has both centralized and distributed properties. The "master" SBC contains software for queuing and distributing scheduling information to its own tasks, and to tasks on other SBCs in the multiprocessor. The "slave" SBCs contain a subset of this scheduling software.

The Task Manager is divided into three major sections: Task Manager Setup, Task Sequencer, and System Event Monitor.

The Task Manager Setup function resides on a UNIX SBC. It is responsible for distributing tasks over the real-time SBCs, figuring out data allocation requirements and task data connections, and for distributing the Task Sequencer and System Event Monitor functions. All its work is performed during system initialization.

A Task Sequencer resides on every real-time board in the system. The Task Sequencer is responsible for executing tasks in the proper order, and for managing data to and from tasks. It does this by traversing a linked group of data structures called a Task Control Graph, which is constructed by the Task Manager Setup for each board in the system.

The System Event Monitor is distributed over the real-time boards in the system, and gives the Task Manager the "master/slave" flavor mentioned above. The System Event Monitor software on the slave SBCs responds to asynchronous events, and sends associated information to the master SBC. The System Event Monitor software on the master synchronously distributes event information to all real-time SBCs in the system. In this way it provides "dynamic" altering of task execution order, as will be detailed later.

The Task Manager is being developed on a VAX 11/780 under the ULTRIX operating system, using the C programming language. ULTRIX is almost identical to Berkeley UNIX (bsd4.2), but also has some VAX specific tools, such as DECNET copy (dcp).

CHAPTER I, TASK MANAGER OVERVIEW

Data-Flow Architecture

Scheduling of Frame 1 task execution is performed using data-flow concepts. In a data-flow system, data is passed from a producer task to a consumer task when the producer generates the data. The consumer task does not request the data and does not begin execution until data arrives; data drives the processing. In a multiprocessor system, the data-flow approach precludes fetches to a global memory. On the other hand, the traditional demand-driven approach requires that the data is requested by the consumer task. In a multiprocessor system, this could require costly fetches to a global memory. A data-flow software architecture is thus conducive to multiprocessing.

Program Abstraction

Software in the multiprocessor is viewed in several levels of abstraction: system, application, process, and task. The system is at the highest abstraction level. It is composed of applications, which are at the next level of abstraction. Applications are composed of (related) processes, and processes are made up of groups of (related) tasks. Tasks are at the lowest level of abstraction, and perform all program execution in the multiprocessor.

The abstraction philosophy provides an effective means of organizing Frame 1 software in a modular fashion. Program modules at higher levels of abstraction are constructed from modules at lower levels, as described above. The system is more manageable, since details in lower-level modules are hidden from higher-level modules.

Levels of Abstraction, Levels of Precedence

Within each level of abstraction, there are levels of precedence. Precedence levels are associated with order of execution, dictated by data-flow and timing constraints. Two program modules in the same abstraction level are at different precedence levels if the data produced by one is directly or indirectly consumed by another. They are at the same precedence level if they could be executed in parallel; their order of processing by the same processor is arbitrary.

CHAPTER II, TASK MANAGER SETUP

A major goal in the design of the Frame 1 Task Manager is to do as much static (off-line) allocation and scheduling as possible. This approach reduces Frame 1 processing overhead.

Data-Flow Graph

A system Data-Flow Graph (DFG) is constructed off line during system initialization, using a high-level System Software Description, described below. The DFG is a complete map of possible program modules (systems, applications, processes, and tasks) that could be processed during on-line operation. It describes precedence and possible parallelism of these modules. Modules are represented by nodes, and the flow of data between them is represented by links between node structures.

The System Software Description provides information about the organization of program units in the system. It defines groupings of program units within those at higher levels of abstraction. The System Software Description also describes the data relationships between program modules. Figure 3 shows a simple example of a System Software Description. The module on the right-hand side of the arrow is dependent on the data generated by the module on the

left-hand side. Also shown in the figure is information about tokens. More will be said about tokens in the next chapter. Figure 4 shows a symbolic DFG constructed from the description in Figure 3.

```
System:
  Applications:                     --> App0.data0
                     App0.data4     --> App1.data4
                     App1.data9     -->
  App0:
    Tokens:
      0 VANILLA  TRUE
      1 ONE_SHOT FALSE
    Processes:                          --> Pro0.data0
                          Pro0.data2    --> Pro1.data2
                          Pro1.data4    -->
    Pro0:
      Tokens:
        2 VANILLA  TRUE
        3 VANILLA  TRUE
      Tasks:                              --> Tas0.data0
                            Tas0.data1    --> Tas1.data1
                            Tas1.data2    -->
    Pro1:
      Tokens:
        0 VANILLA  TRUE
      Tasks:                              --> Tas2.data2
                            Tas2.data3    --> Tas3.data3
                            Tas3.data4    -->
  App1:
    Tokens:
      NO_TOKENS
    Processes:                          --> Pro2.data4
                          Pro2.data8    --> Pro3.data8
                          Pro3.data9    -->
    Pro2:
      Tokens:
        4 VANILLA  TRUE
      Tasks:                              --> Tas4.data4
                            Tas4.data5    --> Tas5.data5
                            Tas4.data6    --> Tas6.data6
                            Tas5.data7    --> Tas6.data7
                            Tas6.data8    -->
    Pro3:
      Tokens:
        0 VANILLA  TRUE
        4 VANILLA  FALSE
      Tasks:                              --> Tas7.data8
                            Tas2.data9    -->
```

Figure 3. System Software Description.

