A reprint from
American Scientist
the magazine of Sigma Xi, The Scientific Research Society

Moonshot Computing

Getting to the Moon required daring programmers as well as daring astronauts.

Brian Hayes

Fifty years ago, three astronauts and two digital computers took off for the Moon. A few days later, half a billion earthlings watched murky television images of Neil Armstrong and Buzz Aldrin clambering out of the Apollo 11 lunar module and leaving the first human bootprints in the powdery soil of the Sea of Tranquility. (Michael Collins, the command module pilot, remained in lunar orbit.) The astronauts became instant celebrities. The computers that helped guide and control the spacecraft earned fame only in a smaller community of technophiles. Yet Armstrong's small step for a man also marked a giant leap for digital computing technology.

Looking back from the 21st century, when *everything* is computer controlled, it's hard to appreciate the audacity of NASA's decision to put a computer aboard the Apollo spacecraft. Computers then were bulky, balky, and power hungry. The Apollo Guidance Computer (AGC) had to fit in a compartment smaller than a carry-on bag and could draw no more power than a light bulb. And it had to be utterly reliable; a malfunction could put lives in jeopardy.

Although the AGC is not as famous as the astronauts, its role in the Apollo project has been thoroughly documented. At least five books tell the story, and there is more information on the web. Among all the available resources, one trove of historical documents offers a particularly direct and intimate look inside this novel computer. Working from rare surviving printouts, volunteer enthusiasts have transcribed several versions of the AGC software and published them online. You can read through the programs that guided Apollo 11 to its lunar touchdown. You can even run those programs on a "virtual AGC."

Admittedly, long lists of machine instructions, written in an esoteric and antiquated programming language, do not make easy reading. Deciphering even small fragments of the programs can be quite an arduous task. The reward is seeing firsthand how the designers worked through some tricky problems that even today remain a challenge in software engineering. Furthermore, although the documents are technical, they have a powerful human resonance, offering glimpses of the cultural milieu of a high-profile, high-risk, high-stress engineering project.

Navigation, Guidance, and Control

Each Apollo mission to the Moon carried two AGCs, one in the command module and the other in the lunar module. In their hardware the two machines were nearly identical; software tailored them to their distinctive functions.

For a taste of what the computers were asked to accomplish, consider the workload of the lunar module's AGC during a critical phase of the flight—the powered descent to the Moon's surface. The first task was navigation: measuring the craft's position, velocity, and orientation, then plotting a trajectory to the target landing site. Data came from the gyroscopes and accelerometers of an inertial guidance system, supplemented in the later stages of the descent by readings from a radar altimeter that bounced signals off the Moon's surface.

After calculating the desired trajectory, the AGC had to swivel the nozzle of the rocket engine to keep the capsule on course. At the same time it had to adjust the magnitude of the thrust to maintain the proper descent velocity. These guidance and control tasks were particularly challenging because the module's mass and center of gravity changed as fuel was consumed and because a spacecraft sitting atop a plume of rocket exhaust is fundamentally unstable—like a broomstick balanced upright on the palm of your hand.

Along with the primary tasks of navigation, guidance, and control, the AGC also had to update instrument displays in the cockpit, respond to commands from the astronauts, and manage data communications with ground stations. Such multitasking is routine in computer systems today. Your laptop runs dozens of programs at once. In the early 1960s, however, the tools and techniques for creating an interactive, "real-time" computing environment were in a primitive state.

Chips and Cores

The AGC was created at the Instrumentation Laboratory of the Massachusetts Institute of Technology (MIT), founded by Charles Stark Draper, a pioneer of inertial guidance. Although the Draper lab had designed digital electronics for ballistic missiles, the AGC was its first fully programmable digital computer.

For the hardware engineers, the challenge was to build a machine of adequate performance while staying within a tight budget for weight, volume, and power consumption. They adopted a novel technology: the silicon integrated circuit. Each lunar-mission computer had some 2,800 silicon chips, with six transistors per chip.

For memory, the designers turned to *magnetic cores*—tiny ferrite toroids that can be magnetized in either of two directions to represent a binary 1 or 0. Most of the information to be stored consisted of programs that would never be changed during a mission, so many

Brian Hayes is a former editor and columnist for American Scientist. *Email: brian@bit-player.org*

of the cores were wired in a read-only configuration, with the memory's content fixed at the time of manufacture.

