The Software of the KDF9

by **Bill Findlay**

1: KDF9

1.1: Background and overview

This paper is one of a trilogy on the English Electric (EE) KDF9 computer. It offers, in some detail, a description of its software. An overview of the KDF9, with a basic account of both the hardware and the software, can be found in [FindlayEE]. I present a fuller account of the KDF9 hardware in a second companion paper [FindlayHW]. Sections 1.2 and 1.3 of the present work tell only enough about the hardware to provide an independent basis for understanding the software. The nearest we have to a KDF9 reference manual is [ICL69].

Chapter 2 considers the programming languages that were implemented on the KDF9.

Chapter 3 describes the multiprogramming operating system known as the Time Sharing Director (TSD).

Chapter 4 leans on that to describe Eldon 2, a multiaccess system based on an enhanced TSD.

Chapter 5 describes DEMOCRAT, a pioneering system that may be unfamiliar even to seasoned KDF9 users.

Chapter 6 gives an account of EGDON, the job-shop FORTRAN system, and COTAN, its multiaccess adjunct.

Finally, the References and Bibliography list my primary sources. Much greater depth and detail than can be given here is available in the works cited, many of which have recently become accessible online for the first time. Personal communications are indicated thus: [*pc*] and not further specified.

Like its companion papers, the present work is largely written in the historic present, as though KDF9 still existed.

1.2: Primer on the hardware

Some facts and figures about the hardware give a sense of the resources available to software in the early 1960s on a modest computer, with a core store of 32K 48-bit words and a store cycle time of 6μs.

A KDF9 comprises three primary and many secondary control units, each being microprogrammed by its read-only memory and capable of running in parallel with the others, subject to appropriate interlocks. The primary control units are **Arithmetic Control** (AC), **Main Control** (MC), and **I/O Control** (IOC).

AC executes ALU-only instructions. AC operands are taken from, and results returned to, the **nesting store** (or **NEST**), which is an expression-evaluation stack with maximum depth 19. If an instruction attempts to pop an operand from an empty NEST, or push an operand onto a full NEST, a Nest Over/Underflow Violation (NOUV) interrupt is caused and Director takes the appropriate action. A maximum of 16 NEST cells are available to a problem program; NOUV is caused *after* pushing a 17th value, and *after* popping an empty NEST. Keeping track of NEST depth is one of a KDF9 assembly language programmer's main chores, and NOUV interrupts are among the most common results of programming error.

It is common for parameters of a subroutine to be pushed onto the NEST, especially when the routine implements an arithmetic function. In a modular program with deeply nested subroutines, it can be difficult to ensure that NOUV is always avoided. Routines can save (on entry) and restore (before exit), all or part of the NEST contents; they are helped in this by conditional jumps that test whether the NEST is empty.

MC takes most of the responsibility for instruction sequencing, including instruction fetching, jumps, subroutines, and interrupts; for mediating access to the KDF9's various stores; and for address generation and indexing. The **main** (**core**) **store** has up to 32768 48-bit words (192K bytes). The **Q Store** is a bank of 16 indexing registers, each of 48 bits and consisting of three 16 bit fields: the **C**-, **I**- and **M**-parts. The C part is used for loop counts and for device numbers in I/O orders; the I part holds either the start address for an I/O transfer, or an increment used to update the M part; the M part is used as the end address for a transfer, or for address modification (indexing). Operand addresses for data fetches and stores are generated in either of two ways: by an instruction containing a 15-bit constant and a single Q store reference, Q*q*; or by an instruction containing two Q store references, Q*k* and Q*q*. The effective address in the first case is the sum of the constant and M*q*; in the second case it is the sum of M*k* and M*q*. There is very flexible provision for auto-incrementing M*q*, using I*q*. The address part of a jump instruction consists of a 13-bit word address and the 3-bit number of an 8-bit **syllable** within that word. Consequently, the instructions of a program are confined to its first 8192 words. The **Subroutine Jump Nesting Store** (**SJNS**) is a stack of 16-bit return addresses, with maximum depth 17. Its management poses similar problems to those of the NEST.

A hardware **timesharing** option replicates the NEST, SJNS and Q Store for each multiprogramming priority level, so that four register sets are provided, avoiding any need to save and restore 48 registers when context-switching. Multiprogramming is not supported unless this option is fitted, but most KDF9s have it. To secure a program's storage it uses virtual addresses starting at 0, offset when running by the contents of the Base Address (BA) register, and checked not to exceed the contents of the Number of Locations (NOL) register, these values having being set for the program by Director. This means that Director can move programs around to consolidate free space when programs terminate, and that no absolute address fixups are needed at load time. Both freedoms provide significant efficiency advantages over the later System/360 design.

IOC supervises up to 16 **buffers**, DMA channels that are capable of simultaneous transfers. Each has its own independent control unit, which is microprogrammed to support its connected device. A single-bit **test register** can be set in response to instructions querying a buffer's status. The operator's console includes a Flexowriter on buffer 0. A peculiarity of the online Flexowriter is that, when a semicolon is typed, the write operation changes itself on the fly to a read operation; subsequent typed input is transferred to the originally designated output area, in character positions following the semicolon. This can be used to program an atomic, prompt-and-response, form of interaction. Moreover, output is typed in red ink, input in black, although the mode-changing semicolon is itself in black, not red.

Slow I/O devices include paper tape punches (110ch/s), paper tape readers (1000ch/s), card readers (10 card/s), card punches (5 card/s) and line printers (15 line/s). The Flexowriter and the paper tape devices use the same code, which employs 'Case Shift' and 'Case Normal' characters to expand the character set. The fast devices are magnetic tape decks (40Kch/s), fixed-disc (not *disk*) drives (24MB, 85Kch/s), and drum stores (320KB, 500Kch/s!).

Problem programs can directly drive I/O devices on buffers allocated to them by Director. Any attempt to access an unallocated buffer causes an invalid operation (Lock-In Violation, LIV) interrupt, and termination of the program. This feature of KDF9 means that programs are inherently device-dependent: they must contain logic specific to the type of device they use. This is less of a disadvantage than it might seem, because card reader/punches, paper tape reader/punches, and printers, all use slightly different characters sets; so programs have to be device-aware for that reason, if no other.

I/O devices employ 6-bit characters, up to 8 of which can be held in one word; however there are no facilities to address packed characters in main store. Characters are packed into words beginning at the most significant 6 bits. Conveniently, in all

variants of the KDF9 character code, 6 zero bits represent a blank space character; and 6 one bits represent a **filler** character, which legible output devices (such as the line printer and Flexowriter) completely suppress.

When an I/O operation starts, the area of core store it designates is **locked out** until the transfer terminates. Any attempt to access this area causes a Lock-Out Violation (LOV) interrupt, and blocks the program's execution until the transfer is done. This means that transfers are atomic as far as user programs are concerned: if they attempt to examine the result in core, they always find that the transfer is over. However, there are operations (i) to see whether a device is busy (BUSYQ*q*), and (ii) to interrupt the program if so (INTQ*q*). If the addressed device is busy, INTQ*q* makes IOC note the identity of the blocked multiprogramming level, for future reference when any of its transfers finish. When a transfer on behalf of the highest priority *blocked* program level terminates, IOC detects this and requests an interrupt, prompting Director to reschedule. A terminating transfer for a *running* program does not cause an unnecessary interrupt.

A KDF9 instruction consists of one, two or three **syllables** of 8 bits, the first two bits of each instruction giving its type. Each instruction word is therefore capable of holding between two and six instructions, dependent on their lengths. One-syllable instructions do not contain an operand address or other parameter, and operate only on the NEST in 'Reverse Polish' style. Two-syllable operations include all operations that require one or more Q store numbers—'indirect' memory fetches and stores, operations on the contents of Q stores, shift orders, I/O orders, and others such as the J*r*C*p*NZS **short-loop jump** instruction. Three-syllable jumps contain a 16-bit instruction address. Directly addressed fetch and store orders contain a 15-bit word address, and the SET instruction contains a 16-bit constant.

1.2: Interrupts and privileged state

KDF9 interrupts may be either: **voluntary** (e.g. OUT), **inadvertent** (e.g. LIV, NOUV), or **housekeeping** (clock, monitor typewriter, and other I/O interrupts). The Reason For Interrupt Register (RFIR) records the currently requested interrupts. It is inspected from time to time by Main Control, depending on the instruction being executed. When an interrupt is accepted, control branches to word 0 of Director, leaving a return address in the top cell of the SJNS. On entry to Director, the BA is set to 0 and further interrupt requests are postponed, although they continue to be recorded in RFIR for Director's future attention. Exceptions are that the RESET (serious failure) interrupt is always allowed; and that the NOUV and LOV interrupts are completely suppressed.

Since NOUV is suppressed in Director state, all 19 NEST cells—16 in the core stack, plus three flip-flop registers, N1, N2 and N3—can be used. Director does so to great advantage in its 'short path' interrupt handler. This is simple enough to be able to work entirely in the N registers, and need not save the interrupted program's whole NEST to make room for itself. It is sufficient to save the contents of the N registers in the core stack and this happens at zero cost, as a side effect of Director's own calculations. This means that interrupt response is rather efficient, the path length between leaving and re-entering a program being about 60 instructions. The 'long path' through Director, where more complex work is done, must explicitly save more of the interrupted program's registers; but that happens relatively infrequently.

The implementation of the SJNS is similar to that of the NEST. A flip-flop register analogous to the NEST's N registers, constitutes the 17th cell of the SJNS. It ensures that interrupts can be taken when the SJNS is full, and it similarly promotes the efficiency of the Director's short path.

Instructions to manipulate the special registers are available only in Director state. They are:

- EXITD: allow interrupts and jump to the address in the top cell of the SJNS (dispatch user program)
- CLOQ*q*: clear any lockouts in the area specified by I*q* .. M*q*
- 'SLOQ*q*': actually called PMHQ*q*, set lockouts in the area specified by I*q* .. M*q*
- CTQ*q*: terminate device activity and clear any lockouts for the transfer specified by Q*q*
- =K0: **if** N1 ≠ 0 **then** switch buzzer on **else** switch buzzer off
- =K1: set relocation registers BA and NOL
- =K2: set device permissions register, CPDAR (1 bit per buffer)
- =K3: switch to a new Q store/NEST/SJNS set
- K4: get the CLOCK/RFIR (Reason For Interrupt Register)
- K5: get the PHUs (Program Hold-Up registers, recording the reasons for programs being blocked)
- K7: get the current register set number and NEST depths, as represented for the =K3 instruction

2: Programming languages

KDF9 is well supplied with the programming languages of its era. Usercode, the EE assembly language, has variants called UCA3 (adapted to card input for EGDON) and KAL4 (for Eldon 2). KAL4 provides the facility for symbolic data names that is sadly lacking in EE's own Usercode. As for high-level languages, these focus on 'scientific' computing. Algol 60, Babel, FORTRAN, IMP, K Autocode, and KDF9 (Atlas) Autocode are all available; COBOL is a notable absentee, it was promised by EE but never materialised.

2.1: Usercode

The KDF9 assembly language, called **Usercode**, is very unusual in having a distributed syntax that embeds parameters within the Usercode order. For example, the J10C2NZ instruction means '**J**ump to label **10** if the **C**-part of Q store **2** is **N**ot **Z**ero'. It is possible that this format was suggested by the order code, which distributes the opcode and address bits around the machine instruction in an equally unconventional manner (presumably for ease of decoding).

Usercode instructions are labelled by integers. An asterisk preceding a label forces the following, labelled, instruction to start at syllable 0 of a fresh word, any unused syllables of the preceding word being padded with DUMMY (no-op) instructions. This is necessary for the label operand of a J*r*C*q*NZS instruction, which does not contain a label address, but instead jumps to syllable 0 of the word before the word containing the J*r*C*q*NZS instruction itself.

Data locations in EE Usercode have rigidly stereotyped identifiers. Variables declared within routines have names of the form V*m*. They may be given initial values, and can be accessed globally if qualified by the name of their containing routine, thus: V1P2. Global variables have names of the form W*m*, Y*m*, YA*m*, …, YZ*m*. Absolute virtual address *m* is written E*m* and, numbered backwards, as Z*m*, so that E0 and Z0 respectively denote the lowest and the highest addresses in the program.

For indexing the identifier is followed by 'M' and the Q store number, thus: YA7M2. Usercode instructions such as V9; YA7M2; and so on, represent data fetches that push values onto the NEST, whereas =V9; =YA7M2; and so on, represent data

stores that pop values from the NEST. Appending the letter 'Q', thus: YA7M2Q; =YA7M2Q; specifies autoincrementing of Q*q*. The 'Q' suffix causes the Q*q* register to be updated, *after* the effective address is determined, by adding the contents of I*q* to M*q* and decrementing the contents of C*q*. This allows stepping through variable number of locations, at addresses given by an initial address and a variable stride. There are conditional jump instructions that test whether the C-part of a Q store is (non-)zero, providing for very efficient counting loops.

The 2-syllable fetch and store instructions take the forms M*k*M*q* and =M*k*M*q*, the effective address being the sum of M*k* and M*q*. Flags suffixed to an instruction optionally specify autoincrementing, halfword addressing, and 'next word' addressing. The 'H' suffix on a 2-syllable fetch or store order causes the operand accessed to be a halfword. In this case the content of M*k* is taken as a base word address, and the content of M*q* is taken as a halfword offset, odd-numbered halfwords being those in the less significant half of the addressed word. The 'N' suffix causes the accessed word to be at an address 1 greater than usual (i.e., the **n**ext word). This allows efficient processing of arrays of pairs of elements stored sequentially in adjacent words—such as the constituent words of a double-precision number—since Q*q* needs to be updated only once for every word pair, using an increment part set to 2. All combinations of Q, H, and N, in that order, are permitted in a 2-syllable fetch or store order.

It can happen that the order of operands in the NEST, while convenient for some purposes, is inconvenient for others. To reorganize the NEST, we have the 1-syllable instructions:

- REV: a, b, ... → b, a, ...
- DUP: a, ... → a, a, ...
- ERASE: a, ... → …
- CAB: a, b, c, ... → c, a, b, ...
- PERM: a, b, c, ... → b, c, a, ...
- REVD: a, b, c, d, ... → c, d, a, b, ...
- DUPD: a, b, ... → a, b, a, b, ...

The 1-syllable ZERO and the 3-syllable SET orders allow for the sourcing of small constants.

EXAMPLE 2.1—COMPUTE THE FLOATING-POINT VALUES $(-b \pm \sqrt{(b^2 - 4ac)}) / 2a$ IN USERCODE
The values ***a***, ***b*** and ***c*** are stored in the variables YA0, YB0, and YC0. Subroutine P40 calculates √(N1). Two frequent tactics for dealing with a common subexpression are exemplified: holding it in a Q store, or buried in the NEST.

`YA0; DUP; +F; =Q1;`	NEST *empty, and* Q1 *contains* $2a$
`YB0; NEGF; DUP; =Q2;`	N1: $-b$, *and* Q2 *contains* $-b$
`DUP; ×F;`	N1: b^2
`YC0; Q1; ×F; DUP; +F; -F;`	N1: $b^2 - 4ac$
`JSP40;`	N1: $\sqrt{\Delta}$, *where* $\Delta = b^2 - 4ac$
`DUP; NEGF;`	N1: $-\sqrt{\Delta}$; N2: $+\sqrt{\Delta}$
`Q2; +F; Q1; ÷F; REV;`	N1: $+\sqrt{\Delta}$; N2: $(-b - \sqrt{\Delta}) / 2a$
`Q2; +F; Q1; ÷F;`	N1: $(-b + \sqrt{\Delta}) / 2a$; N2: $(-b - \sqrt{\Delta}) / 2a$

■

The J*r* instruction is an unconditional jump to the instruction labelled *r*. The instructions: J*r*=Z, J*r*≠Z, J*r*>Z, J*r*≥Z, J*r*<Z, J*r*≤Z, *etc*, test the sign of the top cell of the NEST; to compare the top two cells of the NEST we have: J*r*= and J*r*≠. All of these pop N1 whether they jump or not; J*r*= and J*r*≠ do not pop N2, being the only dyadic operations with this behaviour. To test whether C*q* is (non-) zero we have: J*r*C*q*Z, J*r*C*q*NZ, and J*r*C*q*NZS. The J*r*V and J*r*NV instructions are conditional on the overflow register being (un-)set, while J*r*TR and J*r*NTR are conditional on the test register being (un-)set; these instructions clear the designated register whether they jump or not.

J*r*C*q*NZS is known as the **short loop jump**. It jumps, if C*q* is nonzero, to syllable 0 of the instruction word that precedes the word containing the J*r*C*q*NZS instruction; and has the further effect of inhibiting instruction fetch cycles. Thus the loop executes entirely from the instruction word buffers, with no overhead for instruction fetches. Important algorithms, such as scalar product and polynomial evaluation, fit comfortably into the 12 available syllables.

EXAMPLE 2.2—COMPUTE THE SCALAR PRODUCT $\sum x_i y_i$ IN USERCODE
On entry to the loop Q1 = 10/1/0; i.e. C1 holds 10, I1 holds 1 and M1 holds 0. The vector ***x*** is in the variables YX1…YX10, ***y*** is YY1…YY10; and the NEST contains a pair of zeros. The ×+F instruction is a 'multiply and accumulate' operation: it forms the double-precision product N1×N2 and adds that to the double-precision sum in N3 and N4.

```
*1;   YX1M1;
      YY1M1Q;
      ×+F;
   J1C1NZS;
```

■

Routine names are limited to the form P*p*; a label *r* defined within P*p* may be referenced from outside as R*r*P*p*.

The JS*r* instruction branches unconditionally to the label *r*, which may be a routine, or within a routine, and pushes **its own** address onto the SJNS as a return link. Apart from JS*r*, the main instructions that use the SJNS are:

- EXIT *a*: complementary to JS*r*—pops the link from the SJNS, adds the constant *a* (which represents 3*a* syllables), and branches to the resultant address
- LINK: pops a link from the SJNS and pushes it onto the NEST
- =LINK: pops a link or index value from the NEST and pushes it onto the SJNS

Normal return from a subroutine is effected by EXIT 1; if there is an abnormal return path, it is taken by EXIT 1 and EXIT 2 is the normal return. Switches are programmed by putting the index into the SJNS, by means of the =LINK instruction, then EXIT AR*r* jumps to the selected halfword in the jump table starting at label *r*.

Usercode has device-specific mnemonics for many hardware I/O instructions. For example, the output instruction POAQ*q* can also be written as MWQ*q* when the intended device is a magnetic tape drive, and as TWQ*q* when it is the console typewriter. The use of a specific mnemonic has no real significance and can be misleading, as the identity of a device, and therefore its type, need not be determined until run time.

EXAMPLE 2.3—COMPUTE ACKERMANN'S FUNCTION, A(M, N) IN USERCODE
Here is a (very small) complete program in Usercode. Without the comments it would hard to know what it does. Perhaps it is not terribly clear even with the comments!

```
V6; W0; YS20000;
RESTART; J999; J999;
PROGRAM;                    (main program);
   V1 = B1212121212121212;  (radix 10s for FRB);
   V2 = B2020202020202020;  (high bits for decimal digits);
   V3 = B0741062107230637;  ("A[3,"  in Flexowriter code);
   V4 = B0726062200250007;  ("6] = " in Flexowriter code);
   V5 = B7777777777777777;
      ZERO; NOT; =M1;       (Q1 := 0/0/-1);
      SETAYS0; =M2; I2=2;   (Q2 := 0/2/AYS0: M2 is the stack pointer);
      SET 3; =RC7;          (Q7 := 3/1/0: C7 = m);
      SET 6; =RC8;          (Q8 := 6/1/0: C8 = n);
   JSP1;                    (call Ackermann function);
      V1; REV; FRB;         (convert result to base 10);
      V2; OR;               (convert decimal digits to characters);
      SHL+30; =V5;          (eliminate leading zeros);
      SETAV5; =RM9;
      SETAV3; =I9;
      POAQ9;                (write result to Flexowriter);
999;  ZERO; OUT;            (terminate run);

P1V0;                 (To compute A[m, n]);
   J1C7NZ;            (to 1 if m not equal to 0);
      I8; =+C8;       (n := n + 1);
      C8;             (result to NEST);
   EXIT 1;            (return);

   *1;
   J2C8NZ;            (to 2 if n not equal to 0);
      I8; =C8;        (n := 1);
      DC7;            (m := m - 1);
   JP1;               (tail recursion for A[m-1, 1]);

   *2;
      LINK; =M0M2;    (push return address);
      C7; =M0M2QN;    (push m);
      DC8;            (n := n - 1);
   JSP1;              (full recursion for A[m, n-1]);
      =C8;            (n := A[m, n-1]);
      M1M2; =C7;      (m := top of stack);
      DC7;            (m := m - 1);
      M-I2;           (pop stack);
      M0M2; =LINK;    (return address := top of stack);
   JP1;               (tail recursion for A[m-1, A[m, n-1]]);
FINISH;
```

■

2.2: KAL4 AND OTHER ASSEMBLY LANGUAGES

The Paper Tape Generator [EEL65a] is a somewhat superior version of Usercode, with operands having programmer-chosen identifiers. Strangely, EE did not countenance its use for any purpose other than writing Directors and other system software, in-house. Early Directors were written in 'Neal code', a primitive ancestor of Usercode with much terser instructions and even less comprehensibility. Its only saving grace was some facility for symbolic identifiers. See [EEL64a], Section A1002, which contains the Neal code listing of a Director.

UCA3 is the Usercode variant provided by the EGDON operating system. It differs from conventional Usercode in its adaptations to the limited EGDON character set, and in the addition of features to exploit EGDON's relocatable object module format. So, in UCA3, instructions are separated by commas, not semicolons and each UCA3 module starts with cards that match symbolic names with internal and external routine entry points.

KAL4 [Thomason70] is a thorough rethinking of Usercode. Routines, labels, and variables have identifiers instead of serial numbers. A program may be composed of multiple routines, with nested scopes delimited by `begin` and `end` for locally declared names. KAL4 provides a convenient object code for the Eldon 2 FORTRAN and BASIC compilers. The KAL4 assembler is written in KAL4.

2.3: ALGOL 60

KDF9 is heavily invested in Algol 60 [Naur60], the international algorithmic language that is the common ancestor of most of today's most widely used programming languages. In fact, KDF9 is supplied with two Algol 60 compilers that run under the Time Sharing Director: the Kidsgrove (KAlgol) and Whetstone (WAlgol) systems, which are named after the EE installations where they were written. The EGDON system also eventually gained a third Algol 60 compiler,

The Whetstone and Kidsgrove compilers accept a common variant of the Algol 60 standard, defined to remove ambiguities and other defects of the language described in the *Revised Report*. The authors of KDF9 Algol are quick to acknowledge the debt they owe to the work of Dijkstra and co-workers on the pioneering Algol compiler for the Xl Computer at the Mathematical Centre, Amsterdam.

EXAMPLE 2.4—COMPUTE ACKERMANN'S FUNCTION, A(M, N) IN KAL4

```
begin
PROG=0:J.START;               / initial jump to start of program
PROG=4:J.RESTART;J.RESTART;   / restart for operator intervention
PROG=8:
RADIX10  = ≠1212121212121212; / radix 10s for FRB
DECIBITS = ≠2020202020202020; / high bits for decimal digits
MESSAGE1 = ≠0741062107230637; / "A[3,"  in Flexowriter code
MESSAGE2 = ≠0726062200250007; / "6] = " in Flexowriter code
MESSAGE3 = ≠7777777777777777; / the answer goes here
START:
      ZERO; NOT; SM1;         / Q1 := 0/0/-1
      SET.STACK; SM2; ITWOQ2; / Q2 := 0/2/STACK: M2 is the stack pointer
      SET 3; SRC7;            / Q7 := 3/1/0: C7 = m
      SET 6; SRC8;            / Q8 := 6/1/0: C8 = n
   JS.ACKER;                  / call Ackermann function
      F.RADIX10; REV; FRB;    / convert result to base 10
      F.DECIBITS; OR;         / convert decimal digits to characters
      SHL+30; S.MESSAGE3;     / eliminate leading zeros
      SET.MESSAGE3; SRM9;
      SET.MESSAGE1; SI9;      / Q9 := 0/MESSAGE1/MESSAGE3
      POAQ9;                  / write result to Flexowriter
RESTART:
      ZERO; OUT;              / terminate run

ACKER: /To compute A[m, n]
   JCNZ7.mNE0;                / to nNE0 if m not equal to 0
      I8; =+C8;               / n := n + 1
      C8;                     / result to NEST
   EXIT.1;                    / return

mNE0:
   JCNZ8.nNE0                 / to nNE0 if n not equal to 0
      I8; SC8;                / n := 1
      DC7;                    / m := m - 1
   J.ACKER;                   / tail recursion for A[m-1, 1]

nNE0:
      FLINK; SMM2/0;          / push return address
      C7; SMMQN2/0;           / push m
      DC8;                    / n := n - 1
   JS.ACKER;                  / full recursion for A[m, n-1]
      SC8;                    / n := A[m, n-1]
      FMM2/1; SC7;            / m := top of stack
      DC7;                    / m := m - 1
      M-I2;                   / pop stack
      FMM2/0; SLINK;          / return address := top of stack
   J.ACKER;                   / tail recursion for A[m-1, A[m, n-1]]
end
PROG=128:0;
STACK:
PROG=20479:0;
end
```

■

2.3.1: WHETSTONE ALGOL

WAlgol is a 'checkout' compiler aimed at fast compilation and comprehensively monitored execution. It generates byte code for an abstract machine that bears a remarkable resemblance to the machine code of the later, and WAlgol influenced, Burroughs B6500 architecture. Native to the paper tape oriented EE Time Sharing system, WAlgol was adapted to card input by Patterson *et al*. in the guise of In-core Algol [PS69], serving as a batch compiler for large numbers of short student jobs under EGDON. Transplanted to COTAN, it also provides a facility for online compilation and interactive execution. Under Eldon 2 it offers a 'cafeteria' fast turnaround system, again for short student jobs.

WAlgol is so well documented, in a groundbreaking book [RR64], that it engendered Algol 60 implementations on many other computers, including the EE DEUCE, the Ferranti Pegasus [Ryder64], the NPL ACE, the EE KDF6, the Soviet Minsk range, the EE System 4/50 and the IBM System 360/25 [Leigh74], the Phillips PR 8000, and the Indian ECIL TDC-316.

Until recently, the influence of the Whetstone system on the Burroughs large-systems architecture had remained plausible but conjectural. The B5500 had 12-bit instructions and no up-level addressing (so it could not fully implement Algol 60 non-locals and closures). Its successor, the B6500, had variable-length instructions consisting of 1 or more syllables of 8 bits, and used a 'display' for addressing non-locals, just like Whetstone. Bill McKeeman [*pc*] has said that at least some of these similarities are not coincidental. He writes: "I ended up consulting for Burroughs Pasadena on the B6500 design. Ben Dent was struggling with generalising the B5500 stack to handle up-level addressing. So I told him how. During each weekly visit we discussed the problem until he asked 'How do you know all of this?' So I said, it is in a book—ALGOL 60 Implementation [RR64]. He was irritated at not knowing sooner, got the book, and did not need to talk to me about it again. So I think it is fair to say that whatever he needed, he got from the book." Andrew Herbert tells me [*pc*] that Elliott 903 Algol, written by the software house CAP, is also inspired by the WAlgol abstract machine, though it is not a simple re-implementation.

EXAMPLE 2.5—COMPUTE THE SCALAR PRODUCT $\sum x_i y_i$ WITH WALGOL

Here is the scalar product algorithm in Algol, for comparison with the Usercode in Example 2.2. The arithmetic is all in single precision, as Algol 60 does not recognize more than one **real** type. The body of the loop is treated as a procedure because Algol 60 allows multiple loop controls in a single for statement. One could write complex iterations such as:

```
for i := 2, 3, 5, 7, 11 step i÷3 until 31, i+1 while prime(2×i+1) do S
```

This is implemented, in effect, by making *S* into a procedure and calling it several times, with `i` set to 2, 3. … etc.

Bear in mind that Whetstone is a very strictly one-pass affair: once an operand has been dealt with, the compiler forgets about it. So by the time it is processing the body of the loop, it has forgotten that `i` is the controlled variable! All the iteration code—updating `i`, and checking it against the final value—has to be generated on the fly. This leads to a very indirect implementation in which control alternates somewhat obscurely between the loop body and the iterator.

```
real array X, Y [1:10];
integer i; real sum;
for i := 1 step 1 until 10 do
   sum :=  sum + X[i] × Y[i]
```

Here is the Whetstone code for the loop. I have shown identifiers as operands of the abstract machine orders. In data handling orders they were actually (*level*, *offset*) pairs that addressed the location given by DISPLAY[*level*] + *offset*.

```
UJ iterator;                          jump to the iteration control code for i
access:
   TIA i; LINK;                       save the address of i for use by the iterator
iterator:
   CFZ body;                          save the address of the loop body for use by the iterator
   FORS1; TIC1; LINK;                 first iteration: set initial value, call loop body
   FORS2; TIC1; LINK; TIC 10; LINK;   next iteration: add 1 to i, compare with 10, call loop body
   FSE exit;                          end iteration: jump past loop body
body:
   FBE (…), access;                   enter the 'procedure', save the address of the code to access the controlled variable
   TRA sum;                           push sum's address
      TRV sum;                        push sum's value
         TRA X; TIR i; INDR;          index X with i and push the element's value
         TRA Y; TIR i; INDR;          index Y with i and push the element's value
         ×;                           multiply the top two items in the stack
      +;                              add the product to the top item in the stack
   ST;                                store the top item in the stack to the address one level beneath, i.e. sum
   FR;                                return to the iterator, where i, the controlled variable, is updated and checked
exit:
```

■

Whetstone has a justly deserved reputation for fast compilation. The Translator finishes as soon as the last character of the Algol program goes through the 1000ch/s paper tape reader. A 200 line program compiles in about 5 KDF9 CPU seconds, for a rate of 2400 lines per minute; this is very fast for 1964. Short compile times make Whetstone eminently suitable for a typical academic workload, consisting of a large number of students' programs needing quick turnaround during a laboratory session, and it is popular in the KDF9-owning universities.

Execution speed is another matter. Being interpreted, Whetstone object programs do not run quickly. A slowdown by a factor of between 75 and 100 is to be expected, relative to equivalent Usercode. The compensation is that the Controller (as the interpreter is known) checks for errors that would probably go undetected in a native machine code program.

Another advantage of interpretive execution is that it is a simple matter to instrument the Controller to gather statistical information about the runtime characteristics of real Algol programs. Brian Wichmann carried out systematic investigations of this kind [Wichmann70a-b, Wichmann72, Wichmann73a-b, WJ76], that informed the creation of the Whetstone Synthetic Benchmark [CW76]. The latter was used throughout the industry, for many years, as a machine independent estimator of computer performance on 'scientific' workloads.

2.3.2: KIDSGROVE ALGOL

The Kidsgrove Algol compiler [HH63] aims at efficient machine code and includes an optional global optimization phase. It gained an unfortunate reputation for being unreliable and for generating slow object code, but these criticisms must be seen in context. Most compilers in the early 1960s were at least equally guilty, WAlgol being a shining counterexample. The first FORTRAN compiler was only three years old when both Algol 60 and the KDF9 were announced. FORTRAN is a radically simpler language than Algol, and designed with a view to compilation. Indeed it was designed by its compiler writers. Algol, on the other hand, is defined by an international standard that was conceived with little thought for its difficulty of implementation. Nested scopes, complicated looping, dynamic arrays, recursion, and name parameters all combine to make the work of an Algol

compiler onerous. In particular, these features of Algol make machine oriented optimizations hard to achieve. For example, when a procedure *P* takes the value of a name parameter it might invoke an arbitrarily complex calculation, possibly involving assignments, I/O, and more procedure calls, perhaps even of *P* itself. So the programmable machine registers have to be reset to a standard state every time a name parameter is used—not something that tends to efficient code!

In Algol 60 the general case is by far the worst case. Efficient object code is achievable only by exploiting special cases that are both more frequent in practice and more amenable to implementation. This requires a deep analysis of a program. It is just such an analysis that the Kidsgrove compiler pioneered [Huxtable64]. The sophistication of this work is remarkable for its time.

Among the optimizations it attempts are: common subexpression elimination; faster procedure calls for several classes of simple—especially leaf—procedures; array indexing speeded up by strength reduction; good 'for' loop code when the loop parameters are benign; and allocation of stack storage at procedure level.

To give an idea of how difficult some of these optimizations are in Algol 60: common subexpression elimination is not possible if the subexpression contains a name parameter, *unless* all of the corresponding actual parameters involve only simple variables immune to side effects, or array elements whose subscripts are immune. Checking for side effect safety requires examining the transitive closure of all the procedure calls in the program.

Studies of the efficiency of KAlgol object code are facilitated by its being (relatively) legible Usercode, rather than machine code. An investigation [Wichmann71] revealed several opportunities for 'peephole' optimization of the Usercode, and such an optimization pass was added to the compiler [HW71] with some success.

EXAMPLE 2.6—COMPUTE THE SCALAR PRODUCT $\sum x_i y_i$ WITH KALGOL

Here is how a very early version of the Kidsgrove compiler treats the Algol code of Example 2.5.

(a) Unoptimized:

```
   SET1; DUP;
   =Y7M1;=Y10M1;        (i := 1; loop count := 1);
   SET1; =Y11M1;        (set step value);
195; (top of loop);
   Y11M1; Y10M1; SET10; -; ×D; CONT; (test loop count condition);
   ZERO; SIGN; SHA-1; NEG;           (convert condition to standard Boolean);
J198=Z;                              (jump to loop body if false);
J194;                                (jump past end of loop);
198; (loop body);
   Y8M1;                 (push sum into NEST);
   Y7M1; =Y0M2Q;         (stack i);
   Y4M1;                 (push base address of X into NEST);
   M-I2; Y0M2;           (unstack i);
   +; =M3;               (compute the address of X[i] in M3);
   Y0M3;                 (push X[i] into NEST);
   Y7M1; =Y0M2Q;         (stack i);
   Y5M1;                 (push base address of Y into NEST);
   M-I2; Y0M2;           (unstack i);
   +; =M3;               (compute the address of Y[i] in M3);
   Y0M3;                 (push Y[i] into NEST);
   ×F; +F;               (calculate X[i]×Y[i] + sum in NEST);
   =Y8M1;                (pop NEST into sum);
   SET1; =Y11M1;         (reset step value);
   Y7M1; Y11M1; +; DUP;
    =Y7M1; =Y10M1;       (increment i and loop count);
J195;
194; (end of loop);
```

(b) Invoking minimal optimization gets us the following, where the loop control has been much simplified, taking into account that the starting value, increment and final value are all benign positive constants:

```
   SET1;  =Y7M1;       (i := 1);
   SET10; =C15;        (set loop count into C15);
193;
   Y8M1;                (push sum into NEST);
   Y7M1; =Y0M2Q;        (stack i);
   Y4M1;                (push base address of X into NEST);
   M-I2; Y0M2;          (unstack i);
   +; =M3;              (compute the address of X[i] in M3);
   Y0M3;                (push X[i] into NEST);
   Y7M1; =Y0M2Q;        (stack i);
   Y5M1;                (push base address of Y into NEST);
   M-I2; Y0M2;          (unstack i);
   +; =M3;              (compute the address of Y[i] in M3);
   Y0M3;                (push Y[i] into NEST);
   ×F; +F;              (calculate X[i]×Y[i] + sum in NEST);
   =Y8M1;               (pop NEST into sum);
   Y7M1; SET1; +;       (calculate i + 1 in NEST);
   =Y7M1;               (pop NEST to i);
   DC15;                (decrement loop count);
J193C15NZ;  (loop back if count not 0);
```

(c) After applying the peephole pass of Healey and Wichmann [HW71] we get this for the body of the loop:

```
193;
   Y8M1;                  (push sum into NEST);
   Y7M1; Y4M1; +; =M3;    (compute the address of X[i] in M3);
   Y0M3;                  (push X[i] into NEST);
   Y7M1; Y5M1; +; =M3;    (compute the address of Y[i] in M3);
   Y0M3;                  (push Y[i] into NEST);
   ×F; +F;                (calculate X[i]×Y[i] + sum in NEST);
   =Y8M1;                 (pop NEST into sum);
   Y7M1; NOT; NEG;        (calculate i + 1 in NEST);
   =Y7M1;                 (pop NEST to i);
   DC15;                  (decrement loop count);
J193C15NZ;  (loop back if count not 0)
```

(d) With full optimization enabled, and throwing in a [HW71]-style peephole pass as well (which affects only the first two lines), the Kidsgrove compiler produces this from the complete **`for`** statement, making much better use of KDF9's idiosyncratic Q store:

```
   Y4M1; NOT; NEG; =RM14; (put address of X[1] into M14 and +1 into I14);
   Y5M1; NOT; NEG; =RM15; (put address of Y[1] into M15 and +1 into I15);
   SET10; =C15;           (set loop count into C15);
193;
   Y8M1;                  (push sum into NEST);
   Y0M14Q;                (push X[i], add I14 to M14 for X[i+1]);
   Y0M15Q;                (push Y[i], add I15 to M15 for Y[i+1], decrement loop count);
   ×F; +F;                (calculate X[i]×Y[i] + sum);
   =Y8M1;                 (pop NEST into sum);
J193C15NZ;  (loop back if count not 0)
```

This hoists nearly all of the address calculation out of the loop and completely replaces the controlled variable `i` by usages of the fast Q store registers Q14 and Q15. A huge improvement on (a), it begins to compete with the optimal method of Example 2.2.

■

One must admit that KAlgol object programs do not generally rival the efficiency of hand coding, but it would be unreasonable to expect that they should. By comparison with a human programmer, and for reasons already explained, the compiler cannot always make the best use of the KDF9's unique complement of registers. For example, to avoid causing a NOUV interrupt, KAlgol sometimes adopts a safety-first approach [Wichmann73, WJ76]. When building up a series of partial results, it may hold them as items in the core store stack frame, rather than the NEST, entailing the semantically empty to-ing and fro-ing that can be seen in the first few lines of the loop body at (b), above.

It is, however, possible to overdo this criticism. Part of the blame can be attributed to the NEST itself. It has been said that the only sensible multiplicities in computing are zero, one, and infinity. A nesting store with 16 cells falls uncomfortably on that scale. In 1963 optimal register allocation had hardly been formulated as a problem, let alone solved. The IBM machines that hosted the first FORTRANs had a single 'accumulator', so the topic did not become a subject of active research interest until the advent of System/360 with its 16 'general' registers.

And, despite all this, the Kidsgrove compiler *is* capable of doing expression evaluation very well indeed.

EXAMPLE 2.7—NEST MANIPULATION BY KALGOL

This, admittedly artificial, example shows KAlgol doing an excellent job with the NEST. For the expression, with **`real`** `a, b`:

```
(a+b) × ((a+b) × ((a+b) × (a+b)))
```

KAlgol generates:

`Y4M1;`	N1: ***a***
`Y5M1;`	N1: ***b***, N2: ***a***
`+F;`	N1: (***a***+***b***)
`DUP;`	N1: **(*a*+*b*)**, N2: (***a***+***b***)
`DUP;`	N1: **(*a*+*b*)**, N2: (***a***+***b***), N3: (***a***+***b***)
`×F;`	N1: **(*a*+*b*)** × (***a***+***b***), N2: (***a***+***b***)
`REV;`	N1: **(*a*+*b*)**, N2: (***a***+***b***) × (***a***+***b***)
`DUP;`	N1: **(*a*+*b*)**, N2: (***a***+***b***), N3: (***a***+***b***) × (***a***+***b***)
`CAB;`	N1: **(*a*+*b*) × (*a*+*b*)**, N2: (***a***+***b***), N2: (***a***+***b***)
`×F;`	N1: **(*a*+*b*)** × ((***a***+***b***) × (***a***+***b***)), N2: (***a***+***b***)
`×F;`	N1: **(*a*+*b*)** × ((***a***+***b***) × ((***a***+***b***) × (***a***+***b***)))

this saves five core store cycles, 25 clock cycles and 7 syllables of machine code over saving and repeatedly fetching `(a+b)` in a core store temporary, as an accumulator machine would be forced to do. Bear in mind that the arithmetic operators must be executed as written in the source code expression, for compliance with the semantics of Algol 60 and numerical compatibility with the Whetstone system. The minimal form:

```
Y4M1; Y5M1; +F; DUP; ×F; DUP; ×F;
```

is *not* allowed, even though it is both more efficient and less subject to rounding error.

■

EXAMPLE 2.8—EXPRESSION EVALUATION BY KALGOL

This is a more realistic example, taken from a program that uses Zeller's Congruence to calculate the date of Easter Sunday. For the assignment, with **integer** a, c, h:

```
h := h + c - 7 × ((a + 11×h + 19×c) ÷ 433)
```

KAlgol generates:

Y12M1;	N1: ***h***
Y11M1;	N1: ***h***, N2: ***c***
+;	N1: ***h***+***c***
SET7;	N1: ***h***+***c***, N2: 7
Y9M1;	N1: ***h***+***c***, N2: 7, N3: ***a***
SET11;	N1: ***h***+***c***, N2: 7, N3: ***a***, N4: 11
Y12M1;	N1: ***h***+***c***, N2: 7, N3: ***a***, N4: 11, N5: ***h***
×D; CONT;	N1: ***h***+***c***, N2: 7, N3: ***a***, N4: 11***h***
+;	N1: ***h***+***c***, N2: 7, N3: ***a***+11***h***
SET19;	N1: ***h***+***c***, N2: 7, N3: ***a***+11***h***, N4: 19
Y11M1;	N1: ***h***+***c***, N2: 7, N3: ***a***+11***h***, N4: 19, N5: **c**
×D; CONT;	N1: ***h***+***c***, N2: 7, N3: ***a***+11***h***, N4: 19**c**
+;	N1: ***h***+***c***, N2: 7, N3: ***a***+11***h***+19**c**
SET433;	N1: ***h***+***c***, N2: 7, N3: ***a***+11***h***+19**c**, N4: 433
JSP204;	N1: ***h***+***c***, N2: 7, N3: (***a***+11***h***+19***c***) ÷ 433
×D; CONT;	N1: ***h***+***c***, N2: 7×((***a***+11***h***+19***c***) ÷ 433)
−;	N1: ***h***+***c***−7×((***a***+11***h***+19***c***) ÷ 433)
=Y12M1;	h := ***h***+***c***−7×((***a***+11***h***+19***c***) ÷ 433)

This code is optimal (P204 adjusts the result of a hardware division to Algol 60 semantics).
■

Historical benchmarks, and recent tests with an emulator [FindlayUG], show KAlgol's object code running 25 times faster than Whetstone's. The latter is 75-100 times slower than hand coding, so we have that KAlgol programs are about 3 to 4 times slower; not bad for an Algol 60 compiler designed in 1963 to run on a 48KB, 1MHz computer with three magnetic tape drives for backing store! It is of interest that KAlgol gets much the same Whetstone benchmark rating [Longbottom] as FORTRAN on the B5500, despite the latter being a machine with a 50% faster core store than KDF9.

The lack of relocatable binary in the TSD/POST/PROMPT software development tool set (see Section 3.3) necessitates a rather clumsy approach to augmenting a user's Algol program with the runtime support routines it needs. The compiler appends its generated Usercode to the source of the runtime support and the combined text is submitted to the Usercode compiler. This incurs the cost of compiling the support routines over and over. The Eldon 2 system cuts out a lot of overhead by appending the *machine code* for the user's program to a pre-compiled version of the most commonly used routines. Absent a linking loader, the entry-points for the latter are set at fixed addresses.

2.3.3: EGDON ALGOL

A new Algol compiler was written by one of the authors of KAlgol, with the benefit of experience and hindsight, for the EGDON operating system. It eschews global optimization. A peephole optimizer gives locally good code [Wichmann73b] that performs overall about as well as that from the Kidsgrove compiler. Combining respectable compilation speed (up to 400 cards per minute) and reasonably fast object code, it also offers useful run-time checking.

The limitations of EGDON's card-oriented character set mean that many Algol basic symbols have to be represented in the ugly 'stropped' format; for example: **if**, punched if for Whetstone and Kidsgrove, is punched 'IF' for EGDON. Several other Algol symbols, including, regrettably, '**;**', and '**:=**', also have to be transcribed for input, although some are printed in the proper Algol form in compilation listings!

EXAMPLE 2.9—EGDON ALGOL'S HARDWARE REPRESENTATION FOR PUNCHED CARDS

```
'REAL' 'ARRAY' X, Y (1:10)£
'INTEGER' I£ 'REAL' SUM£
'FOR' I = 1 'STEP' 1 'UNTIL' 10 'DO'
   SUM = SUM + X(I) * Y(I)
```

■

EGDON Algol is a two pass affair. The first pass generates a reverse Polish form, not unlike Whetstone abstract machine code. The second pass converts that to KDF9 native code. Unlike WAlgol and KAlgol, it produces relocatable object files that can be be linked with the output of EGDON's FORTRAN and UCA3 compilers.

An Algol main program, or a separately compiled procedure, is considered to be nested within an invisible outer block that contains the COMMON and PUBLIC variables of the program, as declared within any FORTRAN modules, or set by the program's PRELUDE or PREDATA sections (see Section 2.4).

2.3.4: ALL THE ALGOLS

There is element of friendly rivalry between the proponents of the KDF9's Algol 60 compilers. Being able to take a disinterested view, I consider all three worthy of admiration: Whetstone for pioneering very fast 'checkout' compilers, Kidsgrove for pioneering effective optimizations of an intractable language, and the EGDON compiler for its practicality and integration with the rest of the operating system. Readers wondering why the speed of Whetstone execution was never boosted, or why Kidsgrove's reliability was not enhanced, must realize that they were written in a different time. The problem was not proprietary code; installations had copies of the source. Rather, it was a matter of available effort, there being perhaps 20 full-time KDF9 system programmers in the whole world. They wrote the TSD, EGDON, Eldon 2, and DEMOCRAT operating systems, the Algol and FORTRAN compilers, and all the other utilities and libraries. A year after the first customers got their KDF9s, EE announced its incompatible successor, System 4, the first of which which was delivered before EGDON 3 or Eldon 2 became operational.

2.4: FORTRAN

A variant of FORTRAN II, called EGTRAN, is implemented on the EGDON operating system. Like most FORTRANs of the era it includes a number of non-portable, but very useful, language extensions. Perhaps the most unusual is that a FUNCTION or SUBROUTINE subprogram can be declared RECURSIVE. In that case its local variables are held in a stack rather than in static storage, permitting the subprogram to be invoked while it is already active.

A very nice feature of EGTRAN, and indeed EGDON Algol, is that of PUBLIC variables. Data is normally shared between separately compiled FORTRAN modules by means of COMMON statements, which attribute variable names to offsets within a block of storage. No check is made that these names and offsets are used consistently in every module, and this is a notorious source of error. PUBLIC, on the other hand, shares variables by giving them identifiers that have the same meaning throughout a program. (PUBLIC originated in Harwell's Atlas FORTRAN; see [Pyle64]. The PARAMETER feature of HARTRAN is subsumed by EGDON's PRELUDE and PREDATA facilities.)

All EGDON compilers produce relocatable object code files that may be combined with library routines to form an executable program. Such a program may include routines originating from source code in EGDON Algol, EGTRAN and UCA3. (Reader, I wrote one [Findlay69].)

In order to prevent the demand for FORTRAN from undermining Eldon 2, by increasing the pressure to replace it with EGDON, David Holdsworth at Leeds University wrote a FORTRAN compiler for Eldon 2. Like Whetstone Algol it limits INTEGER variables to 40 bits, so they can be represented exactly in floating point. This greatly simplifies code generation. It is written in KAL4 and produces KAL4 object programs, automatically invoking the KAL4 compiler to assemble its output.

Eldon 2 FORTRAN deals with the problem of NEST overflow in complicated expressions by the masterly tactic of ignoring it. Only one program has ever been found to exceed the 16 available cells!

2.5: IMP

The British 'Autocode' languages have their genesis in R.A. Brooker's early compiler for the Ferranti Mark I at Manchester [Brooker58]. Its successor, Mercury Autocode, also due to Brooker, added major new language features such as 'for' loops. Mercury Autocode then begat Atlas Autocode, seen by its proponents as a more practical alternative to Algol 60. It has nested scopes, like Algol, but passes parameters by reference, like FORTRAN. At Edinburgh University Atlas Autocode was bootstrapped from the Atlas to their KDF9 [BRSW65]. KDF9 Autocode was then used to write the compiler for an improved and modernised Autocode, IMP [Stephens74]. IMP is aimed at systems programming, and was used to write the portable EMAS multiaccess system, initially for the EE System 4-75.

2.6: Other programming languages

Despite EE's promises, KDF9 never got a compiler for COBOL, nor an optimising FORTRAN compiler [EEFORT], but compilers for some lesser languages were written.

ALPHACODE [EEALPHA] is a rather primitive adaptation of a Brooker-style Autocode to the fixed-format card input and limited character set of the first generation EE DEUCE computer; there is a version for the KDF9 [EEL66].

Babel [Scowen69], written at NPL, is an Algol-like language for KDF9, 'with many of the features of Algol W'.

A non-interactive BASIC compiler was written at NPL by Tony Hillman for Eldon 2 [*pc*]. Like Eldon2 FORTRAN, it translates source programs into KAL4.

K Autocode [Gibbons68]—*not* KDF9 Autocode—is an independent implementation of Mercury Autocode, made at ICI for the KDF9. It is said to have dispelled any demand for Algol 60.

STAGE2 [PW70] is a powerful macro-processor written by Waite as the second step in bootstrapping his Mobile Programming System. It was used on the KDF9 to implement the portable MITEM family of text editors [Poole71].

3: The English Electric Time Sharing Director (TSD)

Controlled access to I/O devices, I/O-driven dispatching, hardware context switching, and program relocation hardware, combine to make multiprogramming ('timesharing' in EE parlance) on KDF9 both efficient and secure. Programs communicate with Directors by means of the OUT instruction, KDF9's system call operation, placing the number of the requested service in N1 and any further parameters in N2 and N3, as needed. EE offer a non-multiprogramming Director for machines with too little store for effective multiprogramming, but the more practical option with sufficient store is the Time Sharing Director. For more detail see [EEL64a], Section A1021.

3.1: An overview of TSD

Pre-emptive, priority-based multiprogramming is directly supported by the hardware for up to four programs, with interrupts from IOC signalling the need to re-dispatch, and even the need to deal with a priority inversion. Four active programs are usually as much as can be fitted into 32K words, so the limitation of the multiprogramming hardware to four levels is seldom a bottleneck. It could perhaps be overcome by having Director share the lowest priority level among several programs. That was tried, but not found to be of much benefit. The four available program 'slots' are designated P, Q, R and S in messages from TSD.

Job control is shared between Director and the operators. Programs are divided into two job streams, **A** and **B**. A-programs are loaded automatically. At boot time, the operators define the maximum resource demands of an A-program, in terms of core store and I/O devices that may be used. The operators also define the maximum number of A-programs that may be run concurrently, and assign them to priority levels. Director loads programs to populate the specified levels. When an A-program terminates, any A-programs in lower priority levels are promoted. When sufficient resources become available, a new program is loaded into the vacated, lowest, A-program level.

B-programs are loaded in response to operator commands that specify a priority level and a main storage allocation. A 'call tape' is input from a paper tape reader. This gives the name of the program to be run, a code for the device from which its object code is to be read, and a free-text 'title' to be logged on the Flexowriter. A device code of 'P' calls for the program to be read from a tape reader; 'M' calls for the magnetic tape currently nominated for B program input; 'D' calls for the program library on the drum (if fitted) or the disc drive. A program can also be loaded in response to a programmed overlay request (OUT 1); in this case there is also a 'U' option, which specifies the program held in the requesting program's temporary allocation of disc storage.

There is a conventional format for program names, made up as follows:

- characters 1 and 2 are a unique two-letter code allocated to the installation by English Electric
- characters 5 through 7 may be freely chosen by the programmer as a personal identifier
- characters 8 and 9 are a two-digit version number; this is incremented each time a program is amended by POST or PROMPT (see Section 2.3) and rolls over from 99 to 00

• character 10 is a language code, e.g. ‘U’ for Usercode and ‘A’ for Algol

• character 11 is ‘P’ for ‘program’

• character 12 is a digit in the range 1 through 8, the minimum number of 4KW blocks of core store needed; or ‘U’.

The program name on a call tape may include ‘**–**’ characters, which act as wild cards; Director will load the first program it encounters on the specified medium that exactly matches the non-wild characters. (However, the KDF9 Programming Manual [ICL69] specifically recommends that this should not be done.)

Store is allocated to programs using the classic ‘two-stack’ ploy. B-programs are loaded above Director and towards higher addresses; A-programs are loaded at the top of the store, and towards lower addresses; free space is consolidated into the gap between the A and B allocation regimes. When a program that is not adjacent to the gap terminates, the other programs in its regime are moved so as to slide the freed storage into the gap. It is possible to move programs around in this way because they work entirely in terms of virtual addresses, their access to physical addresses being managed dynamically by the relocation hardware. There is one complication: after an I/O transfer has been initiated the buffer works in terms of physical addresses, so Director must wait for all ongoing transfers in a program’s area to terminate before being able to move it.

Compiled programs are held in a simple sequential format consisting of an ‘A’ block, a ‘B’ block and one or more ‘C’ blocks. An A block is essentially the same as a call tape, but without wild cards. The B block tells Director how big the program is, what its execution time limit should be, how to jump to its first instruction, how to restart it in the event of a recoverable error, and so on. The C blocks contain the executable instructions and initialised data areas. For more details, see [ICL69], Section 26.3.

There are two significant aspects to the management of I/O devices on KDF9: their allocation to problem programs by Director, and their control by problem programs once allocated. Though program logic is coupled to device type, it is decoupled from device identity. To obtain access to a device, a program asks Director to allocate it one of the type, by placing the type code in N2 as parameter to an OUT 5 system call. If such a device is available, Director gives it to the program and returns its buffer number in N1. The program must save this number for future use. As soon as a program is finished with a device it should relinquish it, to maximise resource sharing. This is done by OUT 6, with the device number in N2. Magnetic tapes are handled slightly differently: a tape deck is allocated according to the label of the tape loaded on it, but the logic is otherwise similar.

See Table A. The code 67 is used only to give a tape-labelling utility access to a deck which contains an empty tape, something that is not normally permitted, as it prevents programs from erroneously trying to read non-existent data.

TABLE A: TSD DEVICE TYPE CODES

Type:	**FW**	**TP**	**TR 8-hole**	**LP**	**CR**	**TR 5-hole**	**CP**	**MT**	**GP**	**MT unlabelled**
Code:	0	1	2	3	4	5	7	10	16	67

There is an output spooling facility implemented by Director, often called ‘OUT 8’ after the relevant system call. This is the *modus operandi* that maximizes the scope for multiprogramming, as it decouples the workload from the availability and slowness of physical output devices. No input spooling is available.

Spooling provides up to 32 virtual output streams to each running problem program; see Table B. Output is written first to a designated magnetic tape. When that becomes full, or at the behest of the computer operator, it is rewound and the accumulated data for each stream is transferred by a utility program from the tape to its associated device. A second magnetic tape may be designated to take output while this is happening, and the two tapes may switch rôles as often as necessary to transfer all the outputs to their intended destinations.

OUT 8 takes a parameter word in N2 that controls the spooling action. It is in Q store format. If the C-, I-, and M-parts are equal, and contain a stream number, that stream is closed. Otherwise the I-part is taken to be the starting address, a, of an output area, and the M-part is taken to be its ending address, b. If $b = a + 1$ and the area is laid out as follows:

• word a+0: *stream number*

• word a+1: control word in Q store format: 4095 / -1 / n

then a device-dependent ‘gap’ is written to the stream; for a tape punch stream this takes the form of runout (unpunched) tape, of length n characters, for n / 10 inches; applied to a line printer stream it produces a Page Change character. Otherwise, and more usually, the words in address range a to b inclusive are written by Director to the OUT 8 tape using a transfer-to-End-Message operation, the contents of word a having been replaced with data that more fully identifies the stream.