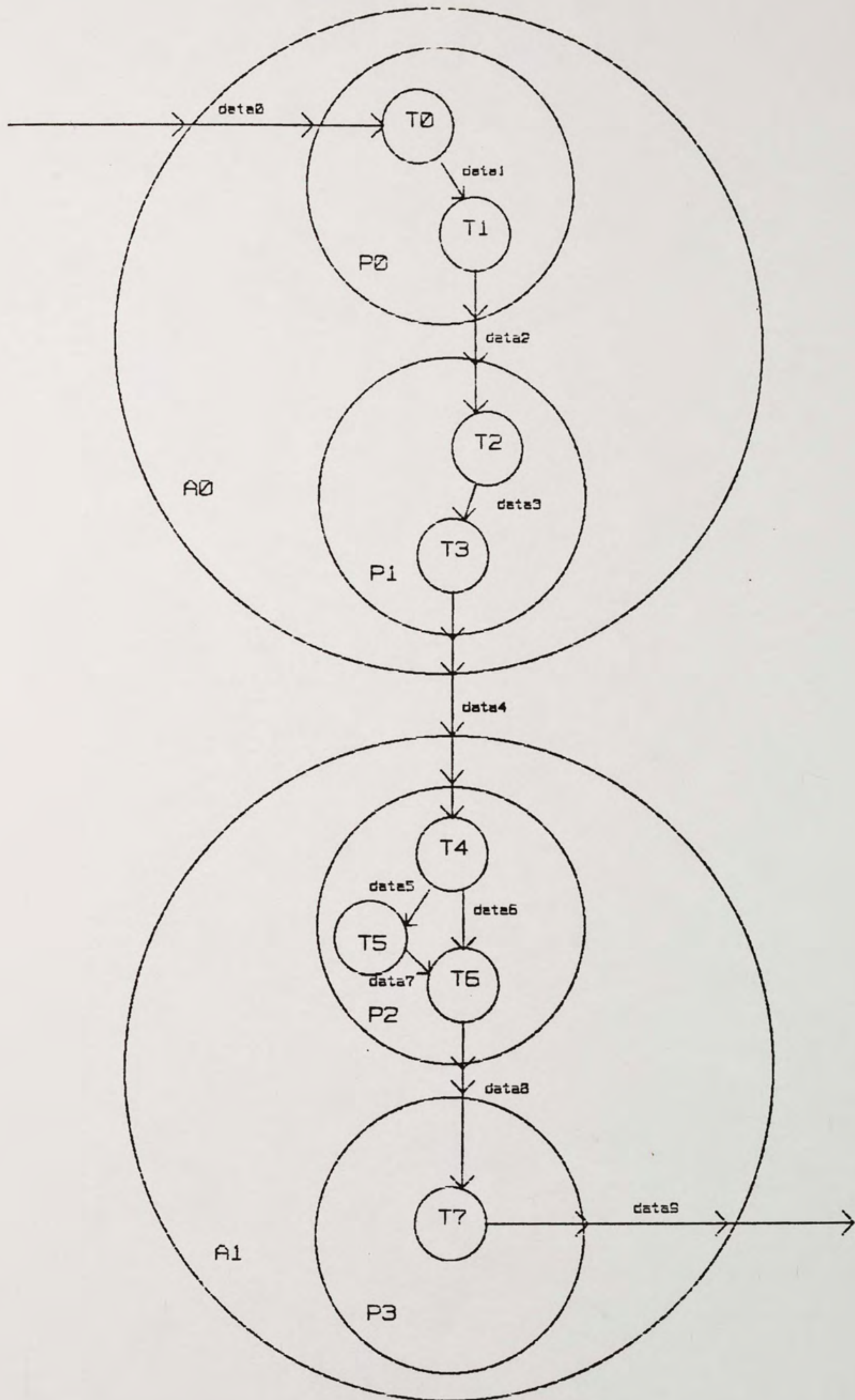


Figure 4. Symbolic Representation of DFG.

Lex and Yacc

The UNIX tools Lex (a Lexical Analyzer Generator) and Yacc (Yet Another Compiler-Compiler) are used to parse the System Software Description and build the DFG data structures. Both Lex and Yacc are program generators. Although Lex and Yacc can generate code of several languages, they generate C code for this project.

Lex generates programs that recognize input strings and partition the strings into matching expressions. When an expression is matched, a user-specified action is performed. All the Lex user needs to do is write a Lex specification that lists the expressions to be matched and, in C code, the action to be performed. There is an action for each expression (Lesk and Schmidt 1978).

Yacc provides a way of imposing structure on program input. A Yacc specification is very similar to a Lex specification. In fact, Lex was developed specifically for use by Yacc (although it can be useful in its own right). Basically, Yacc specifies rules to be followed. When a rule is followed, the Yacc-generated code performs an action (Johnson 1978).

Yacc uses Lex to match strings from a text file, such as the System Software Description. On matching a specified text sequence, Lex returns a token (not to be confused with

the tokens mentioned above). Yacc applies the token toward a rule. When all the tokens for a rule are gathered, Yacc performs an action, such as adding another member to a list of data structures.

Other Setup Functions

A System Resource Table (number and types of boards) is also compiled during system initialization. The DFG is applied to the Resource Table to produce a Task Control Graph (TCG) for each real-time board in the system. Task Sequencers assigned to each real-time board traverse the TCGs during Frame 1 on-line processing. The goal is to take advantage of parallel processing, while minimizing interprocessor communication.

Another operation performed during system initialization is allocation of task data memory for the real-time boards. This operation requires knowledge of the system resources, the TCG, and the input/output (I/O) data information provided in the source code.

Figure 5 shows an example of the I/O information embedded in the source code. Lex and Yacc are used to extract the information. Lex keys on the symbol "/*$" to start parsing. Upon matching this symbol, Lex returns the token CONTROL_BEGIN to Yacc, which gets ready to start collecting the information into data structures. The task's

name is collected on Lex's matching of TASK_NAME, inputs are collected after IN_LIST, outputs after OUT_LIST, and include file names after IO_INCLUDE_LIST. Comments are preceded by "...", and parsing stops when "*/" is matched. Because the embedded information is between "/*" and "*/", it is completely ignored by the C compiler (Kernighan and Ritchie 1978).

The include files are used to learn structure size information for each input and output encountered. This enables proper allocation of data for each task in each TCG. The Appendix contains the Lex and Yacc specifications used to gather the embedded information into data structures.

```
/*h- transatt.c     1816 asc  2-may-1986 10:16 am  ingraham */
Transform_Attitude(inc, inp, outc, outp)
/* Declare parameters, standard interface */
    int inc,    outc;
    int *inp[], *outp[];

/* TASK CONTROL INFORMATION */
/*$task_information  ... a "control" comment starts with /*$
    TASK_NAME:          Transform_Attitude;

    IN_LIST:            ... I/O lists must have unique names
        rpy_packet,     ... I/O data structure declarations
        attitude,       ...  are required to be in include files

    OUT_LIST:
        rpy_prime,
        altitude,

    IO_INCLUDE_LIST:  ... assumes released software directory
        /usr/users/ingraham/include/RTUattitude.h,
*/
#define RPY_PACKET              0
#define ATTITUDE_PACKET         1

#define RPY_PRIME               0
#define ALTITUDE                1
{
/* LOCAL DECLARATIONS */
    struct RPY_packet           *pRPYpacket;
    struct attitude_packet      *pAttitude;
    struct RPYprime_packet      *pRPYprime;

/* BEGIN */
    /* A ficticious but useful example of how a task
       works in the data-flow concept */

    /* Process RPY data */
    /* Get pointers */
    pRPYpacket = (struct RPY_packet *)      inp[RPY_PACKET];
    pRPYprime  = (struct RPYprime_packet *) outp[RPY_PRIME];

    /* Do something useless */
    pRPYprime->roll = pRPYpacket->roll + pAttitude->jitter.roll;
    pRPYprime->yaw  = pRPYpacket->yaw  + pAttitude->jitter.yaw;

    /* Etc. */
    .
    .
    .
}
```

Figure 5. Task I/O Information Embedded in Source Code.