The logic circuits and memory cores were sealed in a metal case tucked away in an equipment bay. The astronauts interacted with the computer through a device called the DSKY (short for "display keyboard" and pronounced *dis-key*), which looked something like the control panel of a microwave oven. It had a numeric keypad, several other buttons, and room to display 21 bright green decimal digits.

Squeezing into 15 Bits

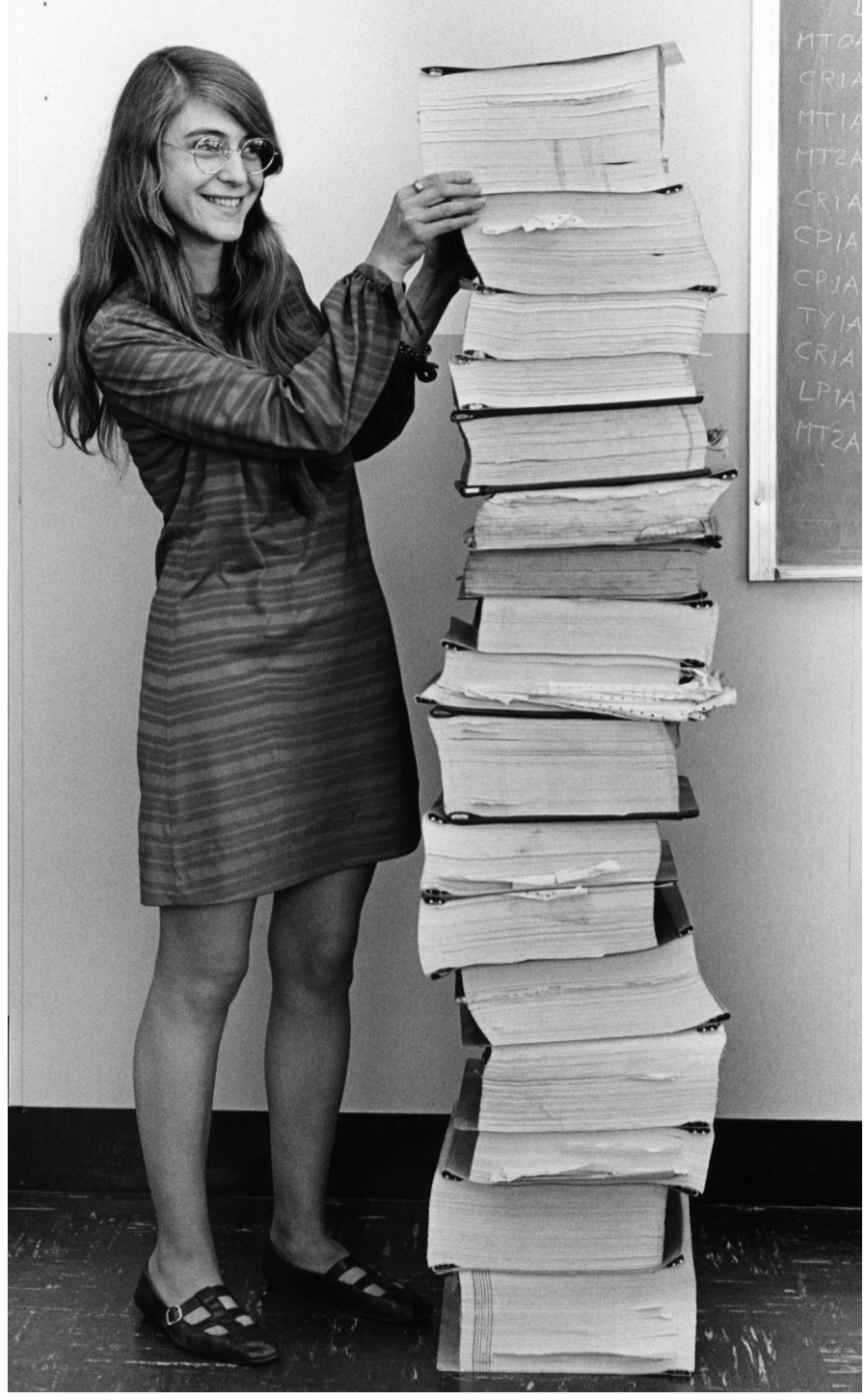
A critical early decision in the design of the computer was setting the number of bits making up a single "word" of information. Wider words allow more varied program instructions and greater mathematical precision, but they also require more space, weight, and power. The AGC designers chose a width of 16 bits, with one bit dedicated to error checking, so only 15 bits were available to represent data, addresses, or instructions. (Modern computers have 32- or 64-bit words.)

