The Origins of
Burroughs Extended Algol

Paul Kimpel

2019 UNITE Conference
Session MCP 4059
Wednesday, 2 October, 9:45 a.m.

Copyright © 2019, All Rights Reserved

The Origins of Burroughs Extended Algol

2019 UNITE Conference
Minneapolis, Minnesota

Session MCP 4059

Wednesday, 2 October 2019, 9:45 a.m.

Paul Kimpel
San Diego, California

http://www.digm.com

e-mail: paul.kimpel@digm.com

Copyright © 2019, Paul H. Kimpel

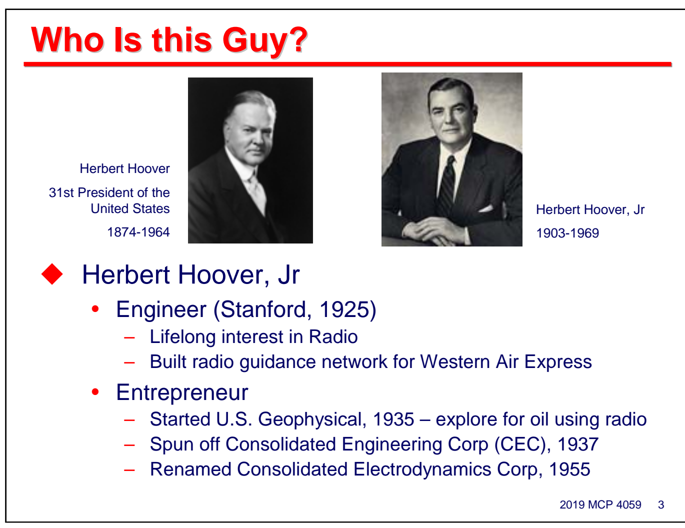
Reproduction permitted provided this copyright notice is preserved
and appropriate credit is given in derivative materials.

Extended Algol has always been the primary language for MCP systems, extending back to the Burroughs B5000 of 1963. No doubt more lines of code have been written in COBOL for these systems, but most of the really significant code has been written in Algol or specialized languages derived directly from Algol. You can't be considered to be a master programmer for MCP systems unless you are also a master Algol programmer. It's our assembler language, our systems language, and for some of us, our everyday, go-to, application development language.

This presentation attempts to trace the development of Algol for Burroughs and Unisys MCP systems, and to highlight how Algol influenced the development of these systems and their software. It starts at the beginning of the commercial computing business for Burroughs and goes through the introduction of the systems in the early 1970s that form the basis of our current MCP (E-mode) architecture – the B6500/B6700. It does not attempt to cover the many significant enhancements to Extended Algol that have occurred since then. That is a whole other talk in itself.

Our story begins in Pasadena, California, a city to the north-east of downtown Los Angeles. "Pasadena" is a Chippewa word meaning "crown of the valley," as it is in the foothills of the San Gabriel Mountains, overlooking the San Gabriel Valley. It has always been home to a thriving mixed population, engaged in art, culture, business, and technology. In particular, you find there the Huntington Library and Gardens, the Norton Simon Museum, the Pasadena Playhouse, the Rose Bowl, the Tournament of Roses Parade, and a whole lot of high technology – including Caltech, and the Jet Propulsion Laboratory.

The Algol language itself did not begin in Pasadena, but we'll get to that.



The man in the picture on the right is someone about whom most people probably do not know. The man in the picture on the left is another story – Herbert Hoover, 31st President of the United States. They have the same name, and are father and son.

Hoover, Jr., like his dad, was an engineer, and like his dad, went to Stanford. He developed a lifelong interest in radio, especially aviation radio. In 1928 he was hired by Western Air Express (later Western Airlines, now part of Delta Airlines) in Los Angeles to implement a radio network for their flight operations. By 1930 he was their chief engineer.

After leaving Western Air Express in the early 1930s, Hoover's interest in radio focused on exploration geophysics – the use of radio to prospect for petroleum. Oil had first been discovered in Southern California in 1876, and a large oil field was discovered in the Los Angeles basin in 1892. This stimulated the creation of technology businesses in the Los Angeles area to support the exploration for and extraction of oil.

In 1935 Hoover started a company in Pasadena, United Geophysical, to develop techniques for exploration of oil by seismology. A few years later, he founded Consolidated Engineering Corporation (CEC) to develop and manufacture instrumentation, again focusing on oil exploration. The company later changed its name to Consolidated Electrodynamics Corporation to highlight their main product line.

CEC went public in 1945, after which Hoover sold his interest in it. United Geophysical was eventually purchased by Union Oil. In the 1950s, Hoover began assuming diplomatic roles. In the Eisenhower administration, he served as a special envoy to Iran and Under Secretary of State for John Foster Dulles.

Hoover contracted tuberculosis as a young man and was never in robust health afterwards. He survived his father by only a few years, passing away after a sudden stroke in 1969.

Most of the information on this slide is based on the Wikipedia article, https://en.wikipedia.org/wiki/Herbert_Hoover_Jr.

Consolidated Electrodynamics (CEC)

- Instrumentation for seismic exploration
  - Sensors, recorders
  - **Mass spectrometer**, 1942
- Mass spectroscopy analyzes compounds
  - Goal is to determine chemical composition
  - Ionizes a sample – passed through magnetic field
  - Yields a spectrum of mass/charge ratios (m/z)
  - Spectrum analysis requires solving *simultaneous linear equations* (*n* equations with *n* unknowns)
  - It's a *lot* of calculations

2019 MCP 4059 4

CEC developed and manufactured a variety of instruments, including including various sensors and data recorders. Most, if not all, of this equipment prior to and during World War II was analog in nature, rather than digital.

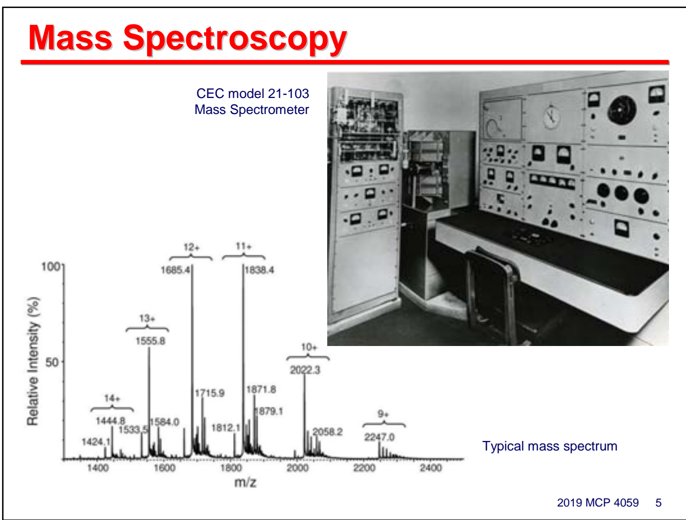
The Big Kahuna of CEC's products, however, was a device known as a *mass spectrometer*. The purpose of a mass spectrometer is analysis of a compound to determine its chemical composition.

It works by ionizing a portion of the sample compound and passing the resulting stream of charged particles through a strong magnetic field. The field spreads the stream into a spectrum based on the ratio of each particle's mass (m) to its ionized charge (z). The position in the spectrum of a particle's m/z ratio indicates what type of element or molecule fragment it is, while the intensity of the stream of particles at that position indicates the relative amount of that material.

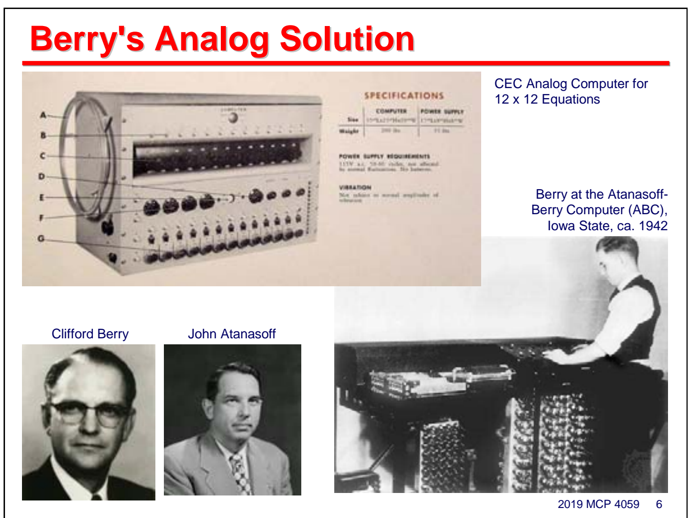
Analyzing the spectra coming from a mass spectrometer requires solving a set of simultaneous linear equations, i.e., solving *n* equations having *n* unknowns. Anyone who has solved such a problem manually for, say, three equations with three unknowns, knows that it's a lot of calculations. Mass spectrometry analysis usually involves significantly more equations and unknowns.

Prior to the advent of digital computers, calculations of this sort had to be done manually. If you were lucky, you had a mechanical calculator, which could add, subtract, multiply (slowly), and divide (even more slowly). Storage of intermediate results was done using pencil and paper. There was no automation. In fact, through the 1940s, the term "computer" referred to a person doing such calculations, not any such device for doing them.

For real-world mass spectrometry problems, the manual calculations required were extremely tedious and slow. Since the work was being done by humans, with lots of recording and reusing of intermediate results, errors in the calculations were common.



This slide shows a picture of the CEC model 21-103 mass spectrometer from the 1950s. The readings from this machine would appear as a chart of mass-to-charge ratio (m/z) vs. the relative intensity of the ion stream at each ratio value. This analog data would then need to be digitized manually to prepare it for further analysis.



CEC had a very smart guy working for them as Chief Physicist, Clifford Berry. Berry had a number of patents in the field of spectrometry. In 1945, he addressed the data analysis problem by designing for CEC an analog calculator that would solve systems of up to 12 equations in 12 unknowns.

A person solving a 12x12 system required about five hours of unrelenting toil using a mechanical calculator. The CEC model 30-103 "Electrical Computer" could do the same in about 44 minutes. With its power supply, it weighed 235 pounds.

Another reason for mentioning Berry and his analog calculator is to point out that he was the same Clifford Berry who worked as a graduate student with Professor Jon Atanasoff at Iowa State University (then Iowa State College) during 1939-1942 on what is arguably the first digital electronic computer in the United States, the Atanasoff-Berry Computer, or ABC. It was not a programmable computer in the sense we think of today, being designed like Berry's later analog device to solve systems of simultaneous equations. Nonetheless, it made significant advances in the design of circuits for electronic calculation.

The ABC is famous for another reason. John Mauchly visited Atanasoff at Iowa State in 1941 and viewed the ABC in its then-current state of development. In 1947, Mauchly and J. Presper Eckert filed a patent application on behalf of Sperry Rand, based on their work on the ENIAC, with broad claims covering digital electronic computing machines. The patent was awarded in 1964, but was then challenged by Honeywell. In 1973, a federal district court invalidated the Sperry Rand patent largely based upon the prior art of the ABC.

