Semper 6 *Plus*

Fortran Programmers' Guide

How to write new modules for Semper

System module specifications

Contents

Introduction: the scope of this manual

Semper is not intended to be used as a library of subroutines which you call from you own Fortran main program; although it can be used in this way, the process is not straight forward, and you are not encouraged to attempt it by any information in this manual. However, its structure *is* intended to let can write new processing ← modules (Fortran subroutines) yourself, provided you follow system protocol in various respects, and incorporate them quite easily in the system as new Semper verbs with their own function and syntax. This manual describes how to do this.

Details are given of picture data formats and access methods; of how other information is held inside the system; of how interpreter services for the new verb are requested; and of how the system is regenerated with the new facility included. A complete example of a new facility is built up in the course of the manual. You are assumed to be familiar with the Fortran '77 language.

> Material that can be omitted on a first reading is printed at the right hand side in shorter lines, like this.

Before you embark on writing your own Fortran, you should become familiar with how your local installation is organised; the file names mentioned in this manual will not usually correspond exactly to those your installation uses, though they will probably be similar, and you may need to search a little or ask for advice. Find the *object library* in which the compiled code of all Semper's modules are held, and how to compile modules into it; find the files common.for, params.for and icset.for included in the source of most modules; find the *linking* command(s) needed to link the *load module*; find the *Verb Descriptors* file semper.vds that specifies each command to the Semper command interpreter, and the *system generation program* semgen that generates from this the Semper main program ; and find out how to run semgen. You may also find it useful to be able to refer to the system error message file semper.err, which contains the messages printed in response to all the standard error codes, though Semper prints this on the terminal in response to the command show errors.

Probably the first changes you will want to make to Semper will not in fact involve additional Fortran anyway, but merely minor adjustments to existing code, installation options or run-time assignments. In a multi-user environment, you will probably be able to create your own version of Semper, with your own particular additional commands; otherwise, you may need to gain access to the appropriate directories and proceed with appropriate caution, backing up files before you begin to make alterations.

There are several reasons why you may eventually wish to write new Fortran modules:

(i) to make faster an operation which, although possible by combining several existing commands, is undesirably slow when so implemented;

(ii) to exchange pictures with some other local filing, processing or display system;

(iii) to make use of some special local input or output peripheral, or some special hardware such as an array processor;

(iv) to implement some totally new procedure, impossible to express in terms of the existing Semper commands;

(v) to integrate with Semper other code or packages you may want to use

(and so on). Be sure that you have considered the alternatives fully before you set out, however: with a little ingenuity, much can be accomplished within the constraints of the existing language, and even if the result would be a less efficient program than a new Fortran module would be, it might still be a sensible use of your time.

Besides the module that is constructed in the course of your reading through this manual, a simple but complete (and heavily commented) prototype module user is provided with Semper, already interfaced with the interpreter and invoked by the command user. This makes a single row by row pass through a source picture, calculating its absolute value and finding the centre-of-mass; two options and two keys are defined with no particular function. You may find copying and adapting this as good a way of learning how to write new modules for Semper as reading through this manual - in the first instance at least. In the long run however, there is no substitute for reading this carefully from beginning to end.

Chapter 1. Basic module construction

1.1 General remarks on picture storage/access

The principal factor to consider when designing or reorganising a module for use with Semper is the fact that the whole picture is not usually accessible at once; the rows that make up the picture are each stored separately, whether on disc, display or tape, and have to be fetched into memory in turn as they are required. The reason for this is that many computers still cannot guarantee enough main memory to hold large pictures, particularly in floating point form - one 1024 point square floating point picture alone requires 4MB of storage.

Semper is in fact normally installed so as to buffer relatively large amounts of data in a *disc cache*; the buffer area concerned is not however available directly to processing modules, and the caching, while greatly improving system performance, does not affect the way in which processing modules are conceived and written. Modules operate in the first instance on up to six 1-D (singly subscripted) arrays or *row buffers*, and call Semper system modules to read or write picture rows to or from these arrays.

This method of access raises no problems for point-by-point operations: one row buffer only is required, and two nested loops are set up, the outer one fetching and storing picture rows in turn, while the inner one resets the points of a given row. However, for more complicated operations such as smoothing or differentiating, each processed picture element depends on the original values at neighbouring elements in all directions, and more thought must usually be given to how many row buffers are required for reasonable efficiency, and how they are to be used.

> If you do have sufficient memory available in your environment, there is nothing to prevent your setting up a local 2-D or 3-D array large enough to store the entire picture, fetching the entire picture into this, processing it in situ and storing it all on disc again when you have finished; this evades any serious buffer problems, but of course the result will not necessarily be transportable to other Semper installations.

Your new module will have access to six row buffers provided in a common block; you can of course declare additional row buffers locally if you wish.

1.2 A sample problem to address

As a framework for introducing the system facilities and details, let us consider the particular problem of obtaining the intensity gradient (derivative, or differential coefficient) of a picture in an arbitrary direction - say, θ anticlockwise from the positive x axis. This can be approximated by adding the nearest neighbour differences horizontally and vertically, multiplied by factors $cos(\theta)$ and $sin(\theta)$ respectively:

$$p'(x,y) = cos(\theta).(p(x+1,y)-p(x,y)) + sin(\theta).(p(x,y+1)-p(x,y))$$
$$= c.p(x+1,y) + s.p(x,y+1) - (s+c).p(x,y)$$

writing c for $cos(\theta)$ and s for $sin(\theta)$. We shall develop a module to effect this step by step, and incorporate it in Semper as a new command GRADIENT; you can check its effect against that of differentiate angle theta.., which in fact carries out the same operation.

1.3 Access to picture points

To begin with, consider a short section of code that calculates the above expression for a single picture row. Suppose that the array (row buffer) rb1 contained the ncol fp (floating point, or real) values making up a given picture row, that array rb2 contained the row above it, and that c, s and cps contained $cos(\theta)$, $sin(\theta)$ and c+s respectively, you could calculate the corresponding output pixels in rb2 as follows:

```
      RB1(NCOL+1)=RB1(NCOL)
      DO 10 I=1,NCOL
         RB2(I) = C*RB1(I+1)+S*RB2(I)-CPS*RB1(I)
   10 CONTINUE
```

(pixels are numbered 1-ncol left to right). You may notice that the problem of dealing with the last pixel in the row, which has no right hand neighbour, is dealt with by *propagating* that pixel outwards beyond the row; the standard row buffers are set up with a number of spare reals at each end to permit this kind of edge processing.

Although the above code operates directly on fp data only, it will in fact serve to process byte, integer and complex pictures also, because Semper in fact includes any necessary form conversions with any row reading or writing operation. More details of this facility are given later.

> To process integer pixels directly in integer form, you should declare integer row buffers, e.g. ib1, ib2.. equivalenced to rb1, rb2.. (suitable arrays are declared, though not used, in user) and operate with these. Wherever possible, Semper will have been installed with this defining a 16 bit datatype, to minimise storage requirements for integer pictures; most integer arguments to Semper modules accordingly share the range limitations of this datatype, though a small number are explicitly declared integer*4 in the standard source. (In some Fortran systems, the *4 declaration may be illegal, in which case they will have been replaced by simple integer declarations when the system was installed; in others, no 16 bit datatype may be available, in which case both integer and integer*4 variables (and form integer pictures) will be 32 bits long. You can find out your own position by examining the values assigned in PARAMS to the parameters lnint and lnint4, which record the length in bytes of the standard integer and the integer*4 datatypes respectively.)

Byte data are not of course provided within the Fortran '77 standard, and you will not be able to process byte pixels directly in every installation. However, if your Fortran system does provide an unsigned integer*1 datatype, there is nothing to prevent your declaring byte row buffers bb1, bb2.. similarly equivalenced to rb1, rb2.. You should note that DEC's VAX/VMS byte datatype, although of the correct length, has a different range (-128 to 127) from that of Semper's byte pixels, and is not suitable for use in this way without great care.

The handling of complex pixels does not depend on Fortran's complex datatype; the real and imaginary parts of a pixel are simply stored as two successive fp values in the array, so that rb1(1) contains the real part of the first point in the row, rb1(6) the imaginary part of the third, etc. Once again however there is nothing to prevent your declaring equivalenced complex arrays cb1, cb2.. if the local representation conforms to this pattern (as is usually the case).

1.4 Buffer strategy

The above process needs to be repeated for each picture row, of course; as you move forward (down the picture, in fact) you could simply refill both arrays rb1 and rb2 with the new rows required, and repeat the existing code; however, you could avoid unnecessary re-reading, making the final module execute faster, by filling rb2 only (free once its contents have been dumped) with the next row, and leaving rb1 (containing what will then be the upper of the two new rows) unchanged. In the cycle after that, you would once again read the lower row to rb1.

You would then need to adjust the code set out above so as to interchange rb1 and rb2 in successive cycles. In fact, the six standard row buffers rb1-6 are consecutively placed in common, so that they can all be addressed as rb1 suitably subscripted (out of the declared range). If i1 and i2 point to two different row buffers in this way, you could achieve the same as the code above with the lines:

```
      I3=I1+NCOL-1
      I4=I2
      DO 10 I=I1,I3
         RB1(I4)=C*RB1(I+1)+S*RB1(I4)-CPS*RB1(I)
         I4=I4+1
   10 CONTINUE
```

In some installations, the row buffers may be too long for switches to be made between them in this way by a subscript of the default integer datatype (though there is never any problem for *single* row). In most VAX installations, for example, the default integer datatype is 16bits long and the row buffers are 32,786B long, requiring subscripts up to 16,384 for individual buffers and larger, out-of-range, values to switch between them. In such cases, you can usually accommodate the problem by declaring the few variables

actually involved in buffer array subscripting to be integer*4.

1.5 Row access

The next step to consider is the fetching of picture rows into the row buffers. You ask the system module semrow to do this for you:

```
SEMROW (1, BUFFER, NFMFP, IROW, 1, LP1)
```

causes the array buffer to be filled with row irow of *logical* picture lp1. Rows are numbered 1,2.. from top to bottom, in the opposite direction to the Y coordinate) of the picture. Logical picture numbers (*LPN*s) identify different pictures amongst those currently open - more details are given later. Semrow 'knows' how large the picture rows are, where and on what device they are stored, and so on, so that we do not need to consider such details at this stage.

