ACHIEVING PORTABILITY IN IMAGE PROCESSING SOFTWARE PACKAGES

S. W. Krusemark
Department of Electrical Engineering, Virginia Polytechnic Institute and State University, Blacksburg, VA 24061 USA

R. M. Haralick
Department of Electrical Engineering, Virginia Polytechnic Institute and State University, Blacksburg, VA 24061 USA

Abstract. The first key to portability is in the use of a kernel of routines that interface to the peculiar operating system of each machine. The kernel provides sophisticated but standard operating system services required by the image processing software. It makes the operating system of each computer appear identical and, nicest of all, when carefully designed it does not pose a difficult implementation problem. Above this interface, all image processing applications programs can be machine-independent, written in a structured language such as RATFOR, without sacrificing power or ease of use on any machine.

1.0 INTRODUCTION

Transportability is achieved by writing image processing applications programs that obtain all operating system services from the standardized operating system interface. Moving to a new machine then requires only the implementation of the kernel supporting the standard calls.

1.1 Footnotes

Some of the research that went in to the kernel was funded by NSF grants to VPI&SU, University of Maryland, and Rensselaer Polytechnic Institute. Detailed technical discussion of the overall set of conventions described here can be found in Hamlet and Haralick (1981) or Hamlet and Rosenfeld (1979) and detailed discussions of the kernel conventions can be found in Krusemark and Haralick (1981) or Guerrieri (1981).

1.2 Operating system interface

Operating systems all support the capabilities for file operations, memory allocation, process control, input/output and interrupt handling. Machine independence can be achieved by obtaining these services only through prespecified subroutine calls that remain the same from computer implementation to computer implementation. This collection of entry points constitutes a kernel that makes all operating systems look the same to the image processing package.

Three factors shape the definition of the kernel:

1. The services must be powerful enough and easy to use.
2. The kernel must be capable of being built around any existing operating system.
3. The implementation of the kernel for a new system must be easy for a local systems expert.

The first two factors tend to increase the size of the kernel; the last limits its size.

In an operating system whose services are nearly the same as those of the interface, the kernel would be only a calling-sequence converter, transforming the subroutine calls into monitor calls with some additional error checking added. When the system provides very different services, however, the code may be as large as a few thousand lines of code. Section 2.0 discusses the operating system interface and section 3.0 discusses the image I/O primitives that sit on top of the operating system interface.

1.3 Command and process interface

An image processing package must communicate with its users, then carry out the tasks they specify. The jobs of scanning commands and interacting with a user are unlike the processing that takes place after

the command is decoded. This job can be accomplished in a command processing module.

In the command-processing module, all interaction with the user results in a standardized description of a processing request. Any routine may later use this information without concern for details of format. Furthermore, the information is checked once and for all at the beginning. For example, a file may be required to exist, have a certain format, etc. These matters can be interactively straightened out with the user before the information is stored for the next module. The processing module that then deals with the standardized request can be conventionally linked to the command routines, or overlaid.

Unfortunately, the general overlay organization is an unsatisfactory mechanism because the implemention capabilities differ widely in different computers. This is a fundamental transportability problem. Our approach to the overlay problem is to make a simple kind of overlay a standardized operating system service called "program exchange," in which the executing code calls for itself to be replaced by another ready-to-execute program module. This simple overlay mechanism is supported by every modern operating system.

1.4 Image I/O primitives

The image I/O routines can be entirely written in a high level language like RATFOR. They format the image data appropriately, and call upon the kernel I/O routines to accomplish the data transfers to and from the image file. The format for an image must be general, allowing images to be any practical size and any number of bands. Logical records can be subimage blocks of arbitrary number of rows and columns. The most common format would have logical records correspond to

one image row. Since image data often does not have high precision, the data can be packed. This bit packing occurs inside the image I/O routines and is completely transparent to the programmer. Section 3.0 discusses the details of the image I/O routines.

1.5 RATFOR

RATFOR is a structured FORTRAN language that has the constucts: IF THEN ELSE, REPEAT UNTIL, WHILE, and blocks of code. Thus an IF statement can execute more than one statement such as:

```
IF( A > B )
   [
   A = B
   B = C + B
   ]
ELSE
  [
  B = A
  A = C + A
  ]
```

Another example is the DO loop:

```
DO J = 1, 20
    ARR(J) = 0
```

to zero out an array of 20 positions. Note that unless otherwise blocked ([]) the object of a DO, IF, or WHILE is a single statement.