CHAPTER III, TASK CONTROL GRAPH AND TASK SEQUENCER

TCG Traversal

Effect of Tokens on Traversal

Tokens provide a means of dynamically modifying traversal of a TCG by a Task Sequencer. Program modules that are not processed every field are assigned tokens to control whether or not they will be processed during a given field. These assignments are provided by the System Software Description, as shown in Figure 3.

During Frame 1 processing, token changing events that occur during a field are queued on a "master" real-time microcomputer board. At the start of the next field these token changing events are distributed to all real-time boards in the system as Token Change Requests (TCRs). These TCRs cause the corresponding tokens in the Token Memory on each board to be changed.

TCG Nodes

Figure 6 illustrates TCG nodes. There are 2 types of nodes in a TCG: those that represent systems, applications, and processes, and those that represent tasks. All nodes are associated with an ID number, which identifies the specific system, application, process, or task being processed. A system, application, or process node also has

a reference to tokens, if tokens are assigned to it, and a link to the next level of abstraction. A task node, since it represents the only program module that actually executes code, has a reference to the executable image and a reference to data requirements.

Pointers to Lists of Pointers

Figures 7 and 8 should be helpful references to the text that follows.

A pointer in each node references a list of pointers. The pointers in the list reference nodes at the next lower level of abstraction. Pointers that reference nodes at different levels of precedence are separated by a NULL pointer. The composite pointer list is terminated by 2 consecutive NULLs.

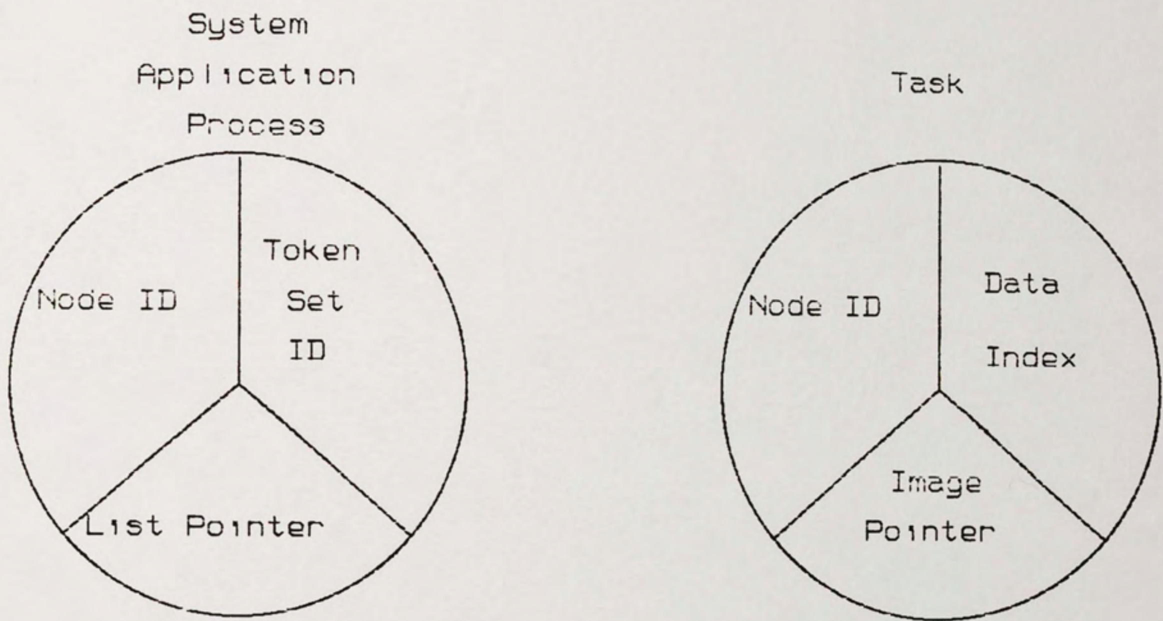


Figure 6. Task Control Graph Node Types.

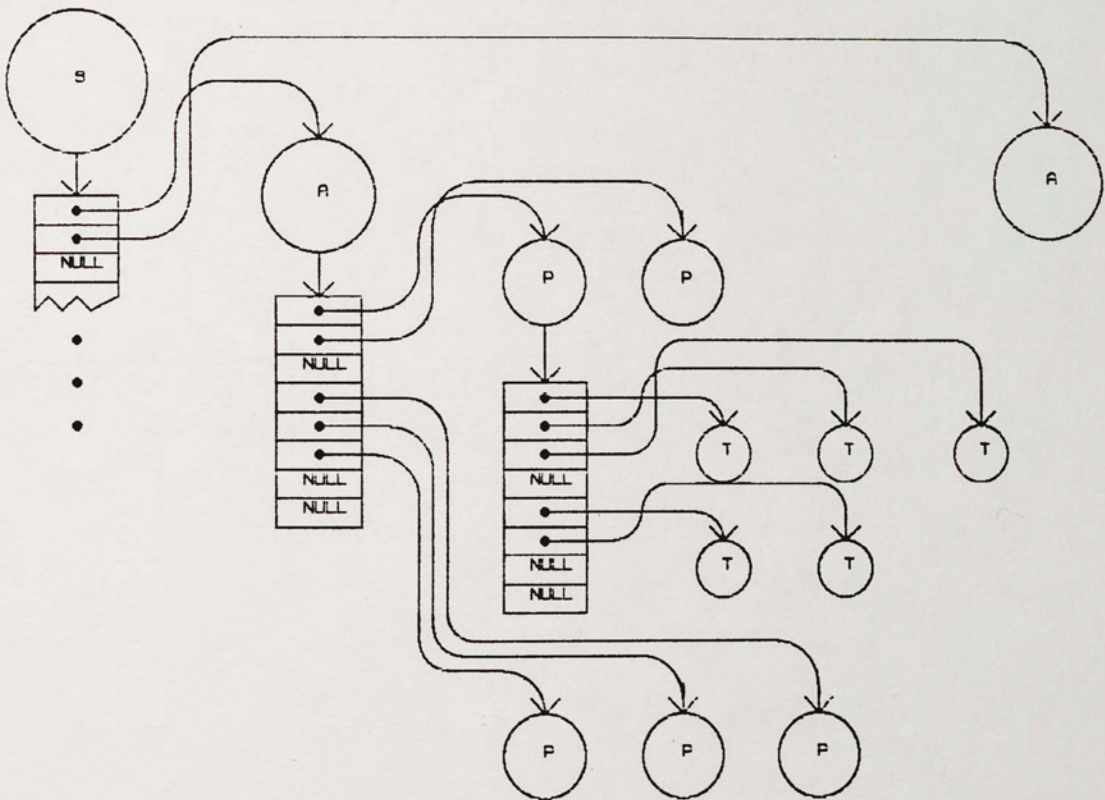


Figure 7. Task Control Graph Linkage.

