

Semper VI

Implementation Guide

Primitive routine specifications
System generation and installation

Copyright © 1987,1988: Synoptics Ltd, All Rights Reserved

Introduction: the scope of this document

This guide explains how you *install* Semper on a computer system, and is for most purposes only relevant therefore if you are doing this yourself, though it is possible some of the information may be of more general value. Its main purpose is to explain the concepts behind the interface that must be provided to link Semper to your computer's environment, and give the detailed specification of the routines that exploit this; it also includes information on adjustments that may be needed in the basic system code, and on running the system generation and help library utilities.

When the installation has been completed to the specifications given in this document, you will have a Semper system with standard display facilities. Such a system does not have the facility to input images via the framestore, e.g. camera input. Control of framestore features, such as switching of overlays, specifying overlay colours and camera input, will require additional Semper verbs to be written and are beyond the scope of this document.

VAX/VMS users are supplied with relatively detailed installation instructions that supplement the general information in this document with a step-by-step guide specific to that environment.

Most of Semper is written in standard Fortran 77. Semper also makes use of the Hollerith data type for the reading and writing of character strings. Holleriths are provided as an extension to the standard by most Fortran 77 compilers. Those operations that need to be machine-dependent, such as character code conversion, display access and efficient disc and tape access, are localised in a small number of *primitive* routines, written afresh as necessary to suit each different environment; these constitute the 'interface' referred to above. You have to write these for your machine, supporting as much as possible of the standard interface Semper expects.

Besides the writing of the primitive routines, a number of minor adjustments also need to be made to the basic code of Semper as part of the installation process; these cover matters such as providing Semper with information about how its environment is configured (e.g. the length of the various Fortran data types, the size of disc blocks, the space to be allocated for various system tables, and the input/output unit numbers to be used), and accommodating variations between Fortran systems (e.g. additional i/o specifiers used in Fortran open statements, and issuing terminal prompts without a carriage return before the user's response).

1. The Primitive Routine Specifications

Unless otherwise stated, the datatypes of all arguments appearing in the specifications below are as Fortran defaults their names; the addition of *4 to an integer name indicates that it is declared integer*4 in the calling program (see section 1.2). The typeface of arguments indicates their mode of use: roman for input usage only, *italic* for output only, and bold for input and output.

1.1 Character access

Semper 6 handles string characters as integer values in the range 32-126, with the following assignments (the standard Ascii assignments), beginning with 'space':

```

32:  ! " # $ % & ' ( ) * + , - . /
48:  0 1 2 3 4 5 6 7 8 9 : ; < = > ?
64:  @ A B C D E F G H I J K L M N O
80:  P Q R S T U V W X Y Z [ \ ] ^ _
96:  ' a b c d e f g h i j k l m n o
112: p q r s t u v w x y z { | } ~

```

These are subsequently referred to as *internal code* (i/c) characters. The file icset contains parameter constants for all the i/c values. A primitive routine is required that converts Hollerith (A1) characters to i/c characters, and another that converts i/c characters to character*1 data.

A1CONV CALL A1CONV(ibuf,n)

converts the n elements of array ibuf from Hollerith (A1) character form to i/c form.

MKCHAR CHARACTER*1=MKCHAR(ic)

returns the character*1 equivalent of the i/c character ic; may be identical to Fortan 77 intrinsic function char if you are lucky in the local collating sequence.

INKEY INTEGER=INKEY(idummy)

returns the i/c equivalent of a single keystroke read from the terminal, by-passing Fortran i/o if possible so as to render the normal terminating carriage return unnecessary. Keystrokes which have no i/c equivalent should be returned with a unique positive integer value that is outside the range 32 to 126. In the file icset you will find parameter constants defined for carriage return, left, right, up and down cursor keys, function keys and character and line delete. Semper currently makes use of carriage return and cursor left and right (parameter constants kbret, kbleft and kbrite) and you must ensure that inkey returns precisely these values. Any keystroke which represents an abandon request (see abandon) must cause a return from inkey with a zero value and the call to abandon, which should immediately follow every call to inkey in Semper, must report the abandon request.

?
CP
ABANDON
Spec

1.2 Pixel forms

Four distinct *forms* (representations or datatypes) are used for pixels in Semper 6, called byte, integer, fp and complex, and identified by *form numbers* 0, 1, 2 and 3 respectively. The integer, fp and complex forms are identical to the Fortran integer, real and complex datatypes (the last being two real values in succession with the real part before the imaginary part), and pixels in these forms are held and manipulated by Semper in Fortran arrays of the relevant datatype. The byte form is unsigned, allowing values in the range 0-255 (not -128 to 127 like the byte datatype in VAX/VMS Fortran); no Semper code refers to it directly, as few compilers provide the necessary datatype, and a primitive routine is required to convert arrays in this form to and from other forms.

