ee9 Implementation Overview

0. Readership

This document is intended for anyone wishing to maintain or adapt the GNU Ada KDF9 emulator, **ee9**. It presents a synopsis of the program structure, with some important points of detail highlighted. **ee9** is composed of about 100 Ada 2012 source files, each containing either the specification or the implementation of a separately-compiled module. It is complemented by a set of bash-compatible shell scripts for ease of use. These modules are described here on the basis of their specific contribution to the functionality of **ee9**.

1. CPU

1.A The fundamental register structures of KDF9: `KDF9`, `KDF9_char_sets`, `KDF9.decoding`

The KDF9 48-bit word, 24-bit halfword, 16-bit Q store field and 8-bit instruction syllable are defined in `KDF9`; the 6-bit character and its associated character sets in `KDF9_char_sets`. The latter is separated out so it can be included in other programs without dragging in the whole of `KDF9`, which amounts to ~2 KSLOC.

`KDF9` uses the basic machine data types to declare structures representing the NEST, the Q Store, the SJNS, the native and decoded instruction types and the instruction buffers, virtual time management, and the privileged state components. It also implements the KDF9 interrupt system.

The NEST is a LIFO of words, the top of the stack being indexed by a variable of type `mod 19`; this means that subtracting 1 from it when 0 wraps around to 18, while adding 1 to 18 wraps around to 0. This is a property of the Ada type; no conditional logic is explicitly written to achieve these effects, which exactly mirror the behaviour of the KDF9 hardware. A similar variable of type `mod 17` serves the SJNS. Operations are provided to pop, push, read and write the top cells of the NEST and SJNS. The read and write operations are used to avoid unnecessary push/pop overheads when operands are directly over-written by results.

KDF9 had 4 sets of registers, the active set being selected by a 2-bit 'context' register. To avoid indexing a 2-D array on every register access, **ee9** works with a single, fixed set of registers; changing the context swaps them, to/from a bank of 4 sets selected by the old and new contexts.

KDF9 instructions are of 1, 2, or 3 8-bit syllables, as indicated by the first 2 bits of the first syllable: 00 for a single syllable, 01 for two syllables, 10 for three-syllable jump orders, and 11 for three-syllable data fetch/store orders. To simplify the decoding, **ee9** computes a compressed opcode value from each instruction. For one syllable orders, it is simply the least significant 6 bits of the syllable. For two and three syllable orders, it is that value with some irrelevant bits masked off and perhaps some other bits from another field of the order OR-ed in. The two syllable I/O orders are further identified by the least significant four bits of the second syllable and these are extracted and examined separately. The case statements that select the emulation routine for each order are driven by these compressed opcode values, which are listed in the package `KDF9.decoding`.

The KDF9 held 12 syllables of instruction code in its two instruction buffers, `IWB0` and `IWB1`, and was able to execute short loops entirely held in those buffers without repeatedly fetching the orders from store. This behaviour is followed quite closely by **ee9**. Note, however, that **ee9** does not attempt to emulate the concurrent operation of Arithmetic Control and Main Control (see *The Hardware of the KDF9*, §7: MAIN CONTROL).

1.B Non-trivial arithmetic: `KDF9.CPU`

Simple integer arithmetic and logical operations on words, halfwords, and 16-bit fields, are all entailed by the declarations of those types in `KDF9`, and so are available to the `KDF9.microcode` package without further ado. More complicated operations, including 48- and 96-bit shifts, multiplication, division, 96-bit arithmetic, and single- and double-precision floating point arithmetic, are defined in the `KDF9.CPU` package.

Dividing a 96-bit number by a 48-bit number is not easy to do quickly and `KDF9.CPU` finesses the problem by treating the numbers as fixed point fractions and using Ada's fractional arithmetic division. This produces a rounded result, which is perfect for the KDF9 DIVD order, but is not what is needed by DIVR, which produces the truncated quotient and a remainder instead. A correct result for DIVR is derived from the rounded quotient by back-multiplication and then adjusting the quotient according to the relative magnitudes of that product and the original dividend.

There is a perhaps confusing proliferation of shift operations in this package and it seems worthwhile to explain their various uses. See EE Report K/GD.y.80, *KDF9: Shifting and Shift Control*. Each KDF9 shift operation is implemented by `KDF9.CPU` in terms of more basic operations that shift either left or right, but are not capable of doing both, unlike the KDF9 orders. These auxiliary routines may also be used in the implementation of other KDF9 instructions.

