AD-750 897

Programming for Transferability

International Computer Systems, Inc.

prepared for

Rome Air Development Center

SEPTEMBER 1972

Distributed By:

NTIS

National Technical Information Service
U. S. DEPARTMENT OF COMMERCE
5285 Port Royal Road, Springfield Va. 22151

RADC-TR-72-234
Final Technical Report
September 1972

PROGRAMMING FOR TRANSFERABILITY

International Computer Systems, Inc.

**Approved for public release;
distribution unlimited.**

Rome Air Development Center
Air Force Systems Command
Griffiss Air Force Base, New York

UNCLASSIFIED
Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY (Corporate author)
International Computer Systems, Inc.,
10801 National Boulevard
Los Angeles, California 90064

2a. REPORT SECURITY CLASSIFICATION
UNCLASSIFIED

2b. GROUP
N/A

3. REPORT TITLE
PROGRAMMING FOR TRANSFERABILITY

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*
Final Report

5. AUTHOR(S) *(First name, middle initial, last name)*
Joel E. Fleiss, Guy W. Phillips, Andrew Edwards, Larry Rieder

6. REPORT DATE
September 1972

7a. TOTAL NO. OF PAGES
136

7b. NO OF REFS
44

8a. CONTRACT OR GRANT NO
F30602-71-C-0311

Job Order.No. 45940000

9a. ORIGINATOR'S REPORT NUMBER(S)
None

9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)*
RADC-TR-72-234

10. DISTRIBUTION STATEMENT
Approved for public release; distribution unlimited.

11. SUPPLEMENTARY NOTES
None

12. SPONSORING MILITARY ACTIVITY
Rome Air Development Center (IRDA)
Griffiss Air Force Base, New York 13440

13. ABSTRACT

This document presents the results of an investigation of design and documentation techniques used in computer programming in order to develop recommendations and guidelines for easing the transfer of computer programs written for one computing environment to another.

The first part of this study is concerned with a general transferability analysis and techniques to be utilized independent of any particular programming language. Emphasis is placed on the importance of transferability considerations during design and coding of the problem program.

The second section of the study deals with higher level and assembly/macro languages. Specific suggestions for improvement of FORTRAN, JOVIAL, and COBOL program design are included.

UNCLASSIFIED
Security Classification

14 KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
Information Processing Macnine-Independent Programming Device-Independent Data Access Algorithms Transferability						

PROGRAMMING FOR TRANSFERABILITY

Joel E. Fleiss
Guy W. Phillips
Andrews Edwards
Larry Rieder

International Computer Systems, Inc.

Approved for public release;
distribution unlimited.

ic

FOREWORD

This document is the final report on contract F30602-71-C-0311, Job Order number 45940000, by International Computer Systems, Incorporated, 10801 National Boulevard, Los Angeles, California. Rome Air Development Center, Griffiss Air Force Base, New York was the monitoring agency and Mr. James A. McNeely (IRDA) was the project engineer.

This report has been reviewed by the Information Office (OI) and is releasable to the National Technical Information Service (NTIS).

This technical report has been reviewed and is approved.

Approved: James A. McNeely
JAMES A. MCNEELY
Project Engineer

Approved:
FRANZ H. DETTMER
Colonel, USAF
Chief, Intel and Recon Division

FOR THE COMMANDER: Fred I Diamond
FRED I. DIAMOND
Acting Chief, Plans Office

ABSTRACT

Transfer of operational computer programs during changes from one computing system to another presents a serious problem throughout the industry. Duplication of software and reprogramming necessary to support the data processing activity must be minimized. This document is the result of International Computer Systems, Inc.'s investigation into design and documentation techniques for easing the transfer of programs from one computing environment to another.

The first part of the study is concerned with a general transferability analysis and discussion of techniques to be utilized independent of any particular programming language. Emphasis is placed on the importance of transferability considerations during design and coding of the problem program.

The second section of the study deals with higher level and assembly/macro languages. Specific suggestions for improvement of FORTRAN, JOVIAL and COBOL program design are included.

TECHNICAL EVALUATION

1. This effort, Programming for Transferability, was undertaken to establish guidelines to be used in writing computer programs for a given computing environment that would ease the transfer of these programs to another system with minimum reprogramming effort.

2. In the past, software has been machine-dependent. When the user of these programming systems must change to a new computing environment, he must convert all his existing programs. Often, the conversion consists of a complete redesign before his programs will run on the new system. This process is very expensive and must be corrected.

3. In an attempt to rectify this condition, (Ref RADC TPO 4), RADC decided to conduct an investigation of the characteristics of the various languages. This study resulted in the establishment of guidelines for the identification and separation of the machine-dependent functions from the machine-independent ones.