You have to make a decision about your integer datatype. For most purposes, a 16bit (integer*2) pixel is the most suitable, allowing negative as well as positive values with magnitudes up to 32,767 and requiring only twice the space of a byte picture; normally therefore, you should try to arrange that the declaration `integer` in Semper's Fortran source means `integer*2`, perhaps via a compiler option. If you positively prefer a 32bit (integer*4) pixel for the sake of the larger magnitude (up to about 2,000,000,000), in spite of the doubled space requirement, you should arrange your compiler options so that `integer` means `integer*4` instead. Of course, it may be that you have no choice, and are obliged to adopt whatever interpretation your compiler applies.

You may like to know, while thinking about this decision, that the Semper source code uses an explicit (and non-standard) `integer*4` declaration for any integer variables whose value may exceed the range of an `integer*2` datum (e.g. disc block numbers). You will not therefore provoke overflow in other areas by opting for the shorter integer length, provided that this `integer*4` declaration is honoured by your compiler; if it is not, *and* the default length is 16bits, you have to edit these to read simply `integer` instead, and your installation will indeed be restricted in the maximum size that various data structures may have - details are given below as they arise.

In summary: try to make `integer` mean `integer*2`, and `integer*4` mean just that.

CFORM CALL CFORM(inbuf,outbuf,inf,outf,n*4)

converts data with form `inf` to data with form `outf`. The data is passed to `cform` in array `inbuf` and returned via array `outbuf`. `inbuf` and `outbuf` may point to the same area in memory, so your conversion loop should run backwards through the array when converting to a longer form. There are `n` data values to convert. If possible, you should provide fast 'block move' code for the cases where `inf = outf`.

It is worth expending some effort to make this routine efficient, as it is heavily used by Semper for the 'on the fly' form conversions which make it appear capable of processing pictures stored in any of the four forms; all transfers to or from the 'disc cache' buffers, including those where no actual form conversion is required, are made with `cform`.

1.3 Data storage

Semper 6 needs to store large volumes of data. This data can take the form of pictures, program libraries and help libraries. Semper can make use of three distinct types of storage media: disc, tape and framestore. Random access disc files are used to store pictures, programs and help information. There is also a special disc file called the 'work disc' which is a temporary file created by Semper to hold some of its data structures, those that are too large to be held permanently in memory, e.g. the display partition table, luts, etc. Tape and framestore devices are only used for storing pictures. Any number of disc and tape files, up to a pre-determined maximum, can be assigned during a Semper session. Only one framestore device may be assigned at any one time.

Semper refers to each storage device by means of a unique integer *device* number. Device 0 is always the work disc and device 1 is always the framestore device. The device number is passed to the disc and tape primitive routines. Therefore disc devices are numbered 0 or 2 upwards and tape devices from 2 upwards. The parameter *ndvs* in the *params* file determines the upper limit for device numbers. Because the device number for the framestore is fixed, no device number is passed to any of the framestore primitive routines.

1.4 Disc access

Basic picture storage in Semper 6 is in disc files, each of which may hold many pictures, and several of which may be in use at once. All such disc files are seen by Semper as being organised in randomly accessible fixed length blocks (*Semper logical blocks*, or simply *blocks* subsequently). An efficient primitive is needed to read and write these blocks to and from memory buffers; this commonly (though not invariably) needs to make lower level calls to system services than the Fortran read and write statements provide. In most installations, substantial numbers of disc blocks are buffered by Semper in memory areas called *disc cache buffers*, and the primitive reads and writes these buffers as a whole, so that it is only at this level that physically random access is desirable. When this route cannot be adopted (for want of addressable memory), physically random disc access is desirable at the individual block level.

If you are not *caching*, the Semper logical block size is effectively dictated by the physical sector size on your computer's drives. Otherwise, however, you have to decide the size yourself. It will probably need to be a multiple or sub-multiple of the physical sector size, so that the cache buffers can be read and written efficiently. Too large a size will waste disc space, since many Semper data structures, including picture rows, are stored *block-aligned*, i.e. beginning at the block boundaries; On the other hand, if the *integer*4* declarations (see section 1.2) in fact achieve a no better than a 16bit datatype, the maximum block number is only 32,767 and too small a block size limits the size of your disc files. Semper requires a block size of at least 18 integer storage units. A block size of 64 bytes is typical when there is no problem with *integer*4* declarations, and 256 bytes or 512 bytes otherwise.

