SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 10, 1009–1027 (1980)

Transportable Package Software

R. G. HAMLET

Department of Computer Science, University of Maryland, College Park, Md. 20742, U.S.A.

AND

R. M. HARALICK

Virginia Polytechnic Institute and State University, Department of Electrical Engineering, Blacksburg, Virginia, U.S.A.

SUMMARY

Package programs allow people who are not computer experts to use the power of machine computation for specialized purposes. Because the designers of scientific packages are more often experts in their scientific field than in computing, they may ignore issues of transportability and ease-of-use until too late, and produce a package that is difficult to use, and difficult to move to a different computer. In this paper we suggest some techniques to aid the scientific package designer.

We suggest a kernel of routines that interface to the peculiar operating system of each machine, providing sophisticated but standard operating system services. This kernel makes the operating system of each computer appear identical and does not pose a difficult implementation problem. Above this interface all code can be machine-independent, without sacrificing power and ease of use on any machine.

We also suggest some novel organizations of processing routines designed to make the system easy to alter and extend. Complete independence of modules encourages centralization of tasks, which is both efficient and essential for easy extension. With this organization, the vast preponderance of package code can be written in machine-independent ANSI FORTRAN. We suggest the use of a preprocessor like RATFOR to make this more pleasant.

The paper closes with an application to an image-processing package, in which a major problem is the flexible sequencing of processing routines.

KEY WORDS Portable software FORTRAN Applications package Image processing

INTRODUCTION

'Applications packages' are an important part of the computing workload on most general-purpose computer systems. Although there are important differences among these package programs, they share the following:

1. A package is large—tens of thousands of lines of code are usual.
2. A package makes significant use of the service facilities of the operating system underlying it, particularly services like files, memory control, and input–output streams.
3. A package contains a great deal of special expertise from its field of application—its algorithms are peculiar to that field, and their programming is less difficult than the ideas behind them.

* This work was partially supported by the National Science Foundation under Grant MCS–77–18719.

0038–0644/80/1210–1009$01.00 *Received 12 January 1979*

It is the interaction among these features that makes package programs so common and so useful: the package captures what practitioners of a field need, in a form that requires far less knowledge of computers, and of the field itself, than that required to program a problem directly. (Of course, a novice may misuse the package to obtain misleading or plain wrong results. We will not explore this problem here.) Once the effort has gone into creating a package, there is considerable pressure to use it with many machines, operating systems and installations. Because so much of the content does not really depend on the computer implementation, but rather on the special algorithms involved, the 'conversion' of packages is feasible, and much attempted. It is unfortunate that the task is usually harder than it first appears, and weeks stretch into months with the package still not 'up' on the new system.

The designers of typical package programs are experts in the applications field, and often computer novices. Although they may be computer experts by the time a package is completed, they seldom then begin again. Furthermore, they do not often design a second or third package, so what they learn is lost. It is therefore worth setting down a collection of ideas that are well known to experts, but that might not occur to the novice designer. Two issues are central: (1) the package should be transportable, and (2) the package should employ the power of the operating system beneath it, so that it is easy to use. These items are often the last to occur to a beginner.

The technique we suggest for package implementation is to encapsulate and standardize each important part—the calls for monitor services, the invocation of specific routines to solve a problem, and the description of the fundamental objects to be processed. Then we substitute the standardized objects for actual, arcane objects. In this way we achieve communication without regard for details that change from system to system, task to task and problem to problem. The payment is that we must then do something to actual systems, tasks and problems so that they appear in the standardized form. 'Transportability' is achieved when the process of imposing the standardized form is fast and reliable. At the same time, by using powerful techniques, we ensure that the transported code is easy to use, and need not sacrifice good human engineering to its ability to run on any machine.

SUMMARY OF IMPLEMENTATION TECHNIQUE

The solution to any large problem may lie in dividing it into smaller ones. Within an applications package we propose a number of 'interfaces'—rigid boundaries with well-defined communications conventions. Establishing these boundaries divides the package into more manageable parts, but further enforces transportability and extensibility properties. On one side of the boundary is machine-dependent code which must be confined to a single section of the system and whose surrounding interface is designed to minimize what is inside. On the other side of the boundary is code that implements visible package features. This code must be subdivided into its interface and supporting routines. To add new features then requires little knowledge of the support.

Many of the techniques we propose apply to almost any kind of package. However, since most of a package's code is *not* application-independent, we include some details drawn from a particular application: image processing. This application may be characterized by its heavy use of files, large demands on fast memory and the need to efficiently compose user-specified operations.

A note on language

We avoid all problems of differing machine architectures by assuming them solved by languages and libraries. For example, the same FORTRAN program for a numerical algorithm may produce different results when executed on machines with different word sizes or floating-point algorithms. But we assume the results are good enough. (Techniques have been devised for handling such numerical problems,[2] and these could be employed along with ours.)