A 15-bit word can accommodate $2^{15} = 32{,}768$ distinct bit patterns. In the case of numeric data, the AGC generally interpreted these patterns as numbers in the range ±16,383. Grouping together two words produced a double-precision number in the range ±268,435,455.

A word could also represent an instruction in a program. In the original plan for the AGC, the first three bits of an instruction word specified an "opcode," or command; the remaining 12 bits held an address in the computer's memory. Depending on the context, the address might point to data needed in a calculation or to the location of the next instruction to be executed.

Allocating just three bits to the opcode meant there could be only eight distinct commands (the eight binary patterns between 000 and 111). The 12-bit addresses limited the number of memory words to 4,096 (or 2^{12}). As the Apollo mission evolved, these constraints began to pinch, and engineers found ways to evade them. They organized the memory into multiple *banks*; an address specified position within a bank, and separate registers indicated which bank was active. The designers also scrounged a few extra bits to expand the set of opcodes from 8 to 34.

The version of the AGC that went to the Moon had 36,864 words of read-only memory for storing programs and 2,048 words of read-write memory for ongoing computations. The total is equivalent to about 70 kilobytes. A modern laptop has 100,000 times as much memory. As for speed, the AGC could execute about 40,000 instructions per second; a laptop might do 10 billion.



Wikimedia Commons

Stacked printouts of software for the Apollo Guidance Computer (AGC) form a tower five and a half feet tall, the height of Margaret H. Hamilton, who joined the project as a programmer in 1963 and a few years later became director of software engineering. Each binder holds the programs for either the command module or the lunar module for a single mission. The photograph was taken at the MIT Instrumentation Laboratory in 1969, shortly before the flight of Apollo 11.

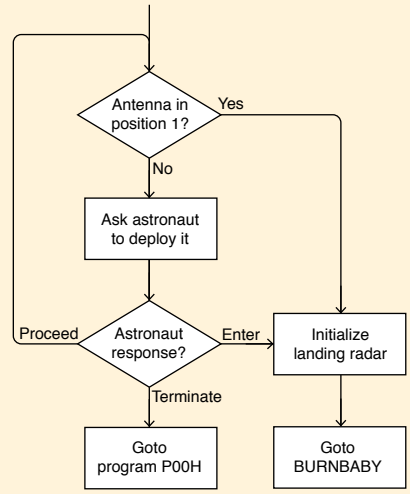
Software Infrastructure

A no-frills architecture, puny memory, and a minimalist instruction set presented a challenge to the programmers. Moreover, the software team at MIT had to create not only the programs that would run during the mission but also a great deal of infrastructure to support the development process.

One vital tool was an *assembler,* a program that converts symbolic instructions (such as `AD` for *add* and `TC` for *transfer control*) into the binary codes recognized by the AGC hardware. The

Reading an AGC Program

line	label	opcode	address	comments
0184	P63SPOT3	CA	BIT6	IS THE LR ANTENNA IN POSITION 1 YET
0185		EXTEND		
0186		RAND	CHAN33	
0187		EXTEND		
0188		BZF	P63SPOT4	BRANCH IF ANTENNA ALREADY IN POSITION 1
0189		CAF	CODE500	ASTRONAUT: PLEASE CRANK THE
0190		TC	BANKCALL	SILLY THING AROUND
0191		CADR	GOPERF1	
0192		TCF	GOTOP00H	TERMINATE
0193		TCF	P63SPOT3	PROCEED SEE IF HE'S LYING
0194	P63SPOT4	TC	BANKCALL	ENTER INITIALIZE LANDING RADAR
0195		CADR	SETPOS1	
0196		TC	POSTJUMP	OFF TO SEE THE WIZARD...
0197		CADR	BURNBABY	



Program P63 in the Apollo 11 lunar lander controlled the early stages of the descent to the Moon's surface. The snippet of source code reproduced here configures a radar altimeter needed for landing. As shown in the flowchart at right, the program first checks the status of the radar antenna; if it is not yet in position, the astronaut is asked to deploy it. Depending on the astronaut's response, the program can check the status again (to make sure the astronaut complied), give up, or initialize the radar regardless of antenna position.

