REAL TIME QUICK-LOOK ANALYSIS
FOR THE OGO SATELLITES

R. J. Coyle and J. K. Stewart
Datatrol Corporation
Silver Spring, Maryland

Introduction
The National Aeronautics and Space Administration has designed a series of general purpose orbiting satellites which have the family name of Orbiting Geophysical Observatory (OGO). Each of these satellites is capable of carrying up to 50 scientific experiment devices, which transmit data to ground stations via a common telemetry channel. There are two tracking stations, at Rosman, North Carolina, and Fairbanks, Alaska, which will receive this telemetry, transmitting it to Central Control at Goddard Space Flight Center in Greenbelt, Maryland, at a data rate of up to 64 KC.

General Description of the Programming System
The requirement of the programming system for OGO was to provide quick-look analysis and control of the status of the spacecraft and selective experiments on board the satellite. (The tracking and orbit determination of OGO is not a function of this program.) All telemetry is recorded for further extensive analysis in non-real time on other equipment. There are several special purpose consoles attached to the computer to accept telemetry, and provide a means of communication to and from the spacecraft. These are appropriately called: PCM Telemetry input equipment, Control and Display Console, and Command Console. It is through the integration of these consoles with the computer program that experiments on board the satellite may be selected to start, or terminate, by a command from the computer. The selective analysis of those experiments transmitting data is initiated by an input request from the Control and Display Console and selective spacecraft status analysis is handled in the same way. It is therefore a requirement of the computer program that it be flexible enough to handle these many and various requests instantaneously. Since the bit rate of 64 KC means a new telemetry frame of data will arrive every 18 ms., the computer programs must be able to make maximum use of the capabilities of the computer.

The SDS 920 computer was chosen by NASA to handle the quick-look analysis and control for OGO. The 920 is a small-scale computer with a 24-bit word and an 8 us. cycle time. It has one index register and uses single-address fixed-point arithmetic logic. It is capable of handling many kinds of input/output, to include: magnetic tape, paper tape, punched cards, on-line typewriter, and high speed printer. The OGO 920 has 8 K of memory and 32 channels of interrupts for input/output use. The OGO installation is illustrated in Figure 1.

Functions of the Real Time Monitor Control System
The programming system for OGO can be thought of in two distinct parts: the Real Time Monitor Control, and Experiment Processors. All routines necessary to connect the computer with the external environment are integrated into the Real Time Monitor Control system. It
must service all interrupts, provide for the receipt of all input including telemetry and manual input requests from the consoles, direct and send out all output to the appropriate receiving device, keep track of time, and determine the sequence in which all functions and processing is conducted. The Monitor Control does not process any of the data itself, but rather acts as a general purpose framework in which selective routines perform the processing. These selective routines are called Experiment Processors and are independent routines operating under control of the Monitor. These routines process telemetry input data, returning to Monitor formatted output for driving digital displays on the special purpose consoles, for printing, and for typing, as well as other information to the Monitor.

The first distinct part of the programming system for OGO, the Monitor Control, is itself divided into three sections: the Schedule Program, Monitor Processors, and Interrupt Processors. The following is a definition of the functions and characteristics of these various sections.

Monitor Schedule Program

The Monitor Schedule Program consists of routines which collectively coordinate, supervise, and schedule all processors in the system, utilizing a priority table of processors. It saves the status of the machine whenever an interrupt occurs, and facilitates a proper return from an interrupt by restoring the condition of the machine at the point of interrupt. The heart of this scheme is the Monitor Schedule Routine whose function is to examine sequentially the entries in the priority table in order to determine the next routine of highest priority to be processed.

The priority table of processors consists of a group of words, or module, which identify and describe each processor with a reference in the table. Due to the difference of functions of the various processors, the modular approach allows for flexibility in assigning priority during the debugging stage of the system. By chaining these modules together, say by the first word of each module referencing the first word of the next module, the modules may be of various length (Fig. 2).