Over the range of machines and systems we expect, there is no alternative to a subset of FORTRAN as the primary programming language for scientific applications. A number of software systems exist for transporting code not written in a standard language;[3] they work by bootstrapping up from FORTRAN through macro- or language-processors, or by bootstrapping from another machine.[4] But the better languages available in this way are either not well-adapted to the target machine (if they are easily moved), or require far too much work to transport. (Interpretive BASIC systems are an example of the former case; self-compiling PASCAL compilers of the latter.) On the other hand, preprocessors exist to 'clean up' FORTRAN, notably RATFOR,[5] which are transportable through bootstrapping. RATFOR is particularly attractive because it has a partial inverse processor,[6] and the package writer or extender must often deal with hunks of code that were written outside the package design. Because RATFOR copies the FORTRAN that lies within its structured control figures, it is necessary to stick to a subset that avoids incompatabilities for the range of machines imagined. In brief, this means keeping away from most input/output and formatting; storing characters so that justifying and word-size conventions don't matter; and avoiding all but the simplest subroutine and storage allocation forms. The 'machine-independent' FORTRAN program is a well-worn idea.[7] However, choice of language subset does not necessarily bring portability because of the necessity of operating system services.

Operating system interface

The operating systems we are considering all support the needs of packages for file operations, and memory/process control. Machine independence is achieved by obtaining these services only through predefined FORTRAN subroutine calls. This collection of entry points is a kernel that makes all operating systems look the same to the package.

Three forces shape the definition of the operating system interface:

1. The services should be powerful and easy to use.
2. It should be possible to build the kernel around almost any existing operating system.
3. The implementation of the kernel for a new system should be easy for a local systems expert.

The first two forces 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 is only a calling-sequence converter, transforming the FORTRAN subroutine calls into monitor calls. When the system provides very different services, however, it may take a lot of code to build them out to the proper form. The section that gives details of the operating system interface does so by describing the FORTRAN calls the kernel must accept, and indicating implementation techniques.

Command and process interface

Transportability is achieved once we have written only restricted FORTRAN code calling only on the standardized operating system interface. Moving to a new machine requires only the implementation of the kernel supporting the standard calls. However, if an applications package is to be understood or modified, it requires far more internal structure.

A package must communicate with its users, then carry out the tasks they specified. The details of scanning commands and interacting with a user are unlike the processing that takes place after the command is decoded. This provides an opportunity to break a package into two modules communicating through a well-defined interface. By making these modules each an independent program, we can further avoid the memory overhead of code that is not in use.

In the command-processing module an 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 can be 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 straightened out with the user before the information is stored. The processing module that then deals with the standardized request can be conventionally linked to the command routines, or overlaid. But in many cases this organization is inadequate, either because complex overlays would be required to parcel out memory, or because processing requires a complete program, not a subroutine. Overlays are unsatisfactory because they are different on different systems, a transportability issue, and because they are slow, an efficiency issue. We therefore arrange for one of the standardized operating-system services to be 'program exchange', in which the executing code calls for itself to be replaced by another ready-to-execute program module.

The 'Process and Command Interface' section examines the problems and advantages of this organization, particularly with regard to the open-ended nature of the processing modules that may be included.

Structure of processing routines

Even with command processing removed, and a mechanism for separating tasks as absolute modules, a package is complex enough that further structure is required. For any particular applications discipline, a collection of useful routines can ease tasks that commonly occur within the package. A common example is the set of figure-drawing routines for a plotter: there is a routine to draw a line, a circle, etc., and by mixing calls to these routines with other code, a complex figure can be drawn.

If there is a class of objects manipulated by many routines, it is worth defining a standard representation for the objects. In currently popular terminology, this amounts to converting a set of library routines into an 'abstract data type'[8] in which the routines perform the operations on the standard objects. Information-hiding[9] is the principle that dictates the grouping of procedures and the scope of data structures. In many packages there are book-keeping operations associated with most processing, which if centralized and encapsulated properly, improve both the understandability and efficiency of much of the package code.

For the example image-processing application, some of the appropriate abstractions and library groupings are discussed below.

OPERATING SYSTEM INTERFACE

In this section we describe the kernel of the operating system interface and its implementation. The programs that rest on this interface will see its service facilities as the only ones available.

Services of actual operating systems are delivered in response to requests that appear like subroutine calls. (The actual hardware mechanism may be distinct from a subroutine jump.) In an operating system that happened to have just the services specified for our interfaces, implementing the interface would amount to converting a FORTRAN calling sequence into one for an executive service call. With a system requiring only a trivial interface, a local expert should be able to complete the work in a few hours.

Even if the operating system services that exist are very different from those needed for the interface, the first step in implementation is to write a low-level routine that makes the existing services available in FORTRAN programs. Then the rest of the programming can be done in FORTRAN, and a good deal of it will be similar to that done for other machines, so that the implementation of the kernel is one of adaptation rather than of writing code from scratch.

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, and simple situations should not require the caller to make use of the full-blown error mechanism. For software development there is another important factor: it must be easy to make sure that *every* error is detected, even those that cannot occur once the software is working properly. The alternative-return mechanism of FORTRAN is adequate for most of these requirements; unfortunately it is not included in the ANSI standard language.