Special provision is made for prompted responses from the Flexowriter console, which are implemented immediately, not deferred. In this case the parameter word in N2 must have a C-part of #100000. The output area must be laid out with a prompt (ending with a ‘**;**’ character) and be followed by an input area for the response. Various restrictions are imposed: a message must not be longer than 8 words, it must contain neither LS nor tab characters, nor ‘**;**’ in the last word, nor ‘**;**’ other than in character 7. Word a+1 is overwritten by Director with suitable format effectors to preserve a tidy layout of the console’s typed log. The restriction on Horizontal Tabs allows Director to allocate columns across the page to distinct functions. Its own output is left-adjusted; output relating to the program in slot P is indented by one tab, that for Q by two tabs, and so on.

TABLE B: TSD OUT 8 STREAMS

Stream Number	**Destination Output Device**
#10 .. 17	A paper tape punch selected by the operator, in 8-hole mode
#20 .. 27	Not used
#30 .. 37	The line printer
#40 .. 47	Not used
#50 .. 57	A paper tape punch selected by the operator, in 5-hole mode
#60 .. 67	Not used
#70 .. 77	The line printer **or** a paper tape punch, at the operator’s discretion

At any one time the Time Sharing Director may be carrying out operations on behalf of up to four programs, as well as internal operations to implement spooling, program loading, operator communication, and so on. To allow these operations to proceed concurrently, an internal multitasking system is implemented. Ten Director threads are supported, two for each program and two for Director itself. Thread scheduling is co-operative, with threads yielding control when they are unable to proceed.

Interrupts that cannot be serviced in the 'short path' make requests for service from a thread. Director runs threads until all of them are blocked, at which point it dispatches the highest priority unblocked program. If all program levels are blocked, Director twiddles its thumbs in a loop that scans the I/O devices for status changes, and keeps the watchdog timer at bay.

Director sums the lengths of a program's spells in charge of the CPU; the total is its **run time**. Its **elapsed time** is given by the 'wall clock' time that has passed since the start of its execution. The difference between these values represents time when the program was not running: it was either waiting for I/O to finish; or was ready to run, but a program of higher priority was running instead. The latter circumstance is out of its control, so, to give a more useful measure of its effectiveness in overlapping I/O with computation, Director also calculates a program's **notional elapsed time**. This estimates the elapsed time a program would have achieved, if run without competition for the CPU from programs of higher priority. On request from the computer operators, Director can exchange the priority levels of a pair of programs, if it seems that one of them has a glut of CPU time while the other is being starved. Since a program's priority determines the PHU that is used by IOC to update its dispatching status, Director must wait for the end of all the pair's I/O transfers before swapping their priorities.

As well as I/O management, Director facilitates program overlay and provides timing services. See Table C, and [ICL69], Sections 17.3 and 26.2.

TABLE C: TSD OUT PARAMETERS

N1	N2	N3	Action
0			Terminate this run normally.
1	Program ...	… name	Terminate this run and overlay the program whose name is in N2-3.
2	Time limit in seconds		Restart this self-overwritten program. Allow it the new time limit given in N2.
3			Return the CPU time used so far in N1. The result is given as seconds to 23 integral places.
4	Tape label		Allocate the tape deck with the reel having the 1-word label in N2. Return its buffer number in N1.
5	Device type code		Allocate an I/O device of the type given in N2; see Table B, Section 3.1. Return its buffer number in N1.
6	Buffer number		Deallocate the I/O device with buffer number in N1. If it is a tape deck, unload any mounted tape reel.
7	Buffer number		Deallocate an allocated tape deck with buffer number in N1. Do not unload any mounted tape reel.
8	*-/a/-*		Spool output. See Section 3.1.
9			Return the time of day in N1. The result is given as seconds since midnight to 23 integral places.
10	Tape ...	… label	Allocate the tape deck with the reel having the 2-word label in N2-3. Return its buffer number in N1 and its 'Tape Serial Number', as read from the label block, in N2.
11	*s:u/a/b*		Write words *a* .. *b* to the drum starting at sector *s* of unit *u*.
12	*s:u/a/b*		Read words *a* .. *b* from the drum starting at sector *s* of unit *u*.
13	*n*		Reserve *n* drum sectors (once only).
14			Return the number of drum sectors already reserved in N1.
15	*c*		Load the program starting at address *c* to the drum.
17			Return the CPU time used so far in N1 and the notional elapsed time in N2. The results are given as seconds to 23 integral places.
41	*s/a/b*		Write words *a* .. *b* to the disc at logical block:sector *s* in the current set.
42	*s/a/b*		Read words *a* .. *b* from the disc at logical block:sector *s* in the current set.
43	*0 or 1*		Select the first (0) or second (1) set of disc platters.
44	*n*		Reserve *n* disc platters. May be done twice to allocate two separate sets.
45	*0 or 1*		Relinquish the first (0) or second (1) set of disc platters.
46	*c*		Load the program starting at address *c* to the current set's program slot.

3.2: OPERATING A KDF9 WITH TSD

It is illuminating to see the forms of interaction afforded by TSD with its operators. To some extent it was TSD itself that called the shots. The following material draws heavily on the EE document 'A1020 – Time Sharing Director – Mark I', [EEL64a], by A. Doust and M.R. Wetherfield, themselves the authors of the Time Sharing Director.

Commands from the computer operators to Director are known as 'TINTs', short for Typewriter Interrupts. They are initiated by pressing the interrupt button on the console Flexowriter. Director responds by typing `TINT;` and waiting for a command to be input. Each such command is terminated by typing '.→'.

In the following list, *dd* indicates a two-digit **d**evice (buffer) number; *tt* indicates a device **t**ype code; *p* indicates a 12-character **p**rogram name (also known as the PRN, or Program Reference Number); *n* is an octal **n**umber; *l* is a magnetic tape **l**abel; and *s* is a multiprogramming **s**lot name (one of P, Q. R or S).

A *s*.→
Terminate program in slot *s*, or terminate its loading.
A *s*+.→
Terminate program in slot *s* and type a basic diagnostic, showing the top of the SJNS and the NEST..

B *s n*.→
Read the octal integer *n* (up to 8 digits) and store its value into the less significant half of E0 of *s*. This can be used to convey options to the program at run time, separately from any tape or card data.

C *dd*.→
Load the magnetic tape on unit *dd*. Director reads the tape label, notes it, and positions the tape at the block immediately following, ready for allocation to a program that requests it by name, using OUT 4 or OUT 10.
D *dd*.→
Unload the magnetic tape on unit *dd*. The tape is rewound and disengaged from the deck, ready for removal.
E *dd***A**.→
Nominate the magnetic tape loaded on unit *dd* for the input of A programs.
E *dd***B**.→
Nominate the magnetic tape loaded on unit *dd* for the input of B programs.

F.→
Dummy (used if interrupt key pressed in error).

G.→
Type the peripheral unit list.
H.→
Type a list of 'wanted' tape labels. These are tapes that programs have requested, but have not yet been loaded.

I *s***0**.→
Even restart for program *s*, i.e. make it execute the jump in the first halfword of word 4. This, and the odd restart, are for use when a program fails in a manner that has been anticipated, and can recover from.
I *s***1**.→
Odd restart for program *s* , i.e. make it execute the jump in the second halfword of word 4.

L *dd*/*tt*.→
Change the peripheral device unit *dd* to type *tt*. If *tt* = 0, the unit is deleted from the free list.

M *x*{*s*?}*y*.→
Output a store print in syllabic octal. *x* and *y* are both octal integers. *y* words are output, starting at address *x*: if *y* is omitted, 1 is understood. If {*s*?} is P, Q, R or S then the base address of *s* is added to *x*. Any other separator (e.g. '**/**') gives the absolute (Director) address *x*.

R *s*.→
Resume *s* (after suspension).
S *s*.→
Suspend *s*.

T *n*/*w*.→
Load a B-program and give it priority level *n*, with a store limit of *w* words. Absence of *w* implies that the program may want the whole store. If the **/** is replaced by **P** the paper tape reader from which its A-block is read is pre-allocated to the program. A-programs are loaded on the initiative of Director, not the human operators. When the resources needed for another A-program become available, Director says:

n tt P *dd s*

n APIU *dd s*

The operator must present the program's A block on the **A**-program **i**nput **u**nit *dd*. Director reads it in and continues:

* *date time*

n s P *p* **or** *n s* M *p*

depending on whether the program itself is read from paper tape or magnetic tape.

If something goes wrong while reading in a program, Director types:

n CRNP F*x*

where: CRNP means 'Can't Read New Program' and *x* is an error code.

U.→
For each program in the machine, type its slot letter (P, Q, R or S), 12 character identifier, priority, base address, and running and elapsed times so far.

V *n*/*n*'.→
Put program in priority level *n* into priority level *n*', and vice versa.

Z *dd*.→
Relabel the magnetic tape on unit *dd*. Director issues the query `TN/ID;` to which the operator replies `N/`*l*.→, e.g.:

```
TN/ID;N/+DIRECTORBINWORK.→
```

In the event of a problem program failing, Director types:

n s `FAILS` *indicator ...*
`REACT`;

This gives the operator the opportunity to abandon, or to restart the run. The *indicator* is a two or three digit code, e.g.: `00L`—Lock-in violation, `00N`—NOUV, and `00T`—Time limit exceeded.

When a program finishes without detected failure, the following message is typed out:

n p///*e*/*run time*/*n.e.t.*/*elapsed time*

where: *run time* is the CPU time used; *n.e.t.* is the notional elapsed time; *elapsed time* is the 'wall clock' time since the program started; and *e* is the 'ending number', which gives the reason for termination.

EXAMPLE 3.1—RUNNING AN ALGOL PROGRAM UNDER THE TIME SHARING DIRECTOR
The following shows the **ee9** emulator running an Algol program under the TSD, its OUT 8 output being spooled to a magnetic tape, then printed down from the tape file by a modern utility. The source code, given as an example in the Algol Manual [EELM69, Appendix 1], was compiled by the resurrected Kidsgrove compiler.

```
/Users/wf/KDF9/emulation/Testing: tsdnine FLUID FLUID_data
This is ee9 V3.3d, compiled by GNAT Community 2019 (20190517-83) on 2019-09-24.
Booting the KDF9 Director Binary/KKT40E007UPU in fast mode (no tracing).
P
KKT40E007UPU
TIME SHARING DIRECTOR  2464 WDS|
02U01
02U02
05U03
01U04
03U05
10U07
10U10
10U11
10U12
10U13
10U14
CORE MODULES;8.|
OUT 8 REEL NO;23001C.|          the operator tells TSD how to label the spooled output tape
A-PROGRAM DETAILS|
LEVELS;N.|
DATE  D/M/Y;23/9/69.|
TIME ON 24-HOUR CLOCK
HOURS/MINS;10/29.|

TINT;T0/30240.|                 the operator tells TSD to load a program, giving it 30240 words of core

10L14 /Iden<+KOUTPUT/0023004>,TSN -00-2339
10L13 /Iden<MS-DUMP.>,TSN 77777777
10L12 /Iden<EFPBEAAG>,TSN -00-0552
10L11 /Iden<ZERO>,TSN 00000000  this is a 'zero' scratch tape on MT1
10L10 /Iden<WHETLIST>,TSN -00-1234
     0 BPIU 02P                 device 2 (TR1) is used to input the B-program

     0º23/09 1029
     0ºP P FLUID3000UPU         the program has been loaded into timesharing slot P, priority 0
     0 02A02P                   TR1 is allocated to the program by OUT 5
02U02                           having read its data, the program deallocates TR1 by OUT 6
```

```
10C11D                                  TSD claims the scratch tape on MT1 for OUT 8 spooling
     0/FINISHED ENDS 0|                 the program writes its farewell message to the FW
     0   OUT 8 P0000001/0023001         TSD identifies its OUT 8 spooling session
     0 FLUID3000UPU     ENDS 0          TSD terminates the program as requested by OUT 0
     0ºRAN/EL/000M29S/000M29S/000M00S

TINT;J11.|                              the operator closes the OUT 8 session on MT1

10L11 /Iden<+KOUTPUT/0023001>,TSN 00000000   TSD relabels MT1 for retention (not scratch)
…
/Users/wf/KDF9/emulation/Testing: mtp MT1P30
Despooling OUT 8 stream 30 for slot P from MT1

TAPE LABEL TSN '00000000', IDENTIFIER '+KOUTPUT/0023001'

SESSION P000001

FLUID3000UPU   23/09/69
  STR30
PROPAGATION OF AN IMPULSE INTO A VISCOUS-LOCKING MEDIUM



    DELTA ALPHA = 0.1000


    ALPHA              LAMBDA       LAMBDA.SQRT(2.ALPHA)        G
    0.0000            INFINITY            1.00000          1.00000
    0.1000          2.0201º +0            0.90340          0.73452
    0.2000          1.2812º +0            0.81029          0.52525
    0.3000          9.2712º -1            0.71814          0.36101
    0.4000          6.9943º -1            0.62559          0.24038

    0.5000          5.3160º -1            0.53160          0.26796
    0.6000          3.9727º -1            0.43519          0.30384
    0.7000          2.8340º -1            0.33532          0.34489
    0.8000          1.8242º -1            0.23075          0.38947
    0.9000          8.9284º -2            0.11979          0.43642

    1.0000          0.0000                0.00000          0.48394
FINISHED ENDS 0
```

■

3.3: POST AND PROMPT

EE offers a basic programming development environment known as POST. See [ICL69], Section 30. It is helpful to describe it here, as a preliminary to an account of the Eldon 2 operating system.

POST allows for the creation, storage, editing and compilation—without operator intervention—of batches of programs held on magnetic tape. When disc drives became available, POST was superseded by PROMPT [EELM67], which offers similar functionality but stores programs for direct access in a simple filing system.

It is important to understand that EE compilers do not generate relocatable object code, but go directly to executable binary. More precisely, the Usercode compiler does so, the Kidsgrove Algol compiler generates Usercode, and Whetstone Algol generates and interprets code for an Algol-oriented abstract machine. This means that with POST and PROMPT there is no support for object module libraries. Instead, library subroutines are held in source code form, and textually inserted into client programs at compile time. See [ICL69], Section 14. A set of libraries is provided, some in Algol and some in Usercode, each containing a different selection of procedures. It is not possible to request the inclusion of just one procedure; the whole library is inserted, raising some of the same issues as '`#include`' files in C.

An interesting aspect of POST and PROMPT is that they do not store program texts in the native 6-bit character code. Instead they make use of an 8-bit code in which each 'Algol basic symbol' [Naur60] is represented by a single 8-bit byte. This means that Algol reserved words such as **begin**, and **array**, are stored very compactly, but identifiers and numeric literals take up a third more space. Programs in Usercode are held in the same way. This moves some of the lexical analysis of programs from the compilers to the source code input and editing suite.

A few non-standard reserved words are added to those required by Algol: **KDF9**, to introduce procedure bodies written in Usercode; **ALGOL**, to revert from embedded Usercode to standard Algol; **library** to call for the insertion of source code library routines; **segment** to break a large program into overlays; and **EXIT** to effect a return from an embedded Usercode procedure. Each has an Algol basic symbol (ABS) code.

Example 3.2 is taken from [ICL69], Section 30, with corrections for obvious typographical errors. It begins with the 'call tape' requesting Director to load the POST program, `KAB00050lUP3`, from the current program library reel. The top-level commands, `ASSEMBLE` and `TRANSLATE`, appear without decoration; `ASSEMBLE` in this context means 'gather together', not 'translate assembly

code'. What follows each is a series of directives introduced by an underlined End Message character, ⇒. Under each directive, requests beginning with a plain → character detail the work to be carried out. A PROMPT run would look essentially similar.

EXAMPLE 3.2—POST EDITING AND COMPILATION

```
M
KAB00050lUP3                        call-in the POST program from the magnetic tape program library

ASSEMBLE; FROM <POSTOUl2>;          reel-to-reel update: name the input magnetic tape reel
ONTO <POSTOUl3> EX <POSTOUl3>;              name the output reel for the amended sources
L;8;→                                       write a log via OUT 8

⇒CORRECT KEZGL0403UPl;              edit an existing program file
O/P L, 8;→
→CONNECT                                    log what follows
→INCLUDE LIBRARY
→AFTER LINE Vll=                            move to context
→DELETE 1 LINE                              delete 1 line
→INSERT Vl2 = P⌊7DC⌋;                       insert a new line

⇒ESTABLISH KESAOOlOOUPl;            create a new Usercode program file
SAMPLE;
O/P L,8;→
V23;                                        new program source code starts here
PROGRAM;
   V0 = B30;
   … etc ...
   J9;
library L59                                 copy library L59 into the program text at this point
FINISH;
→                                           new source ends
⇒

⇒DELETE KEFAB00l9UPl                delete an old program file

⇒COPY REST                          copy all other program files from the input reel to the output reel

TRANSLATE; FROM <POSTOUl3>;         perform compilations of newly 'assembled' programs
ONTO <POSTOUll> EX <POSTOUll>;              name the output reel to hold the object programs
M4;
L;8;→
PROGRAM KEZGL0403UPl;               a program to be compiled
USERCODE;                                   in Usercode
WITH TABLES;                                print symbol tables
STl024;TL3;                                 store and time limits to be embedded in the object program
O/P L, 8→

PROGRAM KEDATAA04AP2;               another program to be compiled
KIDSGROVE;                                  in Kidsgrove Algol
O/P L;→

END TRANSLATION→
END POST→
```

The verbose editing commands, such as `AFTER LINE Vll=` and `DELETE 1 LINE`, can be abbreviated in practice, e.g. to `AL Vll=` and `D 1 L`.

■

4: ELDON 2

The EE Time Sharing Director is most effective with a workload consisting of relatively long-running jobs. With some I/O-limited jobs and CPU-limited jobs to play against each other, it is capable of making very good use of the machine's resources. However, many KDF9s were supplied to universities, where the mix of short experimental runs and many failing (student) compilations requires far too much operator intervention. It was to tackle these difficult workloads, and to offer multiaccess working, that Eldon 2 [WHMcC71, Holdsworth09], an operating system based on an enhanced version of the Time Sharing Director, was developed at Leeds University. It uses a DEC PDP-8, connected via a magnetic tape buffer, as a front-end processor to handle the terminals.

4.1: AN OVERVIEW OF ELDON 2

Eldon 2 was written by a small team with many other duties. This encouraged the reuse of as much of the existing software as was thought to be good enough. Anything not thought good enough was rewritten or replaced. So Eldon 2 retains an enhanced Time Sharing Director, and elements of the PROMPT system, including its ABS code—which is used consistently throughout—and its compilers. The PDP-8 transliterates between ABS and ASCII. The PROMPT disc file structures are retained, but managed by Eldon 2 commands, including a new text editor. The PROMPT filing and editing suite is eschewed: at around 13KW its component programs are far too big for multiple online instantiation.

PROMPT represents a file as a linked list of blocks, the first of which contains such metadata as the name of the file, the identity of the owner, the date on which it was last written, and the addresses of its constituent data blocks. Further blocks contain the program text as lines in ABS code, except that the first byte of each line contains its length in words and some flag bits. Users were able to read, but not to write or delete, the files of other users. A special user called `SECRET` existed, and was somewhat like the modern concept of a 'super user'. The lack of more elaborate access permissions is deliberate; the authors of Eldon 2 had never witnessed malicious behaviour and felt that such controls were not needed! I wonder whether they would take the same view nowadays.

4.2: The implementation of Eldon 2

The multiaccess service provided by Eldon 2 can best be described as conversational remote job entry (CRJE). Compilers and user programs are not run interactively under console control. Users log in, input and edit programs and data files, and submit requests for compilations and runs to either one of two queues. The foreground queue is intended for rapid turnaround of small jobs that are prioritised over everything other than Eldon 2 itself. When a foreground job is put into the queue, the requesting terminal goes unresponsive until the job terminates and its output is sent back. Bigger jobs may be submitted to the background queue, and are run when the resources they demand become available.

Eldon 2, properly so-called, is an adjunct to Director. It runs as a threaded problem program in multiprogramming slot 'S' at priority level 0. It communicates with the PDP-8 to process commands from interactive users and send back their results. A core-resident global workspace and a small set of resident utility routines service a swappable reentrant code segment and a swappable per-user area of 760 words that effect a user's file handling and job submission requests.

Multiprogramming priority levels 1 and 2 are used to service the foreground and background queues. Level 3 is given to a base load program managed by the computer operators; it is never swapped out, and soaks up any CPU time not needed by the online system. Foreground jobs are also locked in core for their duration of at most 20 CPU seconds.

The base load runs somewhat like a Director 'B' program, and queue jobs somewhat like Director 'A' programs, though initiated at the behest of Eldon 2 rather than Director. The human operators retain the power to change the priorities of programs at their discretion, for example when boosting the execution of a base load job would hasten its completion, freeing resources for work of intrinsically greater urgency.

When a user program terminates, or shrinks its store allocation, a Job Organiser utility is run to initiate a new foreground or background job. If necessary, it temporarily swaps out the background job to free enough store to let a waiting foreground job start. It then assumes the identity and characteristics of the new job and reads it in from disc, overwriting itself. In later versions of Eldon 2 a 'cafeteria' service is run by the Job Organiser in competition with the background queue; it lets students compile and run small Algol 60 programs from a 'live' tape reader and small FORTRAN programs from a 'live' card reader, swapping in the appropriate compiler on demand.

There is a nice hierarchy of schedulers, each imposing a load compatible with the time scale on which it operates: the Director 'short path' responds to interrupts and delegates to Director threads; Director delegates to Eldon 2; and Eldon 2 threads delegate to the Job Organiser. Beyond Eldon 2, humans are the scheduling authority of last resort.

OUT 8 de-spooling under the EE TSD is effected by running a problem program to read a full output reel and copy its contents down to the line printer and paper tape punch. Eldon 2 cannot afford the core store nor the program slot that this would require, so de-spooling is done by an Eldon 2 thread that runs when there is no other work making more pressing demands. When even that thread runs out of work to do it executes an INTQq instruction, thereby yielding the CPU to lower priority levels, and being resumed when any of its transfers terminates.

An aside about the KDF9 disc drive (a Data Products dp/f-5022, or similar) is useful here. It has fixed platters recording data on both surfaces. Each surface has 128 tracks in an outer zone, and a further 128 of half the capacity in an inner zone working at half the transfer rate. An outer zone track is divided into 16 individually addressable sectors of 320 KDF9 characters (240 bytes); an inner zone track has 8 sectors. Each of the 16 platters has an arm carrying 8 read/write heads: 4 for outer zone tracks arranged 2 per surface on each of 2 distinct tracks, and 4 similarly arranged in the inner zone. This means that there are 64 seek areas on each platter. Within a seek area all 96 sectors can be accessed without further head movement. Platter 17 has fixed heads, giving 'fast' seek-free access to up to 96 sectors.

EE Directors allocate disc space in sets of whole platters, with 'logical blocks' of 640 words arrayed across the platter, then across the next platter down, and so on, sequential access requiring a seek after every three logical blocks. Eldon 2 transposes this arrangement, allocating space in terms of cylinders, consisting of the same head position on each of the platters. Several cylinders are set aside for use by the system, and for users as 'logical discs', allocated in quanta of half a cylinder. The rest is available for file storage. In both the system and user regimes, space is managed in 640 word blocks, for compatibility with PROMPT. The fixed head platter holds the free-block bit map and an index of programs, such as the Job Organiser, that Director can load and run without file system intervention. Disc space to hold Eldon 2's per-user context amounts to 19 sectors; so data for 5 users takes up 95 of the 96 sectors in a seek area and all 32 user contexts occupy less than half a cylinder.

File storage is backed up to magnetic tape; this is combined with a facility for the automatic off-lining and on-lining, on demand, of seldom used files. For more detail about the implementation of Eldon 2, see [Holdsworth09].

Some statistics indicate the success of the system: the enhanced Director occupies 3200 words, Eldon 2 gets the next 2720, and the remaining 26K or so—over 80%—is available for running user programs. Serving up to 32 terminals on a computer with 192KB of core store and two slow disc drives, Eldon 2 regularly achieves a CPU utilization of 85%.

5: DEMOCRAT

Microkernel systems are distinguished from 'monolithic' kernel systems such as Linux in devolving most of the work of the operating system from the kernel to trusted processes. The microkernel restricts itself to those few activities that cannot sensibly be devolved. Typically they include responding to interrupts, maintaining the security barriers between processes, dispatching the CPU to the highest priority ready process, and providing a means of inter-process communication. The thinking is that kernel code executes in a non-standard environment, which means that, among other things, it is not accessible to the usual debugging techniques. Minimising its scope means maximising the OS code that can be created, and debugged, using ordinary tools.

Such a system for the KDF9 has been described: DEMOCRAT [Wichmann69, Knight68] is a modular, microkernel based operating system developed at the National Physical Laboratory according to design concepts by J.L. Martin [Martin66]. It is remarkable for being perhaps the first of its kind—an extraordinary, pioneering, achievement that precedes (and may have influenced) the more famous RC400 system [Brinch Hansen70]. It deserves to be much better known.

In DEMOCRAT ("Disc Equipped Modularly Organised Computing with Remote Access and Timesharing"!) the microkernel is known as the "Interface". Processes that run under its control communicate with the Interface using OUTs. On a per-process

basis these can be DEMOCRAT OUTs, TSD OUTs, or EGDON OUTs. DEMOCRAT can pass TSD and EGDON OUTs to server processes to be actioned in the manner of those systems. DEMOCRAT OUTs serve to request system services, a few of which are implemented by the Interface and the rest passed again to appropriate server processes. By this means processes can jump from one code module to another (with semantics like a Unix `exec` system call), or invoke another module as a subroutine that returns to its calling module when complete.

Modules are equivalent to problem programs in the TSD: they run in an unprivileged, interruptible mode, and are restricted to their own virtual memory by suitable values in the BASE and NOL registers. DEMOCRAT exploits the latter somewhat more flexibly than does TSD, allocating space to a module that starts below, and is contiguous with, its BASE point, and extends above, and is contiguous with, its NOL point. These areas contain administrative data associated with the module, which therefore does not need to be stored in tables within the Interface. The NOL of a trusted module can be set high enough to encompass another module; for example, the loader module may work in this way, being thus enabled to read a program into core and set it up for execution as a new module.

A process in DEMOCRAT is a flow of control that passes through one or more modules and requests the services of other modules by means of a simple message-passing interface. A process is dispatched when it becomes the highest priority process that is free to run; DEMOCRAT expects to run many more than the four processes supported by the TSD, so it makes little use of the KDF9's timesharing hardware. Priorities are adjusted dynamically depending on the intrinsic urgency of a task and the amount of time it has spent waiting, using a parametrised scheduling algorithm.

A trusted module is allowed a superset of the DEMOCRAT OUTs normally available to users' programs. These allow such a module, for example, access the disc drive in order to provide file system services.

6: EGDON AND COTAN

In the early 1960s the United Kingdom Atomic Energy Authority's Culham and Winfrith laboratories had KDF9s lacking the timesharing hardware option, and so could not use the TSD. The operating system they commissioned for them from English Electric was named after the area around Winfrith, called Egdon Heath in a novel of Thomas Hardy. EGDON [EELM66a] and [EELM66b] began as a system of a type familiar on other scientific computers of the day. Broadly comparable with IBSYS for the IBM 7090, it aims to provide a programming environment that is somewhat compatible with FORTRAN on IBM's 'scientific' machines, and looks to maximise the throughput of relatively long running scientific codes. Unlike the TSD, EGDON requires the KDF9 to be equipped with a disc drive. Its final version, EGDON 3, could use multiple register sets when they were available.

6.1: EGDON

A paper [BHJV66] describing the first version of EGDON presents it as an automated program development system. The authors distance themselves from the idea of 'batch' processing, which they associate with running magnetic tapes full of jobs through one system program after another in the manner of POST. They are liberated from this mode of operation by the use of disc files and punched cards, which make it easier to edit programs and data. Decks of cards containing system commands, source programs, and application data, are loaded into the hopper of the card reader, read by EGDON and processed at once. In practice things are not always so immediate: EGDON spools both input and output in its normal mode, though it can run in a mode with direct input/output when faster turnaround is called for.

In the earliest versions of EGDON the disc was used only for the system's purposes, for temporary workspace, and for the storage of executable programs. Users had to keep their source code and data on punched cards. Later versions allow users' source and data files to be held permanently on disc.

EGDON revolves around the job deck: a pack of cards, or a file of card images, containing a sequence of commands, source code and application data, submitted as a unit of work. A program called JOBORGANISER reads the job deck and arranges for the execution of such system and/or user programs as it calls for. One such job at a time is active in the system. Unlike Eldon 2 EGDON does not multiprogram; rather it tries to finish each job as quickly as possible by throwing the full resources of the system into the effort.

The most profound way in which EGDON differs from the TSD/POST/PROMPT package is its use of relocatable binary (RLB). The object modules comprising a program are converted to a single executable binary by a linking relocator known as the Composer. The resulting load module may be filed, to be run in future by a `*BINPROGRAM` command, or run just the once and then deleted. A complete program can also be held in an RLB file, to be run by a `*DISCPROGRAM` command, which entails an invocation of the Composer to construct the executable. The advantage is that individual RLB modules can be added, deleted, or replaced with freshly-compiled versions, before the run, making this format suitable for programs under intensive development.

A further element of flexibility is afforded by the `*PREDATA` and `*PRELUDE` features, which enable variations to be made in an RLB program at the Composer stage. For example, global variables can be given preset values, and global arrays can be sized suitably for the particular run. The `*PRELUDE` command lets this be done by running what is effectively a FORTRAN program that can make arbitrary calculations and initializations. The `*PREDATA` command allows just declarations and assignments and is implemented more efficiently.

In recognition of the limited core store of the KDF9, EGDON allows for a program to consist of up to 20 'chain segments' or overlays. Each segment is in effect a program which performs part of a job's work, then may be replaced by another segment. Routines can be declared part of a 'global' segment, which is not overwritten on segment change. The global segment would normally include a routine to call the chain segments in turn. EGTRAN COMMON and PUBLIC variables, and the local variables of global Usercode routines, also lie outwith the overlay area, and so allow communication between segments.

The file system enables a very useful feature of the job deck, the `*SUBSTITUTE` command, which includes the contents of a file in place of the `*SUBSTITUTE` card. Such files can also contain `*SUBSTITUTE` commands, to a maximum depth of 6. This allows, for example, a disc file to be inserted into the job deck after the `*DATA` command, avoiding the need to read data cards again and again for each test run. FORTRAN programs, in particular, benefit from having COMMON data declarations inserted into subroutines by `*SUBSTITUTE`, thus ensuring that the COMMON block is made available throughout the program with complete consistency. Moreover, a single file containing COMMON declarations is easy to keep up to date. Those who have wrestled with COMMON in the absence of such a feature will appreciate its value!

6.2: THE EGDON 3 / COTAN 3 FILE SYSTEM

EGDON 3 [EELM68a] represents the culmination of the EGDON project. It provides a well thought-out file system with comprehensive access controls, and closely related user-management and job-accounting facilities. In EGDON terminology, opening a file is called 'attaching' and closing it is 'detaching'. Access can be granted for reading only, for reading and appending, and for reading and arbitrary overwriting; and this to the file's owner, to any other user (PUBLIC access), and to individual named users so permitted by the file's owner.

The EGDON 3 file system is very card-oriented. Conveniently, four card images exactly fill a single disc sector. A file is held in two contiguous blocks, each block itself consisting of a number of consecutive sectors. The first block, limited to 16 card images, is the file's *head* and contains the file's metadata; the *body* contains the user's data. The maximum size of the file is declared when it is created, and space for a body of that size is reserved. The system keeps note of the sectors actually used.

EXAMPLE 6.1—AN EGDON 3 JOB DECK

```
*JOB WFJOB1/JH68//FINDLAYW/5
*XEQ                                  run the program if no error is detected by JOBORGANISER
*NEWDISCPROGRAM AMTSIM//FINDLAYW      create the RLB program file AMTSIM//FINDLAYW
*CHAIN 1                              this is the only segment in this program
*IDENTIFIER AMTSIM                    title of the RLB module to be created by compiling the following...
*ALGOL                                ... using the Algol compiler
*SUBSTITUTE ALGOLPROG//FINDLAYW       take the source code from the file ALGOLPROG//FINDLAYW
*IDENTIFIER RDWRTT
*USERCODE                             compile the following with the UCA3 compiler
 ENTRYNAMES WCHTT, RCHFT*             give EGDON RLB names to the entries of Usercode routine P7
 ROUTINE P7, V31, M, R2*
etc ...
 =M0M9, EXIT1 (FROM RCHFT), END ( OF P7 ),
*IDENTIFIER GARBAJ
*FORTRAN                              compile the following with the EGTRAN compiler
      SUBROUTINE GC0001
etc ...
      END
*FRONTSHEET                          declare global Usercode data areas
 P TMS, YT100, YU0, END,
*PREDATA                             set values used to size global FORTRAN/ALGOL arrays
      COMMON LSL, LAL
      LSL = 50
      LAL = 1000
*DATA
*SUBSTITUTE ATMDATA//FINDLAYW         include the contents of ATMDATA//FINDLAYW as data
*ENDJOB
```

On a future occasion, the program can be run with different array sizes, thus:

```
*JOB WFJOB2/JH68//FINDLAYW/3
*XEQ
*DISCPROGRAM AMTSIM//FINDLAYW        run the existing RLB program AMTSIM//FINDLAYW
*PREDATA                             new values to resize global arrays, without changing program
      COMMON LSL, LAL
      LSL = 200
      LAL = 1000
*DATA
*SUBSTITUTE DATASET2//FINDLAYW
*ENDJOB
```

■

There are arrangements for moving files to magnetic tape if their frequency of use does not justify a place on the disc. At request, a file can be archived, i.e. copied to an archive tape and removed from the disc, the next time the archiver program is run. Calling for a file to be 'preserved' makes a tape copy, but does not remove the disc original. A restore command brings a tape copy back online.

Regular backups, for disaster recovery, are written to a completely separate set of tapes. Files changed since the last time the disc was 'primed' are copied to tape during a backup run. The disc is 'primed' every so often by overwriting its file storage with a complete copy held on tape. In the process this prime tape is merged with the most recent backup, taken during the most recent system shutdown, to form a new prime tape. Periodical priming limits the size of the cumulative backups. Occasional priming may be necessary to restore the file system, e.g. after the disc has been overwritten during preventative maintenance or repair.

Several file *types* and *classes* are recognized. Most files have type `CRD`, type `DIR` is the type of a directory, and so on. File class `A` is used for alphanumeric card images, `D` for RLB files, and so on. It is possible to have several files with the same 'proper' name, but disambiguated by their type and class. For example a FORTRAN source program might be named `PROG/CRD-A`, its RLB code named `PROG/RLB-D` and its executable binary named `PROG/ABP-D`. Other type and class codes are used for various system purposes.

A file head is somewhat like an 'inode' in modern file systems. It does not contain the disc addresses of a file's data blocks, as these follow the file head directly, but it does contain access control and logging information. In fact the metadata is remarkably comprehensive. It is all kept in a legible alphanumeric format, so the owner of a file can modify its metadata by using the text editor. It includes:

- the filename, type/class code, owner's name, the space occupied and reserved, the creation date, concurrency control flags (the number of current readers, or a flag indicating attachment for appending or writing), a flag indicating whether the file has been changed since the disc was last 'primed', and the date on which the file was last restored from archives (all in one card image)
- the name of the last user to alter the file, the time and date of last writing, of last appending, and of last reading, and the number of times the file has been attached for reading (one card)
- the same statistics, and the reading, appending and writing permissions, for the file's owner specifically (one card)
- the same, for users relying on the PUBLIC access permissions (one card)

- the same, for each of the users individually authorized by the file's owner (one card per user)
- checksums of both the file head and body

Access to a file head is gained from the owner's directory. This too is a file of card images, laid out as follows:

- the user's name, identification code, maximum FNS space allowance, maximum permanent file space, file space currently reserved and occupied, the number of files the user has on the disc, and the number in archives (all in one card image)
- the user's maximum total logged-in time and maximum logged-in session length, total time online, time spent waiting for a terminal, number of LOGINs and date of last LOGIN (one card)
- the user's maximum total elapsed time running commands online, the maximum in a single session, total elapsed time running commands, total CPU time running commands, the number of commands obeyed, the time spent waiting to run commands, and the number of commands obeyed (one card)
- for each of the FOREGROUND, REMOTE and BACKGROUND queues: maximum total elapsed time, maximum elapsed time per job, total elapsed time and total CPU time, time spent waiting for a run, number of jobs run, and date of last job run from the queue (one card per queue)

These administrative cards are followed by entries that catalogue the user's files (two per card). Each entry contains the file name and the disc address of the file's head, or, if the file has been archived, enough information to get it back online.

Access to a user's directory is gained via the system directory, `USERNAME`. Each entry in `USERNAME` consists of a user's name and the disc address of the file head of that user's personal directory.

The primitive operations of the file system are also available to problem programs by means of OUTs, with parameters in the NEST that define the file manipulation to be carried out. These include programmable versions of most file system commands, operating on an attached file.

Worth a Thousand Words is a short film [Culham67] made at Culham to promote graphical output—specifically their GHOST package—in place of reams of printed numbers. Its shows the KDF9 in operation, running EGDON, and giving a very fortunate user a one-hour turnaround for his jobs.

6.3: COTAN 3

COTAN, the Culham Online Task Activation Network [Poole70], is EGDON's multiaccess facility. Like Eldon 2 it has a PDP-8 front-end to handle the interactive terminals [JP74]. COTAN is implemented within the EGDON Director as a set of optional threaded overlays. Up to three of these overlays can be in core at once.

COSEC (Culham Single Experimental Console) was the prototype for COTAN, developed using the console Flexowriter as the interactive terminal [PL68]. A terminal simulator was written to allow pre-determined test cases to be fed in automatically. This was key to bringing a relatively reliable system into early use.

COTAN took the next step by extending access to multiple terminals (Teletype Model 33s) connected via a PDP-8 front end that makes the simple transliteration between the ASCII and EGDON character sets [Culham68a, JP74].

COTAN 3 [Culham68b], issued with EGDON 3 and sharing its nomenclature, is the definitive version, benefitting from the fully elaborated file system and other enhancements incorporated into EGDON 3. There were extensive changes in the user interface between COTAN and COTAN 3. I describe only the latter. In what follows, the notation ^*c* means the character obtained by typing *c* while holding down the CONTROL key.

The terminal user interface is excellent. Facilities are provided to correct typing errors, to control the terminal's status, to redact lengthy output, to break into or stop the execution of programs, to respond constructively to command errors, and to obtain prompts for command parameters. Prompts for the parameters of system-defined commands are issued by the PDP-8 in response to a LF character. A knowing user can type parameters without the delay of prompting (at 10 characters per second) by ending each line with an ESC character instead of LF. Typing ^C requests a prompt for another command. Command names can be abbreviated to their shortest unambiguous prefix.

As each line is terminated the PDP-8 sends it to the KDF9, where it is added to a disc file associated with the terminal. If the user notices an error in transmitted but as-yet unexecuted input, the terminal's input file can be opened and edited, like any other, before being submitted for execution.

A CR character both terminates a line and sends a request for the contents of the file (which may contain several commands and their data) to be executed by the KDF9. Should the execution of a command go wrong, means are provided to recover and continue without losing the rest of the terminal's input file.

The connection between the terminal and the PDP-8 is full duplex. So, if a command generates irrelevant output, it can be completely discarded by typing ^O; or the next *d* lines of output can be skipped by typing the digit *d*; or all lines other than those that start with the '`/`' character can be skipped by typing '`/`'. (Error messages begin with '`/`'.)

COTAN 3 includes the ability to run both Whetstone Algol and previously-compiled binary programs in an interactive mode. Bigger jobs can be carried out by submitting a file of card images containing an EGDON job deck to either of two job queues. Work placed in the REMOTE queue is run as soon as the current EGDON job terminates. Jobs placed in the BACKGROUND queue are scheduled in competition with the rest of EGDON's workload, using a response-ratio scheduler that prevents a long job from being perpetually overtaken by short jobs.

When a REMOTE or BACKGROUND job starts to to run, it is possible to communicate with it by means of the `**JOIN` command. As in Eldon 2, this renders the terminal unresponsive until all of the job's (paper tape punch) output has been sent to it. Or not quite: if the job sets up a special read operation, it can take in data typed at the terminal, and compute further with that. However, an EGDON job is never swapped out, so this is an expensive mode of usage. If the user thinks better of it, they can detach from the job by typing ^X and leave it to run independently; or terminate it, by typing ^R to quit the `**JOIN` command and then issuing an `**ENDJOB` command.

The `*EXMACRO` command in an EGDON job deck provides the complementary facility: it executes a file of COTAN commands as one step in a background job.

COTAN's approach to file-handling commands echoes the KDF9 hardware, in having the concept of a 'file nesting store' to hold operands. For example, to concatenate a file B to the end of a file A:

```
**OPEN
A
**OPEN
B
```

```
**ADD
**SAVE
```

I think this is the most questionable of the design decisions in COTAN. The KDF9 disc drive was slow, and the FNS-based command set entails a great deal of toing and froing, much of which could have been avoided with commands that name their operands explicitly, thus: `cat B >> A`!

Not all commands work on the FNS, however. The `**ATTACH` command establishes a currency for a named file, i.e. it is an 'open' operation. That currency may then be used to access the file, for example by the `**EXALG` command which resumes a suspended Whetstone Algol run, or the `**MERGE` command which updates the attached file using amendments taken from the top of the FNS. It is not clear what the criteria are for working with the FNS, as `**AMEND` does, or with an attached file, as `**MERGE` does. (I suspect economies of effort in implementation.)

COTAN command sequences can be stored in files for repeated invocation, but there is no parameter mechanism, so they have to act with fixed data on files previously placed in the FNS, or interactively specified.

A privileged user can set up a new user by using the `**CREATE` command to make a directory, and the `**ALLOCATE` command to award resources, which are conserved by being withdrawn from the privileged user's allowance. `**DEALLOCATE` does the opposite. Once quiescent, a user can be completely removed from the system by a `**DELETE` command applied to their directory file.

EXAMPLE 6.2—A COTAN 3 TERMINAL SESSION

For clarity, control characters in the following are shown in blue italics, and output to the terminal is shown in red (it printed in black on a Teletype). Lines are assumed to be terminated by *lf* unless shown otherwise. The session creates a file containing a FORTRAN subroutine; amends it; then puts a job in the background queue, output being saved in a file, not sent to the printer.

```
?**LOGIN
PASS? password
USER? FINDLAYW                          username
CODE? JH68cr                            job code; submit login command at once
?**INPUT                                create a file in the FNS and copy data into it
?       SUBROUTINE EMPTY
? C DOES NOTHING
?       END
?**COPY                                 give the file a name and copy it to file storage
FILE? FORTcr
?**AMENDesc                             edit the file in the FNS to insert missing lines
B.SUBResc
I.*FORTRANesc
A.C DOESesc
I.      RETURNesc
STOP^c                                  end AMEND input and start a new command
?**LISTcr                               do the edit and then type out the file
*FORTRAN
      SUBROUTINE EMPTY
C DOES NOTHING
      RETURN
      END
?**SAVE                                 update the file storage copy from the FNS
?**JOB
TYPE? BACKGROUND
INPUT? FORJOB                           this file contains the job deck
OUTPUT? FOROUT                          printer output is written instead to this file
CODE? JH68                              this is the job's account code
TIME? 1cr
13 JOBS IN THE QUEUE — TOTAL TIME 57 MINUTES
?**LOGOUTcr                             too long to wait – take a break!
```

■

COTAN offers a very pleasant and effective environment for interactive computing, with a remarkable range of facilities provided on a modest computer. Reading the EGDON manual sections devoted to system generation, I am struck by the notion that the creators of EGDON and COTAN achieved Iron Age results with Stone Age tools.

ACKNOWLEDGEMENTS

I am grateful to the group of supporters—enthusiastic former engineers, programmers, or satisfied users of KDF9—for their encouragement during this project, and for educating me on aspects of KDF9 software that I was ignorant of. I mention in particular David Holdsworth, David Huxtable, Graham Toal, Brian Wichmann and Andrew Herbert. Others, inexcusably overlooked, know who they are: to them also I extend my thanks.

References and Bibliography

Many of the following documents can be viewed or downloaded. Online sources at time of writing are indicated thus:

† See: `academic.oup.com/comjnl`

§ See: `sw.ccs.bcs.org/CCs/KDF9/Wichmann`

¶ See: `www.computerconservationsociety.org`

* See: `www.findlayw.plus.com/KDF9`

[*AL62] 'Design of an Arithmetic Unit Incorporating a Nesting Store'

R. H. Allmark and J. R. Lucking; *Proceedings of the IFIP Congress 62*, pp. 694-698; 1962.

[†BHJV66] 'The EGDON System for the KDF9'

D. Burns, E. N. Hawkins, D. R. Judd and J. L. Venn; *Computer Journal*, Vol.8 No.4; 1966.
Reprinted in: *Classic Operating Systems*, ed. P.B. Hansen; Springer 978-0387951133; 2001.

[Brinch Hansen70] *'The Nucleus of a Multiprogramming System'*

Per Brinch Hansen; *Communications of the ACM*, Vol.13 No.4; 1970.

[†Brooker58] 'The Autocode Programs developed for the Manchester University Computers'

R. A. Brooker; *Computer Journal*, Vol.1 No.1; 1958.

[†BRS66] 'The Main Features of Atlas Autocode'

R. A. Brooker, J. S. Rohl and S. R. Clark; *Computer Journal*, Vol.8 No.4,; 1966.

[BRSW65] *Atlas Autocode Compiler for KDF9*

P. Bratley, D. Rees, P. Schofield and H. Whitfield; Edinburgh University Computer Unit Report No.4; 1965.

[†CF59] 'The Pegasus Autocode'

B.Clarke and G.E. Felton; *Computer Journal*, Vol.1 No.4,; 1959.

[Culham67] *Worth a Thousand Words*

`www.youtube.com/watch?v=JaHqzQtaVmM`

[*Culham68a] *A Users' Guide to COTAN*, CLM-M75

Computing and Applied Mathematics Group, Culham Laboratory; January 1968.

[*Culham68b] *A Users' Guide to COTAN*

Computing and Applied Mathematics Group, Culham Laboratory; December 1968.

[*CW76] 'A Synthetic Benchmark'

H. J. Curnow and B. A. Wichmann; *Computer Journal*, Vol. 19 No. 1; 1976.

[Davis60] 'The English Electric KDF9 Computer System'

G. M. Davis; *BCS Computer Bulletin*, Vol.4 No.3; 1960.

[†Duncan62] 'Implementation of ALGOL 60 for the English Electric KDF9'

F. G. Duncan; *Computer Journal*, Vol.5 No.2; 1962. See also Erratum: Vol.5 No.3, p. 176; 1962.

[†Duncan63] 'Input and output for ALGOL 60 on KDF 9';

F. G. Duncan; *Computer Journal*, Vol.5 No.4; 1963.

[EEALPHA] *DEUCE Alphacode Manual*

DEUCE Library Service; Data Processing and Control Systems Division, English Electric Company Limited; 19??

[EEFORT] 'FORTRAN'

KDF9 Service Routine Library Manual, Section 13.3; English Electric Leo; 196?.

[EEL64a] *KDF9 Program Index*

Computer Library Service; English Electric Leo; 1964.

[EEL64b] *KDF9 Subroutine Index*

Computer Library Service; English Electric Leo; 1964.

[*EEL65a] ‘Paper Tape Generator’
KDF9 Service Routine Library Manual, Section 10.1; English Electric Leo; 1965.

[EEL66] ‘ALPHACODE’
KDF9 Service Routine Library Manual, Section 13.2; English Electric Leo; 1966.

[*EELM66a] *KDF9 Egdon Programming System: Part 1 Programmers Description*
Publication 103 550566; English Electric–Leo–Marconi Computers Limited; 1966.

[*EELM66b] *KDF9 Egdon Programming System: Part 2 Operating System*
Publication 103 550566; English Electric–Leo–Marconi Computers Limited; 1966.

[*EELM67] ‘PROMPT’
KDF9 Service Routine Library Manual, Section 6.1; English Electric–Leo–Marconi Computers Limited; 1967.

[*EELM68a] *Egdon System Reference Manual*
Publication 103 550566; English Electric–Leo–Marconi Computers Limited; 1968.

[EELM68b] ‘Program KMG01, Report Program Generator’
KDF9 Service Routine Library Manual, Section 12.1; English Electric–Leo–Marconi Computers Limited; 1968.

[*EELM69] *KDF9 ALGOL Programming*
Publication 1002 mm (R) 1000565, English Electric–Leo–Marconi Computers Limited; 1969.

[*Findlay69] *A Turing Machine Simulator*
W. Findlay; 1969.

[*FindlayBM] *The KDF9 and Benchmarking*
W. Findlay; 2020.

[*FindlayEE] *The English Electric KDF9*
W. Findlay; 2020.

[*FindlayHW] *The Hardware of the KDF9*
W. Findlay; 2020.

[*FindlayKB] *The KDF9: a Bibliography*
W. Findlay; 2020.

[*FindlayUG] *Users’ Guide for ee9: an English Electric KDF9 Emulator*
W. Findlay; 2020.

[†Gibbons68] ‘K Autocode’
A. Gibbons; *Computer Journal*, Vol.11 No.4; 1968.

[Haley62] ‘The KDF9 Computer System’
A.C.D. Haley; *Proceedings of the Fall Joint Computer Conference*, AFIPS Conference Proceedings, Vol.22; 1962.

[HH63] ‘A multi-pass translation scheme for ALGOL 60’
E.N. Hawkins and D.H.R. Huxtable; *Annual Review in Automatic Programming*, Vol. 3, Pergamon Press; 1963.

[¶Holdsworth09] ‘KDF9 Time Sharing: Eldon 2 is not EGDON!’
D. Holdsworth; *Computer Resurrection, the Bulletin of the Computer Conservation Society*, no.49; winter 2009/10.

[Huxtable64] ‘On writing an optimizing translator for ALGOL 60’
D.H.R. Huxtable; *Introduction to System Programming*, APIC Studies in D. P., No.4, Academic Press; 1964.

[*HW71] *Improving the Usercode Generated by the Kidsgrove Algol Compiler*
R. Healey and B. A. Wichmann; National Physical Laboratory CCU TM3; 1971.

[*ICL69] *KDF9 Programming Manual*
Publication 1003 mm, 2nd Edition, International Computers Limited; October 1969.

[JP74] ‘A software teletype exchange’
D.A. Jones and N.J. Partington; *Software: practice and Experience*, Vol.4 Issue 2; 1974.

[Knight68] ‘An algorithm for scheduling storage on a non-paged computer’
D.C. Knight; *Computer Journal*, Vol.11 No.1; 1968.

[*Leech63] ‘Coset enumeration on digital computers’
J. Leech; *Proceedings of the Cambridge Philosophical Society*, Vol.59 Part 2; 1963.

[Leigh74] ‘An Algol 60 System for Undergraduates’
D.J. Leigh; *Int. J. Math. Educ. Sci. Technol.*, Vol.5 pp. 553-554; 1974.

[Longbottom]
`www.webarchive.org.uk/wayback/archive/20130303230703/www.roylongbottom.org.uk/whetstone.htm`
Roy Longbottom.

[Martin66] Provisional Specification for the KDF9 Software Interface
J.L. Martin; National Physical LaboratoryKSWP/P5; 20-May-1966.

[†Mulholland69] ‘Software to translate TELCOMP programs into KDF9 ALGOL’
K. A. Mulholland; *Computer Journal*, Vol.12 No.3,; 1969.

[Naur60] *Report on the Algorithmic Language ALGOL 60*
ed. P. Naur; 1960. A copy can be seen in [EELM68a], Chapter 7, ‘EGDON ALGOL’.

[*Ogden65] *How to Write Efficient K/Algol Programs*
J. Ogden; University of Glasgow Computing Laboratory; 1965.

[§OW66] ‘Note on Rapid Instruction Analysis by Table Lookup’]
M. O’Halloran and W.M. Waite; *Computer Journal*, Vol. 9 No. 3; 1966.

[†PL68] ‘The development of on-line computing facilities for the KDF9 part 1: COSEC’
P. C. Poole and T. Lang; *Computer Journal*, Vol.11 No.1; 1968.

[Poole69] ‘Some aspects of the EGDON 3 operating system for the KDF9’
P. C. Poole; *Information Processing 68*, North-Holland Publishing Company; 1969.

[Poole70] ‘Developing a multi-access system online’
P. C. Poole; *Software: Practice and Experience*, Vol.1 No.1; 1970.

[Poole71] *Draft MITEM Preliminary Users’ Manua*l
P.C. Poole; Culham Laboratory; 1971.

[*PS69] *In-Core Card Algol*
J. W. Patterson and A.I. Smith; University of Glasgow Computing Laboratory; 1969.

[*PW70] *The STAGE2 Macroprocessor User Reference Manual*
P. C. Poole and W. M. Waite; Culham Laboratory; 1970.

[Pyle64] ‘Implementation of FORTRAN on Atlas’
I.C. Pyle; *Introduction to System Programming*, APIC Studies in D. P., No.4, Academic Press; 1964.

[*RR64] *ALGOL 60 Implementation*
B. Randell and L.J. Russell; Academic Press; 1964.

[†Ryder64] ‘Note on an ALGOL 60 compiler for Pegasus I’
K. L. Ryder; *Computer Journal*, Vol.6 No.4; 1964.

[Scowen69] *BABEL, a new programming language*
R. S. Scowen; Report CCU 7, National Physical Laboratory; 1969.

[†Stephens74] ‘The IMP language and compiler’
P. D. Stephens; *Computer Journal*, Vol.17 No.3; 1974.

[Thomason70] *A New Assembly Language for KDF9*
J.T. Thomason; University of Leeds Computing Laboratory; 1970.

[*WHMcC71] 'The Eldon 2 operating system for KDF9'
M. Wells, D. Holdsworth and A.P. McCann; *Computer Journal*, Vol. 13 No. 1; 1970.

[§Wichmann69] 'A Modular Operating System'
B. A. Wichmann; *Information Processing 68*, North-Holland Publishing Company; 1969.

[*Wichmann70a] *Some statistics from Algol programs*
B. A. Wichmann; National Physical Laboratory CCU Report No.11; 1970.

[*Wichmann70b] *Estimating the execution speed of an Algol program*
B. A. Wichmann; National Physical Laboratory CCU TM1; 1970.

[*Wichmann71] *The Usercode produced by the Kidsgrove Algol compiler*
B. A. Wichmann; National Physical Laboratory CCU TM2; 1971.

[§Wichmann72] 'Five ALGOL compilers'
B. A. Wichmann; Computer Journal, Vol.15 No.1; 1972.

[§Wichmann73a] *ALGOL 60 Compilation and Assessment*
B. A. Wichmann; Academic Press; 1973.

[Wichmann73b] *Basic statement times for ALGOL 60*
B.A. Wichmann; National Physical Laboratory, Teddington, Middlesex; 1973.

[*Wichmann76] 'Ackermann's function: a study in the efficiency of calling procedures'
B. A. Wichmann; *BIT*, Vol.16, pp. 103-110; 1976.

[*Wichmann77] 'How to call procedures, or second thoughts on Ackermann's Function'
B. A. Wichmann; *Software: Practice and Experience*, Vol.7, pp. 317-329; 1977.

[*Wichmann82] *Latest results from the procedure calling test, Ackermann's function*
B. A. Wichmann; NPL Report DITC 3/82; 1982.