RATFOR permits the use of symbolic names instead of numeric quantities. These symbolic names are also helpful mnemonics for the programmer. An example is the use of .OLD instead of the integer 1 as the parameter in an open call to indicate that an old file is to be opened. The RATFOR preprocessor translates .OLD into a 1 so that the output is FORTRAN compatible. The use of the symbolic names instead of the numeric value is accomplished by the RATFOR DEFINE statement. Thus .OLD is translated to the numeric 1. A CHARACTER statement can be

implemented by translating CHARACTER to INTEGER, for example. A table of such DEFINEs can be built that define system wide constants and by using the INCLUDE statement in RATFOR, these DEFINEs can be put in every file on the system. All uses of such symbolic names in this paper start with the period for clarity.

The INCLUDE statement in RATFOR is a mechanism to merge one file into another. This, for example, allows a single COMMON to be defined in an INCLUDE file. If a change is necessary the INCLUDE file is changed and the system is recompiled. Without the INCLUDE one would have to find ALL occurrences of the common and change them (hopefully finding them all) and then rebuild the system. The INCLUDE file can be used for tables of DEFINEs so that machine constants can be changed quickly, for commons, and even sometimes for code.

2.0 OPERATING SYSTEM INTERFACE CONVENTIONS

The operating system services provided by the kernel include program exchange, random sequential file handling, memory allocation, and user interrupt handling. As discussed in section 2.1, we suggest each kernel subroutine be an integer function subprogram whose value indicates a completion condition code. This facilitates error handling. Section 2.2 discusses program exchange, section 2.3 discusses file operations, section 2.4 discusses memory allocation subroutines and section 2.5 discusses interrupt handling.

2.1 Error handling

When experienced users make use of well-tested software, error conditions arise only occasionally. But while the users are learning, or the software is under development, most processing is error handling. In providing error returns, a set of subroutines should make

it easy for the calling program to deal with complex error situations, yet at the same time the overhead should be low. Simple situations should not require the caller to make use of all aspects of the error mechanism. For software development, there is another important factor: it must be easy to quarantee that every error is detected, even those that cannot occur once the software is working properly.

To take care of error processing, each kernel interface routine is an INTEGER FUNCTION. If there are no errors, the routine returns a non-negative number (whose value may have significance as a part of normal processing). However, if there are errors to report, the routine delivers a negative value indicating the type of error.

In the sections that follow, this error mechanism is described in full only for the first routine (OSCHAN in section 2.2).

2.2 Program exchange

At any time during execution, a program can terminate by calling for its successor. The routine accomplishing this is:

```
INTEGER FUNCTION OSCHAN(PROGRAM)
```

where PROGRAM is a string of characters identifying the new program. The only potential failures for OSCHAN involve the nonexistence of PROGRAM.

The error handling works the following way. Let us suppose that the code -3 is assigned to the file error that the named file does not exist. (Perhaps -1 means that there was a read error, -2 is end of file, and so on.) The caller may anticipate that the new program may not exist, so that

```
IEV = OSCHAN(PRG)
IF (IEV == -3)  # DOESN'T EXIST
   WRITE(6, 1)
1 FORMAT(' NO SUCH PROGRAM')
```

is a call in which the anticipated error is explicitly processed.

All routines in the kernel treat errors in this way, but the details will be suppressed in the discussions to follow. This example is typical of the presentation in other ways: the code is written in RATFOR, and little attention is paid to FORTRAN conventions about variable names, in the interests of clarity.

2.2.1 Parameter passing for program exchange. When passing control from one program to another parameters are needed to allow information to be passed as well as control. These parameters could be passed by the user in a program dependent way but since many machines do have fast parameter passing capability and a transportable version of these routines can be written, the routines OSSEND and OSRECV were created. The two routines to send and receive information through a program change are:

```
OSSEND (PARAMETERS,LENGTH)
OSRECV (PARAMETER,READLENGTH,BUFFERLENGTH)
```

The send and receive are capable of repeated use to allow several sets of parameters to be sent. This makes passing several arrays of numbers, file names, etc. possible without the need to recopy them into a single rather long array. An example is to send two arrays with the new program name in PRG

```
OSSEND (ARRAY1,50)
OSSEND (ARRAY2,5)
OSCHAN (PRG)
```

The receive end looks slightly different because the send and receive operate like a push-down stack (first-in-last-out) so:

```
OSRECV (ARRAY2,LEN2,5)
OSRECV (ARRAY1,LEN1,50)
```

(Error checking on returned length can be done.)

