1-1-1984

The system structure for a hierarchical manufacturing cell controller.

Albert Dawson Baker

THE SYSTEM STRUCTURE FOR A HIERARCHICAL MANUFACTURING CELL CONTROLLER

by

Albert Dawson Baker

A Thesis

Presented to the Graduate Committee

of Lehigh University

in Candidacy for the Degree of

Master of Science

in

Industrial Engineering

Lehigh University

1984

This thesis is accepted and approved in partial fulfillment of the requirements for the degree of Master of Science.

5/10/84
(date)

[illegible]
Professor in Charge

Chairman of Department

Acknowledgements

I would like to thank Dr. Louis J. Plebani for all of the personal instruction and advice given me in the course of my thesis work. I would also like to thank my family for their support, encouragement, and inspiration during this project.

Table of Contents

List of Figures

Abstract

This thesis is concerned with the utilization of a real time operating system in the implementation of a hierarchical manufacturing cell controller. The general requirements for such a cell are discussed and analyzed in order to define a generic system output requirements list. Structured design and programming techniques are then used to develop a general system structure, and the functional relationship of the iRMX-86 operating system to the cell controller's hierarchy is discussed. File and database structures for the key system data elements are specified, and a system event scheduling algorithm is defined.

Preface

One of the current challenges in the area of manufacturing engineering is to develop increasingly sophisticated and capable controllers suitable for use in the shop floor environment. As the number of applications for which simple preprogrammed sequence control is adequate diminishes (due to the filling of such requirements), the need for increasingly intelligent controllers capable of interacting with their environments will become more apparent. Simultaneously, the decreasing costs of hardware coupled with the increases in the quality and quantity of software development tools tends to support further expansion of the use of intelligent controllers.

This thesis investigates the possibility of using off-the-shelf components, particularly the iRMX-86 real-time operating system and multibus based hardware, in the implementation of a hierarchical manufacturing cell controller. In many cases the fabrication of custom components for industrial applications tends to make the associated system prohibitively expensive. This is true for both hardware and software system components and provides the motivation for emphasis on the use of stock items.

1. The Manufacturing Cell

At the outset of the project it was decided to employ the proven structured technique of examining the system output requirements in order to define the necessary inputs. Since the emphasis of the exercise was on defining a general structure for a cell controller system, it became immediately necessary to conduct the system anaylsis at the highest possible level and in the most generic terms. This resulted in a list of rather general requirements for cell performance and a set of general assumptions about the cell.

1.1 Cell Requirements

The most basic statement about the purpose of the cell is that it should properly manufacture the specified product in the correct configuration. In order to achieve this objective, there are five basic functions that must be performed. First, the system must control the manufacturing process in a timely and accurate fashion. Timeliness and accuracy are priority considerations since they directly effect the system throughput and scrap rate. Second, the system must control the transfer of materials from place to place in the cell. As above, timeliness and accuracy are important here, but in addition the controller must ensure that material transfer devices, such as robot arms, do not collide with other cell members or each other. Third, the cell must be able to communicate, both with users and with other cell components. Additional communications, such as a link with a host, may be desirable. Fourth, the cell must do a certain amount of record keeping. Minimally, the cell will need to maintain a record of the current status and location of the cell components. In addition, there may well exist a requirement for archival record keeping of the cell's performance and activities. Finally, the cell should be able to inspect the goods that it manufactures. If this is done, it should prove possible to adjust the process control parameters to correct in-process defects and thereby lower scrap rates.

1.2 Assumptions About the Cell

In order to estimate the number and types of functions required by the cell it was necessary to make a few general assumptions about the cell's constituent components, and possible additions to the cell. These assumptions are as follows:

- cell will contain both machine tools and non-machining process stations
- cell will use proximity switches, and may contain a camera for visual sensing

- cell may contain one or more manual operations
- cell will contain conveyors and/or robot arms for material handlina

The immediate result of these assumptions is in recognition of the data types that need to be supported in order to meet the implicit requirements of the above assumptions. These are, basicly, that the cell can download APT code to the NC machine tools, and has available the process parameters for non-machining process stations (such as amp-hours for a plating operation). In addition, the cell controller must respond to external events such as those that could be detected by a proximity switch triggering an interrupt or perhaps a more direct interaction such as that which could be achieved through the use of a vision system. Finally, the cell controller must be able to coordinate the movements of materials through the system. This is not as difficult for the conveyors anticipated, but the issue becomes more complex when the robotic devices are considered. In this latter case, the controller must be aware of the locations of each of the robotic arm members, and continuously check their position and trajectory with reference to other solid objects in the cell. This dictates that the cell controller have extensive data concerning its geometric environment, and the capability of analyzing and acting on that data.

2. The Design Approach

A dedicated structured approach to the cell design was employed from the outset. There are several reasons for this. First, a structured approach facilitates the system analysis since the system is considered in a modular format, with each module being directly related to that above it in the hierarchy. This was eminently compatible with the stated objective of organizing a hierarchical system structure. In addition, the use of structured techniques allows the analyst to focus his attention on individual modules once the modules place in the hierarchy has been defined. An important consequence of this is that the number of "mind boggling" problems that the analyst/designer must consider is quickly reduced as the tasks are decomposed.

2.1 Use of structured techniques

A top down analysis of the controller was performed, with the focus of the effort on the overall structure of the system. As the analysis was performed, a Visual Table of Contents (VTOC) was constructed. This provided a common reference point, and provided a form of ongoing documentation. As the system evolved through design reviews, the VTOC was immediately updated to show the latest changes. This was an important consideration, since multiple programmers used the system structure chart as a reference. A valid modularization of the controller's tasks was the next objective in designing the system. In pursuit of this, the following rules of thumb were considered:

- module function statements consisted of one verb and one object;
- modules were located in a left to right sequence on the VTOC, generally indicating the data flow;
- modules were constrained to fifty pseudocoded statements or less to make them more managable and understandable.

When all the modules were entered in the VTOC, they were numbered according to a scheme which uniquely defined the location of each in the hierarchy.

Throughout the design of the modules, every effort was made to use only the three proper structures: sequence, selection, and iteration. This proved to be important later, during implementation, since this reinforced the modularity of the system structure and the application tasks. Figure 2-1 is a skeleton of the VTOC. (The complete chart is on file with Dr. Louis J. Plebani, Lehigh University Industrial Engineering Department.) Figure 2-1 shows the system hierarchy by module number. Note that jobs are circled and descendant tasks listed.