So you could say that Atanasoff and Berry were responsible for Sperry Rand being denied a patent covering just about all of digital electronic computing. The world would probably be a much different place today if that patent had been upheld.

---

http://www.tjsawyer.com/B205home.php

From Analog to Digital

- Analog computer worked, but was insufficient
  - Limited number of equations/unknowns (12 max)
  - Time-consuming, limited precision (~3 digits)
- CEC started researching digital computation
  - Initially intended to design a specialized calculator
  - Assumed 8 digits of precision adequate
  - Discovered customers did not want just a calculator
- CEC altered course to develop a full computer
  - Hired Harry Huskey to teach engineers digital logic
  - Hired Norwegian mathematician Ernst Selmer to design the arithmetic and control logic
  - Resulted in CEC 30-201, 30-202 prototypes (1952-54)

2019 MCP 4059 7

Back to Berry's 1945 analog Electrical Computer –

The analog calculator certainly worked, but it was quickly found to be insufficient for the needs of CEC's customers. A system of 12x12 equations was not all that large. Alas, the complexity of the circuitry grows for larger systems, so building significantly larger analog calculators was not practical. Getting results in 44 minutes was a lot better than getting them in five hours, but that was still pretty slow. Finally, as with most analog devices, you were lucky if you could get three digits of precision in the results.

Thus, around 1951, CEC started exploring the feasibility of building a digital electronic calculator to solve larger systems of equations. Their initial designs were oriented to a device that would support eight digits of precision.

Ensuing discussions with customers and potential customers uncovered two problems with this approach: (a) eight digits was not enough – most people wanted at least 10 digits, and (b) no one was much interested in spending a lot of money on just a calculator.

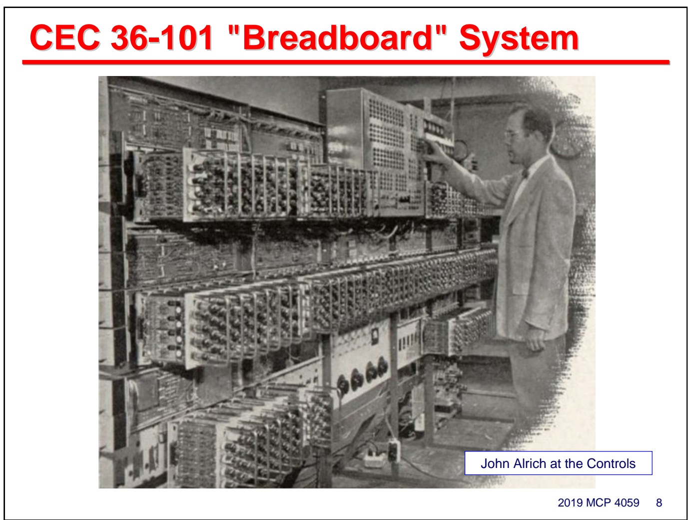
So, CEC went to Plan B and started thinking about designing a full-up, programmable, digital computer. They hired Harry Huskey, who had spent time in the UK working on some of the early British computers, and was currently working at the University of California at Los Angeles on a computer for the National Bureau of Standard, the SWAC. Huskey did not work on the CEC design, but presented lectures to the CEC engineering staff on digital design techniques.

They also hired the Norwegian Ernst Selmer, a number theorist, who was then a visiting lecturer at Caltech, as a consultant. He eventually ended up designing much of the arithmetic and control logic for the the computer.

During 1950-1954, this work produced a mock up, or "breadboard" system known as the 36-101, and two prototype models, known as the CEC 30-201 and 30-202. Based on these, a third version, the CEC 30-203, was developed as a product for sale to customers.

---

http://www.tjsawyer.com/B205home.php



This slide shows the initial "breadboard" system, used to test circuit designs and debug the control logic.

John Alrich was a young engineer who played a significant role in the development of the new CEC computer system. He also had a significant role in the implementation of the IBM 610 Auto Point, sometimes referred to as the world's first personal computer. IBM contracted with ElectroData to do portions its design and construction of the first prototypes. See http://www.columbia.edu/cu/computinghistory/610.html.

CEC → ElectroData → Burroughs

- CEC decided computers weren't their thing
  - Very capital-intensive, outside their main business
  - Spun off ElectroData as public corporation (1954)
  - Moved to 460 Sierra Madre Villa in Pasadena, CA
- ElectroData's success
  - Production model "Datatron 203" announced 2/1954
  - Models 204 (mag tape) and 205 ("Cardatron") by 1955
  - For a while, 3rd largest computer manufacturer in U.S.
- Financial pressures became overwhelming
  - Burroughs having trouble entering computer business
  - Offered to buy ElectroData in 1956
  - ElectroData became the "ElectroData Division"

2019 MCP 4059 9

Recall that CEC got into the computer business because their customers had lots of data coming from their mass spectrometer products that needed to be analyzed. They quickly realized that building, marketing, and supporting electronic computer systems was an entirely different business from instrumentation. In particular, it was extremely capital-intensive.

To extract themselves from this situation, they spun off their Computer Division in early 1954 as a separate, public corporation, retaining 36% ownership, and offering the rest on the American Stock Exchange, initially at $3.50 per share. The new company was named ElectroData. They moved to a new building at 460 Sierra Madre Villa in Pasadena, which became the home for Burroughs West Coast engineering and manufacturing for the next more than 40 years.

The CEC 30-103, so named by CEC's project numbering convention, was renamed the ElectroData "Datatron 203." Deliveries to customers started in early 1954. The initial 203 supported only paper tape input/output, plus a 10 character/second Flexowriter electric typewriter.

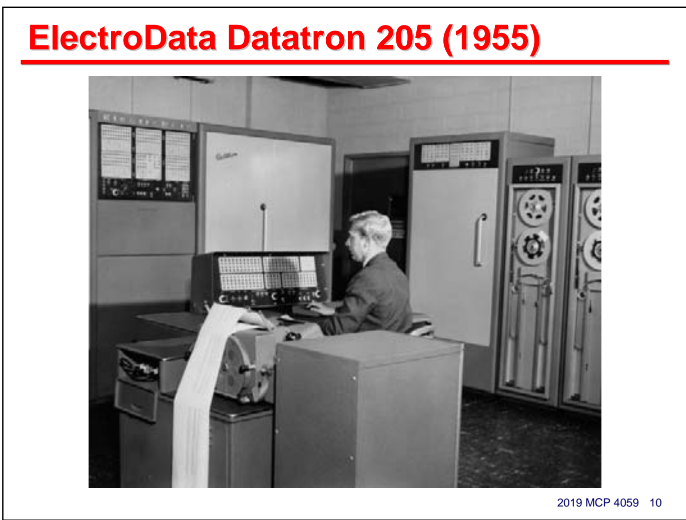
Extensions to the original design to support a few additional instructions plus magnetic tape (3/4-inch, 100 BPI, fixed-block, dual-lane recording) resulted in the Datatron 204 in 1955. Further extensions to implement the "Cardatron" interface for IBM punched-card tabulating equipment (the 089 collator, 523 summary punch, and 407 tabulator) resulted in the Datatron 205 in 1956. The entire series is often known as simply the 205 – and incorrectly as the B205. The B200-series systems were entirely different and did not appear until around 1960.

The Datatron 20x machines proved to be quite popular and sold well, competing with machines such as the IBM 650. For a while in the mid-1950s, ElectroData was the third largest manufacturer of computers in the United States. With this success, however, the same financial pressures that had driven CEC to spin off ElectroData intensified. By early 1956, management could not find a way to attract the necessary capital to continue operations, so began preparations to shut down the company and liquidate its assets.

It was at this point that Burroughs entered the picture. They had been struggling to transition from mechanical to electronic products and enter the commercial computer business. Just as ElectroData was preparing to throw in the towel, Burroughs made them an offer to purchase the company. The deal was finalized in June 1956, with the ElectroData Corporation becoming the ElectroData Division of the Burroughs Corporation.

---

http://www.tjsawyer.com/B205home.php



This slide shows a picture of a Datatron 205. The processor "main frame" with its maintenance panel is in the rear left. A man sits in front of the so-called Programmer's Console in the center with the paper tape equipment and Flexowriter in the foreground. In the rear right are tape drives and their controller cabinet.

As we shall shortly see, that Programmer's Console would later become famous by itself in an entirely different role.

Datatron 20x Details

- Vacuum-tube, decimal, drum memory
  - 4000 11-digit words, 8.4ms access time
  - 80 words, 0.84ms access ("high-speed loops")
  - 142.8 KHz clock rate
  - Digit-sequential operation internally
  - First index register in U.S. ("B register")
  - Optional hardware floating-point
- Peripherals
  - 203 – paper tape, Flexowriter typewriter
  - 204 – adds fixed-block, dual-lane magnetic tape
  - 205 – adds Cardatron buffered card interface to IBM tabulating equipment (089, 523, 407)

2019 MCP 4059 11

The Datatron 20x systems were fairly typical of vacuum-tube, decimal, drum-memory systems in the 1950s. Memory consisted of 11-digit words, each having ten decimal digits plus a sign digit.

The memory was partitioned into a 4000-word main portion, consisting of 20 bands of 200 words each, which had an average access time of 8.4 milliseconds – that's a memory access time, not an I/O time. The second portion of memory, known as the "high-speed loops" occupied four bands of 20 words each on the same drum. Using separate read and write heads and a special feedback loop between the heads, theses additional 80 words were available in an average of 0.84ms each. There were special instructions to move data between the main memory and the loops and to execute code from the loops.

The clock rate was 142.8KHz, and was determined from timing signals recorded on the drum. Data transferred to and from the drum in digit-sequential fashion, so internally all data was processed one digit at a time. The adder and all of the data paths were only one digit wide.

One architecturally significant feature was the four-digit "B" register. This was the first index register to appear in a U.S. computer system. When an instruction with a negative sign digit was executed, the contents of the B register were added to the instruction's four-digit operand address without affecting the instruction word in memory. The idea probably came from Harry Huskey, as some of the British machines on which he had worked earlier had so-called "B lines" that did a similar form of address modification.

A floating-point arithmetic module was developed later and offered as an optional component. It was housed in a separate cabinet next to the mainframe and could be added to a system in the field. With the B register and floating-point, the 20x systems became a popular choice for highway departments and engineering organizations needing a relatively inexpensive computer for design calculations.

As mentioned earlier, the 20x systems eventually supported input/output using paper tape, magnetic tape (3/4-inch, dual-lane, fixed-block, overwrite-in-place), and the Cardatron adapter for IBM card machines.