An alternative version of the Semper system routine disc for use in a non-caching version of Semper is to be found in the file `ncdisc`.

MCDC61 CALL MCDC61(*iop*,*idev*,*n*4*,*num*,*ibuff*,*ierror*)

performs data transfers and disc file assignments and deassignments according to the opcode *iop*:

- iop* = 0: initialise disc access (called once at Semper startup)
- 1: reads blocks *n* to *n+num-1* of device *idev* to array *ibuff*
- 2: writes blocks *n* to *n+num-1* of device *idev* from array *ibuff*
- 3: assigns existing file as device *idev* for full read/write access
- 4: assigns existing file as device *idev* for read-only access
- 5: creates new file of *n* blocks; assigns as device *idev* for full read/write access
- 6: deassigns device *idev*
- 7: deassigns device *idev* and deletes file

In the event of an input/output error of any kind, `mc61` sets *ierror* to 8, and if any error occurs while creating, assigning or deassigning a file, `mc61` sets *ierror* to 44. Otherwise, it returns *ierror* unchanged.

For *iop* = 3 to 7, the file name is supplied as i/c chars in *ibuff*(2,3..), with the number of characters in the name in *ibuff*(1). *iop* = 4 is intended to allow access to other people's (possibly protected) files as well as to prevent accidental data overwriting: actual read-only enforcement is not in fact required, as Semper remembers the opening mode and prevents any write requests being made. *iop* = 5 deletes any existing file of the same name if necessary before creating.

The call to create the work disc (*idev* = 0) should, if possible, create a file that it is automatically deleted on exit from Semper. Semper calls `mc61` to deassign and delete the work disc when it terminates normally, but this will not happen if an error causes Semper to be aborted. If automatic deletion cannot be arranged, the file should be created as normal. If the worst happens, it can always be deleted manually outside Semper. The filename is generated by Semper and will start with the letters 'ZZ'.

In caching installations, Semper makes read and write calls only with *num* equal to the cache buffer size, and with *n* aligned with a cache buffer boundary; file creation in such installations includes any initialisation necessary to allow Semper to *read* the file subsequently, as Semper's first operation on any cache buffer is a read rather than a write.

A typical pattern of access during a Semper session is an initial opcode 0 call followed by several opcode 3, 4 or 5 calls, followed by a more or less arbitrary sequence of opcode 1 and 2 calls (except that no part of a file is written before it has been read), followed by final opcode 6 and 7 calls.

1.5 Tape access

Semper's tape handling reflects the essential differences between tapes and disc; these include the variability of block sizes, the special *tape marks* used as end-of-file markers, and the serial nature of the medium (random access, while possible, is slow; and writing at a given place renders all material further down the tape inaccessible). The primitive routine is required to function at a low level, close to that at which system services commonly operate.

Tapes are identified by device numbers in the same way as for discs except that device numbers 0 and 1 are never assigned to a tape device. The device number assigned, at any given time, to a tape device will not be assigned to a disc device, and vice versa.

MCTP61 CALL MCTP61(iop,idev,n*4,ibuff,ierror)

performs data transfers, block and file skipping, end-of-file mark generation or detection, and tape mounting and dismounting, according to the value of iop:

- 1: positions tape device idev to the *load point* (beginning of tape); the opcode may be requested when the tape is already positioned there
- 2: reads a block from device idev, transferring up to n bytes to array ibuff, and resetting n to the number actually transferred if this is less; sets ierror to 2 if the 'block' proves to be an end-of-file mark
- 3: writes a block of n bytes from array ibuff to device idev
- 4: reserved (unused at present)
- 5: writes an end-of-file mark to device idev
- 6: skips forwards one block, setting ierror to 2 if the 'block' proves to be an end-of-file mark
- 7: skips backwards one block; the opcode is not requested at the beginning of a file
- 8: skips forwards one file, stopping beyond the terminating end-of-file mark
- 9: skips backwards one file, stopping beyond the end-of-file mark terminating the preceding file; the opcode is not requested within the first file on a tape
- 10: assigns tape as device idev for read/write access, mounting physically if necessary
- 11: assigns tape as device idev for read-only access, mounting physically if necessary
- 12: deassigns device idev, dismounting physically if n is zero

In the event of an input/output error of any kind, mctp61 sets ierror to 1 and if there is any assign or deassign failure, mctp61 sets ierror to 44. Otherwise it returns ierror unchanged.