To take care of error processing we suggest that each system interface routine be defined as an INTEGER FUNCTION, and each have a final INTEGER parameter ERR for error processing. If there are no errors, the routine must return a non-negative number (which may have significance as a part of normal processing); in that case ERR is ignored. However, if there are errors to report, the routine delivers a negative value indicating the type of error. The calling routine may perform whatever processing it wishes using this code. If the value of ERR is positive, there will be no internal error diagnostics—the caller takes full responsibility. If ERR is zero, any error will result in a printed message, explaining the rudiments of what happened. If ERR is negative, there is a compromise: no message will be printed for the error whose code is the same as ERR; all other errors will be handled as if ERR had been zero.

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

Process control

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

```
INTEGER FUNCTION CHANGE(PROGRAM, PARAMETERS, COUNT,
ERR)
```

where PROGRAM is a file identification of the new program (see below), PARAMETERS is an array whose positions 1-COUNT contain data to be passed to the new program, and ERR is the error parameter discussed above. The only potential failures for CHANGE involve the non-existence of PROGRAM. 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.) A program that is sure of finding the successor whose name is in PRG, and wishes to pass ARGS(1)–ARGS(5), can make the call

```
E = CHANGE(PRG, ARGS, 5, 0)
```

and know that if anything goes wrong, the message will come from CHANGE. On the other hand, the caller may anticipate that the new program may not exist, so that

```
E = CHANGE(PRG, ARGS, 5, -3)
IF (E = = -3)
  {#DOESN'T EXIST
  WRITE(6, 1); 1FORMAT('#NO SUCH PROGRAM')
  }
```

is a call on which the anticipated error is explicitly processed, but others are left to CHANGE. Finally, if the last parameter has a positive value, the caller must test for all alternatives and perform all error processing.

All routines in the system interface treat errors in this way, but the paraphernalia will be suppressed in the discussion to follow. The 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.

The new program that is the target of a CHANGE picks up the passed information with a call to the routine

```
PASSED(PARAMETERS, COUNT)
```

Whose arguments have the same meaning, but are set rather than received. PASSED interprets the incoming value of COUNT as the space available in the PARAMETERS array, and returns an error if that is inadequate for the data available. A returned COUNT of -1 indicates that the program was not started by CHANGE, and hence there is no data to pass.

If the CHANGE service succeeds, the calling program never again receives control. In most cases the PARAMETERS will be implemented by a temporary mass-storage file; however, some systems (e.g. DEC–10) provide a special communications mechanism through fast registers or monitor core-buffer areas.

Where CHANGE is not supported by a similar real operating system service, two tricks are useful. It may be possible to programmatically execute job-control text, and this may include the ability to start a program. Or a program cannot be started, but commands can be queued to take place at the end of the present execution. Then CHANGE amounts to checking that all will go well, then executing the appropriate commands to start the new program just before (after) termination (depending on the restrictions). A different trick is to place the code for CHANGE at the beginning of the address space, and obtain the new program by reading its absolute code directly into memory over the old image. (This is simple only if the format of the images on mass

storage is simple. It was first used in 1968 by Bill Weiher to improve the command language for the DEC PDP–10, in a way very similar to that proposed for the control program described below.)

When the programs involved in a CHANGE must communicate at length, they can do so by passing the name of a file in the restricted parameter list. To anticipate the ideas of process control presented below, it is possible to produce a kind of subroutine behaviour in the new program. Before calling for the CHANGE, the original program writes out any data it will need into mass-storage files, including information about its stage of execution. Then it invokes the new program with CHANGE. The new program, when it is done, reinvokes the original, passing it an argument that indicates 'special starting case'. When the original program begins, it then knows to replicate the situation where it left off, and continue from there.

File operations

Standardization of file formats is important only when interchange of information between two different systems is required. In most packages, routines will exist to move files (in whatever the internal format used) to/from raw data. An interchange can then be accomplished by trading magnetic tapes as raw data using the simplest format, and without concern that the mass-storage formats on the systems may be entirely different. The standard interface is therefore concerned with file operations, but not with file formats, which ordinarily should be the simplest the local operating system provides.

Because files differ greatly from system to system, it will be necessary to duplicate most of the operations performed by the monitor (and the input–output control system, if there is one) in the interface. Fortunately, most of the coding can be done in FORTRAN, and much of it carries over from machine to machine.

Files can be random-access files or sequential files. We discuss here only random-access files. The operations which programs need to perform on such files are open, read/write, and close. The first and last could be implied by the transfer operations, but in some cases they are needed for 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.

As an object, the primary aspect of a file is its name. Operating systems impose many peculiar conventions on file names, but each must be allowed to use its own. Otherwise users of a package must learn two conventions, and should they need a regular system program like an editor, must in addition know the mapping from an internal name to the 'real' one.

The complete file name is stored in an array (one character per word, of course), and passed to each system interface routine. Space is provided in this array for the routines to assign and maintain some kind of internal description invisible to the user; if this is done then it will be more efficient if the user identifies a file in use with the array in which its name is stored. (In the case of 'new' and 'old' files being processed simultaneously, this identification is essential, as described below.) The array parameter containing the name will be designated FILE for each of the routines to be described.