The KDF9 arithmetic shift operations `SHA±n`, `SHACq` are implemented by the function:

```
function shift_arithmetic (I : KDF9.word; L : CPU.signed_Q_part)
```

which uses as auxiliary routines:

```
function scale_down (W : KDF9.word; amount : Natural)
function scale_down_and_round (W : KDF9.word; amount : Natural)
function scale_up (W : KDF9.word; amount : Natural)
```

KDF9 arithmetic left shifts may set overflow. Arithmetic right shifts in KDF9 *round the result*. `scale_down` is used when a sign-propagating right shift is called for, but no such rounding is wanted.

The KDF9 double precision arithmetic shift operations `SHAD±n`, `SHADCq` are implemented by the function:

```
function shift_arithmetic (P : KDF9.pair; L : CPU.signed_Q_part)
```

which uses as auxiliary routines:

```
function scale_up (P : KDF9.pair; L : Natural)
function scale_down (P : KDF9.pair; L : Natural)
```

KDF9 double precision arithmetic shifts *do not* move bits into or out of D0 of the less significant word, which is set to 0 in the result unless a zero-length shift is specified (in which case the operand is the result). KDF9 double precision arithmetic left shifts may set overflow. Double precision arithmetic right shifts in KDF9 *never* round the result.

This document is licensed under a Creative Commons Attribution 3.0 License: http://creativecommons.org/licenses/by-nc-sa/3.0/

The KDF9 logical shift operations `SHL±n`, `SHLCq` are implemented by the function:

```
function shift_logical (W : KDF9.word; L : CPU.signed_Q_part)
```

which uses as auxiliary routines:

```
function shift_word_left (W : KDF9.word; amount : word_shift_length)
function shift_word_right (W : KDF9.word; amount : word_shift_length)
```

KDF9 logical left shifts do not set overflow. Logical right shifts in KDF9 do not round the result.

The KDF9 double precision logical shift operations `SHLD±n`, `SHLDCq` are implemented by the function:

```
function shift_logical (P : KDF9.pair; L : CPU.signed_Q_part)
```

which uses as auxiliary routines:

```
function shift_pair_left (P : KDF9.pair; L : Natural)
function shift_pair_right (P : KDF9.pair; L : Natural)
```

KDF9 double precision logical shifts *do* include D0 of the less significant word as part of the full 96-bit operand and result. KDF9 double precision logical left shifts do not set overflow. Double precision logical right shifts in KDF9 do not round the result.

The KDF9 circular shift operations `SHC±n`, `SHCCq` are implemented by the function:

```
function shift_circular (W : KDF9.word; L : CPU.signed_Q_part)
```

which uses as auxiliary routines:

```
function rotate_word_left (W : KDF9.word; amount : word_shift_length)
function rotate_word_right (W : KDF9.word; amount : word_shift_length)
```

KDF9 circular shifts of more than 48 places produce a non-obvious result. They were implemented in the hardware by doing a double precision logical shift of two concatenated copies of the `W` operand and taking the more significant word of the result for left shifts and the less significant word for right shifts. When `L` is greater than 48 this yields, respectively, `W` left-shifted logically by `L-48` places and `W` right-shifted logically by `L-48` places. `shift_circular` replicates this behaviour. Note that there are no double precision circular shifts. Circular shifts in KDF9 do not set overflow.

In addition to the explicit shift operations, there are shifts implicit in other arithmetic operations, including multiplication, division, and all floating point operations. Among these, the operations to normalize a fraction are of particular importance. Normalization shifts are slightly slower than other shifts, as they cannot make single steps of 16 places, being limited to steps of 8 places or less by the hardware's leading-zeros detector.

To standardize a (possibly) non-normalized floating-point number, setting overflow when necessary:

```
function normalized (R : CPU.f48)
```

To convert a single precision fraction to a rounded, standardized 39-bit mantissa, and adjust its exponent accordingly:

```
procedure normalize (fraction, exponent : in out KDF9.word)
```

To convert a double precision fraction, and an algebraic scale-factor exponent, into a double precision floating point number, setting overflow when necessary:

```
procedure reconstruct (frac   : in out KDF9.pair;
                       scaler : in KDF9.word)
```

These is little more to be said in general about `KDF9.CPU`; full understanding demands a close reading of its details.