In order for the Schedule Routine to carry out its function of determining the next processor of highest priority with something to do, it must know the status of each processor in the

![Figure 1. OGO Central Control Installation.](image)

![Figure 2. Make Up of First Word of the Modules.](image)
priority table at any given instant of time. This is accomplished through the use of indicator bits in the module, say the left portion of the first word of each module (Fig. 2). The various states that a processor may be in are the following:

'in process' — This processor was interrupted while it was processing, and entry should be made at the point of interrupt.

'ready' — this processor now has something to do; entry should be made at the beginning.

'suppressed' — this processor should not be processed at this time.

'terminated' — this processor has now completed its processing.

'not in core' — this module is available for the addition of a new processor.

These indicator bits, then, inform the Schedule Routine whether or not the referenced processor is now available for processing. Since the priority table entries must be examined from the top each time return is made to the Schedule Routine, a rapid method of examining the indicator bits of each module was needed. To clarify the method that was devised, it is necessary to diverge for a moment and describe two uses of an instruction in the SDS 920 repertoire, which may be used to load the index register.

EAX A Immediate Addressing, X Reg. = A where the address field of the instruction itself is placed in the index register.

EAX* A Direct Addressing, where A PZE 1 the contents of location A, X Reg. = 1 referenced in the address field of the instruction, is placed in the index register.

EAX* A Indirect Addressing, where, A PZE* B if the contents of location B PZE 2 A is in turn indirectly addressed, the address field of A will again be used as the location from which to load the index register.

The SDS 920 computer has the capability of unlimited indirect addressing; that is, it is possible to continue the above indirect addressing of an instruction for many steps, until a location is encountered that is not itself indirectly addressed. It is convenient then to choose the indicator bits of the modules discretionately to take advantage of this feature of the machine as well as the fact that the modules are chained together. If all states of a processor which indicate that it is not now available for processing contain the indirect addressing bit as one of the indicator bits, then only those routines with something to do at the current time would be picked up for examination. For instance, the processor states of 'suppression,' ‘termination,' and ‘not in core’ all result in there being nothing to do at this time on this particular processor.

In Figure 3, the consequences of giving an “EAX* A” would be that modules A, B, and C would be skipped and the first word of module D would be placed in the index register. For module A is ‘suppressed,’ module B is ‘not in core’ and module C is ‘terminated,’ indicating that no processing should be done on these processors at this time. The contents of the index register actually contains the location of

Figure 3. Example of the Use of the Indicators.
the next module, due to the chaining criteria, but this presents no particular problem if reference is always made to the words of the module in reverse; that is, -1,2 or -6,2 etc.

Since it costs only one machine cycle to skip each successive module whose indirect addressing bit is set, the result is a most efficient search of the priority table. The alternative would be to pick up the contents of the indicators of each module and then through various logical compare instructions determine whether any of these situations are the case. It is apparent that the scheme using the indirect addressing feature of the computer facilitates spending the minimum time examining the entries of the rather lengthy priority table of processors.

In addition to the indicator and chain word, each module also contains the first word location of the processor which it references, as well as a block of locations in which the condition of the machine is saved when this processor is interrupted. If the referenced processor has input/output functions, the module will additionally contain the location of the I/O data which is stacked in a table for subsequent transfer or editing. If the processor is an Experiment Processor, it will also contain the location of flag words in the processor by which communication is made with the Monitor. That is, words by which each Experiment Processor can signal Monitor of its various output requirements, and whether it has completed processing, etc. (Fig. 4).

The many processors of the Real Time Control System may be in various states during the course of real time operations so that it is impossible for one processor to communicate directly with another. This is especially true of I/O data and channel select requests. Therefore, two subroutines were devised in the Monitor Schedule Program to handle the passing of data from one processor to another. One subroutine handles the stacking of data onto the receiving processor’s stacking buffer, while the other handles the unstacking of the next data to be processed. The barrel, or wraparound, method was chosen to facilitate the stacking and unstacking from these buffers. That is, a ‘first in-first out’ list type of scheme. To avoid moving the data around in core memory, since this is necessarily inefficient, the items which are stacked and unstacked in the buffers are the locations of the I/O data or channel select requests.

Each I/O processor’s module, then, contains the top and bottom locations of the stacking buffer for that processor. It can be seen that these buffers may be variable in length from one processor to the next since each processor’s buffer is uniquely defined in its module. The module also contains the location of the first item to be unstacked as well as the next free location in which to stack. These locations may vary anywhere within the prescribed stacking buffers area, hence the name ‘barrel method’ (Fig. 5).

When an item is unstacked, the ‘first’ location reference in the module is replaced by the ‘first + 1’ location reference so that the next item is now referenced for unstacking. Similarly, when an item is stacked in a buffer at the ‘next’ location, the ‘next’ location reference is then updated to the ‘next + 1’ location as the next free location in which to stack. The top and bottom locations, delineating the total

Figure 4. Composition of the Modules.
buffer area, can be thought of as being only one unit apart, for the subroutines always treat the bottom location + 1 as if it were the top location. The circular action has now been effected.

**Monitor Processors**

The Monitor Processors handle the input/output requirements of the Real Time Monitor Control, as well as a few other minor functions. The SDS 920 has two buffered channels for input and output data transfer and 32 real time channels, all of which have interrupt capabilities. One of the buffered channels is used exclusively for the telemetry input, the other handles all other I/O functions, except those from the special purpose consoles, time interrupts, etc. Due to the difference in the rates of speed of the devices attached to this I/O channel, it was decided to have a series of processors, each one associated with only one I/O device, rather than a single large I/O processor geared to the buffered channel. Each of these processors, like all other processors, is under control of the Monitor Schedule Program. Its priority order is determined by the position of its module in the priority table and is in direct relationship with the speed of the I/O device to which it refers. For example, the priority of a processor handling the on-line printing is higher than the one handling the output messages to the typewriter.

Since only one channel select request can be executed by a buffered channel at a time, all of the Monitor I/O Processors hand off their I/O select requests to a single processor, rather than randomly selecting the channel themselves. The function of this processor is to select the channel with whatever request is next in line, regardless of the kind of I/O device indicated. This processor is guaranteed that the channel is ready to receive the select request, for as each request is serviced, this processor 'suppresses' itself. That is, it sets the indicator bits in its own module so that it will not be considered for processing again until it is 'unsuppressed.' When the interrupt from the completion of the data transfer on that channel occurs, the interrupt processor servicing this interrupt will 'unsuppress' the channel selecting processor so that it may now continue with the next I/O request, since the channel is free to be reselected. This channel selecting processor has the highest priority of all the other Monitor Processors, thus allowing for the maximum use of the channel at all times.

There are other Monitor Processors which handle the input/output to and from the special purpose Control and Display Console and the Command Console. These processors are controlled and executed in a straightforward manner since each function is handled by a unique interrupt through the 32 real time channels. There are eight such processors connected with these consoles. One additional Monitor Processor handles time in the system. It maintains a gross time figure in minutes, and prints out a GMT time message periodically. It is especially used to flag certain processors dependent on some interval of time that it is time to commence processing. This procedure is called 'readying' a processor.

**Interrupt Processors**

The third category of the Monitor Control deals with processors which handle interrupts. Each Interrupt Processor is coordinated with a separate and unique interrupt channel. The 32 channels of interrupts have a priority system of their own in that lower level channels may be interrupted by any higher level channel. This means that the corresponding Interrupt Processors also may be interrupted during the course of their processing if an interrupt of a higher priority comes in. Therefore, the Interrupt Processors are naturally coordinated with the demands of the interrupt to be processed as well as with the priority level of the channel selected to handle that interrupt. For instance, it was decided that the telemetry interrupt would be on the channel with the highest priority, since a new word of telemetry comes into
the computer every 280 us. The one second time interrupts were given the second highest priority, and so on.

Since all processors, whether Monitor Processors, Interrupt Processors, or Experiment Processors, return control upon completion to the Monitor Schedule Routine, it was necessary to vary this requirement somewhat in the case of Interrupt Processors. For if an Interrupt Processor is itself interrupted, then return should be made to the point of interrupt. For interrupts are a demand to acknowledge some I/O data transfer and must be processed before the normal processing continues. This allows for smooth and coordinated interaction between the different parts of the Real Time Monitor Control system. The technique of drawing a hypothetical line in memory was used to allow an Interrupt Processor to decide quickly whether it had interrupted out of another Interrupt Processor, for all Interrupt Processors are placed above this 'line.' If this is the case, return is made to the interrupted Interrupt Processor and not to the Schedule Routine. It logically follows that when all Interrupt Processors have finished, the one which caused the first interrupt in time will be the last to finish and will return control to the Schedule Routine. One of the advantages of this scheme is that it is not necessary for the Interrupt Processors to run in a 'disabled' mode. That is, where the computer is prevented from receiving another interrupt by executing a special instruction. In fact, the 'disable' instruction is given rarely in this real time system, and when it is used, the duration of the disabled mode is very short. For, since a new telemetry word comes into the computer every 280 us., it must not be disabled for a period of time equal to or greater than that.

Experiment Processors

The second distinct part of the programming system for OGO consists of all those selectable, independent routines called, collectively, Experiment Processors. The function of these Experiment Processors is to perform analysis on the telemetry input data, both of the experiments on board the satellite and the status of the spacecraft itself. These processors operate under control of the Monitor Control programs, returning, upon completion of processing each telemetry frame of data, to the Monitor Schedule Routine. Since there are many more Experiment Processors than would fit into core memory along with the permanent Monitor Control, it is the function of the Monitor to be able to add and delete selective Experiment Processors without disturbing the real time processing. A Monitor Processor, called the Real Time Load Processor, performs this function. It is this real time load capability of the Monitor Control which affords the flexibility of operation required of the system. The Monitor Control can thus provide for maximum use of memory capacity as well as maximum use of processing time.

Requirements of Experiment Processors

All Experiment Processors place three general requirements on the Monitor. They must be able to obtain all data needed from the Monitor, including such parameters as time and orbital position of the satellite, as well as specific data points from the telemetry frame. They must also be able to return the results of their processing back to the Monitor in the form of messages to be output on the on-line printer. Finally, they must be able to communicate with the Monitor that they have completed their processing or to request that special functions be accomplished, such as sending commands to the special purpose consoles. It is the function of the Real Time Load Processor to provide the capability of bringing such an Experiment Processor into memory when requested by depressing the appropriate button on the Control and Display Console, and to make all necessary connections and links between the processor and Monitor.

Coding Rules

A problem arose in the fact that these Experiment Processors were to be written by many different programmers. This would appear to involve a considerable amount of supervision of the coding used to protect the system from 'blow ups,' but actually it was possible to handle this checking automatically by writing a service program called the Static Checker. As its name implies, the Static Checker checks the symbolic coding of an Experiment Processor to determine whether the programmer has complied with the various coding rules laid down for the system.
The rules to be followed by a programmer when writing an Experiment Processor are of two types: real time restrictions, which are necessary to protect the Monitor Control; and relocatable binary restrictions (since the processor will be used in this form), which follow from the nature of relocation itself. The rules are:

Real Time Restrictions
1. No I/O instructions
2. No breakpoint switch tests
3. No halts

Relocatable Binary Restrictions
1. No addresses of type: —Symbolic, Symbolic + Symbolic, or use of * or /
2. No references to absolute memory locations
3. No negative constants or masks

The real time restrictions are quite straightforward, since all I/O is handled by the Monitor; breakpoint switches (sense switches) are reserved for Monitor use, and halts are strictly anathema in real time work.

The first set of relocatable binary restrictions are the familiar ones inherent in relocation and the second restriction, forbidding absolute addresses, is also fairly standard in this kind of work. The third restriction, however, is somewhat unusual and arises from the unique manner in which relocation is handled in the SDS 920. A brief look at the makeup of an SDS 920 instruction word will serve to illustrate this.

A 24-bit instruction word in the 920 is broken down as follows:

```
<table>
<thead>
<tr>
<th>R</th>
<th>X</th>
<th>O</th>
<th>I</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>
```

2s Bits

The fields are, from right to left, a 14-bit address field, one bit for indirect addressing, a 7-bit operation field, one bit for indexing, and finally, the left-most bit which is not interpreted when the instruction is executed. That is, the instructions 0 35 01000 and 4 35 01000 are interpreted by the computer as being identical. Since the address field is the only one that need be relocated in any instruction, the left-most, or sign bit of the word may be used for this purpose, and this is, in fact, the scheme commonly used on the 920.