The first usage of each file will be a call on:

```
OPEN(FILE, TYPE, BLOCK, SPACE)
```

The subsequent usage depends on the value of TYPE, as follows:

TYPE	*Usage*
'old'	The named file must exist prior to the open call. BLOCK and SPACE are ignored, but OPEN returns the block size and space size of the existing file.
'new'	An independent version of the named file is to be created (whether or not a version exists). BLOCK species the units in which reading and writing will be performed, and in which the record numbers of the file will be counted. SPACE gives the maximum size of the new file, if the underlying system requires this.
'sys'	The named file is to be used in the simplest, most general format provided by the underlying operating system (no extra information is to be recorded or assumed, for example). If the file exists, the old one is to be used; if not, a new one is to be created.

In some systems, 'sys' files will not differ from others, but often it will be necessary to maintain special informantion in the file that will distort its form, hence the special type.

No two calls to OPEN (without an intervening CLOSE) 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. Which file wins out in the end is determined by the pattern of CLOSE operations, described below.

OPEN files have their names recorded in an internal table. Because the table is of limited size, CLOSE is used to clear it, as well as to ensure orderly use of files:

CLOSE(FILE,)

reclaims the table space, further action depending on the circumstances of the OPEN. If the name is only open once, there is no explicit system action except for 'new' files—these are added to the system directory, and should an existing version be there, it is replaced. Under the peculiar circumstances that the same name is being used twice, once 'old' for an existing file and once 'new' for a replacement, these two usages have different arrays containing the same name (so marked by the OPEN to keep them straight). Passing one such array to CLOSE selects which of the two versions triumphs in the system directory; a CLOSE on the other is implied.

Some separation of names has been applied to the routines that perform file operations between OPEN and CLOSE, but the description of one displays the features of all:

READER(FILE, RECORD, BUFFER, COUNT, WAIT)

specifies a random-access operation on the file. (If it was OPENed 'new' the operation takes place on the independent version being created, otherwise on the existing version.) RECORD counts in units of the BLOCK size of the OPEN. COUNT words of data are read from the file into the array BUFFER. There is to be no internal buffering or blocking. If the value of WAIT is nonzero, return does not occur until the operation is complete. (If zero, return is immediate, and the caller must use TESTIO or WAITIO—see below.) Records may not be read past the last record written (hence an initial READER call for a 'new' file will necessarily fail); however, no attempt is made to keep track of more than the 'outlier' record, so it is possible to read what has never been written, with unpredictable results.

If WAIT is zero, READER returns a code value, which may be passed to the routines

WAITIO(CODE)

and

TESTIO(CODE)

to determine if the operation started for this CODE is yet complete. The first does not return until it is complete; the second merely checks the current status.

The routine WRITER(...) is analogous to READER, but may encounter the additional difficulty of passing beyond the maximum file size.

In closing this section we note that the file operations specified here constitute a sufficient set of primitives to permit the file access protocol required by image processing applications.

Interrupts, system parameters and memory allocation

FORTRAN makes no attempt to provide communication with an operating system for 'cosmetic' parameters like date and time (or more essential ones like memory usage). Neither does it have any standard way to deal with unexpected error conditions such as division by zero (or software problems such as subscript bounds violation). Finally, FORTRAN static memory allocation does not allow programs to change size to meet varying user demands. All of these problems must be solved to write a package program in FORTRAN that is efficient and easy to use. Too often, portability through FORTRAN is achieved at the cost of awkward programs that are far too difficult and expensive to use.

There is no difficulty about supplying system parameters to a running program. Most systems have service calls to provide the information, which can be delivered in array, subscript positions corresponding to different items. When particular data is not available, the array position can contain an error code.

Memory allocation and interrupt processing are more difficult, because they involve manipulation of addresses. There are systems in which the necessary facilities are simply not available, but when the hooks do exist, they can be attached to FORTRAN by means of a 'main program that is one of the system interface routines. Execution of any package program then must begin with this routine, and it passes control outward into the package FORTRAN code. It does so by assuming the existence of a subprogram:

SUBROUTINE MAIN(DYNARRAY)

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

ALLOCATE (SIZE)

it will be as if it were dynamically altered to

DIMENSION DYNARRAY(SIZE).

(ALLOCATE will likely be an entry point within the interface main program.)

The same initial main program can perform interrupt setup operations. Where the system permits, it can record an internal entry point for transfer when an unexpected error occurs. Then, should that point be reached, it can assume the existence of a package subprogram:

INTERRUPT (TYPE)

to call. 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 INTERRUPT, 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 MAIN, and thus cuts short the interrupted code forever. If MAIN is written to test a global flag, it can know whether it is starting or restarting.

As an example of the use of an interupt service, imagine the problem of terminating unwanted output. Most systems have some means of alerting a running program to a user 'attention' typein, implemented as an interrupt. If the user invokes this during output, the interface main program will take control, then call INTERRUPT. INTERRUPT can set a flag which the print routine tests before each line, and return in the 'dismiss' mode. The result will be that printing will instantly terminate, but the package program will proceed normally.