1.C The core store: `KDF9.store`

The implementation of the core store as an array of KDF9 words is straightforward, apart from the need to implement store lockouts. The procedures `validate_access` and `validate_range_access` are used before fetching or storing a location, and if it is found to be in a locked-out group, raise the `LOV_trap` exception. The latter is handled at the top level of control flow, in the `execute` procedure, which calls `IOC.handle_a_main_store_lockout` to put the lockout into effect. See §2.A. On return to `execute`, the emulation loop resumes.

1.D The microcode: `KDF9.microcode`

This package controls instruction decoding, operand preparation, and instruction dispatching. Although quite big (~1600 SLOC), thanks to the design integrity of the KDF9 it is straightforward.

1.E Interrupts and other exceptions: `exceptions`, `KDF9`, `KDF9.microcode`, `KDF9.Directors`, `execute`

The procedure `execute` is responsible for overall control of the progress of emulation; it contains the (ultimate) handlers for nearly all exceptions. The package `exceptions` declares a set of exceptions of global significance. The raising of most of these exceptions entails the end of execution, as mediated by `execute`.

Another set of exceptions, which model the KDF9's different interrupt reasons, is declared in `KDF9.ads`. Each has a number associated with it; this is the number of the corresponding bit in RFIR (the Reason For Interrupt Register).

When running in boot mode these exceptions cause an interrupt into Director, the responsible instruction being abandoned by raising the exception `abandon_this_order`. `execute` handles this by doing nothing, thereby passing control on to the next order, at location 0, and so effecting the interrupt.

In test program mode all interrupts except OUT, LOV and RESET are ignored; RESET is treated as a failure. In program mode all interrupts except OUT and LOV are treated as failures. OUT is implemented by the Director API emulation in `KDF9.Directors`.

For the special treatment of LOV in the test program and program modes, see §2.A.

This document is licensed under a Creative Commons Attribution 3.0 License: http://creativecommons.org/licenses/by-nc-sa/3.0/

2. I/O

2.A I/O Control and KDF9 peripherals: `IOC` and its descendants

KDF9 I/O is implemented by a set of orders that address a peripheral device by means of the number, in the range 0..15, of the 'buffer' to which it is connected. A KDF9 buffer was in fact a DMA channel, and it was feasible for all 16 to be active simultaneously, the peripheral complement being such that no device could be starved of core cycles. A buffer was specific to the type of its device: a paper tape reader, for example, could be switched between TR buffers, but could not be connected to a CP buffer. In **ee9**, at present, the buffer number of each device is fixed at the start of runtime, either by default or in response to a configuration option.

ee9 invokes the correct procedure to implement an I/O operation by indexing the array `IOC.buffer` with the buffer number operand. The elements of `IOC.buffer` are class-wide pointers to the device objects. When enabled, each device plugs a pointer to itself into the appropriate element of `IOC.buffer`. That pointer is used to dispatch to the method proper to the device type. These types are declared within their defining packages, forming an OOP hierarchy, but there is some overlap of concerns beween the TP and GP packages, because the graph plotter connects to a TP buffer:

`IOC.device` (abstract — objects cannot be created)

- `IOC.absent.device`
- `IOC.fast.device` (abstract — objects cannot be created)
  - `IOC.fast.DR.device` (DRum)
  - `IOC.fast.FD.device` (Fixed Disc)
  - `IOC.fast.MT.deck` (both EE 1081 16-track and Ampex TM-4 7-track Magnetic Tape decks)
- `IOC.slow.device` (abstract — objects cannot be created)
  - `IOC.slow.shift.device` (abstract — objects cannot be created)
    - `IOC.slow.shift.FW.device` (FlexoWriter)
    - `IOC.slow.shift.GP.device` (Graph Plotter)
    - `IOC.slow.shift.SI.device` (British Standard Interface)
    - `IOC.slow.shift.TP.device` (Tape Punch)
    - `IOC.slow.shift.TR.device` (Tape Reader)
  - `IOC.slow.unit.device` (abstract — objects cannot be created)
    - `IOC.slow.unit.CP.device` (Card Punch)
    - `IOC.slow.unit.CR.device` (Card Reader)
    - `IOC.slow.unit.LP.device` (Line Printer)