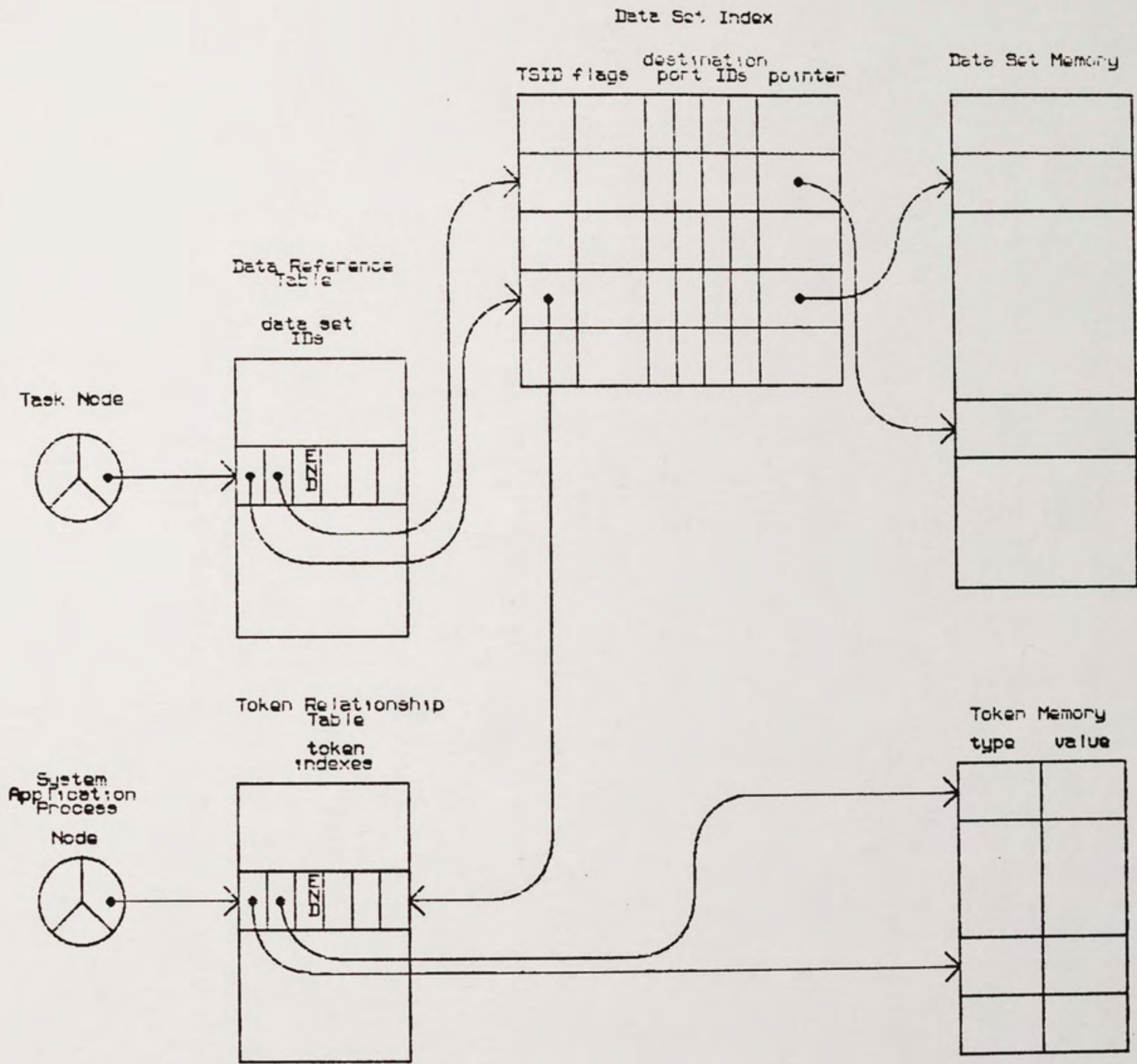


Figure 8. Task Control Graph Tables.

Method of Traversal

The TCG is traversed from higher levels of abstraction to lower levels of abstraction. TCG nodes are "processed" to determine the correct traversal path. Processing of system, application, and process nodes is similar, since their structures are the same. Task nodes require special processing, which includes passing data and control to the task's executable image. TCG traversal is performed by the Task Sequencer. Each TCG node, except for the system node, is referenced by a pointer in a pointer list, as described above. It is assumed in this paper that only one system node exists in each TCG.

For each System, Application, or Process (SAP) node encountered, the Task Sequencer performs processing as follows.

The node's token reference, or Token Set ID (TSID) is checked. If its value is NO_TOKEN, there is no token associated with the node, and processing continues immediately. If its value is zero, or some finite integer, it is used to index the Token Relationship Table (TRT). Each entry in the TRT consists of a list of indexes into the Token Memory (TM), followed by END_OF_LIST.

The TM entries are called tokens, and consist of two fields: a type field and a (binary) value field. The token type may be "vanilla" or "one_shot". A vanilla token must be deliberately reset. A one_shot token is reset by the Task Sequencer at the end of TCG traversal.

The value of each token indexed by the TRT is checked. If a FALSE (logic 0) token is encountered, processing of the SAP node ceases. The token is said to be "guarding" the node. The Task Sequencer increments the pointer to the pointer list and repeats the above procedure. If all the referenced tokens are TRUE, processing of the SAP node continues.

In the next and final step of SAP node processing, the Task Sequencer obtains the pointer to the pointer list for the next level of abstraction. The system node points to an application node pointer list, each pointer in the list providing a reference to a node representing the applications assigned to this TCG. Each application node points to a list of process node pointers, and each process node points to a list of task node pointers.

Tasks and Data

The task node represents the lowest level of hierarchy in the tree. It is the only node type that has a reference to an executable image. Besides the image, the task node

also has a reference to the Data Requirements Table (DRT), shown in Figure 8. Each entry to the DRT contains a list of indexes (Data Set IDs - DSIDs) to the Data Set Index (DSI), followed by END_OF_LIST. Each entry in the DSI contains a Token Set ID (TSID), a flag field, a pointer to a location in the Data Set Memory (DSM), and an array of integers for off-board destination port IDs.