Semrow is in fact a logical function: it returns .FALSE. if successful, but .TRUE. in the event of any trouble (e.g., an out-of-range value for irow, or a disc i/o error). In the event of a .TRUE. return, you abandon processing at once and allow the system to tidy up: this is normally achieved by coding

```
IF (SEMROW(...)) RETURN
```

for all row access. To prepare the data for the row differentiation code at the end of the last section, you can now code

```
IF (SEMROW(1,RB1(I1),NFMFP,J,1,LP1)) RETURN
```

so as to fetch row number j into the buffer pointed to by i1; row j-1 will have been left in the other buffer by the previous cycle of the loop, and the initialisation of the process is considered in the next section.

Following the differentiation loop, you must store the row at i2 as row j of the output picture. You do this by calling semrow with a first argument (*opcode*) of 2 rather than 1; also, since the output will in general go to a different picture from the source, we use a different LPN:

```
IF (SEMROW(2,RB1(I2),NFMFP,J,1,LP2)) RETURN
```

Write calls to semrow also return .TRUE. if the output picture is write protected, so you need take no special action to honour this.

The third argument to semrow indicates the form of the data in the row buffers: the standard form numbers apply, but use of parameters such as nfmfp instead (see §1.7) is preferable. Semrow converts to this form if necessary on reading, and from it if necessary on writing. Semrow may in fact re-form the data in the row buffer itself when writing; this only affects you if you are writing the same data to many rows in turn, and you 'work round' the problem when necessary by using the primitive cform (see the Implementation

Guide) to convert the form appropriately yourself before calling semrow.

There is nothing to prevent you from reading rows back from lp2 for further processing, should you wish to do so, nor from writing fresh data to lp1; you write modules in general assuming complete freedom to mix reading and writing for any picture, and rely on semrow to fault any transfers which are not in fact possible. Complete freedom is in fact achieved at run time only if the pictures are on disc; semrow makes an error return if the module attempts to read back rows from a picture on a display device without read-back capability, or to write rows to a picture which is in fact on tape.

Semper only allows its logical disc blocks to be read or written as a whole; buffers used to hold rows in memory must therefore be large enough to take the row data rounded up to a whole number of disc blocks. You need not worry about the six standard buffers however: the length of these is always a multiple of the disc block size, and Semper refuses to create any pictures whose rows will not fit them.

1.5.1 Note on data storage

On disc, pictures are stored in a contiguous series of blocks; a picture begins with a 256-byte block-bounded label, followed by the layers in turn from back to front, within each of which the rows are stored from top to bottom; all rows are block-bounded memory images. On tape, the label and then the rows (also memory images, similarly ordered) are each stored as single blocks; a single end-of-file mark (tapemark) terminates the picture, and a second consecutive such mark terminates the whole tape. On the display, rows are in general stored scaled, undersampled where necessary and form-converted in a way that fits the local display hardware (the scaling and conversion being reversed during data recovery where this is possible); the rows of a given layer are stored one beneath the other, obviously, and the layers of colour pictures are stored in consecutively numbered frames; labels for display pictures are held in the work disc.

Most of the information about a picture is held in the picture label. The position of a given picture within a disc file is recorded in a directory held at the beginning of the file; no such record is kept for tape pictures, which are located as needed by scanning the tape; and details of display partitions are held in the work disc.

1.6 Looping and initialisation

Now that the heart of the new module is complete, you add some further code that repeats it for all picture rows, initialises and maintains the pointers i1 and i2, and performs initial buffer filling (in processing the top row, which has no neighbour above it, follow the pattern adopted at the right hand side of the rows and provide an extra copy of the top row); and you give it a name:

```
      SUBROUTINE DERIV
      ...
C Initialise row loop: pre-fill first buffer
      IF (SEMROW(1,RB1,NFMFP,J,1,LP1)) RETURN
C Set buffer pointers
      I1=1+LNBUF/LNREAL+2*LNEDGE
      I2=1
      DO 20 J=1,NROW
C Fetch source row
        IF (SEMROW(1,RB1(I1),NFMFP,J,1,LP1)) RETURN
C Pass along row obtaining derivative pixels
        ...
      original code
        ...
C Store output row
        IF (SEMROW(2,RB1(I2),NFMFP,J,1,LP2)) RETURN
C Swap buffer pointers
        I=I1
        I1=I2
        I2=I
   20 CONTINUE
      RETURN
      END
```

1.7 Common variables and parameters

Besides lp1 and lp2, you are now using lncomp, lnreal, lnedge, nrow, ncol and nfmfp without yet knowing where they come from. lp1 and lp2 are in fact variables in common which Semper sets suitably for you before it calls your new module; lnbuf, lnreal, lnedge and nfmfp are parameters set at compile time (giving the length of the row buffers and of the real datatype in bytes, the number of reals allocated for edge processing at either end of each row buffer, and the data type number for reals). A standard set of common block declarations and parameter settings is provided with Semper, usually in a file common.for and you should include these in all new modules, via a statement

```
      INCLUDE 'COMMON'
```

where this or a similar facility is supported by the local Fortran system, or directly otherwise.

> Common is in fact usually sub-divided, with two additional files containing parameter declarations (params.for and icset.for) being included as well by means of nested include statements.

In time, you may like to read the source of the common block and parameter declarations, so as to learn what information is available to you from them; many of the items declared figure later in this manual - the next section explains how ncol and nrow are set from common, for example - and others are explained in the source comments.

The first of the common blocks, sembuf, contains the six row buffers, rb1, rb2.. rb6, together with a little extra space at each end to assist in edge pixel processing.

Semtab contains the Semper variable table - an array of values followed by an array of the corresponding names (see §1.10 for details of name coding); the Semper *fixed* or *protected* variables (select, displa etc.) are held here in fixed positions and can be referred to directly - but note that min, max, mean, me2, sd are named vmin, vmax.. to avoid possible confusion with Fortran generic functions.

Semcbs contains many short arrays comprising *Device* and *Picture Control Blocks* (*DCBs* and *PCBs*), i.e. information on assigned devices and open pictures used during picture storage and retrieval by Semper; it is from the PCBs that you find out picture sizes etc. (see the next section, for example).

Semfrl, sempri, sempli, semcip and semtdb all contain information concerned with command interpretation, such as the current command line buffer, for loop pointers, the program call stack and the program library priority order; perhaps the only variables of direct interest to you are verb, which contains the name of the verb currently being processed (useful if you are writing a module that provides more than one new command), and error, which returns error codes from processing modules.

Semarg is a short block available for local use to any processing module provided it does not call other processing modules which use the same block.

Semcfg contains a few system configuration details liable to alteration at run-time or session start-up, such as the amount of memory used to buffer disc data; most such details are however provided as parameters instead.

Semdpb provides information about a current display partition, and semgdb holds details of the currently selected graphics coordinate system (see §3.4 for details); semldb records the modes of the various LUTs currently defined.

The parameters likely to be found useful include those named lnxxxx, which give the length of various arrays and datatypes (e.g. lnbuf is the length in bytes of the row buffers rb1,rb2.. and lnint is the length in bytes of the integer datatype); and those named nfmxxx and nclxxx, which give the data form and picture class numbers (e.g. nfmint is 1).

Obviously, any casual resetting of common variables may have catastrophic results; don't do it!

1.8 Picture size, class, form and storage information

Amongst the most useful things in common are the series of short arrays in which Semper collects information on the pictures currently open; you can initialise ncol and nrow to the dimensions of picture lp1, for example, by inserting the lines

```
NCOL=NCOLS(LP1)
NROW=NROWS(LP1)
```

at the start of your module deriv. Taken together, the various array items indexed by the LP number lpn (all of which are integer) are said to comprise a *Picture Control Block* (*PCB*) for logical picture lpn. A complete list of PCB items is given in Appendix 2, but those you refer to most often are:

ncols(lpn)	number of columns
nrows(lpn)	number of rows
nlays(lpn)	number of layers
ccoln(lpn)	column number)
crown(lpn)	row number) of coordinate origin
clayn(lpn)	layer number)
classn(lpn)	class number
formn(lpn)	form number

1.9 Accessing keys, options, and other variables

Your module deriv is almost complete; what is still missing is the initial setting of c, s and cps on the basis of a direction theta. You will want to be able to indicate the direction of differentiation as a key to the command (e.g., gradient angle pi/3); the way you refer to a key from Fortran code is by asking VAL:

```
THETA=VAL(NANGLE)
C=COS(THETA)
S=SIN(THETA)
CPS=C+S
```

The argument nangle to VAL is to supply the name angle to it in the coded or packed form used throughout Semper for name handling; forget about giving it the appropriate value until the next section.

> Val returns the value of the indicated Semper variable if it is set, and zero if it is not; see also ival, ivalpn in Appendix 1.

For completeness, suppose you also wanted to be able to calculate the modulus of the gradient (its magnitude, or absolute value) as an option to the command (e.g. gradient magnitude..). For this purpose you will need to insert an extra line of code before label 10 of deriv:

```
IF (MAGN) RB1(I4)=ABS(RB1(I4))
```

and declare magn to be logical at the start of the module; the way you refer to an option from Fortran code is by asking opt:

```
MAGN=OPT(NMAGNI)
```

with nmagni being a similarly coded name.

Opt returns .TRUE. if the indicated Semper variable is set with a non-zero value, and .FALSE. otherwise. Where you wish to assume an optional facility unless it is explicitly negated via a no prefix, you use optno instead: optno(name) returns .TRUE. only if the variable is set with a zero value.

More generally, you use semlu for access to Semper variables. This allows you to unset a variable, sets a variable with a given value, (so as to return values as Semper variables at the end of a command), and set a variable *locally* in the sense that the original state of the variable is recorded and restored at the end of the current command - as the interpreter sets key and option variables. Appendix 1 gives details of semlu, and also of the related 'packaged' forms varset and varint.