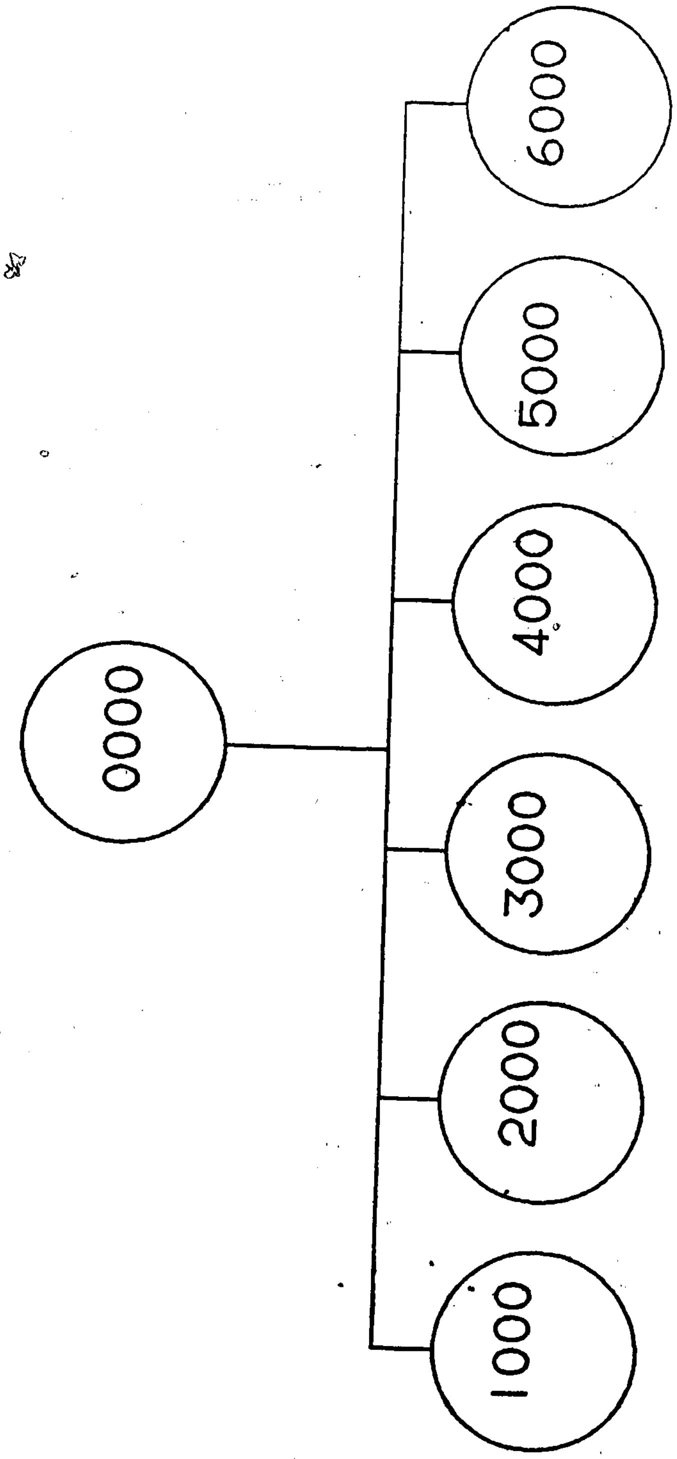

Figure 2-1: CMC System VTOC

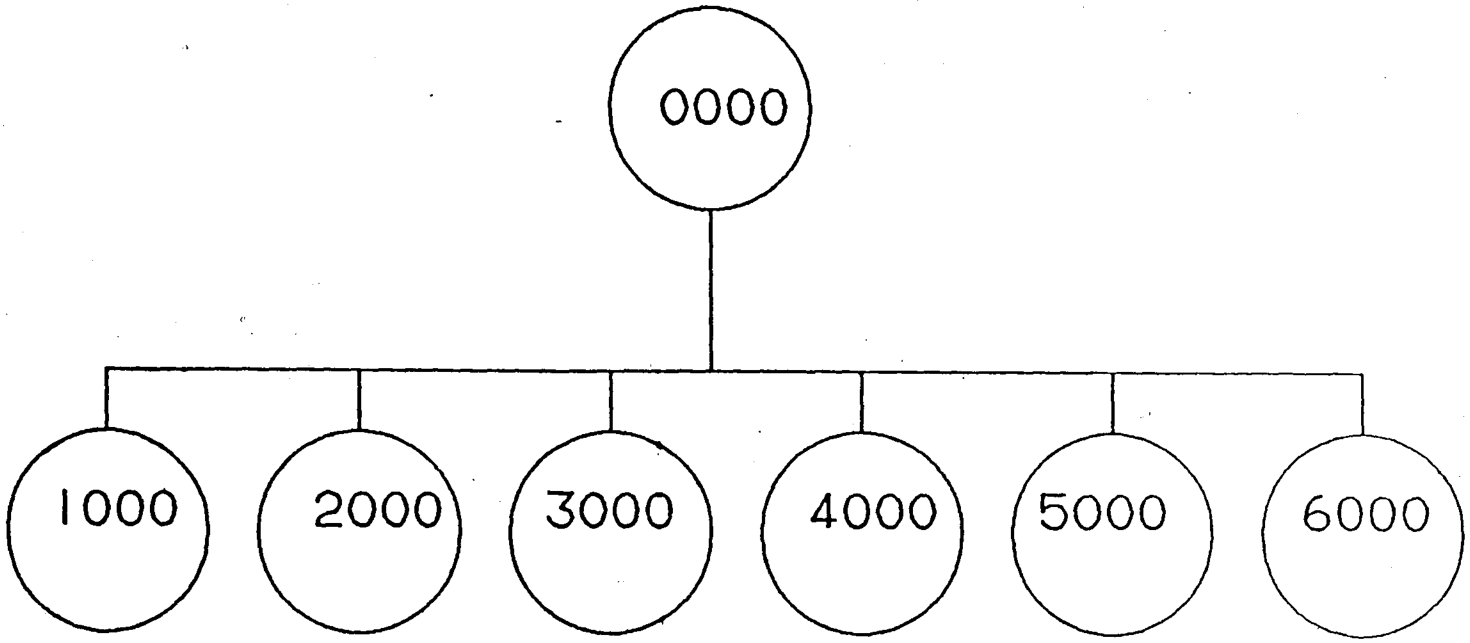

Figure 2-1: CMC System VTOC

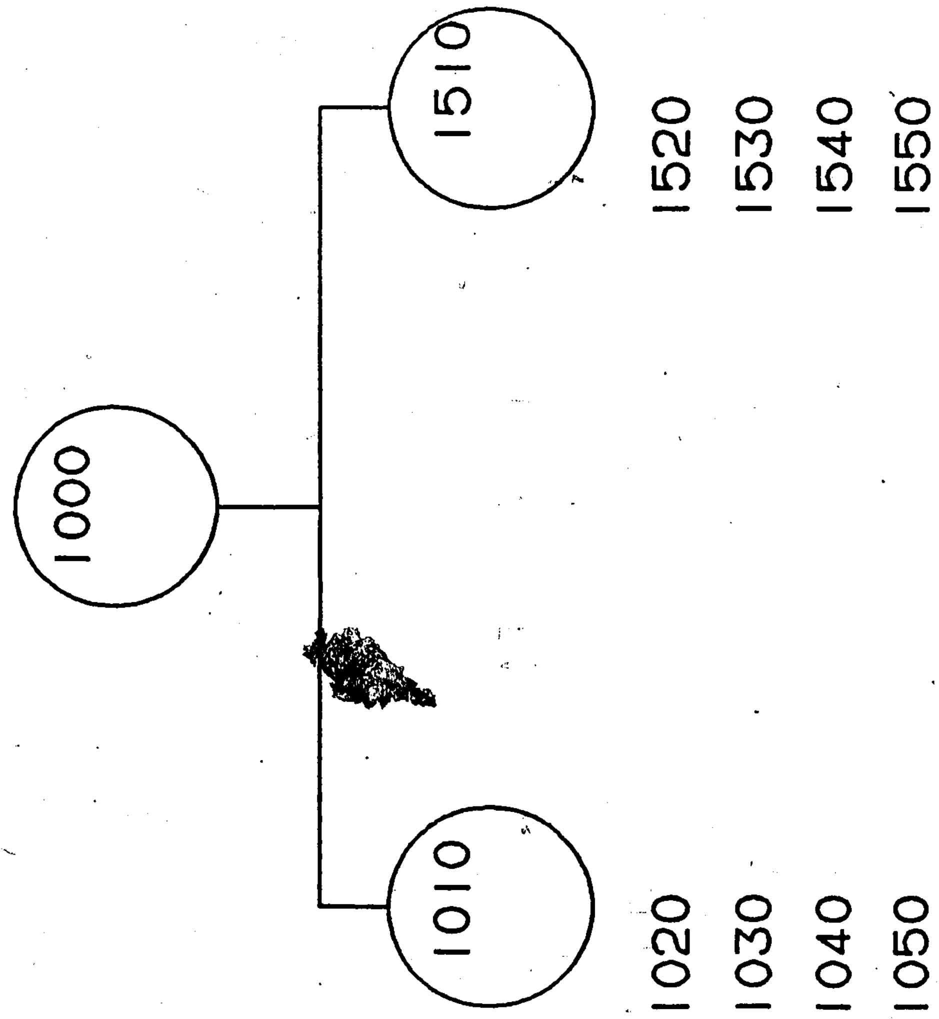

Figure 2-1, continued

7

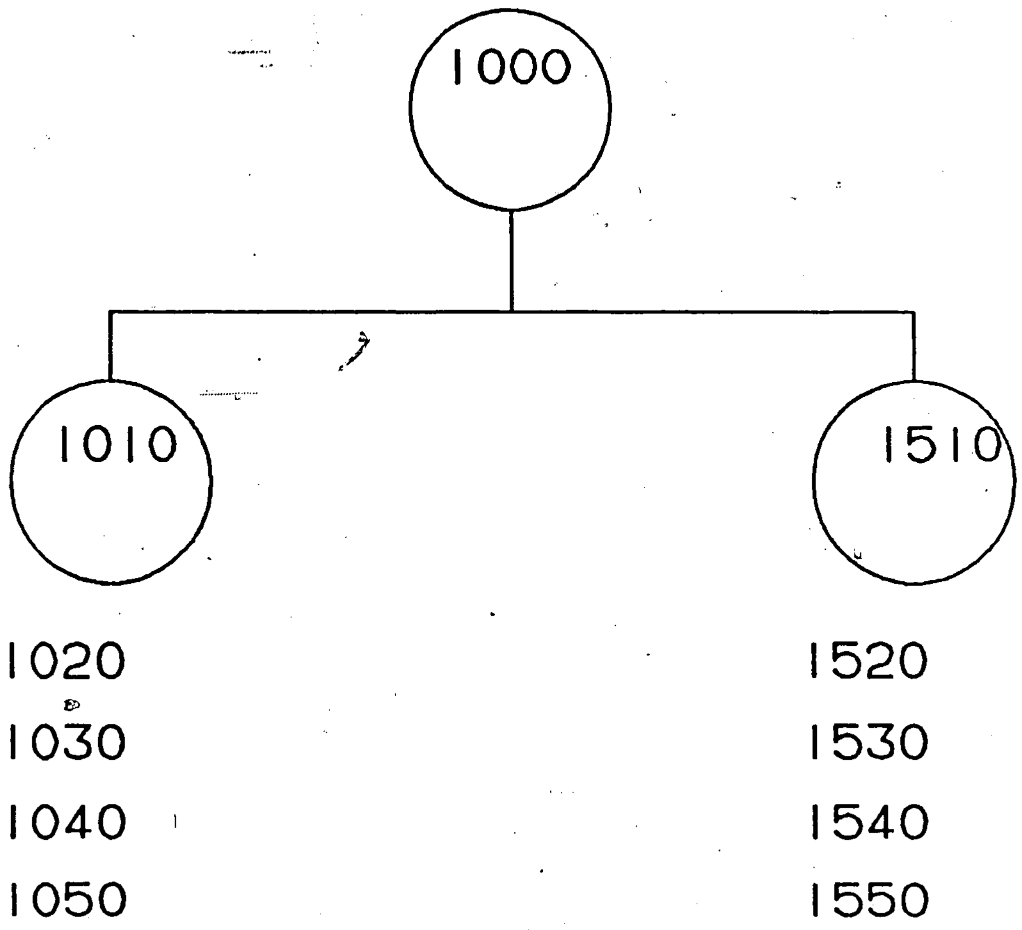

Figure 2-1, continued

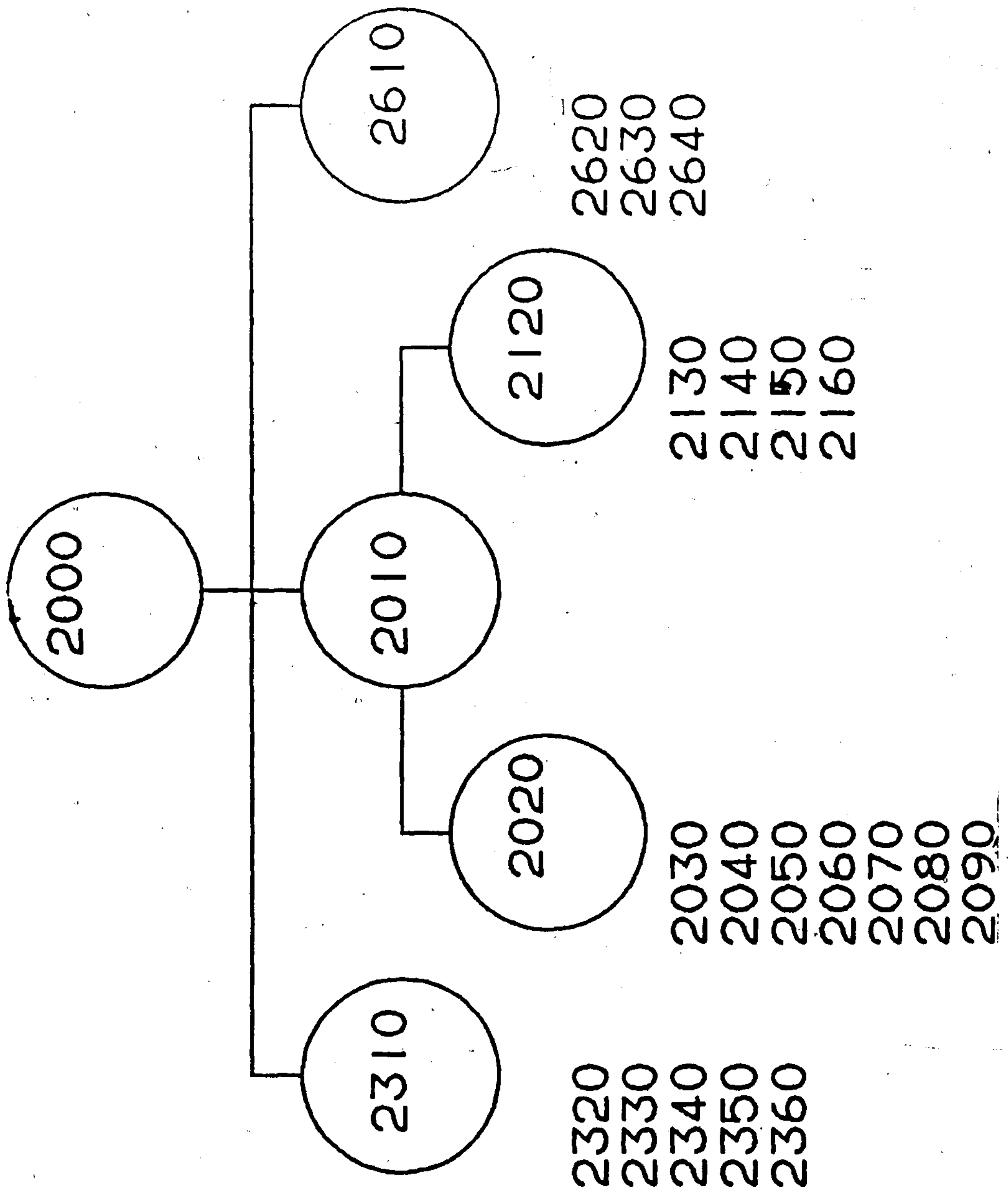

Figure 2-1, continued

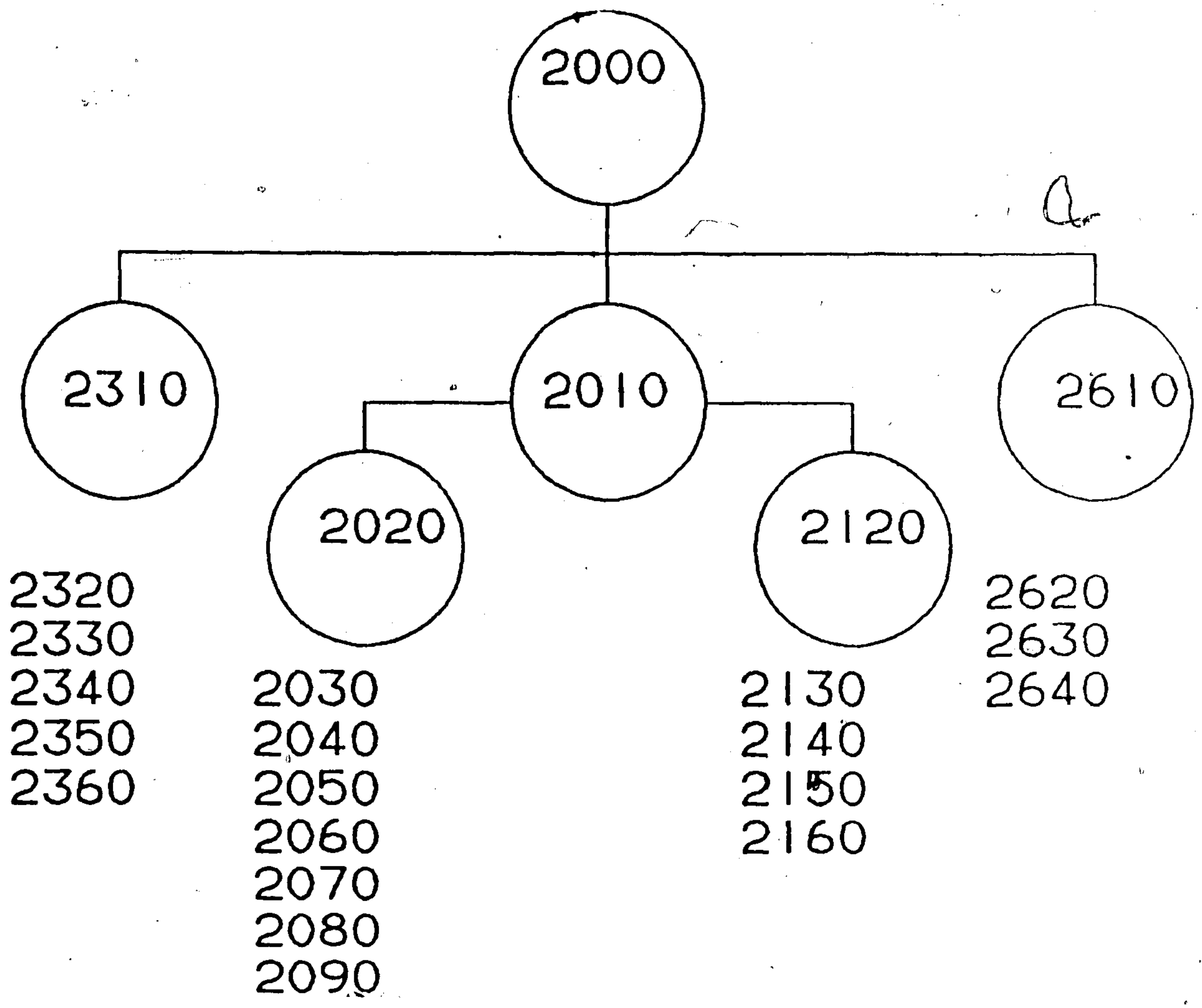

Figure 2-1, continued

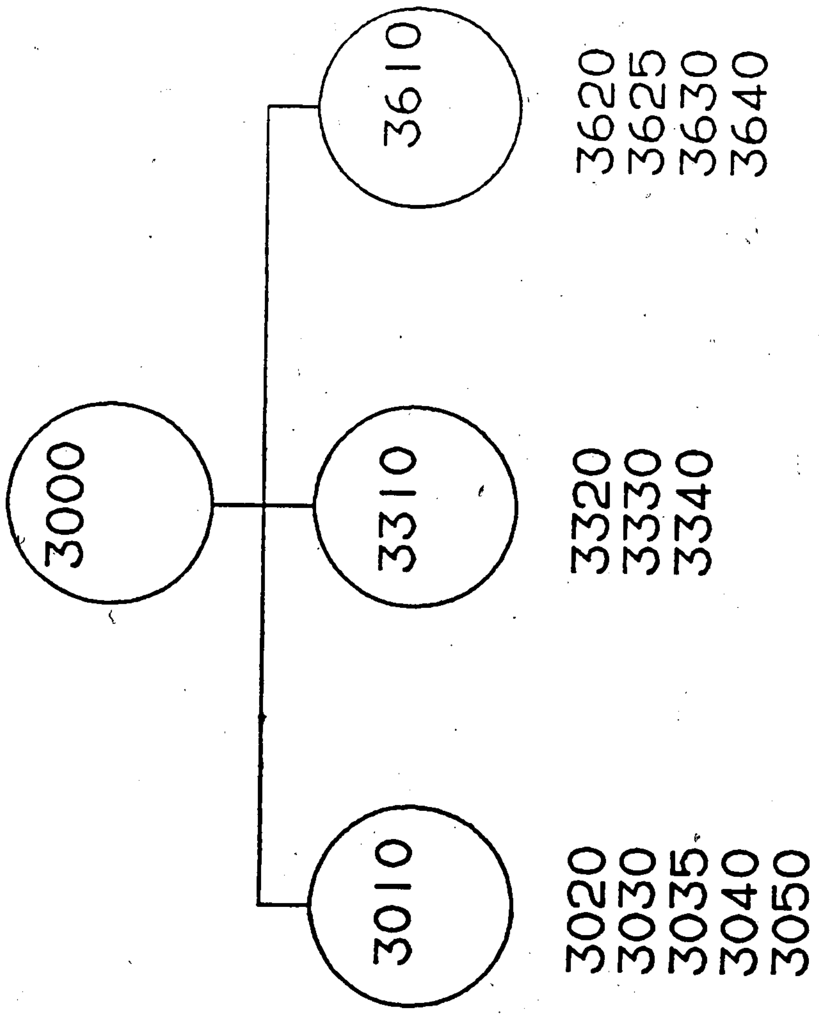

Figure 2-1, continued

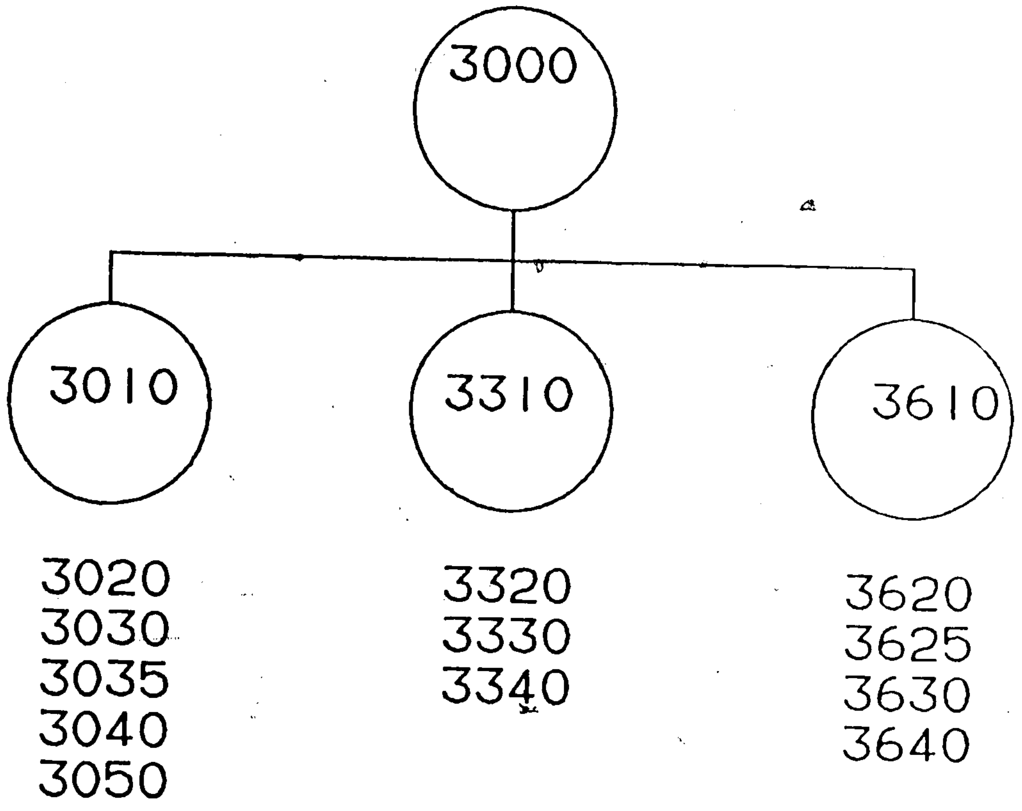

Figure 2-1, continued

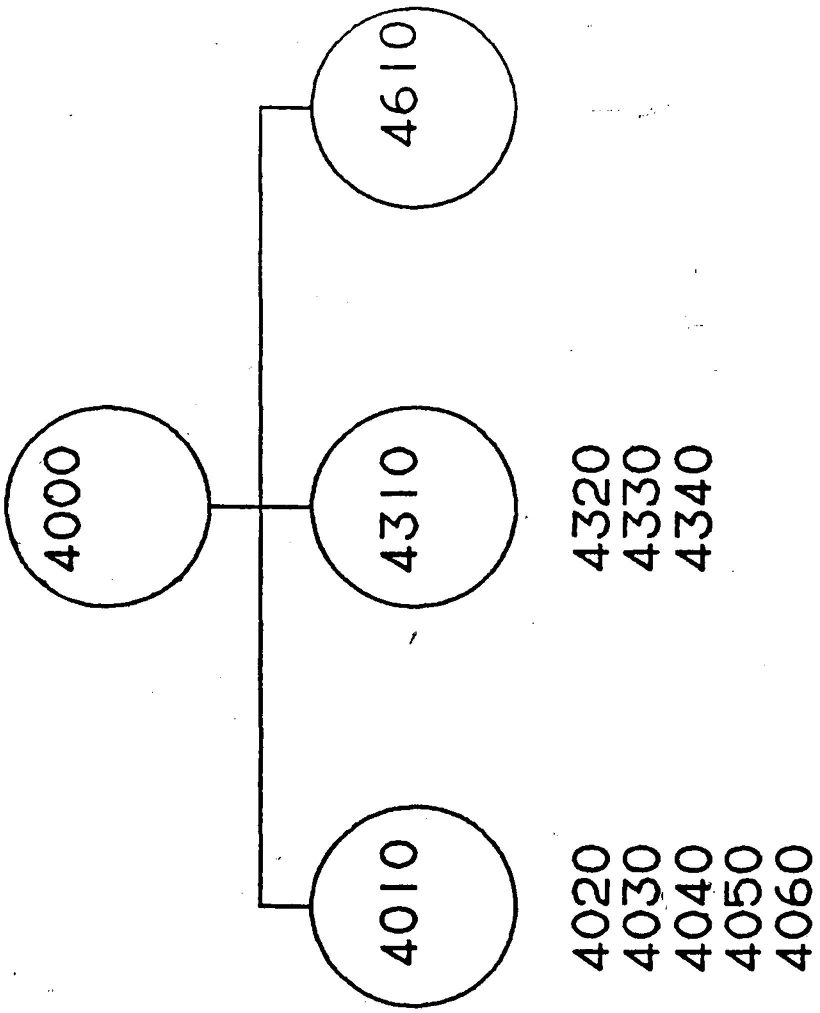

Figure 2-1, continued

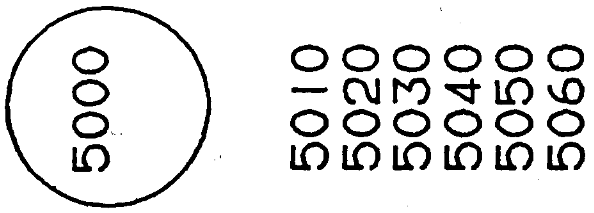

Figure 2-1, continued

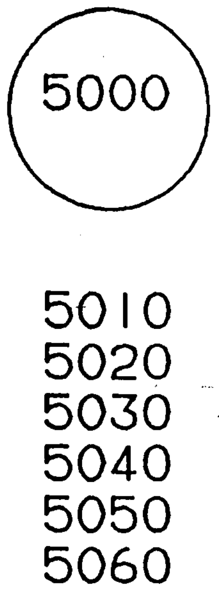