The TSID provides a means for determining the validity of data produced by tasks in potentially guarded processes and applications.

The flags in each entry to the DSI pertain mostly to off-board data. They include an "on-board" flag, a "source" flag, and a "ready" flag. The on-board flag, if logic 0, signals that the referenced data set going to or coming from the system bus. If it is logic 1, then the rest of the flags can be ignored. The source flag signals the off-board data direction: to the system bus if logic 1, or from the system bus if logic 0. The ready flag, if logic 1, signals that an off-board input has arrived.

In processing a task node, the Task Sequencer evaluates the data sets referenced by the node and attempts to execute the task.

If the data set is valid (there are no related tokens or all related tokens are TRUE) and ready, the pointer to the data set is collected into a task pointer vector. If the data set is invalid, the corresponding pointer in the pointer vector is assigned an "invalidity" value to signal the task that this pointer is invalid.

The resultant pointer vector, along with control, is passed to the task's executable image. When execution is complete, the Task Sequencer increments the pointer to the task node pointer list, and initiates processing of the next task node.

Off-Board Data Handling

Movement of data from a task on one board to a task on another board is handled by the Task Sequencer using the operating system software message functions. They include "msg_pcreate," "msg_psend," and "msg_pget."

Msg_create creates a port on an SBC. It requires as arguments the port ID number, the home SBC number, and the number of messages allowed in the port queue.

Msg_psend sends a port-to-port message over the system bus. It requires as arguments the destination port ID, a pointer to the message, and the size of the message in bytes.

Msg_pget gets the next port-to-port message, which the operating system's message facility places into dynamically allocated memory upon its arrival. Its input argument includes the port ID number, and a timeout value, indicating how long to wait for the message, in microseconds. Its output arguments include a pointer to the message data, a pointer to the size of this data.

During Task Manger Setup, each Task Sequencer is assigned a port ID number corresponding to the slot number of the SBC to which it is assigned. Upon arriving at its destination SBC, each Task Sequencer uses msg_pcreate to create the assigned port. A Task Sequencer uses this port to pass data between tasks residing on different SBCs.

Sending Data Off Board

As the Task Sequencer processes a task node, it must check the Data Set Index (Figure 8) to determine the status of each data set for that task. As detailed above, each entry to the Data Set Index (DSI) consists of a Token Set ID, a field of flags, a pointer to the data set memory location, and an array of integers for off-board destination port IDs. If the Token Set ID is zero, or all related tokens are TRUE, then the data set is valid and the Task Sequencer proceeds to evaluate the DSI flags. If the data set is to go off board, the Task Sequencer sets the corresponding DSI ready flag, and adds the data set pointer

to the task pointer vector.

When the task completes, the Task Sequencer checks the ready flags corresponding to each data set going off board. If ready is logic 1, the Task Sequencer passes the first destination port ID from the DSI and the data set pointer to msg_psend. The Task Sequencer indexes through the destination port ID array, passing each ID and the same data set pointer to msg_psend, until END_OF_ARRAY is reached.

The above procedure is repeated for each off-board data set associated with this task. When finished with this task, the Task Sequencer increments the pointer to the task node pointer list and initiates processing of the next task node.

Receiving Off-Board Data

When the Task Sequencer finds that a valid data set is from off board, it must check the data set's ready flag. If the ready flag is logic 0, then the data set has not yet been linked to. The Task Sequencer must determine if the data set has arrived at the port yet. It passes its port ID and a timeout value of zero (the Task Sequencer doesn't want to wait) to msg_pget. If msg_pget times out, the Task Sequencer pushes the pointers to the task and process pointer lists onto the Process Queue, as detailed in the next section, and proceeds to the next process in the

process pointer list. Otherwise, msg_pget sets the pointer to the dynamic memory location of the data set. The Task Sequencer copies this pointer into the DSI pointer field and the task pointer vector, and sets the ready flag to logic 1.

The above procedure is repeated for each off-board data set associated with this task. When finished with this task, the Task Sequencer increments the pointer to the task node pointer list and initiates processing of the next task node.

Queues for Context Switching and More about TCG Traversal

Queues are used to save the task and process pointers, and/or the application pointer, when the current task is waiting for off-board input data. The waiting task is thus queued while a ready task in another process in the current or another application is run. The result is better CPU utilization.

As stated above, each system node structure, application node structure, and process node structure has a pointer into a list of pointers to nodes at the next level of abstraction. During processing, pointers to the lists are incremented to keep track of the current application, process, and task, and the precedence-level boundaries at the process and application levels of abstraction.

If a task is required to wait for data, and there is another process at the same level of precedence as the current process, the process and task pointers are pushed onto the Process Queue. The pointer to the list of process pointers is incremented, and processing of the next process begins.

The above sequence is repeated until the last process at the current precedence level is handled (NULL is reached); the processes in which tasks are not waiting are processed, while the task and process pointers of those that have waiting tasks are pushed onto the Process Queue. If no processes at the current precedence level are queued, then processing of the first process at the next precedence level begins.

If, at the end of the current process precedence level, there are queued processes, and there is another application at the same precedence level as the current application, the application pointer and the pointer to the process list precedence boundary are pushed onto the Application Queue. The pointer to the list of application pointers is incremented, and processing of the next application begins. For each application at the current precedence level, there is an associated Process Queue; the number of Process Queues is equal to the maximum number of applications at the same level of precedence.

Once all applications at the current precedence level have been either processed or queued, the Application Queue is checked. If it is empty, the first application at the next precedence level is processed. If the queue is not empty, the first application on the queue is selected.

Processing begins for the waiting task in the first process on the selected application's Process Queue. If data is still not ready, the task and process pointers remain queued. If data is ready, the task runs, and the task and process pointers on the queue are invalidated. The pointer to the list of task pointers is then incremented, and processing of the next task begins.

After all processes in the Process Queue for the current application have been processed, the corresponding pointer to the process list precedence boundary is incremented, and the first process in the next precedence level is processed. If processes in this level of precedence become queued, then the application pointer remains queued, and the old pointer to the process list precedence boundary is replaced by the current one. When the end of the process pointer list is reached, and the Process Queue is empty, the next application in the application queue is selected. When the Application Queue is exhausted, processing begins on the first application at the next level of precedence.

CHAPTER IV, SYSTEM EVENT MONITOR

The System Event Monitor (SEM) is a collection of interrupt service routines (ISRs) that handles all events in Frame 1. These events include device interrupts, message interrupts, and virtual interrupts. The SEM enables dynamic altering of TCG traversal by providing ISRs that convert Token Changing Events (TCEs) into TCG token changes.