1.10 Coding or packing Semper names

All names in Semper are passed between modules in the form of integers into which up to three characters can be coded or packed. You can find out the packed form of a name by using the command pack if you type pack angle Semper replies with the the number 2167, which defines the characters a, n and g. You can find the appropriate value for nmagni in the same way, and insert

```
PARAMETER (NANGLE=2167,NMAGNI=20847)
```

at the start of deriv accordingly. There is in fact no reason why you should not code theta=val(2167) directly if you prefer this; the pattern above is used only for readability. You can also use the function ipack('mag'), which returns 2167.

In detail, name packing involves assigning integer values c1, c2 and c3, to the three characters of the name, according to the Radix-50 code -

0	space
1-26	A to Z (or a to z)
27	$
28-29	not used
30-39	0 to 9

and then combining these as follows:

```
if c1<20  name=(c1*40+c2)*40+c3
otherwise  name=-((c1-20)*40+c2)*40-c3-1
```

Names with fewer than 3 characters are packed with trailing spaces. You can unpack names if you need to unpack, e.g. unpack 2167 produces the reply MAG.

1.11 Listing of the completed module DERIV

A logical declaration and an end statement are all you now need to complete a workable module deriv:

```
      SUBROUTINE DERIV
      LOGICAL SEMROW,MAGN,OPT
      PARAMETER (NANGLE=2167,NMAGNI=20847)
      INCLUDE 'COMMON'
C Establish direction and magnitude option
      THETA=VAL(NANGLE)
      C=COS(THETA)
      S=SIN(THETA)
      CPS=C+S
      MAGN=OPT(NMAGNI)
C Establish picture dimensions
      NCOL=NCOLS(LP1)
      NROW=NROWS(LP1)
C Initialise row loop: pre-fill first buffer
      IF (SEMROW(1,RB1,NFMFP,1,1,LP1)) RETURN
C Set buffer pointers
      I1=1+LNBUF/LNREAL+2*LNEDGE
      I2=1
      DO 20 J=1,NROW
C Fetch source row
        IF (SEMROW(1,RB1(I1),NFMFP,J,1,LP1)) RETURN
C Pass along row obtaining derivative pixels
        I3=I1+NCOL-1
        I4=I2
        DO 10 I=I1,I3
          RB1(I4)=C*RB1(I+1)+S*RB1(I4)-CPS*RB1(I)
          IF (MAGN) RB1(I4)=ABS(RB1(I4))
          I4=I4+1
   10   CONTINUE
C Store output row
        IF (SEMROW(2,RB1(I2),NFFMP,J,1,LP2)) RETURN
C Swap buffer pointers
        I=I1
        I1=I2
        I2=I
   20 CONTINUE
      RETURN
      END
```

This is the point at which to try compiling your new module and removing at least the syntactic errors that a compiler detects.

Chapter 2

Interfacing a Module with the Interpreter

The information the Semper command interpreter needs about individual commands is held in a file semper.vds of *Verb Descriptors* (*VD*s) used by the system generation program semgen. Interfacing your module with the system involves adding a new line that describes your new command to the file semper.vds, re-running semgen and re-linking the system. This chapter tells you how to do these operations. You may like to begin by reading the present text of semper.vds, as some people will be able to make the changes they need simply by following the existing pattern.

2.1 Layout of the Verb Descriptor file

Within the file semper.vds, each command is described by a line beginning with the verb name, so that your new VD should therefore begin

```
gradient
```

Following the verb you give a series of items which define the keys, options, and other details of the command; the items are delimited by spaces. Only the first three characters of names are significant; upper and lower case letters are treated as equivalent. Individual lines may be up to 80 characters long, and a + sign placed at the end of a descriptor line indicates that the descriptor continues on the next line. Lines beginning with ! (comments), and blank lines, are ignored. The entire file is terminated by a line beginning with the word end.

Sometimes several commands may have completely identical VDs; instead of repeating the VDs for each verb, you then simply list the various synonymous verb names at the beginning of the VD, separated by commas, e.g.

```
Mean, sharpen, hp ..
```

Whether or not a command is logged by the interpreter depends on its position in the Verb Descriptor file: those defined early (involving display or inspection only) are not logged, and logging occurs only for verbs following a 'switch-on' line beginning with the letters log (perhaps in the form of a comment 'Logging begins here'). You should place gradient anywhere you like after this line.

Some lines of the file semper.vds in fact define named macros instead of commands; see §2.6 below for details.

2.2 Key and option definitions

The next things you need in your VD are keys angle, from and to, and the option magnitude. A key is defined by quoting its name followed by an equals sign and an optional expression giving a default value. To set up angle, and to make it default to the value of the variable theta when not quoted, you therefore write

```
gradient angle=theta
```

If no expression is given after the equals sign, the interpreter provides no default for the key, and no local setting is made unless the key is quoted explicitly in a command.

> Default expressions have the same syntax as the expressions in Semper commands, except that embedded spaces are not allowed, and can be as general, containing Semper variables, functions etc. They are evaluated at run time, using the then current values of variables, after any explicitly quoted keys have been processed, and in the order in which the keys appear in the VD.

You should follow the standard pattern of using keys from and to to indicate source and output pictures for your new command, unless you have good reasons for doing otherwise. This requires items

```
from=select to=from
```

which give the usual defaults for source and output. To make it possible to omit the key names themselves, you in fact code

```
$1=select from=$1 $2=from to=$2
```

instead, as it is the names $1, $2.. that the interpreter in fact assumes for omitted keys. (You may need to think carefully about this to see that it achieves what is required!)

An option is defined simply by quoting its name; to set up the option magnitude, you therefore add its name to the VD:

```
gradient angle=theta $1=select from=$1 $2=from to=$2 magnitude
```

> No special action is needed to allow the general keys and options (mark, byte etc.) to be used; these are provided automatically by the interpreter, and should not appear in the VD.
>
> If you want a key to introduce a pair or set of expressions, both the basic key name and the associated name or names must be listed in the VD; the command scale range 10,20, for example, with a default range of 0,255, requires VD items
>
> ```
> scale range=0 ra2=255
> ```

Finally, to set up a key that is to take a textual rather than numerical value, you write a single quote following the equals sign:

```
title text='
```

(see §3.5 below for details of how to use such keys.)

Up to 50 keys and 50 options may be given in any one VD. Their order is immaterial (except where a particular order of default evaluation is required because one default involves another key, as is the case with from and to).

2.3 Processing module calls

The next thing the interpreter needs to know about your command gradient is what processing module (subroutine) to call once it has finished key/option processing etc. You provide this information by giving the module name prefaced by a colon, anywhere in the VD:

```
gradient :DERIV angle=theta $1=select from=$1 $2=from to=$2 magnitude
```

You may not call more than one module in a single VD.

2.4 Picture opening

The last matter you must deal with takes you back to the LP (logical picture) numbers lp1 and lp2 used in deriv for source and output pictures. Each of the pictures used by a Semper module must be *opened* before processing; this means that you must ask Semper to determine where it is stored and what its characteristics are, and to construct the PCB (picture control block) referred to in §1.8; Semper assigns an LP number at this stage by means of which you refer to the picture subsequently. The simplest way of effecting this opening is to add *open requests* to the VD specifying that pictures from and to are to be opened and that the LP numbers assigned are to be placed in lp1 and lp2 for you to use.

Picture opening involves some device-specific actions, such as disc directory lookup, tape file scans or display erasure, and some actions common to all devices, such as label processing and PCB construction. The LP number assigned is simply the index (1,2..) used for the PCB array items assembled for the picture. Besides the items listed in §3.8, the PCB contains other information needed for picture row i/o that is not easily assembled from scratch.

To open the source picture and set lp1 to the LP number assigned, you write

```
open(lp1,old)=from
```

anywhere in the VD. The old indicates that the picture must exist already, and the from that its number is the current value of the variable from. In this case, from will have been set explicitly or by default during key processing before the open requests are dealt with; where the value is less than 1000, the interpreter inserts the device number cd in the usual way. (In fact, a general expression for

the picture number may be given after the equals sign, exactly as for key processing.)

To open the output picture and set lp2 to the LP number assigned, you write

```
open(lp2,new,lp1)=to
```

anywhere in the VD. The new indicates that a fresh picture is to be created, with the same dimensions, title etc. as logical picture lp1 (the *reference* picture), and the to that its number is to be the current value of the variable to. The full VD now reads

```
gradient :DERIV angle=theta $1=select from=$1 $2=from to=$2 magnitude +
  open(lp1,old)=from open(lp2,new,lp1)=to
```

The item lp1 in the open(lp2.. request in fact only supplies default dimensions for the new picture. Where the variables size, si2 and/or si3 are set at the time of opening (usually through key definitions earlier in the VD), their values are used for the new picture dimensions instead. Thus writing

```
size= si2= open(lp2,new,lp1)=to
```

in the VD allows you to override the output dimensions when you want to by quoting something like size 100,50 in your command.

If, as in this example, no defaults are given for the size keys, the interpreter provides standard defaults ((exactly as defined in the section of the Users' Guide describing picture subregions). You may omit the reference LP number altogether, in which case the size keys are essential to define the new picture size, and a class Image picture with form fp is created.

Up to 3 open requests (for lp1, lp2 and lp3) may be given in a VD; they are processed in the order in which they appear in the VD, after all option/key processing.

2.5 Continuation and Routine VDs

The VD for deriv is now complete. However, it is unnecessarily lengthy: since many VDs have considerable numbers of items in common, a *jump* facility is provided to allow them to share one copy of these. An item >name causes the VD to *continue* with the items in the VD for the verb name (which must appear earlier in the file), and a *ket* > on its own causes a return to the original VD level (if any) after the fashion of a subroutine return.