Surrounding the kernel

The routines described in the preceding sections constitute a minimum kernel that permits a package to function on any system at no disadvantage compared with other programs on that system. Their implementation is not always trivial, but amounts to only hundreds of lines of RATFOR and assembler, even on deficient systems. (The Univac 1100 implementation is about 500 lines of RATFOR and 100 lines of assembler, stripped of comments.) The services included in the kernel are not the only ones an operating system might need to supply to a package. For example, interactive terminal input–output and serial file input–output have been omitted. We propose to handle these in a somewhat different way.

Although there may be better ways to do it on some systems, the services omitted from the kernel can be obtained by combinations of kernel interface calls, or use of features built into FORTRAN itself. We therefore propose to 'surround' the kernel with a further set of routines whose calling sequences are fixed as part of the package definition, but which are implemented in machine–independent FORTRAN. Should an implementer have the time and know of a way to vastly improve the behaviour of these 'surround' routines for a particular operating system, they may be treated as part of the kernel, and implemented using special system features. But transporting the package to a new machine does not require any work on the 'surround' routines, since their standard FORTRAN implementations will do. Just as it is advantageous to shrink the true kernel of the system interface so that it is easy to convert, so it is advantageous to allow the 'surround' routines to expand as much as possible, extending standardized calling sequences far into the package at no cost in transportability.

PROCESS AND COMMAND INTERFACE

The mechanism of process control in an applications package can solve three problems at the same time: it can make the package easy to write and open-ended so that

capabilities are easy to add; it can make the package easy to use, with powerful, flexible commands; it can improve the performance of the package by properly utilizing scarce resources such as memory. The essential idea behind these advantages is to make each part of the package a separate executable program, under the control of a 'package controller' that sequences the parts and enables communication between them.

The package controller itself is a complete, executable program. Its task is to handle communication with the package user, then to pass the user's (correct) commands to the routines that will process them in an appropriate sequence. The idea has been used in application programs such as KANDIDATS,[10] and comes from a mechanism for operating-system commands. Common operations (executing a program starting with source files, for example) require a sequence of communicating processors (compiler, linkage editor, loader) to satisfy them. Rather than ask the system user to specify this sequence with indivdual commands (and keep track of various files involved), some systems provide a utility processor that functions just as the package controller described here does. This controller takes the user command, converts it into the form needed for subsequent processing, then in turn relinquishes control to the necessary programs, assuming control in between their executions to direct the next. The co-operation of all programs is required, since the controller must be restarted as each terminates. The CHANGE and PASSED service calls to the standard operating system are the mechanism by which this transfer of control and information is effected.

It is advantageous for the package controller to perform all command processing. Centralizing this function eliminates its duplication in each program, with the dual advantage of similar form for all commands and saving the space of command-processing code. When this centralization is possible, the command finds its way to each program in the form of checked parameters requiring no further processing to direct the action.

Let us imagine a typical usage of a package under a package controller. We suppose that the operations required involve several programs, and a sequence whose progress depends on the results so far. The user begins by initiating the package controller program. This is the only step in which the actual commands of the real operating system are used. (That system may have provision for automatically starting a program as part of the login or deck–identification process, in which case no commands are needed.) The package controller obtains a consistent, correct request from the user. The checking that can go on at this level involves the existence of files (and their proper formatting), setting of parameters, specification of output format and device, etc. In batch operation there is nothing to do when an inconsistency or impossible specification is detected (except to abort the batch run); in an interactive system, however, a considerable dialogue can ensue. Once the package controller has all the data it needs, it writes a small file containing a complete coded description of what is to be done. Then it starts the first program in the processing sequence with CHANGE, passing it the parameters necessary for its operation. (One parameter may be the name of the summary file, but this is uncommon.) When this first program is finished, it reinvokes the package controller with CHANGE, passing as parameters a description of what has been done (or, more important, what has failed). Using this information and its original summary file, the package controller starts the next program, or perhaps seeks new input from the user, and so on. Upon completion of the entire sequence the package controller is again in execution, ready to receive another user command. The final user command terminates the computer run.

Figure 1 summarizes the operation of a processing module under the package controller. In the figure, a typical user request is formulated by the user interface of the controller, and written to the summary file. Then the processing routine is started. It places output on data files, and restarts the controller. The controller relays the data file to the user.