Because the SEM is distributed over the multiprocessor, it relies heavily on services provided by the operating system software. These services include a message facility for board to board communication over the system bus, which in Multibus II jargon is referred to as "message passing." The SEM ISRs, and their interaction with operating system services, are described below.

Interrupts (Events and Exceptions)

Processor Interrupts

Device interrupts and message interrupts are hard interrupts; that is, they activate interrupt lines to the microprocessor unit (MPU). Most device interrupts are routed through a Counter/timer parallel I/O unit (CIO), from which the MPU can read a vector number. The vector number addresses the processor interrupt-vector table, which contains 256 interrupt vectors. Interrupt vectors are

memory pointers used by the processor to fetch the starting addresses of various processor ISRs. Processor control is thus "vectored" to the appropriate ISR in response to an interrupt. Message interrupts are autovectored to the message ISR; that is, MPU internal logic generates the vector number, which is then used to determine the interrupt-vector address.

Virtual Interrupts

A virtual-interrupt mechanism, layered over the processor interrupt mechanism, is supported by operating system software. Virtual ISRs are invoked by processor ISRs either through a function call or using the vint_signal function provided by the operating system software. Virtual interrupts provide a way to "re-vector" a processor (hard or soft) interrupt.

Virtual interrupts initiated using the vint_signal function are message based and are directed or global. A directed virtual interrupt is "directed" to the message handler on the "master" microcomputer board. A global virtual interrupt is broadcast to the message handlers on all (real-time) boards in the system. Both directed and global virtual interrupts require a virtual vector number as the vint_signal argument.

Many important SEM operations are performed by virtual ISRs. The virtual interrupt mechanism allows a processor ISR, such as the message ISR, to remain general purpose. An example of message ISR re-vectoring follows.

A virtual-interrupt packet arrives at the MPC. The MPC interrupts the MPU and control is vectored to the message ISR. The message ISR determines the message type, and uses the virtual vector number to vector control to the corresponding virtual ISR.

SEM ISRs and Their Operation

The SEM provides mechanisms for distributing TCRs in response to Token Changing Events (TCEs), such as initiation of a special-effects sequence from a cockpit. A special-effects sequence is a sequence of programs that run very seldomly, such as a sequence that shows the movement of a projectile fired from a tank toward a target, or a sequence that depicts the explosion upon impact of the projectile.

All real-time boards in the system have a copy of the SEM, which may be different from board to board. The "master" board, which is connected to Frame 3 (IG) master timing syncs, contains SEM ISRs responsible for collecting and distributing TCRs. Figure 9 illustrates control and data flow between SEM ISRs on the master board. Figure 10

does the same for a typical slave board. SEM operation is outlined below.

A hard interrupt is translated into an Asynchronous Token Changing Request (ATCR) by the Token Changing Event Translator (TCET) ISR. The TCET uses the vint_signal function to pass the ATCR, as a directed virtual interrupt, to the master real-time board. When the ATCR packet is received by the master's MPC, the message ISR vectors control to the ATCR receiver (AR). The AR stores the ATCR on the Event Queue. Control returns to the message ISR, and from from there, returns to the interrupted task or the Task Sequencer. A similar sequence takes place for all TCEs that occur during a field.

On arrival of Field Sync (a hard interrupt) to the master real-time board, the contents of the Event Queue are distributed as Synchronous Token Changing Requests (STCRs) to all real-time boards in the system. This operation is performed by the STCR Distributor (SD) ISR, which uses a global virtual interrupt to handle distribution. The field count and the Commit Strobe are included as part of the STCR packet. The field count comes before the STCRs and the Commit Strobe comes after.

As the STCRs are received, the message ISR on each real-time board vectors control to the STCR Receiver (SR). The SR moves the STCRs into a Token Update Buffer (TUB). On finding the Commit Strobe at the end of the STCRs, the SR calls the TUB-to-Token-Memory Copy (TTMC) function, which translates the contents of the TUB into tokens and moves these new tokens into the appropriate locations in the Token Memory.

Control is returned to the Task Sequencer, which will once more traverse the TCG. This time, perhaps, the process node(s) representing the special effects sequence initiated in the previous field will be unguarded, and the associated tasks will be executed.