One interesting note about the Pasadena operation, given the era, was that it wasn't just guys. Sibyl Rock initially worked at CEC as a mathematician, problem analyst, and algorithm designer. She became something of a customer liaison, and was instrumental in helping refine the requirements for CEC's initial computer design. I suspect she was the one who ferreted out from customers that eight-digit precision was inadequate, and that nobody was very interested in just a calculator.

Another significant participant in the early CEC/ElectroData years was Gloria Bulock, an African-American originally from New York City. She was also a mathematician, and has the honor of not only being the first person to write a program for the 30-201 computer, but the first one to write one that *worked*. It was a program for computing prime numbers. She became involved in developing training materials and teaching customers how to program the Datatron machines, then ran the ElectroData Training Department for several years.

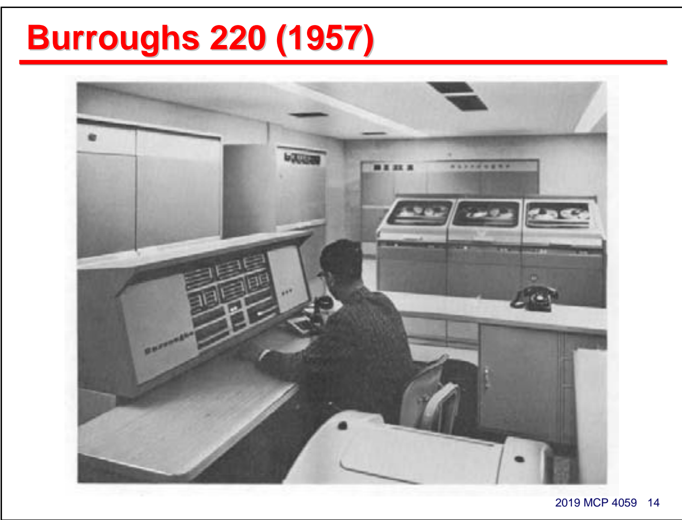
The Datatron machines had a long and useful life, but by the early 1960s they were obsolete and being replaced. The used machines found a second life, although not as computers, but rather as props in 1960s science fiction/spy movies and TV programs. Any fan of Lost in Space will recognize the Datatron Programmer's Console. It was also used as the Bat Computer in the 1960s Batman TV series. Other 205 components also made appearances, especially the tape drives, power supplies (with all those meter dials), and mainframe cabinets with the covers off and vacuum tubes exposed.

There is a whole web site devoted to the use of computer equipment as props. It has a section on the 205:

http://www.starringthecomputer.com/computer.html?c=45

There is also something of a cult that has sprung up around the 205 Programmer's Console, especially among Lost in Space fans. For about $2000 USD you can even buy a replica console. See

http://www.angelfire.com/scifi/B205/



With the relative success of the Datatron 20x machines and the new financial backing provided by the Burroughs acquisition, the ElectroData Division designed a new machine, which became known as the Burroughs 220. As we will see shortly, this system is to play a significant role in the origin of Extended Algol.

Burroughs 220

- Follow-on to the Datatron 205
  - Larger core memory replaces drum memory
  - Still vacuum-tube, decimal, internally digit-sequential
  - 200KHz clock (up from 143KHz)
- Burroughs trying to make strong showing in both commercial and scientific applications
  - Same 11-digit words, hardware floating-point
  - Sophisticated magnetic tape subsystem
  - Cardatron buffered punched-card interface
- Automatic Programming group in Pasadena
  - Developing assemblers and programming aids
  - Working on IBM-compatible FORTRAN compiler

2019 MCP 4059 15

The 220 started as an attempt to replace the 205's memory drum with core memory. That idea did not work out, so a team at the ElectroData Division, headed by Edward (Ted) Glaser designed an entirely new machine. Glaser was the chief logician for the design, and interestingly, was totally blind.

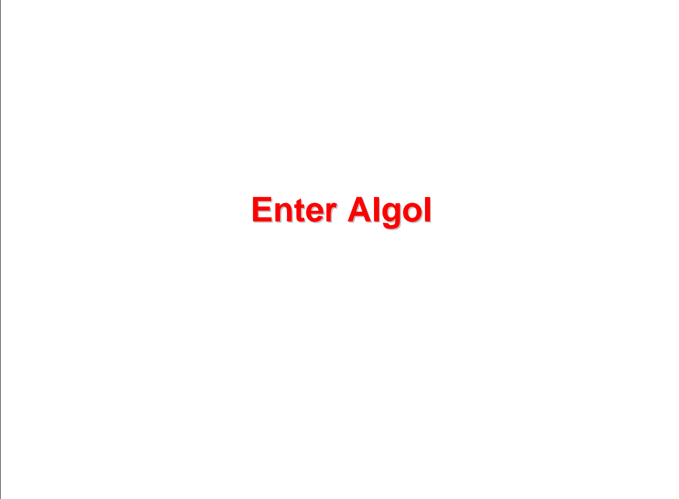
The new machine had a larger and much faster core memory than the 205's drum, up to 10,000 words in size, with an access time of 5 microseconds. The words were the same 11-digit size and format as for the 205. The 220 used a 200KHz clock rate (up from the 205's 142.7KHz), but internally the adder and all data paths were still only one digit wide. Nonetheless, it was a lot faster system than the 205.

The 205 had enjoyed a reasonable success in both scientific and commercial applications. With the 220, Burroughs was trying very hard to penetrate further into both markets. Floating point was now standard and integrated into the CPU. The 220 had a tremendously sophisticated magnetic tape subsystem, still using 3/4-inch tape with dual lanes and the ability to overwrite data blocks in place. The Cardatron interface for IBM tab equipment was slightly improved and offered with the 220 as well.

The ElectroData Division was beginning to recognize the role and value of software for their computer systems. They established a group within Marketing known as Automatic Programming. Programming at this time was considered to be the process of translating a program's design (using flowcharts, decision tables, and other higher-level expressions) into the instructions the machine would execute. Automatic Programming was simply an attempt to automate that step in the software development process. Today we would call it "compiling." They initially focused on symbolic assemblers and other low-level programming aids.

The 220 hit the market at about the same time that IBM released the initial implementation of FORTRAN. FORTRAN proved almost immediately to be highly popular, so Automatic Programming was given the task of developing a FORTRAN compiler for the 220. This project had an impressive set of requirements, principally that it be able to compile programs that would execute without error for *any* of the FORTRAN compilers for *any* of the IBM machines – apparently including those containing embedded machine code. This project was to trigger a change in direction in Pasadena that is central to our story.

Alas, the 220 was an interesting and productive machine in many ways, but the timing was bad. The 220 turned out to be the last of the major vacuum-tube computer systems, released at a time when other manufacturers were introducing transistorized designs. It did not sell all that well.



Thus far, we've talked about some company history for CEC, ElectroData, and Burroughs, and looked at some Really Old Iron, but we've hardly mentioned Algol at all. Now we will.

Programming Was Hard in the '50s

- Difficult machines, primitive tools
  - Lots of programming in absolute machine code
  - Simple assemblers began to appear
- Most computation was numerical
  - Scientific, engineering, mathematical problems
  - Growing interest in automatically translating standard math notation to computer instructions
    - Short Code, Schmitt & Mauchly (BINAC/Univac I, 1950)
    - AUTOCODE, Glennie (Manchester Mark I, 1952)
    - A-0, Hopper (UNIVAC I, 1952)
    - I.T., Perlis (Purdue University, 1955, Datatron 205)
    - FORTRAN, Backus (IBM, 1957, IBM 704)
  - Growing interest in exchanging programs among different computer systems

2019 MCP 4059 17

It is easy for us today to forget how difficult computer programming was in the 1950s. Common instruction set and input/output features we take for granted were still being worked out. Memory was slow, expensive, and not that reliable. Instruction sets were oriented more towards circuit efficiency than programmability and software support. Worse, the programming tools were extremely primitive – a lot of programming was originally done in absolute machine code. Simple symbolic assemblers began to appear by the mid-1950s, but compilers were virtually unheard of.

Most computation was numerical – it's why we call them computers, after all – and even non-numerical operations were implemented as manipulations of numerical values, just interpreted differently for purposes of input/output. Regardless, there were a large number of problems in science, engineering, and mathematics that generated a strong demand for automatic computation.

Because of this strong and growing demand for numerical computation, there was also growing interest in translating standard mathematical notation (or something close to it) automatically into machine instructions. The slide shows a list of some of the better known efforts in this area during the early/mid-1950s. These indicate the direction in which the Automatic Programming group at ElectroData was intending to go.

This growing interest in automatic translation and code generation culminated in the release of the FORTRAN language for the IBM 704 in 1957, after four years of development and about 25 labor-years of effort. It was an immediate success, as it brought the ability to program directly to ordinary scientists and engineers, bypassing the need in many cases for the analyst/designer/coder teams that had been required previously. The success of FORTRAN stimulated efforts to create compilers for the language all across the computer industry, including the ambitious IBM-compatible compiler project for the 220 at the ElectroData Division of Burroughs.

Another problem that users faced in the early/mid-1950s was that, if you had written a program for one computer system but then wanted to run it on another, even of the same manufacturer, you pretty much had to start over from scratch. It was difficult enough to write anything at all and get it to work to worry about standardization and compatibility. FORTRAN started to show some promise in this area, but FORTRAN was IBM's baby, built at great expense to sell *their* computer systems. The last thing IBM wanted was a standardized language that allowed you to run your programs on anyone else's products. Thus, standardization was something that was not going to come out of the computer industry by itself.

The International Algebraic Language

- 1955-1957
  - German GAMM society working on general computing and formula translation
  - Conference in Los Angeles on exchanging computer data and programs
    - ACM, SHARE, USE, DUO
    - Concludes a universal programming language very desirable
- 1958
  - GAMM and ACM meet to exchange proposals
  - Joint session in Zurich to resolve differences
  - Result is "Preliminary Report – International Algebraic Language" (IAL)
  - Becomes known as "Algol-58"

2019 MCP 4059 18

The desire of having a standard programming language that would work across different computer systems was particularly strong in academic and research institutions. The German mathematical society GAMM started working on one for general computing and algebraic formula translation as early as 1955.

Then in 1958, the Association for Computing Machinery (ACM) organized a conference in Los Angeles on ideas and techniques for exchanging computer data and programs between systems. Represented at the conference were SHARE (the IBM user group), USE (the Sperry Univac user group), and DUO, the Datatron User Organization (the Burroughs/ElectroData user group). One of the prime conclusions of the participants in this conference was that a standard, universal programming language was very desirable.

Following the ACM conference, GAMM contacted the ACM concerning unification of their two efforts, and the two agreed to meet in Zurich in 1958 to resolve differences in their proposals. The result of that meeting was a document, the "Preliminary Report—International Algebraic Language," published in the *Communications of the ACM* (vol. 1 #12, Dec. 1958, pages 8-22).

