The Executive and Interpreter

INTRODUCTION TO THE EXECUTIVE

In the most elementary computers, executing a program is a relatively easy task. Without having to accommodate multiple programs with potentially conflicting requirements, the user only has to load the instructions into memory, set the starting address, and watch it run. Indeed, for early computers (and specialized controller chips today), this “load and go” methodology works acceptably well. For the first decade of the computer age, the modern concept of an operating system simply did not exist. Programmers were required to write software to operate tape drives, printers and other peripherals in addition to the logic required to solve the business or scientific problem at hand. Over time, primitive “executive” programs were introduced to reduce the burden of managing hardware. Rather than directly commanding a device such as a card reader, a program would invoke a pre-written routine to operate the external hardware on its behalf. Delegating such “low level” operations to standardized and (hopefully) reliable interfaces improved productivity tremendously by eliminating the tedious chores of system management.

Simultaneously, software products called “compilers” created the ability to express a problem in a language that better reflected the problem to be solved. The introduction of high level languages such as FORTRAN and COBOL provided a huge improvement in programmer productivity, and had the additional advantage of insulating the programmer from the idiosyncrasies of the underlying computer architecture. Of course, prepackaged system routines and high level languages come with a price: Additional memory is needed to store the executive software, and the code generated by compilers is not as efficient as hand-coded machine instructions. Nonetheless, the huge boost in productivity and quality of code greatly outweighed the additional hardware cost.

Running a single program at a time was perfectly acceptable in the early 1960s, where hour-long turnaround times were the norm. In a real-time computer such as the AGC, users do not have the luxury of feeding cards into a hopper, mounting and unmounting tapes, and performing multiple software loads to run a single program.
By its very nature, a real-time system must have a short response time while it processes the constant stream of data coming into the computer. Additionally, the variety of tasks under the computer’s control demands that a number of programs run in parallel. These requirements help establish two notable characteristics of control computers: memory resident software and multiprogramming. Given the limited amount of memory in the AGC, it would be reasonable to assume that much of the software would be stored offline on magnetic tape, with programs being read from tape as needed. This is a practical solution, and was demonstrated in the latter part of the Gemini program. However, tape is bulky, slow and not always reliable, making it undesirable for the rapid response needs of a real-time system. Eventually, memory technology progressed to where the mission software could (just barely) be squeezed into a set of read-only magnetic cores. With all of the programming now resident in memory and accessible at all times, there was no need for a tape system and the complex hardware and software required to manage it.

Multiprogramming, the ability to run multiple programs in parallel (all of which are demanding attention), is key to efficiently managing the diverse problems of spaceflight. It is easy to visualize the different types of problem that a computer must solve simultaneously. Perhaps the best example is during the lunar landing, where the processing demands are the highest. First, the spacecraft must know where it is and how it is moving, so a set of variables called the state vector must be continuously updated. Next, the Digital Autopilot must ensure that the spacecraft maintains the desired attitude commanded by the crew or another program. Providing this attitude data is the job of the landing guidance routines, which calculate targeting solutions based on position, velocity, attitude and engine performance data. At the same time, continuously recalculating the optimal abort trajectory might also be desirable. All of these tasks and many others must execute concurrently. While it is possible to write a large, monolithic program to do all of this, the development and testing of such a monster is difficult at best, and unmanageable at worst. A basic principle of programming is to divide the overall task down into smaller, more manageable modules that are far more reliable, verifiable and maintainable.

Although breaking software down into smaller modules greatly reduces the design and development effort, many of these programs must still run at more or less the same time. With the requirement for so many simultaneous operations, how does a single-processor computer perform them? The magic of running several programs at once, or multiprogramming, exploits a combination of processor speed, the relative slowness of human perceptions, and the reality that not all processing is so time-

---

1 Storage of AGC software on tape was seriously considered, and was close to becoming a reality. Eventually, the project was cancelled, and development did not progress past the prototype stage.

2 Note that multiprogramming (also known as multitasking or time-sharing) is very different from multiprocessing, which uses multiple CPUs within a single computer. Multiprogramming, multitasking and time-sharing are all terms describing the “simultaneous” execution of software on one processor. We will use these terms interchangeably throughout this chapter.
critical that it cannot be interrupted. Insisting that multiple programs execute simultaneously is rarely necessary, and in the case of single-processor computers, physically impossible. Fundamentally, the most important requirement in a real-time computer is being able to handle the worst case processing demands, which in turn drives all the other basic system requirements. With sufficient processing power, the most critical application can finish well before its deadline, with the “leftover” CPU cycles available for other work. Assuming that the hardware is powerful enough to run all of the work within prearranged deadlines, it is possible to ask basic questions about how work is processed. Does it really matter which process runs first? What is the impact if one process hasn’t finished executing, and we turn the CPU over to another process? What mechanisms are available for dispatching work to the CPU?

SCHEDULING: PREEMPTIVE AND COOPERATIVE MULTIPROGRAMMING

Scheduling work: Christmas at Macy’s
In life, as in a computer program, no-one likes to wait. The image of a harried sales clerk trying to accommodate countless impatient customers during the holiday season is almost too painful to witness. How is it possible to manage this situation with some degree of “fairness”, however that might be defined? The frazzled clerk could take the customers one at a time and in sequence, with no regard to how much attention each person might require. This “first come, first served” model is the most basic form of scheduling “fairness”, yet it ignores the fact that someone might have only a quick question, or is likely to spend a large amount of money. At this point, it is time to move past the “first come, first served” philosophy, with the concept of fairness evolving to include the notion of priority. How much should priority play when selecting customers for service? Next, what should the salesperson do when the customer’s attention is directed elsewhere, such as when they leave to try on their new clothes? Should the clerk wait dutifully by the fitting room, or take the opportunity to help another customer? When assisting a customer, what should be done if another person charges to the head of the line, demanding to be serviced immediately? Or, should our enterprising employee simply limit each customer to one minute of time, before having to rejoin the line at the end, serving them in a “round-robin” fashion?

Our little excursion into the retail world perfectly mimics the issues in scheduling work in a computer. Fundamentally, there is the need to manage several conflicting goals: all customers want service immediately and exclusively, some customers are more important than others, and interruptions will always occur when attending to someone else. The ability to service only one person at a time is acceptable only if there is no-one else waiting. As new customers arrive and are forced to wait, the ability for the store to sell merchandise diminishes. A system capable of running only one program at a time faces the same dilemma. Assigning a priority to each task (or the interrupt level) improves the situation somewhat, as some important work is dispensed with quickly, but this is little comfort to the person at the end of the line.
With several customers (programs) demanding attention at once, there is the need to invent a way of dividing the available time amongst them.

Cooperative multiprogramming
There are two different mechanisms available to implement multiprogramming, each with its particular advantages and disadvantages. One of the earliest and most straightforward techniques is “cooperative multiprogramming”. This technique is a reasonable choice for imbedded systems, where resources are small and programs are designed to strictly adhere to the multiprogramming architecture. As its name implies, the programs actively participate and cooperate to implement the multiprogramming environment. All programs periodically surrender control back to the Executive, which reviews the priorities of all pending tasks and selects the highest priority task to run. In this environment, the Executive’s role is reduced to little more than creating, tracking and terminating jobs, and dispatching their work. Dispatching work is the act of turning control of the processor over to a program, allowing that program to begin or resume execution. In a computer that has only one processor, this program is now the only work that is executing. All other programs, including the Executive, are idle, awaiting their turn at the processor. There are many sophisticated algorithms available for dispatching work. One is to presume that the dispatching decisions are based only on priority.

In priority-based multiprogramming systems, there are no assurances that all work waiting to run will in fact be serviced within any given time. As a result, there is the very real possibility that a low priority job will wait indefinitely for service. A practical solution is for the active job to voluntarily lower its priority after its critical processing has completed, leaving itself available for execution once any backlogged work is processed. This type of multiprogramming has additional advantages for constrained systems like the AGC. Hardware and software complexity, and by extension, costs, are usually minimized by the sparse design of the system. Very little time is spent in executive routines, thus improving overall throughput. Despite its simplicity, there are advantages of a design where priority is the sole determinant of whether a job will be run. Multiprogramming systems have the responsibility for managing access to shared resources, such as memory areas containing especially critical data. While executing, a program may be using such a resource, and cannot allow other programs access to it for a period of time. Only after the higher priority work has finished (or lowers it priority) will a lower priority job start and have access to the resource. In effect, the Executive’s priority scheduling acts as a lock on the resource. The ability to safely use a shared resource without a locking mechanism exists only if programs are carefully prioritized so that a job using that resource has the highest priority of any other job that might also be scheduled at the same time.

There are significant constraints to using cooperative multiprogramming. Unsurprisingly, the technique is completely dependent on a program’s willingness to cooperate and return control back to the Executive on a regular basis. Applications must be designed with this cooperation in mind, and must be written with the perspective of how it will interact with other programs. The consequences of a problem occurring in cooperative multiprogramming are serious. If a program is
overly busy working on a problem, or if a software bug prevents it from returning control to the Executive, no other work can reliably run.

**Preemptive multiprogramming**
Preemptive multiprogramming is the second major strategy for managing work in a computer. Unlike cooperative multiprogramming, where the application actively participates in the scheduling of work, programs in a preemptive system do not play any part in determining how or when they will run. Rather than depending on the application to make timely calls to the Executive, a timer generates an interrupt that suspends whatever process is running at that time. These timer interrupts are the basis behind the phrase “time-slicing”, where processing during an interval is spread across several tasks. By design, the application programs are blissfully ignorant of being alternatively scheduled and then interrupted. Each time an interrupt occurs, the state of the currently running program is saved and the Executive scans the list of waiting work. Preemptive systems usually try to give all jobs access to the processor even if they are of low priority. Because a job may be interrupted at any time, there are no provisions to proactively save data or otherwise put the job in a known state before the interrupt occurs. Also, since any job can be dispatched at any time, a process usually cannot have an expectation of completing by a specific deadline.

Strictly speaking, the AGC does not have a formal preemptive multiprogramming Executive. Still, a number of timer interrupts do make calls on the Executive to start jobs. When this occurs, the entire job queue is scanned for the highest priority work, as in a purely preemptive Executive. In fact, the AGC’s Executive is best considered as a mix of both cooperative and preemptive multiprogramming techniques.

**THE EXECUTIVE**

When discussing the features and functions of a computer, our impressions are usually defined by the capabilities of its operating system, or OS. Operating systems are designed to provide a rich set of services to user application programs, such as memory management, scheduling and prioritizing work, error recovery and I/O processing. Upon receiving a request to run a program, a modern OS gathers together the large number of resources necessary to create an execution environment. A block of memory, or an entire virtual address space is allocated, data files are located, and security parameters are identified. The program is loaded from secondary storage into memory, with the carefully built environment ready to receive it. When the application has terminated, it is up to the OS to clean up the mess, releasing the allocated resources for reuse by the next program. Managing these functions requires manipulating a staggering amount of internal tables, queues and buffers, creating a hugely complex system. Of course, none of these functions are relevant in the AGC. Unlike a 21st century general purpose computer, the AGC has the huge advantage of operating in a tightly defined world of hardware and software. Here, rules for application behavior are well understood and rigidly enforced, and reliability is ensured through rigorous testing. All resources required by the software
are anticipated during its development, and are already available when program execution begins. External hardware comprises a few basic device types, all of which use very few instructions to interact with. Such careful design results in software that can be “trusted” (for lack of a better word), eliminating the need for elaborate mechanisms to isolate the hardware and OS from the application. By eliminating or limiting the scope of services that the AGC provides to the mission software, a very small and efficient operating system is quite practical.

This system, simply known as the Executive, is the heart of the AGC. It provides capabilities that are advanced even by modern standards, especially in view of the limited amount of hardware available. In this section, we will focus on the major services provided by the Executive:

- Job initiation, termination and scheduling
- Memory allocation
- Restart and error recovery (program alarms)
- User interface (DSKY)
- The interpretive language and its “virtual machine”
- Event timing
- I/O.

Much of an operating system’s design centers on two basic assumptions: the OS must manage the execution of application programs and provide them with necessary resources, and the OS should protect itself and other applications from any errors or faults that might occur. Large commercial computers, with their abundance of processing power and memory, achieve these goals with considerable effort. However, the AGC is especially notable in its ability to effectively manage the work running in the computer, and its sophisticated restart and recovery design. The lack of hardware resources in the AGC placed a practical limit on what capabilities were available to the Executive. Still, the Executive provided services for the programmer that were sophisticated enough to manage multiple programs, tasks and hardware devices.

Scheduling and dispatching in the AGC
When executing, the representation of a program to the Executive is through data structures containing the essential information about the process. A modern operating system might include data structures for process identification numbers, references to allocated resources, execution priority, entry point addresses and so forth. Creation of this data is the foundation for defining the existence of a process – without it, by definition, there is no program to run. In the AGC, the 12-word data area known as the Core Set contains all of the information necessary to manage the execution of a program. Very little information needs to be placed into the Core Set area. Process information occupies only five words of each Core Set, with the remaining seven words available to the job for temporary variables. The five words used for process management contain the program’s priority, entry point address, a copy of the BBANK register, various flags and a pointer to the Vector Accumulator (VAC) area, if one has been requested. The seven words in the temporary area have a
dual functionality, depending on the type of code executing. When basic AGC instructions are running, the seven free words in the Core Set are unstructured, allowing the programmer to use the storage as desired. Interpretive programs use this area for arithmetic and logical operations. Now a highly structured memory area, these seven words become the Multipurpose Accumulator (MPAC), which performs duties similar to the AGC’s hardware accumulator. For convenience, when a non-interpretive program is running, these seven words are still referred to as the MPAC.

**Requesting a basic job: NOVAC**

Five words of memory in the Core Set contain the basic information of a program executing in the AGC. A Core Set represents the program to the Executive, and contains all the status information necessary to determine if the program is ready for dispatching or not. Several Core Sets are stored in a contiguous table in unswitched erasable memory, with six available for the Command Module and seven for the Lunar Module. As entries in a table, Core Sets are numbered, with Core Set 0 as the first entry in the table, followed by Core Set 1 and so on. An additional Core Set is reserved for the “DUMMY” job, which “executes” only when there is no other work available to run. Available Core Sets are identified by a negative-zero ($77777_{16}$) in the priority field, making it easy for the Executive to scan for an available entry. Small efficiencies, such as organizing Core Sets into “available” and “in use” queues are not practical as the length of the table is so short that there is little benefit from the additional complexity.

To schedule a basic job (that is, one that does not call on the Interpreter) for execution, the starting address of the new job and its priority are passed to the NOVAC executive routine. NOVAC then scans the lists of Core Sets to find one that

<table>
<thead>
<tr>
<th>MPAC</th>
<th>00_{16}</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPAC+1</td>
<td>01_{16}</td>
</tr>
<tr>
<td>MPAC+2</td>
<td>02_{16}</td>
</tr>
<tr>
<td>MPAC+3</td>
<td>03_{16}</td>
</tr>
<tr>
<td>MPAC+4</td>
<td>04_{16}</td>
</tr>
<tr>
<td>MPAC+5</td>
<td>05_{16}</td>
</tr>
<tr>
<td>MPAC+6</td>
<td>06_{16}</td>
</tr>
<tr>
<td>MPAC Mode</td>
<td>07_{16}</td>
</tr>
<tr>
<td>LOC (Program Ctr)</td>
<td>10_{16}</td>
</tr>
<tr>
<td>BANKSET (BBANK)</td>
<td>11_{16}</td>
</tr>
<tr>
<td>PUSHLOC</td>
<td>12_{16}</td>
</tr>
<tr>
<td>PRIORITY / VAC Ptr</td>
<td>13_{16}</td>
</tr>
</tbody>
</table>

**Figure 30: Core Set layout**
is available. Careful design and exhaustive testing of the AGC software demonstrated the maximum number of Core Sets to be sufficient for all mission phases, so NOVAC should never find the Core Set table full. On finding an unused Core Set, NOVAC stores the starting address and banking information, job priority and other data, leaving the job ready for scheduling. Now that the new job is ready to execute, the next step is to determine if its priority is higher than the currently running job. If so, the Executive must suspend the currently running work and allow the new job to begin processing.

Core Sets are very dynamic data structures, and are created and deleted as programs start and end. With one exception, there is no ordering to the jobs in the Core Set table, and executable work does not have to reside in consecutive Core Set table entries. The exception is the first entry of the Core Set table. Core Set 0 is always assigned to the job that is currently executing, with the other Core Sets representing jobs that are not ready to run or are of a lower priority. Since this property is fixed in the Executive’s architecture, any job that wishes to run must swap its Core Set with the currently running job. Information necessary to resume a new or suspended job is stored in the Core Set, so dispatching requires little more than swapping the Core Sets of the new and current job. Note that during a job’s lifetime, its process information may be stored in any of the Core Set areas when it is not active. As jobs with higher priorities start and end, it is not possible to know in advance where an inactive job’s Core Set will reside.

---

3 And yet, it did happen. A full Core Set table caused the 1201 program alarms during the Apollo 11 lunar landing. The effect of this alarm, and the equally nefarious 1202 program alarm (which indicated that there were no Vector Accumulator areas available), is discussed in the restart/recovery section.
**Requesting an interpretive job: FINDVAC**

Seven words of temporary storage might be viewed as ludicrously small, yet work is broken down sufficiently so that the Multipurpose Accumulator in the Core Set is sufficient for small jobs. This is not the case when using interpretive instructions in the program. As presented later in this chapter, the Interpreter creates a “virtual machine” with an architecture and instruction set that is unrelated to the underlying AGC hardware. The Interpreter has far larger space requirements for temporary variables than programs running in native AGC code. To satisfy this demand for storage, a second area in unswitched erasable storage, the Vector Accumulator area is available. Similar to Core Sets, the five VACs are also arranged in a table and each provides 43 words of storage to the job. The first word of each VAC area is reserved to indicate whether the VAC area is in use or not, leaving 42 words available to the application. When scheduling a job containing interpretive instructions, a call to the FINDVAC Executive routine is used rather than NOVAC. FINDVAC scans the list of Vector Accumulator areas, looking for one that is available. Once found, the VAC area is marked as “in use”, but a job cannot run with only a VAC area defined. Now that a VAC area has been acquired, FINDVAC’s second task is to use the NOVAC code to obtain a Core Set area for the new job. The address of the VAC area is placed in a word within the Core Set just allocated, along with the other job information, and the job is ready to be scheduled.

<table>
<thead>
<tr>
<th>Inuse Flag</th>
<th>00_B</th>
</tr>
</thead>
<tbody>
<tr>
<td>33 Words for stack and temporary variables</td>
<td></td>
</tr>
<tr>
<td></td>
<td>V(MPAC)</td>
</tr>
<tr>
<td></td>
<td>V(MPAC)</td>
</tr>
<tr>
<td></td>
<td>V(MPAC)</td>
</tr>
<tr>
<td></td>
<td>V(MPAC)</td>
</tr>
<tr>
<td>Index Register S1</td>
<td>46_B</td>
</tr>
<tr>
<td>Index Register S2</td>
<td>47_B</td>
</tr>
<tr>
<td>Step Register S1</td>
<td>50_B</td>
</tr>
<tr>
<td>Step Register S2</td>
<td>51_B</td>
</tr>
<tr>
<td>QPRET</td>
<td>52_B</td>
</tr>
</tbody>
</table>

| | V(MPAC) | 2 stored after UNIT and ABVAL instructions |
| | V(MPAC) | stored after UNIT instruction |

**Figure 32: Vector Accumulator area layout**
Figure 33: Allocated Core Sets and VAC areas

At this point, a Core Set is now available, plus a VAC area if needed. The priority, along with the program's starting address is saved within the Core Set, making the job ready to run. However, this does not mean the job will begin executing immediately. All Executive requests, such as the FINDVAC and NOVAC just described, require scanning the Core Set table to determine the highest priority job. The result is placed in an erasable storage location named, appropriately, NEWJOB. When NEWJOB is equal to positive-zero ($00000_8$), the currently running job has the highest priority of any job in the AGC. A non-zero quantity indicates not only that a higher priority job is ready to run, the value saved in NEWJOB is also the address of the Core Set (and by implication, the program) which should be run next. Simply having a non-zero value in NEWJOB, however, does not mean that the active job will change immediately, nor that the new, higher priority job has been swapped into the Core Set 0 location. After an Executive call finishes, it resumes the current job, despite the fact there might be a higher priority job waiting. It is up to the active job to interrogate the NEWJOB word and begin the process of starting the new job.

The NEWJOB word is a unique case of hardware and Executive software integration. Most words in erasable storage can, within reasonable limits, be located anywhere in memory. Practical requirements might force some data into the unswitched erasable storage area, and many words must be in contiguous storage locations, but there is usually no requirement for data to reside in specific memory locations. This is not the case with NEWJOB. The hardware passively monitors the NEWJOB storage word, which is strictly defined as location $00067_8$. Although the Executive uses and references NEWJOB no differently than it does any other
memory location, the hardware maintains a special vigil on its use. For cooperative multitasking to work, a regular check for other jobs waiting to run is necessary, which requires a reference to the NEWJOB word. Assuming the goal of testing NEWJOB every 20 ms is maintained, all is well with the system. However, if there is a problem in the software, the error might prevent the regular interrogation of NEWJOB. In this case, there would be no checking for a new job, and the current job would run forever, preventing any other work from gaining access to the processor. The hardware allows 640 milliseconds to pass before it acts to clear this situation by invoking a restart of the entire system. Called the Night Watchman by the engineers, it presumed that the software “punched the clock” (e.g., checked the NEWJOB word) indicating it was still awake and operating normally.

Checking for new work
Now that the new work is scheduled and ready to run, control returns to the currently active job. In using cooperative multiprogramming to dispatch new work, the new job must wait until the active job queries NEWJOB to possibly relinquish control to a higher priority job. The timing for the test of NEWJOB is somewhat flexible, merely having the requirement that the check is performed at least every 20 milliseconds. Such flexibility allows the job to choose the point where control is potentially surrendered, providing an orderly transition from one job to the next. Of course, an exception exists with interpretive programs, where the advantages of interpretive instructions are offset by their long execution times. The interpretive instruction set, designed to implement an entirely different architecture than the native AGC, cannot easily test NEWJOB directly. The option of switching between the basic and interpretive instructions is not practical, owing to the additional overhead and disruption to program logic. A compromise is reached by having the Interpreter system itself check NEWJOB after executing each interpreted instruction.

Testing NEWJOB requires only a two instruction sequence, beginning with a Count Compare and Skip (CCS) instruction. Remembering from the instruction set section, CCS first decrements the contents of the accumulator, then skips to one of the locations following the CCS based on the value referenced by the instruction operand. When testing NEWJOB, we are interested in a positive-zero indicating that the current job is the highest priority job and no further action is necessary, or a value greater than positive-zero, defining the Core Set address of the higher priority job waiting to run. In this second case, the active job relinquishes control by calling the CHANG1 Executive routine. The coding sequence is:

    CCS      NEWJOB    IS HIGHER PRIO WORK WAITING?
    TC       CHANG1    > +0, CHANGE ACTIVE JOB
                = +0, WE ARE HIGHEST PRIO

The elegance of the Executive’s design, and how it interacts with the CCS instruction is apparent. The decision making for cooperative multiprogramming condenses into two instructions, only one of which executes if no higher priority work is waiting. This sequence is not immune from problems, as there is no sanity checking of the contents of NEWJOB before it is tested. Conceivably, it could hold negative-zero or
less than negative-zero, both illegal values that would cause unpredictable results. The FINDVAC and NOVAC routines initially test for invalid data in NEWJOB, but this cannot guarantee the data in NEWJOB has not been overlaid in the meantime. Despite this exposure, the instruction sequence for testing NEWJOB is a fast and efficient way to switch to the highest priority work in the system.

**Suspending the current job, and starting the new job**

The CHANGJOB Executive routine (whose entry point is CHANG-1) is called if the CCS test of NEWJOB recognizes there is a higher priority job waiting. CHANGJOB’s first task is to save the active job’s return address and banking information into its Core Set in preparation for suspending it. The actual suspension of the active job, which occupies Core Set 0, and starting the new job, requires only the swapping of their respective Core Set contents. After this is done, NEWJOB is set to positive-zero, as the highest priority job is now active. Finally, the starting address and banking information are loaded from Core Set 0 into the Z and BBANK registers. With the Z register pointing to the new program and the banking registers defining the program’s memory environment, the transfer of control to the new job is complete.

**Changing the job’s execution priority**

Jobs do not always have to run continuously at their original priority. During the lifetime of a program’s execution, higher priority work may be scheduled, or a critical routine in the job might have finished. As such, more efficient scheduling of work is possible when jobs dynamically adjust their priority to reflect the relative importance of their current work. Voluntarily altering execution priority is a useful element in cooperative multiprogramming, especially when recognizing that the current work does not merit monopolizing the processor. Changing priority is an effective tool only if all jobs are “well behaved”; that is, a job should alter its priority only with an awareness of its impact on other work in the system. Assuring this sort of good behavior is usually possible only during the design phase of software development, when establishing the priorities, sequencing and interrelationships of work running in the system. This requires that the developers of all the mission programs be aware of other programs that will be running at the same time. The benefit of allowing programs to manage their own priority is that elaborate job management schemes in the Executive are no longer necessary. However, the importance of tight controls on priority changing routines cannot be overstated. Where there are no restrictions, users could make inappropriate assumptions on the priority of their programs, often to the detriment of overall system performance.

Changing the priority of an active job is useful as a scheduling tool. There are instances where a job might need to run without being preempted and suspended by higher priority work. Inhibiting interrupts is one possible approach to this situation, but this brute force solution could easily prevent other desirable processing from occurring. Conversely, there are often occasions where there is no harm in letting a waiting job run. A job lowering its priority to below that of the waiting job is effectively suspending itself in favor of other work. A call to the PRIOCHNG Executive routine sets the priority of the job to its new value, and then performs a
scan of the Core Set table for the highest priority work. If a higher priority job is found, a call to CHANGJOB suspends the current job and dispatches the new work. Note that only the priority of the active job is changeable. One job cannot change another’s priority, nor can other work, such as waitlist tasks or other Executive routines change the priority of a job. Jobs are the only type of work in the AGC that need the ability to change their priority. Interrupt routines and waitlist tasks, by definition, run with interrupts inhibited, and consequently have complete control of the processor.

**End of job processing**

All good things must come to an end, even a job in the AGC. The end can come in various ways; through an explicit termination by a DSKY input, a restart, or by the most common means: the program itself calling the Executive routine ENDOFJOB to say “My work here is done.” As there is very little overhead in setting up a job to run, likewise only a small number of operations are needed to terminate and clean up after a job, with most of those tasks centering on preparing memory for reuse. Cleaning up the Core Set and VAC area (if applicable) is straightforward. A negative-zero ($7777_{8}$) is placed in the priority field of the job’s Core Set, indicating that it is available. If a VAC area was reserved, the address of the VAC area is saved into its VACUSE field, which flags that memory area as available. After the job’s memory resources are released, the job has effectively terminated. As the AGC does not have any other resources for a job to allocate, no further cleanup is necessary.