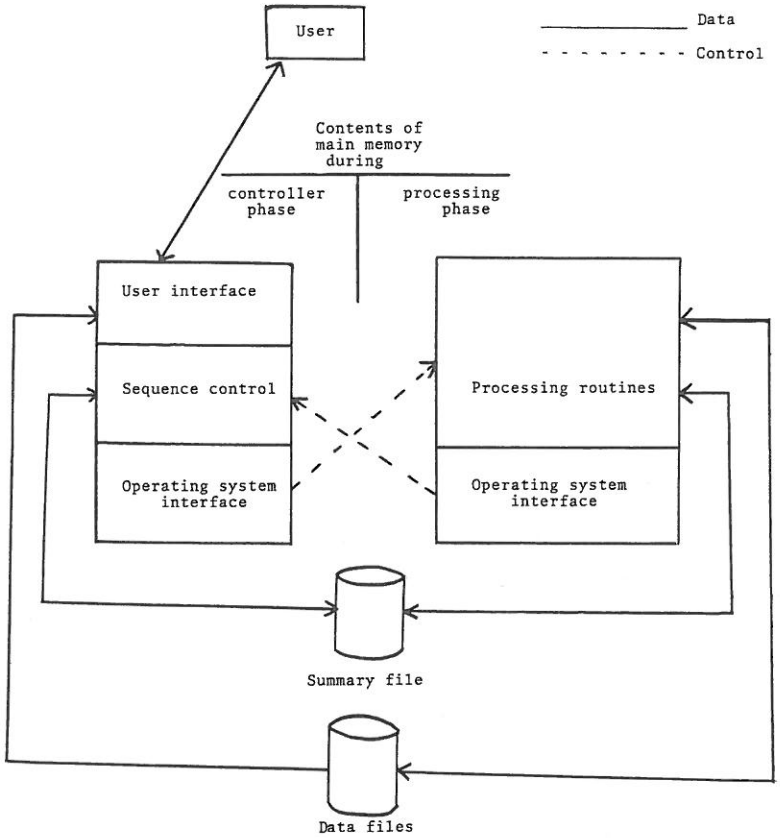

Figure 1. Illustrates the data and control processing communication links for different phases of execution

Although the mechanism of interchanging absolute modules is satisfactorily open-ended, it can be very inefficient. If the processing steps involve files (as they do in the image-processing application described later), each time a new module is cascaded a new temporary file is required for its data. The conventional processor organization in which subroutines are successively called from a driver program, eliminating the need for the intermediate temporary file and turning two passes over the data into one, is far better. But there is a conflict between this mode of operation and the goals of isolating the user communication in the package controller and keeping the potential processing routines open-ended. There are two difficulties in writing the processor as a set of linked routines:

1. How can the controller know the detailed capabilities of the modules it might call, yet keep those capabilities open-ended? Imbedding a list of routines in the

controller makes it difficult to add new ones; worse, the properties of routines (memory requirements, for example) are not known when the controller is compiled.

2. How can a processing module, presented only with a table specifying the sequence of operations to perform, call its internal subroutines in the table-driven order, yet retain the property that it is easy to add or modify subroutines?

The list of processing subroutines and their characteristics may be placed in a permanent file. This file can be dynamically created as a part of an initialization run of the package. On this run the controller must have a list of the module CHANGE names of all parts of the package. It starts each in turn, passing a parameter to identify the initialization call. Each module in turn contains a table of subroutine names and parameter descriptions, and on the initialization call it transmits the contents of this table back to the controller, along with run-time information about those routines. (Since the processing module is in execution, it has information from the operating system about its own requirements, as described above.) Thus the controller can construct the necessary table.

The table of internal routine names within modules also solves the problem of a table-driven sequence of routine calls. The main routine of each module can be only an interpreter of processing sequences as supplied by the package controller, in which its subroutines are the 'operations'. Upon receipt of the processing command sequence, the main routine establishes a correspondence between each routine name and its position in the internal table. The latter number can be the target of an assigned or computed GOTO leading to an actual CALL of the named routine. Thus to add a new routine to a module requires only the addition of the code as a subroutine, an additional CALL statement, and a new name in the internal table. (To make the controller recognize the existence of the routine requires a new initialization run.)

The following display illustrates a simple case of control flow in a processing program driven by a command sequence from the package controller:

```
      ...
#     Routine-name table:
      DATA NAMES (1), NAMES (2), NAMES (3)/'A', 'P', 'X' /
      ...
#     Set variable LAST to length of command sequence.
      DO COM = 1, LAST
#     {Get command name at position COM in command sequence
#       table, locate it in NAMES table (setting variable
#       INDEX to the position).
      GO TO (100, 200, 300), INDEX
  100 CALL A
      NEXT
  200 CALL P
      NEXT
```

```
300  CALL X
     }
     ...
          SUBROUTINE A
     ...
          SUBROUTINE P
     ...
          SUBROUTINE X
     ...
```

More can be centralized in the package controller than just command processing. For example, an important part of an interactive system is the use of devices like menu commands, and 'help' tutorials for the services the package can perform. Putting all of the code for this in the package controller, driven by files associated with each program, the facility can be at once sophisticated, open-ended and sparing of resources.

The controller program can also serve as a test driver for the implementation of the standard operating system interface, by including a section in which all interface routines are used only to see if they work properly. This section can be invoked as a part of the initialization sequence. For example, to test the CHANGE and PASSED interface services, the controller can write a file, then CHANGE to itself passing the file contents as parameters. The new copy of the controller can then compare the passed data with that in the file.

AN IMAGE PROCESSING MODULE FOR NEIGHBOURHOOD OPERATIONS

Operational flexibility is a major problem of image processing software systems. In 'neighbourhood operations', those performed on a local region of a picture to yield one point of the processed image, even a few primitive operations can combine to make a large total available. For example, ten primitives allowed to cascade to depth four gives rise to $10+100+1000+10{,}000$ potential operations. From the software point of view these 11,110 operations can be implemented as ten subroutines, plus a driver program clever enough to do the necessary book-keeping.