This preliminary description of a programming language – it was not yet a specification for one – launched a number of implementation efforts on several different machines, including the well-known JOVIAL and NELIAC variants in the United States, and one other we'll talk about in a few minutes.

As an aside, DUO merged a few years later with another Burroughs user group, CUE (Cooperating Users' Exchange) to form CUBE (Cooperating Users of Burroughs Equipment). Subsequent to the 1986 merger of Burroughs and Sperry, CUBE and USE merged in the mid-1990s to form the present UNITE organization. So the precursors of UNITE were very much at the table during the dawn of the development and standardization of algorithmic languages.

Algol-58 Example

```
procedure Simps (F(), a, b, delta, V);
          begin
      Simps:
          Ibar := V×(b-a);
          n := 1;    h := (b-a)/2;
          J := h×(F(a) + F(b));
      J1:
          S := 0;
          for k := 1 (1) n;
              S := S+F(a + (2×k-1)×h);
          I := J + 4×h×S;
          if (delta < abs(I-Ibar));
              begin
              Ibar := I;
              J := (I+J)/4;    n := 2×n; h := h/2;
              go to J1
              end;
          Simps := I/3;
          return;
          integer (k, n)
          end Simps

area := Simps(poly(), x, x+20, 2_10-5, 5_1025);
```

2019 MCP 4059 19

This slide shows an example of an Algol-58 program, taken from the Preliminary Report, that implements numerical integration using Simpson's method. It exhibits many of the signature features we recognize in modern Extended Algol, and a few that look a little strange.

The Preliminary Report proposed specifying the language at three levels:

- A Reference Language that defines the language.
- A Publication Language used to state and communicate algorithms (e.g., by publishing them in a journal).
- One or more Hardware Representations that are subsets of the Reference Language, allowing them to be handled by the character sets and input/output equipment of physical computer systems.

This example is expressed in the Publication Language.

Meanwhile, Back in Houston…

- Robert S. ("Bob") Barton
  - 1954 – Takes job with Shell Development Research
  - 1957 – Working with young team on "Shell Assembler" for the 205
  - 1959 – Leaves Shell for Burroughs in Pasadena

- Part of team ("the arthropods") follows Barton
  - Joel Erdwinn
  - Clark Oliphint
  - Dave Dahm (still a summer-student employee)
- Barton heads Automatic Programming group
  - Now responsible for the ambitious IBM-compatible FORTRAN compiler project for the 220

2019 MCP 4059 20

Let us set the work towards specifying a universal algorithmic programming language aside for a bit and turn to the person who, almost certainly more than any other, was responsible for Algol on Burroughs systems – Robert S. ("Bob") Barton.

Barton had a degree in mathematics from the University of Iowa and worked at the IBM Applied Science Department in the early 1950s. In 1954 he took a job with a subsidiary of Shell Oil Company, Shell Development Research, which had a Datatron 205 computer system. While there, he managed a group of talented, young programmers developing what became known as the Shell Assembler for the 205, which was quite an advanced macro assembler for its time – especially given the limited physical resources of the 205. In 1959, he left Shell to take a job managing the Automatic Programming group at the Burroughs ElectroData Division in Pasadena.

Part of the team of young turks who worked with Barton in Houston followed him out to California, most notably Joel Erdwinn, Clark Oliphint, and Dave Dahm (who initially was still an undergraduate student and only came out during the summer). Within the ElectroData Division, these young men were known as the "arthropods," because, well, they came from Shell.

After his arrival in Pasadena, Barton was fairly quickly put in charge of the highly ambitious IBM-compatible FORTRAN compiler project for the new 220 computer system. There is an amusing anecdote concerning this, told by Dave Dahm during the 1985 B5000 Oral History conference (p.27):

> I remember when I was a summer employee in the summer of 1959, I was working on the [BALGOL] compiler with another fellow named [Joel Erdwinn], who was the project leader. And we were busily trying to do our version of IAL, and one day Bob Barton came along and he had a FORTRAN manual in his hand. It was a really nice, well-done FORTRAN manual done by IBM. He said, "Why don't you guys do this language because then you wouldn't have to write a manual?" We rejected that as being not a very good reason for picking a language. [Laughter]
>
> So, basically, I would say that the decision, that the compiler we would do would be ALGOL as opposed to FORTRAN was made by a summer employee and a project leader. I don't know that anyone else was really involved in making that decision.

220 BALGOL Compiler

- Barton realizes FORTRAN project is impossible
- Arthropods abandon FORTRAN, start on Algol-58
  - Erdwinn, Dahm
  - Later – Oliphint, Merner, Crowder, Speroni, Knuth
  - Initial compiler released March 1960
- The Burroughs Algebraic Compiler
  - Officially, "BAC-220"
  - Better known as Burroughs Algol, or **BALGOL**
  - Follows Algol-58 more closely than other dialects:
    - JOVIAL (SDC)
    - NELIAC (Naval Electronics Lab, San Diego)
    - MAD (University of Michigan)
    - ALGO (Bendix)

2019 MCP 4059 21

Barton had quickly realized that the ambitious IBM-compatible FORTRAN project was impossible. Here is his anecdote in response to Dahm's (p.50 in the 1985 B5000 Oral History transcript):

> I want to correct Dave Dahm's statement about my trying to get them to do FORTRAN. [Laughter] It's correct to a certain extent in that the job that I had taken, under generally misleading conditions, called for doing an impossible FORTRAN which would also include conversion of assembly language from the 7090, or whatever the machine was at the time, automatically. I knew it couldn't be done, but that was my responsibility. [Erdwinn] would never have done a FORTRAN. I mean, he'd been going through this kind of educational experience at Shell, and he was not the sort of guy that would waste his time doing FORTRAN. He knew too much about language.

So it appears that it really was Joel Erdwinn, perhaps abetted by Dave Dahm and Bob Barton, who decided in 1959 that Burroughs should pursue Algol rather than FORTRAN. Regardless of who made the decision or how it came about, though, Barton and the Arthropods definitely dropped the idea of implementing FORTRAN and focused instead on a compiler for the Algol-58 language described in the IAL Preliminary Report published just the year before.

If there is one point in time where it can be said that Burroughs got hooked up with Algol, this is probably that point. It was such an important decision, apparently made in such an off-hand manner.

Their compiler was initially released in March 1960. It was officially known as the Burroughs Algebraic Compiler and styled as "BAC-220," but it soon became more popularly known as Burroughs Algol, or **BALGOL**. It was not completely conformant with the Preliminary Report, but it did follow the syntax of that report more closely than the other significant Algol-58 implementations, at least those developed in the U.S., including SDC's JOVIAL, the Navy's NELIAC, Michigan's MAD, and Bendix ALGO.

Joel Erdwinn was a highly gifted programmer. Barton says later in the 1985 Oral History conference that BALGOL was Erdwinn's masterpiece. Despite Barton's statement that Erdwinn would never waste his time doing a FORTRAN compiler, Erdwinn left Burroughs shortly after BALGOL was completed to take an important position at the then-new Computer Sciences Corporation (CSC), where he spent the next 20 years or so building compilers on a variety of machines for… FORTRAN! He was also instrumental in development of other system software that Sperry contracted CSC to build for their 1100-series systems.

BALGOL Example

```
2 COMMENT SIMPSON-S RULE$
2 PROCEDURE SIMPS(A, B, DELTA, V$$ F())$
2   BEGIN
2   INTEGER K, N$
2   IBAR = V(B-A)$
2   N = 1$
2   H = (B-A)/2$
2   J = H(F(A) + F(B))$
2 J1..
2   S = 0$
2   FOR K = (1, 1, N)$
2     S = S + F(A + (2K-1)H)$
2   I = J + 4H.S$
2   IF DELTA LSS ABS(I-IBAR)$
2     BEGIN
2     IBAR = I$
2     J = (I+J)/4$
2     N = 2N$
2     H = H/2$
2     GO TO J1
2     END$
2   SIMPS() = I/3$
2   RETURN$
2   END SIMPS()$
```

```
2 FUNCTION TORADS(X) = 3.1415926X/180$
2 FUNCTION DARCTAN(X) = 1/(X*2 + 1)$
2 PROCEDURE LOGISTICSIGMOID(X)$
2   BEGIN
2   LOGISTICSIGMOID() = 1/(1 + EXP(-X))$
2   RETURN$
2   END LOGISTICSIGMOID()$
2
2 SUM = SIMPS(TORADS(30.0), TORADS(90.0),
      0.00001, 2.0$$ SIN())$
2 WRITE($$ RESULT, F1)$
2 SUM = SIMPS(0.0, 1.0, 1**-5, 2.0$$
      DARCTAN())$
2 WRITE($$ RESULT, F2)$
2 SUM = SIMPS(0.5, 3.0, 1**-5, 2.0$$
      LOGISTICSIGMOID())$
2 WRITE($$ RESULT, F3)$
2
2 OUTPUT RESULT(SUM)$
2 FORMAT
2    F1(*SINE INTEGRAL =    *,X10.6,W0),
2    F2(*DARCTAN INTEGRAL = *,X10.6,W0),
2    F3(*LOGISTIC INTEGRAL =*,X10.6,W0)$
2 FINISH$
```

2019 MCP 4059 22

This slide shows an example of a BALGOL program, the same Simpson's integration algorithm shown earlier in the Publication Language. This version includes some functions (**`TORADS`**, **`DARCTAN`**, **`LOGISTICSIGMOID`**) that are passed as parameters to the **`SIMPS`** procedure for integration over a range of values.

The character set is limited by what was available on the IBM 407 card tabulator. That machine did not support the semicolon, so the dollar-sign was used as a statement delimiter instead. Similarly, the 407 did not support quotes, so the asterisk was used as a string quote in FORMAT statements. As in FORTRAN, parentheses were used both for grouping expressions and as array index brackets.

The "2"s running down the left edge of each card image were a requirement for the 220's Cardatron interface to the IBM punched-card equipment. They selected a "band" (formatting pattern) within the Cardatron to translate zone and numeric punches on the card to digit pairs representing character codes in the 220 core memory.

BALGOL Features Over Algol-58

- Input-Output
  - Free-field input of numerics, strings
  - INPUT/OUTPUT list declarations for READ/WRITE
  - FORTRAN-like FORMAT declarations for output
- Language features
  - Implied multiplication: `(X+Y)/2SQRT(Z)`
  - UNTIL iterative statement
  - OTHERWISE clause for EITHER IF statement
  - Generic type declarations
  - Initialization of arrays
  - Code segmentation with program-controlled overlay
  - MONITOR, TRACE, DUMP diagnostics
- Linkage to machine-language routines

2019 MCP 4059 23

Since what the Algol-58 Preliminary Report described was an initial version of the Publication Language, it was missing a few things, e.g., I/O. Like everyone else, Burroughs had to add some extensions to make it a practical language for use on actual computers.