You will find that the standard file semper.vds contains among its early lines

```
$r3 $1=select from=$1 $2=from to=$2 open(lp1,old)=from >
$ft >$r3 open(lp2,new,lp1)=to
```

(recall that $ is treated as alphabetic); the continuation $ft, calling the *routine* $r3, provides the keys from and to, and the corresponding open requests. You can make use of these to shorten your VD so as to read

```
gradient :DERIV angle=theta magnitude >$ft
```

The widespread use of continuation and routine VDs in this way greatly reduces the size of the code generated from semper.vds by the system generation program semgen. You will see several already in use in the VD file, particularly select which provides for opening a source picture (keyed by from) only (and incidentally implements the verb select. Routine VD calls cannot be nested however; the return mechanism only works to one level.

2.6 Defining named macros.

One last feature of the Verb Descriptor file remains to be described, although you make no use of it for deriv, namely how the named macros such as @region are defined. You will see the present macro definitions near the end of semper.vds: each consists of a line beginning with an @ sign and the macro name, and continuing with the replacement text, e.g.:

```
@xy pos x,y
```

The definitions may be continued on to further lines as usual if necessary, and may involve further macro calls, e.g.

```
@region siz r,r2 @xy
```

You are free to add whatever macros of your own you may wish.

2.7 System regeneration

Semper does not in fact refer directly to the Verb Descriptor file at run time, as the structure of Fortran does not allow modules to be called so flexibly. Instead, a main program containing all the module calls, and a module semvds containing the VDs in a compact coded form, is generated from the VD file by the system generation program semgen; these must be compiled and linked with the rest of the system modules, and with your new module deriv, to form a new load module incorporating your new command gradient.

Semgen reads the file semper.vds on an input unit, writes the main program and semvds to an output unit (or to two separate output units), and writes error or verification messages to a further output unit - normally your terminal. (The actual i/o unit numbers used are installation-time parameters). You should try running it now. Normal termination is confirmed by the message System Generation Complete.

If syntactic and structural errors are found in the Verb Descriptor file, or if internal tables overflow, semgen terminates with a

short error message, including the line of semper.vds in which the error was detected. Internal table overflow will only occur if your semper.vds file grows considerably larger than the standard one supplied; when it does, you will have to recompile and relink semgen, increasing the relevant parameter appropriately.

Chapter 3. More complicated Modules

This third part of the manual deals with some further questions that you may find yourself asking in time, such as effecting picture opening directly rather than with the interpreter, reporting errors you detect yourself, and making textual or graphical annotations on display pictures. A number of miscellaneous points needing occasional attention are also mentioned.

3.1 Error notification

The coding of the important system modules as logical functions returning .TRUE. in the event of an error makes it unnecessary to consider errors directly for most purposes. However, there will be some errors that only the new module can detect, and so the mechanism for reporting them must be explained.

Basically, you report an error to the interpreter by returning from the module with the (integer) common variable error (see §1.7) non-zero. You indicate the cause of the problem by different error values (*error codes*), which lead to different error messages being generated by the interpreter. If you type help errors, Semper responds with a full list of the standard error codes; you can use this to choose the most appropriate error code to return.

You will see that many error messages include actual names or numbers of the offending variables, pictures etc.; you provide this information where appropriate via further common variables (most frequently, iderr); look at the system error message file semper.err to see exactly how these are used for each message.

Since your present code at least does not handle 1-D pictures, you might protest if gradient is applied to such a picture; the following code would be necessary:

```
IF (NROWS(LP1).EQ.1) THEN
   ERROR=5
   IDERR=IVALPN(NFROM)
   RETURN
ENDIF
```

which would produce the message Bad size for picture nnnn; provided you set NFROM earlier to the packed form of the name from, the function ivalpn (see Appendix 1) would return the source picture number with the current device number cd prefixed if necessary.

> Sometimes of course you will not find a suitable message in the standard list. In such cases you should simply write an explanatory message to the console (see §4.5) and return with error code 10 - this causes error recovery procedures but produces no further error message. As an alternative, you could add an additional line to the error message file, with the same pattern as the existing lines, defining a new code of your own; use codes begining at, say, 801 to avoid a clash with code use by Synoptics in later releases of Semper.

Error is normally set suitably by system modules such as SEMROW when they make a .TRUE. return, so that the calling module need only return immediately.

3.2 Opening pictures directly

Sometimes you may wish to open pictures yourself without relying on the interpreter - for example, your module may need to decide for itself what size an output picture should be, or indeed whether to produce output at all. You can do this via semopn, which is a logical function returning .TRUE. in the event of an error:

```
IF (SEMOPN (1,NPIC,NC,NR,NL,ICLASS,IFORM,LPN)) RETURN
```

opens an *old* (existing) picture npic, locating it on whatever device it is stored on, constructing the PCB, and returning the LP number assigned in lpn (for you to transmit to semrow etc.); the number of columns, rows and layers, the class number and the form number are returned to you in nc, nr, nl, iclass and iform.

The picture number must include the device part, as semopn does not insert cd for you; the integer functions ivalpn() and semppn() provide convenient ways of doing this (see Appendix 1).

Write-only display devices may be opened in old mode by semopn, but not read by semrow.

If you give a first argument (an opcode) of 2 instead, and set nc, nr, nl, iclass and iform to the size, class and form you want, semopn opens a *new* picture npic instead, usually finding fresh storage for it. As before, LPN returns the assigned LP number to you; however you can also set it before calling semopn to the LP number of any picture whose title you would like copied to the new picture label. This is how the general source-to-output title copy works; provide a zero value in lpn if you want to disable it.

In choosing an output form, you should follow the convention of opening output pictures with the same form as the source, unless any of the general options byte, integer, fp and complex are set, wherever possible. An integer function SEMFRM is provided to make this simple: semfrm(n) returns the corresponding form number if any general option is set, and n otherwise; you normally write semfrm(formn(lp1)) for your output form accordingly.

The coordinate origin of any picture whose title is copied to a new picture is also copied provided that the dimensions of the new picture match those of the old.

When a new disc picture is opened, any old picture with the same number is implicitly deleted, though its storage is not in general actually released until the end of the current command. Reopening in new mode a picture already opened during the current command (which happens in commands such as scale or scale 53) is specially treated: normally the old picture remains independently available via its own

PCB until the end of command; if however the two pictures match in dimensions and form, the existing storage is re-allocated immediately for the new picture, leading to true *in-situ* processing. (See also §3.8, §3.9 below.)

New display opens normally include erasing the partition if the variable erase is set to yes; the values of the variables min, max are recorded at the moment of opening to define the black,white levels for display scaling. If the picture has already been opened during the current command, further opening of the same picture is faulted unless the dimensions match and the form is compatible, and no further erasing occurs.

New tape opens are faulted; the creation of tape pictures is the exclusive responsibility of the command copy in view of the special care required to maintain a safe tape.

Semopn in fact uses the first three row buffers (rb1, rb2 and rb3) as internal workspace (to hold directories, labels etc.), so that these do not in general survive calls to semopn. You will need to plan accordingly should you need to delay opening pictures until after processing is under way.

You do not normally need to *close* pictures after processing; they remain open between commands, with the result that later commands opening the same picture do so without any significant overhead. How many PCBs you may have open at once is determined by an installation-time parameter; you are prevented from opening too many within any one module.

If you need to open an indefinite number of pictures *in turn* during a single command, you can in fact close a picture and release its PCB with semcls:

```
IF (SEMCLS(LPN)) RETURN
```

Alternatively, you can use the magic formula basewh=currwh, which causes Semper to 'forget' about having opened any pictures during the current command; this has the advantage that the pictures are not actually closed unless it becomes absolutely necessary subsequently - tape pictures are tedious to re-open.

3.3 Abandon requests

Few modules need take any action to ensure that abandon requests are honoured, because the system modules semrow and semopn already monitor them, and return .TRUE. with error set to 4 if one is made. If a new module spends a long time without calling these access modules at all, you should include an occasional call to the primitive abandn to ensure that the process can be abandoned when requested:

```
IF (ABANDN(ERROR)) RETURN
```

3.4 Display graphics

Semper allows you to make graphical or textual annotations to display pictures, normally in the separate *overlay* memory, directly in terms of picture coordinates without worrying about partition frames and positions, picture origin offsets, undersampling etc.

Suppose you want to mark the direction of differentiation by deriv. Following the pattern of other commands, you examine the variable mark to determine which display is to be annotated, doing nothing unless mark is set; the necessary test is packaged in conveniently in marset:

```
IF (MARSET(ANNOT,N)) RETURN
```

which sets logical annot to .TRUE. if mark is set, returning its value in N (or the value of DISPLAY if mark is in the range 1-999). You make subsequent code concerned with graphics conditional on the value of annot:

```
IF (ANNOT) THEN
```

and prepare for annotating display picture n via fsinit:

```
    IF (FSINIT(3,N)) RETURN
```

The initialisation lasts for the rest of the command (or until you use fsinit again; once it is done; you use the routine fsarro, which marks an arrow with given start- and end-point coordinates:

```
    R=MIN(NCOL,NROW)/2
    IF (FSARRO(0.,0.,R*C,R*S)) RETURN
ENDIF
```

marks an arrow from the origin extending for half the minimum picture dimension in the direction of differentiation.

Appendix 1 lists a large number of other graphical routines, all with names of the form fsxxxx, which mark borders, subregions, lines, circles and arcs, curves, text, and position lists, erase display regions, set monitor viewing conditions and accept positions indicated via the display cursor. You may use these freely.

> Although display pictures in the form of 1-D graphs and histograms cannot be opened via semopn, they *are* accessible to the graphical routines: for graphs, the Y coordinate corresponds to pixel values; for histograms, the X coordinate corresponds to pixel values and the Y to channel counts. When the display picture is complex, both parts are annotated.
>
> The first argument to fsinit is in fact an opcode establishing the graphics coordinate mode - see help graphics if you are not familiar with these. The values 1,2 and 3 mean frame, partition and picture

modes respectively. In all cases, fsinit fills common semgdb with information in a common form on coordinate mapping parameters, ranges, storage frames, the current values of mkmode and mksize etc. In frame mode, the information is derived directly from the *device control block* (*DCB*) for the display; in the other cases, it comes from the *display partition descriptor* (*DPD*), which is itself loaded into common sempdb. Appendix 2 gives details of the contents of both of these common blocks.