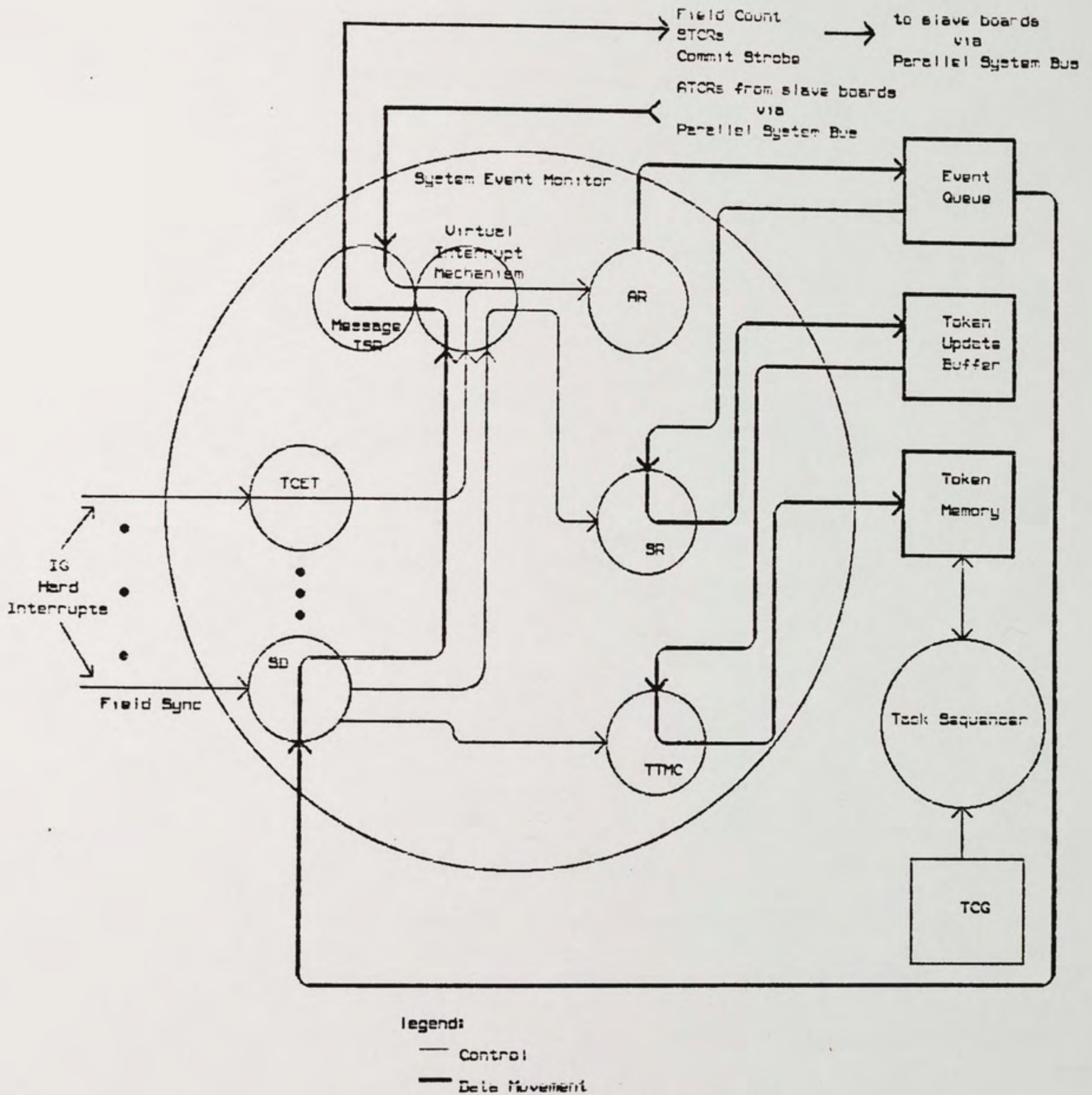


Figure 9. System Event Monitor - Master SBC.

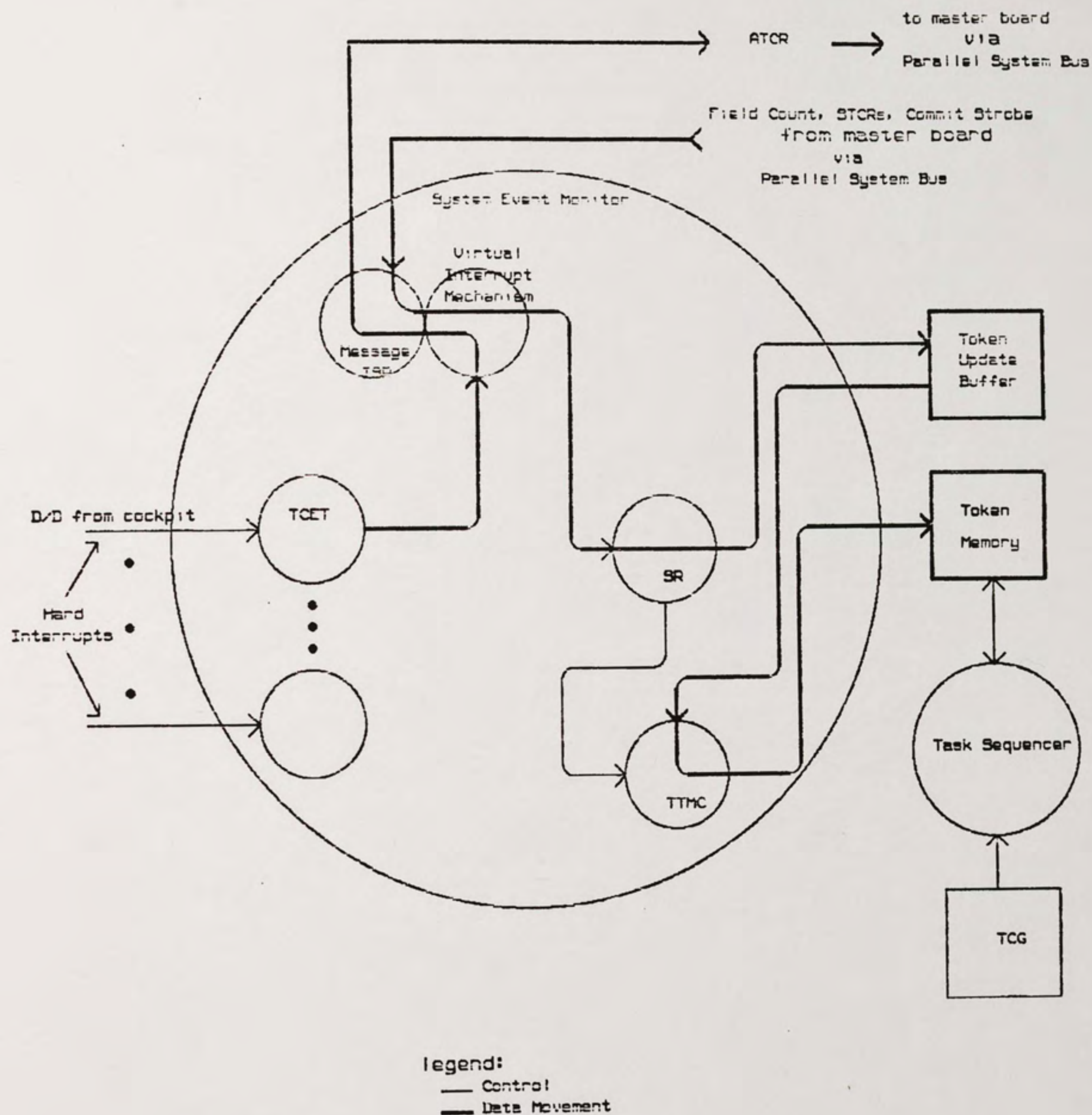


Figure 10. System Event Monitor - Typical Slave SBC.

CHAPTER V, CONCLUSIONS

Simulator functions that have been traditionally performed by single-processor superminicomputers can be performed by a multiprocessor computer system at a tremendous cost savings and performance gain.

The proliferation of powerful, inexpensive 32-bit microprocessors, standard bus structures, and the single-board computers based on them, has spawned a growing interest in the construction of multiprocessor computer systems (multicomputers).

Although multicomputer hardware is inexpensive and somewhat standard, integration problems exist that have never before confronted many computer architects. On a single-processor system, tasks are run sequentially to completion. Tasks that require data from other tasks are simply run after those tasks.

In order to achieve a multiprocessor that operates efficiently and reliably, there must be a facility for distributing tasks, scheduling their execution, and managing data flow between them, especially when they reside on different processors. The Multiprocessor Task Manager is such a facility.

The Task Manager is efficient, reliable, and portable. Efficiency and reliability are achieved by performing most resource allocation, memory allocation, and task scheduling off-line during system initialization. This insures more reliable data connections and reduces on-line processing overhead. Portability is achieved through the Task Manager's use of hardware-specific facilities provided by the operating system software. Since it is shielded from hardware specifics, the Task Manager can be ported to another multiprocessor having entirely different hardware, provided the operating system facilities are functionally similar.