All of this work creates a small dilemma. In a cooperative multitasking system, it is up to the active job to test the NEWJOB location to see if a job is waiting, and then call the Executive to schedule the execution of the next job. As the active job has terminated, there is no longer any program executing to test NEWJOB, creating a situation where it is impossible to dispatch another program. To resolve this, the Executive itself performs a scan of the Core Set table and dispatches the highest priority work.

**JOBSLEEP and JOBWAKE**

Waiting for data input from the crew, or a hardware operation to complete, presents a problem in scheduling work in the system. When a process needs to wait for an extended period of time (a large fraction of a second or longer), simply looping and executing pointless code in an effort to “kill time” is inefficient at best. In cases such as these, it makes little sense for a program to remain active, and is far more practical to “put the job to sleep”, suspending it until the operation or data input completes.

Consider the case where a job requires the crew to enter data in all three DSKY registers. Even quickly punching in the data can take up to a minute – an eternity in computer time! The job waiting on the data could suspend itself, with the expectation that another task will awaken it at the end of the data entry sequence to continue processing. This requires a cooperative effort by another program; in this case, the

---

4 Waitlist tasks are short, high priority programs that run with interrupts inhibited.
program monitoring the keystrokes and waiting for the input to complete. Without an idea of how long it will take to input the data, simply setting a timer to check the status of the input is inefficient. Hardware operations such as torquing gimbals present another example as they do not generate an interrupt when their operation has completed. In this case, a job can arrange for its awakening when the gyros have repositioned.

Although the JOBSLEEP routine does not provide a facility for waking a job at a specified time in the future, another routine, DELAYJOB does provides this capability. When called with the desired amount of time to wait, DELAYJOB puts the job to sleep and schedules a waitlist task using the delay time to schedule starting the waitlist task. When the waitlist task finally executes, it calls JOBWAKE with the Core Set address of the sleeping job. Up to three jobs can be sleeping with delay times in the Lunar Module, and up to four in the Command Module.

With the basics of JOBSLEEP and JOBWAKE established, it is possible to probe deeper into how these operations are implemented. A sleeping job is defined as having a negative priority, so JOBSLEEP will complement this field in Core Set 0. In one’s complement notation, complementing is the same as multiplying by -1, so a job whose priority is 33 will sleep with a priority of -33, providing the benefit of preserving the priority’s original value. Then the address of the next instruction to execute after awakening (the instruction following the JOBSLEEP call) is placed into the job’s Core Set along with its banking information. With the job now asleep, it is no longer an active job, and so the Executive scans the Core Set list for the highest priority job. When one is found, its Core Set address is placed in NEWJOB, and the new work is dispatched through CHANGJOB. If no other jobs are ready to run, the DUMMY job is placed in Core Set 0, and the system waits for the sleeping job to be awakened.

CHANGJOB, as seen earlier, swaps the contents of the new and current jobs in the Core Set list, sets NEWJOB to positive-zero and transfers control to the job referenced at Core Set 0. Waking up a job is similar to scheduling in CHNGPRIO. First, the address of the sleeping job’s Core Set is placed in the accumulator before the call to JOBWAKE. JOBWAKE complements the priority of the job back to a positive value, indicating the job is awake and ready to execute. At this point, the Executive scans of the Core Set table for the highest priority job, which is followed by a call to CHANGJOB. Note that putting a job to sleep is not the same as suspending a job. A suspended job can execute at any time when it becomes the highest priority job in the system, whereas a sleeping job will become active only when explicitly awakened by another job or waitlist task.

A job can theoretically run forever, if its priority is high enough to prohibit any other work from running. This is not necessarily an undesirable behavior. Consider a critical job that must run to completion before any other work can start. Here, the goal is to give this job a high priority to prevent it from being suspended or delayed by other work. As the program’s importance is enough to justify a high priority, deferring the execution of other tasks is an acceptable outcome. The Executive does not take any action on a program that runs for a long time – as no accounting of processing time is ever done, it has no way of knowing if a job has been running for a
second, a minute, or an hour. The only requirement is that the job continues to check the NEWJOB storage location on a regular basis in order to ensure that there are no other waiting jobs.

Major Modes in the AGC
But how is a program in the AGC directed to run in the first place? Short running work such as an interrupt routine or waitlist task is dependent on timers or external events to begin its processing. Restricted to a few milliseconds at most, it must schedule a job if more lengthy processing is necessary. An active job can also schedule other jobs to run, which execute as soon as their priority becomes the highest in the system. The third and final mechanism for the astronaut to start a job is through DSKY commands, using a Verb sequence to change the “Major Mode” of the computer. A Major Mode is best defined as a very long running program (minutes to hours) that controls an important mission function such as launch, lunar landing or rendezvous. In fact, few attributes differentiate a Major Mode from other jobs, and they are generally managed the same as other jobs in the system. Perhaps surprisingly, jobs running as a Major Mode are scheduled at a very low priority, but the rationale for this becomes apparent on closer inspection. Major Mode programs cannot monopolize the system resources, and frequently schedule other jobs to perform small but important functions. Each check of the list of ready-to-run programs in the Core Set might reveal a new, higher priority job, and this suspends the Major Mode program until the new job completes. If the Major Mode program always maintained a high priority in the system, the work that it has just scheduled would never have the opportunity to run. Maintaining a Major Mode program at a low priority ensures that all the work that it schedules will always be given an opportunity to execute.

The waitlist
Control computers must regularly schedule work to perform time-critical functions. Checking panel switches to see if changes have been made, scheduling the Digital Autopilot to run, and even the mundane task of flashing a display off and on are mandated to run at predetermined times. The AGC maintains a “waitlist”, which is a pair of tables containing references for “tasks” that are scheduled to run in the future.

Waitlist tasks run in interrupt-inhibited mode, where no other work in the system can disrupt its processing. Since these tasks effectively prevent other programs from using the AGC, they must necessarily limit their execution time. An informal time limit of five milliseconds is imposed on the tasks. There is no mechanism to interrupt or otherwise enforce this limit – only careful design assures that the task will not exceed its allotted time. Five milliseconds is equivalent to executing 150 to 200 machine instructions, which is certainly not enough to perform complex operations. When a significant amount of processing is required, the task will schedule the execution of a larger job to perform the bulk of the work. The limited number of erasable storage locations available for temporary variables further restricts waitlist tasks. A few words are allocated in erasable storage for programs that run with
interrupts inhibited. These are only valid during the task’s execution, and there is no expectation of their contents being preserved after interrupts are reenabled. That the temporary areas are valid during waitlist processing is assured only because no other program can take control of the processor and reuse the storage while the task is running. Once the task has completed, the temporary storage is free to be reused by the next task in any way that it sees fit. Additionally, the A, L and Q special registers may be used, with the caveat that they are preserved (in the ARUPT, LRUPT and QRUPT areas) and restored before the task has terminated.

Two waitlist tables are identified by their contents. Table LST1 contains the time intervals used in scheduling the start of the tasks. LST2 holds the starting address and memory banking information. The LST2 tables are paired one-for-one with the entries in LST1, with the exception of the first LST2 entry. Normally contained in LST1, T3RUPT contains the time until the next waitlist task, which is counting down until the next interrupt. This exception leaves LST2 containing the address of the next waitlist task to run. The waitlist is aptly named, being just a list of tasks waiting for their scheduled time to run. As seen in the tour of the special registers, the TIME3 counter is used exclusively to schedule tasks in the waitlist. A time interval is placed in TIME3, which counts down one “tick” every ten milliseconds. After reaching zero and overflowing on the next clock tick, the T3RUPT interrupt is raised, transferring control to the waitlist management routine.

A waitlist task is scheduled by providing the start time and address with banking information. Again, “time”, as used by the TIME3 counter, is not in the absolute “time of day” sense. Rather, the time is defined as an interval to a moment in the future, measured in increments of ten milliseconds. Waitlist tasks are ordered chronologically, with the task next scheduled to execute at the top of the list. Each interval defined for a request is relative to the point of its scheduling. That is, if a task is to start 140 milliseconds from now, the first LST1 entry will not necessarily contain the value of 140 milliseconds. The time stored in the LST1 interval table is calculated as the sum of the contents of the TIME3 counter, plus all the intervals in the requests that are ahead of it. For example, a task needs to run 180 milliseconds from now and two entries already exist in the waitlist, both scheduled to run before this new request. A check of TIME3 shows that the next task will run in 30 milliseconds, and the second task will run 40 milliseconds after the first. Properly setting the execution time for the new task, 180 milliseconds from now, requires adding the 30 milliseconds remaining in the TIME3 counter to the 40 milliseconds that will elapse between the first and second tasks. Subtracting the 70 milliseconds of waitlist time from 180 milliseconds results in an interval of 110 milliseconds. The new entry is appended to the end of the waitlist, and will run 110 milliseconds after the task ahead of it executes.

With the tasks in the waitlist organized by start time, a more complicated case occurs where a new task is scheduled to run between two existing tasks. In this example, TIME3 contains a value of 20 milliseconds, after which it will start Task A. Task B is also in the waitlist, and will start 100 milliseconds after Task A. Finally, Task C needs to start in 60 milliseconds, meaning 40 milliseconds after Task A starts. To retain the proper sequence, task B is moved down one entry in the waitlist and
Task C is placed in the resulting gap. A problem now exists with task B, as its start time is now invalid and needs to be recomputed. With Task C now starting 40 milliseconds after task A, the new interval for Task B is 60 milliseconds.

Managing the waitlist is a straightforward matter of inserting and deleting entries in tables. After the active task has started, its entry in the waitlist is no longer needed and it is removed from the table. All of the remaining entries are moved up one position, and the TIME3 counter is set to the interval for the next request. A waitlist task is scheduled only once. If there is a need to run the same task according to a periodic schedule, the task is responsible for rescheduling itself for its next execution.

**Phase tables part 1: restart and recovery**

In complex systems such as spacecraft and aircraft, computer failures can have catastrophic consequences. The range of potential failures is broad, from a missed opportunity for a scientific observation to the destruction of the vehicle and loss of the crew. The need for a real-time system to maintain control of the vehicle is paramount, especially during critical mission phases. Failing this, the system must react to problems in a controlled manner, and if necessary degrade to a known and benign state.

The restart capability in the AGC is quite flexible, and ensures that processing can continue in all but the most extreme situations. If a job requires restart protection, it calls the PHASCHNG routine in the Executive. One or more parameters are passed to this routine to specify which of the predefined restart routines to use, along with any variable data the might be appropriate to that restart option. A restart uses one of several different strategies to recover from a program error. First, the job having difficulty can be terminated with no intention of restarting it. A second option is to
restart the job from the beginning, hoping that the problem will not recur, and performing little in the way of cleaning up data from the previous invocation of the program. A third and far more complex option, is to schedule work that performs data cleanup and reenters the program at a previously designated restart point. Such flexibility is necessary due to the complexity of the software used in the AGC, and the realization that not all program errors are necessarily fatal. By closely matching the recovery option with the problem, it is possible to create the level of robustness that is essential in a real-time system.

In a system as complex as the AGC, a wide range of problems might occur. Some are the result of innocent procedural errors or minor hardware problems, while others arise from unexpected logic errors or unforeseen interactions of hardware and software. While it is not possible or practical to analyze a problem in real time to determine the root cause of the failure, it is reasonable to create a general set of rules and procedures for recovery that are appropriate for each executing program. In cases where a routine has few interdependencies on other programs or data, it might be easier to restart the program from the beginning. A simplified example might be a routine that starts, reads distance information from a radar, displays it on the instrument panel, and terminates. If an error occurs before the processing completes successfully, it is usually easier to try to run it again. A brief moment of data loss should not be a serious situation. In other cases, restarting from the beginning of a program that performs a lengthy computation is not always practical, such as in the case of calculating a targeting solution for a major maneuver. A more appropriate restart process might be to reenter the program to restart the calculations from the last successful iteration and reuse the existing data wherever possible.

Figure 35: Addition of a new waitlist task
These scenarios are simplified examples of some of the possible approaches to software recovery. Given the large number of programs that are usually running at a given time, a single restart solution is usually unworkable. For a word processor to recover from a spelling error by rebooting the PC is certainly absurd; and it is equally inappropriate to simply report that the hard drive in your laptop has failed, and blithely continue on. To address the wide range of recovery requirements, the AGC’s Executive provides a restart mechanism that is dynamic and highly flexible. Note, however, that waitlist tasks are not defined with restart protection in mind.

Complexity in the restart system is managed by assigning programs to one of six “restart groups”. Once defined, a restart is performed regardless of whether a job is active, suspended or sleeping. Each group manages a logically related collection of jobs, and each group can only restart one job. As an example, restart group number four in the Command Module is responsible for both SPS burn programs and also for entry software. Clearly, no conflict exists because these two situations are mutually exclusive events – by the time of atmospheric entry, the SPS will have been jettisoned. Other groups are organized in a similar fashion, such as with Servicer programs handled by restart group two, and group five restarts assigned to IMU management. The absence of restart protection for the Digital Autopilot is notable. The DAP, executing every 100 ms, is an example of a program that is run without any recovery protection. A failure during one DAP cycle is not necessarily a danger, since it will execute again within a fraction of a second. If a serious problem in the DAP were to create a control problem, it would be manually disabled through a switch on main instrument console.

The restart table
The restart table is located in unswitched fixed storage and is organized into six restart groups. Within each group, one or more restart “phases” define the specific restart procedure within the group, each assigned to a unique restart location within the program. As a program progresses, and its restart requirements change, it informs the Executive through the PHASCHNG routine that a different restart phase is now in effect. As several different programs share the same group, an individual job might only use one, or perhaps a few, restart phases within the group. Each entry in the table defines a single recovery process, which is associated with only one program. There is no universal or all-encompassing restart table, and the contents in the Lunar Module version of the AGC software are considerably different from its Command Module counterpart. Four different restart options are available, and are defined through the attributes of the restart phase number. For clarity, it is easier to use the notation “G.P” to describe the restart point, with G as the group number, and P, the restart phase. Using this notation, the list of restart options are:

- G.0 Inhibit group restart
- G.1 Restart last display
- G.Even# Execute two restart routines
- G.Odd# (> 1) Execute one restart routine
In this notation, “even” phase numbers and “odd” phase numbers are exactly that; G.Even# include phases such as 2.2, 2.8, 4.6, and so forth. As G.1 is already assigned, the “odd” phases must begin with 3. An important effect of this convention is that both restart phases G.0 and G.1 are common to all groups, and neither requires defining a unique process to run. During a restart, each entry in the phase table is checked for a zero entry. If so, that restart group is not active and no additional processing occurs. Restart phase one redisplay the data on the DSKY, but does no other recovery processing. Only the remaining two types of restart phases, G.Even# and G.Odd# (> 1) will attempt to restart the program itself. Phases using odd numbers greater than one perform only one operation during that group’s restart processing, such as scheduling either a job or waitlist task. Two restart operations occur in even-number restart phases, and any combination of jobs or waitlist tasks are possible.

Each restart operation requires three words to define, with its contents depending on whether the restart action requires scheduling a job or a waitlist task. Odd-number restart phases performing only one restart operation will only occupy three words, while even-numbered phases require six words to reflect the two operations scheduled for that group. Entries for a job in the restart table require three words: one word for the execution priority, and two words to define the entry point address. Waitlist tasks have a word dedicated to their scheduled start time, and also use two words for their entry address. Although this format appears sufficient, it does not contain all of the information required to schedule its work. In the quest to minimize the amount of storage occupied by the phase table, the several flags necessary for specifying other program attributes are not stored in dedicated memory areas.

![Diagram of Restart and Phase Tables](image)

**Figure 36:** Restart and phase tables
Rather, these flags are implemented as encodings of the three restart table words passed to PHASCHNG. In the case of a job, the first decision is to know if a VAC area is necessary or if only a Core Set will suffice. Using the priority word in the restart table entry, a positive priority value indicates that a VAC area is necessary. If the priority is complemented, a Core Set is obtained for the job but no VAC area. The entry point address is used in a similar manner. When complemented, it indicates that a waitlist task must be scheduled (not a job) and that the word used for defining the job’s priority should be used as the waitlist scheduling time. This encoding scheme, which may appear on the surface to be needlessly complex, actually saves several words in the unswitched fixed storage area.

**Phase tables and restart processing**

The number of options available with restart processing creates a dizzying array of data formats and parameter lists to process. However, the saving of storage space in the AGC is always a high priority, and the concise format of the PHASCHNG parameter lists reflects this. Each program protected by restart code has its own unique set of requirements, all of which must be passed to the PHASCHNG routine. At a minimum, the call to PHASCHNG must include the group and phase number, and flags to indicate important properties of the programs used for the restarts.

Two broad classes of parameters are used when calling PHASCHNG. Calls to PHASCHNG of “Type A” define a fixed, single-word parameter that references an explicit restart routine. “Type C” parameters include additional data words to describe variable data used by the restart routine. Such variable data might be the scheduling time for the defined recovery waitlist task, or the entry point address of the restart code. PHASCHNG also allows a third, “Type B”, format, which is a hybrid that combines one fixed and one variable parameter. Highly condensed, the format of the data word following the call to PHASCHNG contains essential data such as the parameter type and group number. For those parameter lists that support variable data, bits also define what additional parameters are present.

**The design of the phase tables**

Dynamically altering the contents of the restart table is not possible, as it resides in fixed storage. This is reasonable, as the data in the restart table must be protected against any occurrence of a program failure that alters memory. However, restart routines frequently require data that is specific to the recovery needed, and this is saved in the phase table. The restart phase table contains the parameters necessary for recovering each restart group. The parameter strings passed to PHASCHNG are saved in the table, together with their bitwise complements. This odd means of storing data is an essential part of the sanity checking of the system during the recovery sequence. Saving the inverse of the data parameter is an effective means to assure that the restart tables have not suffered from corruption by a program error. Phase tables, like so many other critical data areas, are particularly vulnerable because they reside in unswitched (common) erasable storage. Without any storage protection mechanism in the AGC, damage to the phase tables can occur at any time and without any warning. Storing a second copy of the phase table might be useful,
but there is always the chance of having several words overlaid by identical garbage. For example, if an errant process wrote zeros throughout memory, how would this be detected without extensive logic? In a more general case, how do we determine if two identical copies of data are invalid? While the technique of saving both the data and its complement is far from foolproof, it is trivial to implement and requires only a small amount of additional storage. Upon entering the restart processing, the phase table is checked to see if it is still valid. If a test of the phase table fails, erasable storage is presumed to be corrupted and program recovery stops. With a phase table that is longer valid, it is no longer possible to restart the program and so a hardware “fresh start” is initiated.

A check of the phase table’s health is driven by more than a conservative programming philosophy. When performing a restart, no effort is devoted to attempting to diagnose the root cause of the problem; rather, the restart process begins immediately. While this allows for a fast recovery, so important in a real-time system, it leaves open the question of how extensive the problem is. The most conservative assumption is that any routine could be at fault, so it is necessary to cancel all active jobs and tasks to ensure that the failing component is terminated. Logic errors are always likely suspects, and might generate a restart for any number of reasons. Of course, bad logic usually begets bad data, and it is highly likely that corrupted erasable memory resulted from the problem. Regularly examining several areas of erasable memory is part of the self-checking and restart processing. Unlike the phase table entries, testing of these data areas centers on checking for known values or ranges. If a discrepancy exists, it must be presumed that erasable storage is corrupted, and a fresh start is the only option to clear the error.

Figure 37: The + phase / –phase tables
Phase tables part 2: control of Major Modes

A secondary function of phase table management routines is maintaining the program number of the currently executing Major Mode. This quantity, saved in the MODREG erasable storage location, is displayed on the DSKY as the program number. Maintaining the Major Mode number is little more than a housekeeping task, as a change in MODREG does not result in scheduling a new program. On the other hand, a number of routines check this value, and alter their logic based on the program that is currently running. In a few critical mission phases, the Major Mode progresses to a point where it starts to run significantly different execution logic. This change is presented to the crew as a change in the Major Mode number, making it appear as if a new program has started. For example, the progression of Major Modes during a lunar landing (Program 63, then Program 64, and finally Program 65) is entirely automatic, saving the crew from the distraction of manually entering DSKY commands during a time of high workload. While the follow-on programs might appear as standalone Major Modes, their execution is dependent on the successful completion of the earlier stage.

A bad day at work: program alarms and restart processing

The AGC uses three different mechanisms to handle problems with the software or its inputs, recognizing the fact that not all situations require the same level of recovery. Program alarms, aborts and fresh starts are all possible options in the case of hardware or software problems, and each provides a different level of recoverability. Minor issues such as procedural errors may present themselves as program alarms, but do not require restart processing. In these cases, the program alarm is little more than an error message, such as an indication that the landing radar is not in the correct position, or that a program wishes to use the IMU while that hardware happens to be turned off. Other program alarms or aborts are a more serious matter. For a system resource shortage or a non-recoverable software error, the only recovery option is a restart of the AGC. Restarts from program aborts take on two different identities. A software restart terminates all running work and resets many of the system variables before performing recovery processing using the restart table. The objective of this type of restart is to preserve the execution environment and continue processing with a minimum of disturbance. It is important to realize that a software restart is not the same as “rebooting” a computer. Although work is thrown away, some restarts automatically reschedule programs for execution. More serious problems can result in cases where most of erasable memory is preserved but processing is halted in a so-called “P00DOO” abort. A far more drastic response to errors is the hardware restart, or fresh start. A fresh start clears all the work in the system, reinitializes the hardware and system variables, and places the system in an idle state. A fresh start is the equivalent of “rebooting” the AGC, with all work being lost and erasable memory reinitialized.

---

5 Exploiting this behavior is discussed as part of the Apollo 14 abort switch workaround.

6 Confusingly, the terms program alarm and program abort are often used interchangeably.
Each type of restart has a four-octal-digit code number associated with it that uniquely identifies the problem. A prefix digit to the code describes the type of recovery that is done: a 2 indicates that a P00DOO abort is called, and a 3 in the first digit signifies a software restart. Using this coding scheme, a 1202 alarm will be displayed as 31202 on the DSKY. Invoking a program alarm requires calling the ALARM executive routine with the alarm number. ALARM requests a priority display of the error code on the DSKY, overriding any existing display, and also illuminates the PROG[ram Alarm] light. The crew brings up Verb 05, Noun 09 to display the current alarm number, together with those for any other program alarms that might have preceded the restart. Finally, a number of important variables are set back to their initial state. The integrity of the restart phase table is verified, and restart processing for each active phase is performed. Restart groups are handled in a sequence, from group six backwards to group one. After the restart sequences are complete, the previously executing programs are running again and critical data such as IMU orientation and the state vector are preserved. All told, the entire restart process completes in seconds. After acknowledging the alarm by pressing the Reset key, the astronaut can reenter the request, terminate the program, or take other actions.

**Program abort: P00DOO**

Program aborts represent more significant problems than program alarms, and result in a P00DOO abort. For example, if a program encounters an unrecoverable logic error such as attempting to take the square root of a negative number, it immediately terminates without invoking the restart processing routines. Erasable storage is not cleared, thus preserving critical system data that is required even with the system idling. With the Major Mode no longer executing, the AGC enters an idling state, displayed as Program (Major Mode) 00. Program 00 reflects the state where no Major Mode is active, yet all other routines (attitude control, telemetry downlink, etc) are actively running. As in the case of program alarms, when the error is detected, the P00DOO Executive routine is called with the abort number. Again, the PROG light is illuminated, and Verb 05 Noun 09 displays the abort code to the astronaut. Program 00 appears as the Major Mode after the Reset key is pressed, and the computer is in a state ready to start a new program.

**Program abort: software restarts**

Both a fresh start (hardware restart) and a software restart (often confusingly called simply a “restart”) begin with similar actions to quiesce virtually all system activity. Software restarts focus on bringing previously running program back up, while a fresh start resets the AGC back to an initial idle state, ready to start new work. In both cases, all jobs are canceled and all tasks in the waitlist queue are deleted. Once the system is down, the hardware and software restart routines

---

7 The entry point label in the program abort code is BAILOUT; a phrase not lost on pilots.
continue, but with different objectives. For the recovery process to succeed, the Executive must perform several tasks without distraction, therefore early in the restart sequence all interrupts are inhibited. Timers and flags for the T3RUPT through T6RUPT interrupts are set to prevent their routines from executing until their environment is cleaned up. The three tables associated with units of work – the Core Sets, VAC areas, and the waitlist queue – are cleared and reinitialized. DSKY routines are reset to their initial state, and the display is cleared. Warning lights are turned off, with the exception of the PROG and IMU error lights if these are illuminated. Important system variables, possibly containing clues to the nature of the restart, are gathered and downlinked to the ground for analysis. A full dump of erasable storage would be both time consuming and unlikely to provide much additional diagnostic information.

Rebooting: the AGC fresh start
A fresh start, on the other hand, bypasses the program recovery process and continues shutting the system down to an idle state. Shutdown commands are sent to the SPS, DPS or APS engines if they are firing. Much of the hardware is reset, including the IMU. The loss of the inertial platform requires that the Digital Autopilot be placed into an idle state and all thruster activity terminated. Without an attitude reference computer control during thruster firings is impossible. Since the AGC is progressing to an idle state, no restarts to recover running programs are run, and the restart tables are reinitialized. Finally, with the hardware and software fully reset, Major Mode 00 (Program 00) is displayed on the DSKY.

THE ASTRONAUT INTERFACE: THE DISPLAY AND KEYBOARD

The language of the DSKY
For just a moment, return to the early 1960s and the early design phase of the AGC. Virtually all computing was batch, using punched cards and paper tapes for data storage. Hard disks, or disk drives as they were known, were slowly making their way into commercial computing. Magnetic drums, the precursor to hard disks, offered high speed (for its day) random access to data. All of these units were large, weighed hundreds of pounds and used prodigious amounts of electricity, making them completely impractical for spaceflight computing. The concept of engaging in an interactive dialog with a computer was just starting to emerge, yet it would still be years before systems with practical timesharing capabilities became commonplace. With such a background, consider the dilemmas faced early in the AGC design process. What would an interaction with a real-time system look like? What would an astronaut need to know during a particular mission phase? What kind of data would require to be entered by the crew? Most importantly, what would the “language” of such a dialog look like?