One of the significant extensions was in fact for input/output. Out of the box, BALGOL would read source code and data from punched cards, output compiled code to magnetic tape, and print results to a 407 tabulator. Both the compiler and run-time library could be customized for other peripheral devices, however.

A program read data as sequences of free-form numeric and string values. The variables and expressions to be input or output were defined in **INPUT** or **OUTPUT** list declarations, which in turn were referenced from **READ** and **WRITE** statements. For output, FORTRAN-like **FORMAT** declarations defined the format and arrangement of values on a line. More exotic forms of input/output could be implemented using machine language routines added to the run-time library or included on cards placed after the BALGOL source deck.

BALGOL added a number of other features to those in the Preliminary Report:

- Implied multiplication: where it did not result in ambiguity, two expressions could be multiplied by simply writing them adjacent to each other, without a "*" operator, as is commonly done in standard math notation.
- The **UNTIL** statement, similar to the modern **DO** … **UNTIL**.
- The **OTHERWISE** clause for the **EITHER IF** statement. The Preliminary Report did not include any form of "else" for conditional statements – that had to wait for Algol 60.
- Generic type declarations: like FORTRAN, variables were not explicitly declared in Algol-58. The type could be specified according to the leading characters of the variable identifier.
- Code segmentation: the code for large programs could be divided into overlays, but they had to be explicitly loaded by the program a run time.
- Diagnostic facilities: **MONITOR**, **TRACE**, and **DUMP** declarations, somewhat like those in modern Extended Algol.
- The ability to link machine language routines to the compiled BALGOL code.

BALGOL Operational Advantages

- Fast, single-pass compiler (mag tape-based)
- Optimized for compile-and-go environment
- Configurable compiler environment
  - Generator program → compiler tape
  - Customize device types and I/O routines
  - Use larger memories (min 5000 words)
  - Augment/replace standard library
- Save and rerun object programs
  - Mag tape
  - Punched cards
  - Paper tape

2019 MCP 4059 24

Perhaps more important than its language features were some operational features that proved to be very popular with 220 sites, especially at universities and research organizations who wanted to provide convenient computing resources for their students and research staffs.

- It was a fast, single-pass compiler that was often limited by the speed of the punched-card peripherals. The compiler and library resided on one magnetic tape, and the compiled code plus run-time library routines linked into the compile code were written to a second magnetic tape.
- It's operation was optimized for a compile-and-go environment. After a successful compilation, the computer operator simply had to press **START** on the system console to run the program.
- The compiler could be configured to the memory size and peripheral devices available on a given system. Although not available with the initial release, by 1961 it included a Generator program that could customize the compiler to a particular environment. It could also amend the run-time library with new or modified machine-language routines.
- Although the compiler wrote the executable program to magnetic tape, the release included utilities to dump and load programs to and from punched cards and magnetic tape. Since the compiler was so fast (for its day), many users with small to medium-size programs did not bother saving the object code – they simply recompiled their program from source each time.

Impact of BALGOL

- Proved value of compiler operational efficiency
  - Fast, one-pass compilation
  - Compile-and-Go environment
  - Monitoring and debugging aids
- Made the case for *regular use* of higher-level languages over assembly language
- *Customers loved it*
- Convinced Burroughs that Algol was viable
  - Planners believed it would displace FORTRAN
  - Showed that a different architecture was needed
- Provided much basis for design of B5000

2019 MCP 4059 25

While the Burroughs 220 was not a very successful product from a sales and revenue standpoint, BALGOL had a very significant impact on such success as the 220 had. It also helped put Burroughs on an architectural path that it almost certainly would otherwise not have taken.

As mentioned on the prior side, the operational benefits of fast compilation, a compile-and-go orientation, and the availability of monitoring and debugging aids in the language made it a productive and easy-to-use tool. While the IBM FORTRAN compilers generally concentrated on producing the best possible object code and the fastest possible run times, they did so at the cost of very long compile times and somewhat laborious operating procedures. You could perhaps classify the IBM compilers as built for object run-time performance, and BALGOL as built to Get Stuff Done.

Even after the advent of IBM FORTRAN, compilers were still held somewhat in suspicion, and Real Programmers still wrote using assembly language, or if you were a hard-core Real Programmer, in raw, seething machine code. While BALGOL did not generate highly optimized code, what it generated wasn't bad, and the convenience and speed with which you could usually get useful results helped make the case for using higher-level languages over assembly language and machine coding.

Importantly, customers loved it. BALGOL gave Burroughs a reputation for software prowess that it had not had previously.

Even more importantly, BALGOL convinced the product planners, engineers, and software developers within the ElectroData Division that Algol was a viable language and had great potential. Many of them believed the superior features of Algol would cause it to replace FORTRAN. It also showed them that while BALGOL worked pretty well on the 220, Algol in general needed architectural support in the hardware and would benefit from a different architectural model. As such, it provided a lot of the basis for the design of the next large Burroughs system, the legendary B5000.

In the end, the belief that Algol would replace FORTRAN proved to be naive. That was probably inevitable, given the size of IBM and the muscle it was willing to bring to the market place. But looking back from today, when we can see the influence that Algol has had on almost all procedural languages developed since that time, many of the ideas introduced in Algol did replace those introduced in FORTRAN.

Then There Was the *Other* BALGOL…

- Burroughs wanted a compiler like BALGOL for the 205
- They contracted with a Caltech grad student to write a compiler for $5500
- Donald Knuth wrote it *over the summer* in 1960
- Knuth continued to consult with Burroughs while at Caltech, until moving to Stanford in 1968
  - Worked on BALGOL (and wrote the comments)
  - Wrote first memory allocation scheme for the B5000

Donald Knuth

2019 MCP 4059 26

BALGOL has always been closely associated with the Burroughs 220, but it's not well known that a second BALGOL compiler was written.

About the time the initial version of 220 BALGOL was released in 1960, the ElectroData Division decided they should have a similar compiler for the 205. The 205 by this time was getting old, but there were still a number of them in service, and were still well-regarded by their users.

After rejecting a pricy proposal from TRW, Burroughs entered into a contract with a student preparing to enter graduate school at Caltech. The price was $5,500 USD. The name of the student was Donald Knuth. He wrote the compiler *over his summer break* in 1960. Knuth continued to debug the compiler until December of that year, but it worked, and and was distributed to customers. Given the size and relatively difficulty of coding a large program like this for the 205, it remains a significant programming feat.

After getting his PhD at Caltech, Knuth continued to consult for Burroughs until he left for a position at Stanford University in 1968. He made contributions 220 BALGOL (including writing all of the comments). He also worked on the B5000, writing its first dynamic memory allocation scheme.

This brings us up to the early 1960s. Burroughs had had a disappointing success with its presumed flagship system, the 220. A couple of follow-in designs were proposed, but never reached the hardware stage.

Despite disappointing sales of the 220, the success, popularity, and remarkable ease-of-use of 220 BALGOL had a marked impact, not only on customers, but on many in the product planning and engineering groups of the ElectroData Division. The 220 had been a poor architecture on which to implement Algol-58, and was even less well-suited for the next version of the language, Algol-60. Largely driven by the forceful influence of Bob Barton, these groups began to think about what they could do about that.

By August 1960 they had a functional design for a pair of systems called the 4000/4400. That design included many of the features we would recognize today – Polish evaluation of expressions, automatic segmentation of code and data, and operation centered around a stack. Actually, this design had two stacks, one for expression evaluation and local operand storage, and one for procedure call history.

The 4000/4400 never advanced out of the preliminary design phase, but its ideas morphed into the system that most people think of as the origin of Algol for Burroughs systems – the B5000.

Burroughs B5000 / B5500

- Radical departure in hardware architecture
  - Specifically designed for Algol-60
  - Stack-oriented operation, code & data descriptors
  - Hardware support for Call-by-Name ("thunks")
  - Automatic segmentation & overlay ("virtual memory")
  - Multiprogramming & multiprocessing (2 CPUs)
  - Comprehensive operating system (MCP)
  - Programmed exclusively in high-level languages
- Reintroduced as B5500 in 1965
  - Large, fast Head-per-Track disk subsystem
  - Several new instructions, mostly for MCP use
  - Ancestor of B6x00/7x00, A Series, ClearPath MCP

2019 MCP 4059 28

At the time it was introduced, and for many years afterward, the B5000 represented an astonishing departure from traditional computer system architecture and design.

- It had been designed specifically to support Algol. The Algol-60 dialect was not finalized and published until several weeks before the first customer delivery in April 1963, and in some areas the hardware did not support all the requirements of the Algol 60 Revised Report, so in some sense it was an Algol-58-and-4/5s machine. I think of the B5000 as the first computer system actually designed to run software.
- It supported stack-oriented operation, with the dual stacks of the 4000/4400 merged into one.
- One of the most challenging features of Algol-60 was its call-by-name semantics, which required dynamic evaluation of expressions passed as procedure parameters each time they were referenced. The B5000 included hardware features to make this efficient and easy to implement in compilers.
- Automatic code and data segmentation were fully implemented, with hardware support (code and data descriptor words, presence-bit interrupts) to automatically detect non-resident segments and signal the operating system to dynamically allocate memory and load them into memory. We now use IBM's clever term "virtual memory" for this, but there has never been anything virtual about it.
- The entire system was to be controlled by a comprehensive operating system, which became known as the Master Control Program, or MCP. It took a while, and a few architectural enhancements to get all of the features implemented and working well, but this goal was achieved.
- Perhaps most astonishing for its time, all programming for the system was to be done in higher-level languages, even the operating system and other system software. There was no assembly language, and most users needed to know nothing about the internal characteristics of the system. With the late addition of an awkward and ill-fitting character manipulation mode, the B5000 supported COBOL as well as Algol, and eventually FORTRAN and BASIC compilers were written.

With the benefit of some experience in the field, Burroughs introduced an enhanced version of the system, named the B5500, in 1965. The processor had several new instructions, most of them used only by the MCP. Perhaps the most significant enhancement was a large, fast Head-per-Track disk subsystem to replace the dual 32K-word drums on the otherwise tape-based B5000. This disk subsystem finally gave the system the storage capacity it needed to fully support its automated segmentation/overlay and multiprogramming/multiprocessing capabilities.

Writing Algol in Algol

- If the compiler is written in itself…
  - How do you compile the compiler?
- B5000 method – *bootstrapping*
  - Defined a temporary implementation language: OSIL
  - OSIL used for B5000 MCP and Algol compiler
  - Assembler-like processor
  - Generated B5000 code, *but ran on the Burroughs 220*
- Wrote two compilers, side-by-side
  - Official one in Algol, then hand-compiled into OSIL
  - Debugged and updated both versions in parallel
  - Once the OSIL version could compile Algol – then the Algol version could compile itself

2019 MCP 4059 29

I mentioned that the B5000 was designed specifically to support Algol, and that all programming was to be done in higher-level languages. This requirement extended to the Algol compiler itself. So just how do you do that? If the compiler it to be written in its own language and there is no existing compiler for that language, how do you compile the compiler?