The push-down stack concept allows layers of code similar to subroutine calls and returns. Each exchanged program reads in only the last series of arrays sent by OSSEND.

2.3 File operations

Because file operations differ greatly from system to system, it is necessary to duplicate most of the operations performed by the local operating system (and the input-output control system, if there is one) in the interface. Fortunately, most of the coding can be done in RATFOR, and much of it carries over from machine to machine.

Files can be random access files or sequential files. The operations which programs need to perform on such files are SETUP, OPEN, READ/WRITE, and CLOSE. The first, second, and last could be done by the transfer operations, but they are needed for certain special actions, and they always provide useful error control. Furthermore, an OPEN operation can be used to verify the existence of a file, and to acquire its present characteristics, even if no transfers are contemplated. Similarly, a CLOSE operation can perform an action like deleting the file.

The complete file name is stored in an array (one character per word), and passed to a standard interface routine: OSINFD. This routine moves the name into an array called FILE. The array FILE, is called a file descriptor, and is then passed to each system interface routine. The routine looks like:

```
OSINFD(FILENAME,FILE)
```

Space is provided in the array FILE for the routines to assign and maintain some kind of internal description invisible to the user. The array parameter containing the file descriptor is designated FILE in each of the following routines to be described.

A new random file descriptor must be initialized with the correct size information before the open. This is the job of OSPINF whose calling sequence is:

OSPINF(FILE,ATTRIBUTE,VALUE)

This routine puts the (ATTRIBUTE,VALUE) pair into the file descriptor array FILE. The .MODE (record mode) attribute determines whether the file is treated as an integer, real, etc., file. The .LREC (record length) attribute determines the length of each record. The .NREC (number of records) attribute determines for random files the number of records in the file. These are the most commonly used options. The other attributes will not be discussed due to space limitations. An example is to set number of records:

OSPINF(FILE, .NREC, 60)

The OPEN of a random file is done by a call to OSOPNR:

OSOPNR(FILE, TYPE)

where TYPE can be .NEW or .OLD. TYPE is .NEW to create a new file and .OLD to open an old file. To open a new file, the file descriptor must have the required number of records and record length. To determine the number of records and record lengths of an old file, the routine OSGINF is available. OSGINF is the reverse of OSPINF.

No two calls to OSORNR (without an intervening OSCLOS) may name the same file, with one exception: using different arrays for the name, a file may be opened once .OLD and once .NEW at the same time. The intent is to allow a copy-and-update operation that does not happen in place.

All files must be closed since unclosed files may not necessarily exist after a program exchange. OSCLOS provides the close service:

OSCLOS(FILE,OPTION)

The action of the close depends on the TYPE of open and the OPTION:

OPTION	TYPE	ACTION
.DELETE	.OLD	The existing file is deleted. Any open NEW files are not touched.
.DELETE	NEW	The new file is deleted. This may leave an old copy (which may or may not be also open at the time).
.KEEP	.OLD	The existing file is closed.
.KEEP	.NEW	This is where any problems that will occur can occur. The disk is checked for a pre-existing file. If one exists, it is deleted or otherwise removed from consideration. The NEW file is closed so that the file replaces the pre-existing file. If a pre-existing file does not exist, then the new file is closed such that it appears with the correct names etc.

The actual data transfer to or from open files is done with:

```
OSRDR(FILE,RECORDNUMBER,BUFFER,READLENTH,WAIT)
OSWTR(FILE,RECORDNUMBER,BUFFER,WRITELENGTH,WAIT)
```

The routines read from and write to the file having FILE for its file descriptor. In the BUFFER array is the data. The mode (INTEGER, REAL, etc.) is determined by the mode specification of the last call to OSPINF. Since the read and write lengths can be other than a full record, the RECORDNUMBER is the starting record number (the first record is one). If the length is greater than a single record, then multiple records are transferred. If the length is shorter than a full record, the extra data written is garbage on a WRITE and on READ it is ignored.

If the WAIT parameter is .WAIT, return does not occur until the operation is complete. If the wait parameter is .NOWAIT, then the

data transfer is started but control returns to the calling program immediately allowing the program to continue until the data is actually needed, at which time

OSWAIT(FILE,WAIT)

is called. The WAIT can be used again if the program wants to only check if the operation is done.

The last operation specific to the random files is the ability to change the number of records on a random file.

OSGROW(FILE, NREC)

allows the NREC (number of records) to be changed on a random file. There are two ways to implement this depending on the machine: 1) to copy the file, necessary on machines that cannot grow files, and 2) to simply let the file get bigger. If the code is implemented using OSGROW rather than the machine dependent capability of making a file can change its size then transportability is enhanced.

Sequential input/output is more complicated because the kernel handles files and devices such as terminals, printers, magnetic tapes, etc. The first call is to OSINFD which creates the file descriptor. The second call is to OSPINF and sets the characteristics of the operation. Then the file is opened by a call to:

OSOPNS(FILE,TYPE)

The variable TYPE can take the value .INPUT or. OUTPUT.

The same as specified for the Random files can happen for a .INPUT and .OUTPUT opened with the same name. The operation of .INPUT is like .OLD and .OUTPUT is like .NEW when the close is done. Devices are specified by special reserved file names such at TT for terminal and PRINT for line printer.

The reads are done by:

OSRDS(FILE,BUFFER,ACTUALLENGTH,BUFFERSIZE)

Since in many cases the calling program will not know the actual length until after the read is done, the amount of space available is specified by BUFFERSIZE.

The write is done by:

```
OSWTS(FILE,BUFFER,LENGTH)
```

Since the data length is known, this is the simplest of the routines.

2.4 Memory allocation

Dynamic memory allocation can be difficult because it involves manipulation of addresses. To solve this in a transportable system, the mainline code for each image processing operation becomes a subroutine:

```
SUBROUTINE OSMAIN(DYNARRAY)
```

The programmer writes this routine as if it were the main program, but is provided with an argument at the outset. The parameter is an array that may be passed about among the routines of the package, and which has been set up by the calling side of OSMAIN so that it can change in size up to some set internal maximum. Initially, DYNARRAY will have one element, but following a call to

```
OSALOC(SIZE)
```

it will be as if it were dynamically altered to

```
INTEGER DYNARRAY(SIZE)
```

OSALOC changes the size of the dynamic array DYNARRAY that was passed to the user via OSMAIN. The user requests the number of integer words (SIZE) that is needed and the routine OSALOC checks to see if this is available. If it is, a value of .OK is returned, otherwise a negative value is returned. This array is intended to be the main user work area.

This dynamic allocation can be implemented in two different ways. The first is very simple and easy to do: simply allocate a very large array and pass it to OSMAIN. Then OSALOC simply checks if the request is for too much memory. The second is to set the base of the array at the end of the user memory space. Then OSALOC extends user memory by the requested amount. Thus the user only pays for what is used.

2.5 Interrupt handling

As an example of the use of an interrupt service, imagine the problem of terminating unwanted output. Most systems have some means of alerting a running program to a user "attention" typed in, implemented as an interrupt. If the user invokes this during output, the interface main program will take control and then call OSINTR. OSINTR sets a flag which the print routine tests before each line, and returns in the "dismiss" mode. Printing terminates as soon as that flag is tested and found to be present. After printing is stopped the package program proceeds normally.

The same initial system main program (described in the last section) can set up interrupt processing. It can set up the system interrupt to branch to a particular location. Then, should an interrupt occurr, it can call the user supplied routine:

OSINTR(TYPE)

TYPE describes the problem that caused the interrupt so that the processing in the package can be intelligent. By providing two kinds of returns from OSINTR, the package can "dismiss" the interrupt normally and continue (perhaps after taking corrective action); or, it can specify a "restart" in which the main program again calls subprogram OSMAIN, and thus cuts short the interrupted code forever. If OSMAIN is written to test a global flag, it can know whether it is starting or restarting.

3.0 IMAGE I/O ROUTINES

The image I/O primitives take care of file I/O for image data files. They permit image data to be handled at one logical level higher than the I/O provided by the kernel. The image I/O routines can be written as entirely portable code and include an open, a read, a write, and a close. Haralick (1977) discusses the image access protocol conventions described here. The open (RDKINL) is passed an array containing values some of which must be initialized for opening a new file and others of which do not have to be. This array is called the IDENT array. On opening an old image file the image files IDENT array is returned. This array is accessable to the user and is passed to the read and write routines (RREAD and RWRITE).

3.1 IDENT array

The IDENT array must have certain specified parameters defined in order to be legal for opening a new image file. The IDENT array is always zeroed first and so a 'not set' value is zero. The values that must be set are the logical size of the image, and its mode (integer, real, double integer, or double precision). If it is an integer image then either the number of discrete levels of grey shades, the minimum/maximum grey-tone values, or the number of bits to hold the grey-tone shades must also be set. Other optional parameters include the subimage block size (if different from an image row), the number of bands in the image, and the number of symbolic bands. If not set these optional parameters default to reasonable values.

3.2 Error handling

Error handling for the image I/O primitives as well as for all machine independent code uses an event variable IEV and alternate return %XXXX mechanism. The argument %XXXX is an alternate return as defined by FORTRAN, the XXXX standing for a statement number such as 1234. When no errors are generated in the routine a normal return is taken and processing continues at the statement following the call. If, however, an error occurrs the IEV event variable is set and the alternate return is taken and processing continues at the statement with the label XXXX given in the call.

3.3 Random file open

The subroutine RDKINL (Random Disk Initialize) performs the random file open for both new and old files.

```
CALL RDKINL( FD, IDENT, OLDNEW, IEV, %XXXX )
```

The IDENT is the identification array as described earlier. The argument OLDNEW has one of the two values denoted by the symbolic definitions .NEW and .OLD, for new and old files, respectively, to be opened.

On a new file the values in the IDENT array are checked for consistency and unspecified values are initialized. Then a random file of the correct size is created and the IDENT array is written to the first record of the file.

On an old file the routine first checks for its existence on the disk. If it does not exist an error is generated and the alternate return is taken. If it does exist it is opened and the first record is read into the IDENT array.

3.4 Random file read/write

The routine RREAD takes the data on the image and copies it to the buffer and the routine RWRITE copies the data in the buffer to the image, unpacking and packing where needed.

```
CALL RREAD
  ( FD,BUF,BND,BLKNO,IDENT,WAIT,IEV,%XXXX )
CALL RWRITE
  ( FD,BUF,BND,BLKNO,IDENT,WAIT,IEV,%XXXX )
```

Since image access is random, the two arguments BND and BLKNO are needed to get the correct band and block number from the SIF file. Blocks are numbered starting in the upper left corner and proceed down the left side to the bottom of the image. The word block corresponds to a subimage or logical record.

The buffer (BUF) should be dimensioned for the correct size of one block. Only one block can be obtained at a time. However, if a value of zero (0) is specified for the band number argument (BND) the block specified is returned for all bands. Normal use is to specify the band number and block number thus returning only one block from one band. Also, the most common shape of a block is a row of the logical image. The argument WAIT has a value indicating whether or not the I/O must complete before returning.

3.5 File close

CLOSE compliments the open, in this case RDKINL.

```
CALL CLOSE( FD )
```

All files must be closed. The subroutine CLOSE can be used to close any file descriptor. (It is not an error to close an already closed file, but all files must be closed before exiting or they may cease to exist as expected.)

3.6 Use and example of image I/O routines

The following example will help to clarify the order and use of the image I/O routines.

The RATFOR INCLUDE, MACA1, defines the standard system-wide set of symbolic definitions is the first line of "code". The file descriptors FDI, FDO are for input and output name information. The data BUF is passed from above so that the calling program needs to make it large enough to do what needs doing.

The code first opens the input file with RDKINL, sets up the output image sizes, and opens the output also with RDKINL.

The next section is the actual algorithm which in this example simply copies the input to the output.

After the algorithm is finished for all bands and all lines then the files must be closed, and finally a return to the calling program is made.

Errors are checked such as the check that the number of points per line is less than or equal to the buffer size. Errors once detected are dealt with at the bottom of the routine in the manner shown. This starts the alternate return chain.

```
 INCLUDE MACA1
#
 SUBROUTINE NUMBCH( FDI, FDO, BND, NBND, LBUF,
                    THRSH, IEV, * )
#
 REAL THRSH
 CHARACTER FDI( .FDLENGTH ), FDO( .FDLENGTH )
 INTEGER BUF( LBUF ), IDENT( .IDLENGTH )
 INTEGER OBND, IBND
 INTEGER JDENT( .IDLENGTH ), BND( NBND )
#
 CALL RDKINL( FDI, IDENT, .OLD, IEV, %9000 )
#        open input file and temp file
#
 IF( IDENT( .IDNPPL ) > LBUF ) GOTO 9010
#       check if too large an image
#       set up output IDENT record
#
```

```
  DO I = 1, .IDLENGTH
     JDENT(I) = 0
#
  JDENT( .IDNPPL ) = IDENT( .IDNPPL )
#         output number of points per line
#         same as input file
#
  JDENT( .IDNLINS ) = IDENT( .IDNLINS )
#         output number of lines = input
#
  JDENT( .IDNBITS ) = 8 # number of bits is 8 bits
#         ( 0 to 255 )
  JDENT(.IDNBNDS ) = NBND
#
#         default to 1 band, row format
#         image INTEGER image.
#
  CALL RDKINL( FDO, JDENT, .NEW, IEV, %9000 )
#
#         This section of code is the 'real'
#         image processing.
#
#         This example is simply a copy from
#         input to output
#
  NLIN = IDENT( 7 )
#
  DO OBND = 1, NBND
     $(
     IBND = BND( OBND )
     DO LIN = 1, NLIN
         $(
         CALL RREAD( FDI, BUF, IBND, LIN, IDENT,
                     .WAIT, IEV, %9000 )
         CALL RWRITE( FDO, BUF, OBND, LIN, JDENT,
                      .WAIT, IEV, %9000 )
         $)
     $)
#
  CALL CLOSE( FDI )          # close input file
  CALL CLOSE( FDO )          # close output file
#
#
  RETURN
#
  9000 CONTINUE
#
#             Error in lower routine.
#             IEV already set.
  GOTO 9999
  9010 CONTINUE
#
  IEV = -3013    # buffer smaller than data to go
                   in it.
```

```
#
 9999 CONTINUE
 CALL CLOSE( FDI )  # even on error must close
                      all files
 CALL CLOSE( FDO )  # input and output
#
 RETURN 1           # take alternate return
#
 END
```

4.0 OMITTED CONVENTIONS

Since the kernel is a complicated concept it is difficult to fully explain in the small amount of space available in this paper. The full technical discussion is available in Krusemark and Haralick (1981). Also left out of the discussion in this paper is the concept that devices such as terminals and printers are basically a form of sequential file. The undiscussed sequential I/O primitives OSRDS and OSWTS allow terminal graphics as well as the normal mode of terminal interaction.

One thing that some software packages do not do, but which is important to do is a form of history keeping. This record keeping should be maintained in the standard image file so that any and all steps that a particular image has been through can be recorded. Because of space limitations we can not describe here the conventions we suggest for these routines.

5.0 GIPSY

There is a system called GIPSY (General Image Processing System) at VPI & SU that uses these routines and although it has been in existence a short while we have already transported over 100,000 lines of code from an IBM 370/VM running CMS to a VAX 11/780 running VMS in about one man week. It took less than seven man weeks for a programmer initially unfamiliar with the VAX 11/780 to write the kernel. GIPSY currently has over 200,000 lines of code.

6.0 REFERENCES

Guerrieri, E., Software/O.S. Interface Kernel User Manual, Preliminary Version, Techincal Report, IPL, Rensselaer Polytechnic Institute, March 1981.

Hamlet, R.G. and R.M. Haralick, Transportable "Package" Software, Software Practice and Experience, to appear.

Hamlet, R.G., and A. Rosenfeld, Transportable Image-Processing Software, Proc. Nat. Computer Conf., Vol 48, AFIPS Press, June 1979, pp 267-272.

R.M. Haralick, Image Access Protocol for Image Processing Software, IEEE Transactions on Software Engineering, SE-3 (1977), pp. 190-192.

Krusemark, S.K. and R. M. Haralick, Operating System Interface, Technical Report, SDA 81-1, VPI&SU, April 1981.

1982