If you want to code new graphical facilities that are to operate in all three graphics coordinate modes, you may like to use fsoptn to establish the appropriate mode and frame/partition/picture number; this is the routine used by the commands erase and ramps, for example. See Appendix 1 for details.

The display primitives fsln61, fstx61 etc. *may* also be used for annotations, but this will rarely be necessary. They operate in terms of still another coordinate system, *display* coordinates (see the Implementation Guide). To convert any of the above coordinate systems to display coordinates, once you have called fsinit appropriately, you use code of the form

```
IX = NINT( FSXSCA * X + FSXOFF)
IY = NINT( FSYSCA * Y + FSYOFF)
```

Finally, a note on display output buffering. In some installations this means that annotations you make do not actually become visible until the next time the output buffer is flushed, which may not be until the end of the current command; you can force an immediate flush, giving users the chance to abandon the command at once if they don't like what they see, by inserting after your annotation code:

```
IF (FSFLUS(0)) RETURN
```

3.5 Textual keys

The value of textual keys (e.g. title text 'Raw data for plate ',n) is handled differently from the value of simple numerical keys. The current command line is held by the interpreter in an integer array linbuf, and the interpreter sets the key variable to the position (the Fortran subscript) within linbuf at which the text begins (or unsets it if the key is not used in the command). Before use, the text needs *evaluating*; the whole process is most easily effected with semktx:

```
IF (SEMKTX(NTEXT,'prompt',TEXT,LEN,UC)) RETURN
```

processes the text for a key named by ntext, placing the evaluated text as characters (in Semper's *internal code* - see the Implementation Guide) in the integer array text; len should be set before the call to the length of text, and indicates on return the number of characters in it; if you set logical uc to .TRUE. on entry, text is forced to upper case. If the string 'prompt' is not

blank, then a value is solicited from the terminal when the key is not set, using the string as a prompt (no carriage control being needed in this); errors in the value are trapped internally and the prompt reissued as necessary.

Semtyp (see Appendix 1) provides a lower level routine evaluating text from one buffer to another.

3.6 Single row and multi-layer pictures

Semper is designed primarily for 2-D pictures, but the filing system supports 1-D (single-row) and 3-D (multi-layer pictures) as well. Beyond the filing system, however, support is not uniform. If your code will not work for a one row picture, then you should test the source picture size and fault it if necessary, or else include a 1-D version of the algorithm in the module; the latter course is obviously preferable where it is reasonably easy.

Processing modules also vary considerably in the extent to which they can process multi-layer pictures. The simple alternatives open to modules coding essentially 2-D procedures are *either* to protest if the number of layers in the source picture is not one, *or* to provide an outer loop repeating your procedure for all picture layers, e.g.

```
      DO 30 K=1,NLAYS(LP1)
        ...
        IF (SEMROW( , , , ,K, )) RETURN
        ...
   30 CONTINUE
```

In some cases, you may be able to process a single layer, or a group of layers, indicated via the variables (general keys) layer,la2.

3.7 Moving picture origins

You can record a new origin for a picture, should you need to, by calling semcen:

```
      IF (SEMCEN(LPN,NC,NR,NL)) RETURN
```

records nc, nr, nl as the new origin column,row,layer numbers, provided these lie within the picture boundaries.

3.8 In-situ operations

Most commands function with independent source and output pictures, so that output rows do not overwrite source rows as processing proceeds. As noted in §3.2 above, this may not in fact be the case if a command uses the same picture number for source and output, and you may find the same LP number allocated for both pictures, indicating identical storage and other characteristics.

In writing a new module, you must therefore consider whether or not the processing algorithm used is viable *in-situ*, i.e. whether or not the output rows are produced

in an order which might overwrite source rows to be read later. If it is not, you must trap the *in-situ* case (by the identity of source and output LP numbers) and treat it as an error.

On the other hand, *in-situ* processing is sometimes faster and more convenient, reducing overall storage requirements, so that you may wish to exploit it. Where only a small number of pixels are actually changed by an operation, for example, you may be able to avoid reading or writing rows that are not altered.

3.9 Managing forms and intermediate storage

As noted in §1.3 and §1.5, you do not need to think about source and output picture forms for most purposes, as Semper converts row forms on input and output as necessary; code you write to operate one row buffer form will also work on the simpler (shorter) forms, though with no greater a maximum row length, and on the more complicated forms also, though with lower precision (and the loss of any imaginary parts of pixels). From the point of view of speed, it would be desirable to provide separate code for integer, fp and complex forms (byte is not a standard Fortran datatype); however, this would mean lengthy code and tedious coding. Most standard processing modules in fact use fp internally; many accommodate complex as well via outer loops or increment switching; a few provide parallel integer code, and a few use integer only.

Additional thought *is* needed in cases where an algorithm involves more than one pass through the picture rows, and intermediate results may have a very different range from the final result. The command image (the inverse Fourier transformation command) provides a good example: a command image to .. byte produces output in byte form, which the final data may indeed fit; if however the results of intermediate passes are held in such an output picture - and the convention of not altering source pictures means that they cannot be held in the source - they will overflow hopelessly.

Two courses of action are possible in these circumstances. Firstly, you can open the output more than once, quoting different forms in the two cases, e.g.:

```
IF SEMOPN(2,ITO,NC,NR,NL,ICLASS,NFMFP,LP2)) RETURN
..code for all but final pass..
IFORM=SEMFRM(FORMN(LP1))
IF (SEMOPN(2,ITO,NC,NR,NL,ICLASS,IFORM,LP3) RETURN
..code for final pass..
```

For output on disc, semopn handles repeated use of the same picture number like this safely, allocating different storage with each change of form; for display output, the common grey scaling enforced on the output pictures may frustrate the process. Secondly, you can open a *temporary* disc picture for intermediate storage by calling semopn with opcode 3:

```
IF (SEMOPN (3,NPIC,NC,NR,NL,ICLASS,IFORM,LPN)) RETURN
```

behaves exactly like semopn(2.. except that it ignores npic and finds space for

the picture on some unprotected disc device from which it is deleted automatically when your module returns.

3.10 Maintaining the current picture number

Normally you need take no action in this respect, but you may occasionally wish to override the default selection mechanism. Semopn selects the first picture you open successfully in a command (whether through the VD or directly); for this reason you should not open any auxiliary input picture you might have before your main source.

Subsequently, the last picture written to during a command by semrow is selected at the end of the command, provided it is on the current device cd; until then, the relevant LP number is held in a common variable olpn, and you can override the selection, should you need to, by setting select directly and ensuring that olpn is zero before returning from your module.

3.11 Note on disc caching

In spite of the remarks in §1.1 to the effect that the presence or absence of disc caching the Semper installation does not affect processing module design, some idea of what can and what cannot be achieved by the cache may be helpful.

The basic principle used is the grouping together of many disc blocks for transfer to and from the physical disc only in larger units such as tracks; serial reading and/or writing through a picture then involves physical disc transfers only occasionally rather than on every picture row, reducing enormously the time otherwise spent in requesting control of the disc, moving the head, etc. The caveat to be given here is that although the caching of several areas allows the benefits to be achieved even when (e.g. during Fourier transformation) serial accesses (forwards or backwards) are being made at several different places independently, it must be appreciated that a random pattern of row accesses will be efficient only if the accesses are not too widely scattered, so that all of the disc blocks concerned are within the cache area; random accesses over larger areas will in fact be slightly less efficient when caching is employed than when it is not.

Note also that there is no provision for 'turning caching on or off' during a session, nor for making the memory area used for the cache available for other purposes.

Chapter 4. Utility Modules

4.1 Establishing picture subregions

Since subregions can be specified in general by so large a number of variables - left, right, bottom, top, far, near, position, po2, po3, size, si2, si3, layer, la2 - you usually ask tstsrg to test them for you instead of examining them directly. A call

```
IF (TSTSRG(1,LPN)) RETURN
```

uses the variables to establish a subregion of logical picture lpn, and sets variables in common semarg as follows:

smgi1,2,3	first column,row,layer number in subregion
smgi4,5,6	last column,row,layer number in subregion
smgi7,8,9	number of columns, rows, layers in subregion
smgl1	.TRUE. if subregion specified (default is entire picture)
smgl2	.TRUE. if subregion is entirely disjoint (see below)
smgl3	.TRUE. if subregion pixels fall exactly on top of source pixels (i.e., no interpolation needed)

You can use smgi1-6 as DO loop limits etc., or pass them to other modules as described in §4.2.

Opcode 1 to tstsrg truncates subregions which initially extend beyond the picture boundaries, and faults only cases where the subregion is entirely disjoint. Opcode 2 is identical except that the subregion is faulted if it does not lie entirely within the picture. Opcode 3 allows integer values of the variable sampling to describe de-magnified subregions; and opcode 4 allows rotation (variable angle) and non-integral re-sampling (variable sampling, or skewing (uv, in conjunction with u,u2 and v,v2).

In addition to the above, smgr1,2 are set to the x,y picture coordinates of the subregion top left; and smgr3,4 and smgr5,6 to the vectors corresponding to one step across and one step up the subregion respectively; these are chiefly, but not exclusively, useful with opcode 4.

A subregion specified in this way can be marked on a display via mrkreg:

```
IF (MRKREG(0)) RETURN
```

outlines the region on the display indicated by the variable mark, if this is greater than zero.

4.2 Ranges, statistics, and histograms

You can find the range (minimum and maximum pixel value) of a picture or subregion conveniently by asking range:

```
IF (RANGE(1,LPN)) RETURN
```

sets the common variables vmin, vmax (the Semper variables min, max) to the range of the within logical picture lpn. You specify a subregion, if you want one via smgl1 and smgi1-6, which have the same meanings as they do for tstsrg (see §4.1 (so that calling this immediately before the range call is a convenient way of arranging a subregion scan); to scan the whole picture however, you need only set smgl1 to .FALSE.. Provided you are scanning the entire picture, range recovers the range from the picture label where possible in preference to scanning the data itself.