The answer for the B5000 was a technique known as *bootstrapping*. The engineering team created a temporary implementation language known as the Operating System Implementation Language, or OSIL. It was to be used only within the engineering groups for the B5000 MCP, Algol compiler, and probably some diagnostic tools. It's coding format was similar to assembly language. OSIL generated B5000 machine code, but it did not run on the B5000. It ran instead on the Burroughs 220.

What the Algol team did was a very careful design of the compiler with complete flowcharts. They then wrote two versions of the compiler, one in Algol (which they no way to compile initially) and one which they hand-translated from the Algol version to OSIL. When emulators and actual hardware for the B5000 started to become available, they began debugging the OSIL version. As corrections were applied to the OSIL version, equivalent corrections were applied to the Algol version. Eventually they started trying to compile the Algol version of the source with the OSIL version of the compiler. Once the OSIL-generated compiler could successfully compile and run the Algol version of the compiler source, then the Algol version could compile itself and OSIL was set aside.

> *Note:* In the presentation at the UNITE conference, I said that OSIL was like a three-address macro processor and that one did not write B5000 instructions directly with it. That was my memory from a brief encounter with an OSIL manual in 1970.
>
> Some research I did at the Charles Babbage Institute (located in the Andersen Library at the University of Minnesota) shortly after the conference uncovered a document describing OSIL and what appears to be a listing of a short OSIL program. From these it appears that OSIL was much closer to a typical assembler, and that one really did code B5000 instructions using it. That makes it a much lower-level tool than I had originally thought.
>
> Given that OSIL ran on the 220 rather than the B5000, that it was intended only as a tool to write the initial B5000 Algol compiler and MCP, and that it was discarded as soon as the Algol and ESPOL compilers had been developed, I stand by my statement that the B5000 and later systems never had an official assembler.

System Programming in a HLL

- The surprise when Algol could compile Algol…
  - Original compiler was about 8000 lines of Algol
  - OSIL "compile" on the 220 took ***9 HOURS***
  - Algol compile on the B5000 took ***4 minutes***
  - Algol-generated codefile was smaller, too
- Writing Algol in Algol worked so well…
  - Needed to rewrite MCP for the new HPT disk
  - MCP team took the Algol compiler, and…
    - Ripped out all the stuff for storage allocation, I/O, etc.
    - Added a few low-level features for hardware control
  - Called the result **ESPOL**
- MCP systems have never had an assembler

2019 MCP 4059 30

Getting a compiler written in Algol to compile itself was obviously quite an accomplishment, but the really interesting part of the story is that compiling the OSIL version of the compiler on the 220 took *nine hours*. The first time the Algol version (consisting of about 8000 lines of code) could compile itself on the B5000, it took *four minutes*. The code it generated was smaller than the hand-optimized OSIL-generated code as well.

OSIL was also used to write the initial MCP for the B5000. Apparently it was not a pleasant task. When the Head-per-Track disk became available around 1964, the decision was made to completely rewrite the MCP to take advantage of the new disk's capabilities. In addition, the leading programmers from the Algol group transferred into the MCP group, and given how well writing Algol in Algol eventually worked out, there was no way they wanted to use anything remotely as primitive as OSIL for the new MCP.

Unfortunately, the Algol compiler had been designed to run under the MCP, and required the MCP both for its own operation and that of the programs it compiled. So the language experts now in the MCP group took the Algol source and in about three weeks ripped out all of the stuff that required MCP support – automatic storage allocation, I/O, support calls, etc. Then they added to this stripped-down compiler the few things they thought they needed for coding an operating system to get close enough to the hardware. They called the resulting language and its compiler the Executive System Problem Oriented Language, or ESPOL.

Thus, MCP systems have never had an official assembler. OSIL was completely abandoned as soon as ESPOL was available. All system software, even low-level card-bootstrap routines, were written in ESPOL. All software that ran under control of the MCP, including compilers and utilities, were written in Algol or one of the other languages. This idea has been carried forward since – ESPOL migrated to the follow-on B6500/6700 system and was eventually replaced by a new language, NEWP. The idea even migrated to the B3500/Medium Systems line, which had started doing system-level programming in assembler, but later developed their own high-level systems language, BPL.

Issues with B5000 / B5500 Algol

- Array rows limited to 1023 words
- Lexical scope addressing
  - Cannot address intermediate nested environments
  - Can address outer-block and local-procedure only
- Character manipulation
  - B5x00 used high-order bit in word as a "flag" (tag)
  - Flag bit indicated control words (descriptors, etc.)
  - Implemented Word-Mode and Character-Mode states
    - Character-Mode originally intended to support COBOL
    - Implemented in Extended Algol as **Stream Procedures**
    - *Ignores flag bits and all memory address protection!*
    - Extremely useful – extremely dangerous
    - Prompted development of `POINTERs`, `SCAN`, `REPLACE`, etc.

2019 MCP 4059 31

As astonishingly new and capable as the architecture of the B5000/5500 was, it had some limitations and sub-optimal characteristics.

First, the automated data segmentation features caused arrays in Algol to be allocated in memory row-wise. A single-dimension array was allocated as one contiguous area in the MCP's memory heap. All rows for a two-dimensional array were allocated individually in the heap, with a "dope vector" of descriptor words pointing to the location of each row. Higher levels of dimensionality created tree structures of dope vectors with the data rows at the leaves of the tree. This approach had a lot of advantages, because each row could be allocated individually, and in fact they were not allocated physically until first referenced. The rows could also be overlaid to disk and rolled back in individually as well.

The problem was that the rows were limited by the size of fields in the descriptor control words to a length of 1023 words each. There were ways to get around this in Algol using DEFINEs (which we'll discuss shortly), albeit at the cost of adding extra dimensions to the array and having to partition index values for access to each element. The COBOL and FORTRAN compilers implemented support for this behind the scenes. This approach worked, but it was not very efficient.

Second, the architecture of the B5000/5500 had some very innovative support for the semantics of addressing variables in Algol, but the designers missed one important one – the hardware could address variables local to a procedure and in the outer block (global environment) of a program, but they could not address any intermediate lexical scopes between the two. This could be a real problem if you chose to nest procedures within each other. The next slide illustrates this issue.

Third, Algol was conceived as a language for numerical computation. It had essentially no facilities for manipulating characters or strings of characters. In addition, the B5000/5500 used the high-order bit in its 48-bit word as a "flag" bit to signal the word was a control word. This made it difficult (and usually fatal) to pack a full eight 6-bit characters into a word. This made the system a poor fit for the commercial market Burroughs was desperately trying to reach.

The solution, made late in the design, was to tack a fairly crude "character mode" onto the side of what otherwise as quite an elegant architecture. It was intended primarily to support COBOL, but was exposed in Algol through a construct known as a Stream Procedure. This capability proved to be extremely useful in Algol, but also extremely dangerous, as it required disabling all checks for flag bits and most memory address protection.

Lexical Scoping Example

```
begin comment Knuth's Man-or-Boy Test;
    real procedure A (k, x1, x2, x3, x4, x5);
    value k; integer k, x1, x2, x3, x4, x5;
    begin
        real procedure B;
        begin k:= k - 1;
              B:= A := A (k, B, x1, x2, x3, x4);
        end B;
        if k <= 0 then
            A:= x4 + x5
        else
            B;
    end A;

    file dc (kind=remote, units=characters, maxrecsize=72);
    write (dc, <"Result = ",j11>, A (10, 1, -1, -1, 1, 0));
end.
```

Not allowed on B5000/5500

Note: Result = -67; run with STACK=9000

2019 MCP 4059 32

The problem the B5000/5500 had with lexical scoping is best illustrated by an example. This is Donald Knuth's well-known "Man or Boy Test" for Algol compilers, which here is written for the B6700/modern MCP dialect of Extended Algol, since you couldn't do this example in B5x00 Algol.

The colored boxes show the lexical nesting of address environments in the program. The outermost light-blue box represents the outer block or global environment of the program. The light-yellow box within that represents the body of the global procedure **A**. The light-green box within **A** represents the body of the nested procedure **B**.

Note that **B** calls **A**, passing parameters **k**, **B** (itself), **x1**, **x2**, **x3**, and **x4**.

The reference from the body of **B** to **A** is valid, because **A** is in the outer block of the program, and is therefore addressable by the hardware.

The reference from the body of **B** to **k**, however, is not valid for the B5000/5500, because **k** is in neither the outer block nor **B**'s procedure body. The B5000/5500 Algol compiler would allow blocks and procedures to be nested this way, but would issue syntax errors for attempts to address intermediate lexical environments.

This limitation was fixed on the B6500 and later systems by introduction of the so-called "D" (Display) registers to enable addressing directly to the intermediate lexical environments.

Stream Procedure Example

```
INTEGER STREAM PROCEDURE GETCHAR(A, OFFSET);
  VALUE OFFSET;
  BEGIN COMMENT
      RETURNS THE CHARACTER CODE AT THE LOCATION OF "A" OFFSET BY
      "OFFSET" CHARACTERS;
  LOCAL REP;                      % HOLDS DIV-64 REPEAT COUNT
  SI:= LOC OFFSET;                % SOURCE IS ADDRESS OF OFFSET WORD
  SI:= SI+6;                      % ADVANCE SOURCE BY 6 CHARACTERS
  DI:= LOC REP;                   % DEST IS ADDRESS OF REP WORD
  DI:= DI+7;                      % ADVANCE DEST BY 7 CHARACTERS
  DS:= CHR;                       % MOVE OFFSET 7TH CHAR TO REP 8TH
  SI:= A;                         % SOURCE IS ADDRESS IN A
  REP(SI:= SI+32; SI:= SI+32);    % SI:= *+ (OFFSET DIV 64)x64
  SI:= SI+OFFSET;                 % ADVANCE SOURCE BY (OFFSET MOD 64)
  DI:= LOC GETCHAR;               % DEST IS ADDRESS OF GETCHAR RESULT
  DS:= 7 LIT "0000000";           % CLEAR HIGH-ORDER 7 CHARS OF RESULT
  DS:= CHR;                       % MOVE SOURCE CHAR TO 8TH OF RESULT
  END GETCHAR;
```

Equivalent in B6700 Algol to: **`REAL(A[OFFSET],1)`**

2019 MCP 4059 33

Probably every B5000/5500 Algol programmer had a love/hate relationship with stream procedures. On the one hand, you could do extremely interesting and useful things with them – efficiently move large numbers of characters or words from one location to another, parse and concatenate character strings using rules of arbitrary complexity, convert between binary and decimal representations, and manipulate bit strings within or across word boundaries. You could also do I/O, but it wasn't easy.

On the other hand, it was a low-level notation, just one step above a free-form assembly language. To do all but the most trivial things, you had to know quite a bit about the operation of "character mode" within the processor. There were lots of restrictions, like modulo-64 counts and offsets, that did not make any sense unless you knew how the registers worked in this mode. Worst of all, you were working with absolute memory addresses, whether you realized it or not, and all of the bounds protection available in the "normal mode" that the rest of Algol used was disabled in character mode. People could (and did) create havoc with these routines.

Despite their difficulties and dangers, Stream Procedures gave Extended Algol programs powerful and efficient text processing capabilities that were simply not available in other dialects of Algol, were difficult or impossible to do in COBOL, and that were agonizing and inefficient to do in FORTRAN. Overall, Stream Procedures enhanced the capabilities of the system and of Extended Algol as a language that could be applied to a wide variety of problem domains.

The other thing that Stream Procedures did was to show that a better way was needed. This led to the safer and more powerful concept of character string pointers, the **`SCAN`** and **`REPLACE`** constructs we have in modern Extended Algol, and the architectural support for them in the hardware.

The example on the slide is roughly equivalent to the **`REAL(`**<pointer expression>**`,`** <arithmetic expression>**`)`** construct that was introduced in B6500/6700 Algol. It extracts a single character from an array row, offset some number of characters from its beginning, returning the numeric value of the character as an arithmetic expression.

The final system to consider in our story is the B6500, which was first shipped to customers in 1969. The B6500 had a very difficult beginning, in terms of both hardware and software reliability. All systems in the field were upgraded in 1971 with a re-engineered processor module, and the system was relabeled the B6700. A larger and more powerful system, the B7700 was released in 1973. Further revisions through the 1970s resulted in the B6800/7800 and B6900/7900 systems.

Burroughs B6500/6700/7700

- Solved all major problems with B5500 Algol
  - Lexical scoping (32 "D" address registers, now 16)
  - Moved flag bit to separate 3-bit "tag" field
  - String instructions (**POINTERS**, **SCAN**, **REPLACE**)
  - Longer array rows (orig. 2^{20}, now 2^{32} words)
  - Segmented arrays, resizing of arrays
- Other Algol extensions
  - Powerful and convenient sub-tasking capabilities
  - DOUBLE, COMPLEX data types
  - Eventually – object-oriented structures
- Enhanced over time to become A Series and current ClearPath MCP systems

2019 MCP 4059 35

The B6500/6700/7700 were the direct successors to the B5500. Burroughs took all of the lessons they had learned from the B5000/5500 and produced an entirely new architecture. It was closely related to the B5500 architecture, and carried forward all of its significant concepts, but generalized and expanded on them.

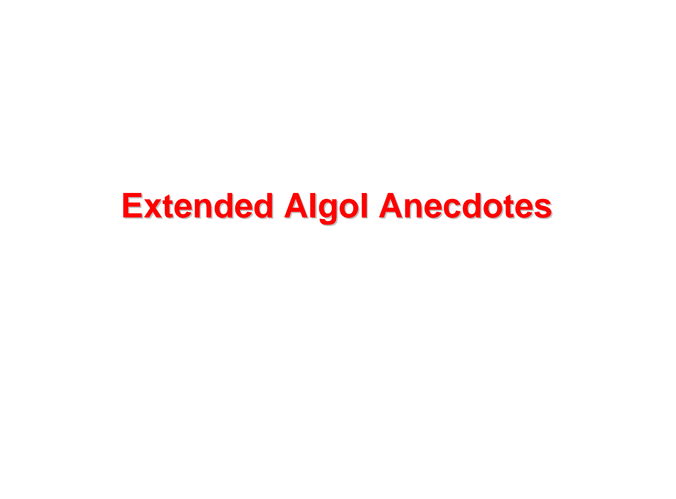
This new architecture also solved all of the major problems the B5500 had suffered.

- The problem of addressing intermediate lexical scopes was solved by the introduction of the "D" (Display) address registers to point to all of the currently-active scopes of a program. Initially there were 32 of these, allowing 32 levels of nesting, but for once that was too many, and later systems reduced the number to 16.
- Stream Procedures and all of their dangers were replaced by new instructions and Algol constructs (**POINTER**, **SCAN**, **REPLACE**, etc.) to provide powerful and efficient character string manipulation.
- Array rows could be much longer – up to 2^{20} words on the B6700 (and 2^{32} words on modern systems).
- Long array rows could be segmented, effectively moving the double-indexing tricks used on the B5500 into the hardware. Array rows could also be dynamically resized, either up or down.

There were many other extensions to Algol, too numerous to detail here, but a few significant ones were:

- The B5500 had a mechanism for a task to fire off other tasks (ZIP, still in modern Algol, and similar to a WFL **START** statement), but there wasn't a good way for the initiator to monitor and control those other tasks. The B6500 generalized the concept of a task and extended the Algol lexical scoping mechanism to include dependent sub-tasks. It also implemented locking and synchronization primitives to allow inter-program communication and control.
- The B6500 had better support for double-precision arithmetic than the B5500. Based on that, Algol received **DOUBLE** and **COMPLEX** data types.
- Eventually some object-oriented features were added to the language – dynamic linked libraries, **STRUCTURE BLOCK**s and **CONNECTION BLOCK**s.

The B6500, et al, have continued to be enhanced over time, producing the A Series and the current ClearPath MCP systems. There have been lots of extensions and improvements, especially in memory addressing and capacity, but it is still basically the same architecture that was released in 1969.



Thus the path that Extended Algol has taken within the Burroughs environment from its beginnings within the GAMM and ACM committees, through early implementations of Algol-58, to its current state as a highly enhanced variant of Algol-60.

At this point, I would like to take a half-dozen of the extensions that have made Extended Algol such a useful and long-lived tool, and discuss how they came about.

The DEFINE

- Richard Waychoff, 1961
  - One of original B5000 Algol compiler authors
  - Was discussing symbol table design with Don Knuth
  - Knuth thought for a minute and said,
    "With that organization of a symbol table, you can allow one symbol to stand for a string of symbols."
- Originally, DEFINE was non-parametric
  - Dave Dahm implemented parametric DEFINEs in the late 1960s

2019 MCP 4059 37

If there is one feature of Extended Algol that has made it a language for much more than implementing numerical algorithms, it is the **DEFINE** construct. It really has nothing to do with Algol – it's more of a compiler feature that allows the programmer to give a name to a string of syntactic tokens. By writing the name in source code, the compiler replaces it at compile time with the string of tokens it represents. This means that **DEFINE**s are a form of macro instruction.

The **DEFINE** was introduced very early in the development of the original B5000 compiler. Richard Waychoff, one of the authors of the B5000 compiler, has written a very lively memoir of the B5000 project (see References). In it, he describes how the **DEFINE** declaration came about, during a discussion in mid-1961 with Donald Knuth, who was still a graduate student at Caltech and consulting with Burroughs part time:

> I was happy to see don again and launched into a description of the Algol Syntax Chart, Recursive Descent and the separation of the three functions; scanning, parsing, and emitting. don was favorably impressed. I was especially proud of my knowledge of the B5000 Character Mode. It had led me to an organization of the symbol table that seemed much better than the other organizations with which I was familiar. So I described that with more enthusiasm than the other subjects. don put the index finger of his right hand to his lips, closed his eyes, and went into hyperspace for about 30 seconds. When he came back, he said, "With that organization of a symbol table, you can allow one symbol to stand for a string of symbols." I thought that that was the best idea that I had heard in a long time. I carefully put together my arguments and all of the necessary details and presented it to Lloyd. He thought that it was a good idea. So it was immediately a part of B5000 Algol.

So, for better or worse, Knuth gets the credit for the **DEFINE**. Originally, it was non-parametric – just an identifier standing for a string of symbols.

Waychoff goes on to describe a nasty debate that arose a couple of years later, apparently over the fact that unlike procedure declarations, the scope of a **DEFINE**'s symbols is the context of its use, not its declaration. Dave Dahm in particular objected to this and campaigned to have the feature removed from the language. The battle raged for days and was only resolved when Lloyd Turner (the project leader) did so by fiat. The **DEFINE** stayed.

Dahm must have gotten over his objections to the **DEFINE** at some point, because some years later he implemented the parametric version.

Dollar Cards and Sequence Numbers

- It all started with a bad B5000 card reader…
  - As Algol grew above 2000 cards
  - Compilation from cards became a nightmare
  - So, Waychoff and Bobby Creech went to a bar…
    - Worked out a scheme to keep source on tape
    - Set aside columns 73-80 for sequence numbers
    - Merged tape with correction cards by sequence number
- Created $-card to signal source mode, e.g.,
  - `$ CARD`
  - `$ TAPE`
- Later, more options added
  - Listing control
  - Void cards from tape
  - Resequence, create NEWTAPE, etc.

2019 MCP 4059 38

Another feature in Extended Algol that is a compiler thing rather than a language thing is the "dollar card," now officially termed a CCR (Compiler Control Record) and on other systems a "pragma." These records in the source file begin with a "**$**" and supply information to the compiler about the program being compiled or how the program should be compiled.

As Richard Waychoff tells it in his memoir, dollar cards came about because, of all things, a bad card reader. The Algol source for the original B5000 compiler was initially maintained on punched cards. The B5000 used by the Algol team had an early card reader apparently based on the design of the excellent Burroughs check readers. Alas, the device had a bad habit of frequently either refusing to feed cards properly or mangling them beyond usability.

As the size of the compiler grew above 2000 cards, compiling the compiler became nearly impossible due to the interruptions and damage to cards caused by the bad reader. The first solution was to copy the card deck to magnetic tape and compile from tape. Of course, whenever a change needed to be made to the compiler source, they had to correct it in the card deck and go through the agonizing process of copying the deck to tape using the troublesome reader.

So in the great tradition of software innovation, Waychoff and another programmer, Bobby Creech, walked into a bar. Over steins of beer, they discussed the problem and came up with the idea of reading cards that consisted only of changes, and having the compiler merge those cards with the source tape during compilation. This wasn't actually that much of an innovation, as tape-based master file update programs had been doing the same thing since the Univac I, but it was apparently the first time someone had thought of the source code as a master file and corrections to the program as transactions against the master. To control the merging, they assigned columns 73-80 on the card to hold a sequence number – which was used as a record key.

Since the "**$**" was not otherwise used in B5000 Algol, Waychoff and Creech decided to recognize cards beginning with that character as a signal to the compiler. Initially, this was used to control whether the compiler should read the entire program from cards or to merge cards by sequence number with a base source on tape.