The conclusion of this simple analysis is that to make maximum use of small building blocks required by a modular approach to the software structure, a complex neighbourhood operation should be performed, if possible, as the composition of simpler neighbourhood operations. In this manner, the composition capability of the package driver can be utilized to its fullest and the writing of new code to add a new neighbourhood operation can be minimized, since fullest use of old code can be made. In the remainder of this section, we describe a technique for composing operations.

We define a neighbourhood operation to be a process which accepts for its input a fixed number of logical records and produces for its ouput one logical record. The result of processing a file of logical records with the successive composition of a number of different neighbourhood operations is one output file of logical records.

In order to generate such an output file, each process must have its input buffer filled with usable rows and upon execution generate rows for the next process's input buffer.

In the simplest execution cycle, after the last process has generated a row, and it has been written out, the next row of the input file is placed in the input buffer to the first process and the cycle begins again. In more complex execution cycles, after the last process has generated a row and it has been written out, previous processes must be sequentially checked to see if any of them are able to generate new rows. If so, the process nearest to the last that can do so is allowed to run. Processing then cycles forward from this until the last process completes.

Thus we see that each process has associated with it a buffer of data which constitutes the input to the process. Each process produces output which must be placed in the input buffer of the next process. After the process executes, some of the rows in the input buffer may no longer be needed; other rows in the input buffer may no longer be usable the next time the process is executed.

In order to keep track of the condition of each process's input buffer, the composition controller must have a buffer state vector. The ith component of the buffer state vector indicates the number of rows remaining in the buffer for the ith process which are usable by the ith process during its next execution.

Suppose the last process which was executed has index p. Let $p \leftarrow p+1$. Test the input buffer state of process p. It is either full or not full. If the input buffer is full, then run process p. If not, set $p \leftarrow p-1$, and cycle back to testing the input buffer state of process p.

The state of the input buffer of the process just executed and the state of the input buffer of the next process to be executed must now be updated. The updating amounts to decrementing the former by the number of rows no longer usable during the next execution of the process and then incrementing the latter by the number of rows just generated and placed there.

With all the book-keeping operations centralized in a driver, each neighbourhood subroutine is responsible for only a single operation which inputs the number of needed lines, and outputs those generated. For example, the following is an expand/compress subroutine that alters the resolution of a picture:

```
SUBROUTINE EXPAND(LINEIN, SIZEIN, LINEOUT, SIZEOUT,
  GENERATED)
INTEGER LINEIN(SIZEIN), LINEOUT(SIZEOUT)
#The input and output lines and their lengths
REAL RATIO, STEP
INTEGER FIXSTEP
RATIO = FLOAT(SIZEIN)/FLOAT(SIZEOUT)
STEP = AMIN1(0.5, RATIO*0.5)+1.0
DO N = 1, SIZEIN
  {#Create the new line from the old
  FIXSTEP = STEP #Convert to subscript
  LINEOUT(FIXSTEP) = LINEIN(N)
  STEP = STEP+RATIO
  }
GENERATED = 1 #Return the number created.
RETURN
END
```

A driver to cascade such routines in a processing sequence must read an input image file, supply its records as needed to the first processing subroutine, call the necessary subroutines in sequence and as the synchronized data allows and as records are produced by the final processing routine, write them to an output image. We now sketch such a driver, using the interface routines described earlier. To avoid a dicussion of serial input–output we assume that the row size of the input file is the same as the random-access block size, and that this same row size is appropriate for the output image.