For data words, however, the full 24 bits are used and it is for this reason that Experiment Processors needing negative constants or masks with a left-most one must generate them. This is the only real coding restriction placed on the programmer and in practice is not serious, since the 920 has the capability of forming both 1's and 2's complements with single instructions.

The Static Checker accepts a symbolic deck and produces a listing containing error diagnostics, if any (Fig. 6).

**Formatting**

It was apparent that the most straightforward way to maintain the Experiment Processors for use with the real time system was in the form of a library on magnetic tape. In line with this, two fundamental decisions were made concerning the physical makeup of these processors, both related to the efficient use of the computer's memory at execution time: they were to be in relocatable binary (as previously mentioned) and they were to be broken up into uniform-length blocks. Since it was considered unreasonable to require programmers to code their Experiment Processors in this latter manner, it was necessary to devise a method to take the programmer's processor, coded in a single sequence, and convert it to the desired form automatically.

In addition to checking the Experiment Processor, this conversion to uniform-length blocks is also provided by the Static Checker during the same run. This was easily facilitated, since the Experiment Processor is input to the Static Checker in symbolic form and the necessary additional instructions and pseudo-
operations may be inserted in that form (Fig. 7).

It will be noted that a few extra things have been inserted as well. For instance, the Experiment Processor obtains telemetry and other data from the Monitor by referencing cells which are absolutely located in the Monitor. Therefore, these locations are defined by BOOL pseudo-operations inserted by the Static Checker; the programmer need only refer to them symbolically. The Prelink and Postlink, which are bookkeeping linkages to the Monitor, are also inserted at this time.

The breaking up of the consecutive sequence into uniform-length blocks is effected by inserting an ORG (origin definition) pseudo-operation at fixed intervals. The actual connection between one block and the next is handled by a 'dynamic link,' the BRU (unconditional branch) instruction which is inserted at the end of each block. The advantage to be gained by this approach is that the blocks of a processor need not be contiguous in memory, and in practice this is usually the case. ORG's are used to define the start of each block in order to ensure that the result of assembling the symbolic output of the Static Checker is a series of paper tape records of uniform length, simplifying the maintenance of the library of Experiment Processors on magnetic tape.

The problem of pseudo-operations which define more than one memory location, such as:

- DEC 1,2,3  Places decimal 1-3 in 3 successive locations
- OCT 4,5,6,7,10 Places octal 4-10 in 5 successive locations
- BCI 4,ALPHA FIELD Places BCD characters in 4 successive locations
- BSS 20  Reserves 20 successive locations.
is handled by a special diagnostic in the Static Checker which flags any such pseudo-operation which causes the 'straddling' of two blocks as a coding error. The programmer must then rearrange his processor so that the locations defined will fit into one block.

Other Service Programs

The Static Checker has a companion program, the Dynamic Checker, which works with an Experiment Processor in Machine Language: the result of assembling the output of the Static Checker. The Dynamic Checker is essentially a simulator which provides the opportunity to debug an Experiment Processor under conditions as similar as possible to those of the real time Monitor. The Experiment Processor is embedded in the Dynamic Checker and is 'fed' telemetry frames via the same communication cells used in the real time case. The output of the processor is printed for subsequent analysis by the programmer.

The Dynamic Checker uses for input a magnetic tape containing simulated telemetry frames. Its modus operandi is to read a frame, branch to the Experiment Processor, and check to processor's communication cells for commands and/or output, repeating this cycle until the tape end-of-file is reached. The input tape is generated by another service program, the Tape Builder, which contains a pseudo-random number generator and which, essentially, generates a magnetic tape of simulated telemetry frames containing nothing but white noise. This is to check the Experiment Processors for one of their basic requirements: that they be capable of accepting any data and still not 'blow up.' The Tape Builder, however, has the provision of accepting programmer-coded subroutines which allows the data generated by these subroutines to be inserted in specified words in the telemetry frames. In this manner, the programmer may insert realistic data in the words of the telemetry frame which are analyzed by his Experiment Processor and thus may use the Dynamic Checker as a diagnostic tool.