> If an abandon request is made before range has finished, the range of the data already scanned is returned: in these circumstances range does not return TRUE, but does set error to 4, so error should be tested or cleared following any call to range.
>
> If the Semper variable preset is set, range returns immediately, so that the current values of vmin, vmax survive; accordingly, you can usually provide a preset option without any code of your own.

Whenever the entire picture has been scanned, range records the range determined in the picture label; other opcodes allow the record to be deleted or updated directly. Semper notes any write operations on a picture and deletes the range record at the end of the command; should it ever be necessary to apply range to a picture you have altered within the current command, you should use opcode 2 which differs from 1 only in that it ignores any recorded range and scans the data itself unconditionally. Opcodes 3 and 4 allow you to delete a range record, and record a new range, respectively; you should take care however never to record a range that is not exactly correct!

You can also use a module meansd(lpn) to determine the mean and standard deviation of a picture or subregion as well as the range; and a module genhst(lpn,nchan) to construct in the first nchan reals of rb1 a histogram of a picture or subregion spreading nchan channels over the range vmin, vmax. Both of these are identical to range with respect to how the region to be scanned is specified; meansd records the range in the label when appropriate.

> The output of genhst is suitable for storing as a class Histogram picture if the values of vmin, vmax are added as two additional pixels.

4.3 Fourier transforms

The modules ft2d and invft2 which perform forward and inverse 1-D and 2-D Fourier transformation have a number of optional features concerned with efficient calculation of power spectra and correlation functions, and consequently look more difficult to use than they are. All that is actually needed for a forward transform is

```
IF (FT2D(ITO,NCLFOU,NFMCOM,.FALSE.,.FALSE.)) RETURN
```

which transforms logical picture lp1 to a new picture number ito (containing the right half-plane only if lp1 is not complex). Similarly,

```
IF (INVFT2(ITO,NCLIMA,.FALSE.,.FALSE.,0.)) RETURN
```

inverse transforms logical picture lp1 (including renormalisation) to a new Image picture number ito, with a form determined by the usual general form options (defaulting to fp / complex according to whether lp1 is a half-plane of full-plane transform).

4.4 Calling other processing modules

Few standard processing modules call other top level modules, though they do share lower level modules to some extent. However, there is in fact nothing to stop you from calling other processing modules from a module you are writing, as long as you can deduce from the documentation, the file semper.vds and/or the code itself exactly how they function. Essentially, you need to imitate the actions of the interpreter before calling the module: you set verb (in case the module implements several different commands), set key/option variables *locally* (using semlu(2,..)), and open any pictures necessary.

4.5 Other textual input/output

Any text that you may wish to output to the terminal or to any log file must be output by means of one of several routines provided. You pass the text string as a character variable. The character variable record may be used for this purpose. You use the routine semcon for console output and semlog for log output. Any diagnostic messages (error messages in particular) must be output with semdia. For example:

```
IF (SEMCON('Processing complete')) RETURN
IF (SEMDIA("Bad input data values',NDIERR)) RETURN
```

The second argument in the call to semdia defines what type of diagnostic message it is (informational, warning, error or fatal error). The values to use are provided as the parameters ndimes, ndiwar, ndierr and ndifat.

Note that Fortran formatted WRITE statements can still be used by writting the text into a character variable, e.g.

```
      WRITE (RECORD,193) CHISQ
193   FORMAT ('Confidence of fit ',F8.2)
      IF (SEMLOG(RECORD)) RETURN
```

There is nothing to stop you conducting a terminal dialogue within a Semper processing module just as you might within any other program, should you need to gather data unsuitable for transfer via the normal key/option variable mechanism. A number of routines have been provided for this. Firstly, the routine sempro should be used to output any prompt string. A call to readln will then read a line of text from the keyboard (returned as an array of integer Ascii codes), with all of the line editing features that are present for normal Semper command input. The line of text should then be output by means of the routine seminp. This last call makes it possible for all the details of an interactive session to be logged in the usual way. For example:

```
IF (SEMPRO('Enter number of iterations: ')) RETURN
IF (READLN(TEXT,N,ERROR)) RETURN
CALL SEMCHS(RECORD,TEXT,N)
IF (SEMINP(RECORD)) RETURN
```

Converting from integer Ascii codes to character form and back again is carried out by the routines semchs and semics. Encoding and decoding of names and numbers in the form of integer Ascii codes is done with the routine semxa1 and decoding numerical expressions is done with the routine semexp. You can write text in this same form to the display with the routine fstext (see §3.4).

A lower level of keyboard input is possible with the routine inkey which reads a single keystroke, without echo and returns the integer Ascii code for the keystroke. All of the possible keystroke values are defined as parameters in the file icset.for.

The details of all the routines mentioned in this section are given in Appendix 1.

4.6 Low level data access

A level of device i/o below semrow is furnished by the system modules disc and tape, specifications for which appear in Appendix 1. These will rarely be needed, and should only be called directly if you feel sufficiently experienced since they bypass picture directories, labels, picture level WP flags etc. , though they do honour the device level WP flags. Similar but even stronger warnings apply to any use of the *primitive* modules mcdc61 and mctp61; the more ephemeral nature of display picture however means that use of the framestore primitives fsxx61 is relatively innocuous.

For most purposes, you should not refer directly to picture labels; however they are in fact normally left in an integer array equivalenced to rb1 on exit from semopn, and can also be read/written directly via a module semlab (see Appendix

1). Their contents, mainly defined via parameters, are listed in full in Appendix 2, and it will be seen that some space *is* still available at present, though this is liable to be used in later versions of Semper.

Appendix 1. Module Specifications

These specifications are brief for quick reference, and in many cases important general explanations are given at the appropriate points earlier in the manual. The typeface of dummy arguments indicates their mode of use: Helvetica for input usage only (values passed to the module) *Times-Roman italic* for output only (values returned by the module), and Times-Roman for input/output.

Unless otherwise stated, the first letter of argument names indicates their datatype (real, integer etc.) in the standard Fortran way; '*4' appended to a name however indicates an explicit integer*4 declaration (see §1.3). Unless otherwise stated, .TRUE. returns from logical functions indicate error conditions (made with error already set appropriately), and *characters* are Semper internal code characters (integers in 32-126; see the Implementation Guide). The *primitive* modules written locally for each implementation are specified in the Implementation Guide, and are not included here.

DISC IF (DISC(iop,idev,nitems*4,mem,blkn*4,memf,discf)) RETURN
performs disc file access, via the cache when there is one, according to iop:

0 initialises cache buffer contents (start of session only)
1 reads nitems items from block blkn.. of disc file idev to array mem, converting from form discf to form memf
2 writes nitems items from array mem to block blkn.. of disc file idev, converting from form memf to form discf
3 flushes and marks clear all buffers for disc file idev (used on deassigning files); if idev < 0, flushes buffers for all files but retains contents (used for security updates).

Disc checks that idev is legal and assigned to disc, and refuses to write to protected devices; however, it operates below the level at which individual picture protection operates, and so should only be used when absolutely necessary. In non-caching installations, transfers are rounded up to a whole number of disc blocks and the receiving buffer mem must be large enough to accommodate this on reads.

FSARC IF (FSARC(xcen,ycen,rad,th1,th2)) RETURN
following an initial fsinit call to establish a graphics coordinate system, marks a circular arc centred at xcen,ycen with radius rad, covering the angular range th1-th2 (radians) anticlockwise from the +X axis.

FSARRO IF (FSARRO(x1,y1,x2,y2)) RETURN
following an initial fsinit call to establish a graphics coordinate system, marks an arrow from x1,y1 to x2,y2.

FSBORD IF (FSBORD(idummy)) RETURN
following an initial fsinit call to establish a graphics coordinate system, marks the picture / partition / frame border as appropriate.

FSCIRC IF (FSCIRC(xcen,ycen,rad)) RETURN
following an initial fsinit call to establish a graphics coordinate system, marks a circle centred at xcen,ycen with radius rad.

FSCURV IF (FSCURV(x,y,n,closed)) RETURN
following an initial fsinit call to establish a graphics coordinate system, marks a series of straight lines linking the positions x(1),y(1) to x(n),y(n), with an additional line closing the curve if logical closed is .TRUE.

FSERAS IF (FSERAS(iop,x1,x2,y1,y2)) RETURN
following an initial fsinit call to establish a graphics coordinate system, erases the rectangular region spanned by x1-x2, y1-y2, according to iop:

1 image memory only
2 overlay memory only
3 both image and overlay memory

FSFLUS IF (FSFLUS(idummy)) RETURN
'flushes' the display buffer (outputs anything currently buffered), and clears Semper's internal 'flush request' flag.

FSINIT IF (FSINIT(mode,ndis)) RETURN
initialises the coordinate mode and target display for subsequent graphics via the FSXXXX routines, according to mode:

1 frame mode ndis = device:frame
2 partition mode ndis = device:partition
3 picture mode ndis = device:picture

Specifically, the common block semgdb is constructed, for reference by subsequent routines; in partition and picture mode, the DPD is loaded to common semdpb as part of the process.

FSLINE IF (FSLINE(x1,y1,x2,y2)) RETURN
following an initial fsinit call to establish a graphics coordinate system, marks a line from x1,y1 to x2,y2.

FSLIST IF (FSLIST(x,y,n,mkmode,mksize)) RETURN
following an initial fsinit call to establish a graphics coordinate system, marks the n positions x(i),y(i); the style and size of mark is determined by mkmode,mksize exactly as it is for fsmark.

FSMARK IF (FSMARK(x,y,mkmode,mksize)) RETURN
following an initial fsinit call to establish a graphics coordinate system, the position x,y in a style determined by mkmode as follows:

1 upright cross, size 2*mksize+1 pixels horizontally and vertically
2 diagonal cross, size 2*mksize+1 pixels horizontally and vertically
3 upright box, size 2*mksize+1 pixels horizontally and vertically
4 diagonal box, size 2*mksize+1 pixels horizontally and vertically
5 single pixel

FSOPTN IF (FSOPTN(*mode,ndis*)) RETURN
tests options frame, partition and picture, sets mode appropriately, and sets ndis to the value of the assumed key $1 (cframe in default for frame mode, display in default for the other modes); used for initialising multi-mode graphics facilities such as the erase command.