Figure 2-1, continued

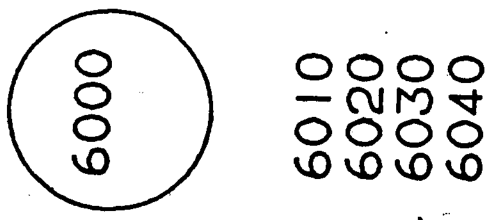

Figure 2-1, concluded

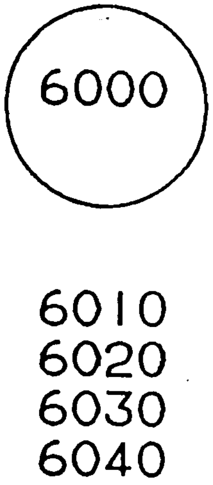

Figure 2-1, concluded

2.2 VTOC Review

When the chart was complete and numbered, it was reviewed for a variety of critical features. First, module function statement verbs were checked for consistency. Where inconsistencies were found, the verbs were changed to be compatable with the rest of the logical structure. Modules were then checked for independence. One reliable way to perform this check is to review the passing of control codes (not data) and ensure that the codes are passed up the hierarchy, not down the structure. An example is detection of an EOF, which should be done by a low level read module then passed up the hierarchy. The locations of the modules in the chart were checked again, this time to verify that they were properly subordinated. This is to make sure that all modules are properly related to their calling modules and they to theirs, and so on up through the hierarchy to the top level module. Next, the control span of each module was reviewed to see if it was in reasonable limits, i.e., between two and nine. If any module had less than two or more than nine subordinate modules, it was considered to be a potential design flaw, and reviewed for possible breakup or relocation in the system structure.