Fast devices do one core cycle per word; slow devices do one core cycle per KDF9 character (called a '`symbol`' in **ee9**, to avoid confusion with the host computer's Latin-1 character set). Thus their timing properties differ. Moreover, `IOC.slow.unit` devices always traverse one or more complete 'unit records'—cards or printed lines—whereas `IOC.slow.shift` devices traverse just as much of the external medium as is needed for the characters transferred.

ee9 not only emulates the data transfer for each I/O operation, it also simulates the physical time elapsed in the transfer. A second operation cannot be started on a buffer while it is considered to still be busy with a transfer that was previously initiated. The KDF9 programmer has orders—TLOQq, INTQq, and BUSYQq—that allow a program to interrogate the state of a transfer on a buffer; and an order—PARQq—which allows the success of a terminated transfer to be determined. Applying any operation other than BUSYQq to a busy device, or attempting to access the same core store groups as those the transfer is accessing, causes the program to be locked out until the transfer terminates.

The `IOC` method that emulates an I/O instruction, in reality, effects the whole transfer immediately; then computes the KDF9 device's predicted end-of-transfer (PR interrupt) time, and sets the necessary store lockouts, before returning. The physically immediate end of the transfer does not give rise to observable differences from the behaviour of the hardware, because problem-program transfers are effectively atomic, this being ensured by the lockout mechanisms. As far as a program is concerned, once a transfer has been initiated, its result can be inspected only after it has terminated: that is, only after its end-of-transfer time has been reached.

This is not true of the Director, because lockouts are inoperative in Director state; but Directors are written to avoid any danger this freedom might expose them to, *inter alia* by explicitly setting and checking the implicated lockout registers. The immediately-following discussion therefore assumes that the transfer was started by a problem program that is not running under Director control, i.e. not in **ee9**'s boot mode.

Once started, a transfer can lead to two quite different sequences of event, depending on whether it terminates without other effect on the program, or leads to the program being locked out.

If the program succeeds in running past the nearest PR time without being locked out then **ee9** merely takes note of this fact, clears the transfer's lockouts, and sets the buffer idle. This is done in `IOC.act_on_pending_interrupts`, which is called at the end of an instruction cycle when `the_elapsed_time > the_next_interrupt_time`, and itself calls `IOC.finalize_transfer` when it finds a busy buffer whose completion time has passed.

`act_on_pending_interrupts` also sets a new value for `the_next_interrupt_time`, which is either the *next* expected PR time; or 2^{64}-1 if all buffers are idle; or, in boot mode, at most 1 virtual second in the future, to prevent a 'double clock' RESET interrupt. The KDF9 actually checked for interrupts only at certain points in its microprogram, not at the end of every instruction; see *The Hardware of the KDF9*, §7, MAIN CONTROL. **ee9** is more responsive: it checks for, and actions, any interrupt requests at the end of every instruction execution.

On the other hand, if the program transgresses on a locked-out store area, or attempts another operation on a busy buffer, then a different logic comes into play. To model this, **ee9** takes a lead from the techniques of discrete event simulation: `the_elapsed_time` is advanced to the predicted completion time of the responsible transfer, the buffer is set idle, its

This document is licensed under a Creative Commons Attribution 3.0 License: http://creativecommons.org/licenses/by-nc-sa/3.0/

store lockouts are cleared, and execution of the problem program continues. The net effect is that the program sees the elapsed time as having jumped forward exactly as would have happened in reality, between its being suspended and being resumed. These effects are mediated by three routines in `IOC`: `handle_a_buffer_lockout` takes the case of a busy device, `INT` takes that of the INTQq instruction, and `handle_a_main_store_lockout` is called by the core store access routines; they all invoke the procedure `KDF9.advance_the_clock` to update `the_elapsed_time`, and then call `act_on_pending_interrupts` to deal with the rest of the necessary housekeeping.

When **ee9** is working in boot mode, things happen rather differently.

If Director is active when it is discovered that `the_elapsed_time > the_next_interrupt_time` then the PR flag is set in the RFIR (Reason For Interrupt) register; Director is not interruptible, but will eventually notice the interrupt request and deal with it. If a problem program is active, then all of the previously-described apparatus comes again into play, but instead of merely resuming the program, a PR interrupt is effected if the transfer was initiated on behalf of a program of higher priority than the one running. That decision is based on the contents of the Program Holdup (PHU) registers, allowing Director to reschedule the CPU. (To be precise, the interrupt requested will be EDT if the transfer was started by Director, or if the PHUs indicate a possible priority inversion over access to the interrupting buffer.)

