THE ENGLISH ELECTRIC KDF9

by **BILL FINDLAY**

1: KDF9

1.1: BACKGROUND AND OVERVIEW

Announced in 1960, the English Electric KDF9 [Davis60] was one of the most successful products of the early UK computer industry. It still inspires great affection in former users. KDF9 offered about a third of the power of a Ferranti Atlas for about an eighth of the cost. (So much for Grosch's Law!) Much more cost-effective than Atlas, it was also more successful in the marketplace, selling about 30 systems against a handful of the Ferranti machines. This paper presents a synoptic description of KDF9, concentrating on the system-level architecture of the English Electric (EE) hardware and software. For the nearest we have to a KDF9 programming reference manual, see [EEC69]. A relatively digestible summary of the English Electric hardware documentation can be found in [FindlayHW], and a more detailed description of the KDF9's various software systems is given by [FindlaySW].

Even in an era of architectural experimentation, the designers of the KDF9 were bold and innovative. The CPU used separate hardware stacks for expression evaluation and for subroutine linkage. Its three concurrently running units shared the work of fetching, decoding, and executing machine-code instructions, synchronized by a variety of hardware interlocks. The KDF9 was one of the earliest fully hardware-secured multiprogramming systems. Up to four programs could be run at once under the control of its elegantly simple operating system, the **Time Sharing Director**. (In the UK computer parlance of the era, a 'timesharing' system implements multiprogramming, not multi-user interactivity.) Each program was confined to its own core area by hardware relocation, and had its own sets of working registers, which were activated when the program was dispatched, so that context switching was efficient. A program could drive I/O devices directly, but was limited by hardware checks to those the Director had allocated to it.

Later operating systems included DEMOCRAT, at the National Physical Laboratory (NPL); Eldon 2, at the University of Leeds; and EGDON/COTAN, developed primarily at UKAEA Culham. Eldon2 and COTAN were multi-access systems. Remarks about Director in this paper refer to the Time Sharing Director unless the contrary is stated.

The Kidsgrove and Whetstone Algol 60 compilers for KDF9 were among the first of their class. The Kidsgrove compiler stressed optimization; the Whetstone compiler produced an interpretive object code aimed at debugging. Instrumenting the latter made it possible to develop the Whetstone benchmark.

KDF9 should be seen in context. EE's first computer was the DEUCE, a re-engineered version of NPL's Pilot ACE, which was itself adapted from Turing's ACE design. EE then produced several transistorized process-control machines. Technology developed for the latter was used in the KDF9, EE's second-generation successor to the DEUCE. As the KDF9 project came to fruition, EE merged with Leo Computers, forming English Electric Leo. Then English Electric Leo merged with Marconi's computer group, forming English Electric Leo Marconi (EELM). Then EELM merged with Elliott Automation, forming English Electric Computers (EEC). Then EEC merged with ICT, forming ICL. Within five years the distinctive English Electric computer culture had disappeared. Finally, in 1990, ICL was absorbed by Fujitsu. This is how Turing's legacy to the UK computer industry ends, not with a bang but a whimper.

The body of this paper is in the historic present, a nice conceit. As §6 will show, it is not *entirely* a conceit.

1.2: AUTONOMOUS UNITS AND THEIR STORES

A KDF9 computer comprises three primary and many secondary control units, all of them microprogrammed and capable of running in parallel with the others, subject to appropriate interlocks. The primary control units are **Main Control** (MC), **Arithmetic Control** (AC), and **I/O Control** (IOC). All of these units work to the beat of a clock giving two 250ns pulse trains (P1 and P2) separated by 500ns.

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 words, each of 48 bits. The **Q Store** is a bank of 16 index registers, each of 48 bits. The **Subroutine Jump Nesting Store** (**SJNS**) is a stack of 16-bit return addresses, with maximum depth 17 **links**. Note the absence of a conventional 'program counter' register. This is explained later.

AC directly executes simple ALU instructions, and delegates to the multiplier/divider unit and the shifting unit, for more complex arithmetic operations. AC operands are taken from, and results returned to, the **nesting store** (or **NEST**), which is an expression-evaluation stack with maximum depth 19 words. In addition there is a 1-bit **overflow** register, typically set when the capacity of a result is exceeded during calculation, and a 1-bit **test** register, settable to record a variety of 2-state conditions.

IOC supervises up to 16 DMA channels (known as **buffers** in KDF9 terminology) that are capable of simultaneous, independent transfers. Each buffer operates under the control of its own autonomous microprogram.

The Q Store, SJNS and NEST are implemented with the same technology: as core storage with a read cycle time of 1μs and a write cycle time of 1.5μs. The main store is comparatively slow, having a read or write cycle time of 6μs, and the architecture of the KDF9 goes to some lengths to mitigate the effect this has on performance.

A hardware **timesharing** option replicates the NEST, SJNS and Q Store, so that four register sets are provided, thereby obviating the need to save and restore 48 registers when context-switching between a maximum of four concurrently running processes or **problem programs**.

KDF9 runs under the supervision of a Director program, a set of privileged routines normally resident at the bottom of store. Its main function is to respond to interrupts, and thereby provide essential services to problem programs, including management of the timesharing facility. Communication with the operators is by means of messages typed on the monitor console Flexowriter. It has a button that is pressed to gain the attention of Director by causing a typewriter interrupt. Merely using the keyboard does not cause an interrupt.