4. This effort provides the tools to be used in the isolation of these functions into different modules. If a programmer must use the machine dependent functions, only this module must be reprogrammed; thus, providing a reduction in cost of transferring programs from one system to another.

James A. McNeely

JAMES A. MCNEELY
Project Engineer

TABLE OF CONTENTS

TABLE OF CONTENTS

(Continued)

TABLE OF CONTENTS

(Continued)

SECTION 1. COMPUTER PROGRAM TRANSFERABILITY ANALYSIS

1.1 INTRODUCTION

1.1.1 The Problem of Transferability

Transferability is a measure of the ease of moving a computer program from one computing environment to another. Thus, if a program needs to be totally redesigned and coded when moved to a new computer, it has zero transferability. On the other hand, if a program can be moved to, and operate upon a new computer with no additional programming effort, it is 100% transferable between those computers. The degree of transferability of a program can refer to the specific case of two known computers or the general case where the new machine is unspecified. Unfortunately, in both instances, the amount of success encountered is often very small. There is promise, however, that techniques may be developed which will significantly improve this situation.

Many factors affect a program's transferability. These include the specific computers involved, the level of language used, the particular program application, and the techniques used in creating the original program design. For some of these factors, it is difficult to anticipate the requirements, and thus difficult to plan for ultimate program portability. Other aspects are more under the programmer's control and thus more conducive to planning. Nevertheless, even in those areas subject to improvement, little concentrated effort has been expended towards achieving a high degree of program transferability.

In general, the data processing community continues to be burdened by the apparent inability or unwillingness of computer manufacturers to produce greater computer compatibility among different machines. Some manufacturers have taken the initial steps towards compatibility among their own computer models while others have failed to take even this step to any meaningful degree. Meanwhile, compatibility among computers of differing manufacturers is virtually nonexistent except in a very few specific cases.

Software technology has also failed to produce sufficient progress towards alleviation of the program transfer problem. The industry's costs associated with program transferability have proliferated because no significant agency has provided the impetus for meaningful research in this area.

The software community has not provided the necessary guidelines or standards for program design, coding and documentation, and the hardware community has not provided equipment compatibility.

The lack of compatibility or standardization has cost government, the military and the business community huge sums for reprogramming and conversion. Similar losses are incurred due to continued usage of obsolescent equipment that must be retained because undertaking program library conversion would be too expensive and time consuming. The cost and time involved in making such transfers can seriously affect the feasibility and profitability of the systems involved. Every aspect of the computing environment may be affected in some way.

- Personnel may need retraining, including programmers, operators, data clerks and end users.
- The logical and physical form and flow of data may have to be altered (peripherals and computers).
- The overall processing strategy may change (job control).
- The operational software may need extensive internal modifications.

Frequently, when transfer of a program is mandatory and after exploration of all the alternatives, a decision to recode the program is made. At this point, unnecessary costs are often incurred because of inadequate documentation of the original design of the program and the high rate of employee turnover in the computer industry. Whereas reprogramming costs should be only a fraction of the original expenditures, more often they are very comparable.

Formalization of design and documentation procedures and recommendations specifically directed towards facilitating program transferability are steps that have been neglected by the data processing industry.

The purpose of this document is to suggest guidelines for controlling the cost and improving the efficiency of future program transfers.

1.1.2 Methods of Transferring Programs

This section lists and discusses existing methods of implementing transferable programs. The discussions are intended to expose factors that make transfer expensive.

The methods discussed are:

- Programming in a higher level language
- Manual and automatic code conversion
- Interpretive techniques
- Macro languages
- Rewrite

Since it is unlikely that a competitive new method that will revolutionalize all transferability problems will appear in the immediate future, this document focuses on improving the applicability and reducing the cost of existing methods.

Section 2 of this document discusses three higher level languages: FORTRAN, COBOL and JOVIAL. The syntax and semantics of these and other higher level languages, by design and evolution, facilitate their translation into the language of nearly any machine. Conversely, competitive advantage dictates that future machines will have the features and organization required to run programs written in higher level languages. As a result, some programs written in higher level languages can be transferred almost automatically to any machine for which the particular compiler exists. Exceptions to fully automatic transferability arise for a variety of reasons:

- There has been little pressure to avoid non-standard implementations - the apparent advantages of improving the languages outweighed the disadvantages of creating incompatibilities.
- Standard specifications have been variously interpreted.
- Even the standards permit certain details to be machine/system dependent.

- Certain environmental conditions cannot be standardized.

The normal method of transferring programs written in higher level languages is to provide and use "similar" compilers for the new machines and cope with the differences. Often a small percentage of statements must be found and manually replaced or altered. When a popular version of a language is replaced by an improved or standard version, an automatic conversion aid might be developed. An example is SIFT, which converts FORTRAN II to FORTRAN IV.