A lockout when running a problem program in boot mode effects a LOV interrupt: the whole issue is punted to Director.

2.A.1 The bootstrap: within `IOC.slow.shift.TR`

Both reading the hardware bootstrap of 9 words, and binary program loading in non-boot mode, are implemented here.

2.A.2 The Flexowriter console: `IOC.slow.shift.FW`

The Flexowriter, represented in **ee9** by the user's terminal window, has several unusual features: it includes an 'edge-punched card' reader; it outputs text in red ink; and is the source of FLEX interrupts, by means of which the operator gets the attention of Director. The edge-punched card reader is emulated using the external file "`FW0`"; see *Users Guide for **ee9***, § 3.2. The red output and the non-escaping underline are emulated, at option, by using ANSI SGR terminal escape sequences to style the displayed text appropriately. FLEX interrupts are emulated by typing control-C then RETURN.

2.A.3 Magnetic tapes: `IOC.fast.magtape`

Magnetic tapes are represented by Ada direct access files, using the `Ada.Direct_IO` library package, to allow selective overwriting of blocks. This is necessitated by the MWIPE and MGAP operations of the 1081-type tape deck.

The "write permit ring", which was a metal band inserted into the tape reel concentrically with the hub, depressed a switch to enable the write heads. It is modelled by the file's access permission. Making the file read-only simulates an absent ring, allowing operations that do not change the contents of the tape, and failing those that do.

Each tape block, and each length of erased tape, is represented by one or more "slices", a slice being a record in the direct access file. A slice has two components: a string and per-slice metadata. The string contains the Case Normal Latin-1 transliteration of all or part of a KDF9 tape data block. In the case of an erasure slice the string reserves space for possible future over-writing by data, but its contents are of no significance. The metadata is as follows.

- Byte 0—whether the slice represents a data block (with code 'D'), a length of tape erased by the MWIPE operation (code 'W'), a length erased by the MGAP operation (code 'G'), an even-parity tape mark (code 'e'), or an odd-parity tape mark (code 'o'). Tape marks are present in 7-track tapes only.

- Byte 1—a code made up as follows:
  (a) if the block is 'LBM' marked; i.e. if it was written by a MLWQq or MLWEQq instruction instead of the normal MWQq and MWEQq instructions, and so responds positively to the MLB order: +64;
  (c) if this is the last slice of a multi-slice block: +8; and
  (b) if this is the first slice of a multi-slice block: +1.

  The second byte therefore takes the following possible values (decimal = octal = Latin-1):

  00 = 000 = `NUL` ⇒ no flags
  01 = 001 = `SOH` ⇒ first slice of block
  08 = 010 = `BS` ⇒ last slice of block
  09 = 011 = `HT` ⇒ only slice of block (first and last)
  64 = 100 = `@` ⇒ LBM flag
  65 = 101 = `A` ⇒ first slice of block with LBM flag
  72 = 110 = `H` ⇒ last slice of block with LBM flag
  73 = 111 = `I` ⇒ only (first and last) slice of block with LBM flag

- Byte 2—the length of the string in this slice.

An emulated tape block is represented by a number of consecutive slices such that the total length of their strings is sufficient to encompass the KDF9 data or erasure. The maximum string size has been set so that blocks of 256 words (as used by POST), and short blocks containing an OUT 8 print-line image, both have a space efficiency of better than 95%. The maximum block size (not slice size) is 32KW, or 256K characters, that being the size of the largest core store.

Ampex 7-track tape files are recorded in exactly the same way, with the addition of single-slice blocks representing the two kinds of tape mark. These have 'I' in byte 1, have length 1, and contain the character '§' in the single data byte. When read into core this transfers as #170000000000 to the single input word.

No attempt is made to model either interblock gaps or the erased tape extending from the Beginning of Tape Window (BTW) to the start of the first block of data. However interblock gaps are taken into account when estimating the elapsed time of magnetic tape transfers.

2.A.4 Disc and drum stores: `IOC.fast.FD`, `IOC.fast.DR`

Disc and drum storage is represented by the direct access file `FD0` and `DR0`, respectively, using `Ada.Direct_IO`. The storage is accessed as an array of data blocks without metadata content, one record of the file corresponding exactly to one KDF9 disc sector (320 characters) or one drum sector (1024 characters). Data is stored in the files using the Case Normal Latin-1 transliteration.