Like the Static Checker, the Dynamic Checker has a second function in addition to a debugging one. This is the function of timing the Experiment Processor. One of the two interval timers available on the SDS 920 is used for this purpose; the result being that, at the end of the run, the Dynamic Checker prints out the worst-case (longest) execution time in machine cycles for the Experiment Processor being tested. As will be seen later, the two parameters: number of blocks (given by the Static Checker) and worst-case execution time (given by the Dynamic Checker) are the ones which will determine whether this particular processor may be loaded and executed at any particular time when the real time system is running.

In addition to the above service programs, there is, of course, a Tape Librarian program whose task is to maintain the library tape of Experiment Processors; adding, deleting, or replacing processors as appropriate. In addition, the Tape Librarian includes a computed checksum with each block written on the library tape. These service programs taken together provide a means of carrying an Experiment Processor from the coding stage through inclusion on the library tape in a fairly automatic manner (Fig. 8).

Loading Experiment Processors in Real Time

The Monitor has two restraints which determine whether an Experiment Processor may be loaded.

![Figure 8. Flow of Experiment Processor from Coding to Library Tape.](From the collection of the Computer History Museum (www.computerhistory.org))
added at any given time: whether there is room in memory and whether there is enough time to execute it. The portion of memory available for use by Experiment Processors, is called free storage and is allocated in blocks of the same size as the subdivisions of these processors. The amount of time available for Experiment Processor execution is called slack time and is the number of machine cycles presently not being used in the 18 ms. interval between successive frames of telemetry. A parameter table is maintained for use by the Real Time Load Processor containing one word for each processor on the library tape. Each parameter word contains two fields which specify the amount of memory (in blocks) and amount of time (in machine cycles) required by that particular Experiment Processor. When a request is entered into the computer to add a particular Experiment Processor, this request is handed off to the Real Time Load Processor which may easily check whether there is enough free storage and slack time to accommodate this Experiment Processor. If not, the request is placed in a special stack called the Request Stack. If the processor can be added, however, the Load Processor positions the library tape, reads the Experiment Processor into memory, relocates it, and connects it to the Monitor. Actually, the Load Processor makes requests to the Monitor for every tape operation given and it is the interrupt that occurs after each record is skipped or read which causes the Load Processor to be entered again. (This includes backspacing and rereading in case of a checksum error.)

If the request given to the Load Processor is to delete an Experiment Processor, either because this request has been made from the console or because the processor has signalled the Monitor that it has finished its run, then the Load Processor locates this processor, disengages it from the Monitor, returns the processor's blocks to free storage, and adds the processor's execution time to the slack time count. The Load Processor then checks its Request Stack to see if there are any requests to add processors which now may be accommodated.

The above description is considerably abbreviated and does not cover some of the special considerations of the Load Processor as actually written, such as error checking, backspacing, and rereading due to checksum error, etc. Also, although it is convenient to speak of 'picking up' blocks of free storage and 'returning' them to free storage, nothing is moved around in memory the change is actually one of allocations.

The basic pieces of information the Load Processor works with are four in number (Fig. 9). PARAM is a parameter table giving the storage and time requirement for every Experiment Processor on the library tape. The block count is in the operation field and the execution time is in the address field. Thus, as illustrated, Experiment Processor 34 requires 3 blocks of storage and has a worst-case execution time of 700 cycles.

Figure 9 also shows SLACK, which is a cell containing the available slack time in cycles, 1300, here. FREE is a cell which refers to Free Storage, the fourth element. It can be seen that an elementary list processing technique has been employed here. The address of the first

![Figure 9. Elements Used by the Monitor Load Processor.](https://www.computerhistory.org)
word in each block in free storage contains the starting location of the next so that they are chained together with the address of FREE giving the starting location of the end block in the chain. Furthermore, from bottom to top in the chain, the operation field of the first word in each block contains a serial count, starting at zero. This count is also carried over to the operation field of FREE and thus this cell not only provides a link to all of the blocks in free storage but also contains the number of blocks in the chain. The linking of blocks shown is somewhat idealized. In actual practice, these links become quite ‘scrambled.’