As the usefulness of this capability became recognized, additional "**$**" options were implemented to control the compilation listing, void (delete) sequences of cards coming from tape, create a new, updated source tape from the merge of an older tape and its correction cards, resequence the source records, etc.

Percent-Sign Comments

- Standard Algol comments
  - `COMMENT BLAH, BLAH, BLAH ;`
- B5000/5500 Algol
  - Used a Stream Procedure for scanning source
  - Needed an efficient way to detect end-of-card
  - Compiler overwrote column 73 of card image with "**%**"
  - Called the "stoplight" character
  - When "**%**" detected, compiler advanced to next card
- Didn't take long to figure out a free "**%**" anywhere on a card would stop the scan
- Effectively made rest of the card comments

2019 MCP 4059 39

The Algol-60 Revised Report provided only two ways for a programmer to insert comments into the source code of a program, (a) by preceding the comment with the symbol **COMMENT** and terminating it with a semicolon, and (b) by placing the comment after an **END**, terminated by the next **END**, **ELSE**, **UNTIL**, or semicolon. The first was fine for long, multi-line comments, but assembly language programmers were used to being able to write short comments off the the right side of assembly instructions. The second could be used only after an **END**.

In the implementation of the B5000/5500 Algol compiler, card boundaries were not recognized as a delimiter. An identifier or reserved word could start at the end of one line and continue in column one of the next line. The parsing of tokens was handled largely by a Stream Procedure, but unlike the modern **SCAN** construct, having the Stream Procedure stop the scan after some maximum number of characters had been processed was awkward. Therefore, the Stream Procedure needed an efficient way to detect end-of-card.

The solution was to place a delimiter character after the last position of source text on a card (i.e., column 73 once sequence numbers were used) that the Stream Procedure could detect. The character chosen was "**%**" as it wasn't being otherwise used. This was called the "stoplight" character, probably because it stopped the scan, and a percent sign looks a little like a two-lamp traffic light.

It could not have taken that long for someone to realize that if you coded a "**%**" somewhere within the text of a source line (outside of a quoted string, of course), the compiler would assume it had detected the stoplight character at the end of the line and proceed to the beginning of the next line. Anything after that "**%**" would be ignored by the compiler, and therefore was effectively a comment.

So percent-sign comments are not so much a feature of Extended Algol as an accident of its original implementation.

Partial-Word Syntax

- Bit-field manipulation in Algol
  - `X:= A.[30:22];`
  - `Y:= A & B[7:8] & C[39:7:8];`
- Burroughs 220 had a similar feature for digits
  - Some instructions could operate on part of a word
    - Designated as the "sL" field (start-Length)
    - Start with digit "s" and use "L" digits *to the left*
  - Digit numbering: `± 1234 56 7890`
  - Example: `STA WD,63`

A Register: `+7631450822`

sL = 63

Memory at address WD: `+1719825634`

2019 MCP 4059 40

Perhaps second only to the **DEFINE** declaration, the utility of manipulating partial words (bits and strings of contiguous bits within 48-bit hardware words) in Extended Algol has helped to extend its range well beyond that of a language for specifying numerical algorithms. There are two basic forms:

- *Field isolation*, which extracts a string of bits from a word value and returns the bits right-justified over zeroes. For example, **A.[30:22]**, where **A** is any expression, takes the word value of the expression and extracts a field of 22 bits starting with bit #30 and extending towards the low-order end of the word.
- Field concatenation, which inserts a string of bits from one word value into another. For example, **A & B[7:8] & C[39:7:8]**, where **A**, **B**, and **C** are any expressions, takes the word value of **B** and inserts its low-order eight bits into the low-order eight bits of the word value of **A**. Then it takes the low order eight bits of the word value of **C** and inserts that into the first result starting at bit #39. The overall result is the original value of **A** with the specified fields of **B** and **C** overwriting the specified fields of **A** and leaving the rest of the bits in **A** unaffected.

Bit numbers in these examples range from 47 at the high-order end of a word to 0 at the low-order end, as is the case with the B6500 and later systems. Bit numbering in B5000/5500 Extended Algol was the reverse of this.

These operations are implemented in the hardware and are extremely efficient. They replace the shift-and-mask operations found on other systems, and are used extensively for packing and unpacking word-oriented data structures (i.e., arrays), which traditionally is the only type of data structure Algol has supported.

It turns out that the Burroughs 220 had a similar, but more limited, feature that would manipulate fields of decimal digits within its words. A digit field was designated by a two-digit code termed "sL" (start-Length), so that "63" designated a three-digit field as shown on the slide. Several instructions supported manipulation of partial words, such as STA (Store A Register). In the example shown on the slide, this form of store would place the digits in the 63 field of the A register into the same field of the operand word, leaving all other digits in the memory word unaffected.

The feature was used extensively by the BALGOL compiler to tightly pack tables in order to maximize the size of program the compiler could handle. The B5000/5500 designers no doubt recognized the utility of this 220 feature and carried it forward in a more general form to the new architecture.

Array Row I/O

- B5000 Algol/MCP did not have array-row I/O
  - FORTRAN-like formatted I/O with lists
  - RELEASE statement for files
    - Buffer-level I/O (somewhat like modern Direct I/O)
    - Buffer only accessible as a Stream Procedure parameter
    - No blocking/unblocking support
    - Inefficient, a pain to use
- B5500 Disk File MCP introduced new I/O scheme
  - Read into and write from Algol array rows:

    ```
    READ (F, 30, A[*]) [EOF];
    ```
  - Supports blocking and unblocking of records
  - Supports intelligent buffer handling (e.g., read-ahead)
  - Significantly more efficient

2019 MCP 4059 41

The next anecdote concerns array-row I/O. For the majority of Extended Algol programmers, this is perhaps the most common way they move data into and out of a program.

It turns out that this is not an original feature. It did not exist in the B5000 Algol compiler and MCP. Prior to the B5500, one way you could do I/O was with **`FORMAT`** and **`LIST`** declarations as just discussed.

The second way you could do I/O was with **`RELEASE`** statements. This was somewhat like array-row I/O, but it was buffer-level I/O, similar to modern Direct I/O. What's worse, the buffers were accessible only by Stream Procedures. Even worse than that, as with Direct I/O, there was no support for blocking and unblocking logical records within the buffers. You had to do that yourself, probably in a Stream Procedure. Thus, these statements were inefficient and somewhat difficult to use.

The B5500 Disk File MCP and Algol compiler introduced a new scheme, which we know today as array-row I/O. With this scheme, a **`READ`** or **`WRITE`** statement specifies a file identifier, an arithmetic expression specifying the number of units to transfer between the file's buffer and a one-dimensional array row within the program, and a reference to that array row. Any blocking or unblocking is handled automatically by the MCP. This approach also allowed the MCP to do more intelligent buffer handling, e.g., read-aheads when doing sequential I/O. It was significantly more efficient than formatted I/O and lots easier, safer, and more convenient than **`RELEASE`** statements.

On the B5500, all Algol I/O was done in units of words, and the transfer between buffer and array row had to start at the first word of the array row. Starting with the B6500, this was generalized to allow character-oriented I/O and for the program also to use an indexed array reference or pointer expression as the transfer starting point.

Input-Output List Declarations

- Algol formatted I/O has a LIST declaration
  - `LIST L1 (A, B, C+2, FOR I:=1 STEP 1 UNTIL N DO [ M[0,I], M[I,0], M[I,I] ]); WRITE (LINE, FMT, L1);`
  - Used with formatted READ & WRITE statements
  - On B5000, literal list could not be in the READ/WRITE
- Another carry-over from 220 BALGOL
  - INPUT and OUTPUT declarations
  - Created a co-routine called by the I/O formatters
  - `OUTPUT L1 (A, B, C+2, FOR I=(1,1,N) ( M(0,I), M(I,0), M(I,I) ))$ WRITE ($$ L1, FMT)$`

2019 MCP 4059 42

I have included this final Extended Algol feature as a curiosity, as it is something I have never had occasion to use, and I have always wondered why it existed.

That feature is the **LIST** declaration. It is used with Algol formatted I/O, which is somewhat similar to FORTRAN formatted I/O. In formatted I/O, a **READ** or **WRITE** statement specifies a file identifier (or in FORTRAN, a unit number), the identifier for a formatting string declared elsewhere in the program, and a list of variables and expressions into which the data is to be read or out of which the data is to be written. Thus, in FORTRAN you might write:

```
900 FORMAT (F6.3, 2X, E13.6, 2X, F6.0)
    WRITE 6, 900, A, B, C
```

where "**6**" is a unit number and "**900**" is the label of the format string. In Extended Algol, you would equivalently write:

```
FILE LINE (KIND=PRINTER, MAXRECSIZE=80, FRAMESIZE=8);
FORMAT F1 (F6.3, X2, E13.6, X2, F6.0);
WRITE (LINE, F1, A, B, C);
```

The "**A,B,C**" in both **WRITE** statements is the list of variables that supply the data to be written. Another way to do this in Extended Algol is to move the list to a separate declaration and reference that declaration in the **WRITE** statement, thus:

```
LIST L1 (A, B, C);
…
WRITE (LINE, F1, L1);
```

But why would you bother to do this? It turns out in B5000 Extended Algol, *you had to*. I/O lists could not be written in-line within the I/O statement. With the release of the B5500 and Disk File MCP in 1965, the compiler supported I/O lists written in-line.

This is another feature carried over from 220 BALGOL to the B5000. In BALGOL, I/O lists were specified in **INPUT** and **OUTPUT** declarations and then referenced along with a format identifier in **READ** and **WRITE** statements. Extended Algol **LIST** declarations have simply been carried forward into the modern language all the way from the B5000, but the only place I have ever seen them used is in some Algol programs from the early 1960s that are in the CUBE library, which was recovered in 2018.

References

- ElectroData and 205 History – Tom Sawyer
  - http://www.tjsawyer.com/B205home.php
- B5000 Oral History Transcript (1985) – CBI
  - http://hdl.handle.net/11299/107105
- Stories of the B5000… – Richard Waychoff
  - http://www.ianjoyner.name/Files/Waychoff.pdf
  - https://archive.computerhistory.org/resources/access/text/2016/06/102724640-05-01-acc.pdf [scanned original]
- Algol Source Code and Emulators
  - http://www.phkimpel.us/ElectroData-205/
  - http://www.phkimpel.us/Burroughs-220/
  - http://www.phkimpel.us/B5500/
- This presentation
  - http://www.digm.com/UNITE/2019/

2019 MCP 4059 43

For anyone who is interested in the very early history of programming languages, Donald Knuth prepared a lecture several years ago titled "A Dozen Precursors of FORTRAN." A video of his presentation can be downloaded at https://www.computerhistory.org/collections/catalog/102622137.

END

The Origins of

Burroughs Extended Algol