After the contents of the VTOC were satisfactorily reviewed, it was time to step back and review the overall structure. At this point it is of prime importance to check and make sure that all functions required to pursue the cells objectives are accounted for. Conversely, it is worthwhile to verify that all functions designed into the system contribute to the cells objectives. Those that did not make a contribution were considered good candidates for elimination.

3. iRMX-86: The Operating System

The use of iRMX-86 in the implementation example is interesting from a variety of viewpoints. It is a real time operating system oriented towards system development. It is user configurable for a variety of functions, and it supports user-written system-call routines. In addition, it is interrupt driven. Finally, it is is intended to be included in OEM products.

3.1 iRMX-86 System components

An iRMX-86 system is typically composed of the Nucleus, and a user selected assortment of service subsystems which can include the Basic Input/Output System (BIOS), the Extended Input/Output System (EIOS), the Application Loader (loader), the Human Interface (HI), the Universal Development Interface (UDI), and the application programs. (See Figure 3-1) It is the Nucleus that runs the system through its control over access to the processor and memory resources. Therefore, I will discuss the Nucleus separately from its servitors.

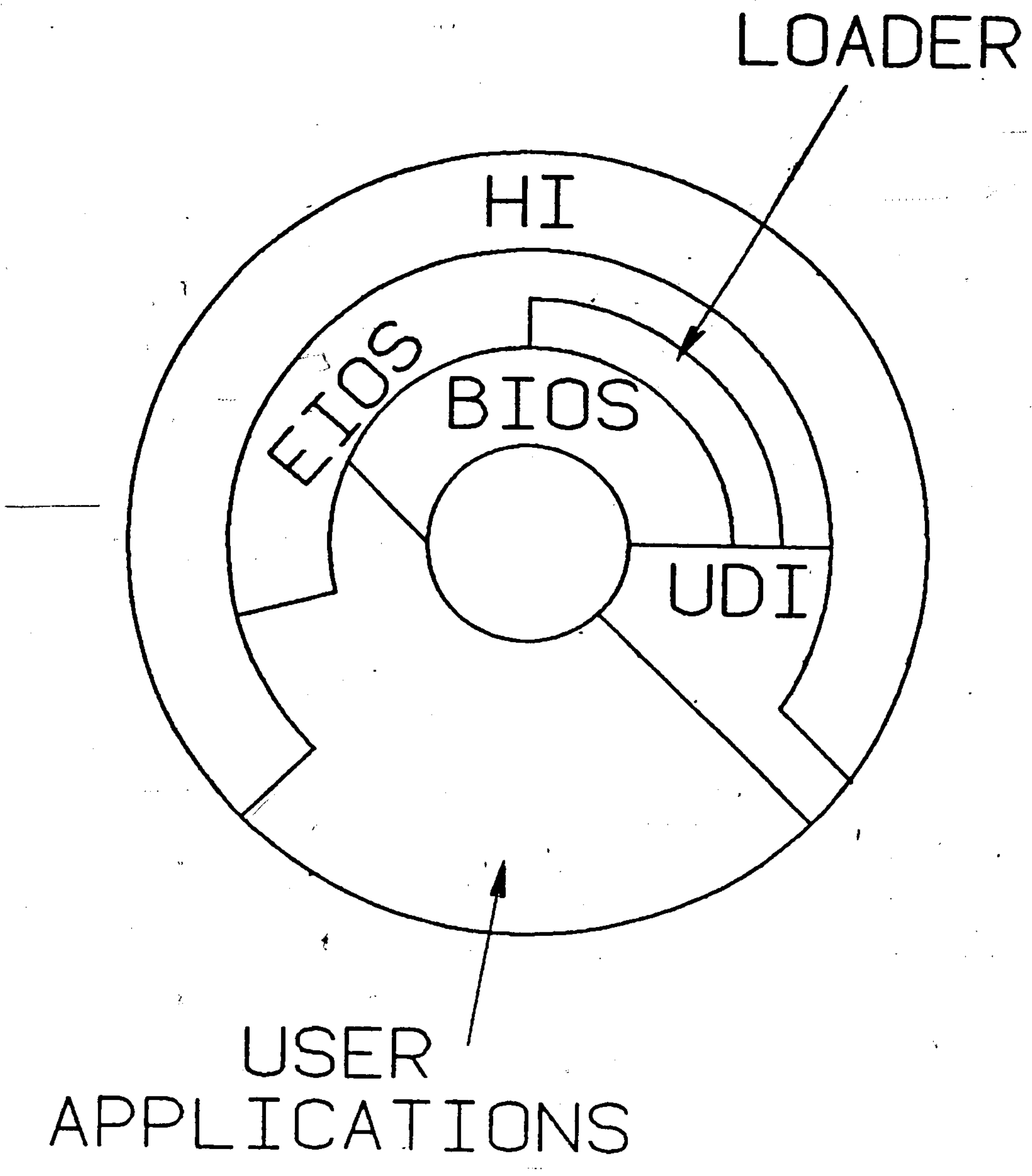

Figure 3-1: iRMX-86 System Organization

3.2 The Nucleus

The Nucleus is the core of any iRMX-86 system, responsible for the executive operation of the system. As such, it has five major activities under its jurisdiction. Not necessarily in order of importance, these are as follows:

- scheduling
- memory allocation
- inter-process communications
- response to external events: interrupts
- provision of basic building blocks to the service subsystems.

Before discussing any of the major functions, however, it is necessary to define the terminology and function of the basic building blocks mentioned above.

3.2.1 Definition of iRMX terms

These building blocks are referred to as "objects" , and consist of the following object types.

- Tasks
- Jobs
- Segments
- Mailboxes
- Semaphores

- Regions

Tasks are the active objects of the system. They do the work of the system and can be considered as having two goals . Their primary goal is to do a specific piece of work. The secondary goal is to obtain control of the processor in order to achieve their primary goal.

Jobs are the environments in which tasks exist. Jobs consist of tasks, task related objects, object directories, and a memorv pool. Every system includes a job tree, starting with a root job at the top. This root job is typically created by the HI when a terminal is activated, and system jobs are structured down from this original job.

Segments are the medium used by tasks for data storage and communication. Tasks requiring memory for these purposes can request a segment of the proper size from the Nucleus.

Mailboxes are objects which tasks reference in order to send or receive other objects . For example in order for a task to send an object, the sending task sends a token to a mailbox; the receiving task must then visit the mailbox and obtain the token thereby allowing the receiving task to access the object.

Semaphores are used by tasks to signal other tasks.

The semaphore achieves this by dispensing abstract units as requested by tasks and available to the semaphore.

Regions are used as custodians of specific collections of shared data. The salient feature of regions is that a task with access to a region cannot by suspended or deleted until it surrenders access to that region. This is necessary to avoid the possibility of data corruption or deadlock.

It is now possible to discuss the other essential functions of the Nucleus.

3.2.2 Scheduling

Scheduling is a critical function and therefore should be discussed next. As is apparent from the definition of the object "task" above, the application tasks are in continual competition for control of the processor. The Nucleus adjudicates allocation of the processor according to two task characteristics; the task's execution state and the task priority. Task priority is an assigned integer value between zero and 255. It is possible to assign the task priority using a "Set Priority" Nucleus call. High priority is defined as zero and low priority as 255. The task's execution state is one of five possible conditions: running, ready, asleep, suspended, or asleep-suspended. A task is either put to sleep for a specified amount of time, or else until a request has been granted. Tasks can be suspended by another task, by awaiting an interrupt or by itself. (There is an associated suspension depth increased by every "suspend" call and decreased by "resume" calls.) And of course a combination of the two, the asleep-suspended state, is possible. The running task is defined as the ready task with the highest priority. In cases where ready tasks have equivalent priority they are scheduled on a FIFO basis. Thus the processor is allocated by the Nucleus on the basis of task readiness and priority. It is possible for a low priority task to have its processing interupted by a higher priority task which achieved the ready state after the low priority task had started running.