APPENDIX, LEX AND YACC SPECIFICATIONS

```
%{
#include "test.h"
#include "y.tab.h"

extern TASK *p_task;

BOOL CONTROL = FALSE;

STRING  Add_Member();
MEMBER *p_mem = NULL;

STRING s_yyin = "stdin";
%}

control_begin             "/*$"
control_end               "*/"
list_member               [a-zA-Z0-7_./]+/(,|;)
whitespace                [ ]
comment                   "...".*

%%
{control_begin}           {
                            CONTROL = TRUE;
                            return(CONTROL_BEGIN);
                          }
{control_end}             {
                            if (CONTROL)
                            {
                              CONTROL = FALSE;
                              return(CONTROL_END);
                            }
                          }
TASK_NAME:                if (CONTROL) return(NAME_KEY);
IN_LIST:                  if (CONTROL) return(IN_KEY);
OUT_LIST:                 if (CONTROL) return(OUT_KEY);
IO_INCLUDE_LIST:          if (CONTROL) return(INC_KEY);
{list_member}             {
                            if (CONTROL)
                            {
                              yylval.member_name
                                = Add_Member(yytext);
                              return(LIST_MEMBER);
                            }
                            else
```

```
                                        ;
                                  }
{comment}                         ;
                                  |

%%

STRING Add_Member(buffer)
char buffer[];
/* Get member from stack or add member to it */
{
  MEMBER *p_tmp;

  /* initialize temporary pointer */
  p_tmp = NULL;

  FOR_ALL(p_tmp, p_mem)
    if (strcmp(p_tmp -> s_name, buffer) == 0)
      break;

  if (p_tmp == NULL)
    {         /* allocate space for p_tmp */
    p_tmp = (MEMBER *) malloc(sizeof(MEMBER));

    p_tmp -> s_name =
      (STRING) malloc(strlen(buffer) + 1);
    strcpy(p_tmp -> s_name, buffer);

    p_tmp -> p_next = p_mem;
    p_mem = p_tmp;

    return(p_tmp -> s_name);
    }
  else
    return(p_tmp -> s_name);
}

yyerror(buffer)
char *buffer;
{
  fprintf(stderr, "%s line %d:
                  s_yyin, yylineno, buffer);
}

yywrap() { return(1); }
```

```
%{
#include <ctype.h>
#include "test.h"

VOID Add_List_Struct();
TASK *p_task_tmp = NULL;
LIST *p_list_tmp = NULL;
MEMBER *p_mem_tmp = NULL;
extern TASK *p_task;
extern TASK *p_task_list;

char s_error[512];
%}

%start control

%union
{
  STRING member_name;
  TASK   *p_tsk;
}

%token NAME_KEY IN_KEY OUT_KEY INC_KEY
%token CONTROL_BEGIN CONTROL_END
%token <member_name> LIST_MEMBER

%type  <p_tsk> tasks task lists list
%%

control :tasks
                { p_task_list = $1 -> p_next; }
        ;

tasks   :task
                { $$ = $1; }
        |tasks task
                { $$ = $2; }
        ;

task    :CONTROL_BEGIN lists CONTROL_END
            {
                p_task_tmp =
                  (TASK *) malloc(sizeof(TASK));
                p_list_tmp =
                  (LIST *) malloc(sizeof(LIST));
                p_mem_tmp =
                  (MEMBER *)malloc(sizeof(MEMBER));

                p_task_tmp -> p_next = p_task;

                p_task = p_task_tmp;
```

```
                p_task -> p_list = p_list_tmp;
                p_task -> p_list -> p_member =
                  p_mem_tmp;

                $$ = p_task;
            }
        ;

lists   :NAME_KEY list
            {
                p_task -> p_list -> keyword =
                  "TASK_NAME";
                Add_List_Struct();
                $$ = p_task;
            }
         IN_KEY  list
            {
                p_task -> p_list -> keyword =
                  "IN_LIST";
                Add_List_Struct();
                $$ = p_task;
            }
         OUT_KEY list
            {
                p_task -> p_list -> keyword =
                  "OUT_LIST";
                Add_List_Struct();
                $$ = p_task;
            }
         INC_KEY list
            {
                p_task->p_list -> keyword =
                  "IO_INCLUDE_LIST";
                $$ = p_task;
            }
        ;

list    :LIST_MEMBER
            {
                if (p_task == NULL)
                {
                  p_task_tmp =
                    (TASK *) malloc(sizeof(TASK));
                  p_list_tmp =
                    (LIST *) malloc(sizeof(LIST));
                  p_mem_tmp =
                    (MEMBER *)malloc(sizeof(MEMBER));
                  p_task = p_task_tmp;
                  p_task -> p_list = p_list_tmp;
                  p_task -> p_list -> p_member =
                    p_mem_tmp;
```

```
                    }
                    p_task -> p_list -> p_member -> s_name =
                      $1;
                    $$ = p_task;
                }
              |list LIST_MEMBER
                {
                    p_mem_tmp =
                      (MEMBER *) malloc(sizeof(MEMBER));
                    p_mem_tmp -> p_next = p_task -> p_list
                      -> p_member;
                    p_task -> p_list -> p_member = p_mem_tmp;
                    p_task -> p_list -> p_member -> s_name
                                                    = $2;
                    $$ = p_task;
                }
              ;
%%

VOID Add_List_Struct()
{
  p_list_tmp = (LIST *) malloc(sizeof(LIST));
  p_mem_tmp = (MEMBER *)malloc(sizeof(MEMBER));

  p_list_tmp -> p_next = p_task -> p_list;

  p_task -> p_list = p_list_tmp;
  p_task -> p_list -> p_member = p_mem_tmp;
}
```

REFERENCES

Beimes, Robert. M68000 Family... The Performance Standard for Microprocessors. Motorola High-Technology Seminar Series, 1985.

Ingraham, Peter, and Bloomer, John. System Bus Evaluation, Multibus II versus VMEbus. Daytona Beach, FL: General Electric, Simulation and Control Systems Department, 1985.

Intel. Multibus II Bus Architecture Specification Handbook. Santa Ana, California: Literature Department, Intel Corporation, 1984.

Johnson, Stephen C. Yacc: Yet Another Compiler-Compiler. Murray Hill, NJ: Holt, Rinehart and Winston, 1978.

Kernighan, Brian W., and Ritchie, Dennis M. The C Programming Language. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1978.

Lesk, M. E., and Schmidt, E. Lex - A Lexical Analyzer Generator. Murray Hill, NJ: Holt, Rinehart and Winston, 1978.