For iop = 10 or 11, the file name is supplied as i/c chars in ibuff(2,3..), with the number of characters in the name in ibuff(1). Ideally, iop = 11 mounts the tape with the write-enable ring removed; however, actual read-only enforcement is not in fact required, as Semper remembers the opening mode and prevents any write

requests being made.

If the tape primitive cannot be provided with the above functionality in a given installation, the simplest alternative is to provide a 'dummy' routine which simply returns without doing anything except for setting `ierror` to 44 on any assign request (`iop` = 10 or 11); this results in a Semper with no direct tape access, and requires all use of tapes to be made via whatever utilities are available for copying disc files to or from tape.

Where it is only backward tape movement that cannot be provided, Semper can still make a limited use of tapes by rewinding and skipping forward as necessary; in such installations, no requests for opcodes 7 and 9 are made. If rewinding is also impossible, `mctp61` sets `ierror` to 2 before returning when opcode 1 is requested; tape access in such cases is limited to copying pictures to or from tapes in serial order.

Under operating systems which strive too hard to make tapes appear exactly like a succession of disc files, the ability to move freely between files may be difficult to implement; the effect can sometimes be achieved indirectly by performing a close operation on one file followed immediately by an open with a different file name or sequence number.

A typical pattern of access for a given tape during a Semper session is an initial opcode 10 or 11 call, followed by an opcode 1 call to establish an initial position, and then a more or less arbitrary sequence of opcode 1 to 9 calls, followed by a final opcode 12 call.

1.6 Framestore access

Semper 6 expects a relatively complicated interface with a display device, conceiving it primarily to be of the framestore type. Partial implementation of the specification below is also useful, and lead to an installation with corresponding partial functionality.

The framestore *image* memory is considered to be organised as one or more separate *frames* of the same size, (not necessarily square) numbered 1,2.. and addressed individually by an (x,y) coordinate pair in which x counts pixels rightwards across the frame, y counts pixels downwards, and 0,0 is the top left. Typically, each frame is 8 bits deep, allowing 256 distinct grey levels; more can be exploited however, and less can be accommodated. Each image memory pixel is also considered to have a corresponding one-bit *overlay* pixel; the overlay is used for superposing text, graphics etc. on images without corrupting the image data. If a framestore does not have an overlay plane, it is often possible to simulate one by using the either the top or the bottom bit of each pixel as the overlay bit. If this approach is not suitable, then any output that would normally go to the overlay will have to overwrite the corresponding parts of the stored image.

You have to make decisions about the division of your framestore memory into frames. Semper allows users to subdivide frames conveniently at run time into

smaller display *partitions* that can be used independently for picture storage, so in many cases the division is largely artificial and probably undesirable. With a typical simple framestore device providing a 512 point square memory, there is no reason to do other than treat it as a single frame of this size; a single 1024 square frame is best treated as just that rather than as four independent 512 square frames, as the latter effect can be achieved by partitioning the 1024 frame when needed without prejudicing the ability to store a single image of 1024 pixels square as well.

Frames need to reflect hardware continuity of pixels in two respects: (i) transfers should be possible between host computer memory and rows and columns of pixels extending anywhere across a single frame ; and (ii) the video signal generator should present the complete frame or a subregion of it on a monitor screen. This may prevent your treating several physically distinct frames as a single larger frame for Semper purposes.

The final factor affecting the decision on frame organisation is *full colour* picture generation, if the hardware has this capability. In the full colour mode (as opposed to *false colour* mode), a series of three pictures, representing the red, green and blue components of an image, must be fed to the three colour inputs of the monitor; Semper 6 chooses to place these pictures in three successively numbered frames (or in corresponding subregions of them). It is necessary therefore to choose an organisation in which whatever different parts of the framestore memory can be combined in a full colour display are presented to Semper as three successively numbered frames, in the order red, green, blue.

A considerable range of different hardware can be coerced to serve as the Semper display device, mimicking the framestore conception to a greater or lesser extent. Where a *binary* device, offering black and white dots only, is used, grey levels can be simulated within the primitives by allocating a small square block of dots (e.g. 4 square) and setting variable numbers of these according to the brightness required; such blocks are called *grey pixels* below, and the number of addressable pixels along the side of a block is called the *grey pixel size*.

