KDF9 Software and sample programs.

Much of this section consists of simplified extracts taken from a comprehensive account written by Bill Findlay called *The Software of the KDF9* – see:
http://www.findlayw.plus.com/KDF9/The Software of the KDF9.pdf
http://www.findlayw.plus.com/KDF9/The%20Software%20of%20the%20KDF9.pdf

Operating systems.
English Electric Ltd. Implemented an Operating System for the KDF9 called the Timesharing Director (TSD). Later Operating Systems implemented by customers included Eldon 2 at the University of Leeds, and Egdon, COSEC and COTAN developed primarily at the UKAEA's Culham Laboratories. These latter systems offered multi-access features, usually with PDP-8 front ends to handle the interactive terminals.

TSD.
Under the 'elegantly simple' TSD system, up to four user-programs could be run at once. Job control in TSD was shared between the Director and the computer's human operators. Commands from the operators to TSD were known as 'TINTs', short for Typewriter Interrupts. They were initiated by pressing the interrupt button on the KDF9's console Flexowriter. Programs were divided into two job streams, A and B. A-programs were loaded automatically. At boot time, the operators defined the maximum resource demands of an A-program, in terms of core store and I/O devices that could be used. The operators also defined the maximum number of A-programs that may be run concurrently and assigned them to priority levels. TSD loaded programs to populate the specified levels. When an A-program terminated, any A-programs in lower priority levels were promoted. When sufficient resources became available, a new program was loaded into the vacated, lowest, A-program level. B-programs were loaded in response to operator commands that specified a priority level and a main storage allocation.

ELDON 2
The English Electric Time Sharing Director (TSD) was most effective with a workload consisting of relatively long-running user-programs. With some I/O-limited jobs and CPU-limited jobs to play against each other, it was capable of making very good use of the machine's resources. However, many KDF9 computers were supplied to universities, where the mix of short experimental runs and many failing (student) compilations required far too much human operator intervention under the standard TSD Operating System. It was to tackle these university workloads, and to offer multiaccess working, that Eldon 2 [see Ref. 1 below] was developed at Leeds University. ELDON 2 used a DEC PDP-8 minicomputer, connected via a magnetic tape buffer, as a front-end processor to handle the end-users' terminals.

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 the UKAEA commissioned for these sites from English Electric was named after the area around Winfrith called Egdon Heath, in a novel by Thomas Hardy. EGDON [see Ref. 2] began as a system of a type familiar on other scientific computers of

the day. Broadly comparable with IBSYS for the IBM 7090, it aimed to provide a programming environment that was somewhat compatible with FORTRAN on IBM's 'scientific' machines, and looked to maximise the throughput of relatively long-running scientific programs. Unlike the TSD, EGDON required the KDF9 to be equipped with a disc drive. Its final version, EGDON 3, could use multiple register sets when they were available.

DEMOCRAT.
This was a modular, microkernel based operating system developed at the National Physical Laboratory [Ref. 3]. The name DEMOCRAT stood for "Disc Equipped Modularly Organised Computing with Remote Access and Timesharing"!)

Programming languages.

Usercode, the English Electric KDF9 assembly language, had variants called UCA3 (adapted to card input for EGDON) and KAL4 (for Eldon 2). KAL4 provided the facility for symbolic data names that is lacking in the manufacturer's original Usercode. To understand the following description, it is helpful to bear in mind the KDF9's instruction set as described in section N4X3.

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 'Jump to label 10 if the C-part of Q store 2 is Not Zero'. It is possible that this format was suggested by the KDF9's instruction set, 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 rCqNZS 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 JrCqNZS instruction itself.

Data locations in KDF9 Usercode have rigidly stereotyped identifiers. Variables declared within routines have names of the form Vm. 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 Wm, Ym, YAm, …, YZm. Absolute virtual address m is written Em and, numbered backwards, as Zm, 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 Qq. The 'Q' suffix causes the Qq register to be updated, after the effective address is determined, by adding the contents of Iq to Mq and decrementing the contents of Cq. This allows stepping through variable number of locations, at addresses given by an initial address and a variable stride.