Each AGC instruction has two parts: an *opcode* and an *address*. The opcodes are written as abbreviations, such as CA and TC, but they represent three-bit numeric codes. The table below defines the opcodes appearing in this program fragment. Addresses are also given in symbolic form but represent 12-bit values. A program called an *assembler* translates each opcode-address pair into a 15-bit word stored in the computer's memory.

AGC Opcodes

CA, CAF	ADDR	"Clear and add." Load the contents of ADDR into the accumulator by first setting the accumulator to zero and then adding.
BZF	ADDR	"Branch if zero." If the contents of accumulator are equal to zero, jump to ADDR.
TC, TCF	ADDR	"Transfer control." Jump to ADDR. The instruction also saves a return address, so that a subroutine can jump back to the place it was called from.
RAND	CHAN	"Read-AND." Read from channel CHAN, then apply the Boolean function AND to each bit from the channel and the corresponding bit in the accumulator. (The result is 1 only if both bits are 1.)
EXTEND		Interpret the next opcode as part of an expanded instruction set.
CADR	ADDR	"Complete address." CADR is not a true opcode but a constant designating a full 15-bit address (hence the distinctive coloring).

Lines 184 through 188 check the antenna position. The state of the antenna is recorded in the sixth bit of input-output channel 33; if the antenna is properly positioned, this bit is 0. The program computes the logical AND of the channel reading with the constant 000000000100000, which has a 1 at position 6 and 0 for all the rest of the bits. If the antenna is in position, this operation leaves a value of zero in the *accumulator* (the main site for arithmetic and logical operations). In that case the instruction BZF redirects the program to the location labeled P63SPOT4 at line 194. If the value in the accumulator is *not* zero (implying that the antenna is not yet in position), execution "falls through" to the next instruction at line 189.

The block of instructions beginning at line 189 presents the request to the astronaut. The aim is to call a subroutine named GOPERF1 that displays a message in the cockpit. Here a complication arises. Because the memory of the AGC is divided into several "banks," program P63 cannot directly call the GOPERF1 subroutine. Instead it invokes a subroutine named BANKCALL, which in turn calls GOPERF1. The address of GOPERF1 is placed in the program immediately after the TC BANKCALL instruction, where BANKCALL can retrieve it. Meanwhile, the constant at location CODE500 has been loaded into the accumulator and will be used by GOPERF1 to determine what message to display.

The instructions at lines 192–194 are the three locations to which the GOPERF1 subroutine can return. If the astronaut enters a "terminate" command, program P63 exits and transfers control to P00, the computer's idle routine. A "proceed" command sends the program back to the top of the loop, at line 184, to confirm that the antenna is now in position 1. The "enter" option bypasses this check and calls (via BANKCALL) the SETPOS1 subroutine to initialize the landing radar. When SETPOS1 returns, control passes to BURNBABY, the program that fires the descent engine.

Comments within the program *(gray text)* are ignored by the assembler but are crucial to human understanding of the program. They also offer glimpses of the programmers' personalities.

assembler's primary author was Hugh Blair-Smith, an engineer with extensive background in programming the large computers of that era. The assembler ran on such a mainframe machine, not on the AGC itself. All of the flight-control programs were assembled and committed to read-only memory long before launch, so there was no need to have an assembler on the spacecraft.

A digital simulation of the AGC also ran on a mainframe computer. It allowed programs to be tested before the AGC hardware was ready. Later a "hybrid" simulator incorporated a real AGC and DSKY, as well as both analog and digital models of the rest of the spacecraft and its environment.

Another tool was an *interpreter* for a higher-level programming language, designed by J. Halcombe Laning and written mainly by Charles A. Muntz, both on the MIT team. The interpreted language provided access to mathematical concepts beyond basic arithmetic, such as matrices (useful in expressing control laws) and trigonometric functions (essential in navigation). The price paid for these conveniences was a tenfold slowdown. Interpreted commands and assembly language could be freely mixed, however, so the programmer could trade speed for mathematical versatility as needed.

An AGC program called the Executive served as a miniature operating system. Also designed by Laning, it maintained a list of programs waiting their turn to execute, sorted according to their priority. The computer also had a system of *interrupts*, allowing it to respond to external events. And a few small but urgent tasks were allowed to "steal" a memory cycle without other programs even taking notice. This facility was used to count streams of pulses from the inertial guidance system and from radars.