8 Fully manual control of the thrusters is always available.
A small thought experiment is revealing in how people communicate their needs outside the world of computing. Assume you are a tourist arriving in New York with a very limited English vocabulary. It is nearing lunchtime, and you are hungry for a quick meal. Looking for advice, you find a police officer and say one of the few words you know: “eat”. With a few gestures, you can convey the idea you are looking for a place for lunch. So far, so good. But, New York prides itself as a “city of restaurants”; the question now becomes, “what would you like to eat?” Knowing that New Yorkers seemingly thrive on a single dietary staple, the answer is obvious. “Pizza”. The police officer now has all the information he needs to help you find a pizzeria for lunch, and after a few more gestures you are on your way. From this simple exchange, the first step is to analyze the words “eat” and “pizza” on a grammatical level. “Eat” is a verb, describing an action or an operation on an object. “Pizza” in a noun, and is the subject of the verb. The verb-noun phrase, while grammatically awkward due to its brevity, conveys enough information to be understood. Returning now to the world of computing with this knowledge, this structure is recognizable as the basis for almost every interaction. Hardware operations such as ADD [accumulator with the contents of] LOCATION, or text oriented commands found in systems like Unix or DOS (EDIT FILENAME.TXT), both are composed of a verb followed by a noun. Even modern WIMP interfaces (Windows, Icons, Menus and Pointers), despite their graphical presentation, simply

![Figure 38: Display and keyboard](image-url)
change the way this notation is commanded. Under “File” in a typical pull-down menu, there is “Print” (a Verb), which displays a pop-up box for specifying the filename (the Noun). Within every level of computing, almost every action can be expressed in terms of the humble verb and noun.

Therefore, it should come as no surprise to learn that two primary keys on the DSKY are labeled “Verb” and “Noun”.

Using the language of Verbs and Nouns makes interaction with the AGC remarkably easy. A dialog with the computer begins with pressing the Verb key, which defines the operation to be performed, followed by a Noun code specifying the data that is to be operated upon. The primary operations performed through the DSKY are:

- Display, monitor or update data used by the AGC
- Execute and manage programs to control major mission phases
- Execute commands to control spacecraft hardware or to modify running programs.

With this background as introduction, it is time to begin the discussion of the Display and Keyboard, or DSKY. The DSKY is the visible component of the AGC, and is centrally located on the instrument panel for easy viewing by the crew. As the astronauts’ primary interface to the AGC, the DSKY provides both data input and output services, plus status indications on other guidance and navigation components. Approximately 7 inches (17.8 cm) by 8.5 inches (21.6 cm) tall, the DSKY includes a 19-key keyboard, a display section for Verbs, Nouns and program numbers, three data registers, and numerous warning lights. Digits are composed of the usual seven-segment arrays, but are green electroluminescent displays, unlike the LEDs or liquid crystal displays so common today. The DSKY is assembled into a single unit, and is physically separate from the computer. A single DSKY is in the Lunar Module, but two DSKYs are in the Command Module. In the CM, the first is located on the main display console, midway between the leftmost and center positions, with a second at the optics station in the lower equipment bay at the foot of the crew couches. To illustrate the overwhelming importance of the computer to the mission, consider the locations of these units in the spacecraft. All vehicles, regardless whether they are a car, an airplane or a spacecraft, place the most important information directly in front of the driver or pilot. In the CM, the DSKY is placed squarely in the astronauts’ field of view, and is one of the four primary flight instruments along with two FDAlS (attitude indicators) and the Entry Monitor System. In the Lunar Module, the DSKY is located between the Commander and the LMP at approximately waist height. In both spacecraft, ease of monitoring and inputting commands drove the DSKY’s installation location.

Displays for the program number, also known as the Major Mode, and the Verb and Noun displays occupy the upper right-hand section of the DSKY. An electroluminescent light labeled CMPTR ACTY is to the left of the program number and illuminates when the computer is actively processing data, and is extinguished when the DUMMY task is running. Below the Verb and Noun displays are three, five-digit data registers used for displaying data specified by the Noun.
Preceding the digits is a sign that is used only with decimal data to indicate whether the quantity is positive or negative. Nouns operate on a wide variety of data formats, from integers to octal digits, to real numbers with varying degrees of fractional display. A surprising characteristic of the display registers is the lack of an explicit decimal point. Editing numbers to determine where the decimal point falls would require computing resources, extra keystrokes by the crew and computer time, but add only a minor amount of additional readability. Octal data, of course, does not use a decimal point in its notation. In some cases, more than three data items need to be displayed by a Noun. For example, if the time before an event is displayed in minutes and seconds, both of these two-digit quantities can fit in a single register. To ensure that the data is readable, minutes will occupy the first two digits of the five-digit display, with seconds in the last two digits and the middle digit blank.

Three data registers present an excellent compromise between engineering realities and human factors, as many of the important concepts in flight are expressed using three values. Attitude uses three quantities for pitch, roll and yaw, and both position and velocity vectors are defined by three values. Time is perhaps the most familiar case, using hours, minutes and seconds to express itself accurately. During the times of highest cockpit workload, the crew will be constantly scanning their instruments, looking for any indications of trouble or reassurances that the systems are performing nominally. To efficiently scan a large number of displays, and
integrate that information into a coherent picture, is perhaps the most important and challenging task for a pilot. Three pieces of information, such as the DSKY register display, is the most a person can comfortably absorb and process after a quick glance. Overloading the pilot with data that might be interesting, but not essential to the task at hand, quickly becomes counterproductive.

Keys on the DSKY reflect the numeric language used in the dialog between the crew and computer. Programs, Verbs and Nouns do not use “names” in the conventional sense; they are identified by a numeric code. Of the 19 keys on the DSKY, ten are for the numeric digits, plus two for the + and – signs. In addition to the Verb and Noun keys, five keys are used to interact with the software:

ENTR [Enter]: Enter provides two different functions to the crew. First, Enter indicates that the Verb, Noun or register entry is complete and ready to be used by the software. In some procedures, the computer will request that the astronaut perform a task by displaying Verb number 50, known as a “Please Perform” request. Pressing Enter acknowledges that the requested operation is completed.

PRO [Proceed]: The Proceed key is most frequently used to indicate that the program can continue using its previously calculated data. Many programs, especially those performing complex maneuvers (engine burns, rendezvous, etc) display a large number of parameters that the crew can either accept or change. In these dialogs, the computer flashes a display with the Noun code for data that is specific to that phase of the maneuver. If these numbers are acceptable, the crewmember commonly presses PRO, telling the software the values are acceptable. If they are not, they are changeable using the data entry Verbs. Proceed is also entered as a response to a flashing Verb 99, “Please enable engine ignition”. By pressing the PRO key, the astronaut gives the AGC its final permission to fire the engine automatically. In conjunction with Program 06 (AGC Powerdown), pressing and holding PRO places the computer in Standby mode, where it consumes significantly less power. Bringing the AGC out of Standby mode also requires pressing the PRO key.

CLR [Clear]: Operating just as it implies, CLR allows the crewmember to clear out an entry on the display. When keying in a Noun or Verb code, or data into one of the registers, pressing CLR will blank that part of the display to allow data to be reentered from the beginning.

RSET [Reset]: A program alarm has a wide range of effects on AGC operation, ranging from a comparatively benign error message to a complete system fresh start. When these alarms occur, the PROG (Program Alarm) light illuminates and the DSKY blanks. Pressing RSET acknowledges the alarm, allowing the crew to display

---

9 An alphanumeric keyboard is totally impractical for a spacecraft flight computer, where rapid and concise data entry is paramount. One could easily visualize the near-comic problem of an astronaut, faced with a traditional keyboard, trying to “hunt and peck” commands while wearing bulky gloves.
the error code and continue if possible. If an error made while entering data generates an OPR ERR light, the RSET button clears the condition and allows the crewmember to reenter the key sequence.

KEY REL [Key Release]: Although the AGC is a fully multiprogrammed system, the DSKY can present information from only one program at a time. There are often occasions where an astronaut is using the DSKY to work with one program, and another routine is demanding attention with an important display request. That program will announce its need for service by illuminating the KEY REL light on the DSKY, and wait for the crewmember to press the KEY REL key. Once KEY REL is pressed, the KEY REL light extinguishes and the waiting program’s data replaces the current DSKY contents. Note that there is not a requirement for the crew to immediately stop work to respond to the KEY REL light. The astronaut can finish his work on the DSKY, and then bring up the waiting program’s display. Interrupting work on the current program in order to attend to another is a confusing and unworkable approach, as “toggling” the DSKY between two programs is not possible. Once the waiting program takes control of the DSKY, the astronaut must complete the dialog before returning to the other program.

Several status, warning and error lights complete the DSKY display area, and they are controlled directly by the software, not by external sensors. Not all lights are common to the Command Module and Lunar Module computers. For example, the landing radar ALT[itude] and VEL[ocity] lights are found only in the LM. A full list of the lights follows:

UPLINK ACTY [Uplink Activity]: Ground controllers frequently perform remote updates of the AGC for tasks as mundane as updating the system clock, to providing the latest guidance and navigation data. Since the crew cannot use the DSKY during the uplink, illumination of the UPLINK ACTY light advises the crew that the ground has control of the computer. Once the light has extinguished, the crew is free to resume operating the DSKY. UPLINK ACTY has two other, more obscure functions in the Command Module. During rendezvous navigation, UPLINK ACTY illuminates when a large (> 10 degrees) attitude change is necessary for tracking, yet the AGC has not requested that the crew maneuver to a new attitude. Also, in the minutes before Translunar Injection, the Launch Vehicle Digital Computer (LVDC) in the Instrument Unit of the Saturn V enters a mode called Time Base 6, which provides the guidance and control routines for the maneuver. When the LVDC begins Time Base 6 the CM’s AGC is notified, and it illuminates the UPLINK ACTY light for ten seconds to notify the crew that the Translunar Injection sequence has started.

TEMP [IMU Temperature]: TEMP indicates that the Inertial Measurement Unit temperature is out of limits. Maintaining its temperature within a narrow range is important for the precision of the IMU, and accurate regulation requires elaborate heating and cooling equipment. Ambient air at a pressure of 1 atmosphere is circulated throughout the sealed spherical case, and a water-glycol loop extracts
waste heat. The TEMP warning light illuminates any time the internal temperature is outside the 126° F to 134° F (52.2° C to 56.7° C) operational range. Failure of the IMU is assumed if the temperature is not within these limits, and the AGC will “cage”, or reset, the IMU to prevent further damage; an intervention that results in the loss of all attitude reference information.

NO ATT [No Attitude]: For the AGC to perform any attitude control maneuver, the IMU must be powered up, within temperature limits, and aligned to a known orientation. Additionally, its interface electronics, the Coupling Data Units (CDUs) must be functioning within normal limits. Failure to meet any of these criteria results in the assumption that the IMU is unable to provide valid data, and no attitude reference is possible. This does not mean that the IMU has suffered a mechanical or electronics failure. NO ATT will illuminate when the IMU begins its power up sequence, and whenever it loses its orientation for any reason (such as gimbal lock). Here, the expectation is that the IMU is physically in good condition, but needs manual intervention before it can be used.

GIMBAL LOCK [Gimbal Lock]: Gimbal lock occurs if the middle of the three gimbals exceeds a critical angle, rendering it unable to rotate freely and maintain its orientation. This is the gimbal on the Z, or yaw, axis. Once entered, a gimbal lock situation results in the IMU losing its orientation, so realignment is necessary for recovery. Crew awareness of the IMU orientation is a key element of gimbal lock prevention, so the GIMBAL LOCK light will illuminate when the middle gimbal angle reaches ±70 degrees, still 15 degrees from actually locking up. The light provides a warning that gives the crew time to prevent further yaw motion towards gimbal lock. Only when the middle gimbal reaches ±85 degrees will the IMU actually enter a locked state. At this point, all orientation is lost and the NO ATT light will illuminate.

STBY [Standby]: To conserve power, a standby mode is available in the AGC. This reduces electrical consumption to 15 watts from the 70 watts used during normal operation. To enter Standby mode, the crew selects Program 06 and presses the PRO key until the STBY light goes on. If desired, the AGC and IMU can be powered down at this point.

PROG [Program Alarm]: Program alarms are generated by software errors or hardware failures, but are not necessarily fatal and often do not require extensive recovery. The crew normally presses the RSET key to acknowledge the alarm, and use Verb 05 Noun 09 to display the alarm code. More serious program alarms result in a variety of automatic restart options, which will illuminate the RESTART light.

KEY REL [Keyboard Release]: An issue posed by multiprogrammed systems is how to alert the crew that another program which is running in the background needs attention. When one program is using the DSKY, another program needing service will light the KEY REL lamp, requesting the astronaut to complete his work and release the DSKY for the waiting program to use. There is no information on the name of the routine requesting the DSKY, but the crew’s knowledge of the currently running programs usually allows them to make a reasonable assumption of which
program or routine is making the request. When the crewmember is ready to work with the waiting program, pressing the KEY REL key brings up the waiting display.

RESTART [Restart or Fresh Start]: Some errors in the AGC are serious enough to warrant restarting the failing program or perhaps even the entire computer. The type of error determines the extent of the restart, but terminating programs is usually required. As part of the restart and recovery sequence, the RESTART light will illuminate, notifying the crew that a problem exists and the computer will attempt to recover. As restarts often follow from program alarms, the crew can press the RSET key to clear the PROG and RESTART lights, and then use Verb 05 Noun 09 to determine the cause of the restart.

OPR ERR [Operator Error]: Although the “language” of the DSKY is not complex, strict rules apply to the entry of requests and data into the DSKY. Not all combinations of Nouns and Verbs are valid, and data entered must match the format specified by the Noun. Any input that is not valid will generate an OPR ERR light after pressing the Enter key. Operator errors are usually trivial problems, and incorrectly keying data is often the cause. Clearing the error requires pressing the RSET key and reentering the Verb or Noun sequence.

TRACKER [Tracking Error]: Originally used to indicate an error in the star tracker, this warning light kept its name after the star tracker hardware was deleted from the Apollo specifications. In the Command Module, errors in the optics assembly CDU electronics or problems in reading valid VHF ranging data cause the TRACKER light to illuminate. In the Lunar Module, it is illuminated if the rendezvous radar CDU electronics fail, or if landing radar data is not reasonable. In this last case, the TRACKER light may be accompanied by the ALT and VEL lights.

DAP NOT IN CONTROL [Digital Autopilot not controlling the LM]: Found only in the Lunar Module, this unlabeled light illuminates any time the Digital Autopilot is not controlling the spacecraft. Such a situation occurs when the DAP is off, or in idle or minimum impulse mode.

PRIORITY DISPLAY [Priority Display waiting to use the DSKY]: Found only in the Lunar Module, this unlabeled light is controlled by Program 20 (P20, Rendezvous Navigation). It indicates that important information is ready, and the crew should not delay in attending to it. P20 runs in the background during rendezvous maneuvers while other targeting programs run as the Major Mode.

ALT and VEL [Landing radar Altitude and Velocity lights]. During the Lunar Module’s powered descent, the landing radar continually measures the vehicle’s altitude above the surface as well as velocity components in all three axes. For the landing radar to report valid data, a number of operational constraints must be satisfied, such as staying within the acceptable attitude range and the data being valid for a minimum amount of time. Violating any of these constraints generates a signal to the AGC that indicates the data from the landing radar is not valid, and this in turn will cause the appropriate light to illuminate. Loss of radar altitude or velocity data can be a serious situation, especially during the later phases of the
The astronaut interface: the display and keyboard

landing, as the IMU cannot provide altitude data that is sufficiently accurate for a safe landing.

**DSKY operation**
A dialog with the AGC usually takes the form of a Verb, followed by a Noun. Both Verbs and Nouns are identified by unique two-digit codes. After the Verb and Noun are loaded into their registers, pressing the Enter key indicates that the entry is acceptable and ready for processing. One example of a DSKY request might be:

```
Verb 06, Noun 33, Enter
```

This sequence is somewhat tedious to write, and a shorthand notation is used extensively throughout the documentation and flight procedures. In this notation the Verb and Noun sequence above is abbreviated to V06N33E.

When the Verb or Noun keys are pressed, the entry in that display flashes as the digits of the code are entered, and continues to flash until the Enter key is pressed. Overwriting a previously accepted entry is possible by pressing the Noun or Verb key again and reentering the digits. If a mistake is made when entering data into a data register, the CLR key blanks the register and the data can be rekeyed. There are a few exceptions to this syntax. Extended Verbs, which do not require a Noun, use only the Verb code followed by the Enter key. Verbs that recycle or terminate a program also do not require a Noun, and only require pressing Enter to start the processing. Some Verbs cannot be keyed into the DSKY, as they are requests made by the software for the crewmember to perform some action. For example, Verb 50 (Please Perform a Checklist Item) might tell the astronaut to perform an attitude maneuver, align the inertial platform, or reposition the landing radar. Verb 99 is displayed in the last five seconds before engine ignition, and requests the final authorization to continue and fire the engine. If either of these Verbs is keyed into the DSKY, the OPR ERR light will illuminate.

The notation of Verb ##, Noun ##, Enter is a convention, but not a hard requirement. A sequence of Noun ##, Verb ##, Enter is equally valid. A displayed Verb is also “reusable” by keying in another Noun value, and bypassing the need to punch in the Verb code again. For example, entering Verb 16, Noun 95, Enter on the Command Module DSKY displays parameters related to an engine burn. The monitor Verb 16 is chosen because it displays the Noun as it changes, counting down the time until engine ignition. While this display is active, the astronaut can also monitor the AGC’s clock time by requesting Noun 36, without having to reenter Verb 16.

**Regular and extended verbs**
Two major types of Verb exist. “Regular” Verbs (numbered 00 through 37) are used to display, monitor and update data. Included in this range are Verbs that control overall AGC operation, such as starting, terminating and recycling programs, or restarting the entire system. “Extended” Verbs (numbered 40 through 99) run programs that perform a single discrete operation, such as setting flags, configuring panel displays, or altering the logic of the Major Mode. Most
extended Verbs run without requiring additional input, and in those cases where specific data is needed, it presents several Verb-Noun combinations to the astronaut. A typical example is initializing the Digital Autopilot through Verb 48 (Request DAP Data Load Routine). Several pieces of data are necessary before the DAP is started, including spacecraft configuration and vehicle weights. After several Verb-Noun dialogs between the initialization software and the crew, Verb 48 begins DAP operation. Extended Verbs monopolize the DSKY until their operation is complete, and a second extended Verb cannot be entered until the first has completed.

**DSKY operation – entering and retrieving data**

Verbs for viewing or manipulating data are broken down by data type (decimal or octal), and whether the data is loaded, monitored in real-time or simply displayed. Monitoring Verbs allow the crew to observe data such as velocity or altitude changing over time, updating at a rate of once per second. Before choosing the Verb to display data on the DSKY, the first task is to consider the Noun used with the Verb. This defines the type of data and the number of DSKY registers that are required. Most of the data is in a decimal format, such as times, velocities or angles. Computer-specific data, such as memory addresses and their contents, or data expressed as bit patterns use octal digits. Nouns reference a wide variety of data, and require one, two or all three data registers in order to display their contents, and each Noun component is assigned a register where it is displayed. For example, the hours, minutes and seconds components of Noun 33 to display the time are assigned to registers 1, 2 and 3, respectively. These assignments are fixed by the Noun format and cannot be rearranged into a different sequence.

Seven display Verbs are available, five of which are variations on displaying octal data in different registers. Arguably, the most frequently used display Verb is Verb 06, which presents decimal data in any or all of the three registers. Verb 06 understands the register requirements of the Noun it is operating with, and adapts the display accordingly. Regardless of the number of components that a Noun requires, Verb 06 will display only those that are requested. In the case of Noun 92 (Command Module Optics Shaft and Trunnion Angles), Verb 06 knows that only registers 1 and 2 are needed for the display, so register 3 will remain blank.

Monitoring data changing in real-time, such as altitudes, velocities or angles, gives the crew essential situational awareness of the state and performance of their spacecraft. While it is possible to repeatedly call up fresh data by rekeying the display request, this option is impractical during a high-workload phase of the flight. Perhaps the most visible need for monitoring real-time data is during the lunar landing. The crew references the DSKY constantly, and also cross-checks the computer’s values against a cue card with pre-computed values for each point of the descent. As the descent progresses, the crew can verify their progress as they reach important milestones, and take corrective action if a problem emerges.

The list of monitoring Verbs follows a similar pattern to those used for displaying data. Most are for monitoring octal data in various registers, but Verb 16, the real workhorse, continuously displays decimal data. Updating can be suspended by
pressing the PRO key or Verb 33 (Proceed Without Data). Pressing the Enter key instructs the AGC to resume continuously updating the display.

The five data load Verbs are organized by the registers they update. Each accepts the Noun’s components, either singularly or in combination. Verbs 21, 22 and 23 load one Noun component into data register 1, 2 or 3 respectively. As entering a Verb and Noun code each time a component is loaded into a data register is tedious at best, multiple component loads are available by Verb 24 and Verb 25. Verb 24 loads components into registers 1 and 2. It switches automatically to Verb 21 when loading register 1, then to Verb 22 when loading register 2. Verb 25 operates similarly, using Verbs 21, 22 and 23 for loading all three registers. Decimal data is identified by a + or – sign preceding the digits as they are entered. When entering data, all leading zeros must be entered as well. Failing to enter all five digits in a register will illuminate the OPR ERR warning light, and the entry will have to be rekeyed.

**DSKY error checking**

A few basic rules for DSKY operation are necessary to prevent entering invalid data. First, leading zeros are necessary for all entries, with two digits required for Verb and Nouns and five digits for registers. For any given Noun, all components will use either decimal or octal notation; mixing the two types in one Noun is not allowed. The explicit datatype of each Noun must always be used; entering an octal for a decimal quantity is not allowed.

Input punched into the DSKY requires a ‘sanity check’ before it is passed along to the program. With so many keystrokes used during a flight, it is inevitable that a miskeying will occur. Only basic checks are performed on the input Nouns, and little effort is taken to assure that the data is appropriate for the Noun. Mistyping an angle 086.00 as 008.60 will not be recognized as an error, as both are valid numbers. An invalid entry will cause the OPR ERR light to illuminate, but there is no other indication of the specific problem. The crewmember must review the entry, identify the problem and reenter the request. The following situations are not legal, and will illuminate the OPR ERR warning light:

- Entering the digits 8 or 9 into a register without a preceding + or – sign. Not providing a sign indicates octal data, which only uses the digits 0 through 7.
- The software does not implement the Verb or Noun code entered.

---

10 A note about the use of the rather archaic phrase “punched”, as in “punching in the data”. Throughout the early decades of computing, the dominant form of input was the punched card. Data was represented using holes punched into the card, which were read by mechanical or optical scanners. Few artifacts of computing have endured as long as the lowly but ubiquitous punched card. Although they have long disappeared from the world of data processing, the nostalgic and curious can still get a glimpse of what they were like. Airline flight coupons of the type printed at the check-in counters use the same card stock, and are virtually the same size as the venerable punch card.
• Providing decimal data when entering a machine address. Only octal data is allowed for addresses.
• A value out of limits for the component, such as specifying more than 360 degrees for an angle.
• Mixing decimal and octal data within a Noun.
• Attempting to load data into a Noun that is not loadable, such as a checklist code for the crew to act upon.
• Entering more than two digits for a Verb or Noun code, or more than five digits into a data register.
• Pressing the Enter key without a Verb and Noun already loaded (except for extended Verbs, which do not need a Noun).
• If a Priority Display is waiting, and the “mark” button is pressed on the optics assembly.

**DSKY input processing**
A group of routines under the formal name, “Pinball Game – Buttons and Lights” (more informally simply called “Pinball”) handle entering data, interpreting the Verb and Noun codes and dispatching commands. Such a whimsical name is actually quite appropriate, as Pinball dedicates much of its code to data punched into the keyboard and managing the lights and digit displays. Although the movement of data from the output buffer to the DSKY display is driven by the T4RUPT interrupt processing, all the display contents are managed by Pinball routines. Input through the DSKY begins when an astronaut presses a key on the panel, raising the KEYRUPT1 keyboard interrupt. The keycode, identifying the key just pressed, is placed in the lower five bits of input channel 13. The interrupt routine runs quickly to schedule the Pinball routine CHARIN to process the data. CHARIN categorizes the keystroke as a Verb or Noun, a digit, or any of the other operation keys, and determines what processing must take place. Pressing the Verb key, for example, sets flags indicating that the subsequent characters must be interpreted as a Verb code, and sets bits in the display output buffer, DSPTAB, to flash the Verb and Noun fields. Essential sanity checking on the input is performed first. Invalid keystrokes, such as pressing a non-digit key when numeric data is expected, will terminate CHARIN and illuminate the OPR ERR light. As numeric data is punched into the DSKY, it is first displayed in the leftmost digit of the Verb, Noun or register, and each new digit appears successively to the right. This is different from the familiar case where each keystroke shifts the existing display left, with the new digit inserted in the rightmost position. Only after pressing the Enter key does the interpretation of the data begin.

Extended Verbs (numbered 40 through 99) are quickly dispatched by transferring control to the appropriate routine. A new job is not scheduled for performing extended Verbs, rather the Verb is processed as part of the CHARIN routine. Often an extended Verb requests additional data, as in the case of Verb 48 (Load Digital Autopilot Parameters), which presents several Verb-Noun combinations to the astronaut, requesting information on the spacecraft configuration and desired autopilot characteristics. After collecting the necessary data, Verb 48 schedules the Digital Autopilot tasks and terminates itself. Other extended Verbs can proceed
without additional data, and set flags or schedule other work before ending. As extended Verbs execute as part of the CHARIN routine, only one extended Verb can execute at a time. In the event that a crewmember tries to execute a second Extended Verb before the first one has finished, the OPR ERR light illuminates and prevents a second extended Verb from executing concurrently.

Regular Verbs (when combined with Nouns) display, monitor or load data in a wide variety of formats. Regardless of the operation requested, an elaborate set of tables and routines convert the data entered into a format usable by the software routines. As seen in the hardware description section, most data is stored using a fractional notation. Additionally, the units employed internally in the AGC are exclusively metric, yet human factors demand that the crew uses English units. These requirements, plus the different numbers of components for the Nouns, result in a complex design for referencing the appropriate scaling and conversion routines.

As an example of data loading, we will use the case of the three components of Command Module Noun 89, which specifies the location of a lunar surface landmark. Noun 89 is a mixed Noun, whose data components use the following format:

<table>
<thead>
<tr>
<th>Landmark Latitude:</th>
<th>XX.XXX Degrees</th>
</tr>
</thead>
<tbody>
<tr>
<td>Landmark Longitude/2</td>
<td>XX.XXX Degrees</td>
</tr>
<tr>
<td>Landmark Altitude</td>
<td>XXX.XX Nautical miles</td>
</tr>
</tbody>
</table>

Each component of the Noun uses a different input format and scaling. Here, latitude is saved as a double precision value, as is the Longitude/2. Longitude normally is expressed as a range 0 to ±180 degrees, but, as its name implies, Longitude/2 is scaled so that the data entered is less than ±90 degrees. Landmark altitude, expressed as the distance from the center of the Moon, is also double precision, but scaled quite differently than the other two components. Parsing each Noun and its individual components requires a complex assemblage of tables and routines. The two types of Nouns, ordinary and mixed, require that those tables alternate between two different formats, further increasing complexity. Ordinary Nouns are numbered 01 to 40, use direct memory references to the data, and the values are all of the same data type (angles, velocity or octal). Mixed Nouns, those which are numbered above 40, are much more complex. Each component can be a different datatype from the others, and referencing the data for the display requires an extra step called indirection. Ordinary Nouns are defined by referencing data located in contiguous storage, and use the same scaling and conversion resources for each component. They require only two data structures to reference the data and routines to manipulate the DSKY data. Mixed Nouns present a completely different situation, as their numerous variants require no fewer than eight tables for processing the data.