FSREGN IF (FSREGN(*nx,ny,x1,x2,y1,y2*)) RETURN
following an initial fsinit call to establish a graphics coordinate system, returns the size nx,ny and limits x1-x2,y1-y2 of a display subregion defined by the keys size, si2, position and po2 and options left, right, top and bottom; effectively, a 2-D version of tstsrg operating in the three graphics coordinate systems.

FSTEXT IF (FSTEXT(text,n,x,y,justx,justy)) RETURN
following an initial fsinit call to establish a graphics coordinate system, marks the text in text(1-n) at position x,y; the text is left-, centre- or right-justified according to whether justx is negative, zero or positive; and bottom-, centre- or top-justified according to whether justy is negative, zero or positive.

FSVIEW IF (FSVIEW(lview)) RETURN
following an initial fsinit call to establish a graphics coordinate system, sets up viewing conditions according to the defaults established by fsinit, modified by keys lut, zoom, pan and pa2 if these are set. If lview is .FALSE., fsview does nothing; its commonest use is implementing the general option view via a call
if (fsview(opt(nview)).

FSXWIR IF (FSXWIR(xinit,yinit,*x,y*)) RETURN
following an initial fsinit call to establish a graphics coordinate system, returns a single display cursor position in x,y; where it can be controlled, the cursor is made to appear first at xinit,yinit.

FT1D CALL FT1D(array,n,isign,hplane,modsq,ln)
effects a 1-D Fourier transform of the n pixel array array *in-situ*. Logical hplane indicates half-plane transform case (real image data, with central origin; right half-plane of conjugate symmetric transform, with origin at left), failing which data are complex and the origin is central in both planes. isign=-1,1 indicates forward,reverse transforms respectively. No normalisation is made in either direction. Logical modsq causes the squared modulus of the data to be taken after a forward transform or before a reverse transform, and includes an additional logarithm calculation on forward transforms if ln is .TRUE.

FT2D IF (FT2D(ito,iclass,iform,modsq,ln)) RETURN
transforms logical picture lp1 to a new picture number ito, with the given class number (which should be nclfou or nclspe) and form number (which must be nfmfp or nfmcom); the output contains the right half-plane only if the source is not complex. Logicals modsq and ln have the same meaning as they do for ft1d.

GENHST IF (GENHST(lpn,nchan)) RETURN
scans logical picture lpn (or a subregion indicated in the same way as it is for range - see §4.1), constructing an nchan channel pixel histogram in rb1(1-nchan). Imaginary parts of complex pictures are counted together with real parts. rb1 and rb2 are used as workspace.

INKEY IF (INKEY(ikey,ERROR)) RETURN
reads the next keystroke without echo and returns its internal code value in ikey. All the valid key codes are defined as parameters in the file icset.for.

INVFT2 IF (INVFT2(ito,iclass,modsq,zeroc,rnorm)) RETURN
inverse transforms logical picture lp1 to a new picture number ito, with the given class number (usually nclima) and a form controlled by the general form options (defaulting to fp if the source is a half-plane transform and to complex otherwise). Logical modsq requests an initial modulus squared calculation, and logical zeroc an initial zeroing of the central pixel. According to whether rnorm is negative, zero or positive, the resulting image is divided by its own central value (for acf), divided by its dimensions (for normal use), or multiplied by rnorm (for xcf).

IPACK INTEGER=IPACK(char)
returns its character*3 argument as a Semper packed name; a convenient alternative to providing the value as a parameter.

IVAL INTEGER=IVAL(name) = NINT(VAL(name))
returns the value of the named Semper variable, rounded to the nearest integer.

IVALPN INTEGER=IVALPN(name) = SEMPPN(IVAL(name))
returns the value of the named Semper variable, treated as a picture number (i.e. with the current device cd inserted if appropriate).

MEANSD IF (MEANSD(lpn)) RETURN
scans logical picture lpn (or a subregion indicated in the same way as it is for range - see §4.1), determining the mean and sd as well as the range; the range is recorded in the label when appropriate. The information is returned in common variables vmean, vme2, vsd, vmin and vmax. For complex pictures, a complex mean (vmean, vme2) is returned, and vsd is set to the rms modulus deviation from this.

MRKREG IF (MRKREG(idummy)) RETURN
marks, in the overlay of display picture mark, the subregion defined by smgi7-8, and smgr1-6 (see tstsrg); no initial fsinit call is needed.

OPT LOGICAL=OPT(name)
returns .TRUE. iff the named Semper variable is set with a nonzero value.

OPTNO LOGICAL=OPTNO(name)
returns .TRUE. iff the named Semper variable is set with a zero value.

RANGE IF (RANGE(iop,lpn)) RETURN
provides picture range record management facilities for logical picture lpn according to iop:

1 scans logical picture lpn (or if smgl1 is .TRUE., the subregion indicated by smgi1-6 (see tstsrg), setting vmin, vmax to the pixel range (recovering from label or recording in it where appropriate)
2 =1 but scans unconditionally, ignoring any existing record; not in fact used at present
3 deletes any existing range record from the label
4 records current vmin, vmax in the label

For complex pictures, the range is set so as to accommodate the imaginary as well as the real parts. Abandon requests during the scan cause the range of the data

scanned before the request to be provided in vmin, vmax, and .FALSE. to be returned with error set to 4.

READLN IF (READLN(itext,n,ERROR)) RETURN
reads a line of text from the terminal, with full line editing facilities provided. When the <return> key is pressed, the text is returned in the array itext(1-n) in internal code form. The size of the array is passed in the variable n and the number of characters read is returned in n.

SEMCEN IF (SEMCEN(lpn,nccol,ncrow,nclay)) RETURN
records column, row, layer nccol, ncrow, nclay as being the coordinate origin of logical picture lpn.

SEMCHS CALL SEMCHS(*char*,itext,n)
converts internal code string itext(1-n) to character variable char, truncating or padding with blanks as necessary.

SEMCLS IF (SEMCLS(lpn)) RETURN
closes logical picture lpn, i.e. marks the PCB free.

SEMCON IF (SEMCON(string)) RETURN
outputs the line of text contained in the character argument string to the console output stream. Trailing blanks are not output.

SEMCP2 LOGICAL=SEMCP2(m,n)
returns .TRUE. if m and n are not powers of 2 and at least 4, except that n is ignored if equal to 1.

SEMDEL IF (SEMDEL(iop,number)) RETURN
performs various picture deletion operations according to iop:

1 deletes the single picture number
2 deletes any 'temporary' pictures found in any disc devices
3 writes a new empty directory to device number
4 deletes picture number even if its label is malformed (which prevents wp testing); used by delete malformed as a last-resort recovery route

Deletion does not involve actual data overwriting, but directories, PCBs etc. are amended to prevent further access to the picture, and its storage is treated as available for subsequently created pictures. iop=1 refuses to delete pictures marked wp.

SEMDIA IF (SEMDIA(string,isever)) RETURN
outputs the line of text contained in the character argument string to the diagnostic output stream. Trailing blanks are not output. The type of diagnostic message is indicated by isever (the parameters ndimes, ndiwar, ndierr and ndifat are provided for this purpose).

SEMDPD IF (SEMDPD(iop,number)) RETURN
reads/writes (iop=1/2) the DPD for display picture number to/from the common block semdpd.

SEMEXP IF (SEMEXP(itext,length,iptr,*value*,scanmd)) RETURN
returns in value the value of an expression beginning at element iptr of character array itext (dimension length). The pointer iptr is advanced beyond the expression (or beyond the buffer if this becomes exhausted). Iff logical scanmd is .TRUE., the expression is scanned but not evaluated (value is returned as 0.); this mode is used by the command interpreter, but is not likely to be useful in processing modules.

SEMFAC LOGICAL=SEMFAC(n,*ifacs*,*nfacs*)
breaks n into factors of 4, 5, 3 and/or 2, in that order, returning .TRUE. if not factorisable in at most 8 factors, with at least one factor 4; number of factors found returned in nfacs, and factors in array ifacs.

SEMFRM INTEGER=SEMFRM(iform)
returns the corresponding form number (0-3) if option byte, integer, fp or complex is set, and iform if they are not.

SEMICS CALL SEMICS(char,itext,n)
converts character variable char to internal code string itext(1-n), truncating or padding with blanks as necessary.

SEMINP IF (SEMINP(string)) RETURN
outputs the line of text contained in the character argument string to the logical output stream devoted to non-command input text. Trailing blanks are not output. All non-command input should be output via this routine, as this ensures that every detail of an interactive session can be logged.

SEMKTX IF SEMKTX(name,'prompt',*iout*,lenout,ucflag)) RETURN
returns in iout(1-lenout) the character string value for key name, forced to upper case if logical ucflag is .TRUE.; on entry, lenout must contain the maximum length of iout; if name is unset, semktx issues the supplied 'prompt' and takes the value from the terminal, handling any errors in the user's response internally; a blank string entered results in a return with lenout zero.

SEMLAB IF (SEMLAB(iop,label,lpn)) RETURN
reads/writes (iop=1,2) the label for logical picture lpn to/from the integer array label. It refuses to write tape labels. Great care is needed in making any alteration to picture labels, as mistakes could cause considerable data loss.

SEMLOG IF (SEMLOG(string)) RETURN
outputs the line of text contained in the character argument string to the log output stream. Trailing blanks are not output.