3.2.3 Memory Management

Memory allocation is directly related to the job tree structure. When a job or task is created the Nucleus must obtain resources from the parent job. These memory resources are assigned from the memory pool of the parent job according to the memory pool size paramenters specified at the time of the object's creation. The memory pool is defined as the memory available to a job and its descendants. This leads to a hierarchy of memory pools with the same structure as the job tree, and in fact, a structure very similar to that of the cell controller system. The memory pool size can be dynamically controlled by the tasks through the system calls which can be used to examine and set pool attributes. Should a task require additional memory, it can request a memory segment of the proper size from the parent job. Should there be inadequate memory available from the parent, the task can request resources from the parent's parent, or indeed from ancestors further up the job tree.

Memory allocated to a task is a collection of segments where a segment is defined as a contiguous sequence of 16 byte paragraphs with a base address evenly divisible by 16. When a request for a segment is made the Nucleus checks to verify that enough memory is available to fill the request. If so, the Nucleus returns a token for the segment to the requesting task. If not, the Nucleus indicates this condition to the requesting task, which handles the situation on an individual basis.

3.2.4 Interprocess Communication

Interprocess communication is typically achieved through mailboxes and semaphores. Although regions are available, they are not recommended for use in systems involving HI calls due to the possibility of a system lockup resulting. Mailboxes tend to support intertask data communication. The Nucleus directs that tasks waiting at a mailbox receive their objects as soon as they are available. This involves the use of two queues. The task queue, which is either FIFO or priority based, is where tasks wait for their objects. The object queue, FIFO based only, is where objects wait for tasks to receive them. When an object is received, the task's execution state changes. A task which is asleep becomes ready; a task which is asleep-suspended becomes merely suspended. If there is no object available at the mailbox when the task visits it, the task has two options. Either it can wait according to a duration parameter specified in the call, or it can take an exception code (E$TIME) and continue.

Semaphores involve the sending and receiving of abstract units for purposes of mutual exclusion, synchronization, and resource allocation. Semaphores only have a task queue which can be FIFO or priority based. The semaphore constantly tries to satisfy a request for units by the task at the head of the queue. If enough units are available at the time of the request, the task remains ready. If there are not enough units available the task can elect to wait or receive an exception code as above.

3.2.5 Response to External Events: Interrupts

The interrupt structure of the system is what makes it responsive to asynchronous external events. Upon receipt of an interrupt signal, the interrupt processing routine can take control of the processor, service the request, and return control. There are three key concepts in the use of interrupts under iRMX-86. These are the interrupt vector table, the interrupt level, and disabling interrupts.

The interrupt vector table is composed of vectors numbered 0-255. Vectors 0-55 are reserved for the system, but 56 through 127 are reserved for external interrupts. Numbers 128-183 are available to users, and the balance (184-255) are reserved. An interrupt triggers a call to the interrupt vector table, where the call is redirected to the interrupt handler.

Interrupt levels are related to the funnelling process of the 8259A PIC. Using a master PIC and six slaves, the system recognizes signals on a particular line as being associated with the master (vectors 56-63) or the slave levels (numbers 64-127). When the system receives an interrupt at an enabled level, it transfers control to the address in the interrupt vector table associated with the interrupt level.

Since a user may require that an interrupt does not cause an immediate break in the task currently in processing, it can be necessary to disable the interrupt. For example, the system analyst may deem it unwise to interrupt a high priority task with a low priority interrupt. Disabling the interrupt lines can be achieved through system calls from application tasks or by the operating system supporting pre-emptive priority based scheduling. In the former case the task must explicitly define the disabled levels. In the latter case, the operating system disables levels according to the priority of the running task. An example of this is for a priority 20 task; here the slave levels 00-77 and 25 the master levels M2-M7 are disabled.

When an interrupt is serviced, it is redirected to its handler through the interrupt vector table. In some cases, the handler can completely service the interrupt. If, however, the servicing requires substantial amounts of time, or system calls beyond those available to handlers, it is then necessary to invoke an interrupt task to complete the service. The interrupt handler can share data with its associated task (through the Enter$Interrupt call).

3.3 The Outer Layers

If the Nucleus is the core of the system, then the service subsystems (BIOS, EIOS, HI, and the Loader) and the application programs are the outer layers. These outer layers provide services to the Nucleus by facilitating its communications with and control of the system. They also provide services to the user by simplifying the system calls necessary to achieve a given piece of work.

3.3.1 BIOS--The Basic Input/Output System

BIOS is a flexible and powerful system due to its low level orientation. It gives the user complete control over details, even to the extent of requiring the user to specify his own buffering algorithm. Most important, however, it deals with asynchronous system calls. This capability is required in an interrupt driven system.

3.3.2 EIOS--The extended Input/Output System

EIOS is an extension of BIOS, requiring an extra 13 Kb of memory if included in the system. Although not as flexible as the basic system, it is much more convenient from a system development point of view since it automatically handles most I/O details.

It provides automatic buffering, defeatable if need be. Its chief limitation is that it deals only in synchronous system calls. This precludes the application tasks from achieving any work during communication intervals.

3.3.3 HI: The Human Interface

The HI creates a root job for each terminal activated. This job is called the interactive job, and the HI assigns its memory pool parameters according to configuration specifications. The HI allows a terminal operator to load and execute certain commands from the keyboard. These are typically file management, disk management, and general utility commands such as SUBMIT, or DEBUG. The HI achieves this by starting the Command Line Interpreter(CLI). The CLI reads the commands from the terminal and invokes the system calls appropriate to the terminal input. HI calls are typically concerned with command parsing and processing, I/O and message processing, and program control.

3.3.4 The Application Loader

The Application Loader loads tasks under direction of the operating system. These include both application tasks and operating system tasks. The Loader also provides calls for loading programs from secondary storage into main memory.

The Loader is capable of loading absolute code, position independent code (PIC) and Load-Time-Locatable (LTL) code. Absolute code is code that has been processed by LOC 86 to run at a specific address. PIC is loaded by requesting memory segments from the job pool and loads it there. LTL, similar to PIC, has the base addresses of its pointers adjusted by the Loader so that the pointers are independent of the processors register contents. This fixup allows LTL code to be used by tasks having in excess of 1 code segment and 1 data segment.

4. CMC--The Implementation Example

The implementation example was directed at establishing a flexible, generic hierarchical cell controller structure. This developmental structure was strictly intended to be a framework for future work and was not intended to operate a cell in this format. Before further discussing the implementation example , it is necessary to specify the constraints and conditions regarding it.

4.1 Implementation Constraints and Conditions

Two immediate constraints in the implementation process involved the hardware and software components of the system. Specifically this involved using the systems available in the Industrial Engineering Microcomputer Lab. This included the following hardware items.

- Intel 86/380 development system
- 35Mb Priam Winchester drive
- 2 Memory boards yielding 512 Kb
- Disk controller board
- 86/30 processor board (8086 based)
- Televideo 925 terminals
- 8 inch floppy drive

Software available to run on the system included:

- iRMX-86 with libraries and utilities
- PL/M-86 compiler with libraries and utilities
- ASM-86 compiler with libraries and utilities
- FORTRAN compiler with libraries and utilities
- PASCAL compiler with libraries and utilities

Several factors had a considerable impact on the programming of the system. First, the manufacturer suggests that EIOS be used as a development tool to avoid programming overhead associated with BIOS. Further, the structured approach we assumed for programming required heavy use of program stubs. This allows feedback to be generated from the stubs, particularly given the utility and ease of generating such using a high level language (such as PASCAL) to return messages to the terminal screen upon successful execution of the trivial stub. This had two fundamental implications. First that EIOS would be included in CMC, the name given the controller, and second that for development purposes all jobs in the structure would be I/O jobs. Since only I/O jobs can successfully use all the EIOS system calls, these two items are mutually supportive.

4.2 Programming the CMC System Structure

It was at this point that the full benefits of a structured approach were realized. Since the system programming language (PLM-86) is highly structured, and since the CMC system structure chart (or VTOC) is similarly structured, it was possible to directly translate the top levels of the VTOC into the top level jobs of the CMC system. This task was further simplified by the system constraints, specifically that all jobs would be I/O jobs and that program stubs would be used. Further, since the parametric requirements for each job were unclear at this stage of the development, it was dcecided to use a generic parameter selection for all.

The upshot of this was that it was possible to create three generic dummy modules: a job creation module, a task creation module, and a trivial task module. These modules were then cloned, and the module names (job names and task names) were taken from the chart. The source code files for each module were given a number as a name, where the number was taken directly from the chart. Thus the entire system structure was created, degugged and linked.

This scheme provided several advantages. The use of I/O jobs supported the anticipated feedback generation scheme. The modularity of the system supported top down coding and testing, so that as modules were added to the system, bugs could be more readily identified and resolved. After the first three dummy modules were coded and debugged, creation of the entire system structure was rapid. Once the entire CMC structure was intact and linked, it was possible to pursue implementation of applications tasks to replace the stubs used at inception.