Many Nouns and extended Verbs are unique to the Command Module or Lunar Module. Because there is so little commonality, it is important to qualify which spacecraft the code is for.
**DSKY output**

Getting data out of the AGC and onto the DSKY is, in many ways, more difficult than getting the keystrokes into the system. Consider the problem of managing the data flow to the display. It has 21 digits and three +/- signs (one pair for each register), plus several status and warning lights, each of which is individually addressable. Tiny mechanical relays act as switches to turn each of the segments of the display on and off. Implementing separate output channels makes programming relatively easy, but also requires an exorbitant amount of hardware and cabling. Using multiple channels for sending data to an I/O device has the huge appeal of speed, but this is unnecessary for a device like the DSKY. Humans can process data at only a fraction of the speed at which a computer can generate it, so it is best to forgo fast software and expensive hardware in favor of more appropriate technology. This exchange of complexity and cost is possible through multiplexing the data over one output channel between the computer and the DSKY, allowing the single channel to carry data intended for different destinations. For this technique to function, some overhead is necessary in the form of additional bits appended to the output data, which are used to identify the targeted DSKY field. Each digit, sign and light in the DSKY is associated with a unique identifier, which is loosely analogous to a memory address. Data intended for DSKY output is buffered in an 11-word table in erasable storage named DSPTAB. Two 5-bit fields, each corresponding to a decimal digit, reside in bits 10 through 1 in the word. Each such field contains a relay code used to display numeric data, rather than the actual number itself, simplifying the task of selecting and driving the relays that display the digit segments. Bit 11 in the DSPTAB word is employed for other purposes, such as the + and – sign in the data registers, and the indicator to flash the displays as data is entered. Bits 15 through 12 have a dual role. DSPTAB entries in memory use bit 15 (the sign bit) to indicate whether an update to the display is required. A positive value indicates that the data reflects the DSKY display, and no updates are necessary. When bit 15 is set to one, the word is interpreted as negative and indicates that the DSKY requires an update from the data buffer. When a data word is ready for output, a display addressing code replaces bits 15 through 12.

The display itself is an electromechanical device, using over 100 miniature latching relays to create the number segments on the panel. Using latching relays in the DSKY display creates a valuable dual benefit. First, traditional relays are susceptible to vibration and acceleration forces, where unrestrained relay contacts may open and close unexpectedly and produce meaningless displays. Additionally, mechanically latching the contacts open or closed eliminates the need to maintain power to the hardware, thus reducing the electrical demands of the device. Latching the relay open or closed does not happen instantaneously, however. Power must be applied for at least 20 milliseconds to assure that the relay has settled into its new state and can permanently maintain its configuration.

Sending data out to the DSKY occurs during the 120-millisecond T4RUPT processing cycle. DSPTAB is scanned for any negative entries, which indicate that new data is ready to be sent to the display. Recalling that the DSKY output channel
(channel 10s) multiplexes its data, a destination address code is necessary to define which display elements are being updated. As with all channel output operations, the data word is copied from DSPTAB to the accumulator. After each data word has been loaded, bits 15 through 12 are replaced with an identifying code for the corresponding display digits. Decoders in the DSKY hardware interpret this code and route the update to the appropriate set of relays, which in turn drive one or more display elements. After each write to the output channel, the routine waits 20 milliseconds for the relays to latch into place before continuing with the next data word. During the execution of a monitoring Verb, where all 15 digits in a data register might change, an update of the entire display can take as long as 160 milliseconds. Updating a large number of digits in the display becomes a relatively slow, mechanical process, and the display perceptibly “ripples” as each new digit is placed in the display.

**Noun tables**

Managing the various Noun formats requires a large number of tables to define each type. Eight tables in fixed storage describe Noun characteristics such as the number of components in each Noun, the datatypes used for each Noun component, and information on the proper internal scaling of the data. Ordinary Nouns are the most straightforward, as their data resides in contiguous memory and all components use a common datatype. These factors combine to require a small subset of the Noun tables. The large number of mixed Noun variants is responsible for the bulk of the complexity seen in the table structures. Combining both ordinary and mixed Nouns in the three main Noun tables reduces the overall complexity somewhat, but at the price of maintaining two different data layouts within each table.

The first group of Noun tables includes NNADTAB, NNTYPTAB and RUTMXTAB, which are respectively, the Noun addresses, type and mixed Noun tables. Entries in these tables have the important property of being referenced using the Noun number as an offset into the table. The remaining tables, IDADDTAB, SFOUTAB, SFOUTABR, SFINTAB and SFINTABR are addressed using indirect addresses located in one of the Noun tables. To completely describe how the Noun tables are used for DSKY input and output, a short introduction to each of the structures is necessary.

**NNADTAB:** As the first of the three main Noun tables, NNADTAB supplies references to the data referenced in each Noun component. The first 39 words of NNADTAB point to the starting address of the three ordinary Noun components. The remainder of the table uses a different format for mixed Nouns and contains two pieces of data: a component code, which defines the number of Noun components and some of their characteristics, and a pointer into another table, IDADDTAB, which in turn points to the individual mixed Noun components in storage.

**NNTYPTAB:** The second of the main Noun tables, NNTYPTAB also uses different formats based on the type of Noun. NNTYPTAB complements NNADTAB by providing information on the Noun components’ data representation. Entries for
ordinary Nouns include the component code data, an index into tables used to perform data scaling operations, and datatype information. Like NNADTAB, the information is different for mixed Nouns, and only datatype information is saved for these entries.

RUTMXTAB: The last of the main Noun tables, RUTMXTAB is used by mixed Nouns only. An entry in this table is organized into three fields, corresponding to a Noun component, each of which contains an index into the data scaling tables. Four tables are indexed by the index field, two each for DSKY input and output scaling factors.

SCALING TABLES: All Noun components entered or displayed on the DSKY have a scaling factor associated with them. Two pairs of tables are used for the scaling process, and are dedicated to either input or output. The first pair of tables (SFINTABR and SFOUTABR) contain the scaling routine addresses used for each data type. A second pair of tables (SFINTAB and SFOUTAB) contain scaling constants that are applied to the data during the conversion process. Indexes into these scaling factor routine and constant tables are located in NNTYPTAB or RUTMXTAB.

IDADDTAB: Unlike ordinary Nouns which reference data in contiguous storage locations, data for mixed Nouns can be anywhere in erasable storage, eliminating the possibility of using only a single address to point to the data (as is the case with ordinary Nouns). Additionally, mixed Nouns can have one, two or three components, in contrast to the standard three components of ordinary Nouns. Each entry in the IDADDTAB table contains the address of the variables referenced by a mixed Noun. The variable number of entries in each mixed Noun precludes using an index calculated from the Noun’s number, as is the case with NNADTAB and NNTYPTAB. Instead, an offset into IDADDTAB is part of each Noun’s entry in the NNADTAB.

**DSKY specific scaling**

In the earlier discussion of data formats, the concept of fractional notation for data was introduced. Here, maintaining the magnitude, or scaling factor, was a burden imposed upon the programmer. Consistency in applying data scaling simplifies calculations, but does not always result in the most intuitive representation of the data. For example, the value of pi is 3.1415926..., but in the AGC it might be implemented as 0.31415926 using a scaling factor of 10⁻¹. At the same time, the altitude of the spacecraft in lunar orbit might be 1,800 km above the center of the Moon (approximately 100 km above the surface). A scaling factor of 10⁻⁴ is necessary to convert this to its fractional value of 0.1800. While this value is completely acceptable, calculations become complex when including pi, scaled at 10⁻⁹, and other variables with yet other scaling factors. Although keeping track of the mix of scaling factors is not particularly difficult, extraordinary care is necessary to prevent overflows or other computational errors. Managing scaling factors is especially important for input and output, and
The astronaut interface: the display and keyboard

Figure 40: Noun tables
is an integral part of DSKY processing. Each datatype (angle, velocity or time) is assigned a standardized scaling factor and precision, and these must be used consistently for all computations.

TELEMETRY UPLINK

Not all data in the AGC originates from internal calculations or crew inputs. Several times during a mission, flight controllers enter data directly into the computer using the special uplink frequency. Adjustments to the mission clock, landmark locations and system variables occur regularly, but the most frequent and important update is to the state vector, the variable that defines the position and velocity of the vehicle in space. Three primary tracking stations, equally spaced around the world, monitor the position, distance and velocity of the vehicle, and relay the data to Houston where controllers perform the state vector calculations. At the same time, the crew takes sextant navigation fixes to calculate their own version of the state vector. Although the onboard solution is often as accurate as the ground-based calculations, conservative mission rules dictate that the ground controllers uplink their solution to the computer.

The state vector is not a trivial piece of data, and consists of six double-precision values. As an argument for the need to automate computer inputs, consider that updating this data by hand requires nearly 100 flawless keystrokes on the DSKY. As the numbers are encoded in fractional notation, it is impractical for the crew to decide whether the data is reasonable by simply looking at it. For critical information like the state vector, speed and accuracy requirements demand removing the human from the update process. Sending the update directly from ground computers neatly bypasses the most error-prone element in the process, and even lengthy updates require only seconds to perform. This process does not occur automatically, and two safeguards exist to prevent undesired or spurious updates. First, the AGC must be in an idle state, running Program 00, before an update may begin. Once an update begins, the Major Mode automatically switches to Program 27 which automatically terminates any previously running program. Next, the astronauts must flip the telemetry uplink switch from Block to Accept to physically allow the updates to enter the computer. Without these two conditions in place, no AGC updates are possible. In a stroke of brilliant simplicity, the format of the uplink data uses DSKY key sequences to enter and manipulate data. An uplink consists of the same key codes used by the physical DSKY keyboards, and uses the conventional Verb-Noun format for updating memory.

During the mission, tracking stations maintain nearly continuous transmissions to both the Command Module and Lunar Module. Both communication links transmit in the S-Band radio frequencies, with 2106.40625 MHz used for the CM and 2101.8 MHz for the LM. A 70 KHz subcarrier channel contains the digital data, which the LM’s Digital Uplink Assembly, or the CM’s Updata Link electronics extract and pass to the AGC. Assembling the individual bits into a 15-bit word from this serial
data uses two nonprogrammed instructions in the AGC. The uplink data is placed in
a special register named INLINK, which is located at location 458 in storage.

At the beginning of the uplink transmission, INLINK is all zeros, with the
exception of bit one, which is one. Uplinking a zero bit to the AGC triggers a
SHINC instruction, which shifts the uplink register left by one bit and inserts a zero
in bit 1 of the word. SHANC, like the SHINC instruction, shifts the uplink register
one bit to the left, but places a one in the rightmost bit. Successively shifted left by
SHINC and SHANC as new bits are received, the original bit 1 in INLINK
eventually arrives in position 16, triggering the UPRUPT interrupt. Processing of the
current work is suspended, and control transfers to the DSKY’s Pinball routine to
process this character.

Bad data (data not received correctly) is unacceptable, as a bad IMU reference or
state vector has serious consequences for guidance and navigation. Transmitted data
is often noisy and error prone, and a sufficiently long run of static could easily be
interpreted as valid data. Two tests for data validity are performed to assure the
correctness of the uplinked data. First, each 5-bit keystroke character is transmitted
three times as part of the single 15-bit uplink word. Comparing all three copies
against each other is reasonable insurance against corrupted data, but an additional
check gives an even greater level of confidence in the data. Of the three copies of the
uplinked key code, the second 5-bit string is the complement of the key code. To
illustrate this, consider the keyboard code for “Verb”, which is binary 10001. The
INLINK register is shown in Figure 41.

![Figure 41: Data in INLINK register](image)

The first five bits, in positions 15 through 11, contain the Verb code, 10001. The
next five bits, in position 10 through 6 are an inverted copy of the Verb code (binary
01110), and the final bits repeat the normal representation of the Verb code.
Checking the code against its complement, and then another, non-complemented
copy of the code reliably eliminates the possibility that random radio noise would
generate a valid input sequence.

After validating the uplinked character, it is passed to the CHARIN routine of
Pinball, where it is processed in the same manner as all other DSKY keyboard input.
From here, key sequences sent via remote control from the ground are no different
from having a crewmember physically press the keys in the spacecraft. Most uplink
operations modify data in erasable storage, but many updates are to ranges of
erasable storage words that might not have a Noun associated with them. Solving
this issue requires the introduction of a special set of extended Verbs. Designed to
function only with the uplink software, Verbs 70 through Verb 73 are not available
for the crew to use. Upon recognizing one of these Verbs in the uplink data stream,
the Major Mode display changes to Program 27 to indicate that the update is in progress.\footnote{12}

Each of these Verbs performs a specific update function. Verbs 70 and 73 update the AGC’s clocks, using the uplinked data to adjust the time up or down. Invoking Verb 71 allows the update of several words of contiguous storage. The first entry is the count of the number of words to follow, the first of which is the starting address parameter, with the update data trailed by Verb 33 (Proceed Without DSKY Inputs) to finalize the update.

Verb 72 is similar to Verb 71, except that the updates are not necessarily in contiguous storage. After the number of words to be updated has been passed to Verb 72, pairs of storage addresses and data are sent to the computer.

Note that in the cases of these four extended Verbs, data is not loaded into any of the three display registers. Freed of this physical limitation, Verbs 71 and 72 can accept between 10 and 20 words of data per update. Verbs 70 and 73 adjust the timings in the AGC. Verb 70 updates the liftoff time, and Verb 73 updates the AGC clock.

\begin{verbatim}
V71E  Begin updates
   5E  5 entries to follow
01016E  Data will be stored at location 010168
14000E  First word of data
01603E  Second word of data
16067E  Third word of data
V33E  Update complete
\end{verbatim}

Figure 42: Sample Verb 71 in Uplink

At this point, it is not yet apparent how these update Verbs prevent problems with slow and noisy data transmissions from the ground. The most important safeguard is to buffer the updates in memory during the update process, leaving critical AGC data untouched during this period. Only after the uplink is complete and all the data is validated will the update process begin. At this point, the ground sends a Verb 33 sequence to indicate that the uplink is complete (no additional data is expected), and the software can now copy the buffered data over the AGC’s existing data. First, a restart phase is defined so that any interruption or failure will insure that the changes are implemented. Next, the actual update of the data is performed. Because it is running as an extended Verb, which in turn is running as a Pinball routine, it has the luxury of running with interrupts inhibited. Under this

\footnote{12} Here is an excellent example of how the Major Mode does not necessarily correspond to an explicitly running program. The update routines are only extended Verbs, but the update process is such a change to the normal state of the system that a new Major Mode is justified. Although the DSKY’s “PROG” display switches to Program 27 from Program 00, a new job is not scheduled to run.
configuration, there is no need to worry about losing control of the processor and spoiling the update, even for a short time. After copying the update data from the buffer to its destination, the software returns to get the next set of DSKY commands from the ground. When all the updates are complete, the ground sends Verb 34, which terminates all update operations and returns the Major Mode back to Program 00.

SYNCHRONOUS I/O PROCESSING AND T4RUPT

Communicating with the hardware external to the AGC is a major responsibility of a control computer such as the AGC. In addition to the wide range of hardware that the computer interfaces with, the AGC must also adapt to the responsiveness of each device. At one end of the spectrum, software routines identify an action for a piece of hardware and send a command to that device with the expectation that that action occurs immediately. For example, thruster firings are important I/O requests which occur frequently and demand timings on a millisecond scale. Thrusters must fire the moment that the data is placed in the I/O channel, and they need not respond with any status information. Other devices can be managed satisfactorily without such near-instantaneous responses. Only a few of these asynchronous operations exist in the AGC, and they are usually limited to initializing a device prior to its first use. For example, powering up and initializing the inertial platform requires about two minutes to complete. Such a device, with its low latency requirements, eliminates the need for high resolution timers or dedicated interrupt vectors and their associated software. Perhaps the most familiar instance is the DSKY, which responds quickly to software commands, but in turn, must interface with the slowest component of all, the human crew.

Another group of devices are characterized by needing periodic attention, whether to check their status, to update their displays or to monitor their physical movement. The handling of these tasks is performed by the T4RUPT routine, named for the interrupt that drives its execution. Devices managed by the T4RUPT do not raise an interrupt or otherwise alter the AGC’s overall processing. Rather, T4RUPT runs according to a fixed schedule, interrogating I/O status bits and directing processing based on their settings. Unlike thruster timers or DSKY keyboard interrupts, T4RUPT manages I/O synchronously, forcing each device to wait its turn to have its status bits polled. Outputting data for the DSKY display also operates under a similar fixed schedule.

T4RUPT is scheduled through the T4RUPT clock, one of the four event timers in the AGC, and it schedules eight distinct operations, or “passes”, 120 ms apart, repeating the sequence every 960 ms. Breaking T4RUPT into several passes has two key advantages. Many devices are serviced more than once during the T4RUPT 960 ms cycle, which improves the responsiveness of the computer to the devices’ needs. Also, using several small tasks necessarily limits the amount of processing that is performed during each pass. As all timer interrupts are entered in interrupt-inhibited mode, all work must necessarily be very brief. It might appear that the
timings for T4RUPT are oddly precise, especially when a change of a few milliseconds would make little difference in handling the devices. That is, surely rounding to a full second between the time T4RUPT starts its first pass would be more intuitive? The answer lies in the limitations of the TIME4 clock. At centisecond (i.e. 10 ms) resolution, it simply cannot operate at timings of exactly 1/8, or 0.125 second.

Prior to determining which 120 ms phase is scheduled next, T4RUPT performs two tasks. First, the DSKY output table is checked to see if any output is waiting to be displayed. If an update to the DSKY display or its status lights exists, the output word and its relay codes are assembled in the accumulator and the data is written to output channel 108. Output channel 108, like all output channels, is seen as a set of “flip-flops”; digital devices that maintain their data until a change is commanded. All the while data is in output channel 108, the AGC maintains the signal in its “flip-flops”. But keeping the DSKY relays endlessly energized is unnecessary. As the relays are designed to latch into position within a short while, power can be safely removed after the latches are secure. To apply power for much longer than 20 milliseconds would not only result in unnecessary power consumption but also risk overheating the relay coils. With these constraints in mind, T4RUPT will schedule another interrupt to clear the output channel and remove power from the relays. When this special case of T4RUPT processing occurs, the next interrupt is scheduled for 100 ms in the future to maintain the 120 ms T4RUPT processing cycle.

Checking the status of the Proceed key is the second DSKY-related task T4RUPT performs before any 120 ms cycle. Unlike all the other DSKY keys, Proceed does not generate a KEYRUPT interrupt. Rather, it sets bit 14 of input channel 328, which is interrogated at the beginning of each 120 ms pass. When a crewmember presses Proceed, T4RUPT will detect this state and schedule a Pinball job to process this keystroke.

So far, IMU processing has taken up only two of the eight available passes through T4RUPT, leaving several cycles available for servicing other devices. Up until this point, the implied assumption in this discussion of T4RUPT is that processing is similar for both the Lunar Module and Command Module. However, this is true only for IMU and DSKY processing, which are the only TRUPT4-managed devices that are common to the two spacecraft. The remaining operations performed in the T4RUPT processing cycle are tailored for each spacecraft’s environment. In the Lunar Module, for example, the remaining T4RUPT processing passes focus on controlling the radars and converting their data for analogue displays, and for monitoring changes in the RCS configuration. Lacking radars and computer-selected thrusters, the Command Module’s AGC uses T4RUPT to manage the optics assembly. Data from other AGC programs will request an orientation of the sextant and telescope assembly, and the T4RUPT routine controls the movement of those devices.

Although eight passes are scheduled during each 960 ms T4RUPT cycle, only four different routines are executed. To improve I/O responsiveness, each routine is scheduled twice, always 480 ms apart. For example, the IMU’s monitor executes during the second pass of T4RUPT, and will also run 480 ms later during its sixth
pass. A check of the RCS configuration in the LM is scheduled for the fourth pass, and again during the eighth and final pass, 480 ms later.

T4RUPT processing in the Command Module is significantly different than its Lunar Module counterpart. IMU processing, such as health checking, gimbal lock detection, power-on and power-off, are essentially identical for the two spacecraft. However, T4RUPT in the CM has no radars to command, nor input changes defining the RCS configuration. Rather, T4RUPT processing centers on the optics system, its health and status checks, and driving the sextant to its desired position.

**LM T4RUPT passes 0 and 4: RCSMONIT**

In the lunar descent, vehicle controllability is of paramount importance, especially during the final landing phase. Attitude control is managed by the Digital Autopilot, creating a responsive “fly-by-wire” system that takes commands from both the targeting software and the astronaut inputs. Given that the nature of a lunar landing allows “one attempt, no second approaches”, a failure in any spacecraft system must be addressed immediately. In the case of the Reaction Control System, the crew can quickly shut down failing attitude control thrusters by closing their propellant valves using switches on the LMP’s panel. These shutoff valves are organized by thruster pairs, and are designed to minimize controllability problems if a thruster has to be disabled. This intervention is communicated to the AGC by setting a bit in input channel 32a. Eight bits, one for each thruster pair, are checked during each 480 ms T4RUPT cycle. When a thruster pair shutoff is detected by the T4RUPT routine, the Digital Autopilot’s understanding of the RCS configuration is updated, and the DAP adjusts its control laws to prevent selecting the disabled thruster jets. In contrast, the Command Module has no comparable set of T4RUPT routines and the computer is not notified of any changes in the RCS configuration. If a thruster is disabled, the crew must manually update that spacecraft’s DAP configuration using Noun 46.

**LM T4RUPT passes 1 and 5: RRAUTCCHK**

The Rendezvous radar antenna dish is located above the “face” of the LM ascent stage, and can rotate on two axes to point directly at the CSM transponder. The radar antenna is mounted on the end of a shaft assembly and gimbals allow the dish to rotate left and right. The radar shaft is then mounted perpendicularly on the trunnion assembly, which in turn is mounted on the ascent stage. Gimbals on the trunnion are oriented parallel to the Y-axis, and allow rotation of the shaft/antenna assembly in the vertical axis. Resolvers in the shaft and trunnion gimbals send angular data to the CDUs, generating counter interrupts in the AGC.

The rendezvous radar tracks the CSM target vehicle in two different ways. First, its internal electronics can acquire, lock onto, and automatically move the radar antenna dish to follow the relative movement of the CSM. The second mode is for the AGC to compute the CSM’s position and drive the radar antenna to where it expects the CSM to be. As the LM maneuvers, or if different orbits cause the relative positions of the two vehicles to change, the AGC commands a new position for the antenna to point. During T4RUPT passes, the AGC sends commands to the shaft
and trunnion gimbals to reposition the antenna to its new orientation. This motion can occur slowly, at 1.3°/sec, or more rapidly, at 7°/sec. In all likelihood, several T4RUPT passes will execute during this repositioning operation. T4RUPT monitors the antenna’s progress, and stops driving the gimbals when the desired orientation is reached. If the radar is tracking automatically and is not relying on the AGC for pointing control, T4RUPT will monitor the antenna’s shaft and trunnion angles and if they exceed a critical value then the AGC will command the antenna back within limits.

During a T4RUPT pass, the radar’s health and status is checked. Like the IMU, a power-up request requires special handling to initialize the electronics properly, and to prevent jobs that wish to use the radar from referencing it during this time. Powering down, or a failure in the radar electronics, forces a check for any work that might require the radar. A program alarm terminates any work using the radar, and the TRACKER warning light illuminates on the DSKY. T4RUPT also performs basic checking of the landing radar, verifying that the altitude and velocity sensing

![Image](image_url)  
**Figure 43:** LM rendezvous radar at top of ascent stage
electronics have not failed. A problem in these electronics will not cause a program alarm or terminate any active work, as to do so might cause a loss of data during the critical landing phase of the mission. Note that this is different from the “radar data not good” discrete, which is often due to an antenna positioning error and is not the result of a hardware failure. Although programs continue to run, the TRACKER warning light illuminates to warn the crew the radar program is not able to follow its target.

**LM and CSM T4RUPT passes 2 and 6: IMUMON**

Two T4RUPT passes are dedicated to checking the status and health of the Inertial Measurement Unit. Applying power to the IMU and its related systems will set the POWER ON request in bit 14 of input channel 308. T4RUPT tests this bit every 480 ms, and if it remains set for one full T4RUPT cycle (960 ms) then a 90 second delay in any IMU processing is begun. By that time, the IMU should have completed its power up and initialization sequence, and the gyros should be up to speed. If no errors are detected, T4RUPT performs additional failure checks, and if everything is in satisfactory order will initialize the CDUs and direct the gimbals to move to their coarse align orientation. The NO ATTITUDE lamp is illuminated on the DSKY, informing the crew that the platform is not usable until it is aligned to a known reference.

After the testing of IMU state changes is complete, the T4RUPT pass continues by checking other bits in input channels 308 and 338. Failure indications such as temperature out of limits, or failures in any of the PIPAs, CDUs or the IMU itself, are presented in these channels. When any of these conditions arise, the IMU is not considered usable, and is placed in a coarse align mode, losing its attitude reference in the process. When T4RUPT detects such a failure, it illuminates a warning light on the DSKY panel to alert the crew to the loss of the IMU. In other cases, the IMU and its associated electronics may be operating normally, but other health checks might reveal issues. A problem in the PIPAs, identified in input channel 338 bit 13, is a serious fault and generates a program alarm. Used by the IMU to measure changes in acceleration, and, over time, velocity, a PIPA failure results in the loss of velocity data to the AGC, but does not cause the IMU to lose it attitude reference. While critical, the IMU is still operable in a degraded mode in which it provides only attitude references.

If AGC telemetry uplink or downlink is too fast, an error in channel 338 is raised and T4RUPT schedules a program alarm to alert the crew and clear the error. An additional test checks whether the crew is requesting the IMU be caged. Never performed during normal operations, caging the IMU forces the platform to a known, fixed orientation, and resets its associated electronics. All attitude and velocity data is lost and the vehicle must rely on a backup system (the SCS in the CM, and the AGS in the LM) for attitude control. Only after the platform is realigned and an updated state vector is either calculated or uplinked, is the platform usable for guidance, navigation and control. Once T4RUPT detects the caging request, it resets the software and illuminates the NO ATT warning light.

The IMU hardware itself does not detect or report on a potential gimbal lock
situation. Only during the T4RUPT processing is the middle (Z, or yaw axis) gimbal tested against gimbal lock conditions. When the middle gimbal moves into the range between 70 and 85 degrees, the GIMBAL LOCK light is illuminated. At this point, the light is only a warning, although a very stern one, that the spacecraft will lose its attitude reference if rotation around the Z-axis continues. In this region, the attitude reference is still valid, and no harm results if the rotation is reversed. If the middle gimbal continues past 85 degrees, gimbal lock occurs and all attitude references are lost. The IMU is again placed into a coarse align state, which is unusable as an attitude reference, but is ready for the crew to perform realignment procedures.

**LM T4RUPT passes 3 and 7: DAPT4S**

For the Inertial Measurement Unit to provide an attitude reference, it must be oriented to a known, fixed orientation in space. This orientation might be the launch or landing site, or an attitude required by the spacecraft for a major maneuver. As the LM rotates, the gimbals in the IMU allow the platform to maintain its fixed orientation in space. DAPT4S takes the gimbal information and makes two sets of single-precision matrices for use in attitude computations. The first translates the gimbal angles into the coordinate system used by the spacecraft body axis, which are recognizable by the pilot as roll, pitch and yaw. Second, it rotates that matrix back to the coordinate system used by the gimbals to produce an easily referenced, single-precision matrix representing the gimbal angles. DAPT4S is unique in that it executes four times during T4RUPT processing. In addition to being called explicitly in T4RUPT passes 3 and 7, RRAUTCHK also transfers control to DAPT4S in passes 1 and 5, causing the attitude data to be calculated every 240 ms. Not all functions of the Digital Autopilot require highly precise values for the spacecraft attitude, making the single-precision version of this data perfectly acceptable. None of the data calculated during these passes is used by T4RUPT routines, the data is saved in an area of memory ready for use by other routines.

**CM T4RUPT passes 0 and 4: OPTTEST/OPTDRIVE**

During OPTTEST/OPTDRIVE T4RUPT passes, the AGC first determines if the optics need to be moved to another orientation. If so, the optics are driven to the position commanded by the tracking or IMU alignment program or through a DSKY entry supplied by the crew. The sextant trunnion gimbals are limited in the range it can move – while the sextant shaft may rotate a full 360 degrees, the trunnion is limited to 90 degrees. Within this trunnion movement, the sextant’s view is only 67 degrees before the spacecraft structure obscures the view. When driving the optics to their new location, an output register is set with the number of “pulses” requested to move the optics assembly. Each pulse translates to a discrete amount of motion, but the two optics axes do not move at the same rate for each pulse transmitted. One pulse sent to the trunnion drive motor will move the sextant 40 arc-seconds, yet the same pulse will command the shaft to move the sextant 160 arc-seconds. These translate into a trunnion rotation rate of 3.8°/sec,
and a shaft rate of 15.3°/sec. The two CDUs are limited to 165 pulses in a single T4RUPT pass, or 1.83 and 7.33 degrees of rotation in the trunnion and shaft, respectively. This constraint assures that the previously commanded rotation is completed by the time of the next T4RUPT pass. During the repositioning process, the software checks the orientation of the optics not only to assure that the target orientation is achieved but also to prevent a potential hardware problem. Electrical and mechanical limitations of the optics assembly require avoiding movements near the trunnion’s limit of travel. If the optics were to come near this zone, the T4RUPT will halt any further commands to the trunnion.

**CM T4RUPT passes 1 and 5: OPTMON**
The optics assembly comprises a wide field of view, unity-power telescope and a 28-power sextant. Eyepieces for the telescope and sextant are mounted on a panel in the lower equipment bay, and observe through a window in the hull of the CM. In general, the AGC controls only the sextant, monitoring the angle at which it is pointing and driving it to a new orientation. Only when the telescope is explicitly slaved to the sextant will it be under the control of the computer. The sextant is used to identify the proper orientation for the inertial platform, and to obtain navigation fixes by taking sightings of the Earth and Moon.

Managing the optics system in T4RUPT begins with a check of the CDUs, which convert the sextant’s angular orientation into counter pulses that are usable by the AGC. A failure in this hardware sets a bit in the AGC, which T4RUPT senses and then raises a program alarm to alert the crew that the sextant is unusable. Next, the optics mode is tested to see if the sextant is being operated manually, or if it is commanded automatically by the AGC. A manual mode, of course, will inhibit any software requests for driving the sextant to a new position and will generate a program alarm if this is attempted. A third possible state is for the optics to be in the process of being “zeroed”. Zeroing is performed before every sighting operation, and drives the sextant to a known position and resets the CDUs.

**CM T4RUPT passes 3 and 7: RESUME**
Unlike the Lunar Module AGC software, the Command Module version does not use all of the T4RUPT cycles: passes 3 and 7 simply return control to the T4RUPT processor without performing any operations.

---

13 A rather cynical comment says that only the United States government could be fooled into purchasing a 1x power telescope. This, of course, ignores the fact that the telescope provides an exceptionally wide field for locating the navigation stars, and can be slaved to the sextant. After centering the star in the telescope, attention turns to the sextant with its much finer resolution for the actual sighting.
HIGH LEVEL LANGUAGES AND THE INTERPRETER

Since the start of the computer era, there has been the need to express problems for the computer in a form that is understandable to humans, yet easily mechanized for execution in a processor. Early systems placed the burden of effort wholly on the programmer, forcing them to structure their problems in steps that corresponded one-for-one with the basic machine operations. One can only imagine the difficulty in translating symbolic mathematical equations into the arcane language of basic arithmetic and procedural operations. In the section on AGC instructions, Figure 24 showed an example of using basic instructions to add a list of numbers. While quite useful as a demonstration of several interesting AGC instructions, it also showed the level of detail necessary for performing even trivial tasks. Expending a large amount of effort to develop such a routine is acceptable if the code remains static. However, if there is a change in the AGC instruction architecture, or if the routine is ported to an entirely new architecture, we are forced to rewrite the code to reflect the new hardware characteristics. The time and expense of such a recoding effort is often so high that it negates using a newer, more flexible architecture. A not completely implausible example would be implementing the routine to sum a table of numbers in the Space Shuttle computer. The Shuttle’s General Purpose Computers are a completely different design than the AGC, so our program must be rewritten to reflect the different architecture. Figure 44 shows how this might look in the computer’s native, low-level language.14

While the Space Shuttle code is arguably more compact and efficient, there is still the issue of rewriting the summation program to use multiple registers, indexes and a

| SR | R2,R2   | SET SUM IN REGISTER 2 TO ZERO          |
| SR | R3,R3   | STARTING INDEX IS ZERO                |
| LA | R4,4    | INCREMENT IS 4 BYTES IN THE TABLE      |
| LA | R5,20   | FIVE TABLE ENTRIES * 4 BYTES/ENTRY    |
| LOOP | A | R2,NUMBRTAB(R3) | ADD THE NEXT ENTRY IN THE TABLE |
| LOOP | BXLE | R3,R4,LOOP | INCREMENT THE INDEX, CHECK IF WE ARE DONE, IF NOT, CONTINUE |
| NUMBRTAB | DC | '2' |
| NUMBRTAB | DC | '5' |
| NUMBRTAB | DC | '7' |
| NUMBRTAB | DC | '4' |
| NUMBRTAB | DC | '6' |

Figure 44: Summing a table of numbers in the Space Shuttle computer

---

14 Some readers will recognize this code as IBM 360 assembler. The Shuttle General Purpose Computer, the AP-101S, is based on the classic IBM System 360 architecture. Although the AP-101S is not completely compatible with the S/360 series, it is sufficiently similar that using S/360 assembly for this example is acceptable.
complex branch instruction. Clearly, the goal is to express the problem in a form that
is as independent of the underlying architecture as possible. At the same time, the
notation should more closely reflect how the user would express the original
problem. High level languages are a huge improvement over machine level coding, as
they reflect the problem statement more closely, and are far easier to read and
maintain. Literally hundreds of languages exist, each with its own target audience,
but some of the most familiar are C, Java, and yes, even Cobol. The power of a high
level language is easily demonstrated by the trivial assignment of the value “1” to the
variable “X”. Coding this in the AGC might look like this:

```
CA ONE COPY THE VALUE OF 1 INTO ACCUMULATOR
TS X SAVE THE VALUE OF 1 IN STORAGE
...
X DEC 0 STORAGE AREA CALLED “X”
ONE DEC 1 VALUE OF ONE
```

The same expression can be coded in the C language:

```c
integer x = 0;
x = 1;
```

While this small code fragment is not a particularly dramatic improvement over the
AGC code, mathematicians will immediately understand the intent. We now expand
the idea of high level coding, using our number summation program written in the C
language:

```c
integer numbtab[5] = {0,1,6,4,2,3};
integer sum = 0;
integer i = 0;
for (i=0; i++ ; i<5) {
    sum = sum + numbtab[i];
}
```

While not completely intuitive, this code is far more readable than our attempts with
machine coding, and at first glance appears to be the solution to our coding needs.
While the program is more “human friendly”, it cannot execute in its current form
without a conversion to the processor’s native machine instructions. A software
program known as a compiler performs this task, taking each of the high level
statements and generating the equivalent machine instructions. Unfortunately, there
are important limitations to the quality of code that a compiler can generate. In
particular, early compilers did not produce efficient code, compared with the best
hand-written machine instructions. The fact that both execution and memory
requirements are larger for compiled programs, made high level languages
impractical in the resource-limited AGC.

A compromise between the high efficiency of machine code and the easy
programmability of a high level language is realized in the AGC’s Interpreter. As
implemented in modern computers, interpreters perform a similar function to that of a
compiler. Both begin with the source code written as a high level language. After
taking the source statements and generating machine instructions, a compiler will never need the original source code again. Interpreters do not generate machine level instructions for the hardware to execute, but use a different process to convert high level statements into a form that can be processed. The simplest interpreters begin with the original source code, and execute the program directly without converting the program into a more efficient representation. Each source statement is reparsed and reinterpreted each time it is encountered, regardless of how many times the statement has been executed previously. As expected, this approach is very slow and inefficient, and there are no intermediate results from the interpretive process to save and reuse.

The output of more sophisticated interpreters is not machine instructions as produced by a compiler, but an intermediate language, often called “pseudo-code” or “P-code”. This interpretive code is the “language” of the interpretive system, and is analogous to a processor’s machine instructions. An imaginary, or “virtual”, processor is written in software, and it reads and executes the pseudo-code as its “basic instructions”. A huge benefit of such a virtual machine is the ability to create any desired processor characteristic, bypassing the limitations inherent in the underlying hardware’s architecture. In the description of the basic design of the AGC, the lack of support for index registers and sophisticated datatypes is apparent. With an interpreted system, many of the limitations of the AGC’s architecture are neatly eliminated. Interpreters are targeted for developing applications such as the mission programs, rather than system-level coding in the Executive. Restricting the interpretive software to mission software eliminates the complexities involved with managing hardware operations such as interrupt handling. This is usually not a problem, since Executive programs rarely use complex datatypes or complex indexing, and cannot afford the greatly increased execution time that interpreted programs require.

Unlike contemporary languages like Java, whose statements are converted to “P-code” and then executed, there is no comparable high-level language for the AGC. The task of efficiently converting human readable code to an internal format requires sophisticated optimizing compilers and interpreters, which are fiendishly difficult to develop, and were simply not available to the AGC developers. To create the most efficient programs possible, the AGC developers wrote the intermediate code directly, bypassing the troublesome conversion from a high level language.

**Parsing high level language statements**

To understand the AGC’s interpretive system, it is essential to return to a fundamental issue. What is the process of converting the high level logic and equations into executable machine instructions? Beginning with a simple equation:

\[ a = \frac{(b^2 + c^2)}{(b^2 - c^2)} \]

The limitations of typing on a computer require equations to be coded on a single line:

\[ a = (b**2 + c**2) / (b**2 - c**2) \]
While perhaps not as graphically satisfying as our original expression, the new representation is equivalent and has its own advantages. The sequence of operations is more apparent, using the convention of reading from left to right. Parentheses are used to group operations together and eliminate any ambiguity in identifying which terms should be executed first. Because superscripts are not possible in this simple single-line notation, exponents are replaced by the characters “**”. Using this notation, \(b^2\) is now rewritten as \(b**2\). Starting with the expression immediately following the equals sign, the value of \(b\) is squared, as is the value of \(c\), and the two are added together. Now that the numerator has been evaluated, it is time to begin work on the denominator. However, both the numerator and denominator must be evaluated before the division operation can proceed. To preserve the work already done on the numerator, the intermediate value of \((b^2 + c^2)\) is set aside in a temporary location that we will call \(T1\). Next, the value of \(c^2\) is subtracted from the value of \(b^2\). Like the first half of the equation, this intermediate value is saved as well, this time in a location called \(T2\). The only remaining operation is to divide the temporary value of \(T1\) by the temporary value \(T2\), and save the result in the variable named \(a\).

Processing an equation requires reading and parsing the entire statement, and analyzing the result to determine the proper execution sequence. The notation used has its roots in computer science, and is called a “tree structure”. Many problems in computer science lend themselves to being expressed as trees, especially those with complex interactions. In its simplest form, a tree structure has “branches” and “leaves”, akin to its physical namesake. Operands such as variables are the leaves, are connected to branches, and are joined by a single computer operation. In addition to benefiting the programmer with a graphical description of the problem, translating a tree structure into basic computer instructions is quite straightforward. An expression, \(3 + 2 = 5\), has two numbers added together to create a sum. In a tree structure, this expression is described in Figure 45(a). Read from the bottom-most leaves upwards, we take the two operands, three and two, and add them together to create five as the sum. Extending this form to an expression such as \(X = 3 + 2\), we have the variable \(X\) assigned the sum of three plus two in Figure 45(b); and 45(c) creates a generalized expression which retains the same structure, yet now is usable for any set of input variables.

![Figure 45: Expressions in tree structures](image)
The full range of binary operations are available to the programmer, including addition, subtraction, multiplication and division. Unary operators also exist, and include not only negation, but also mathematical operations such as square and root plus trigonometric functions.

![Unary and trigonometric functions](image1.png)

\[
x = -a \quad x = a^2 \quad x = \sqrt{a} \quad x = \sin(a)
\]

**Figure 46:** Unary and trigonometric functions

As with all building blocks, each such individual element appears barely worthy of even passing attention. It is only when they are combined that they become truly interesting. Consider a slightly more complex expression, \( E = mc^2 \). This familiar equation requires both a unary and a binary operation. Traversing the tree upwards, the first step is to square the value of \( c \). The result is multiplied by \( m \), and the final value placed in \( E \).

![Tree representing \( E = mc^2 \)](image2.png)

**Figure 47:** Tree representing \( E = mc^2 \)
The ability to assemble basic operations into tree structures to define an equation is now becoming apparent. A final, more complex example allows us to address the more important issues that arise in real world problems. Consider the polynomial expression \( ax^2 + bx + c = 0 \). Solving for the value of \( x \) requires the quadratic equation:

\[
x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}
\]

Two values for \( x \) are possible, so we must break it into two separate expressions.

\[
x = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \text{ and } x = \frac{-b - \sqrt{b^2 - 4ac}}{2a}
\]

The trees differ only by the plus or minus term in the numerator, so our next example works equally well with either expression. A tree structure of the first equation looks like:

![Tree Diagram](image)

**Figure 48: Quadratic equation tree**
The quadratic equation uses two variables (a and b) more than once, which complicates our tree somewhat. The variable b is both squared and negated, and a is in both the numerator and denominator. The equation is already in its simplest form, so it is not possible to combine or cancel out either variable.

**Traversing the quadratic equation**

We return to one of the possible roots of the quadratic equation:

\[
x = \frac{-b + \sqrt{b^2 - 4ac}}{2a}
\]

The traversing process starts in the lower left-hand corner of the tree, with the goal of traversing all the branches to the right and upwards. Only a few rules are necessary for navigating the tree, and they reflect the natural process of evaluating an expression from left to right.

- Start at the lowest, left-most leaf on the tree. If there are multiple branches on a leaf, start with the left-most branch.
- Variables, constants and temporary variables are all handled in the same manner.
- A branch is traversed only once.
- All functions leave their results in the accumulator.
- Upon reaching a function by going up one of its branches, take the other branch down before continuing back up. Continue going down the tree until a leaf is found.
- When a leaf is found, go up one level and execute that function.
- If two leaves are found underneath a function, select the left-hand leaf first, then the right one.
- When traversing up to a function that has another untraversed branch, look ahead to see if that branch contains another function. If it does, and we have arrived at the function through the left-hand branch, save the contents of the accumulator in a unique temporary variable. Once saved, the accumulator is cleared to zero. Otherwise, the intermediate values already calculated up to this point would be lost when traversing to the other branch.
- Operations are performed only when going up branches on the tree, never down.

The process of traversing the quadratic equation tree uses the following steps:

1. Start with the variable leaf b.
2. Negate the value of b.
3. Addition has two operands. Looking down the tree, there are two branches underneath this function, and both are functions themselves. Coming up from the left branch, the accumulator must be saved in a temporary variable that we will call T1. Next, the accumulator is cleared to prevent interference with the next set of arithmetic instructions. The next branch is down and to the right, to the Square Root function.
4) Square Root is a unary function. No special handling is necessary, and we continue down to determine its operand.

5) Subtraction, like addition, requires two operands. Both are functions, requiring that the intermediate variable be saved when we finish traversing the left branch.

6) The first function is Squaring. Like Square Root, Square is a unary operation, so we continue down the next branch and see the variable b. This is squared, placed in the accumulator, and we move up the left branch of the Subtract function.

7) Back at the Subtract function, all of the operations on the left-hand branch are completed. Noticing that the right-hand branch is another function, we will again save the accumulator, this time in temporary location T2.

8) Moving down the right-hand branch of the Subtract, we encounter the Multiply operation. On the left branch we have the constant 4, so no further progress is possible here. The branch to the right will provide the second operand.

9) The branch takes us to another Multiplication function, with its two branches leading to the variables a and c. There is now enough information to perform the Multiply, so the product of a and c is placed in the accumulator.

Figure 49: Labeled quadratic equation tree
10) Moving up with a value in the accumulator, the next step is to perform the Multiplication of the accumulator and the constant 4. Once the Multiply is complete, the traverse up the tree can continue.

11) Returning to the Subtraction operation, there is the left-hand value stored in the temporary value T2, and the value left in the accumulator from the just completed traverse of the right-hand branch. The contents of the accumulator are subtracted from T2, the result saved in the accumulator, and the traverse continues upwards.

12) Back to the Square Root, now with a value that is usable for that function. Take the Root of the accumulator contents, replacing the existing value in the accumulator.

13) We now find ourselves in a similar situation as we were in step 11. The value for the left-hand side (-b) is saved in temporary variable T1. The addition operation takes T1 and adds it to the accumulator.

14) Calculations for the numerator are now complete, and attention moves to the denominator of the equation. The division operation has a value provided through its left-hand branch, but the right-hand side is a Multiplication function. We will save the intermediate result in another temporary variable, T3, and begin the traverse down the right-hand branch. Again, the accumulator is cleared when creating the temporary variable.

15) The Multiplication has the two variables available as leaves on the tree, so we can perform the Multiplication and put the result in the accumulator.

16) The Division now has both variables. The numerator from the left branch in variable T3. This is divided by the contents of the accumulator built from the operations on the right branch. Since the denominator is in the accumulator, we need to exchange it with T3 before the Division takes place.

17) We’re finished! All the arithmetic operations are complete, and the final value is saved in variable X1.

Stacks and temporary variables
This admittedly elaborate example is useful for showing several key concepts of how equations are processed by an interpreter and executed. Parsing and generating the tree structure is the most familiar and practical methodology to represent the left-to-right evaluation of a mathematical statement. With only a single accumulator, the limitations of the AGC become apparent. Finding ourselves with intermediate values that are not usable immediately, saving the accumulator in a temporary location is necessary. In managing these temporary variables, a few surprising properties of a tree structure become apparent. First, an intermediate value needs to be saved only once, and needs to be retrieved only once as well. Although there may be many intermediate values saved at one particular time, a pattern emerges in the sequence of storing and retrieving the data. The most recently saved temporary data is always the first brought back into the equation. Any data saved previously is recalled only after more recently saved
data is itself retrieved, a process called “first in, last out”. To see this property demonstrated, assume three intermediate values T1, T2 and T3 are saved in sequence: T1 is saved first, then T2 and lastly T3. Only after the point where T3 is retrieved will T2 be needed for its part of the equation. Continuing further, only after T2 is used will T1 become available in the expression.

A restaurant kitchen offers a perfect physical analogy of how the temporary variables are referenced. Plates are stacked in a dispenser cart, which contains a spring to keep the top plate visible and easy to grasp. Whenever a clean plate arrives from the dishwasher, it is placed on top of the stack, pushing down whatever other plates might already be in the dispenser. When the chef needs a new plate, she takes it from the top of the stack and the remaining plates “pop up”. With this analogy, we have described the most common and powerful data structure in computer science, the stack. When data has to be stored in an ordered sequence and retrieved in reverse order, stacks are used to store and keep track of the data. The stack is especially appropriate where one program calls another, which in turn, calls a third, which calls a fourth, etc. Local variables and parameters are pushed onto the stack at each successive program call in the same manner that plates were placed onto the dispenser. As each program completes processing and control returns to its calling program, the data is “popped” from the stack.

A particularly important characteristic for the AGC is that a stack uses memory very efficiently. Consider the case if, instead of a stack, we explicitly assigned a word of memory for each temporary variable that we might use. Programs generally do not simultaneously use all temporary variables at all times; yet without a stack, all of the storage for temporary variables must be allocated in advance. Memory is wasted since only a few temporary variables are needed at any one time, leaving many words of storage sitting idle. Granted, it is possible to develop elaborate schemes to reuse specific memory areas, but these are cumbersome and error prone. In contrast, a stack “grows” in storage only as a program requires it.

Stack implementations follow the physical analogy of the plate dispenser only so far. Each time a plate is added or removed, the remaining plates physically move up or down in unison. In designing a software version of the dispenser, there is no need to move the data back and forth in memory. Rather, we reserve enough storage to hold the maximum amount of data that might be expected in the stack. The first item is placed in the first storage location in the stack memory area, and is (by definition) on the “top”. When the second item of data needs to be placed on the stack, the first data word is not moved out of the way. Instead, the new word is placed in the second storage location in the stack. This action creates a dilemma: specifically, how is it possible to keep track of the location where the “top” of the stack is? Solving this problem is done by introducing a new variable that is an integral part of a stack implementation, the stack pointer. Stack pointers, as their name implies, always reference the most recent item in the stack.

“Pushing” a new item of data onto the stack requires that the stack pointer change its reference to the next available location, into which the data is copied. “Popping” data off the stack operates in the opposite way. After retrieving the data at the top of the stack and passing it to the requesting program, the stack
pointer is updated to reference the next oldest entry. Note that only the value of
the stack pointer changes during the push and pop operations, and the data in the
stack remains fixed. Clearing out the locations that previously held the “popped”
data is not a mandatory housekeeping task, but it is often good practice to do so.
All stack operations must use the stack pointer to reference the data, and only the
top element is legitimately available. From this set of assumptions, any storage
not within the scope of the stack at any given moment is irrelevant.

Double and triple precision datatypes
The notion of “datatypes”, that is, whether a variable is single or double precision,
integer or fractional, has had little relevance until this time. In the task of analyzing
an equation and creating a process tree, the exact characteristics of the datatypes
were not important. Only now, as the process tree is converted into machine
instructions do these characteristics need to be considered. The example for finding
one root of a quadratic equation used single precision values, but double or even
triple precision variables are possible simply by generating a few extra machine
instructions for each arithmetic operation. Generating an extra instruction or two
might be acceptable for multiple precision values, if done sparingly. However, when
complex datatypes (such as vectors and matrices) are used, repeatedly generating the
same instruction sequences consumes large amounts of memory. Using a traditional
subroutine library for complex operations is one alternative, with parameter lists of
input and output for exchanging data with the main program. Unfortunately, this
approach also consumes memory at a prodigious rate, and incurs the overhead of
moving data in and out of the parameter list.

At this point, extending the AGC architecture further only results in marginal
functional improvements. Even if the new system extensions are desirable, the
additional level of system complexity is expensive in terms of the development and
testing effort.

Thus, we conclude that the AGC is ripe for a radical departure from its currently
implemented architecture.

THE INTERPRETER

Thus far, this chapter has focused on the system level software found in the
Executive. System routines are characterized by their own type of complexity and
operate in a world of interrupts, timers and process management. Logically complex,
the programming focus of the Executive software is hardware, and it uses simple
datatypes such as single and double precision variables. While the Executive routines
are essential, they are only the necessary overhead required for the most important
software: the mission-specific programming. Mission software is very different from
that in the Executive. Each mission program aims to achieve a specific objective,
whether it be navigating to the Moon, the landing itself, or the rendezvous after the
lunar module ascends from the surface.

Developing the mission software to solve these problem is a huge undertaking.
Ideally, mission programmers want to concern themselves with the nuances of the problem, and not with the idiosyncrasies of the underlying system. In these cases, a high level language would be preferable, but their relatively inefficient code and excessive memory requirements would be impractical in the AGC. Despite the desire to remove developers from the nagging details of the machine, the AGC’s resource constraints are so severe that only hand-coded and optimized programs are practical. The obvious need to extend the AGC architecture and the inability to do so, forced the designers to consider a radical departure from the evolutionary steps used up to this time. If extending the existing hardware is impractical, the alternative is to invent an entirely new system architecture and implement it in software. Freed of (most of) the limitations of the underlying AGC hardware, a new architecture can implement features to ease the development of mission software. The Interpreter, as this new system architecture is named, supports several important features:

- Single, double and triple precision variables
- Complex datatypes, such as vectors and matrices
- Vector operations, dot and cross product, and shifts
- Transcendental functions, such as square root and trigonometric functions
- Two index registers, plus index increment registers
- A large stack area and stack pointer
- Simplified addressing – banking registers are not explicitly used.

The Interpreter complements rather than replaces basic AGC coding, as not all the mission software programming requires the capabilities of the Interpreter. Additionally, mission programs remain wholly dependent on Executive routines for process management, I/O, interrupts and other system level functions. As such, while the Interpreter has a number of features in common with a hardware CPU, it is less a machine emulator than it is an extension of the programming environment. Programs may begin with basic AGC instructions and then enter the Interpreter to evaluate a complex expression, only to return to executing basic instructions when the calculation is complete. Fundamentally, the role of the Interpreter is to extend the AGC’s architecture using software written in the AGC’s native code. Elements such as specific registers or status bits that are not found in the physical machine are implemented in erasable storage, using software logic to define the rules of their operation. In the AGC’s interpretive system, the goal of a rich and comprehensive instruction set and the desire to hide details of memory banking from the user forces an unpleasant reality. With only a 15-bit word available, interpretive instructions and their operands cannot fit into a single word. The Interpreter’s instruction set must be extended into a second word to hold all the necessary bits. However, by breaking through the psychological barrier of “one word equals one instruction”, a wide range of possibilities present themselves.

Two issues deserve immediate attention when designing an interpretive environment: the number of operation codes and the characteristics of memory addressing. The experiences with the basic AGC instruction set and its three bit
opcode proved that an interpretive instruction must have sufficient bits to represent the entire instruction set without resorting to elaborate tricks in order to extend an inadequate design. But how many bits are truly necessary? Obviously, three bits for an opcode is unacceptably small, five bits is only marginal, but using the entire 15-bit word (for more than 32,000 unique instructions) is absurd. At the same time, using only a fraction of the word for the opcode is wasteful, presenting the question of what to do with the remaining bits. Addressing in the AGC is limited to 15 bits, so using the leftover bits in the opcode word for addressing does not provide any additional capability. The solution for maximizing the usage of the interpretive instruction word has profound consequences for writing and storing a program in memory. Choosing seven bits for the length of the interpretive opcode provides a generous 128 unique instructions, and also allows two opcodes to fit comfortably within a single word. By squeezing two opcodes into the same word, we create one of the unique properties of the interpretive programming system. Unlike the basic AGC instruction set, instructions for the Interpreter are not in a single, fixed format, nor are the operands necessarily contiguous with their opcode. After the word that holds the two interpretive operation codes, operands for the first opcode are specified, followed by those for the second opcode. This format is complex, and neither the interpretive language nor the assembler provide the programmer with any coding assistance. Each word of the interpretive program is coded individually, with the programmer solely responsible for operand placement.

![Figure 50: Interpreter half-memories](image-url)
The Interpreter's address word uses only 14 bits to reference the location of the data, leaving bit 15 to define which (if any) index register is used. However, 14 bits can reference only 16K of storage, which is far less than even the AGC's limited memory. Circumventing this serious restriction of storage access requires an adjustment to our view of memory management in the Interpreter. By introducing the concept of "half memories" to an interpretive program we can view fixed storage as two independent 16K areas.\textsuperscript{15} However, not all of the 16K is fully used, nor are the half-memories contiguous. All references in half-memories must be in switched fixed storage, and banks 00\textsubscript{8} through 03\textsubscript{8} are inaccessible. The first half-memory occupies fixed storage banks 04\textsubscript{8} through 17\textsubscript{8}, giving 12K of addressable storage in the first half-memory. The second half-memory uses fixed banks 21\textsubscript{8} through 37\textsubscript{8}, for a total of 15K words, with bank 20\textsubscript{8} unused by the Interpreter.

The instruction format of the Interpreter

The structure of the interpretive instruction set is quite different from the basic instructions of the AGC. Up to this point, software was converted to machine instructions using a single, self-contained instruction for each line of our source code program. Interpretive instructions depart from this convention by taking one or two instructions and coding them over several lines of source, which translates into several words of storage. Instructions begin with the Interpretive Instruction Word (IIW), which contains one or two operation codes. Each opcode is analogous to an AGC basic instruction code, and may include arithmetic, logical and branching operations. Source code instructions are written from left to right, reflecting the sequence in which they should execute, but the first opcode is stored in the lower seven bits of the IIW. The second opcode, if it exists, is placed in the upper half of the word, in bits 14 through 8.

Each opcode of the Interpretative Instruction Word requires zero, one or two Interpretive Address Words (IAW). An address word is exactly that – the address of an operand required by an interpretive instruction. The number of IAWs is determined by the operations in the instruction word. An operation may not require an operand, such as a trigonometric function that acts only on the accumulator contents, or a "vacuous" operand on the Interpreter's push down stack. In both cases, there will not be an IAW for this operation following the instruction word. Other operations can require one or two IAWs, depending on the interpretive instruction. Operands for the first interpretive opcode immediately follow the instruction word, and any IAWs for the second opcode follow next in storage. Figure 51 shows a representative layout of an IIW, followed by IAWs.

\textsuperscript{15} Breaking fixed storage into half-memories might be properly thought of as a memory banking technique, not unlike how fixed and erasable storage is managed. While this observation is correct, the literature never uses the word "banking" to describe half-memories, so we will refrain from using this term as well.
### Interpretive opcode encoding and addressing

The encoding of the Interpretive Instruction Word is complex, and is driven partially by the need to differentiate itself from the Interpretive Address Word. Interpretive opcodes are located in bits 14 through 1 of the instruction word, with bit 15 equal to zero. When placed in storage, the entire word is complemented to create a negative value, readily identifiable from normally positive Instruction Address Words. Complementing the IIW becomes the first step in the instruction fetch process. The first operation code (called OP1) is in bits 7 through 1 of the word, and is isolated by masking out the higher order bits. Subtracting one from the opcode leaves us with the final value for the interpretive opcode. A different process extracts the second operation in the instruction word, OP2, which is in bits 14 through 8. Repetitive shifting of the IIW to move these bits to the low end of the word is possible, but this is a time consuming process. A special hardware assist is available by saving the IIW into storage location 000238, the Edit Opcode (EDOP) special register. When data is saved in the EDOP register, it is automatically shifted right by seven bits, filling the higher order bits with zeros. After retrieving the operation code from the EDOP register and subtracting one, the second interpretive opcode is ready for decoding and execution.

Interpretive Address Words are coupled with an interpretive operation code and define the address for its operands. In most cases, IAWs use only addresses as

---

16 There is one exception to this case. The address is complemented, and bit 15 in an IAW is set to one when indicating that the X2 index register is used.
operands, pointing to data in either erasable or fixed memory. The few special addressing cases are notable, because they create useful extensions to the Interpreter. Immediate operands, where the operand itself is the data, are available in the cases of shifting and index register manipulation. Indirect addressing, when the operand points to a word in storage that contains yet another address at which to save the data, is possible only when storing the contents of the Multipurpose Accumulator. Indexing using one of the interpretive index registers, X1 and X2, is a welcome addition to the AGC repertoire, and replaces the powerful but awkward INDEX instruction. Indexed addresses are possible only with special operation codes, where the choice of index register is embedded within the IAW. In cases where the first interpretive opcode does not require an operand, coding IAWs for the second opcode is no problem, because the Interpreter has no difficulty in determining what operation code the IAWs are associated with. However, the case where the first operand intends to use the stack as an operand (a vacuous address) can present a problem. If the second operand requires an operand, then the Interpreter requires a means to determine which instruction the IAW is associated with. The solution is through the STADR opcode, which is discussed below.

Programmers must exercise caution when coding the address words, as they do not contain identifying fields that assign the IAW to a specific opcode. No sanity checking is performed during the source code assembly or interpretive execution to assure that the IAWs are correctly associated with the operation codes. A forgotten IAW, for example, will not prevent the Interpreter from fetching a word that “should” be the intended operand for the other operation code in the IIW. Address words are not exclusively for data addresses. Branching and other transfer-of-control instructions use IAWs to specify the destination of the next instruction to execute.

Together, instruction and address words are termed an “interpretive string”. This is a more accurate characterization than referring to an interpretive “program”, since the “string” is not executable code in the traditional sense. The Interpreter itself is the executable unit of work in the AGC when a string is processed, and the string is simply its “data”. This might appear arcane, but the difference becomes relevant when following the execution of a job in the AGC. Special registers containing addresses within the executing program (such as Z or Q) will never reference any location in the string, only in the Interpreter.

Breakdown of the instruction classes
The Interpreter presents a rich set of well over 100 instructions, plus a special “no operation” code. Of the seven bits available, the last two bits define important characteristics of an instruction and also group them into four distinct addressing schemes.
### Addressing Class Characteristic

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Unary instructions, which do not require an operand. Operations include transcendental functions and short shifts. Unary instructions should not be confused with operations using vacuous addressing.</td>
</tr>
<tr>
<td>01</td>
<td>Arithmetic operations using a non-indexed address to reference storage. Addressing is limited to only local erasable or the half-memory from which the instruction was fetched.</td>
</tr>
<tr>
<td>10</td>
<td>Branching and index manipulation instructions. Branching instructions may use all of the interpretive storage areas (both half-memories). Addressing in index manipulation instructions can reference any area in interpretive storage, and may also use the address word as an immediate operand.</td>
</tr>
<tr>
<td>11</td>
<td>Arithmetic instructions that use index registers X1 or X2 modify the operand address word. Values in the index register are subtracted from the operand address, complementing the looping technique seen in the Count, Compare and Skip basic instruction. Only storage in erasable storage or the interpretive half-memory is accessible.</td>
</tr>
</tbody>
</table>

While a maximum of 128 opcodes appears more than sufficient to provide a wide variety of operations, closer inspection reveals that many instructions are simply variations of a single operation. As an example, consider the four types of short shift – the left and right forms of shift, and shift and round. For each type, there are four distinct opcodes for shifting one, two, three or four bits. Hence, 16 opcodes are dedicated to the short shifts. As the opcode's 2-bit suffix differentiates between indexed and non-indexed operations, two opcodes are necessary for the instruction to use both types of addressing. Organized according to function, the interpretive instruction set is broken down into the following groups:

- Store, load and push-down
- Scalar and arithmetic
- Vector arithmetic
- Transcendental functions
- Vector functions
- Shifts; short shifts, general and normalized
- Branching, sequence generation and subroutine linkage
- Switch instructions
- Switch test and branch
- Index register manipulation
- Miscellaneous instructions.

This breakdown is necessarily broad, and instruction assignments often fall into a number of categories. Vector operations are especially problematic. Shifting the components of a vector is a good example: should it be categorized as a vector or a shift operation? In this case, the instruction is assigned to the shift operations, as the operation (shifting) is not unique to the vector datatype. Other operations are
equally ambiguous, and this book will defer to the categories established in the primary AGC documentation.

The Multipurpose Accumulator and the datatypes mode of the Interpreter

Arithmetic and logical operations in the basic AGC hardware are performed in the accumulator, located in the lowest part of addressable memory. The wide variety of datatypes used in the Interpreter make the existing hardware accumulator woefully insufficient. Without adequate hardware support, software becomes the only means to create an accumulator that will handle a variety of datatypes ranging from single precision to vectors. Earlier, we defined the Multipurpose Accumulator (MPAC) as a seven word area in a job’s Core Set. For jobs that do not invoke the Interpreter, these words are available for general use by the program. No formal structure is defined for the MPAC in non-interpretive programs, and all of the storage may be used as the programmer sees fit. When used during interpretive string execution, the MPAC assumes a strictly defined format that is under the control of the Interpreter itself. The Interpreter also establishes the concept of the data “mode” for instruction execution, representing the datatype first loaded into the MPAC at the start of the interpretive string. These datatypes are defined as double or triple precision scalar quantities, or as three-element double-precision vectors. Interpreter programs can also operate with single-precision scalar quantities and with matrices (3 by 3, double-precision elements), but such operations do not establish a new datatype mode. During the execution of the interpretive string, the mode changes only when a different datatype is loaded into the MPAC, or when the mode is explicitly changed during a mathematical operation.

As the MPAC and its data mode are determined by software, not hardware, a highly flexible architecture is possible. Consider the basic AGC hardware, where the primary datatype is the single-word integer. All hardware elements, such as the special registers, I/O channels and the most basic instructions, work on this 15-bit structure. Most mathematical operations using double precision data require programmers using basic instructions to manipulate both parts of the two-word quantity. To provide native support of the full range of datatypes, the Interpreter uses the same basic AGC operations but hides the implementation details, allowing the developer to concentrate more on the program requirements.

Operations such as pushing data onto the stack or storing data in memory are not tied to a specific datatype. Taking its cue from the datatype “mode”, a PUSH instruction will place a double precision, triple precision or vector onto the stack. The notion that the Interpreter supports several datatypes is significant, but a large part of its power and flexibility lies in its ability to adjust its processing based on the type of data being manipulated.

---

17 To be sure, a few instructions do operate on two consecutive words at once, such as Double Clear and Add (DCA)
Indexing in the Interpreter

Indexed addressing in the Interpreter overcomes a key limitation in the basic AGC architecture; specifically, the lack of a hardware index register. While the AGC’s basic instruction INDEX has a novel method of addressing elements in an array, the Interpreter’s indexing operations are much more conventional. Many of the arithmetic instructions (both scalar and vector) can use indexing to reference their operands, as can the general shift instructions. By reserving the last two bits in the operands to indicate that the interpretive instruction will use indexed operands, the Interpreter now knows to look within the operand for information on which index register to use. Index registers X1 and X2 are available, and either may be used wherever indexing is possible. Two “step” registers, S1 and S2, are paired with the index registers, and are used by the Transfer on Index instruction for loop control. Unlike other registers in the Interpreter, the index and step registers cannot operate with fractional data. For an index register to modify an address, or for the step register to decrement its index, only single-precision integer data is allowed.

Loading data into an index register can take several forms, depending on the location of the data. In cases where the data to initialize the index is well defined and unchanging, Address to Index True (AXT)\(^1\) uses the operand as immediate data and not as an address. In less well defined situations where the initial index value is not known until execution time, Load Index From Address (LXA) uses its operand to point to the data in storage. Two instructions allow us to preserve the contents of the index registers. Store Index into Erasable (SXA) copies the contents of the register into erasable storage. A two-way exchange operation is also available. Exchange Index with Erasable (XCHX) takes the contents of the index register and swaps it

\(^1\) In fact, the AXT instruction actually has two distinct opcodes, AXT,1 and AXT,2.
with the word in erasable storage. Arithmetic functions are limited to two addition operations and a subtraction. Adding the contents of a fixed or erasable storage word to the index register is through Index Register Add (XAD). Index Register Subtract (XSU) operates similarly, taking the value in storage and subtracting it from the index register. Finally, Increment Index (INCR) also adds a quantity to the index register, but interprets the IAW as an immediate operand. All overflowing index operations leave their overflow corrected result in the index register and never in the operand in storage.

Unlike index registers, step registers do not have dedicated instructions available to manipulate their contents. The roles of step registers are narrowly defined, as they are a static value used only with the TIX instruction for decrementing an index. Given the limited scope of the step registers, there is little need for a rich set of instructions for moving and manipulating data within them. While the Interpreter has a comprehensive set of instructions for loading, storing, and manipulating the index register contents, it does not extend this functionality to the step register. One instruction, Set Single Precision (SSP), is useful for initializing step registers. The erasable storage location specified by the first operand is loaded with the contents of the second operand. Saving the contents of a step register is rarely necessary, and using basic AGC instructions is the only practical option available for saving S1 and S2.

During the time that indexing is not used during the processing of the interpretive string, both the index and step registers are available for holding temporary data. In addition to their indexing function, index registers have capabilities similar to an accumulator. Loading and storing data, addition and subtraction, plus exchange functions are available, allowing limited arithmetic capabilities in parallel with the MPAC.

For an instruction to use an index register, additional notations must be added to the opcode and its operand. First, the opcode notation is changed with the addition of a "$^*$" as a suffix to the opcode’s name. A Double Add instruction, DAD, is now coded as DAD* when indexing of its operand is required, and the last two bits in the seven bit operation code are changed to 11 from 01. It is important to note that the basic operation of Double Add is unchanged, and only the process of obtaining the operand is different. However, this notation is not yet sufficient for completely defining an indexed instruction. When the DAD* instruction is decoded by the Interpreter, all that is known at this point is that indexing is requested. Given the limited number of bits available even in the interpretive opcodes, there are no available bits to specify which of the two index registers is to be used. The operand must specify which index register to use by placing a comma after the operand address, followed by the register number. The indexed Double Add instruction now looks like this:

DAD*       Double Add the indexed table entry
ADDTABLE,1 Address of table, using X1 for indexing

Note that the Interpreter does not know of the choice of index register until it fetches the IAW from storage. Saving the index register specification in the operand
is feasible only because 14 bits are used for the address and bit 15 of the IAW is
reserved to indicate which index register is used. Index register X1 is specified when
bit 15 of the IAW is set to zero. Setting bit 15 to one indicates X2 is desired, but has
additional consequences. Since Bit 15 is also the sign bit, the entire quantity is seen as
a negative value. To simplify the decoding process after fetching, the entire IAW is
saved in a complemented form when the X2 register is requested.

The notation for interpretive instructions that perform operations directly on the
index registers is similar to that used for operation codes. Dedicating an operand
word to the single bit required to specify the index register is wasteful, and the
relative abundance of interpretive opcodes permits using a unique opcode for each
index register. The opcode explicitly identifies which index register to use and only
one operand is necessary to complete the instruction. The resulting notation for
instructions operating on index registers is similar to that for indexed operands. A
comma follows the name of the instruction, which is then followed by the index
register number. Not surprisingly, operations that manipulate the index register are
themselves not indexable.

\[ \text{SXA,2} \quad \text{Store index X2 into erasable memory} \]
\[ \text{X2SAVE} \quad \text{Address of Index X2 save area} \]

**Figure 53:** Interpretive instruction directly operating on an index register

The most familiar datatype used with indexing is a table, often called an array. A
table is a set of related data organized in a list format, occupying contiguous memory
locations. A familiar example is the AGC’s star catalog, containing the celestial
coordinates of the 37 stars used for navigational fixes. Each star has its position in
space defined by a vector, making each entry in the table six words long. The need to
examine tables in memory is so prevalent that the final indexing instruction, Transfer
on Index (TIX), combines modifying the index register, testing its value, and
branching based on its contents. Arguably one of the more sophisticated interpretive
instructions, it replaces several frequently used programming sequences. More than
simply an assist to indexing, TIX provides capabilities found in the FOR and DO
loop constructs of high level languages such as C and Java. In high level languages,
looping instructions define a block of code that it executed multiple times, until a
particular ending condition is reached. Typically, an index or other variable is
modified each time through the loop and is tested to determine whether the ending
condition has been reached.

The first task in using the TIX instruction is initializing a step and index register
pair; either X1 and S1, or X2 and S2. The index is loaded with the length of the table
and the step register with the amount by which the index is decremented. After
loading the registers, the body of the loop is entered, where indexed instructions that
reference the table are found. The TIX instruction is located at the bottom of the
loop, and compares the value in the index register with the contents of the step
register. TIX subtracts the value in the step register from the index register, which
now points the index to the next element in the table. The index and step registers are compared, and if the index register is greater than the step register, TIX transfers control to the top of the loop. Only when the index register is less than or equal to the contents of the step register will TIX not perform any operation, and thereby exit the loop to resume with the next operation.

Using the star catalog as an example, assume that the table begins at location 01000₈. A program wishing to reference each entry in the catalog starts with the table reference pointing to the end of the table, and a zero value in the index register. As each entry in the table is a six-word vector, a six is placed in the step register. References to the table use the ending, rather than the starting address, owing to the fact the value in the index register is subtracted from the operand address. The example in Figure 54 uses an indexed VLOAD instruction to bring a table entry into the MPAC, with its operand pointing to the end of the star catalog table located at 00336₈ + 01000₈ = 01336₈. The first reference takes the operand address of 01336₈ (the end of the table) and subtracts 00336₈, the contents of the index register, to point at the start of the star table at location 01000₈. After the processing of this first star entry is complete, the loop continues to the TIX instruction. TIX subtracts the six found in the step register from the index register, leaving 00330₈ in the index. Repeating this sequence for each entry, the loop terminates when the index reaches zero.

It is often useful to have instructions that negate, or bitwise complement, data as it is loaded into the index register. The complementing variations of AXT and LXA are Address to Index Complemented (AXC) and Load Index from Erasable Complemented (LXC). As with the AXT instruction, AXC uses an immediate

```
SSP AIX,1
S1
DEC 6
DEC 222 LENGTH OF STAR TABLE
IS 336 OCTAL
LOOP VLOAD* LOAD STAR VECTOR INTO MPAC
STARTAB+336,1
TIX,1
LOOP
ENDLOOP
STARTAB 2DEC +.8342971408 X COMPONENT OF STAR 37
2DEC -.2392481515 Y COMPONENT OF STAR 37
2DEC -.4966976975 Z COMPONENT OF STAR 37
2DEC +.8748658918 X COMPONENT OF STAR 1
2DEC +.0260879174 X COMPONENT OF STAR 1
2DEC +.4836621670 X COMPONENT OF STAR 1
STARTAB+336
```

Figure 54: Indexing into the star catalog table
operand. LXC follows from LXA, and its operand references a location in erasable storage.

**Right- and left-handed instructions**

Some interpretive instructions are designated as “right-hand only”, meaning they must be either the only operation in the instruction word, or the second of two present. In both cases, the instruction is coded on the “right hand” side of the two possible instruction positions in an interpretive statement. This requirement reflects the condensed nature of the interpreted code, where two operations occupy a single instruction word. Addressing storage locations in the AGC is limited to word boundaries, and there is no mechanism in the interpretive language to indicate a branch to the “left” or “right” operation. Thus it is impossible to transfer control to an operation in the right-hand coding position. This becomes an issue when branching instructions are used. In Figure 55 an unconditional branch (GOTO) is coded in the “left hand” position, followed by a short shift to the right (SR1) in the right-hand position. SR1 will never execute since the programming flow will never reach that instruction. Ignoring the constraints of “left hand” and “right hand” for instructions creates a conflict between what appears to be coded, and the actual software that will run.

```
GOTO   SR1    # GOTO, then Shift Right 1
NEWLOCAT # Destination address of GOTO
```

*Figure 55: Inaccessible right-hand instruction*

In the case of a GOTO instruction, two coding options are available to ensure that it is in the right-hand position. As the only operation on a single line, GOTO defaults to the right-hand position (Figure 56). When GOTO is the second of two interpretive instructions it resides in the right-hand position, following the DLOAD instruction (Figure 57).

```
GOTO   GOTO IS IN RIGHT-HAND POSITION
NEWLOCAT DESTINATION FOR GOTO
```

*Figure 56: One instruction, GOTO in right-hand position*

```
DLOAD   GOTO   GOTO IS IN RIGHT-HAND POSITION
NEWDATA   DATA ADDRESS FOR DLOAD
NEWLOCAT DESTINATION FOR GOTO
```

*Figure 57: Two instructions, GOTO in right-hand position*
In cases where the right-hand opcode is the only instruction coded in the Interpreter Instruction Word, the Interpreter will fetch and attempt to execute the non-existent left-hand operation. Leaving the left-hand operation undefined is not a problem, but it does create the need to introduce a null operation code. A null opcode is fetched and decoded like all other instructions, but no operation is performed when the null code is “executed”. This null instruction, often called a NOOP (NO OPeration), is placed in the operation code field by the assembler. It is important to note that the NOOP code is implied, and never written explicitly; nor does it have a formal opcode name associated with it. NOOP does not modify data, change the state of the Interpreter or alter the flow of control in the interpretive string.

Overflow processing in interpretive code
Overflows are possible during the execution of an interpretive string, and are handled somewhat differently than in the basic AGC instruction set. When an overflow (or underflow) occurs, the instruction sequence does not change, nor are interrupts inhibited. Rather, an overflow corrected result is saved in the MPAC or the storage location specified by the Interpretive Address Word, and the overflow flag OVFINd is set. Testing the overflow flag is now the duty of the programmer, as the Interpreter will continue processing despite the existence of the overflow condition. Instructions to test the OVFINd flag, and branching to another location exist to handle these cases. If no test is made, the condition will remain undetected and processing will proceed with invalid data.

Vector arithmetic
Vectors are contrasted with scalar quantities. Scalar data is only a singular value, such as temperature or energy. Vectors, in the context of physical objects, have both magnitude and direction. An automobile may be moving through two dimensional space, and have a component of motion in both dimensions. Combined, a resultant vector defines the overall motion of the car. In our three dimensional universe, we have adopted vector notation to describe the position, motion and orientation of a body. The three mutually perpendicular axes which form a vector are assigned names, which by convention are x, y and z. Starting from a given point, say a specific location on the Earth, we will also define an orientation for our vector’s frame of reference. Such assignments are often arbitrary, so let’s specify that the X-axis points straight up from the center of the Earth, perpendicular to the surface; the Y-axis points north; and to complete the triad, the Z-axis points east (Figure 58).

A practical example might begin with a reference point at Tower Bridge, London. An airplane is flying directly over the bridge, heading towards New York along the Great Circle route. This route initially heads slightly north of west on a heading of 287 degrees true. Thus, its position can be defined as 35,000 feet on the X-axis (height above the bridge), and zero feet north and east as the plane is directly over the bridge. Its velocity of 600 miles/hour, relative to the bridge, is zero miles/hour in the X-axis (indicating level flight), 574 miles/hour in the negative east direction (west) and 175 miles/hour northwards. From these individual components, the magnitude of the vector by applying the well-known Pythagorean theorem is:
Figure 58: Orientation of vector space on the surface of the Earth.

\[ c = \sqrt{a^2 + b^2} \]

In a triangle, the hypotenuse is thought of as the magnitude of a two-component vector. Extending this idea to a three-dimensional vector create three components \( x \), \( y \) and \( z \), with the magnitude \( M \) calculated as:

\[ M = \sqrt{x^2 + y^2 + z^2} \]

To appreciate using arithmetic operations that involve vectors requires introducing new mathematical principles that extend variables into possessing more than one component. Of the sixteen vector operations available in the AGC interpretive environment, basic operations such as add and subtract are conceptually similar to their scalar counterparts. Other operations, such as dot and cross product are unique to vectors. Depending on the operation, either a scalar or vector operand is used in operations with a vector, and a scalar or vector quantity will be the result.

In the Interpreter, vector datatypes are composed of three double-precision fractional quantities. All vector functions require loading the vector into the Multipurpose Accumulator, the seven-word area located in a job’s Core Set. However, the six words of the vector are not loaded into contiguous locations in MPAC storage. The first component of the vector, the \( x \)-component, occupies locations 0 and 1 of the Core Set. The second, the \( y \)-component, is stored in locations 3 and 4, and the \( z \)-component is in 5 and 6. Location 2 of the MPAC follows locations 0 and 1, creating a three-word area for triple precision scalar numbers. As there is no requirement for the vector components to be loaded into adjacent storage areas, there is no harm in separating the \( y \) and \( z \) components from the area used for triple precision data.

The description of the arithmetic vector operations begins with moving the
vector into the MPAC. VLOAD places the six words of the vector quantity into locations 0, 1, 3, 4, 5 and 6 of the MPAC, and sets the MPAC mode to vector. Two vectors are added or subtracted using the VAD or VSU interpretive operations. In these operations, each of the three components is added (or subtracted) from its corresponding component in the MPAC. Thus, with a vector \( V_{\text{MPAC}} = (0.1, 0.3, 0.7) \) in the MPAC, adding the vector \( V = (0.2, 0.5, -0.2) \) will produce \( V_{\text{MPAC}} = (0.3, 0.8, 0.5) \). If an overflow occurs, the OVFIND flag will be set and the overflow-corrected result placed into the MPAC. Vector subtraction, like its scalar equivalent, is not a commutative operation. Unlike addition, where \( A + B \) produces the same result as \( B + A \), \( A - B \) produces a different value than \( B - A \). VSU has the same dilemma when taking the contents of the MPAC and subtracting it from a vector in storage. One solution would be to temporarily move the vector from the MPAC to a location in storage or the pushdown stack, load the other vector into the MPAC, perform the subtraction, and then save the result. All of these steps are eliminated through the “Backwards” VSU instruction (BVSU). In a rare example where erasable storage is the target of an arithmetic operation, this instruction subtracts the contents of the MPAC from the vector in storage, leaving the contents of the MPAC unchanged.

In the Interpreter, a division operation on a vector is defined as the division of each of the vector components by a scalar quantity. Starting with a vector \( V = (0.3, 0.5, 0.1) \), division by 2 gives the result \( V = (0.15, 0.25, 0.05) \). The interpretive operation Vector Divided by Scalar (V/SC)\(^{19} \) performs the scalar division of the vector stored in the MPAC, where it also leaves the result. As the quotient is a fractional value with several digits of precision, a remainder is unnecessary and is not generated.

Multiplication of a vector is realized in a number of ways, depending on whether it is multiplied by a scalar quantity or another vector. Multiplying a vector by a scalar quantity in the Vector Multiply by Scalar (VXSC) instruction is similar to the V/SC vector division. Each vector component is multiplied independently by the scalar quantity, with the result remaining in the MPAC. For example, a vector \( V = (0.3, 0.1, 0.8) \) multiplied by 0.4 leaves \( V = (0.12, 0.04, 0.32) \) in the MPAC.

**Vector operations: dot and cross product**

Additionally, vector algebra provides us with two new multiplication operations: the dot, or scalar product, and the cross product. These new operations move from the comfort of intuitive operations such as adding or dividing vector components, and introduce new mathematical concepts. Rather than simply changing the magnitude of a vector, dot product yields a scalar, and cross product creates an entirely new vector. Calculating a dot product starts with two vectors, each with components \( x, y \) and \( z \). First, the two \( x \)-components are multiplied together, then the \( y \)-components, and finally the \( z \)-components. Adding the results completes the dot product, which is a scalar value. Mathematically, we can express this as:

\[ V \cdot W = x_1y_1 + x_2y_2 + x_3y_3 \]

\[ V \times W = (y_2z_3 - y_3z_2, z_1x_2 - z_2x_1, x_1y_3 - x_3y_1) \]

\(^{19}\) Note that a slash “/” is a valid character in the name of the instruction code.
\[ V_1 \cdot V_2 = (V_{1x} \cdot V_{2x}) + (V_{1y} \cdot V_{2y}) + (V_{1z} \cdot V_{2z}) \]

As an example, assume there are two vectors, \( V_1 = (3, 1, 5) \) and \( V_2 = (2, 7, 4) \). Multiplying together the corresponding components and adding their products gives the following result:

\[
(3 \times 2) + (1 \times 7) + (5 \times 4) = 33
\]

Dot products facilitate computing the amount of work performed, where work is defined as the application of a force over a particular distance. In many cases, the forces are not acting parallel to the direction of motion of the object.

A second multiplication operation between two vectors is called a cross product. Vector cross products take two three-component vectors and create a third vector that is perpendicular to the other two. Cross products are frequently used to compute torques and other solutions involving rotating bodies. Software in the AGC also uses cross products in areas as diverse as calculating attitude maneuvers and guidance computations.

There are situations where operations on vectors require that the magnitude of the vector is exactly one, or unit length. Most vector operations involving trigonometric functions require unit vectors in their calculations, making it necessary to convert many vectors from their original scaling. Any vector can be converted to its unit form by dividing each component by the vector’s magnitude. Assume there is a two-component vector whose x and y components are 2 and 3 units long, giving a magnitude equal to \( \sqrt{13} \). Dividing the components by the vector’s magnitude gives us \( x = \frac{2}{\sqrt{13}} \), and \( y = \frac{3}{\sqrt{13}} \), resulting in the unit vector. Note that the ratio of the two components is unchanged, so the direction of the vector remains the same. The UNIT interpretive operation converts a vector in the MPAC to a unit vector, and leaves the result in the MPAC.

**Complements, squares and absolute values of a vector**

Each component of a vector is important because it provides directional information, but many applications require only the magnitude of the vector. Two related functions provide the length of a vector, and its square. Because the magnitude is a sum of squares, it maintains the essential property of an absolute value; that is, it is never negative. Vector Absolute Value (ABVAL) and Vector Magnitude Squared (VSQ) operate in a similar manner. Both take the vector components of the MPAC, squaring each component and adding them together. When requesting a VSQ operation, this result is placed into the MPAC. Otherwise, if the programmer requests an ABVAL operation, the square root of the sum is placed in the MPAC. Both operations set the mode to double precision.

In the same manner as negating a scalar quantity, such as converting a 7 to -7, it is also possible to complement a vector. By reversing the sign of each component, we reverse the direction of the vector without altering its magnitude. The unary Vector Complement (VCOMP) function takes the vector contents of the MPAC and complements the signs, leaving the result in the MPAC.
Defining a vector
A number of important vector operations begin not with a vector, but with a trio of individually computed double precision values that must be reinterpreted as a vector datatype to be consistent with later mathematical operations. The Vector Define (VDEF) instruction begins with the first component of the vector, $V_x$, already in the MPAC. The remaining two vectors components $V_y$ and $V_z$ are popped off the pushdown stack to join $V_x$ in the MPAC. With all three components necessary for a vector now in place in the MPAC, the mode can now be set to vector.

Matrix multiplication
Mathematical operators for guidance, navigation and control often extend beyond the relatively simple structures such as vectors. Frequently, matrices are necessary to represent the vehicle’s state, or in performing transformations from one frame of reference to another. This requirement makes it essential that the Interpreter provide at least basic matrix support. Given the need for simplicity and speed in the AGC, it might come as a surprise to see support for such a complex datatype. However, in view of their widespread use throughout all areas of flight software, the Interpreter would be seriously limited without the ability to perform these operations.

Multiplication of matrices is not especially complex, but does require a number of repetitive steps. Multiplication operations are restricted by the sizes (rows and columns) of the two matrices. For matrix A of size $m \times n$ to be multiplied by matrix B, B must be of the form $n \times p$, and the resulting matrix is sized $m \times p$. Stated in a more intuitive way, the number of columns in A must equal the number of rows in B. The formal expression for matrix multiplication is:

$$C_{ij} = \sum_{r=1}^{n} A_{ir} B_{rj}$$

where $n$ equals the number of rows in A and the number of columns in B, $i$ is the number of columns in A, and $j$ is the number of rows in B.

A practical example is if matrix A is $2 \times 3$ (two rows by three columns), matrix B must have three rows. If B is sized $3 \times 4$, then the matrix resulting from the multiplication will be $2 \times 4$. Graphically, our example becomes:

$$\begin{bmatrix} 1 & 5 & 4 \\ 2 & 1 & 3 \end{bmatrix} \times \begin{bmatrix} 3 & 9 & 2 \\ 2 & 8 & 5 \\ 4 & 1 & 8 \end{bmatrix} = \begin{bmatrix} 29 & 53 & 59 & 49 \\ 20 & 29 & 33 & 29 \end{bmatrix}$$

Matrices in the AGC are implemented using double precision variables. Although the Interpreter limits the size of matrices to $3 \times 3$, operations on larger matrices are possible by using elaborate programming techniques. Limited to only seven words,

20 The most notable exception is the error transition matrix, or W-matrix. Used by the Kalman filter in the navigation routines, the W-matrix may be sized as a $6 \times 6$ or $9 \times 9$ matrix.
The MPAC is too small to hold the eighteen words necessary for a $3 \times 3$ matrix. Since arithmetic operations require that one operand resides in the MPAC, the Interpreter simplifies the problem by requiring that one of the multipliers be a vector, which by definition, is also a $1 \times 3$ matrix. The most important side effect of these two restrictions is that the result of the multiplication will have a size of $1 \times 3$ or $3 \times 1$, which are both vectors! With a vector operand in the MPAC, the matrix resides only in storage, freeing the Interpreter from the need to load, store or operate on matrices in the MPAC. However, matrix multiplication is not commutative – meaning that given two matrices $A$ and $B$, multiplying $A$ times $B$ does not produce the same result as multiplying $B$ times $A$. Given the restriction of having only one operand, a vector, in the MPAC, non-commutative multiplication forces us to introduce two distinct multiplication operations, MXV and VX. Both instructions use the vector in the MPAC and the matrix in storage, and resolve the commutation issue by performing the multiplication in both directions. Matrix, Post-Multiplication by Vector (MXV) multiplies the matrix by the vector in the MPAC, leaving a vector in the MPAC. Matrix Pre-Multiplication by Vector (VXM) reverses the sequence by multiplying the vector by the matrix, again placing the vector result in the MPAC.

One function that the Interpreter does not implement is matrix division. Matrix division is a complex and computationally intensive operation, and in some cases a solution might not even exist. Indeed, it is usually far easier to eliminate the need for matrix division operations by restructuring the problem to avoid its use.

Flagwords
Mission software has the obvious need to adapt its processing as the different stages of the problem are computed, as well as recording the states of the onboard systems or maintaining the selections requested by the crew. A new datatype, the flagword, is used for this task. Flagwords, as their name implies, are a group of words reflecting the various states of the software, and are located contiguously in erasable storage. Each of the 15 bits of a given flagword defines a different parameter or indicator as a binary value. Limited to the yes/no, on/off nature of binary data, the flagword bits inform basic decision tests by the software. Several hundred flagword bits, also known as switches, are defined between the Command Module and Lunar Module. The CM software uses 12 flagwords, and the LM has 14 flagwords.

For example, a seemingly trivial but quite important example is the MOON-FLAG flagword bit in the Command Module software. When set to zero, it indicates that the spacecraft is in the Earth’s gravitational sphere of influence; when set to one, the Moon is the dominant gravitational body. Flagwords and their bits are vital for maintaining the logical state of mission programs. Only mission software uses flagwords for managing their states; the Executive and related routines use an independent set of words to manage their execution. Knowing the state of the software is so important that flagwords are downlinked continuously to the ground for diagnostic purposes.

If the use of a word containing bits that specify various spacecraft conditions
sounds familiar, it is likely because flagwords resemble the I/O channels discussed earlier. Channels in the AGC are the interface between the computer and the spacecraft hardware; one bit might indicate that the liftoff discrete has arrived, and another bit might command a thruster. In both cases, requests and information are communicated between the spacecraft and the computer through the channels. Conversely, the domain of flagword switches lies entirely within the AGC. While software will interpret the channel data, and generate commands based on it, data contained in the flagwords stays wholly within the computer.

Operations on switches are necessarily limited. With a 1-bit switch, the only possible operations are setting a bit to zero, setting it to one, or inverting whatever value is already in the switch. The implementation of flagwords also exposes an issue that arises in multiprogramming systems, and is particularly troublesome in the AGC. At any time during its execution, a program can reference or change a flagword. Unfortunately, coupled with this flexibility is a significant drawback; if there were not any means to serialize access to the flagword, two processes might attempt to access or update a particular bit at the same time. Even with the vastly increased capabilities of the interpretive system over the basic AGC instruction set, there is still no means to ensure that one process has exclusive access to the flagword. Specifically, the issue is that without a basic AGC instruction to operate on an individual bit, several instructions are needed to perform any bitwise operations. A window several milliseconds long results, in which the switch is unprotected from changes made by other processes. At any point during this window, an interrupt might be raised which in turn schedules other work that also needs to update one of the flagword switches. This creates the situation where two processes are trying to update the same location in memory, which certainly would result in untold problems. The only practical solution is to use the brute-force technique of inhibiting all interrupts during the time the flagword is processed. With interrupts disabled, it is impossible for another process to take control of the CPU and attempt an operation on a switch. After the Interpreter finishes the switch operation, interrupts are reenabled, thereby allowing other processes access to the flagwords.

The Interpreter provides a rich set of operations to set, clear and invert switches within the flagwords. There are also more complex operations, including testing switches and branching based on their state. Although the underlying operations in the switch instructions are logical AND, OR and XOR, the Interpreter allows manipulating only one bit at a time. This is in contrast to the AGC’s basic instruction set, which will perform logical operations on several bits at once within the word. This is not a significant limitation, as switch bits represent such diverse conditions that there is little need to operate on more than one bit at a time. Beginning with the most basic switch operations, SET, CLEAR and INVERT are the most intuitive. A one is placed in the switch by the SET instruction, a zero by the CLEAR, and the bit is flipped by INVERT. From these three basic instructions, we can derive the other eleven flagword switch operations. Recognizing the frequent need to set a switch and then follow that instruction immediately with a branch gives us a set of three more operations, SETGO,
CLRGO and INVGO. Here, an unconditional branch follows the operation on the flagword switch.

Two instructions complement those that operate on a bit and immediately branch. BON and BOFF do not alter the contents of the switch, but conditionally branch based on its settings. The two instructions test if a switch is set or cleared and if the test evaluates to “true” then the branch is taken. The remaining six instructions combine testing the switch, modifying the switch’s value, and then conditionally branch based on its original state. Using the example of BONSET and BOFSET, the original switch value is saved, and the bit is set to one. Next, the decision to branch is based on the original value of the switch. Given a switch set to one, the BONSET instruction tests the bit, obtains a “true” result, and performs the branch. With the switch bit cleared and set to zero, BONSET’s test evaluates to “false”, and the sequence of interpretive statements continues unchanged. Note that in both cases, the switch is set to one regardless of its original value. BOFSET operates in a similar manner, testing the switch for “off”. Other instructions follow this same structure. Rather than setting the switch and then conditionally branching, BONCLR and BOFCLR will clear the switch, and BONINV and BOFINV will toggle the switch’s contents to 1 from 0, or to 0 from 1.

With so many switches available to manipulate by the AGC software, there is the problem of how to efficiently address a large number of bits within a set of storage words. Only seven bits are available for an interpretive opcode, giving 128 possible combinations of values. At first glance, we might be able to develop a notation defining an opcode and the bit within the flagword it references. But the sheer number of individual switches in the Command Module and Lunar Module rules out encoding the switch number with the opcode. Accepting the necessity of dedicating an operand word that uniquely defines the switch bit within the flagword, the next task is to develop a format to efficiently reference individual switches. This is done in terms of the flagword number, and the switch number within the flagword. The 15 switches within the word can be specified using four bits. With fewer than 16 flagwords in the AGC software, uniquely identifying the specific word requires no more than four bits. From this encoding scheme, it is apparent that at least eight bits are all that is necessary for uniquely identifying an individual switch. At the same time, the number of switch manipulation instructions is worrisome. The fourteen operations consume over ten percent of all possible operation codes. An alternative is to define a single interpretive opcode for all switch operations, and qualify which of the fourteen possible operations is desired. Noting that the number of switches forces us to use an operand word, and that only eight bits of the operand are required to identify a specific switch, the six remaining operand bits are available to indicate the specific switch operation. The interpretive operation code is now common to all flagword manipulation instructions, and the specific operation (SET, CLRGO, BOFINV, etc) is defined within the operand. The final layout of the operand used is shown in Figure 59. Beginning with bits 4 through 1 of the operand, the number of the switch bit within the flagword is coded, and bits 8 through 5 define the unique switch operation code. Bit 15 is set to zero to assure a positive value for the operand, leaving six bits
available to define up to 64 flagwords. The CSM and LM software required no more than four bits to define all of their flagwords, but the switch operand layout allowed for many more if needed.

**Shifting data in storage**

An extensive library of shift instructions exists in the Interpreter, using two distinct implementations. Frequently, shift operations replace multiplication or division when one of the operands is a power of two. Shifting data is often necessary for scaling variables before further calculations, to avoid a loss of precision, or to isolate logical data in the word.

Short shifts, those which shift data one, two, three or four bits, are unary operators, with all the information encoded into a single 7-bit opcode. Short shifts of scalar values come in four variations: shifting left or right, and whether the result is rounded or not. Within each variant of short shift, we also have the number of bits to shift. All told, sixteen unique scalar shift operations are available. Shifts present the same problem of proliferating opcodes that flagword manipulation posed, but short shifts are designed to eliminate an operand from the instruction. As a unary operation, that is, without an operand reference, short shifts can only operate on quantities in the MPAC. In the description of the fields defined for short shift operations, it becomes apparent that nothing in the opcode states whether the shift is to operate on scalar or vector quantities. While the operation code does not differentiate between scalar and vector, a reliable means already exists to decide which operation to use. The MPAC’s mode, which consistently tracks the type of data last operated on, provides the cue. When set to double or triple precision, a scalar shift is selected, otherwise a vector shift is performed.

As unary operators must contain all relevant instruction information within the opcode, the short shifts necessarily require sixteen different opcodes. To ease the effort of parsing so many operations, each attribute of the shift is encoded in a dedicated field of the interpretive opcode. Beginning with the rightmost bits of the 7-bit opcode, we assign four bits for the direction of the shift. Moving left, we use one bit to define whether the shifted result is rounded or not, and the final two bits specify the number of bits to shift (Figure 60).

Short scalar shifts perform all operations in triple precision, shifting data through all three words of the MPAC regardless of the current mode. The use of such a long datatype demands that the programmer be vigilant about unwanted values in the MPAC. Particularly in the case of unused lower order words, the unexpected bits may become part of the shifted result. When shifting to the right,
zeros are placed in the highest order bits; conversely, zeros enter the lowest order bits of the triple precision value during a left shift. If the sign of the quantity changes during a left shift, an overflow condition is raised, setting the OVFIND flag. Rounding is an option available for all scalar shifts. This involves adding 0.5 to the least significant word and allowing the result to carry forward through the other two words. After the addition, the third and least significant word is set to zero, effectively producing a double precision result and leaving the MPAC mode unchanged.

Short vector shifts borrow many of their concepts from their scalar counterparts. As with short scalar shifts, the lack of an operand forces all operations to take place in the MPAC, and all three vector components are affected. Unlike the short scalar shifts, shifting is performed only on the double precision quantities found in a vector, as no space is available in the MPAC for triple precision vectors. Rounding is not an option available to short vector shifts because there is no third word to round with. This allows us to reuse the rounding bit. Combining the bit previously used for rounding with the two which specify the magnitude of a scalar shift provides a three bit field sufficient for specifying a vector shift from one to eight bits.

To shift a quantity more than four bits for a scalar or eight bits in the case of a vector requires a new set of instructions that use an operand word to contain all the shifting parameter data. Short shifts are necessarily restricted in the number of bits they can shift, but long shifts allow shifting up to the number of bits in the datatype. A double precision scalar value, for example, can accept shifts of 28 bits left or right. The operand word (Figure 61) contains four fields that define the characteristics of the shift operation. Bits 7 through 1 contain the number of bits to shift, and bit 8 defines a “pseudo sign bit” that is used to detect sign changes in indexed shifts. Bit 9 describes the direction of the shift, right or left, with the flag to specify whether rounding is desired in bit 10. Bits 15 through 11 are fixed, and are set to 01000. Note that the type of shift, scalar or vector, is not included in the fields of the operand word, as the datatype is again obtained from the MPAC mode. All long shift operations share the same opcode, plus one indexed variant, so long shifts also use the MPAC mode for specifying whether a scalar or vector shift is required. As in the
case of short scalar shifts, all operations on scalar quantities start with operating on triple precision values in the MPAC. Rounding operations are also the same as in short scalar shifts, where the triple precision value is rounded into a double word and the third word is set to zero.

**Scalar arithmetic**

Double precision quantities are the most prevalent datatype in the AGC’s Interpreter, and, as a result, few instructions operate on single or triple precision data directly. This survey of the scalar operations began with the four basic arithmetic operations: add, subtract, multiply and divide. The four instructions DAD, DSU, DMP and DDV are their interpretive equivalents. They take a double precision operand and perform the operation on the double precision quantity in the MPAC. Any overflow that might occur sets the OVFINDD overflow flag, and DAD, DSU and DMP leave an overflow corrected result in the MPAC. An overflow occurring with DDV leaves ±0.99999999 in the MPAC. Unlike basic AGC instructions, an interpretive instruction never inhibits interrupts when an overflow occurs. Many instructions that operate on a double precision value use the MPAC’s triple precision area, but these are not true triple precision operations because the MPAC mode remains double precision.

Multiplication and division require special handling, as their results often require a triple precision word to maintain an acceptable level of precision. Recalling that numeric data is represented using fractional notation, multiplying two fractional numbers together yields a smaller fractional value as a result. This situation is made worse if we start with a number that is small to begin with. However, much of the precision is retained by producing a triple precision result, even as the MPAC mode is held at double precision. After the multiplication or division operation is completed, the programmer can either strip off the unnecessary bits in the third word through the ROUND operation, or can recover the bits back into a double precision quantity through the NORM function. Using one of the shift instructions is also an option to recover bits from the third word of a triple precision result.

These four basic arithmetic operations must necessarily expand into a total of six interpretive instructions. All arithmetic operations would normally occur in the MPAC, but subtraction and division are not commutative operations; where A – B
does not equal B – A. Insisting that every operation leaves its result in the MPAC forces unnecessary complexity into the programming and wastes both storage and processing time. For example, without the ability to store the result of an operation in storage, subtracting the contents of the MPAC from a location in storage requires exchanging the MPAC and the data in storage to set up the proper arrangement of the two terms. The subtraction operation is performed, and then finally the terms are exchanged again. Such a cumbersome process is avoidable through the introduction of interpretive instructions that operate “backwards.” Here the calculation is performed against the term in erasable storage, and the MPAC contents remain unchanged. For backwards subtractions, the BDSU instruction is used. For dividing a value in erasable storage by the MPAC contents, BDDV is available. BDDV is unique, in that it does not behave the same as its “forward” counterpart, whose result is a triple precision quantity saved in the MPAC.

**Loading the MPAC**

Four instructions are available to load data directly into the MPAC. DLOAD, TLOAD and VLOAD place double precision, triple precision and double precision vector data into the MPAC, and, importantly, set the MPAC’s mode to that of the datatype loaded. Operands are obtained through direct and indirect addressing to storage, and through vacuous addressing by popping the data off the pushdown stack. The Interpreter architecture provides native support for the three datatypes mentioned above, plus limited support for matrices, but the ability to manipulate single precision values is notably absent. A large part of this rationale is that the Interpreter is designed to manipulate multiple precision and complex datatypes, and single precision is arguably most efficiently handled with basic AGC instructions. Still, a significant amount of data required by the interpretive routines is in a single precision format, and so the Interpreter provides a minimal amount of support for the datatype.

SLOAD is the fourth means of loading data into the MPAC. A single word in storage is placed in the first word of the MPAC using either a direct or indexed storage address. Using vacuous addressing to obtain the operand is not possible, as the pushdown stack does not support single precision datatypes. Since all other interpretive operators use only multiple precision or vector datatypes, the MPAC mode must be set to one of the supported datatypes, and double precision is used. When loading only a single word, the remaining two words of the MPAC’s triple precision area are cleared. Without setting the second and third words to zero, any data remaining from previous computations might be confused as part of the “double” precision value. Once the MPAC is properly cleaned up and the mode set, the single precision value is now ready for processing.

**Stack operations**

The Interpreter’s provision of the stack represents a significant improvement in function over the basic AGC hardware. Recalling the earlier description of program trees, a stack is a data structure that allows storing data in an orderly first-in, last-out sequence. Conceptually, the convention is to “push” data onto the stack to save it,
and each time data is retrieved from the stack it is “popped” off and removed from the stack entirely. The reality of the Interpreter’s implementation is rather different. The image of “pushing” and “popping” data creates the convenient illusion that each stack operation physically shuffles data back and forth, with the “top” of the stack at some arbitrarily fixed location in storage. In real-world implementations, the Interpreter included, data is fixed in storage, and reference to the “top” of the stack is through a variable called a stack pointer. Push operations append data onto the existing contents of the stack and update the stack pointer to the new “top” of the stack. “Popping” removes the data from the stack by updating the stack pointer to point to the data “beneath” the top entry in the stack. Normally, it is irrelevant whether the top entry is cleared as part of the popping process, since from the stack’s perspective, once the data is popped, the data is no longer accessible. However, the Interpreter does not clear the data from memory after popping the stack, leaving the data accessible to a cunning programmer.

In the Interpreter, the stack pointer is a variable named PUSHLOC which resides in the job’s Core Set. The stack itself occupies the first 38 words of the Vector Accumulator Area, with the “top” of the stack originating at offset zero. Pushing a double precision variable onto the stack, for example, places the data into contiguous locations in the stack area and increments PUSHLOC by two. When an interpretive instruction requests data from the stack, the data is passed to the opcode processing routines and the stack is popped by decrementing PUSHLOC by two. A powerful aspect of the Interpreter is the manner in which it pushes and pops datatypes. Taking its cues from the MPAC mode, a double or triple precision word or a double precision vector is pushed onto the stack. However, popping the stack through an interpretive instruction is driven by the mode appropriate to that instruction. For example, when a vector load instruction requests data from the stack (as a vacuous operand) the top six words on the stack are interpreted as a vector and loaded into the MPAC. Note that the Interpreter does not check to assure that the data in the stack entry was originally loaded as a vector, or that the mode of the instruction matches the mode of the MPAC. It is perfectly reasonable to push three consecutive and unrelated double precision values onto the pushdown stack, and use them later as an operand to a vector function. While this might be perceived as a useful “feature”, it also makes tracing errors extremely difficult.

With the interpretive system highly reliant on the pushdown stack, there is occasionally the need for a mechanism to manipulate its contents directly. The pushdown stack is never loaded directly from storage; rather, all data must first pass through the MPAC. The first of three stack manipulation instructions, named appropriately, PUSH, takes the contents of the MPAC and pushes it onto the top of the stack. Whether to push a double precision, triple precision or vector onto the stack is indicated by the mode of the MPAC, eliminating the need to specify the datatype. Once the data from the MPAC is pushed onto the stack, there is often the need to reload the MPAC with data for the next computation. Combining two instructions at once, Push Down and Load Double (PDDL) and Push Down and Load Vector (PDVL) first push the contents of the MPAC onto the stack and then reload the MPAC with either a double precision or vector
quantity, setting the MPAC mode to the datatype just loaded. As expected, the
source of the data for the MPAC may come from direct or indexed storage. With
the stack being loaded from the MPAC, there is now a curious dilemma: what is
the final result of executing PDDL or PDVL with a vacuous operand? This special
case of PDDL and PDVL is defined as a data exchange between the pushdown
stack and the MPAC. Notice that there is no requirement that the data pushed
onto the stack and that loaded from the stack be the same datatype. Using the
PDVL instruction as an example, assume that the MPAC mode is set to double
precision. PDVL will push that double precision quantity down onto the stack,
but not before the vector on the stack is popped off and saved in a temporary
location. Finally, that vector is moved from the temporary location into the
MPAC and the mode is set to vector.

The pushdown stack is not a complex structure, but does have strict rules on how
data is placed into it, and how it is removed. Unfortunately, there is no means to
verify whether the data moving between the stack and MPAC actually reflects the
datatatypes required by an interpretive instruction. A series of instructions operating
exclusively on double precision quantities can conclude by an instruction to save a
vector onto the stack, and the Interpreter will not be the wiser. In essence, there is
nothing preventing a programmer from bypassing the little integrity that the stack
provides, and even its internal structure is not immune from operations that alter its
contents. Ideally, as a higher level data structure, the physical characteristics of the
stack should be invisible to us: the number of words allocated, the direction of the
stack pointer movement, and even the physical location of the stack should be
hidden from the programmer. In reality, in a system as storage constrained as the
AGC, the luxury of such a “black box” implementation of the stack is not practical.
The amount of space allocated to the stack in the Vector Accumulator Area is a
compromise of many factors, but ultimately, must be large enough to hold the
maximum number of entries anticipated in an interpretive mission program. There
are no program alarms or error recovery defined for stack overflows, placing the
responsibility for stack management squarely on the programmer.