Before the program of which the driver is a part is executed, the package controller must obtain from a user the sequence of operations to be performed. The processing is then controlled by this information, passed through the interface routines CHANGE and PASSED. We suppose that the array TODO contains a sequence of codes representing the subroutines to be called. The first one is reading and the last is writing, whose code values we suppose to be 1 and 10: the sample routine presented above has (say) code 5. Each of these routines must have a certain number of image rows available before it can perform its operation; these we suppose prestored as an array in NEEDROWS. Thus in particular NEEDROWS(1) = 0, NEEDROWS(5) = NEEDROWS (10) = 1. An array READY is used to keep track of the input buffer state for each process (with indices corresponding to those in TODO). The major task omitted from the driver code to be displayed is the shuffling of buffer pointers. A pool is kept in a dynamic array, but the code to control the offsets is not shown. Comments supply all these lacks in the display. Where a variable or parameter is not relevant to the ideas being presented, it is given as 'X'. Execution of the processing module begins in the main program of the system interface, which passes out a dynamic array to the driver:

```
SUBROUTINE MAIN(BUFS)
INTEGER BUFS(1)   #The dynamic array, to be used as a buffer pool
INTEGER INF(15). OUTF(15), CONTROLLER (15)   #Arrays containing
                                             #the names of files
                                             #presumed to
                                             #already
                                             #have proper values
DEFINE EOF  -10   #Error code for file reading
INTEGER ROW       #The number of rows in the input picture, to carry
                  #over to the output picture

INTEGER POOLSIZE   #Total number of buffers needed in the pool
INTEGER NEEDROWS(100)   #Input rows required by each process
INTEGER TODO(100)   #Numbers of the subroutines in the order needed
                    #(thus TODO(1) = 1, and the last-used
                    #element of TODO contains 10
INTEGER READY (100)   #To keep track of current input buffer status
INTEGER INDEX   #Subscript of TODO currently being processed
INTEGER INREC, OUTREC   #Positions in the files
#Variables used for temporary storage are omitted
ROW = OPEN(INF, 'OLD', X, X, X)      #Blocking on input file;
                                     #(assume no errors)
```

```
X = OPEN(OUTF, 'NEW', X, X, X)        #Assume that input and output files
                                      #differ
X = ALLOCATE(ROW*POOLSIZE, X)         #Expand the dynamic array for the
                                      #buffer pool
#Set up pointers into the buffer pool for each process
#Initialize READY to all zeroes
INREC = 0; OUTREC = 0    #Start off the record counters
FOR (INDEX = 1;; INDEX = INDEX+BUMP)
  {#Perform the next unit operation that is ready, moving down
    #the list and backing up as little as possible
  PROC = TODO(INDEX)
  IF (READY(INDEX)<NEEDROWS(PROC))
    {#This one cannot proceed
    BUMP = -1
    NEXT
    }
#Select the proper subroutine to run by its code
  DO N = 1, 1
    {#Really a case statement
    GO TO (1, ..., 5, ..., 10), PROC
    1 #Read process
      #Compute an offset OB appropriate to the input buffer
      #for the process in TODO(INDEX+1) which is to follow this one
      E = READER(INF, INREC, BUFS(OB), ROW, EOF) #Read next
      #record
      IF (E == EOF)
        {#End file
        X = CLOSE(INF)
        X = CLOSE(OUTF)
        #Ignore side effects and return to the package
        #contoller program
        X = CHANGE(CONTROLLER, X, X)
        #Control cannot reach this point if the controller exists
        }
      GENERATED = 1    #Number of rows produced
      INREC = INREC+1
      NEXT    #Exit from phony case
    ... #Omitted subroutine call statements
    5   #Expand|compress process
      #Compute input and output buffer pointers in IB and OB
      CALL EXPAND(BUFS(IB), ROW, BUFS(OB), ROW,
        GENERATED)
      #GENERATED is returned
      #As specified, this is an identity transformation
      NEXT
    ...
    10   #Write process
      #Compute buffer offset IB
```

```
        X = WRITER(OUTF, OUTREC, BUFS(IB), ROW, X)
        GENERATED = 0   #A dummy process code with NEEDROWS nonzero
                        #must follow the write in TODO, which being
                        #never able to proceed, sends the control back
        OUTREC = OUTREC+1
        NEXT
      } #End of phony case statement
      #A unit operation has now been done; arrange for the next
    READY(INDEX) = READY(INDEX)-NEEDROWS(PROC)
    #Number of unused rows still available to the current process
    #Rotate buffer pointers appropriately to reflect that GENERATED
    #rows have been created for the process in TODO(INDEX+1)
    BUMP = 1
    }#End of main loop
  END
```

CONCLUSION

We have discussed some techniques designed to increase the portability, modularity and structure in package software. We have divided the package software into pieces, separating out a machine-dependent kernel to provide standard operating system services such as program exchange, memory allocation and file access. We have illustrated the case of the program exchange feature for high-level process control. For a lower-level process control, we have suggested selecting a call to subroutine out of a fixed sequence. For composition of operations within a process, we have suggested an operation controller with buffer management to allow one operation to pass on its results to the next without the intermediate creation of files.

ACKNOWLEDGEMENT

Many of the ideas presented in this paper arose in discussions with David Milgram, whose experience with image processing and operating systems was invaluable.

REFERENCES

1. F. P. Brooks, Jr., *The Mythical Man-Month*, Addison–Wesley, 1975. (The essays 'The second system effect' and 'Plan to throw one away' are particularly appropriate)
2. J. M. Boyle and K. W. Dritz, 'An automated programming system to facilitate the development of quality mathematical software', in *Information Processing '74*, J. Rosenfeld (Ed.), North–Holland, 1974, pp. 542–546.
3. P. C. Poole and W. M. Waite, 'Portability and adaptabilty' (sic), in *Advanced Course on Software Engineering*, F. L. Bauer (Ed.), Springer-Verlag, 1973, pp. 183–277.
4. G. Leach and H. Golde, 'Bootstrapping XPL to an XDS Sigma 5 computer', *Software—Practice and Experience*, **3**, 235–244 (1973).
5. B. W. Kernighan, 'RATFOR—a preprocessor for a rational Fortran', *Software—Practice and Experience*, **5**, 395–406 (1975).

6. B. S. Baker, 'An algorithm for structuring flowcharts', *J. Assoc. Comp. Mach.*, **24**, 98–120 (1977).
7. B. G. Ryder, 'The PFORT verifier', *Software—Practice and Experience*, **4**, 359–377 (1974).
8. B. H. Liskov and S. N. Zilles, 'Specification techniques for data abstraction', *IEEE Trans. on Software Engineering*, **SE-1**, 7–19 (1975).
9. D. L. Parnas, 'A technique for software module specification', *Comm. ACM*, **15**, 330–336 (1972).
10. R. M. Haralick and G. J. Minden, 'KANDIDATS, an interactive image processing system', *Computer Graphics and Image Processing*, **8**, 1–15 (1978).