It may help you assimilate the specifications below to know that a typical pattern of framestore access during a Semper session consists of an initial `fsas61` call to assign the device, followed by an `fsvw61` call to establish viewing conditions; then an arbitrary sequence of `fser61`, `fsro61`, `fsri61`, `fsln61`, `fstx61`, `fsvw61`, `fslu61` and `fsxw61` calls, with an `fsfl61(1)` call whenever it is important that the monitor screen is completely up-to-date and an `fsfl61(2)` call whenever (if ever) a user's close command releases an entire display for output to hard-copy hardware; and a final `fsde61` call.

All framestore primitives are coded as logical functions; they return `.false.` normally and report any kind of error condition by setting `ierror` to 40 and returning `.true..`

1.6.1 Assignment, deassignment and data buffer control

FSAS61 LOGICAL=FSAS61(nfrs,nfx,nfy,nmx,nmy,nchx,nchy,ngp,iwo,ierror)
assigns the framestore, establishing its configuration, and performs any device initialisation necessary. A particular configuration for the framestore may be requested with non-zero arguments as follows:

nfrs - number of frames

nfx, nfy - horizontal, vertical frame dimensions, in pixels

nmx,nmy - horizontal, vertical monitor signal dimensions, in pixels (the number of pixels in each line of, and the number of lines in, the video signal)

nchx, nchy - horizontal, vertical dimensions of character block, in pixels (see below)

ngp - grey-pixel block size (see above)

iwo - write-only flag (write-only if non-zero)

Where any of these parameters can be adjusted at run time, fsas61 takes the necessary action. It should, in any event, return in all of the arguments values that are applicable to subsequent calls to the framestore primitives until a fsde61 is called to deassign the framestore.

Any initialisation performed by fsas61 need not include establishing monitor viewing conditions (frame, zoom, scroll, look-up table), as these are established by later calls to fsvw61; and it should *not* erase any framestore memory, as this is a convenient route for passing pictures into Semper from other programs. When initialising a framestore that may be shared between several Semper users, fsas61 must not alter the state of the display, as this could interfere with other users already using the framestore. It follows that such a framestore must not be reconfigured by fsas61. On a workstation, a window can be used to simulate a framestore. In this case, fsas61 should try to create a window with a size given by nfx and nfy, adjusting the size if necessary to ensure that the window fits onto the screen.

FSDE61 LOGICAL=FSDE61(ierror)

deassigns the display device. On a workstation where a window was created to simulate a framestore, the window should be destroyed, e.g. assigning one of several framestores attached to the system.

All fsas61 and fsde61 calls are made from one Semper module, assfs, which can be easily adapted to accommodate further variations in the assignment or initialisation process to suit particular local requirements.

FSFL61 LOGICAL=FSFL61(*iop,ierror*)

flushes any output display device buffer in one of two ways according to the value of the opcode *iop* and the type of display hardware in use:

- iop*=1: is relevant to installations with framestore hardware which make physical transfers to the framestore only when a relatively large main memory buffer becomes full; *fsfl61* flushes any such buffer so that any pending material becomes visible on the monitor
- 2: is relevant to installations with a hard-copy device (e.g. a laser printer), where the entire display is retained in main memory or in some intermediate file until explicitly released by the user; *fsfl61* despatches any such buffered display to the hard-copy device

In each environment, *fsfl61* ignores requests for the other opcode.

1.6.2 Memory access

References to memory positions off the display frames are possible, given the user's freedom to write code of his own with direct calls to the primitives. Therefore the framstore primitive routines should clip all information that is passed through. Standard Semper code truncates pictures, lines and text as necessary.

FSER61 LOGICAL=FSER61(*iop,nx,ny,ix,iy,ifr,ierror*)

erases (resets to zero / black) a region of frame *ifr*, *nx* pixels across by *ny* down, with the top left at *ix,iy*, according to *iop*:

- iop* = 1: erases image memory only
- 2: erases overlay memory only
- 3: erases both memories

FSRO61 LOGICAL=FSRO61(*row,n1,n2,isam,iform,ix,iy,ifr,black,white,icol,ierror*)

outputs pixels *n1* to *n2*, in steps of *isam*, from array *row* as a picture row rightwards across image frame *ifr* beginning at *ix,iy*; accepts pixels in *row* in any of the four Semper forms according to the form number *iform*. The values should be scaled so that a value equal to black maps onto the minimum pixel value (zero) and a value equal to white maps onto the maximum pixel value, allowing for the possibility black > white (reversed contrast). Any values outside the range black to white should be set to the corresponding extreme pixel value. This scaling facility means that Semper does not need to know the range of values actually supported within the framestore image memory.