© William Findlay, Version of 2021-11-13
This document is licensed under a Creative Commons Attribution 3.0 License: http://creativecommons.org/licenses/by-nc-sa/3.0/

1.3: Data Formats

A KDF9 data word consists of 48 bits numbered from D0 (the most significant bit) to D47 (the least significant). Similar big endian numbering is applied to part-word bit fields. Several data formats are supported:

- as a 48-bit, 2's complement binary integer, fraction, or fixed-point number
- as a 48-bit floating point number, consisting of a sign in D0, an excess-128 exponent in D1-D8, and a mantissa in D9 to D47; where the mantissa, together with D0, forms a 2's complement fraction
- as half of a 96-bit, 2's complement binary integer, fraction, or fixed-point number, where D48 (D0 of the second word) is set to zero, so that the effective capacity is 95 bits
- as half of a 96-bit floating point number, consisting of sign in D0, excess-128 exponent in D1-D8, a first part of the mantissa in D9 to D47, D48 set to zero, and a second part of the mantissa in D57-D95; where D49-D56 are set equal to (exponent – 39) unless that would be negative, in which case the whole of D48-D95 is set to 0; and where the mantissa, together with D0, forms a 2's complement fraction
- as two 24-bit **halfwords**
- as three 16-bit 2's complement integers that constitute the **Counter** (**C-part**, D0-D15), the **Increment** (**I-part**, D16-D31) and the **Modifier** (**M-part**, D32-D47) of an indexing word in **Q store format**
- as a control word for I/O instructions, in which the C-part is a device number; the I-part a main store start-address, or unused; and the M-part a main store end-address, or an operation count, or unused
- as 8 characters of 6 bits each

Single-precision floating-point results are normally rounded: 1 is added to D47 of the result if the first truncated bit is not 0; but double-precision results are always unrounded: the truncated bits are ignored. Floating-point results are always **standardised** (normalised, so that D0 ≠ D9); the only floating point operation that gives a well defined result from a non-standardised operand is the STAND instruction, which effects normalisation.

There are no double-word fetch or store instructions: double words need a pair of fetch or store instructions to copy them into or out of the NEST. The less significant word of a pair is held in the deeper of its two NEST cells.

There are instructions for fetching and storing 24-bit halfwords. Fetching a halfword expands it to a full word in the NEST by filling in the 24 least significant bits with zeros. Storing a halfword from a word in the NEST extracts its most significant 24 bits. A halfword can hold a 24-bit 2's complement binary integer, fraction, or fixed-point number, or a floating-point number (the latter consisting of the most significant 24 bits of the 48-bit floating-point format).

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 the KDF9 character codes, 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.

1.4: Instruction Formats

A KDF9 instruction consists of one, two or three **syllables** of 8 bits, the first two bits of each instruction giving its type: 00_2 for one-syllable instructions, 01_2 for two-syllable instructions, 10_2 for three-syllable jump instructions and 11_2 for ('directly addressed') fetch and store instructions of three syllables. 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. They are carried out by **Arithmetic Control**.

Instructions of two or three syllables are primarily the responsibility of **Main Control**.

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*q*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.

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 **short loop jump** instruction, which does not contain a label address, but instead jumps to syllable 0 of the word preceding the word containing the J*r*C*q*NZS instruction itself.

2: The CPU

2.1: Main Control, Arithmetic Control and I/O Control

The control logic of KDF9 comprises two blocks: the I/O block and the Computing (CPU) block. The I/O block is provided with instructions by the Computing block, but otherwise works autonomously, taking priority for access to main store. The Computing block itself is composed of Main Control (MC) and Arithmetic Control (AC). AC carries out 1-syllable instructions and does part of the work for some 2- and 3-syllable instructions. AC in turn operates the multiplier/divider unit, the arithmetic unit, and the shifting unit. MC does the rest. It works collaboratively with AC, using interlocks to manage their concurrency. MC leads AC, skipping over 1-syllable instructions without time penalty; AC cannot overtake MC. They interact most strongly on shift instructions, where MC calculates the shift length, but AC performs the shift; and on fetch/store instructions, where MC handles data transfer to/from the store and AC handles data transfer to/from the NEST. This parallelism does much to compensate for the relatively slow core store.

For fetches and stores, MC sets an address register with the source or destination address. For a fetch operation it then initiates a core cycle—when available, the word is transferred from the store buffer to whichever of two fetch buffers is empty. If both contain data that has not yet been accepted by AC, the transfer is interlocked until AC catches up. The fetch buffers mean that MC can implement up to two fetch instructions, their core cycles being overlapped with computation by AC. This is particularly valuable in the very frequently executed inner loops that evaluate scalar

© William Findlay, Version of 2021-11-13
This document is licensed under a Creative Commons Attribution 3.0 License: http://creativecommons.org/licenses/by-nc-sa/3.0/

products. (KDF9 was very much a 'scientific' computer.) For a store operation the destination address is saved until AC delivers the value to the store buffer; MC is then able to write that word into core. If a fetch is from the same address as the previous store, the content of the store buffer is copied to a free fetch buffer, so eliminating a redundant core cycle.

Instruction fetching is also the responsibility of MC. AC and MC each has its own current-instruction-address registers. These registers jointly perform the rôle of the 'program counter' in a more conventional machine. There are two instruction-word buffers. MC fetches an instruction word to the instruction-word buffer that is not currently being inspected by AC. Thus instruction-fetching can also overlap with computation by AC.

The art of 'optimum programming' on KDF9 is concerned with the careful relative placement of fetch, store, jump and arithmetic operations, with a view to maximizing the parallelism between store cycles and calculation. In this respect KDF9 is very like modern CPUs.

2.2: Addressing and the Q Store

Problem programs address their instructions and data starting from a (virtual) effective address of 0. When store is accessed, the hardware offsets this address by the program's starting position in main store. At the same time it checks that the virtual address does not exceed the number of locations allocated to the program by Director. This confines each program to its own area (unless Director allocates overlapping areas to two or more programs, which can be useful for some purposes). In Director state no offsetting is done, so that Director starts at physical location 0.

The address part of a jump instruction consists of a 13-bit word number and the 3-bit number of a syllable within that word. Consequently, the instructions of a program are confined to its first 8192 words. EXIT (return from subroutine), EXITD (return from Director to a problem program) and OUT (invoke Director from a problem program) have the same basic format as a jump. EXIT's address part is treated as an offset, to be added to the link in the SJNS. OUT and EXITD are jump-type instructions with ignored bits where the address would normally be.

Effective addresses for data fetches and stores are generated in either of two ways: by 3-syllable instructions containing a 15-bit constant address part and a single Q store reference, Q*q*; or by 2-syllable instructions containing two Q store references, Q*k* and Q*q*. The effective address in the first case is the sum of the constant and the contents of M*q*; in the second case it is the sum of M*k* and M*q*. Q0 always contains 0, providing an efficient way of getting a zero base.

Data locations in EE Usercode have rigidly stereotyped identifiers. Local variables of subroutines have names of the form V*m*; global variables have names of the form W*m*, Y*m*, YA*m*, …, YZ*m*. If indexed, the identifier is followed by 'M' and the Q store number; the 2-syllable instructions take the operand form 'M*k*M*q*'. Usercode instructions such as V9; YA7M2; M1M2; and so on, represent data fetches that push values onto the NEST; =V9; =YA7M2; =M1M2; and so on, represent data stores that pop values from the NEST.

Flags suffixed to the instruction optionally specify index updating, halfword addressing, and 'next word' addressing.

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 a predetermined (but variable) number of locations, at addresses given by an initial address and a predetermined (but 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 '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 D24-D47 of the addressed word.

The 'N' suffix, on a 2-syllable fetch or store order, 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.

The 3-syllable fetch and store orders take 6μs, the 2-syllable fetch and store orders take 7μs, and an additional 1μs is needed for Q register updating in both cases.

2.3: Expression Evaluation and the Nest

An adder associated with the Q store carries out the arithmetic involved in determining effective addresses. Overflow on address and Q store arithmetic is ignored. All other calculations, both fixed-point and floating-point, take operand(s) from the NEST and return result(s) to the NEST. It is common for parameters of a subroutine to be pushed onto the NEST, especially when the routine implements an arithmetic function.

If an instruction pops an operand from an empty NEST, or pushes an operand onto a full NEST, a Nest Over/Underflow Violation (NOUV) interrupt is caused and Director takes the appropriate action. Keeping track of NEST depth is one of a KDF9 programmer's main chores, and NOUV interrupts are among the most common results of programming error. 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 (J*r*EN) or not (J*r*NEN).

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, ...

Simple instructions—including integer +, −, logical AND, OR, REV, DUP, ERASE, *etc*—take from 1μs to 4μs. Floating-point addition (+F) takes from 7μs, multiplication (×F, ×D, *etc*) takes from 14μs, and division (÷F, ÷I, *etc*) takes from 36μs; the variation being due to data-dependent pre-shifting and post-normalization.

The 1-syllable ZERO and the 3-syllable SET allow for the sourcing of small constants. The most efficient way to push -1 is: ZERO; NOT; and the most efficient way to get +1 is: ZERO; NOT; NEG;—the same number of syllables as SET 1; but slightly faster. Subtracting 1 is best done by: NEG; NOT; and adding 1 by: NOT; NEG. It is nice that these

© William Findlay, Version of 2021-11-13
This document is licensed under a Creative Commons Attribution 3.0 License: http://creativecommons.org/licenses/by-nc-sa/3.0/

'hacks' set overflow in exactly the same cases as SET 1; +; or SET 1; –;. SIGN compares N1 with N2 and yields –1, 0, or +1, according as the sign of (N2–N1), while avoiding the possibility of an overflow. SIGNF is similar, but for floating point operands. MAX sorts, swapping N1 and N2 if necessary, leaving N1≥N2 and setting overflow unless N2 was already strictly less than N1. MAXF does the same, but for floating point operands. BITS replaces N1 with a count of the number of 1-bits it contained. (It is interesting to speculate that BITS may relate to Turing's cryptological work.) TOB yields a binary integer from a mixed-radix integer and a radix pattern; and FRB does the converse.

The top three cells of the NEST (symbolically, N1 at the top, then N2 and N3) are flip-flop registers. These registers are managed by the hardware, and Director, transparently to problem programs. A maximum of 16 NEST cells are available to a program; NOUV is caused *after* pushing a 17th value, and *after* popping an empty NEST. Since an interrupt is not effected until it is convenient for MC, AC may have performed several further NEST operations, leaving the contents of the NEST unpredictable. So, no recovery from NOUV is possible for a problem program.

EXAMPLE—TO COMPUTE THE FLOATING-POINT VALUES: $(-b \pm \sqrt{(b^2 - 4ac)}) \div 2a$

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: $(-b - \sqrt{\Delta}) \div 2a$
`Q2; +F; Q1; ÷F;`	N1: $(-b + \sqrt{\Delta}) \div 2a$; N2: $(-b - \sqrt{\Delta}) \div 2a$

2.4: CONTROL FLOW AND THE SJNS

The J*r* instruction is an unconditional jump to the instruction at label *r*. The instructions: J*r*=Z, J*r*≠Z, J*r*>Z, J*r*≥Z, J*r*<Z, J*r*≤Z 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; inconsistently, but conveniently, 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.

The JS*r* instruction branches unconditionally to the instruction at label *r*, and pushes **its own** address onto the SJNS as a return **link**. This creates issues of SJNS management similar to those that arise with the NEST, and the instructions J*r*EJ and J*r*NEJ are provided analogously to J*r*EN and J*r*NEN. The only other instructions that access the SJNS are:

- EXIT *a*: complementary to JS*r*—pops the link from the SJNS, adds the constant *a* (which is an integral multiple of 3 syllables), and branches to the resultant address
- OUT: pushes its own address onto the SJNS and causes an OUT interrupt, thereby switching control to Director in non-interruptible, privileged state
- EXITD: complementary to OUT, but also used when context-switching—pops the link from the SJNS and branches to the resultant address in program state, with interrupts enabled on completion
- LINK: pops a link from the SJNS and pushes it onto the NEST
- =LINK: pops a link 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 the EXIT instruction notated EXIT AR*r* jumps to the selected word in the jump table starting at label *r*.

All interrupts, like OUT, use the SJNS for their return address. LOV, like OUT, pushes its own address; other interrupts push the address of the next instruction to be executed.

EXAMPLE—TO COMPUTE THE DOUBLE-PRECISION SCALAR PRODUCT $\sum x_i y_i$

On entry to the loop, Q1 = *n*/1/0, where *n* is the length of the vectors; the vector $\boldsymbol{x}$ is in the variables YX1…YX*n*, $\boldsymbol{y}$ is in the variables YY1…YY*n*; 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;
```

The first pass through the loop takes 36μs: AC must wait for MC to finish each of the first two fetches, and MC must set up the short-loop mode when it first encounters the J1C1NZS instruction. Subsequent iterations of the loop take only 25μs: the time in MC is completely overlapped by the time AC takes in the ×+F instruction.

© William Findlay, Version of 2021-11-13
This document is licensed under a Creative Commons Attribution 3.0 License: http://creativecommons.org/licenses/by-nc-sa/3.0/

3: Input/Output

3.1: Peripherals

Up to 16 I/O buffers can be fitted. Each has its own independent control unit, which is microprogrammed to support its connected device. The Flexowriter is always on buffer 0, and a paper-tape reader with hardware bootstrap facility is always on buffer 1. Peripheral devices fall into two classes: **slow**, or **character** devices; and **fast**, or **word** devices. Buffers for slow devices do a core cycle for each character transferred; for fast devices they do a cycle for each word.

The slow devices include the Flexowriter (10ch/s), paper tape punches (110ch/s), paper tape readers (1000ch/s), card readers (10 card/s), card punches (5 card/s) and line printers (about 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. It makes poor use of the 8 bits on the tape: 6 bits encode the character, 1 bit establishes even parity, and 1 bit is used so that the all-zero (blank) character is not represented by unpunched tape, which is taken as contentless leader or trailer.

Graph plotters are an option. They are attached to a paper tape punch buffer, which is manually switchable between the punch and the plotter. An I/O instruction is provided to test whether the punch or the plotter is the active device.

The fast devices are magnetic tape decks, fixed-disc drives, and drum stores.

The magnetic tape physical format uses 16 tracks across the tape: 6 data tracks, a parity track and a clock track are duplicated for each character. A character is deemed validly read if either copy can be read without error. Tapes run at 100inch/s, with 400ch/inch, for a transfer rate of 40kch/s. Inter-block gaps are about 1/3 of an inch. A faster, but seldom seen, magnetic tape deck runs at double speed (80Kch/s), recording on a steel band instead of plastic tape.

The fixed-disc drive has 17 disc platters, each with a capacity of 6144 blocks of 320 characters, for 98304 blocks (31,457,280 characters) per drive. Up to four drives can be attached to the disc buffer, special provision being made for this in its microprogram. Fixed heads are mounted in the outer zone of the first platter, providing access without a seek to one of the fastest tracks. Each other platter has its own independently seeking arm, carrying eight read/write heads. The discs spin at 1000 rev/min; a seek takes from 156ms to 367.5ms, with an average of 231ms.

The drum consists of 320 addressable sectors, each of 128 words, for a total of only 40960 words of storage, accessed at the extraordinary transfer rate (for the time) of 500kch/s. It is of use mainly for storing frequently executed programs, such as the program source editors and compilers. Few were deployed.

3.2: I/O-driven Multiprogramming

When an I/O transfer is started, the designated core store area is **locked out** by the buffer, so that any attempt to access it concurrently, whether to fetch or store data, to fetch instructions, or use it for another I/O transfer, causes a Lock-Out Violation (LOV) interrupt. This suspends the offending program until the responsible transfer finishes and the buffer removes the lockout. Lockouts are established with a granularity of 32 words, so for greatest I/O overlap it is important to ensure that each main-store area used for I/O is aligned on a 32-word boundary.

Attempting an I/O operation on an already-busy buffer also causes a LOV interrupt. Applying an INTQ*q* instruction to a busy buffer yields the CPU to a lower-priority program by causing a PR interrupt; applying INTQ*q* to an idle buffer has no effect. PARQ*q* and BUSYQ*q* both transfer state from a buffer to the CPU's test register, where it can be used to condition a jump instruction. BUSYQ*q* checks whether a buffer is presently engaged in a transfer. PARQ*q* checks and clears a parity error flag on a buffer's last completed transfer; any other command to a buffer with an uncleared parity error causes a LIV interrupt.

Program blocking by I/O operations is automatically managed by I/O Control, using a 12-bit Program Hold-Up (PHU) register to keep track of the state of each program priority level. A buffer notes the CPU's privilege state at the start of a transfer. On completion, if Director started the transfer, the buffer requests an EDT (End of Director Transfer) interrupt; but if the transfer belongs to a problem program, its PHU is cleared. If the cleared PHU belongs to a program of higher priority than the one currently running, a PR interrupt is requested. Since a suspended program may be waiting on a shared device (typically, the Flexowriter) made busy by a program of lower priority, every other PHU is then examined to see if it refers to the same buffer. If so, an EDT interrupt is requested instead of PR. This enables Director to resolve the priority inversion over the shared device.

3.3: I/O Instructions

There are two significant aspects to the software control of I/O devices on KDF9: their allocation to problem programs by Director, and their control by problem programs once allocated.

There is an output spooling facility implemented by Director, often called 'OUT 8' after the relevant system call. Programs send data to one of up to 64 output streams, which are incrementally written to a magnetic tape by Director. When full, a tape is read back and its streams are disentangled for output to paper tape punches and line printers. This is the *modus operandi* that maximizes the scope for multiprogramming, as it insulates the workflow from the availability of physical output devices. No input spooling is provided.

Alternatively, problem programs can directly drive I/O devices on buffers allocated to them by Director. Any attempt to access an unallocated buffer causes a 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 somewhat different characters sets; so programs have to be device-aware for that reason, if no other.

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 (i.e. obeying the OUT instruction with 5 in N1). If such a device is available, Director allocates 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 de-allocate it, to maximise resource sharing. This is achieved 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.

The instructions that control devices take their parameters from a designated Q store, Q*q*. C*q* contains the buffer number of the allocated device, as returned by OUT 5. Depending on the specific instruction, I*q* and M*q* might contain starting and ending *virtual* addresses for a data transfer operation, the transfer count being determined by their difference; or M*q* might contain a repetition count for a control operation; or I*q* and M*q* might both be ignored. If the least significant bit of an I/O instruction is 1, it sets the device off-line before initiating the operation. The device is set

© William Findlay, Version of 2021-11-13
This document is licensed under a Creative Commons Attribution 3.0 License: http://creativecommons.org/licenses/by-nc-sa/3.0/

on-line by the operator pressing a button on its control panel. In this way a program can set up a transfer on a tape reader, for example, but wait until the operator has loaded a tape before actually reading.

After checking its validity, the CPU delegates an I/O instruction to its specified buffer, via I/O Control; if it is a data transfer instruction, the CPU also provides physical addresses converted from the virtual addresses given in I*q* and M*q*.

There is some attempt at making the effect of a given instruction generic, so that PIAQ*q*, say, acts in a similar way on all input-capable devices, but this is not carried through with complete consistency. Some simpler devices respond to different orders in the same way: PIAQ*q* and PIEQ*q*, for example, have the same effect on a paper tape reader.

A completely generic feature of the I/O architecture is the provision of **variable length** transfer instructions. These take a (maximum) transfer count, as usual, but terminate early upon transferring an **end message** character (written '→', code 75_8). When a variable-length transfer terminates on end message, any remaining character positions of the last word transferred are set to zero.

Another feature, generic to the slow devices only, is the availability of the misnamed **character** transfer option. These instructions read one character into, or write one character out of, the least significant bits of each word in the transferred area. This is the only I/O mode that permits the transfer of an arbitrary number of characters without the side effect of termination on End Message. When applied to paper tape, character mode further allows for all 8 bits of a frame to be read or written, so that foreign codes can be handled (at the cost of forgoing hardware parity checks).

A peculiarity of the Flexowriter is that, when a semicolon is written, an output operation changes itself on-the-fly to an input operation; subsequent typed input is transferred to the originally designated output buffer, in character positions following the semicolon. This can be used to program an atomic, prompt-and-response, form of interaction.

The PM*x*Q*q* group of instructions provides media and manipulative operations, such as rewinding a magnetic tape. A variety of test instructions enable device state to be transferred to the test register (e.g., whether a magnetic tape is positioned at the Beginning Of Tape window); these are also encoded as instructions of the PM*x*Q*q* group.

The Usercode programmer is given multiple, device-specific mnemonics for each hardware I/O instruction. For example, the output instruction POAQ*q* can also be written as MWQ*q* when the intended output device is a magnetic tape drive, and as TWQ*q* when it is the console typewriter. The use of a specific mnemonic has no significance at run-time; effectively it acts like a comment appended to a generic mnemonic in the source program.

Only a fraction of many possible I/O opcodes have defined effects. Moreover, some are defined only for one class of device (input or output). Attempting an undefined I/O operation causes a LIV interrupt.

4: Director State

4.1: Interrupts

KDF9 interrupts may be either:

- **voluntary** (OUT or INTQ*q* obeyed by the problem program)
- **inadvertent** (illegal instruction, NEST over-/under-flow, etc)
- **housekeeping** (monitor typewriter, clock and I/O device interrupts)

The special register RFIR (Reason For Interrupt Register) records the currently requested interrupt(s). RFIR 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 an instruction address in the top cell of the SJNS. If a store-access instruction with Q store updating experiences a Lock-Out Violation, the Q register update is suppressed. When the lockout clears, and the interrupted program is resumed, the instruction can be retried without doubly updating the Q register.

On entry to Director, BA (Base Address) is set to 0 and NIFF (Non-Interrupt Flip-Flop) is set to inhibit further interrupts, but not their recording. However, the RESET interrupt is never inhibited and the NOUV interrupt is completely suppressed. When fetched to the NEST, by the privileged K4 instruction, RFIR is automatically cleared. K4 delivers the following data:

- D0-D15: CLOCK COUNT; the integer in D0-D15 is incremented every 32μs; a CLOCK interrupt occurs when overflow from D1 sets D0, but if D0 is already set a RESET interrupt occurs instead
- D22: the PR interrupt is caused by the end of a peripheral transfer started by a program of higher priority than the program currently running; the PR interrupt is also caused by an INTQ*q* instruction applied to a busy device
- D23: the FLEX (Flexowriter) interrupt is caused by the interrupt key on the console typewriter
- D24: the LIV (Lock-In Violation) interrupt is caused by an illegal or privileged instruction, by the use of an unallocated peripheral, by a main store address outside the current reservation, or by address wraparound, or by a negative effective address
- D25: the NOUV (Nest Over- or Under-flow Violation) interrupt occurs when an attempt is made to overfill an already-full NEST or SJNS, or to pop an already-empty NEST or SJNS
- D26: the EDT (End of Director Transfer) interrupt occurs at the end of a peripheral transfer initiated by Director, or if a priority inversion has been detected among programs currently locked-out and waiting for a shared device
- D27: the OUT (system-call) interrupt is caused by the OUT system-call instruction
- D28: the LOV (Lock-Out Violation) interrupt is caused when a program attempts to access any in a locked-out group of 32 words, or when it attempts to command a busy peripheral device
- D29: the RESET interrupt is caused by jumping to an invalid syllable address (6 or 7), or by a watchdog 'double clock' (implying that a previous CLOCK interrupt has not been handled by Director in the intervening time)

Since NOUV is completely suppressed in Director state, all 19 NEST cells—16 in the core stack, plus the three flip-flop registers, N1...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. This means that context switching in response to PR interrupts is rather efficient, the elapsed time between interruption and re-entering a program being of the order of 320μs, or about 60 instructions. The 'long path' through Director, where more complex work is done, must save many more of the interrupted program's registers; but that happens relatively infrequently.

The implementation of the SJNS is similar to that of the NEST. The Jump Buffer, a single flip-flop register analogous to the NEST's three 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'. NOUV is caused *after* a

© William Findlay, Version of 2021-11-13
This document is licensed under a Creative Commons Attribution 3.0 License: http://creativecommons.org/licenses/by-nc-sa/3.0/

problem program pushes a 17th link, and *after* it pops an empty SJNS. The excess link left in the Jump Buffer by a 17th SJNS push is merely overwritten by the interrupt's return address; neither is significant, because no recovery from the NOUV condition is possible.

4.2: Director-Only Registers and Instructions

The other special registers concerned in multiprogramming are:

- BA (Base Address): contains the **group address** (top 10 bits) of the 32-word block at which the program area starts, and is used to offset virtual addresses so as to effect dynamic relocation
- NOL (Number Of Locations): contains the BA-relative group address of the highest group the program can access
- CPL (Current Priority Level): the 2-bit CPU priority of the current program (3 is lowest, 0 is highest)
- CPDAR (Current Peripheral Device Allocation Register): a 16-bit register, each bit corresponding to a buffer. If a bit is set, the current program has access to the corresponding buffer
- PHU (Program Hold-Up): the 4 PHU stores.

On entry to Director BA is set to 0, so Director runs at the bottom of core. Strangely, the comparison of effective addresses with NOL is not suppressed in Director state; Director must set NOL to the maximum for the machine, if it wants to guard against causing a LIV interrupt when accessing main store above its own upper limit.

I/O data transfer instructions specify virtual addresses in Iq and Mq. The CPU checks that I$q \le$ Mq, checks both against NOL, and adds to both the value in BA, before passing them on to I/O control. In this manner the operand addresses are validated and converted from virtual to physical form. A failed check causes a LIV interrupt.

With the timesharing option installed, there are four instances of the NEST, Q stores and SJNS. On context switching between programs, Director can switch quickly between instances by changing the active set number, but must still save and restore the NEST- and SJNS-stack depths, BA, NOL, CPL, CPDAR, the overflow register and the test register.

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

- EXITD: clear NIFF and jump to the address in the top cell of the SJNS; interrupts are inhibited throughout the execution of EXITD, to allow the machine to adopt a stable state
- CLOQq: clear lockouts for the transfer specified by Qq and clear PHUn, where n is the value of CPL
- CTQq: terminate device activity and clear lockouts for the transfer specified by Qq.
- =K0: **if** N1 $\neq 0$ **then** switch buzzer on **else** switch buzzer off **end**
- =K1: copy bits N1:D24-D33 to NOL, bits N1:D34-D35 to CPL and bits N1:D38-D47 to BA
- =K2: copy bits N1:D32-D47 to CPDAR, where N1:D32 corresponds to buffer 15 and N1:D47 to buffer 0
- =K3: switch to a new Q store/NEST/SJNS set, with new NEST depths:
  N1:D0-D1 are the new register set number, N1:D2-D6 are the new NEST depth and N1:D7-D11 are the new SJNS depth; the register set number is independent of the program priority level. The =K3 instruction must be followed by at least 6 DUMMY instructions, since it takes 6μs to take effect and during this period the machine is in an indeterminate state.
- K4: push CLOCK/RFIR onto NEST.
- K5: push PHUi:D6-D11 onto NEST, for i in 0..3, with PHU0:D6-D11 in N1:D0-D5, PHU1:D6-D11 in N1:D6-D11, PHU2:D6-D11 in N1:D12-D17, and PHU3:D6-D11 in N1:D18-D23.
- K7: push the current register set number and NEST depths, as represented for the =K3 instruction.

N.B. All =Kk instructions copy N1, and do not pop it from the NEST.

5: Software

5.1: Operating Systems

The KDF9 was designed with an operating system—the Time Sharing Director—very much in mind. Controlled access to I/O devices, I/O-driven dispatching, and program relocation hardware, combine to make multiprogramming on KDF9 both efficient and secure. Machines could be supplied without the timesharing hardware. To support them EE provided the Non-Time Sharing Director, which has many of the features of the more comprehensive system, with the exception of multiprogramming. But it is the Time Sharing Director that throws most light on the KDF9's innovative architecture.

Pre-emptive, priority-based multiprogramming is directly supported by the hardware for up to four programs, with measures to deal with priority inversion. Four active programs are often more than can be accommodated in the 32KW core store, so the limitation of the timesharing hardware to four levels is seldom a bottleneck. It could be overcome by having Director share the lowest level among several programs, but that has never been deemed to be necessary.

Job scheduling is shared between Director and the computer operators. Programs are divided into two categories, **A** and **B**. B-programs are loaded in response to commands from the operators; 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 used. The operators also define the maximum number of A-programs that may be run concurrently, and assign them to timesharing 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.

Store is allocated to programs using the classic 'two-stack' ploy. B-programs are loaded above Director and into higher addresses; A-programs are loaded at the top of the store, and into lower addresses; free space is consolidated into the gap between the A and B store-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: once an I/O transfer has been initiated the buffer works in terms of physical addresses, so Director must wait for all ongoing transfers into a program's area to terminate before being able to move it.

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 the OUT 8 spooling system, program loading, operator communication, and so on. To allow these operations to proceed concurrently, an internal multi-tasking system is implemented. Ten Director threads are supported, two for each program and two for Director itself. Thread scheduling is co-operative, with threads

© William Findlay, Version of 2021-11-13
This document is licensed under a Creative Commons Attribution 3.0 License: http://creativecommons.org/licenses/by-nc-sa/3.0/

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. A comparison of elapsed time and notional elapsed time for a program indicates how well it is being scheduled. Director can exchange the priority levels of a pair of programs, if it finds 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 buffers to update its dispatching status, Director must wait for the end of all the pair's I/O transfers before swapping their priorities.

The EE Time Sharing Director is most effective with a workload consisting of a relatively small number of relatively long-running jobs. Given an I/O-limited job to play against a CPU-limited job, 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 job mixes, and to offer multi-access working, that Eldon 2 [WHMcC71], 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. Eldon 2 proper, an overlaid and multi-tasking adjunct to Director, runs in program mode at level 0. It communicates with the PDP-8 to process commands from interactive users; it also prints down output that was spooled to tape by OUT 8. Level 1 is devoted to running short, on-demand, foreground jobs that are locked in core until termination. Longer, queued, background jobs are run at level 2. The Job Organiser program runs, to initiate a new foreground or background job, when core is freed by a program terminating or contracting. The Job Organiser swaps out a background job, if necessary, to let a foreground job start. 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. In later versions of Eldon 2 a 'cafeteria' service is run instead; it lets students compile and run small Algol 60 programs from a 'live' tape reader and small FORTRAN programs from a 'live' card reader.

Thus there is a nice hierarchy of workers: the Director 'short path' delegates to Director threads; Director delegates to Eldon 2; and Eldon 2 tasks delegate to Job Organiser. Beyond Eldon 2, humans are the scheduling authority of last resort. Achieving a CPU utilization of up to 85%, Eldon 2 runs non-interactive jobs concurrently with commands from up to 28 terminals—all of this on a computer having 192KB of core store, and two 24MB file discs (with an average seek time of nearly a quarter of a second)!

The situation at UKAEA Culham Laboratories was different. Their workload was of a type that might well have suited the EE Time Sharing Director, but their machine was not fitted with the timesharing hardware; so they took the opposite tack and developed EGDON [BHJV66], a 'sausage machine' batch processing system of the kind familiar on other scientific computers of the day. COTAN [Poole70] is EGDON's add-on multi-access facility, also using a PDP-8 front-end on a magnetic tape buffer, but implemented entirely within the EGDON Director as a set of threaded overlays. COTAN's command language echoes the KDF9 hardware, in having the concept of a 'file nesting store' to hold the operands of file-handling commands such as editors.

DEMOCRAT [Wichmann69] is a pioneering, microkernel operating system developed at NPL, remarkable for being perhaps the first of its kind. Largely because its API is incompatible with that of the EE Time Sharing Director, it was never used for serious production work.

5.2: 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 badly lacking in EE's original version. As for high-level languages, these focus on 'scientific' computing. Algol 60, Babel, FORTRAN, IMP, K Autocode, and KDF9 Autocode are all available; COBOL is a notable absentee, but there is a Report Program Generator (RPG) from EE.

KDF9 is one of the first computers to support 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.

KAlgol [HH63, Huxtable64] aims at efficient machine code and includes an optional global optimization phase. The compiler has an unfortunate reputation for being slow and the optimizer is accused of generating unreliable 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. KAlgol takes a conservative approach to management of the NEST, which offsets some of its advantages [Wichmann73, WJ76]. Leeds avoided the clumsy process whereby the run-time support library is re-translated for each KAlgol program: compilation speed was improved by simply appending the object program to a pre-assembled module containing the most-commonly used routines. A fuller and fairer account of Kidsgrove Algol is given in [FindlaySW].

WAlgol is a 'checkout' compiler aimed at fast compilation and comprehensively checked execution. It generates byte code for an abstract machine that bears a remarkable resemblance to the machine code of the (later) Burroughs B6500 architecture. Native to the paper tape oriented EE Timesharing system, WAlgol was adapted to card input by Patterson *et al*. in the guise of Incore Algol, 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.

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, the NPL ACE, the EE KDF6, the Soviet Minsk range, the EE System 4/50, the IBM System 360/25, the Phillips PRS8000, and the Indian ECIL TDC-316.

© William Findlay, Version of 2021-11-13
This document is licensed under a Creative Commons Attribution 3.0 License: http://creativecommons.org/licenses/by-nc-sa/3.0/

Brian Wichmann, at NPL, modified the Whetstone interpreter to gather byte code execution-frequency statistics for a large number of programs. This data was used to synthesize the famous **Whetstone Benchmark** [CW76], a program that reproduces these statistics in a single run, permitting its use as a simple but effective tool for estimating computer performance in scientific applications. See also [FindlayBM] for more on the crucial rôle KDF9 played in the development of computer peformance benchmarking.

The EGDON operating system eventually gained a third Algol 60 compiler, written by the implementers of KAlgol with the benefit of hindsight. It omits global optimization, but includes a very good peephole optimizer. It combines respectable compilation speed and reasonably fast object code with decent run-time checking.

EGDON also has a FORTRAN compiler, known as EGTRAN. EGDON compilers generate relocatable object code which is combined with library routines to form an executable program, by a utility called the Composer. Programs may include routines with source code in Algol 60, FORTRAN and UCA3. (Reader, I wrote one!)

Another FORTRAN compiler was written at Leeds University for Eldon 2. It deals with the problem of NEST overflow in complicated expressions by ignoring it. Only one program was ever found to exceed the 16 available cells!

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

The 'Autocode' languages have their genesis in Brooker's pioneering compiler for the Ferranti Mark 1. Enhanced versions were produced for Ferranti's Pegasus and Sirius machines; KDF9 Autocode derives from them. In a parallel line of descent, Mercury Autocode, also due to Brooker, added major new language features such as 'for' loops. K Autocode [Gibbons68] is an implementation of Mercury Autocode for the KDF9. Mercury Autocode also begat Atlas Autocode, seen by its proponents as a more practical alternative to Algol 60. At Edinburgh University, a KDF9 version of Atlas Autocode—IMP [Stephens74]—was aimed at system programming, and was later used to write the EMAS multi-access operating system for the EE System 4/75 and ICL 2900 Series.

6: KDF9 Redux

So much, then, for the historic present. Has KDF9 a future? Many defunct computer systems have been resurrected in emulation and now KDF9 joins them. The following is partly drawn from the **ee9** user's guide [FindlayUG].

At the start of a run **ee9** casts around for files to represent the KDF9 peripherals. The Flexowriter is associated with the user's terminal window (CTRL-C requests a Flexowriter interrupt). Other devices are associated with files. External data may be read and written—at option—in the ISO Latin-1 character set, with automatic conversion to and from the KDF9's internal character codes (which are somewhat device-dependent)—or in the native character set. Several characters in the KDF9 paper tape set are absent from Latin-1, so a transliteration is used to represent them externally.

In **boot** mode **ee9** executes the hardware 9-word bootstrap read from TR0, then jumps to word 0 in Director state. In **program** mode **ee9** reads into core, from TR0, a binary program prepared by a compiler such as the new Usercode assembler by David Holdsworth. Its execution starts at word 0, in program state. The emulator itself services OUTs executed by the program, so that it is not necessary to have a Director running. The most important OUTs of the EE Time Sharing Director are implemented.

ee9 is intended to be portable to any operating system that offers a basic POSIX API. It is written in Ada 2012, using the open-source GNAT compiler. Suitable host systems include macOS, FreeBSD, Linux, and Microsoft Windows equipped with Cygwin or (better) with Windows Subsystem for Linux. To date, **ee9** has been made available on the Intel x86–64 architecture, under macOS and Linux; on the Intel x86–32 architecture, under Linux and Windows; on the ARM11 architecture for the Raspberry Pi, under Raspbian; and on the PowerPC G5 architecture, under MacOS X 10.5 (Leopard).

The *Users' Guide* details the (virtually complete) subset of the full KDF9 architecture currently implemented by **ee9**. Although not perfect, it is more than adequate to run a wide variety of historically–significant programs, including the Whetstone and Kidsgrove Algol 60 systems, John Leech's HLT group-theory program, and the Time Sharing Director. The Algol systems, in particular, allow the famous Whetstone benchmark to be run on its original platform. Emulated by a computer of 2019 vintage, it runs about 500 times faster than it did on the real KDF9 hardware.

The greatest obstacle to achieving a perfect KDF9 emulation is the poor quality of the available documentation. Regrettably, EE's manuals are poorly written, sloppily edited, incomplete, inconsistent, and sometimes plain wrong. As a simple example, the programming reference manual fails to say what is left in the NEST after arithmetic overflow or division by zero. As another, the surviving I/O Control documentation gives incorrect opcodes for all I/O instructions. A few instructions have opcodes or effects that remain conjectural. For example, the 'read C store' order is attested in marginal notes taken during an EE KDF9 training course, but it seems not to be present in extant software. Perhaps it was used by hardware diagnostic programs.

If any readers can supply more concrete information about these or any other KDF9 matters, I would be very glad to hear from them.

Acknowledgements

I am grateful to the group of supporters—enthusiastic former engineers, programmers, or satisfied users of KDF9—for their encouragement during this project. I mention in particular David Hawley and Brian Randell, for their crucial caches of EE documents; David Holdsworth for his Usercode compilers and hardware insight; and David Holdsworth, David Huxtable, Brian Wichmann, Graham Toal, and Roderick McLeod for resurrecting the Whetstone and Kidsgrove Algol systems. Others, not mentioned, know who they are: to them also I extend my thanks.

© William Findlay, Version of 2021-11-13
This document is licensed under a Creative Commons Attribution 3.0 License: http://creativecommons.org/licenses/by-nc-sa/3.0/

REFERENCES AND FURTHER READING

[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: From Batch Processing to Distributed Systems*, ed. P.B. Hansen;
Springer 978-0387951133; 2001.

[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.4; 1962.

[EEC69] *KDF9 Programming Manual*,
Publication 1003 mm, 2nd Edition, International Computers Limited; October 1969.

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

[*FindlayEE] *The English Electric KDF9;* W. Findlay; 2021.

[*FindlaySW] *The Software of the KDF9;* W. Findlay; 2021.

[*FindlayHW] *The Hardware of the KDF9;* W. Findlay; 2021.

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

[*FindlayUG] *Users' Guide for **ee9**: An English Electric KDF9 Emulator;* W. Findlay; 2021.

* For the above six documents, and many others, see: www.findlayw.plus.com/KDF9/

[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*; D.H.R. Huxtable and E.N. Hawkins,
Annual Review in Automatic Programming, Vol. 3, Pergamon Press; 1963.

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

[Naur60] *Report on the Algorithmic Language ALGOL 60*; ed. P. Naur; 1960.

[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.

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

[WHMcC71] *The Eldon 2 operating system for KDF9*; M. Wells, D. Holdsworth and A.P. McCann,
Computer Journal, Vol. 13 No. 1; 1971.

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

[Wichmann73] *ALGOL 60 Compilation and Assessment*; B. A. Wichmann,
Academic Press; 1973.

[WJ76] *Testing ALGOL 60 Compilers*; B. A. Wichmann and B. Jones,
Software: Practice and Experience, Vol.6 No.1; 1970.

For the above three documents, see: sw.ccs.bcs.org/CCs/KDF9/Wichmann/index.html

[WR67] *Competition for memory access in the KDF9*; C. S. Wallace and B. G. Rowswell,
Computer Journal, Vol.10 No.1; 1967.

© William Findlay, Version of 2021-11-13
This document is licensed under a Creative Commons Attribution 3.0 License: http://creativecommons.org/licenses/by-nc-sa/3.0/