SEMLU LOGICAL=SEMLU(iop,name,value)
performs various operations on the Semper variable with packed name name, according to iop:

-1 returns .TRUE., with value set to value, if it is set ; else .FALSE. with value zero.
0 unsets the variable (unless protected or read-only); returns .FALSE. always.
1 sets the variable to value (unless read-only), entering it in the table if necessary; returns .TRUE. only if table is full.
2 =1, but the setting is *local*, i.e. the original state/value is restored at the end of the current command; used for key/option setting by the interpreter
3 records the current state/value of name for restoration at the end of the current command, without changing the current state (and ignoring value); effects a local command for the variable

SEMOPN IF (SEMOPN(iop,npic,ncol,nrow,nlay,iclass,iform,*lpn*)) RETURN
performs various picture opening and directory / label / DPD management operations according to iop:

1 opens *old* (i.e. pre-existing) picture npic, returning the assigned LPN in lpn, and dimensions / class / form details in ncol.. iform
2 opens *new* picture npic, with dimensions / class / form as given inncol.. iform, returning the assigned LPN in lpn. If lpn is non-zero on entry, semopn takes it to designate a picture from which a title (and coordinate origin, if the dimensions are the same) are to be taken for the new picture.
3 opens *temporary* picture (npic ignored), on any writeable disc device, with details as for iop=2; the picture is deleted at the end of the current command.

rb1, rb2 and rb3 are liable to be used as workspace; logical lblinc in common indicates on exit whether the picture label has been left in rb1 (in integer form) - it is only some *old* mode tape opens where backspacing is not supported for which this is not possible. Attempts to open a *new* tape picture return error=27, but leave a suitable label in rb1.

SEMPPN INTEGER=SEMPPN(n)
'Processes Picture Number' n, returning n itself if n>1000 but adding 1000*cd otherwise.

SEMPRG CALL SEMPRG(*itext*,*last*,commnd,'prompt')
reads a line from the terminal (common variable input = term1) or the run file (input = runfle) into array itext, setting last pointing to the last non-space in the line,' (to 0 if all characters are spaces) and converting to internal code. If logical commnd is .FALSE., a single line of up to 80 chars is accepted; otherwise the supplied 'prompt' is issued at the terminal (with a space at either end, the first providing a carriage control character) and continuation line processing is performed. NB: Program text is not read via semprg but via semlin instead.

SEMPRO IF (SEMPRO(prompt)) RETURN
outputs to the terminal the text in the character argument prompt as a prompt string. Trailing blanks are output and the output of any carriage return/line feed

sequence is suppressed.

SEMROW IF (SEMROW(iop,buffer,iform,irow,layer,lpn)) RETURN
reads/writes (iop=1,2) row irow of layer layer of logical picture lpn to/from the array buffer. iform indicates the form of buffer; semrow converts to/from it as necessary while transferring data. Semrow faults protection violations, buffer overflows, and row/layer number overflows.

SEMTFC LOGICAL=SEMTFC(lpn,*hpl*)
tests the recorded origin for a Fourier transform picture, indicating via returned logical hpl whether it is a half-plane or full-plane transform, and returning .TRUE. if the origin is in neither of the acceptable positions.

SEMTYP IF (SEMTYP(in,lenin,iptr,*iout*,lenout,scanmd)) RETURN
evaluates a type-item list starting at element iptr of in(lenin), returning result in iout(1-lenout); on entry, lenout gives the maximum length of iout. Overlength result strings are simply truncated without a .TRUE. return. iptr is advanced beyond the text (beyond in if the source is exhausted) and points to the problem if .TRUE. is returned. Iff logical scanmd is .TRUE., the text is scanned but not evaluated; this mode is used by the command interpreter, but is not likely to be useful in processing modules.

SEMVW IF SEMVW(idev,ifr,lut,izoom,ix,iy)) RETURN
provides a routine controlling viewing conditions at a lower level than fsview; semvw sets conditions that present position ix,iy (display coordinates) of frame ifr of device idev (which must equal fs at present), with zoom factor izoom and through lut lut.

SEMXA1 IF (SEMXA1(iop,itext,length,iptr,value,ivalue)) RETURN
reads/writes names/numbers from/to element iptr onwards of character buffer itext (dimension length), advancing iptr beyond the item (or beyond itext if the buffer is exhausted), and ignoring leading spaces when reading. The available values of iop are:

0 reads next non-space from itext(iptr..) to ivalue
1 reads a name from itext(iptr..) to ivalue (processing indices)
2 reads a number from itext(iptr..) to value
3 writes a name from ivalue to itext(iptr..)
4 writes a number from value to itext(iptr..)
5 reduces length so as to strip trailing spaces from itext

iop = 0 to 4 return .FALSE. unless itext overflows before the item can be transferred; iop = 5 returns .FALSE. unless length reaches zero. Bad subscript values in a name cause iop=1 to return .FALSE. with error non-zero, so name reads should normally be followed by an error check. Initial spaces are ignored when reading; iop = 0 advance iptr up to the item; iop = 1 to 4 advance it beyond the item (to length+1 if the buffer overflows).

TAPE IF (TAPE(iop,iform,idev,ifile,iblock,number,buffer)) RETURN
positions tape device idev to file ifile block iblock, and makes transfers between the tape and the array buffer, according to iop:

1 reads block of up to number items, resetting number if actual block is shorter
2 writes block of number items
3 writes end-of-file mark
4 returns without transfer (used for positioning only, e.g. rewinding)
5 verifies idev legal and assigned to tape

iform indicates the form of the data in buffer, and is only used to establish the appropriate block length in bytes.

TEXTU1 LOGICAL=TEXTU1(text,n,maxn,iold,nold,new,nnew)
replaces nold elements beginning at position iold of text(1-n) by new(1-nnew); adjusts N to revised length, and returns .TRUE. only if this would exceed maxn; used for character string substitutions.

TSTSRG IF (TSTSRG(iop,lpn)) RETURN
establishes a subregion of logical picture lpn via the standard 2-D and multi-layer keys and options; full details are given in §4.1, and are not repeated here.

VAL REAL=VAL(name)
returns the value of the named Semper variable, or zero if it is unset.

VARINT LOGICAL=VARINT(name) = VAL(NAME).EQ.AINT(VAL(NAME))
indicates whether the named Semper variable is integral (or zero).

VARSET LOGICAL=VARSET(name) = SEMLU(-1,name,*x*)
indicates whether the named Semper variable is set.

Appendix 2. System Data Structures

The information in this Appendix is highly condensed and unlikely to be of use to the inexperienced, but provides reference information that may be found valuable ultimately. The addition of *4 to a name means that it is explicitly declared INTEGER*4 in the source code.

A2.1 Permanent structures

Disc file structure

header block, directory block(s), data blocks

Header block contents

bytes 1-13 S e m p e r . d i s c version release *or*
S e m p e r . h e l p version release *or*
S e m p e r . t e x t version release
14-16 File size in blocks (I*3, most sig byte first)
17-18 Directory size in blocks (I*2, most sig byte first)

Directory structure

Help libraries: not described here (see source code if available)
Program Libraries: not described here (see source code if available)
Picture discs: a series of 2-element segment descriptors (*SDs*) with the following contents:

1 contents: 0 => segment free; 1-999 => segment contains picture indicated; negative => segment contains temporary picture (awaiting deletion); 1000 => last used SD in directory
2 starting address (first disc block number); disc size + 1 for last used SD

The entire directory or SD list fits within one row buffer.

Disc data

Help Libraries: not described here (see source code if available)
Program libraries: not described here (see source code if available)
Pictures: consist of a 256 byte, block-aligned, label followed by the data for the picture rows in turn; all block aligned; rows first, and then layers, stored consecutively.

Work disc

The work disc (device 0) does not in fact survive between sessions, and does not have the header-directory-data structure that other disc files have.
Blocks 1+ndevs are used to hold the names of the currently assigned devices, each block-aligned, in i/c byte form, with a preceding byte char count
The next group of blocks form an effective display picture directory, holding a series of ndpds entries each consisting of a block-aligned display partition descriptor (*DPD*) followed by a block-aligned picture label; the DPD contents are:

dpmin,dpmax	(reals) grey-scale min,max
dplef,dprig	(reals) x picture coordinates of display left,right
dpbot,dptop	(reals) y picture coordinates of display bottom,top
dpma,dpmb	(reals) x coord. map parameters: xdis = dpma * x + dpmb

dpma2,dpmb2	(reals) y coord. map parameters: sim.
dptyp	display type (1,2,3,4 => grey-level, graph, histog, ymod; -1,0 => undef partn, empty partn)
dpfra,dpfra2	first,last frame numbers (same unless partition contains multi-layer picture)
dptlx,dptly	x,y display coordinates of partition top left
dpsiz,dpsi2	partition dimensions (display coordinates)
dpimo	display coordinate offset of imaginary part (0 => no imaginary part)
dpsrc	if nonzero, picture number whose coordinate system is mapped on to display
dpdev	display device number (currently always 1)
dpnum	partition number
dplut	default lut number to be used for viewing partition

The next group of blocks provides storage for nluts look-up tables, each large enough to hold 3*lutlen integer arrays.

The final blocks provide back-up and stack storage used by the command interpreter in connection with program calls, for loops etc.

Picture labels

have the following (byte) contents:

1-6	identifying chars S e m p e r
lbnc1,2	row length (lbnc1*256+lbnc2)
lbnr1,2	number of rows (sim.)
lbnl1,2	number of layers (sim.)
lbcc1,2	origin column number (sim.)
lbcr1,2	origin row number (sim.)
lbcl1,2	origin layer number (sim.)
lbclas	class number
lbform	pixel form number
lbwp	write-protect flag (protected if non-zero)
lbyear,mon,day	(year-1900)/month/day created
lbhour,min,sec	hour/min/sec created (these six stored consecutively)
lbncrr	number of chars in encoded range; 0 => no range recorded
lbrr1-2	picture range encoded as i/c decimal chars, with i/c comma separating; follows lbncrr immediately
lbplty	type code (1/2/3) for position lists only
57-99	reserved/unused
lbnctt	number of chars in title; zero => no title
lbtt1-2	title chars (i/c); follows lbnctt immediately

Label items are accessed by using the above parameters to subscript an integer array into which the label has been read and re-formed by semlab.

A2.2 Memory-Resident Structures

Device Control Blocks

apply to all devices other than the work disc (device 0); they consist of a set of corresponding elements from several integer arrays, subscripted by the device number, as follows: