Transportable image-processing software*

by R. G. HAMLET and A. ROSENFIELD

University of Maryland
College Park, Maryland

INTRODUCTION

The computations of image processing, like those of many technical disciplines, require substantial programs to perform. These programs are often organized into "packages" with the intent of making them easy for the (computer) novice to use. Access to a package of programs is an important resource, since its creation is beyond the capabilities of all but a few research groups. Unfortunately, while packages are invaluable, they could often be improved in the following ways:

1. They could be easier to use. The intellectual task of communicating with the package is too difficult, the commands too peculiar and errors too easy to make. When things go wrong, very little help is available.

2. They could make more efficient use of the underlying machine and its operating system. A package may use very sophisticated algorithms for its discipline-oriented operations, while at the same time using the most cumbersome mechanisms for controlling the resources of the machine. Its authors are seldom systems programming experts.

3. They could be easier to move from machine to machine. In the process of getting the package to work at all, many peculiarities of the programming language (in its local implementation) and the local system become entwined in the code and getting it to run elsewhere may be difficult or impossible.

4. They could be easier to understand, modify and extend. To add a new routine or alter the behavior of an existing one may not be too difficult for the program's author, but for others it may be impossible. If many changes are made independently, combining them without conflict is difficult.

Improvements in package programs must resolve the conflict between quality (Items 1, 2 and 4) and transportability (Item 3). Here, "transportability" is to mean more than the ability to export software from a development site to many others. We have in mind a software system moving freely among research groups using a variety of machines, in which modifications arise simultaneously in several places. In this situation the structure of the programming package is all-important—it must be rigid enough to support changes that conform to the style of the original; yet, changes must be easy to make. In the research environment no single group can long afford to maintain and support a large, changing system, so the system must be so constructed as to take care of itself.

Although the scheme presented here applies to many kinds of packages, it is designed to support image processing. From a systems point of view, this means that the computations of the package are characterized by a short interchange of control information with a human user, which determines the amount of resources needed (and these vary greatly), followed by operations that are either input-output limited, or in which there is little overlap between input-output and computation. This rough characterization is used to decide the compromise between quality and transportability.

In the sections to follow, we assume that transportability is a requirement, then attempt to find a way to attain it with the smallest loss of quality. The second section considers the programming language to be used; the third section deals with the operating system and program organization is the subject of the fourth section.

PROGRAMMING LANGUAGE

Even a cursory survey of existing computers shows that FORTRAN is the only programming language with a standard, widespread implementation. FORTRAN is widely implemented partly because it is already so popular, but also because it was designed to fit the von Neumann architecture, still in widespread use. Most FORTRAN implementations "extend" the ANSI standard of 1966 in some (nonstandard) way; of course, these extensions are not transportable.

The deficiencies of FORTRAN are widely recognized, but they are largely the other side of the transportability coin—the language allows no control of computer resources; except for the ability to write arithmetic formulas, it is not very high-level and the facilities for separating, protecting and centralizing information are minimal. If a modern language designed with software engineering in mind were

---

* This work was partially supported by National Science Foundation Grant MCS 77-18719.
widely available, its programs could be moved more easily than FORTRAN programs can be. For example, a language like Alphard\(^2\) produces programs that are easy to move. The kicker is that Alphard compilers (when there are some) can be expected to be very difficult to implement on a variety of machines, since the compiler (and its run-time support) must make up the gap between the machines and the transportable programs.

The history of digital computers certainly shows that it is foolish to await the rapid spread of good ideas—somehow the bad ideas wind up cheaper—so there seems little danger in making FORTRAN work properly rather than waiting for (say) Pascal.\(^3\) Furthermore, an ambitious language may never spread to the range of machines on which FORTRAN already exists. Major vendors will have to deal with better languages; many mini- and micro-processor systems may never support them. We thus consider how FORTRAN can be tamed and transported.

RATFOR\(^4\) is the most popular version of structured FORTRAN. It has the virtues of a published definition and a partial inverse processor.\(^5\) Perhaps it should have been designed with one less iteration construct and one more conditional construct, but its wide acceptance more than compensates for such matters of taste. If we specify RATFOR as the implementation language for a transportable package, we must consider the transportability of RATFOR itself. Although the preprocessor exists for many machines, that is not sufficient. Minor variations result from conflicts between the definition\(^4\) and its presentation in a text\(^6\) and from an unfortunate choice of delimiting characters unavailable on many machines. The solution is evidently to transport RATFOR along with the package, and the techniques for doing so are well developed.\(^7\) RATFOR is written in RATFOR, and once any preprocessor exists, a pure FORTRAN version is available, for use on any machine. (It is tempting to use this mechanism to extend RATFOR, for example, to include a fancy macroprocessor;\(^8\) we judge that the departure from the published standard is not worth the power gained.)

Even with a universal RATFOR available, there may be difficulties in transporting programs and difficulties in writing them, because RATFOR is a "permissive"' translator—most of a source is never examined, but simply passed along to FORTRAN. There are three difficulties:

1. Errors in the source are first detected by the FORTRAN compiler, and difficult to relate back to RATFOR.
2. Nothing prevents the FORTRAN imbedded in the RATFOR structure from being machine-dependent, so that although it gets through compilation on a machine, it will not run properly.
3. RATFOR makes no attempt to eliminate a number of legal but error-prone constructs in FORTRAN, notably involving undeclared variable names and inconsistencies in usage. These usually result in strange run-time behavior.

Difficulty (1) is more annoying than fundamental; the others can be eliminated without compromising transportability, by a mechanism similar to preprocessing—the RATFOR source can be checked for problems before being translated, compiled and run. The PFORT verifier\(^9\) is a tool of this kind that attacks Problem 2—it checks that the FORTRAN does not go outside the 1966 ANSI subset and that certain machine-dependent tricks are not used.

There is one transportability problem of any word-oriented language that PFORT makes no attempt to solve, that of numeric precision in the presence of different word sizes and arithmetic algorithms. Techniques have been devised to attack this difficulty,\(^10\) but for the purposes of many packages, it is sufficient to trust the mathematical subroutine library of the target computer.

Problem 3 remains. For all that programmers try to keep usage consistent, mistakes are easy to make. In FORTRAN, a well-meaning programmer cannot see if his intentions were carried out. Since most of the errors that we want to detect occur across the boundaries of separate compilations, checks must operate on the complete package of subprograms as a single source. It is convenient to distribute a package as a single file on magnetic tape, so the preprocessor that goes with it should divide the routines for separate compilation and checking can then be done on the composite source. The following seems a minimal set of operations, and its implementation is no more difficult than building an identifier scanner coupled with a symbol table of usages:

a. Check for declaration of all variables and observance of conventions in variable names. (The latter is necessary to avoid conflict in libraries.)

b. Check for consistency across subprograms—argument counts and types, COMMON sizes and types, etc. More restrictive conventions can be enforced here; for example, one can forbid potential side effects.

c. Prepare a cross-reference table for all symbols of the composite program, listing usage and location in the source. (This can be helpful in decoding FORTRAN-generated error messages.)

Writing in RATFOR according to a set of conventions, then checking for those conventions before preprocessing and compilation, is almost as pleasant as programming in a modern language. On many systems it is more cumbersome, since several programs are involved in a sequence, but the payoff in debugging time saved is excellent.

**OPERATING SYSTEM INTERFACE**

FORTRAN's deficiencies as a low-level language—its inability to get at this or that machine feature—have always been supplied by a "few little assembler routines." As operating systems have grown and excluded regular programs from direct manipulation of shared resources, the most valuable machine instructions have become the system service calls. Since shared resources are scarce, a program that makes intelligent use of operating system service calls can
run more efficiently than one that does not. Two obvious examples are

1. A program that calculates an optimal memory allocation will give better response at less cost than one which runs with the largest space it might ever need. Two obvious examples are
2. A program that knows its pattern of record requests on a file can seek these records more efficiently than can any standard access method.

Proper support services are available in almost every system, but in varied form. For transportability we need a standard FORTRAN-callable interface to system functions. This interface must be kept so small that its implementation on a new machine is easy. At the same time, there is a need to make operating system services easy to use, and to tailor them to the application. These conflicting needs can be resolved by separating the interface into two parts—a "kernel" and a "surround."

In the kernel we seek the bare minimum of code, which we expect to be machine-dependent. The kernel is to be a collection of entry points that transmit essential system functions outward without regard for convenience of use. This kernel is always too large to best serve transportability, partly because there are many needed features. Part of its size results from seeking a set of features common to all systems—the smallest set may not be implementable in some cases. This factor also works against the quality of the kernel—it tends to mimic the worst system on which it must be implemented rather than the best.

Outside the kernel we disguise its awkward properties with another level of interface, the "surround." The surround contains only machine-independent FORTRAN code. It is therefore appropriate to make its routines easy to use and not worry about their extent. The surround has the special property that although its calling sequences are fixed, and it is viewed as a part of the operating system interface, its code may be juggled in package conversion. In contrast, the kernel routines require modification to implement their standard calling sequences; in the package code outside of the surround the code is movable, and the calling sequences themselves are subject to alteration. It can happen that on some particular system one of the surround routines is easy to rewrite as a direct system call, with important advantages in efficiency. So long as the entry sequence is not changed this is encouraged, but no package user needs to make the change and it has no effect should the package be retransported.

**Functions supported by the kernel**

A detailed description of the interface kernel, with FORTRAN calling sequences and implementation hints for many machines, is presented in Reference 11; here we only indicate the necessary functions.

Random-access file operations form the heart of the kernel. It must be possible to create mass storage files, manipulate their names and protections and read or write them in arbitrary-sized blocks in a true random-access fashion. The input-output operations themselves, and the file formats, should be at the lowest level the operating system provides, to minimize memory and processing overhead. Thus it is important that the operations move data directly to/from FORTRAN arrays without invisible buffering; where possible, the operations should be started and the calling FORTRAN routine permitted to continue, waiting for completion only when necessary. It is common to provide routines of this kind for FORTRAN use and the implementation is straightforward.

FORTRAN provides no memory control facilities. In image processing it is often desirable to calculate memory space required for a given picture (particularly for input-output buffers). The usual implementation of this scheme in FORTRAN uses "get" and "put" routines to move blocks of words from and into dynamically allocated space. This is unsatisfactory because the overhead is high whenever the elements are addressed in small groups. It is much better to provide a single array whose addressing is efficient and which can grow and shrink as needed; in almost every system it is possible to place such an array in memory so that it can indeed change its real size.

Process control is needed in the kernel to support the open-ended programming techniques suggested in the sixth section. The minimum facility required is the ability for an executing program to "call down" another as its replacement, without the overhead of more than an input operation. In some systems implementation is difficult, but a variety of tricks exist.

The final portion of the kernel is concerned with user communication, in the form of cosmetic features and error control. Most systems can provide information such as the date and time. Of more importance are parameters such as the best record sizes to use for disk operations and the precision/storage capacities of machine words. Run-time information should include resource usage and limitations, particularly for memory. The more a package can find out about what is really happening in the underlying system, the better it can communicate with its human users about problems encountered in execution and the more efficient it can be. Error control is also important. Most errors are unexpected in the sense that they appear as failures at a very low level, and are then communicated up to be processed by the package code. The kernel must see to it that all errors are in fact handled in this way, and in some systems that can be very difficult, since the error appears first as an asynchronous interrupt. Something similar to PL/1's ON unit can usually be arranged, leading to cumbersome but complete control.

**Functions supported by the surround**

Serial file operations can be easily built on the random-access ones of the kernel and there is seldom any reason to rewrite these outside the machine-independent FORTRAN versions, since these can be better adapted to a package's needs than the usual serial routines of either FORTRAN or
most systems. For example, it is easy to specify a buffered scan through a file in which (say) every tenth record is actually read; or, a file can be reblocked to take advantage of the availability of large buffers.

The surround can also be used to eliminate functions from the kernel that cannot be accomplished on all machines. For example, if immediate-return input-output operations are impossible, placing the entries in the surround allows implementation of the starting operation as "start and wait" and waiting as "no operation."

Most interactive communication with package users can employ the FORTRAN formatted input-output package. The memory overhead of the format-scanner routines is high, however, so formatted i-o can be eliminated by including some functions in the surround.

Experience with two systems

An operating-system interface has been implemented on two very different systems. The first is Univac 1100 Exec 8, an "old" system. The second is PDP-11 UNIX, which is about as "new" as operating systems come. The design philosophies of these two are also almost opposite—the Univac is very low-level, compensating for its deficiencies with large library packages; UNIX is designed to support high-level programs.

A technique designed to reduce the implementation effort was used for the kernel. On each machine a routine was written in assembler to provide FORTRAN access to the necessary system calls. For example, on Exec 8, one such call is for programmatic execution of a control statement; under UNIX one is for direct execution of another program. Neither system has the other's service; their different services are needed for a "change to new program" function of the kernel. Once these basic services are available, the interface routine to employ them is written in FORTRAN.

The code is peculiar to one machine, but it is often easy to adapt to another. In the example, 30 lines of FORTRAN are common to both systems. The Univac routine has 20 extra lines setting up its peculiar system call, while for UNIX this takes only one line.

In another example, the input-output part of the kernel keeps a table of open files and their characteristics. The format is different for the two systems, but the code that uses the table is exactly the same.

Code characteristics of the complete kernel are summarized in Table 1.

Because the interface routines are largely independent of each other, they can be tested individually. An interesting point is that the test driver is machine-independent, so it can be distributed with a package to help the systems programmer who must convert the kernel. Debugging is further aided by the existence of the package itself. It may contain bugs, but when one of its working features goes wrong, it tends to point to an error in the kernel.

OPEN-ENDED SOFTWARE

The package software that is to be built in FORTRAN on the operating system interface can be expected to undergo almost constant modification, to fit the changing needs of a community of research users. Under this stress it is easy to imagine a "good" package turning into many disparate "bad" packages as the code is modified by many unskilled hands. Insult is added to injury when even the most slipshod changes are hard to make, requiring extensive study of the existing code, then extensive debugging. The internal structure of the package must protect against such changes.

Careful adherence to programming and documentation standards are often offered as solutions to the problems of program modification. But it is observed that not everyone will follow standards and not everyone who tries, succeeds. The bad easily drives out the good. The only structure that will preserve itself is one that is easier to work within than to violate.

The primary technique for structuring software to encourage change is that of centralization and information hiding. Operations should be confined to a single place in the code and encapsulated with access to just the information needed to perform properly. For example, in a package program, the user interface is a candidate for this treatment. If all interactive communication is confined to a collection of routines, rigidly bounded by a well defined interface across which the information passes to other parts of the software, several advantages are gained:

1. It is easy to learn to employ the communication routines, because only the stable interface need be mastered.
2. Collecting the code in one place makes it easier to study, and if it is modified, the modification applies uniformly to all parts of the system.
3. When the centralized code is not in use, it may be possible to get rid of it, reducing the memory overhead.

There are two common methods of circumscribing operations in software. The operation may be implemented as a separate, stand-alone program, linked to other programs only by files. Or, the operation may be implemented as a collection of sub-routines and data structures within a program, communicating with the rest of the program through parameters and global data. In the first mechanism separa-

---

**TABLE 1**

<table>
<thead>
<tr>
<th>Property</th>
<th>Exec 8</th>
<th>UNIX</th>
</tr>
</thead>
<tbody>
<tr>
<td>Systems programmer experience (excluding learning assembler)</td>
<td>5 years</td>
<td>5 hours</td>
</tr>
<tr>
<td>Assembler &quot;service caller&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of system calls</td>
<td>50</td>
<td>15</td>
</tr>
<tr>
<td>Code instructions (excluding dispatch tables)</td>
<td>80</td>
<td>100</td>
</tr>
<tr>
<td>Hours to design, code</td>
<td>15</td>
<td>10</td>
</tr>
<tr>
<td>FORTRAN Routines</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of interface entries</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>Support function routines</td>
<td>20</td>
<td>14</td>
</tr>
<tr>
<td>Total RAYFOR statements</td>
<td>700</td>
<td>530</td>
</tr>
<tr>
<td>Hours to initially design, code</td>
<td>30</td>
<td>10</td>
</tr>
<tr>
<td>Hours to convert</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
tion is easy to enforce, but it is less easy to provide support functions for the independent programs; in the second mechanism support routines are readily available and the problem is to preserve the separation of the parts. Both mechanisms are valuable in an image-processing package.

**Communicating independent programs**

Of course, any two programs can "cooperate" by interchanging information in files. The user interface of a package is a good example. One program is the collection of routines that interact with the human user, and another is devoted to actually processing the information so obtained. Communication between the programs is through a file of commands/results. The separation is perfect in the sense that the processing program does not interact with the user, and the user routines do not process the information they receive. Furthermore, memory overhead is handled perfectly by this organization—the scanner, command tables, etc. are entirely gone once processing starts, leaving only a standardized command, already checked so that little error recovery code need be part of the processing. The payment for this ideal separation comes when one of the independent programs is changed—how can the changes be taken into account by the other programs without modifying them also? In the example, suppose the behavior of one processing program is changed. How can this be automatically reflected in the user dialogue? Similarly, how can the communications program really check input commands when it does not know exactly what the processing program intends to do with them?

These problems can be solved by arranging another level of communication between the independent programs. Each can notify the others of its capabilities in an initialization run, resulting in the creation of a kind of "configuration" file. This file records what programs exist, what operations they perform and describes the commands for those operations. When a change takes place it is only necessary to repeat the initialization run to have its effect felt throughout the collection of programs. This organization suggests another independent program function, that of explaining the system's capabilities and operation. A "help" program would make use of the configuration file to explain difficulties and provide on-line documentation, with this high-overhead operation entirely divorced from all "working" programs, yet necessarily up-to-date.

The two essentials for cooperation of independent programs are the ability of one to invoke another (and itself be reinvoked to inspect the results), and the definition of processing tasks in a format that can be concisely described in a configuration file. The operating system interface provides the former and the latter is a natural consequence of any command language that can be formally described.

**Linked sub-routine organization**

The primary reason for encapsulating package operations as complete, independent programs rather than as loadable overlays is that overlaying is done very differently on different machines, and is usually a high-overhead process. Nevertheless, connecting groups of sub-routines by conventional linking is often a better organization than that of separate programs. In particular, this organization is essential for cascaded processing such as neighborhood operations on an image. In this situation the separate-program organization would lead to as many passes through the image file as there are operations, while subroutines called in sequence would require only one pass. The question for linked sub-routines is how we can centralize support functions and make it easy to add sub-routines and integrate them with existing ones.

There is no difficulty in passing information about subroutine capabilities across program boundaries—the description in the configuration file can be broken down by routine within program. Rather, the problem is that existing routines interact in an intricate pattern, and a new or altered routine must be allowed to participate without its author mastering very much of the complex code environment. We illustrate how this can be done for the composition of neighborhood operations.

Neighborhood operations in cascade can be performed on a very restricted portion of an image. There is always a "window" that, moving serially through a file, contains all the pixels needed to perform one step of the composite operation. Some operations in the cascade make use of the results of others as well as data from this window, but it is straightforward to arrange row buffers so that all of the necessary data is present at once. Control flow is more complex, since one operation (or sequence of operations) may have to be repeated before the next can proceed. An open-ended package requires that the routines which participate in the sequence be written as unit operations, transforming a fixed number of rows of input into a similar output.

To add a routine requires no understanding of the complex driver mechanism that adjusts buffer pointers and sequences the operations, but only the understanding of the parameter conventions for receiving unit input and delivering unit output. The existing routines are selected and called through tables, which must be updated to reflect changes. This organization is as flexible and easy to extend as might be imagined. The routine-name table is the basis for building the configuration file, so an independent program interacting with users knows which neighborhood operations are available, and what parameters each requires. Any sequence of these operations can be passed to the program in which the unit-operation routines are driven by the same table. The user-communication program can even determine memory requirements, and break the sequence up with intermediate files if not enough core is available for the necessary window.

**PRESENT AND FUTURE PLANS**

Our immediate goal is the production of an image-processing package design that is both transportable in the wide sense described in the first section, and of high quality. As
a practical demonstration of this design we imagine the following pieces of software:

1. Operating system interface kernels for several machines, along with documentation describing the implementation on a new machine. Operating system interface surround.

2. Preprocessors for RATFOR and for extended syntax checking of transportable code, themselves transportable.

3. An independent user-interface module capable of checking commands for and invoking any number of processing routines, their operations described and driven by tables. This module will include “help” information driven by the same tables.

4. A sample processing module implementing neighborhood operations with almost arbitrary cascade and parallel capabilities.

Once this design has proved itself, it will be appropriate to extend it to a full-blown image processing package by the addition of other processing modules. This task will be less difficult than the complete creation of a package, but it is not easy. The major advantage is that once the effort has been expended, nothing like it should ever be needed again—the package should be adaptable to new situations at very low cost.

REFERENCES