Brian Hayes

In the Labyrinth of Code

When I first tried reading some AGC programs, I found them inscrutable. It wasn't just the terse, opaque opcodes. The greater challenge was learning to follow the narrative thread of a program with its many detours and digressions. Instructions such as TC and BZF create branch points, where the path through the sequence of instructions abruptly jumps to some other location, and may or may not return to where it came from. Following the trail can feel like playing Chutes and Ladders.

The learning curve for the AGC assembly language is steep but not very tall, simply because there are so few opcodes. To make sense of the programs, however, you also need to master the conventions and protocols devised by the MIT team to get the most out of this strange little machine. The fragment of source code reproduced on the opposite page provides some examples of these unwritten rules.

I was particularly confused by the scheme for invoking a *subroutine*—a block of code that can be called from various places in a program and then returns control to the point where it was called. In the AGC the opcode for calling a subroutine is TC, which not only transfers control to the address of the subroutine but also saves the address of the word following the TC instruction, stashing it in a place called the Q register. When the subroutine finishes its work, it can return to the main program simply by executing the instruction TC Q. This much I understood. But it turns out that a subroutine can alter the content of the Q register and thereby change its own return destination. Many AGC programs take advantage of this facility. In the snippet on the opposite page, one subroutine has three return addresses, one for each of three possible responses to a query. Until I figured this out, the code was incomprehensible.

Current norms of software engineering discourage such tricks, because they make code harder to understand and maintain. But software conforming to current standards would not fit in 70 kilobytes of memory.

Marginalia

In contrast to the cryptic opcodes and addresses, another part of the AGC software is much easier to follow. The comments that accompany the code are lucid and even amusing. These annotations were added by the programmers as they created the software. They were meant entirely for human consumption, not for the machine.

Most of the comments are straightforward explanations of what the program does. "Clear bits 7 and 14." "See if Alt < 35000 ft last cycle." A few gruff warnings mark code that should not be meddled with. One line is flagged "Don't Move," and a table of constants has the imperious heading "Noli Se Tangere" (biblically inspired Latin for "Do Not Touch"). The style of the comments varies from one program to another, presumably reflecting differences in authorship.

Most intriguing are the messages that venture beyond the impersonal, emotionless manner of technical documentation. A nervously apologetic programmer flags two lines of code as "Temporary, I hope hope hope." A constant is introduced as "Numero mysterioso." An out-of-memory condition provokes the remark "No room in the inn." In a few places the tone of voice becomes positively breezy. The passage shown on the opposite page has the following request: "Astronaut: Please crank the silly thing around." As the program checks to see if the astronaut complied, a comment reads, "See if he's lying." One can't help wondering: Did the astronauts ever delve into the source code? Some of them, most notably Buzz Aldrin, were frequent visitors to the Instrumentation Lab.

Hints of whimsy also turn up in names chosen for subroutines and labels. A section of the software concerned with alarms and failures includes the symbols WHIMPER, BAILOUT, P00DOO, and CURTAINS. Elsewhere we encounter KLEENEX, ERASER, and ENEMA. There are a few Peanuts comic strip references, such as the definition LINUS EQUALS BLANKET. The program that ignites the rocket motor for descent to the Moon is titled BURNBABY, an apparent reference to the slogan "Burn, baby, burn!," which was associated with the 1965 Watts riots in Los Angeles.

Perhaps I should not be surprised to find these signs of levity and irreverence in the source code. The programmers were mostly very young and clearly very smart; they formed a close-knit group where inside jokes were sure to evolve, no matter how solemn the task. Also, they were working at MIT, where "hacker culture" has a long tradition of tomfoolery. On the other hand, the project was supervised by NASA, and every iteration of the software had to be reviewed and approved at various levels of the federal bureaucracy. The surprise, then, is not that wisecracks were embedded in the programs but that they were not expunged by some humorless functionary.

In an email exchange, I asked Margaret H. Hamilton about this issue. A mathematician turned programmer who worked on several other MIT projects before joining the AGC group in 1963, Hamilton later became the lab's director of software engineering (a term she coined). "People were serious about their work," she wrote, "but at the same time they had fun with various aspects of comic relief, including things like giving parts of the onboard flight software funny or mysterious names." She also conceded that NASA vetoed a few of their cheeky inventions.

What Could Go Wrong?

Hamilton has said that the Apollo project offered "the opportunity to make just about every kind of error humanly possible." It's not hard to come up with a long list of things that might have gone wrong but didn't.

For example, the AGC had two formats for representing signed numbers: *one's complement* and *two's complement.* Mixing them up would have led to a numerical error. Similarly, spacecraft position and velocity were calculated in metric units but displayed to the astronauts in feet or feet per second. A neglected conversion (or a double conversion) could have caused much mischief. Another ever-present hazard was arithmetic overflow: A number that exceeded the maximum positive value for a 14-bit quantity would "wrap around" to a negative value.

You might suppose that such blunders would never slip through the rigorous vetting process for a space mission, but history says otherwise. In 1996 an overflow error destroyed an Ariane 5 rocket and its payload of four satellites. In 1999 an error in units of measure—pounds that should have been newtons—led to the loss of the Mars Climate Orbiter.

The cramped quarters of the AGC must have added to the programmers' cognitive burden. The handling of subroutines again provides an illustration. In larger computers, a data structure called a *stack* automatically keeps track of return addresses for subroutines,

even when the routines are deeply nested, with one calling another, which then calls a third, and so on. The AGC had no stack for return addresses; it had only the Q register, with room for a single address. Whenever a subroutine called another subroutine, the programmer had to find a safe place to keep the return address, then restore it afterward. Any mishap in this process would leave the program lost in space.

As an outsider imagining myself writing programs for a machine like this one, the area where I would most fear mistakes is the multitasking mechanism. When multiple jobs needed to be accomplished, the Executive always ran the one with the highest assigned priority. But it also had to ensure that all jobs would eventually get their turn. Those goals are hard to reconcile.

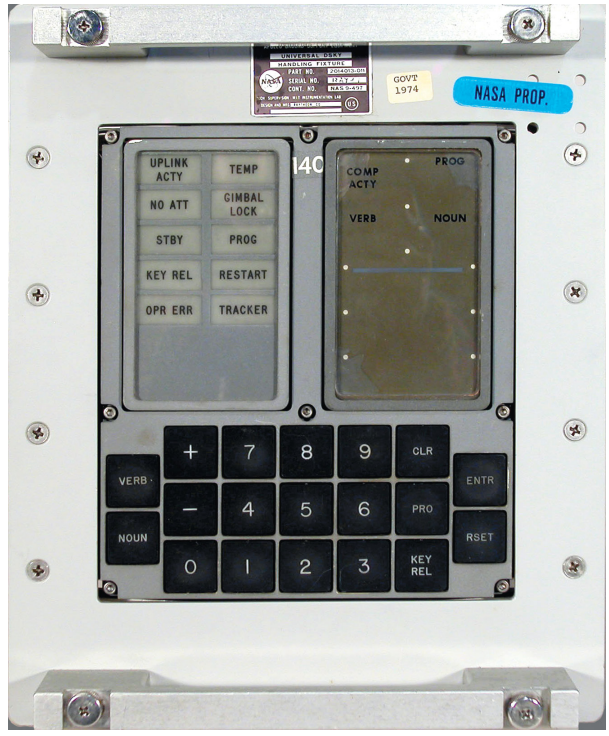
Interrupts were even more insidious. An event in the outside world (such as an astronaut pressing keys on the DSKY) could suspend an ongoing computation at nearly any moment and seize control of the processor. The interrupting routine had to save and later restore the contents of any registers it might disturb, like a burglar who breaks into a house, cooks a meal, and then puts everything back in its place to evade detection.

Some processes must not be interrupted, even with the save-and-restore protocol (for example, the Executive's routine for switching between jobs). The AGC therefore provided a command to disable interrupts, and another to re-enable them. But this facility created perils of its own: If interrupts were blocked for too long, important events could go unheeded.

The proper handling of interrupts and multitasking remains an intellectual challenge today. These mechanisms introduce a measure of random or nondeterministic behavior: Knowing the present state of the system is not enough to predict the future state. They make it hard to reason about a program or to test all possible paths through it. The most annoying, intermittent, hard-to-reproduce bugs can often be traced back to some unanticipated clash between competing processes.

A Five-Alarm Landing

None of the AGCs ever failed in space, but there were moments of unwelcome excitement. As the Apollo 11 lander descended toward the lunar surface, the DSKY display suddenly announced a "program alarm" with a code number of 1202. Armstrong and Aldrin didn't know whether to keep going or to abort the landing. At Mission Control in Houston the decision fell to the guidance officer, Steve Bales, who had a cheat sheet of alarm codes and access to backroom experts from both NASA and MIT. He chose "Go." He made the same decision following each of four further alarms in the remaining minutes before touchdown.



The AGCs in the command module and the lunar module were accessed by the astronauts through a "display keyboard" (DSKY) mounted in the module's control panel. Astronauts specified actions by entering a program, a verb, and a noun, all represented by two-digit numbers. The DSKY shown here is from the Smithsonian Air and Space Museum and never flew on an Apollo mission.

Back at MIT, members of the AGC team were listening to this exchange and scrambling to confirm what a 1202 alarm meant and what might have caused it. The explanatory comment at the appropriate line of the program listing reads "No more core sets." Every time the Executive launched a new job, it allocated 11 words of read-write memory for the exclusive use of the new process. The area set aside for such *core sets* had room for just eight of them. If the Executive was ever asked to supply more than eight core sets, it was programmed to signal a 1202 alarm and jump to a routine named BAILOUT.

During the lunar descent, there were never more than eight jobs eligible to run, so how could they demand more than eight core sets? One of those jobs was a big one: SERVICER did all the computations for navigation, guidance, and control. It was scheduled to run every two seconds and was expected to finish its work within that period, then shut down and surrender its core set. When the two-second interval was up, a new SERVICER process would be launched with a new core set. But for some reason the computation was taking longer than it should have. One instance of SERVICER was still running when the next one was launched, forming a backlog of unfinished jobs, all hanging on to core sets.

The cause of this behavior was not a total mystery. It had been seen in test runs of the flight hardware. Two out-of-sync power supplies were driving a radar to emit a torrent of spurious pulses, which the AGC dutifully counted. Each pulse consumed one computer memory cycle, lasting about 12 microseconds. The radar could spew out 12,800 pulses per second, enough to eat up 15 percent of the computer's capacity. The designers had allowed a 10 percent timing margin.

Much has been written about the causes of this anomaly, with differing opinions on who was to blame and how it could have been avoided. I am more interested in how the computer reacted to it. In many computer systems, exhausting a critical resource is a fatal error. The screen goes blank, the keyboard is dead, and the only thing still working is the power button. The AGC reacted differently. It did its best to cope with the situation and keep running. After each alarm, the BAILOUT routine purged all the jobs running under the Executive, then restarted the most critical ones. The process was much like rebooting a computer, but it took only milliseconds.

Smithsonian Air and Space Museum

Annoying alerts that pop up on the computer screen are now commonplace, but Hamilton points out they were a novelty in the 1960s. The program alarms appearing on the DSKY display were made possible by the priority-driven multitasking at the heart of the AGC software. The alarms took that idea a step further: They had the temerity to interrupt not just other computations but even, when necessary, the astronauts themselves.

A White-Knuckle Job

Some of the veterans of the AGC project get together for lunch once a month. That they still do so 50 years after the Moon landings suggests how important the Apollo program was in their lives. (It also suggests how young they were at the time.) In 2017 I had an opportunity to attend one of these reunions. I found myself asking the same two questions of everyone I met. First, in that minefield of mistakes-waiting-to-happen, how did you manage to build something that worked so well and so reliably? Second, weren't you scared witless?

In reply to the latter question, one person at the table described the development of the AGC as "a white-knuckle job." But others reported they were just too focused on solving technical problems to brood over the consequences of possible mistakes. Blair-Smith pointed out that the informal motto of the group was "We Can Do This," and it wasn't just bravado. They had genuine confidence in their ability to get it right.

The question of exactly *how* they got it right elicited lively discussion, but nothing came of it that I could neatly encapsulate as the secret of their success. They were very careful; they worked very hard; they tested very thoroughly. All this was doubtless true, but many other software projects with talented and diligent workers have run into trouble nonetheless. What makes the difference?

Recalling the episode of the 1202 alarms, I asked if the key might be to seek resilience rather than perfection. If they could not prevent all mistakes, they might at least mitigate their harm. This suggestion was rejected outright. Their aim was always to produce a flawless product.

I asked Hamilton similar questions via email, and she too mentioned a "never-ending focus on making everything as perfect as possible." She also cited the system of interrupts and priority-based multitasking, which I had been seeing as a potential trouble spot, as ensuring "the flexibility to detect anything unexpected and recover from it in real time."

In my mind, how they did it remains an open question—and one deserving of scholarly attention. Engineering tradition calls for careful forensic analysis of accidents and failures, but perhaps it would also make sense to investigate the occasional outstanding success.

Preservation and Access

The Smithsonian Institution's Air and Space Museum holds some 3,500 artifacts from the Apollo program, but the AGC software is not on exhibit there. A few smaller museums have helped preserve printouts, but the programs are widely available today almost entirely through the efforts of amateur enthusiasts.

In 2003 Ronald Burkey was watching the film *Apollo 13*, about the mission imperiled by an explosion en route to the Moon. The DSKY appeared in several scenes, and Burkey, who works in embedded computer systems, set out to learn more about the AGC. Casual inquiries gradually transformed into a dogged pursuit of original documents. His aim was to create a simulator that would execute AGC programs.

Burkey learned that the Instrumentation Laboratory had deposited listings of some Apollo 11 software with the MIT Museum, but the terms of the donation did not allow them to be freely distributed. After long negotiations, Deborah Douglas, director of collections at the museum, secured the release of the printouts, and Burkey arranged to have them scanned. Then several volunteers helped with the tedious job of converting 3,500 page images to machine-readable text.

Meanwhile, Burkey was building not only a simulator, called the Virtual AGC, but also a new version of the assembler. (Initially he had no access to the source code for the original assembler, which in any case would not run on modern hardware.) A crucial test of the whole effort was running the transcribed Apollo 11 source code through the new assembler and comparing the binary output with the 1969 original. After a few rounds of proofreading and correcting—some of the scans were barely legible—the old and new binaries matched bit for bit.

In recent years printouts from several other Apollo missions have been made available to Burkey and his collaborators, mostly by members of the MIT team who had retained private copies. Those programs have also been scanned, transcribed, and reassembled. All the scans and the transcribed source code are available at the Virtual AGC website, http://ibiblio.org/apollo. Also posted there are programming manuals, engineering drawings, and roughly 1,400 memos, reports, and other contemporaneous documents.

The Apollo program might be considered the apogee of American technological ascendancy in the 20th century, and the AGC was a critical component of that success. I find it curious and unsettling that major museums and archives have shown so little interest in the AGC software, leaving it to amateurs to preserve, interpret, and disseminate this material. On the other hand, those creative and energetic amateurs have done a brilliant job of bringing the history back to life. Their success is almost as remarkable as that of the original AGC programmers.

Bibliography

Blair-Smith, H. 2015. *Left Brains for the Right Stuff: Computers, Space, and History*. East Bridgewater, MA: SDP Publishing.

Burkey, R. 2019. Virtual AGC-AGS-LVDC-Gemini Project Overview. Last modified January 30. http://ibiblio.org/apollo

Cherry, G. W. 1969. Exegesis of the 1201 and 1202 alarms which occurred during the Mission G lunar landing. Memorandum from MIT Instrumentation Laboratory to NASA Manned Spacecraft Center, August 4, 1969. Accessed March 25, 2019. http://ibiblio.org/apollo/Documents/CherryApollo11Exegesis.pdf

Eyles, D. 2017. *Sunburst and Luminary: An Apollo Memoir.* Boston, MA: Fort Point Press.

Hall, E. C. 1996. *Journey to the Moon: The History of the Apollo Guidance Computer*. Reston, VA: American Institute of Aeronautics and Astronautics.

Mindell, D. A. 2008. *Digital Apollo: Human and Machine in Spaceflight*. Cambridge, MA: MIT Press.

O'Brien, F. 2010. *The Apollo Guidance Computer: Architecture and Operation*. Chichester, UK: Praxis Publishing.

Savage, B. I., and A. Drake. 1967. *AGC4 Basic Training Manual*. Volume 1. Apollo Guidance, Navigation, and Control Memo E-2052. Cambridge, MA: MIT Instrumentation Laboratory. Accessed March 25, 2019. http://ibiblio.org/apollo/NARA-SW/E-2052.pdf

Read an extended interview with Margaret Hamilton online.