This document is licensed under a Creative Commons Attribution 3.0 License: http://creativecommons.org/licenses/by-nc-sa/3.0/

2.A.5 The Standard Interface buffer: `IOC.slow.shift.SI`

Little is known about this device as yet, so its present implementation is a placeholder for future developments.

2.A.6 Dynamic configuration: `*.enable`, `*.re_enable`, `IOC.equipment`, `IOC.absent`

The complement of I/O devices to be included in the configuration for a run is set up dynamically. A default configuration is specified in the package `IOC.equipment`. It may be superseded by an options file. `IOC.equipment` provides a procedure, `configure`, which is called after processing the options file and calls the `enable` procedure of each included device. `enable` dynamically creates an object of its type and the `Initialize` procedure of the type plugs a pointer to the device into the `buffer` array. This setup may itself be superseded by a miscellany parameter. `IOC.equipment` provides a second procedure, `re_configure`, which calls the `re_enable` procedure of each device so specified. At present, only DR, FD and SI devices have a `re_enable` procedure. A similar procedure, `install_GP0`, is called if the miscellany parameter requires GP0 to replace TP1. The package `IOC.absent` provides the `absent.device` type, which ensures that accessing an unassigned buffer fails with a clear error message.

2.B Buffered I/O streams: `host_IO`

Buffered I/O streams act as middleware between the KDF9 I/O operations and the POSIX I/O system calls that actually perform data transfers. They avoid having to do a system call for each KDF9 I/O operation. An output stream may be flushed to its output file, emptying its buffer. It is possible to connect a stream to several different files in succession, to enable the continuation-file feature; see *Users Guide for* ***ee9***, § 3.3.

There is provision for injecting a given string into a stream; it is presently used only by `IOC.slow.shift.FW` to support the edge-punched card reader.

The procedure `do_not_put_byte` does everything expected of `put_byte`, except for actually transferring data. It is presently used only when Flexowriter output is suppressed, to maintain the statistics of the device correctly.

The procedure `put_escape_code` writes to a device directly, without affecting its stream buffer; it is presently used only for writing terminal control escape codes, to effect colour change and underlining for the Flexowriter emulation.

The procedure `put_EOL` outputs a line terminator to a stream, in the form needed by the host OS; see §3 of this document.

2.C POSIX thin binding: `POSIX`

Only those system calls necessary for the implementation of **ee9** are included. For want of any better home, `POSIX` also contains the procedures `data_prompt` and `debug_prompt`, which interact with the user when the end of data is reached on an input device, and when execution reverts to single-stepping, respectively.

2.D User interface I/O: `HCI`

A fairly general logging feature is implemented, with the possibility of output to a selection of different kinds of log. At present logging to an external file, and logging to the user's terminal are provided.

2.E Logging: `logging`, `logging.{panel, file}`, `generic_logger`

The package `logging` declares a logging API, which is implemented in the packages `logging.panel`, for the terminal, and `logging.file`, for the external file log. The package `generic_logger` allows for the declaration of distribution lists, i.e. sets of log destinations. It is instantiated by `HCI`. The `logging.panel` package also provides simple prompt/response interaction for control of single-stepping, access to the debugger, etc.

2.F Usage of Ada Text I/O: `Text_IO`, `Long_Float_Text_IO`, `Direct_IO`, `Enumeration_IO`

The standard text I/O packages are used as expedients in various places, such as `logging.file`, `settings.IO`, and `IOC.slow.shift.FW`, whenever a simple line-oriented facility suffices. `Enumeration_IO` is instantiated in the `settings` package for `diagnostic_mode_IO`, `execution_mode_IO`, and `authenticity_mode_IO`; and in `settings.IO` to create `colour_IO` and `width_IO` for the GP-related options. `Direct_IO` is used to implement the `IOC.fast.magtape.device`. The **ee9** package `file_interfacing` provides trivial `open` and `close` operations with exception handling for `Ada.Text_IO` file types.

2.G Graph plotter and Encapsulated PostScript (EPS) I/O: `plotter`, `postscript`

A `host_IO` stream is used to write the EPS file that represents the roll of paper in the graph plotter. An EPS file includes a line near its start containing the maximum *x*- and *y*-coordinates used in the picture. These are not known until the end of the plot, so `postscript.initialize_PS_output` makes a note of the position of these strings in the GP0 file, and replaces them with placeholders. `postscript.finalize_PS_output` seeks back to these placeholders and over-writes them with the actual values. `plotter` uses `postscript` to implement plotting actions.