Control transfers (jump instructions) in Usercode. The Jr instruction is an unconditional jump to the instruction labelled r. The instructions: Jr=Z, Jr≠Z, Jr>Z, Jr≥Z, Jr<Z, Jr≤Z, etc, test the sign of the top cell of the NEST; to compare the top two cells of the NEST we have: Jr= and Jr≠. All of these pop N1 whether they jump or not; Jr= and Jr≠ do not pop N2, being the only dyadic operations with this behaviour. To test whether Cq is (non-) zero we have: JrCqZ, JrCqNZ, and JrCqNZS. The JrV and JrNV instructions are conditional on the overflow register being (un-)set,

while JrTR and JrNTR are conditional on the test register being (un-)set; these instructions clear the designated register whether they jump or not. JrCqNZS is known as the short loop jump. It jumps, if Cq is nonzero, to syllable 0 of the instruction word that precedes the word containing the JrCqNZS 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. 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 MkMq and =MkMq, the effective address being the sum of Mk and Mq. 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 Mk is taken as a base word address, and the content of Mq 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 next 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 Qq 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 following 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 of a Usercode program.
Compute the floating-point values $(-b \pm \sqrt{(b2 - 4ac)}) / 2a$

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: ***b2– 4ac***
`JSP40;` N1: ***√Δ****, where* **Δ** = ***b2– 4ac***
`DUP; NEGF;` N1: **-√Δ**; N2: **+√Δ**
`Q2; +F; Q1; ÷F; REV;` N1: **+√Δ**; N2: **(*-b* – √*Δ*) / *2a***
`Q2; +F; Q1; ÷F;` N1: **(*-b* + √*Δ*) / *2a***; N2: **(*-b* – √*Δ*) / *2a***

Compilers and high-level languages.
As for high-level languages for the KDF9, these focussed on ‘scientific’ computing. Algol 60, Babel, FORTRAN, IMP, K Autocode, and KDF9 (Atlas) Autocode were all available; COBOL is a notable absentee, it was promised by English Electric but never materialized.

Algol 60.
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 English Electric installations where they were written. The English Electric Kidsgrove and Whetstone Algol 60 compilers for the KDF9 were among the first of their class. 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 Algol Report.

FORTRAN
A variant of FORTRAN II, called EGTRAN, was implemented on the EGDON operating system. Like most FORTRANs of the era, it included 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.

IMP
The British family of ‘Autocode’ languages had their genesis in R.A. Brooker’s early compiler for the Ferranti Mark I at Manchester [Ref 4]. 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 had nested scopes, like Algol, but passed parameters by reference, like FORTRAN. At Edinburgh University Atlas Autocode was bootstrapped from the Atlas to their KDF9 [Ref 5]. KDF9 Autocode was then used to write the compiler for an improved and modernized Autocode, IMP [Ref 6]. IMP aimed at systems programming, and was used to write the portable EMAS multiaccess system, initially for the English Electric System 4-75 computer.

Other languages.
Despite English Electric’s promises, KDF9 never got a compiler for COBOL, nor an optimising FORTRAN compiler, but compilers for some lesser languages were written.

ALPHACODE [Ref 7] is a rather primitive adaptation of a Brooker-style Autocode to the fixed-format card input and limited character set of the first generation English Electric DEUCE computer; there is a version for the KDF9.

Babel [Ref 8], 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. Like Eldon2 FORTRAN, it translates source programs into KAL4.

K Autocode [Ref 9]—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 [Ref 10] 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.

KDF emulation and simulation facilities.
Since 2007 there has been a renewed interest in KDF9 software archaeology amongst members of the Computer Conservation Society in the UK. This has resulted in a great deal of emulation activity over the last few years and the placing on-line of impressive amounts of original documentation. Refer to http://sw.ccs.bcs.org/KDF9/ Links will be found to much useful KDF9 source documentation, working software, photographs and KDF9 reminiscences. The people involved in this work include David Holdsworth, Brian Wichmann, Bill Findlay and Hans Pufal.

References.

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

2. *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.

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

4. *The Autocode Programs developed for the Manchester University Computers*. R. A. Brooker; Computer Journal, Vol.1 No.1; 1958.

5. *Atlas Autocode Compiler for KDF9*. P. Bratley, D. Rees, P. Schofield and H. Whitfield; Edinburgh University Computer Unit Report No.4; 1965.

6. *The IMP language and compiler*. P. D. Stephens; Computer Journal, Vol.17 No.3; 1974.

7. *ALPHACODE*. KDF9 Service Routine Library Manual, Section 13.2; English Electric Leo; 1966.

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

9. *K Autocode*. A. Gibbons; Computer Journal, Vol.11 No.4; 1968.

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