If *icol* is non-zero, *fsro61* outputs the pixels as a picture column downwards from *ix,iy*. This opcode is not currently used but it will be used in the next version of Semper.

FSRI61 LOGICAL=FSRI61(*row,n,iform,ix,iy,ifr,black,white,icol,ierror*)

recovers a picture row of *n* pixels rightwards across image frame *ifr* beginning at *ix,iy* to array *row*. Returns pixel values in any of the four Semper forms according to the value of *iform*. The pixel values should be scaled so that the minimum pixel value maps to black and the maximum pixel value maps to white, allowing for the possibility that black > white.

If *icol* is non-zero, the pixels are accessed downwards from the position *ix,iy*. This opcode is not currently used but it will be used in the next version of Semper.

FSLN61 LOGICAL=FSLN61(*ix2,iy2,ix1,iy1,ifr,ierror*)

draws line from *ix1,iy1* to *ix2,iy2* of overlay frame *ifr*; the direction matters only when hardware is of the pen plotter type.

FSTX61 LOGICAL=FSTX61(*ibuf,n,ix,iy,ifr,ierror*)

writes a string of text, defined by the *n* i/c characters in array *ibuf*, to overlay frame *ifr*, centred at *ix,iy*.

Semper expects characters to be generated with a fixed (i.e. non-proportional) font; each character is allocated a block of pixels which allows one blank column at the right and one blank row at the bottom, and the blanks are included in the dimensions reported by *fsas61* (see above). A string is assembled by abutting these character blocks. The position in the string to align with *ix,iy* is the string centre rounded to the nearest pixel to the right and down.

1.6.3 Viewing conditions

Semper provides for simple integer zoom and scroll control of its framestore; and also for relatively sophisticated control of the output look-up tables (*luts*) and colour presentation modes. No standard interface is however defined at present for controlling how the overlay memory is presented; this is generally assumed to be visible on the monitor at all times in some form or other, and you may need to combine overlay presentation control with your management of *luts*, which are assumed by Semper to be applied only for displaying the contents of the image memory. There is no facility provided in Semper for managing input *luts*.

Semper envisages in general the ability to hold several *luts* in the hardware, to write or read any of these, and to select any one of them for *active* use in controlling the monitor signal generated, independently of which frame is being viewed (i.e. *luts* are *not* associated with particular display frames).

A *mode* (value 1, 2 or 3) is associated with each *lut*, which may be monochrome, false colour or full colour; in the monochrome mode, a single intensity transformation table is used to map pixel values from a single image frame to all three monitor colours (or a single monochrome monitor input); in the false colour mode, the *lut* consists of three separate tables which are used to map pixels from a single image frame to the three monitor colours; and in the full colour mode, the same three tables are used to map pixels from three distinct (but consecutively

numbered) frames to the three monitor colours. In making a given lut active, the corresponding mode must also be correctly established.

A given table consists of an array of integers with a length determined by the number of possible pixel values supported within an image memory frame, giving replacement intensity values, in the range 0 to whatever local maximum is appropriate, for each stored pixel value. You must set the parameter constants lutlen and lutmax in the file icset to specify the lut size and maximum lut value.

Luts are numbered from 1 up to a maximum number you chose. This will usually exceed the number of luts the hardware supports. The parameter constant lutnum in the file icset determines the maximum number of luts that Semper can maintain. fsvw61 and fslu61 manage the details of which luts are actually held in the hardware. fslu61 is called whenever a change is made to one of the luts. If the lut is currently held in the hardware, fslu61 can overwrite its contents with the new data. A different lut is activated with a call to fsvw61. If the lut is already stored in the hardware, fsvw61 can activate it more rapidly than a lut that must be loaded from the host. For a simple framestore with a single lut, only the active lut can be held in the hardware. New data will have to be loaded from the host when fsvw61 activates a different lut, and when fslu61 is called to change the active lut. For a more sophisticated framestore with n luts, you might maintain copies of the luts numbered from 1 to n-1 in the hardware and use the remaining space to hold the last activated lut numbered from n upwards.

FSVW61 LOGICAL=FSVW61(lutn,izoom,iblack,nx,ny,ix,iy,mx,my,ifr,ierror) establishes monitor viewing conditions that present a region of frame ifr (or frames ifr, ifr+1, ifr+2 in the full colour mode) at zoom factor izoom, with pixel mx,my at the centre of the monitor screen; if iblack is non-zero, the screen is blanked outside the nx,ny pixels whose top left is (frame position) ix,iy. The present version does not call fsvw61 with a non-zero value for iblack. Requests for unsupported functions are simply ignored.

If the hardware is incapable of blanking outside arbitrary regions, as described above, there is little loss of functionality in simply ignoring this feature. Blanking outside the memory frame however should be implemented if possible, so as to avoid displaying spurious data when the monitor viewing window is scrolled beyond the limits of the memory frame. If blanking is not possible, an alternative is to limit scrolling to keep the monitor window always within the memory frame.

The lut number lutn is also activated by fsvw61, together with the associated mode. No action is necessary of course if this lut is already active; if the lut is inactive but held within the hardware, fsvw61 takes whatever action is necessary to activate it. If it is not held within the hardware either, then fsvw61 asks Semper to supply the lut data and mode as follows:

```
INTEGER LUTBUF(enough)
...
IF (SEMLUT (1, LUTN, MODE, LUTBUF)) THEN
  FSVW61=.TRUE.
  RETURN
ENDIF
```

Note that ierror is *not* set to 40 in this case. The data is then transferred to the hardware by fsvw61 and activated (perhaps with a call to fslu61). False and full colour luts are passed in lutbuf as three consecutive sets of values in the order red, green, blue.

FSLU61 LOGICAL=FSLU61(iop,lutn,mode,lutbuf,ierror)
transfers data to or from hardware luts according to the opcode requested:

iop = 1: reads lut from hardware lut number lutn to array lutbuf, returning in mode the corresponding lut mode. When it is not possible to obtain the lut mode from the hardware, the user has the option to specify a mode on the command line. This is passed to fslu61 in mode as a non-zero value and this mode should be assumed for the purposes of reading the lut data from the hardware. If, however, mode is set to zero, a fault should be reported. A fault is also reported if the requested lut is not buffered in the hardware.

iop = 2: copies lut data from array lutbuf to the hardware. mode specifies the lut mode. If lut lutn is not currently held in the hardware, fslu61 does nothing.

Fslu61 is called whenever the contents of a lut change.

FSXW61 LOGICAL=FSXW61(ix,iy,ierror)
activates a display cursor, with an initial position ix,iy (on whichever frame is currently being viewed; see below), allows the user to move the cursor with a mouse, trackball or terminal keys, and returns in ix,iy the coordinates of the position finally selected (by a mouse 'read' button, a terminal carriage return etc. as you choose). The cursor should preferably be displayed in such a way as not to destroy any of the information stored in the framestore, e.g. by displaying it in a separate overlay.

1.7 Miscellaneous

Semper users can normally issue some form of *abandon request*, such as striking a control-C at the terminal; this allows them to interrupt indefinite loops, abandon lengthy procedures or correct mistakes as soon as they are noticed. A primitive is required to sense these requests.

ABANDN LOGICAL=ABANDN(*ierror*)

returns .true., and sets *ierror* to 4, if an abandon request has been made since the last time *abandn* was called, emptying any terminal type-ahead buffer if possible. Otherwise *abandn* returns .false. without changing *ierror*. An abandon request is any special keystroke or break signal which can be trapped by the operating system and detected by *abandn*. Semper calls *abandn* at regular intervals, for example, whenever a picture row is accessed, and always after any call to *inkey*.

MCTIME CALL MCTIME(*ibuf*)

sets the seven elements of array *ibuf* to the current year (e.g. 1986), month (1-12), day (1-31), hour (0-23), min (0-59) sec (0-59), and centisec (0-99), supplying zero for any information not available.

WAITS CALL WAITS(*secs*)

suspends or otherwise delays Semper for *secs* seconds of real (not central processor) time; limited or approximate functionality is still useful.

Three routines for bit manipulation are defined in a way commonly provided by Fortran compiler extensions:

IBTNOT INTEGER=IBTNOT(*i*)

returns the bit-wise not of the argument.

IBTOR INTEGER=IBTOR(*i1,i2*)

returns the bit-wise or of the two arguments.

IBTAND INTEGER=IBTAND(*i1,i2*)

returns the bit-wise and of the two arguments.

1.8 Primitive testing programs

A set of (relatively) simple test programs is provided for verifying the behaviour of most of the primitive routines; these cannot of course test their functionality exhaustively, and may in any case have to be adapted slightly to accommodate significant departures from the full interface defined above. They are for most purposes however very helpful while you are writing the primitives, and examining their source may also help clarify points that are not explained clearly above.