2.H OUT-invoked I/O: `KDF9.Directors`, `IOC.fast.DR.OUTs`, `IOC.fast.FD.OUTs`, `IOC.fast.MT.OUTs`

In non-boot modes, **ee9** approximates OUT 8, i.e. Director-spooled output, by immediate (unspooled) output to the ultimate destination device. This is coordinated by `KDF9.Directors`, which also handles programmed overlays (OUTs 1 and 2), and allocating and deallocating slow peripherals (OUTs 5, 6 and 7). Mounting labelled magnetic tapes (OUTs 4 and 10), is punted to the package `IOC.fast.MT.OUTs`. The drum store API (OUTs 11, 12, 13 and 14) is punted to the package `IOC.fast.DR.OUTs`; the disc store API (OUTs 41, 42, 43, 44 and 45) is punted to `IOC.fast.FD.OUTs`.

2.I The Program Hold-Up Store: `KDF9.PHU_store`

Emulates the behaviour of the PHU registers when running in boot mode.

2.J Settings file I/O: `settings.IO`, `settings.IO.{colour_IO, width_IO}`, `settings.{diagnostic_mode_IO, execution_mode_IO, authenticity_mode_IO}`

Should be self-explanatory.

This document is licensed under a Creative Commons Attribution 3.0 License: http://creativecommons.org/licenses/by-nc-sa/3.0/

3. Host system dependencies: package `OS_specifics`, `get_O_BINARY`

The only non-portable source code in **ee9** is contained in the body of `OS_specifics`. It has a single, OS-independent package specification, adaptation to the intended host OS being achieved by selecting a body file at compilation time.

`OS_specifics` offers the following simple API:

```
procedure make_transparent
```

This does absolutely nothing on macOS and Linux. On Windows it makes a POSIX file read or write data transparently, without Microsoft's conversion of line terminators, which would corrupt data such as KDF9 input in paper tape code. **ee9** handles line terminators automatically, and does not need this 'assistance'.

To enable this mode, `make_transparent` calls `_setmode` with the `O_BINARY` flag as defined in the `fcntl.h` header file. To make that accessible to the Ada program in a portable manner, we have the C function `get_O_BINARY`:

```
#include <fcntl.h>

int get_O_BINARY ()

{return (int)O_BINARY;}
```

It returns `O_BINARY` as a C `int`, and `make_transparent` simply passes that value on to `_setmode`.

```
function UI_in_name return String
```

This returns the interactive input device name appropriate to the host OS; i.e. `/dev/tty` for macOS and Linux, and `CONIN$` for Windows.

```
function UI_out_name return String
```

This returns the interactive output device name appropriate to the host OS; i.e. `/dev/tty` for macOS and Linux, and `CONOUT$` for Windows.

```
function EOL return String
```

This returns the line terminator appropriate to the host OS: LF for macOS and Linux, and CR LF for Windows.

4. The `ee9` and `execute` procedures

`ee9` is the 'main program'; it checks and registers the command parameters, then calls `execute`, which contains the logic to co-ordinate program loading and instruction sequencing for the various execution modes.

5. Diagnosis and debugging

ee9 is compiled with all of Ada's language-mandated checks enabled. Experiment shows a negligible increase in speed when they are all turned off. A second mode of compilation is provided by the **`mk9`** shell command, which enables many more compile-time warnings, and activates the **`pragma`** `Debug` feature whereby a call to a diagnostic procedure is included in the object program only at option. There is also a run-time debugging flag, and the diagnostic `put_message` procedure generates output only when that flag indicates that the output is wanted.

Usercode-format core prints are implemented using a recursive analysis of both control flow and data flow in the loaded program. Words of core are flagged as orders, data, or undecided. These markers drive the way in which each word is presented in the core print: either disassembled, or as data in a variety of styles.

The flow analyses are undertaken only when Usercode-format core printing is requested in a settings file.