The manual management of the stack does have a useful side effect. Absolute
knowledge of how far a stack is pushed down during a series of calculations gives rise
to an obvious but overlooked fact: knowing the amount of data already pushed onto
the stack gives the location of the top of the stack. If the programmer is assured that
the current operations will not place more than 30 words on the stack, eight words in
the pushdown stack remain unused. In the AGC, the availability of unused erasable
memory locations often presents an irresistible temptation. The Set Pushdown Stack
Pointer (SETPD) instruction gives us the ability to manually override the value of
the stack pointer. By explicitly setting the stack pointer, PUSHLOC, to a known
vacant location, the unused stack area can be used as a temporary storage area for
intermediate results. Another more common use of the SETPD instruction is when a
computation has progressed to a point where a decision is necessary on how further
processing will continue. If the present calculation is abandoned in order to start
another one, using SETPD to reset PUSHLOC back to zero effectively clears the
stack and throws away its contents.
Store instructions

Once the thrill of a well-executed interpretive string is over, there is the need to save the results in storage. Contrasted with the basic AGC instructions that are used to move data from the accumulator, the variety of datatypes and indexing options require a far more complex set of instructions for storing data. Starting with the apparently trivial objective of saving the contents of the MPAC, important decisions quickly arise on whether to save a double or triple precision variable, a scalar value or a vector.\footnote{Storing a double word when a vector is in the MPAC is not supported in the Interpreter, allowing us to make the weak argument that the Interpreter is a “strongly typed” language.} Optionally, there is the ability to use the indexing capability built into the Interpreter. In all, six combinations of datatypes and indexing operations are possible. While it would be possible to create unique operation codes for each combination, the Interpreter architecture provides all the elements necessary to implement a single STORE instruction.

Flexibility in the STORE instruction requires abandoning the traditional Interpretive Instruction Word format for an entirely new layout adapted for storing data. Specifically, the STORE instruction combines both the instruction opcode and the erasable storage address into a single word of storage. It designates eleven bits (bits 11 through 1) for its address field, bits 14 through 12 for the opcode, and always has zero in bit 15. Setting bit 15 to zero, thus creating a positive number, is a significant departure from the convention used by the Interpreter, where all IIWs are saved as negative numbers. A positive value for the STORE instruction has the potential to cause confusion, as positive values are also used for operand addresses.\footnote{We will consider this issue in our upcoming discussion of the STADR instruction.} With only three bits in the opcode field, eight variations of the store instruction are possible. Perhaps the most important feature of the STORE format is that eleven bits provides addressability throughout all 2K of erasable storage, thereby allowing the instruction to save data without the need to explicitly manipulate the EBANK register.

<table>
<thead>
<tr>
<th>opcode</th>
<th>Erasable Memory Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>15</td>
</tr>
</tbody>
</table>

Figure 62: STORE instruction format

The prospect of eight variations on storing data seems a bit excessive, but only a few distinct addressing modes are implemented. The STORE instruction is not limited to moving the MPAC contents directly to erasable storage. Either of the two index registers may modify the destination address, creating three different addressing modes available to STORE: no indexing, indexing using X1 and indexing using X2. Left with an additional five possible variations, powerful extensions to the otherwise utilitarian STORE instruction are available.
When writing a program, a useful observation becomes apparent: after data is stored, the implication is that the particular program fragment is complete and no further processing remains. In an interpretive string, this is seen as the STORE operation followed by exiting the Interpreter and returning to the execution of basic AGC instructions. A second case is more common, where processing continues in the Interpreter and computation begins on another piece of data. Invariably then, the STORE instruction is followed immediately by a load into the MPAC of the next piece of data. This second case occurs so frequently that it is useful to create a combined “Store and Load” instruction. Such an instruction fits comfortably in the repertoire, as it requires only two operands: the store location and the address of the next piece of data to load. The instructions STODL and STOVL first copy the MPAC contents to erasable storage and then reload the MPAC with a new operand. STODL loads a double word into the MPAC, setting its mode to double. Likewise, STOVL loads the MPAC with a vector, setting the mode to vector. The efficiency of combining two common instructions is obvious, and the benefits of these combined instructions extend beyond the notational shorthand. Savings on processing cycles are not trivial, as the interpretive process is already quite slow. The time saved by not fetching, decoding and dispatching a new instruction adds up quickly, allowing the AGC to pursue work that is more productive. In addition, extensive use of STODL, STOVL and STCALL result in the saving of hundreds of precious words of memory in both the Command Module and Lunar Module computers. Loads of triple precision values are not supported in the combined STORE/LOAD instructions, nor are loads of single precision values. Neither of these datatypes are sufficiently common to justify a separate instruction.

Although the targets of STODL and STOVL are limited to direct and vacuous addressing (that is, indexing is not possible), two additional opcodes do allow indexing. Indexing is specified as in other interpretive instructions, with a ‘*’ being appended to the end of the instruction name. The indexed version of STODL becomes STODL*. As in other interpretive instructions, the index register is appended to the operand address, separated by a comma. The differences between these two syntaxes are seen below.

<table>
<thead>
<tr>
<th>STODL</th>
<th>STOREADDR</th>
<th>STODL*</th>
<th>STOREADDR</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOADADDR</td>
<td>STOREADDR</td>
<td>LOADADDR,1</td>
<td></td>
</tr>
</tbody>
</table>

Not Indexed | Indexed

A third variant of STORE, STCALL, reflects the observation that after storing an operand there is often the need to call to another routine to continue processing. STCALL, in a departure from the load and store model, follows the data store with a call to another location in the interpretive string. As no new data is loaded into the MPAC, its mode remains unchanged. Identical to a sequence of STORE followed by
a CALL instruction, STCALL places the address of the following instruction into the interpretive return register, QPRET.

Instruction formats for STODL, STOVL and STCALL are the same as that for STORE: bit 15 equals zero, followed by a three-bit opcode and eleven bits for the erasable storage address. Coding in the source statements is also similar, where the operand address follows the operation code, all on the same source statement. Following the operation code is the operand address of the data to be loaded into the MPAC, or the address of the subroutine that will be called. When examining the instruction format of the STORE variants, a seemingly insignificant characteristic becomes apparent. With bit 15 of the instruction word equal to zero, it is now indistinguishable from Interpreter Address Words. The flexibility of the interpretive programming language now exposes an issue when using STORE instructions. Consider the situation where an instruction uses a vacuous address, taking its operand from the pushdown stack rather than fetching an operand whose address lies in the interpretive string. Vacuous addresses rely on the fact that the operand address is not explicitly coded, implying that the pushdown stack is the source or destination of the data.

When the Interpreter fetches the word in storage for what it expects to be an operand, a check of bit 15 is meant to reveal whether an operand or another instruction word is present. If bit 15 is set to one (a negative value), the word is interpreted as an operand address and vacuous addressing is not used. Conversely, when bit 15 is zero, we have encountered an instruction word and the pushdown stack is used as the operand. But what happens if an instruction intending to use a vacuous address is followed immediately by a STORE instruction? By the rules just described, the Interpreter recognizes STORE as an operand address, and uses it to fetch data. Once treated as an operand, the STORE instruction cannot execute as intended, leaving the programmer with a difficult-to-find error.

<table>
<thead>
<tr>
<th>Explicit Operand</th>
<th>Vacuous Operand</th>
</tr>
</thead>
<tbody>
<tr>
<td>INST1 DLOAD</td>
<td>INST1 DLOAD</td>
</tr>
<tr>
<td>DLOADAD</td>
<td>DLOADAD</td>
</tr>
<tr>
<td>STORE STORELOC</td>
<td>INST1 INST2</td>
</tr>
<tr>
<td></td>
<td>STORE STORELOC</td>
</tr>
</tbody>
</table>

INST1, INST2 are arbitrary instructions

Problem: Is STORE an Instruction or Operand?  Solution: STADR tells Interpreter STORE is an Instruction

| INST1 DLOAD | INST1 DLOAD |
| STORE STORELOC | STADR STORE STORELOC |

Figure 63: Uses of the STORE instruction and STADR
STADR is a novel instruction used to resolve this conflict. STADR insures the intended program execution by first instructing the assembler to complement the following instruction when it is saved in storage. Thus, a STORE instruction that might resolve as 02715₈ is now saved as 7506₂₈. During execution, STADR instructs the Interpreter to complement the instruction one more time, and the original STORE instruction executes as expected. STADR can only exist as a right-hand operation code, as it operates only on the next word of an interpretive string.

Managing scaling
A scalar function is used to set the magnitude of a triple precision value in the MPAC to between 0.5 \( \leq N \leq 0.999999999 \). Remember that the fractional datatype in the AGC is to simplify the number format itself; elaborate floating point data formats are not part of the AGC architecture. As calculations progress on a particular variable, it is possible that the magnitude of the data becomes less and less, as seen when multiplying small numbers together. The precision of the resulting quantity, while perfectly accurate, becomes impractically small to use with subsequent calculations, creating the risk of underflowing the accumulator in the ensuing operations. Through the Normalization (NORM) instruction, the fractional value is shifted to the left until the highest order "1" bit is in bit 14 of the high order word. For example, NORM will alter the value of 0.741 \( \times 2^{-9} \) to 0.741 \( \times 2^{-1} \), allowing further multiplications without the risk of underflow. As calculations in the AGC place the responsibility for knowing each variable’s magnitude squarely on the programmer, he must maintain a record of how much the magnitude of the data has changed. Unfortunately, the programmer might know the exact scale of a variable in the middle of a computation only generally, perhaps at best to within only an order of magnitude. NORM solves the problem of losing scaling information by requiring an operand, into which it saves the negated value of the number of shifts performed.

UNIT is a unary operator acting only on the vector in the MPAC. In spaceflight, vectors are used in a wide variety of applications ranging from the position and velocity of a vehicle on its way to the Moon, to the small relative motions between two spacecraft about to dock. These vectors exist with widely different magnitudes, where managing the scaling is an aggravating necessity. Many calculations, especially those involving trigonometric operations, cannot easily use these values with significantly different magnitudes, and require their data in the form of a unit vector. Unit vectors in themselves are completely unremarkable, except for the essential property of having a magnitude equal to one. Converting an arbitrary vector to a unit vector preserves the relative magnitude of each component, leaving the direction of the vector unchanged. Additionally, UNIT scales the vector so that any concern about very large or small components is minimized.

To see how unit vectors work, consider the following example. Assume a car is traveling 75 km/hr; 13.3 km/hr along the X-axis, 73.8 km/hr along the Y-axis, and 0 km/hr along the Z-axis. Converting these components to a unit vector creates an X-
axis value of 0.178, a Y-axis value of 0.984, and 0.0 for the Z-axis, resulting in a magnitude of one. The UNIT vector function converts the vector in the MPAC into the vector form, and leaves two other useful values in the Vector Accumulator Area. These values are intermediate quantities generated when calculating the unit vector. The square (dot product of itself) of the original vector is saved in VAC offset 34, and the original magnitude of the vector is saved in VAC offset 36. These values are available to the programmer.

**Miscellaneous arithmetic instructions**

Double Precision Multiply and Round (DMPR) is a variation on the Double Precision Multiply (DMP) instruction. Rather than placing a triple precision product into the MPAC, the multiplication result is rounded to a double precision value before it is saved in the MPAC. To assure a true double precision result, the most significant two words are placed in the MPAC. As double and triple precision words share the same area in the MPAC, the third and least significant word of the MPAC’s triple precision data area is set to zero. Multiplication then takes place as expected, in the same manner as the DMP instruction.

Similar to Double Precision Add, Triple Precision Add (TAD) adds the contents of the MPAC with all three words of the triple precision operand. As double precision data is the predominant scalar datatype in interpretive strings, it is not surprising that addition is the only triple precision operation. This is not as restrictive as it might appear. Complementing a triple precision value, which can be used as part of subtraction logic, is available as one of the scalar functions.

Combining the testing of a double precision value’s sign with an optional data load into the MPAC is possible with the SIGN instruction. This first tests whether the double precision value in erasable storage is positive or negative. If the variable is greater than or equal to zero, processing continues without any further action. If a negative value is found, the triple precision value in the MPAC is complemented regardless of whether the mode is double or triple precision. If the MPAC’s mode is set to vector, all components of the vector in the MPAC are complemented.

**Trigonometric instructions**

With so much of the guidance and navigation expressed using angles and ellipses, it is no surprise that trigonometric functions form the backbone of their equations. Four trigonometric operations SIN, COS, ARCSIN and ARCCOS (SINE, COSINE, ASIN and ACOS are also valid names) are available as unary operations to replace the double precision value in the MPAC with a triple precision solution. SIN and ACOS are computed by the Hastings approximation, which calculates a reasonable estimate using high order polynomials. While more accurate techniques are available, methods such as a Taylor series require far more computational resources. Despite using approximations, a single trigonometric operation still requires several milliseconds to calculate: 5.6 ms for SIN, and 9.1 ms for ACOS. To economize on memory, code that calculates COSINE and ARCSIN is not written explicitly, but rather is derived from solving for SIN and ARCCOS. A COSINE, for example, is calculated from \(\sin\left(\frac{\pi}{2} - x\right)\), and ARCSIN is derived from \(\frac{\pi}{2} - \text{ARCCOS}(x)\).
This incurs the small cost of additional processing, but provides savings by not dedicating memory to several mathematically related functions. The remaining trigonometric functions, TANGENT, SECANT and inverses are not included in the Interpreter instruction set. In the few cases they are required, they can be derived from the existing functions.

Subroutines calls and transfers of control
As in other languages, interpretive strings perform a series of computations, and are followed by a transfer of control to another location to perform another task. Frequently, these transfers of control are dependent on the state of some piece of data. Questions such as “Has the astronaut pressed the ENTER or PROCEED key?” or “Are we in the correct attitude for the next maneuver?” determine the future processing of a piece of code. This discussion now turns to three groups of instructions that are dedicated to sequence changes: subroutine calls, conditional and unconditional branches, and transfers into and out of the Interpreter.

Beginning with the most familiar type of sequence changes, the Interpreter provides a full complement of branches, with and without condition testing. In all types of transfer of control, the target address points to the first instruction of the Interpretive Instruction Word. By limiting references to operation codes on whole word boundaries, returning subroutines can only transfer control back to left-hand opcodes. While this is unsurprising in the AGC basic instruction set, the ability to store two interpretive operations in a single word presents a problem of addressing within the IIW. As seen in previous discussions, the fixed word size in the AGC demands that bit allocation is a zero-sum game. In assigning an additional bit for specifying an opcode within the IIW, another bit previously used for the address must be sacrificed. Ultimately, the inconvenience of forcing destination addresses to word boundaries is outweighed by the additional cost of interpretive addressing limitations.23

The most basic transfer of control, GOTO, is an unconditional branch to another IIW in the interpretive string.24 As an unconditional transfer of control, GOTO is written as a right-hand opcode, ensuring that no instruction follows it in the IIW. As expected, an unconditional branch as a left-hand operation would make any instruction coded in the right-hand field inaccessible. A Computed GOTO (CGOTO) augments the unconditional branch with the means to dynamically modify the branch’s destination address. Used to perform multiway branches based

23 To reference individual opcodes within an IIW requires at least one additional bit in the interpretive address to specify which of the two opcodes is the branch destination. In the Interpretive Address Word, this forces us to reduce the addressing range of the IAW to 13 bits, or 8K words, from the current 14 bits and 16K words. Addressing would be further complicated by the fact that the reduced addressing range would impose the use of memory banking.

24 Contemporary programmers are allowed to cringe at the sight of a GOTO instruction. It is important to remember, however, that much of the AGC software development was well under way by the time Edsger Dijkstra made his case against the GOTO statement.
on a computed value, a CGOTO instruction has two operands: the first IAW points to an erasable storage location whose contents serve as an index value for the second operand; the second operand is a destination address, usually referencing a table of address words in storage, with each address word containing a final destination address of the CGOTO. When CGOTO executes, the contents of the first operand are added to the table address specified in the second operand, and then this address is used to fetch the final destination address of the GOTO. The complexity of an instruction combining both indexing and indirection makes the following example useful:

<table>
<thead>
<tr>
<th>NOOP</th>
<th>CGOTO</th>
<th>IGNORE LEFT HAND OPCODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>INDEX</td>
<td>ADDRESS OF INDEX VALUE</td>
<td></td>
</tr>
<tr>
<td>GOTOTBL</td>
<td>TABLE OF GOTO ADDRESSES</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>INDEX</th>
<th>DEC 0</th>
<th>IN ERASABLE STORAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>GOTOTBL</td>
<td>CADR</td>
<td>GOTO1</td>
</tr>
<tr>
<td></td>
<td>CADR</td>
<td>GOTO2</td>
</tr>
<tr>
<td></td>
<td>CADR</td>
<td>GOTO3</td>
</tr>
</tbody>
</table>

Figure 64: Computed GOTO example

Calling a subroutine is a universal concept in software. It is used as a structured programming technique for where a single routine performs functions common to several different programs. In servicing many different calling programs, subroutines must also have the ability to return to any of those programs, each residing at a different location in memory. The process begins with the calling program saving its return address in a well defined location, and then branching into the subroutine. The requirement that the calling program saves its return address is critical. When written, a called program has no idea where it will be called from, and thus no idea where it must return to. After completing its processing, the subroutine transfers control back to the calling program using the return address as the point where processing resumes. In the Interpreter, the QPRET storage area located in the Vector Accumulator area is the location where all calling programs store their return addresses. QPRET is entirely separate from, but similar in function to the AGC’s Q register. Since each VAC are has its own QPRET area, interpretive strings can call subroutines without regard to the calling sequences of other interpretive jobs. Consistent with the limitations on addressing individual opcodes, the Interpreter saves the address of the next instruction word in the interpretive string, rather than of the next interpretive operation code in the IIW.

Calling a subroutine in the Interpreter is through the CALL instruction, with the address of the subroutine in the Interpretive Address Word. When executed, CALL places the address of the next word of the interpretive string into QPRET and branches to the address in the operand. A related operation called the Computed Call (CCALL) is similar to the Computed GOTO. The operand in erasable storage is
added to the subroutine address to create a new calling address. CCALL performs a subroutine call to the computed address and saves the return address in QPRET. Creating a destination address calculated from this addition allows the programmer to dynamically select which subroutine is required in a given situation. CALL and CCALL, like all unconditional branching instructions, must be coded as right-hand instructions.

After completing the processing in the subroutine, it is time to return to the calling program using the address saved in the QPRET storage area. Executing the Return Via QPRET (RVQ) instruction is similar to an indirect branch, as the address stored in the QPRET location becomes the destination address. But what if the current subroutine is entered through a sequence of previous subroutine calls? Since QPRET is only a single storage word and not a dynamic structure like a stack, each called program has the responsibility of saving its return address before it makes a subroutine call on its own. Although the Interpreter implements a stack, it is used only for data computations, and not for nesting subroutine calls. Lacking any hardware or software structures to retain the calling history, it is the burden of each called program to save QPRET before it makes a subroutine call of its own. The dearth of erasable storage in the AGC limits the depth of subroutine calls. Although QPRET is addressable in the VAC area, a dedicated instruction, Store QPRET (STQ) is used to save the contents of QPRET in erasable storage. Unfortunately, no comparable dedicated instruction exists to restore QPRET. Reloading QPRET is a manual and imperfect process, requiring loading the saved return address in the MPAC and then saving it in QPRET. After the restoration of QPRET is complete, the program can return using the RVQ instruction.

**Sequence changing and indirect**

Program logic often requires transferring control to different locations based on the current state of its processing. For example, a timing loop may sample the clock, and then execute one of several functions based on the current time. Unfortunately, implementing a branch instruction that uses a dynamically chosen target address is particularly difficult. Multiway test and branch statements, as implemented in C and other languages, are impractical due to the large amount of storage required and the difficulty of implementing such statements within the IIW/IAW format of the Interpreter. None of the transfer of control instructions use indexing, so index registers are not available as a means of dynamically choosing a destination address. Another possible option is generating a branch address at execution time, using the Computed GOTO or Computed Call instructions, but these are not appropriate for all transfers of control. However, setting the target address to point to an erasable storage location creates a powerful new possibility for transfer-of-control instructions. All Interpreter instructions must be in fixed storage, but it is perfectly allowable to have operand references to erasable storage. Although an instruction in erasable storage cannot serve as the target for a transfer of control, this allows defining a branch to erasable storage as a new address mode, called indirect. In indirect addressing, the destination of the branch instruction is in erasable storage, which is not a legal address for an IIW. Rather than aborting the instruction due to a
bad destination address, the data in the erasable storage location is used as another storage address, not an instruction word. Presumably, this address references a legitimate fixed storage address. The Interpreter, recognizing that the destination address is in erasable storage, uses the contents of the erasable word as the "real" branch destination. Not surprisingly, instructions that transfer control cannot be limited to a single level of indirection. Consider the case where the contents of the indirect word saved in erasable storage is itself in erasable storage. One level of indirection results in an invalid address, which requires us to use additional levels of indirection in the hope of finding a fixed storage address to branch to.

**Conditional branches**

Without a means to test data and alter processing flow based on the results of that evaluation, few programs of any sophistication are possible. Conditional branches combine into one instruction both a test of the MPAC contents and going to a new location. Ideally, the ability to test the MPAC against a value located in storage would simplify the task for programmers and improve the readability of the code. However, testing against a user-defined value requires additional overhead when fetching a double or triple precision value from storage and performing the comparison. While admittedly useful, the Interpreter avoids the processing and storage overhead of testing the MPAC against values in storage by making all comparisons a test of the MPAC against a value of zero. From testing against zero, the MPAC can be tested for a positive value, negative, or equal to zero; and by extension, for being not equal to any of these. Operating only on scalar quantities further simplifies conditional branches, and comparisons against vectors or matrices are not possible.

Six conditional branching instructions offer five different tests of the MPAC’s contents. These instructions first evaluate the triple precision value in the MPAC against zero. Next, a test for a particular condition (such as positive or negative) is made, and if the result is true, the branch is taken. Branch Plus (BPL) tests if the MPAC is positive. Branch Zero (BZE) checks if the MPAC is zero. Branch Minus (BMN) tests whether the MPAC contents are negative. A fourth operation, Branch on High Order Zero (BHIZ) tests only in the higher order word in the MPAC against zero. BHIZ has two novel uses. First, if a single precision value is in the MPAC, only that value is evaluated, as the remainder of the MPAC might contain invalid data. BHIZ is also useful when generating a transcendental number, and we want to find out if the computation has reached a particular level of precision. Subtracting the result of the latest iteration with the previous one, we know we have reached at least four decimal digits of precision when the high order word is zero.

Two additional conditional branches do not test the value in the MPAC; rather, they test whether a previous arithmetic operation has set the overflow indicator, OVFIND. As discussed earlier, overflows in the Interpreter do not inhibit interrupts, and only an explicit test for overflows can identify the condition. Branch on Overflow (BOV) tests the overflow indicator, and if set, executes the branch. Branch on Overflow to Basic (BOVB) is used where processing an overflow condition is beyond the capabilities of the Interpreter. If so, BOVB exits the Interpreter and
branches to a location containing basic AGC instructions. Transferring control out of and back into Interpreter is a powerful capability discussed more fully with the RTB instruction.

**Addressing scope of branches and calls**
Branching instructions can reference any location in the Interpreter’s addressing range, in contrast to the addressing of other instructions, which are limited to the current half-memory. These transfers of control become quite powerful, but sanity checking of destination addresses does not occur during program execution. Nothing in the Interpreter prevents a branch or subroutine call from transferring control outside of the interpretive string and into basic AGC code, and the results would certainly be unpredictable. The Interpreter, unable to tell whether a word in storage contains an interpretive instruction, a basic AGC instruction, or data, will blithely execute whatever the contents of the word decode to. A run of misinterpreted instructions will create garbage data that will sooner or later prompt a program alarm, followed by a restart to clear the error.

**Entering and exiting the Interpreter**
Strictly speaking, an operation that transfers control within the Interpreter also allows transfers out of the Interpreter’s control. So much of the AGC hardware is hidden from the programmer through the virtual machine created by the Interpreter that most Executive level operations are inaccessible. Leaving the Interpreter entirely becomes the only means to access the hardware status bits, some memory areas, or reading and setting bits in the I/O channels. A common reason for exiting the Interpreter is to access the Executive routines that perform operations on specific registers and I/O channels. Under this design, a common technique is to leave the Interpreter, call the processing routine and then return to the Interpreter’s control. Leaving the Interpreter is by executing the Return to Basic (RTB) operation, which suspends the fetching and execution of IIWs. RTB combines the exit from the Interpreter with a subroutine call using basic AGC instructions. Establishing a standardized call to a subroutine requires saving the return address in the AGC’s Q register, which is the return address for all basic instruction subroutine calls.

A novel twist of RTB is that the Q register is not loaded with the address of the IIW immediately following the RTB operation. Returning from the subroutine directly into an interpretive string would quickly fail, as the IIWs would then be fetched and executed as basic AGC instructions. Instead, the program must reenter the interpretive system. By placing the entry address of the Interpreter itself in the Q register, the familiar “TC Q” at the end of the AGC routine brings the execution sequence back into the Interpreter, which then restarts processing of the IIW immediately following the RTB. This design is especially elegant, as the called routine is now available to both basic and interpretive programs. No special coding

---

25 Arguably, RTB could just as easily be renamed “Call Basic Subroutine”. 
is necessary to adapt the routine to these two modes, creating important simplifications in the design of the code. The sole restriction for the called routine is that it must reside in fixed storage, reflecting the requirement that the Interpreter can only process fixed storage addresses. Data passed to the routine must also be independent of whether a basic or interpretive program is performing the call. This precludes using the VAC area as a location for exchanging data between the programs.

Executing RTB does not signify that the job is through with interpretive processing. Rather, RTB is executed with the intention that its time in basic mode will be very brief – a quick excursion to perform an important task before returning back to the Interpreter. No explicit provisions are made to preserve the interpretive environment for the time when the subroutine completes and returns control to the Interpreter. Importantly, no changes are made to the banking registers, forcing the basic AGC routine to use the same memory scheme as the Interpreter. The job’s Core Set and VAC area are addressable when executing the basic AGC instructions, and the job is free to modify either as it sees fit. Upon returning to the Interpreter, these storage areas are not refreshed or restored in any way; they retain the changes made while in basic mode. Because the interpretive environment is preserved, there is no need to restrict RTB to the class of right-hand opcodes. As the intent is to resume the interpretive string immediately after the point it was exited, right-hand instructions must be allowable.

RTB allows us to leave the Interpreter temporarily, but always with the intention of returning to its control. A more permanent means of departing the Interpreter is through the EXIT statement. Unlike RTB, the banking registers are restored to the values that existed before the Interpreter was called. The Core Set and VAC are not cleared, but the VAC area is now available for reuse. After EXIT is processed, execution continues under hardware control with the next basic AGC instruction following the interpretive string.
The Apollo Guidance Computer
Architecture and Operation
O'Brien, F.
2010, XVIII, 440 p. 30 illus., Softcover
ISBN: 978-1-4419-0876-6
A product of Praxis