In regard to higher level languages, the intent of this document is to help avoid or isolate those usages which may be incompatible or non-transferable.

The low cost of transferring higher level language programs depends on the availability of a broadly-amortized compiler for the new machine. If an unpopular higher level language is chosen, a user may have to unilaterally bear the burden of new software development for the sake of transfer.

Manual and automatic code conversion is usually applied to converting assembly language source code to the source code for a different language, either the assembly language of a different machine or a higher level language.

Assembly languages are designed for a particular machine and provide some way of specifying every function and combination of functions that machine may perform. Assembly languages are most often used when no compiler exists for the machine; to specify functions not contained in the higher level languages; to avoid some highly general and hence inefficient code generated by compilers; or to avoid interference between compiler output and the desired program, (e.g. a FORTRAN editor-loader written in FORTRAN would have to be very carefully designed to stay out of its own way).

The actual method of transferring by code conversion is as follows:

- Determine a realistic development budget for the converter program.

- For each aspect of the source language, design a destination language equivalent. Unfortunately this is not usually an instruction for instruction equivalence.

- Select those aspects that can be handled within development budget or other constraints. Develop a program that will read source machine code. Identify aspects and extract parameters, either flagging troublesome aspects for manual attention or 'punching' the chosen target machine equivalent source code.

The wide disparity among computer architectures, especially with regard to addressing schemes, may often make it virtually impossible to design an effective program conversion tool. Typically, in this situation, only the invention of very clever correspondence algorithms can help to produce even partially efficient code conversion. For this reason, automatic conversion tools are sometimes only last resort or after-the-fact efforts at program transfer.

Assuming that automatic conversion is possible, the process of converting a program then entails the following steps:

- Manually edit the input source code, deleting or replacing functions that are unnecessary or inappropriate in the destination system.

- Process the program via the converter.

- Debug the result, manually editing the converter output using the warning and error flags provided by the converter, plus the ordinary test and debug procedures and aids.

- Check-out the program on new system.

Maurice Halstead, in "Using the Computer for Program Conversion", Datamation, May 1970, discusses the economics of amortizing a conversion aid program called a decompiler. He points out that a conversion aid handles the easier parts of conversion. To improve a conversion aid by 50% to 100% of its current capability requires as much additional effort as has already been expended. The residue needing manual conversion is increasingly difficult. As a result, a conversion aid appears to be fairly expensive before it becomes worthwhile, and must be amortized over a large workload. Furthermore, some degree of manual re-coding will amost always be required.

Figure 1 on the following page shows Halstead's conclusions in graph form. Note that the units cannot be arbitrarily chosen; 16 units of development effort corresponds to a converter that will handle 96% of the input code. If it is estimated that 32 man weeks would be required to do a 96% converter, then the units are 2 manweeks.

In another area, interpretive techniques, several software products on the market are economically viable because they are usable on and for a variety of machines. These techniques have been used to produce transferable meta-assemblers, compilers, and application packages.

The products themselves are written in special languages usually designed for two properties: The language is optimized for the convenient expression of facilities needed by the software product, and the language may be easily interpreted for a large number of machines in the anticipated market. The language is usually similar in format to an assembly language for a machine that would be ideal for the application.

To have such a software product running on a new machine usually requires only a few man months to write an interpreter for its internal language. For example, International Computer Systems' DUAL, a meta assembler, is implemented using a very advanced interpretive process. DUAL can be transferred to a new machine for 2% of its original development cost.

The transferability advantage quite outweighs any loss of operating speed (which can be considerable unless proper techniques are used) through interpretive execution. Note that some higher level language compilers produce object code which is largely a series of subroutine calls, and that the execution of such could be considered interpretive. Also, it should be noted that well implemented interpretive products can be made to execute within 10-20% of efficient assembly level code. This is very competitive with efficient (optimized) higher level language compiler generated code.

The macro assembly concept, on the other hand, permits the definition of languages having some of the transferability advantages of higher level languages as the object code efficiency of machine languages. Macro assemblers allow users to define ways in which source input is to be processed into forms recognizable by the assembler. To the extent that existing assemblers allow a common macro invocation format, a program written as a sequence