The limitations of the approach are that as job requirements become clearer, it would be desirable if not necessary to refine the job creation parameters. This is particularly true with regard to such items as priority and memory pool. In addition, full implementation of the controller would require refinement of the I/O jobs, since sensor activated interrupts would require the asynchronous system calls of BIOS. The latter could require extensive work, so the drawbacks are not to be minimized. However, given the developmental nature of the system, it was felt that, despite the drawbacks, the approach taken was satisfactory.

4.2.1 cti$mfg$cell: A Specific Example

The top level job in the system is named cti$mfg$cell. The name was taken from the VTOC, shortened and delimited by "$" to meet PLM requirements, and used as the name of the top module. (This is the origin of "CMC", the name used for the controller.) The VTOC reference number is 0000--denoting the top level module--therefore its source and object code files were named 0000.SRC and 0000.OBJ respectively. A view of the general procedure in pseudocoded statements is useful for further discussion.

```
CONTROL MANUFACTURING CELL JOB
INCLUDE LIBRARY FILES
DECLARE NEXT LEVEL JOBS AS EXTERNAL PROCEDURES
DECLARE DATA TYPES
DECLARE JOB CREATION PROCEDURE
        SET JOB PARAMETERS
        CREATE THE NEXT LEVEL JOB
        CATALOG THE OBJECT FOR THE NEW JOB
END THE PROCEDURE
Repeat declarations for the rest of the jobs.
Call the job creation procedures in turn.
END
```

The job begins with a statement of its name, then causes the library files appropriate to its system calls to be linked in. The next level jobs (termjob, mathdl$job, process$job, ei$sense$job, and db$cntl$job) are declared to be external procedures since they are not defined in this module. Then any data, pointer or parameter names have their types (such as WORD or BYTE) declared. Then a procedure for creating the next level job is declared, parameters set, and the job creation call is made. The object for that job is then cataloged, and the creation procedure is ended. Similar procedures are declared for the rest of the next level jobs, and the calls are made to invoke the procedures previously declared.

It is apparent that the real work of the procedure is done by the job creation system call and the "catalog object" system call. The "catalog object" call does just that; it catalogs the token for the next level job (returned by the job creation call) along with a name for that object in the job's object directory. In the current case, the names used were the module numbers from the VTOC. This allows other objects to access the token simply by knowing the name. However, it is the "CREATEI/OJOB" call where the work of defining the new job's environment is achieved, so a detailed examination of it is in order. This call has the format:

io$job = RQ$CREATEIOJOB(pool$min, pool$max, except$handler, job$flags, task$priority, start$address, data$seg, stack$ptr, stack$size, task$flags, msg$mbox, except$ptr)[1]

A discussion of the parameters and how they were set in the CMC system demonstrates how the job tree and task environments are defined.

The pool$min and pool$max parameters define the allowable memory pool size for the job in 16 byte paragraphs. The minimum allowable under EIOS is 32. However, if the Nucleus is allowed to define the stack$ptr parameter (see below), then pool$min should equal 32 plus the number of paragraphs needed for the stack. This was estimated at 32, so pool$min was set at 64. Pool$max was set at 256 to prevent excessive memory acquisition by a single job. Except$handler is a pointer to an exception handler, which was defaulted to the system handler. Job$flags contains Nucleus information about the job. Task$priority defines the priority for the first task of the created job. This was set at an

----

6

1 iRMX-86 RELEASE 5 EXTENDED I/O SYSTEM REFERENCE MANUAL, INTEL Corp., Santa Clara, CA , 1983, Page 7-4

artificially high "0" to ensure that each job's first task runs once before resetting its priority. The start$address parameter points to the first instruction of the new job's initial task, which was specified in CMC as a string NEXT-LEVEL-JOBNAME, where the jobname was the PL/M-86 job name from the next lower level of the VTOC. Data$seg was set to 0 to indicate that the new job would request segments as needed. Stack$ptr was set to zero, thus allowing the Nucleus to define the pointer value. Stacksize, implicitly determented by the pool$min assignment, is confirmed as 512 bytes. The Nucleus is advised if the job should run immediately, or wait for a start I/O job call and if 8087 instructions are contained in the new job's initial task by the value of task$flags. Ex$ptr is a pointer to a word where the exception code associated with the call will be returned by the system. The only other return is io$job, a token for the new job.

Thus the environment for the new job is assigned. The second level then creates jobs or tasks according to the VTOC. Where additional jobs are created, the process is identical. Where the jobs begin creating tasks, the procedure looks the same, except that the job creation call is replaced by the task creation call.

task = RQ$CREATE$TASK(priority, start$address, data$seg, stack$pointer, stack$size, task$flags, ex$ptr)[2]

The parameters have the same definition as the equivalently named job creation parameters except for the terms "priority" and "task". Here, "priority" is the priority of the task being created unless set to zero. In that case, the task assumes the maximum allowable priority as defined by the job. (Basically, a descendant cannot have a higher priority than any of its ancestors.) "Task" is the token returned by the system for the new task, and is cataloged for future access.

Thus the top level job creates the second level jobs. Some second level jobs create application tasks, the rest create other jobs at the third level. These in turn create more application tasks and it is the application tasks that do the system's work. Although the specifics of the application tasks are beyond the scope of this thesis (see "The Application Software Structure for a Hierarchical Industrial Controller System" by J.E. Dorney, Master's Thesis, Lehigh

----

[2] iRMX-86 NUCLEUS REFERENCE MANUAL, Intel corp., Santa Clara CA, 1982, Page 12-37

University, 1984), it is useful to discuss the basic operation of the cell, and define the essential output control algorithm. I can then define some basic elements required in the database to support cell operations.

5. Basic CMC System Operation

A description and discussion of the conceptual operation of the cell controller is now in order. At power-up, the system boots itself. As soon as a terminal is activated, HI creates the interactive job for that terminal. The user can then log on and invoke CMC. CMC responds by requesting a system mode specification: program (which allows the user to access and update the data base), report (which causes statistics to be copied to the terminal or filed on the 8 inch floppy), and run (which causes the system to inquire about the part number to be manufactured and the quantity). This last mode is the one I will discuss. When the required information is provided, the system formats a command table based on the user supplied information. The system then reads the command table sequentially, issuing calls to device and process modules in response to the codes in the command table. As workpieces move through the system, fresh ones are introduced, triggered by the evacuation of the fixture in front of the waiting workpiece. The system counts the number of units processed and stops the introduction of parts into the system when the number required plus the number scrapped has been reached. After production is stopped, the system awaits a new command. The command table and output scheduling algorithm are central issues in this process. However, when discussing the command table it will necessarily define its basic resource, the data base.

5.1 The Command Table and the Data Base

Creating the command table presumes that two types of information are available. First is what might be described as a "routing sheet" which contains a list of parts required, assembly specifications, and process specifications. In addition, the controller will need to know the locations (and appropriate gripper orientations where a robotic device is concerned) of various critical points in the cell. Typically these would be such items as part feeders or fixtures/load points, and would be referenced to the cell's base coordinate frame. The availability of this data presumes that a graphic model of the cell has been created and analyzed.

5.1.1 The Data Base Records and Keys

The data base will thus need to contain at least two types of data, what I have defined as "routing sheet" data and what is essentially location and orientation information, hereafter the "location table". The keys and fields of these two data bases are largely defined by the considerations above. First, the routing sheet. This entity must use the user supplied part number as the key. This enables the system to locate the appropriate routing sheet and access the data. The routing sheet must contain the data mentioned above, plus some extra fields specific to the cell. These extra fields are user supplied information which support the cell's decision processes. They are the transaction type code and the transfer device code. These are defined--as are the other fields--below.

1. Part Number (PN) - this is the key field, and is one of the assemblies appropriate to the cell.

2. Sub Assembly Part Number (SAPN) - the part number of the sub assembly or piece part.

3. Sub Assembly Part Number Quantity (SAPNQ) - the number required.

4. Transaction Type code (TTC) - Move or Process.

5. Process Parameters (PP) - parameters for non-machining processes, pointer to NC code for download to machine tools.

6. Process or fixture required (PF) - code for process or fixture number.

7. Transfer Device Code (TDC) - specifies which device is to handle the workpiece next.

The requirements for the location table are straightforward, however the organization is not. The complicating factor is that more than one top assembly (the PN) may use the same sub assembly, and that more than one sub assembly with the same SAPN may be required in one top assembly. Therefore the table must be keyed so that the correct geometric data is associated with each part. This compels the use of a concatenated key based on the part number, the sub assembly part number, and the number of SAPN required. Thus the record format is as follows:

1. PN*SAPN*N : concatenated key, where N is a number from one to the number of SAPN required(SAPNQ).

2. SAPN feed location (SAFL) : defines coordinates and the gripper approach vector for part feeding mechanism, (and a safe point near it if no vision processing is available).

3. PF location (PFL) : the coordinates and gripper approach vector for the process or fixture pointed to by PF (and coordinates for a "safe point" near it , as above).

4. SAPN assembley index (SAI) : the index from the fixture point specified by PFL to the assembley area.

5. SAPN orient (SAO) : the orientation and approach vectors for the gripper for assembly.

6. SA final (SAF) : the final location of the gripper when assembly is complete.

Although this is not in third normal form due to the redundancy in the PFL and SAFL fields, I feel that the reduced access time associated with reading only one record justifies this conceptual error.

5.1.2 Creating the Command Table

When the user enters the PN and quantity parameters, the system accesses the data base and writes the contents of the "routing sheet" into a new file called the command table. When multiple SAPN's are required, the new table is formatted to repeat that entry a number of times equal to the quantity required, and enumerate the SAPN field to SAPN*N. Two additional fields are added, the "tag" and the "index". The tag is a counter issued by the system, and is initially set to zero. It is incremented by one each time a new part is introduced into the system. The index is initialized to one, and a new index equal to one is issued each time a new part is introduced into the system. The index is incremented by one each time an associated command table line has completed its output. The tag uniquely identifies each part in the system, and is also used for bookkeeping purposes. The index tracks the progress of each tag in the command table, thereby providing a unique reference to each assembly. The system then accesses the location table and merges it into the command table on the key PN*SAPN*N.

5.2 The CMC Scheduling Algorithm

The CMC event scheduling algorithm is simply stated, and works hand in hand with the command table. First, the system checks to see if the initial FP is empty. If it is, and if the number of tags issued is less than or equal to the number of assemblies specified by the user plus scrap incurred to this point, the system increments the tag and issues it. Whenever a tag is issued, it also has an index appended and initialized to one. Next the system reads all command table lines currently having a PN*TAG*INDEX entry and if the index has changed, processes those with new indices from the bottom of the table to the top. (This allows fixtures to be cleared starting from the output end of the cell and working back.) This processing involves issuing calls to the necessary tasks as defined by the command line parameters. The tasks then perform the necessary calculations, gather the appropriate data, and format the output. The output tasks are then put to sleep to await a signal that the last command line (on a per tag basis) execution is comlete. Where movement to a new FP is indicated in the next command line, the output tasks will also await a signal that the new fixture is clear. When the appropriate signals are received, the output is transmitted and the index of the pointer PN*TAG*INDEX is incremented by one. This procedure repeats until the command table is empty, i.e., the correct number of parts have been introduced into and exited from the system.

This proposed algorithm is tailored to use the strengths of the real time operating system. For example, task calls are issued only when required, rather than periodicly. This directs the processor resource to the best advantage of the system. Further, processing the next command line while the current line is still in the execution phase is effecient, since it reduces the time spent waiting for output parameters when the next line becomes the current line. As is mentioned above, the systems moves items from the output end first so as to sequentially clear prior fixtures. This enhances cell throughput by promoting workpiece moves as soon as a FP is cleared.

6. Conclusions

Contrary to some of the literature encountered, I found that the use of a real time operating system presented many positive aspects in implementing a hierarchical manufacturing cell controller. First, the structured nature of the system programming language supported and enhanced the use of structured design and analysis techniques. This is important in system development since it supports the use of modules and program stubs, which are of great assistance for debugging and integration purposes. In addition, and perhaps most importantly, the highly modular nature of a CMC-like system supports the writing of N tasks for N events, a much more direct effort than writing large, intricate, comprehensive programs.

Perhaps the most distinct advantage accrued in using the real time operating system centers around the fact that in systems such as a cell controller, most operations are event driven and event oriented. The interrupt service structure of a system such as iRMX-86 is specifically tailored to operate in an event oriented environment. Through the use of a scheduling entity such as the CMC command table, the event oriented system can always be looking ahead and planning its output before it is required. In this fashion, the system immediately responds to a cell event by sending output which causes the next event. This is clearly advantageous in an event oriented system. iRMX-86's pre-emptive priority based scheduling scheme supports this overall structure by ensuring that low priority interrupts do not impede more critical activities, and by ensuring that the most critical events are serviced first.

The area posing the greatest challenge was in the definition of data communication inside the system. Internal communications are necessarily comprehensive since different data types and segment lengths need to be exchanged between the tasks. Therefore, it is necessary to completely define the necessary data characteristics and exchange methods during system design. The heavy reliance on a structured approach thus extended to the data exchange framework as well. Indeed, this framework appeared very straightforward on the original VTOC. In general, the original VTOC showed all modules updating the System Status block, and external tasks requesting updates from the block. However, as specific requirements for certain types of tasks were identified, modules were refined and any necessary data exchange fixups were made. This tended to cumulatively degrade the modularity of the exchange scheme, and complicate the issue. The operating system alleviated this to a certain extent through the use of task and object queues in mailboxes. This allows a task to queue a request for an object, then wait for it. However, to ensure smooth operation of the system it is necessary to anticipate this request and design the system so that the module generating the object required runs at the proper time. This can be done by setting the priority of the object task to a high enough level, but the designer must anticipate such a circumstance and ensure that the object task does not run at the expense of a more critical (in system terms) module. Certainly it appears that in the balance, the advantages of the real time operating system based hierarchical cell controller outweigh the disadvantages, and should be investigated further.

6.1 Areas for Future Study

The issue of real time data exchange in a hierarchical structure is identified as an area for future study, based on the observations above. In addition the issue of tracking and directing cell components using transformation matrices as opposed to the use of sensory interactive feedback arose, and should be investigated. Thus, a minimally developed cell controller implementation should be attempted using a real time operating system such as iRMX-86. Only implementation down to the device level can address the issues, probably revealing new ones in the process. Therefore I recommend that a model consisting of the controller, a robot, a conveyor, and some dummy targets be constructed for research purposes.

The purpose of such an effort should be to address the two issues above, a synergistic effort since they are interrelated. In addition, a vision system should be included for comparison with the effectiveness of the mathematical model in defining cell member boundaries and trajectories. During this process, the data exchange requirements and methods should be cataloged and examined for similarities in structure and processing. In this way it may be possible to develop an algorithm for the specification of data exchange requirements for a real time system.

7. Bibliography

Barbera, J.J., J.S. Albus and M.L. Fitzgerald, "Hierarchical Control of Robots Using Microcomputers", PROC. 9th INT. SYMP. ON INDUS. ROBOTS, March 13-15, 1979, pp. 405-422.

Barbera, A.J., J.S. Albus, M.L. Fitzgerald and Marilyn Nashman, "Sensory Interactive Robots", National Bureau of Standards, Washington, D.C.

Barbera, A.J., J.S. Albus, M.L.F Fitzgerald, "Programming a Hierarchical Robot Control System", National Bureau of Standards, Washington, D.C.

Leeson, Marjorie, SYSTEMS ANALYSIS AND DESIGN, Science Research Associates Inc., Chicago, 1981.

Miller, William E., DISTRIBUTED COMPUTER CONTROL SYSTEMS 1981, Pergamon Press, New York, 1982.

Paul, Richard P, ROBOT MANIPULATORS: MATHEMATICS, PROGRAMMING AND CONTROL, MIT press, Cambridge, Mass.

Private meetings with Professor L. J. Plebani, Department of Industrial Engineering, Lehigh University, on the following dates: October, 1983 through May, 1984.

Classroonm discussion and private conversation with Professor N. Odrey, Department of Industrial Engr., Lehigh University, November and December, 1983.

Report: "Design of a Microprocessor Based Hierarchical Control System", submitted by A.D. Baker and J.E. Dorney as part of Lehigh University Ind. Engr. course #433, December 12, 1983.

Intel iRMX-86 USER GUIDE, Intel Corp., 3065 Bowers Avenue, Santa Clara, CA 95051, 1982.

Intel iRMX-86 Human Interface Reference Manual, Order No. 9803202-03, Intel Corp., 3065 Bowers Avenue, Santa Clara, CA 95051, 1982.

Intel iRMX-86 Loader Reference Manual, Order No. 143318-002, Intel Corp., 3065 Bowers Avenue, Santa Clara, CA 95051, 1982.

Intel iRMX-86 Nucleus Reference Manual, Order No. 9803122-04, Intel Corp., 3065 Bowers Avenue, Santa Clara, CA 95051, 1982.

Intel iRMX-86 Programming Techniques, Order No. 142982-003, Intel Corp., 3065 Bowers Avenue, Santa Clara, CA 95051, 1982.

Intel iRMX-86 Release 5, Basic Input/Output System Reference Manual, Order No. 172766-001, Intel Corp., 3065 Bowers Avenue, Santa Clara, CA 95051, 1982.

Intel iRMX-86 Release 5, Extended Input/Output, System Reference Manual, Order No. 172767-001 , Intel Corp., 3065 Bowers Avenue, Santa Clara, CA 95951, 1982.

Vita

Albert Dawson Baker was born in Buffalo, New York on 11/1/51, the son of Milton Joseph Baker and Dorothy Dawson Baker. He attended Hobart College where he was awarded a Bachelor of Arts degree in 1974, then studied electrical engineering at Lehigh University and was awarded a Bachelor of Science degree in 1978. After four years with the Bendix Corporation, Al returned to Lehigh where he is attaining a Master of Science degree in Industrial Engineering. He will be pursuing a career in industrial automation development with the General Electric Company.