Under this scheme, the process of testing whether a requested Experiment Processor may be added is handled quite easily, simply by using its parameter word from the table PARAM: the address field of the parameter word (execution time) must be less than or equal to the address field of SLACK, and the operation field of the parameter word (block count) must be less than or equal to the operation field of FREE.

Figure 10a. ‘Removing’ Blocks from Free Storage—First Step.

Figure 10b. ‘Removing’ Blocks from Free Storage—Second Step.

Figure 10c. ‘Removing’ Blocks from Free Storage—Third Step.
Adding an Experiment Processor

In order to add an Experiment Processor, blocks must be 'removed' from free storage so that the processor may be read into them (Fig. 10). The 'removal' of a block is a two-step process, in which the address of FREE is stacked in a table called START and the contents of the location given by that address (the starting location of the block) is then stored in FREE. FREE now refers to the next block in the chain, and this process may be continued as often as necessary. This is, of course, the familiar 'popping up' of list processing and here it is done three times since Processor 34's parameter word calls for three blocks. The final step is to subtract the execution time in the processor's parameter word from SLACK, setting it to the unused time now available.

The table START is used for two things. First it is used to specify the blocks of memory into which the Experiment Processor is to be read. START is also used for relocation with a companion table BITS which is generated to contain successive multiples of the block length, starting at 0. Since the Experiment Processors are assembled with a base origin of zero, it can be seen that BITS consists of the unrelocated starting location of each block. Figure 11 shows how this relocation is carried out on a sample instruction.

The method of relocation allows an instruction in any block to reference a location in any other block occupied by that processor. The dynamic link between blocks could then be effected by the final BRU *+1 in each block, simply by flagging these instructions for relocation. In practice, however, the Tape Librarian program replaces these terminal branches with computed checksums when an Experiment Processor is added to the tape library. The Load Processor uses these checksums to determine whether the Experiment Processor has been loaded properly, with the usual re-reads in case of failure. When the processor has been successfully read in, the relocation sequence in addition replaces the checksums with the original branches, connecting each block to its successor and the last block to the Monitor.

It might be pointed out that there is no fixed requirement that the blocks have a length 200 nor, in fact, need their length be a power of two, although this simplifies the relocation scheme. This figure was chosen as a convenient length and may be changed if experience with the system warrants it.

Deleting an Experiment Processor

Figure 12 shows how the blocks of the Experiment Processor added in the example above are 'returned' to free storage. The starting location of the processor is obtained from the

---

**Figure 10d. 'Removing' Blocks from Free Storage—Final Condition.**

**Figure 11. Example of Relocation.**
module in the priority table to which it was assigned and placed in a working cell, called DUMMY here. Since the terminal branch at the end of every block provides a link to the next block, this 'pushing down' process can be carried out in a four-step process analogous to the one in which blocks were 'removed' from free storage. The block count is updated in the process so that at completion FREE again contains the correct block count when the number of blocks given by the processor's parameter word are 'returned' to free storage. Finally, the processor's execution time is added to SLACK to set it to the unused free time now available.

As shown by these examples, program blocks are never actually moved in memory; they are
simply read into the first available blocks and, since the blocks are dynamically chained with branch instructions, these blocks need not be contiguous in memory. A processor is deleted essentially by disconnecting it from the Monitor and allowing the storage it occupies to be reused. Since the Tape Librarian program converts all block reservations in an Experiment Processor to the corresponding number of zero words when the processor is added to the library tape, these memory blocks need never be cleared.

Conclusion

We feel this project is of interest since it represents an unusual situation in data processing. This is due to the fact that, while there was a requirement to selectively process different data points on demand, there was not a need to process all of the data available. Given the selection of the specific computer, however, this project attempted to optimize its use by providing the capability of doing the maximum useful work that memory space and processing time will allow. In this paper we have pointed out techniques to minimize the execution time of a powerful Real Time Monitor and to allocate reusable storage in a flexible and efficient manner. While using the computer to sample and monitor data is not a typical data processing application, it is hoped that this paper has indicated the efficiencies which we feel are inherent in the real time approach.