This document is licensed under a Creative Commons Attribution 3.0 License: http://creativecommons.org/licenses/by-nc-sa/3.0/

6. Utility programs that run on the host

a2b: reads data from standard input in a stated code and copies it to standard output in another form. Conversions are available between raw bytes and paper tape code, between paper tape code and Latin-1, and from paper tape code to octal in half/word, Q-store, syllable and character formats. **a2b** can also generate a program call tape, in KDF9 code suitable for use with a Director, based on the 'A block' that starts a binary object program.

kidopt: outputs a 'poke' settings file line to initialize the options used by the Kidsgrove Algol compiler.

mtp: reads a magnetic tape file and writes an understandable analysis of it to standard output.

rlt: is a shell command that runs the KDF9 program **RLT**, which writes a valid label block on all of the magnetic tape files. **rlt** reads the provided data file `RLT_data.txt` to get the desired labels for the six tape files `MT0` through `MT5`. To change the labels written, either amend that file and re-run **rlt**, or supply the name of another file to be used, thus: `rlt other_labels`, to use (e.g) the file `Data/other_labels.txt`.

plt: is a shell command that runs the KDF9 program **PLT**, to create or update a **p**rogram **l**ibrary magnetic **t**ape file.

For more information, see the *Users Guide* and the accompanying document *README*.

7. `kalgol` and `tsdnine`

Two of the shell files that make **ee9** (much!) easier to use are sufficiently complicated to warrant some explanation.

7.A The `kalgol` shell command

The resurrected Kidsgrove Algol system is something of a chimaera, consisting as it does of modules that are original components, modules that were recently recreated, and modules that exist to replace the operating system context (specifically, the POST and PROMPT development environments) within which it used to run. These components are diverse in form, and were created by David Holdsworth and David Huxtable (see the *README* file) using tools that produce results not immediately compatible with **ee9**. Consequently, they require some conversion to an authentic format.

The compiler expects to be given the source code already lexically analysed and converted to Algol Basic Symbols. Lacking both POST and PROMPT, this preliminary pass is implemented by the host-native C program **mkchan**, which may be compiled using the `mk9` command in `Build` with the `KAlgol` parameter. That command also converts the KDF9 executables `MKSYS2` and `KAB00DH--USU` to **ee9** format, using **a2b**.

`KAB00DH--USU` is the program that **kalgol** invokes to carry out the compilation *per se*; it is the driver for the compiler's many overlays (known as 'bricks'). `MKSYS2` converts the bricks from a machine-independent textual format, held in the text file `systape.txt`, to an **ee9** magnetic tape file.

Compiling **mkchan**, and converting `KAB00DH—USU` and `MKSYS2`, need be done just the once each time a new version of the compiler is released, which happens infrequently. At the time of writing `systape.txt` is updated much more often, and it is expedient to convert it to **ee9** format each time a compilation is carried out.

So the **kalgol** command:

1. uses **rlt** to establish a set of scratch tapes
2. runs `MKSYS2`, using **nine**, to transcribe the compiler overlays to one of those tapes
3. runs **mkchan** to convert the source program from Latin-1 to Algol Basic Symbol code
4. runs **kidopt** to generate a settings file for the compiler
5. and only then runs `KAB00DH—USU`, using **nine**, to do the compilation.

On completion of the Algol compilation, **kalgol**:

6. uses **mtp** to extract the Usercode object program from one of the compiler's magnetic tape files
7. tidies up the object code with the **neat** script, using the Unix stream editor, **sed**
8. and finally, compiles the prettified Usercode with the **ucc** script, using the **kal3** assembler.

Whew!

The **kids** command merely invokes **kalgol** to do the compilation, and **nine** to run the object program.

7.B The `tsdnine` shell command

The problem solved by the **tsdnine** command is that Director takes data in from its paper tape readers in KDF9 paper tape code, whereas users want to present data files on the same readers in Latin-1 code. Moreover, program loading is commanded by first reading a 'call tape' that designates the program name and load medium. This too needs to be in paper tape code. **tsdnine** therefore creates a paper tape input file that contains, concatenated together: a call tape, generated by **a2b** from the 'A block' of the program to be run; the program itself; and the data file (if present) converted from Latin-1 to paper tape code by **a2b**.

Having made these arrangements, **tsdnine** delegates the actual running of Director to another script, **tsd**, which invokes **ee9** in boot mode and displays any legible outputs at the end of the run.

8. Odds and ends

Can you think of anything else that needs explanation? If so, let me know: kdf9@findlayw.plus.com.

This document is licensed under a Creative Commons Attribution 3.0 License: http://creativecommons.org/licenses/by-nc-sa/3.0/