2. Adjustments to the Fortran code

Three things must be considered before you actually compile the main bulk of the system source code. The first is the values assigned to a set of parameter constants that control maximum array sizes, inform Semper about the details of your particular environment etc.. These are defined in a document called `params`, which you should examine carefully. Within the section bracketed by comments of the form:

```
C ***** CHANGE *****  
...  
C ***** ***** *****
```

many of the crucial system configuration details are established, and you will need to alter many of the values assigned here in the light of your own requirements and capabilities. This document is included in the source of almost all Semper modules compiled, via an include statement. Since this statement itself is not in fact provided for within the Fortran 77 Standard, you may need to adjust its syntax, or at worst insert the text in full wherever necessary. As supplied, three include files are involved: `common`, which includes `params`, which includes `icset`. Most Semper modules include `common`, but a few only include one of the two shorter files.

The second matter is the interpretation of `integer` and `integer*4` declarations, discussed in section 1.2 above. If the latter are unacceptable to your compiler, you will have to edit all the `integer*4` declarations back to `integer`, and accept slightly restricted functionality in a few respects. The standard code, incidentally, avoids the difficulties sometimes arising over the interpretation of constant actual arguments in subroutine or function calls (are they `integer*2` or `integer*4`?), by using `integer*4` parameter constants where necessary.

Thirdly, you must examine all of the source code with an editor, searching for sections bracketed by `change` comments of the form given above; these are called `change` sections, and draw attention to points which may require action to accommodate your particular environment. Appropriate instructions are given in comments within each such section.

These matters attended to, you can compile all the supplied code (except for the independent system generation and help library management modules) into an object code library for subsequent linking with a main program.

3. System Generation

No main program source is provided with Semper, as this is machine generated at the time of installation (and subsequently, whenever additional commands are added to the system by local users), together with a subroutine `semdvs` which it calls, by the system generation program `semgen`. This reads a supplied *verb descriptor* document `semper.vds` on one Fortran unit, and writes the main program and `semdvs` to another. Accordingly, you must now compile and link `semgen` itself; its source contains some `change` sections, which you should consider in the usual way; it calls the primitive routine `alconv`, but is otherwise entirely self-contained. If possible, store `semgen` in a linked form ready for immediate use whenever necessary later.

Normal termination of `semgen` is confirmed by a terminal message *System generation complete*, together with some statistics on the processing commands defined. The programs output should be compiled and linked together with the other modules in the object library (and the standard Fortran run time support library, of course) to complete the process of generating a working Semper load module. The System Users' Guide gives general information on the various Fortran input/output units used by a Semper session, which you may like to consider when establishing this load module; session initialisation code for Semper is localised in the subroutine `semini`.

4. Help Library Management

A second independent utility `helpman` provides a utility for generating and managing *help libraries* in the standard form supported by Semper. This reads supplied (or locally generated) files of on-line *help* text interleaved with keyword directives, and creates or alters a disc file containing the information in a suitably indexed and cross-referenced form. You must compile and link `helpman` now, examining its source for change sections in the usual way; if possible, store it linked ready for immediate use whenever necessary later.

You use `helpman` interactively in the first instance; it prompts at the terminal for the name of the help library (disc file) to be used, and if this does not already exist, suggests that you use the `initialise` command to create it - typing `help` produces a list of all the commands recognised, and a brief explanation of their function. `Initialise` requires you to guess the size of file you will need: don't worry about guessing badly, as you can easily repeat the process later in the light of experience.

Initially you will want to type a command such as `run semper`, which diverts terminal input temporarily to the file `semper.shl`, and loads the information from it into the help library. Type `?` (or `type full ?`) lists all entry points currently defined; `type name` (or `type full name`) types the text stored for entry `name` etc.; `status` types information on the amount of space used, free etc.

Later, you may try the `log` command, which outputs a given entry in text form to a *log file* attached to a `helpman` session; this allows you later to edit the text to a file which you then make the subject of a `run` command so as to reload the revised text into the library. The `delete` command is used to get rid of entries you do not want; old versions of an entry are implicitly deleted whenever you reload them via `run` (or enter them directly via `add`). Space is not reclaimed immediately when entries are deleted, and from time to time you need to use the `compress` command to recover this.

You can choose which of the supplied help files to include; you may add additional entries of your own (an entry installation is particularly recommended to note details peculiar to your own environment); and you may delete entries selectively. You might like to create more than one help library, so that supplementary material is not automatically made available to all Semper users, and so on.