COST SAVINGS VS CONVERTER INVESTMENT

FOR BACKLOGS OF 100 AND 1000 UNITS

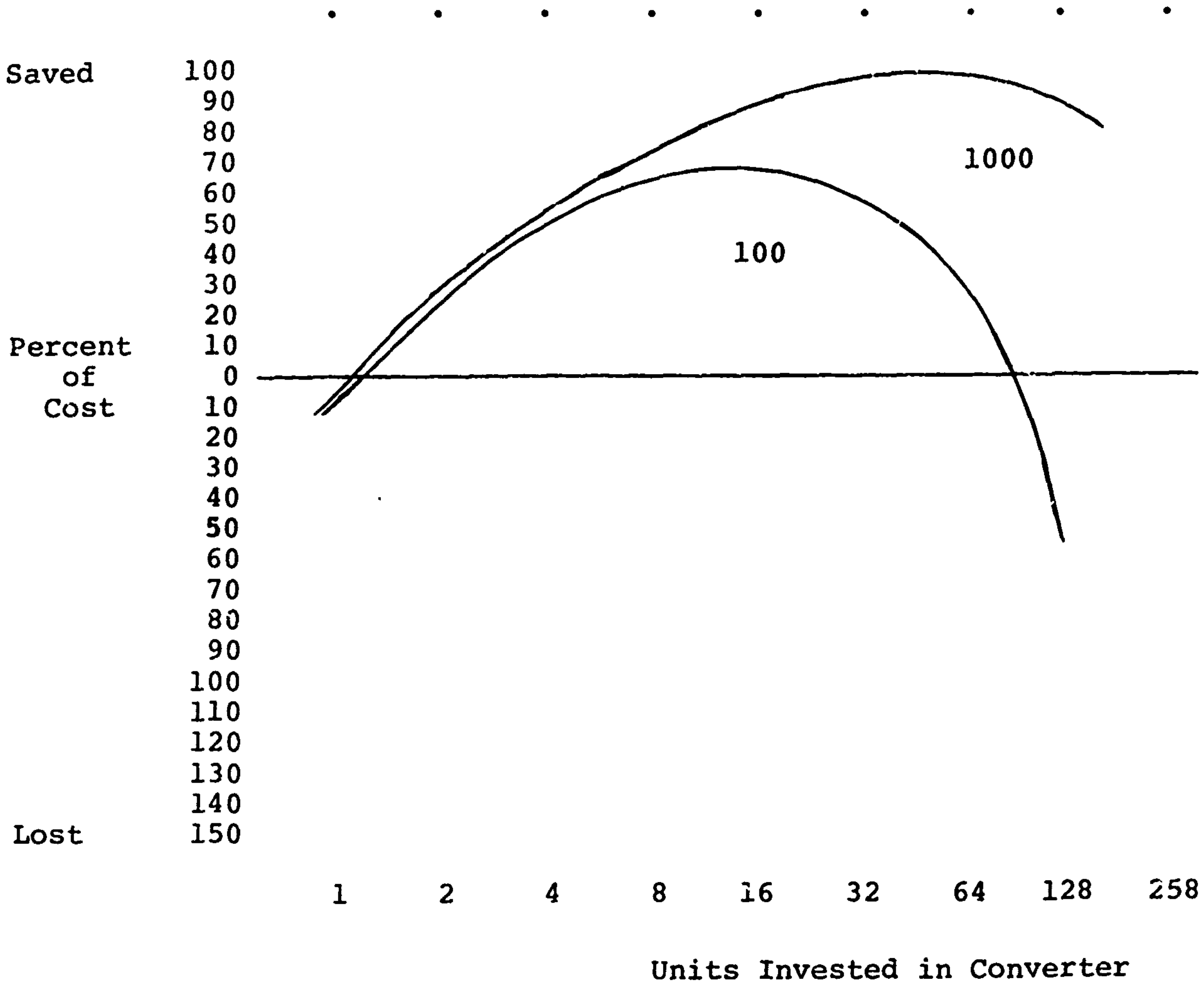


FIGURE 1

Converter Economics

of macro instructions can be transferred to any of several machines, simply by writing equivalent macro definitions for the various assemblers. The cost of writing the macros is far less than the cost of rewriting the program.

Various triple address pseudo-computers, for example, may be utilized with one set of macros. The code below shows a simple example of alternative expansions.

```
     ADD   AUGEND,ADDEND,RESULT

     CLA   AUGEND
     ADD   ADDEND
     STØ   RESULT

        or

     L     R,AUGEND
     A     R,ADDEND
     ST    R,RESULT
```

This technique now permits the coding of programs more efficient and nearly as transferable as those written in higher level languages.

A final alternative, rewriting a program or part of a program by hand, might seem desperate but can often be justified, especially if there is already strong incentive to re-design the program. From Section 1.1.3 the cost per use of converted and rewritten versions of the program can be estimated. The cost/use figures can be compared to the value/use estimates to give a convert/rewrite decision.

Dependencies may sometimes be isolated in a module (see Section 1.3.2 Modularity) which is expected to be rewritten. If such modules form only a small percentage of the total program while still absorbing a large part of the dependencies of the program, cost savings may result.

1.1.3 Transferability Economics

The natural figure of merit by which any transfer effort or technique may be judged is its cost effectiveness, specifically, the cost per use of the transferred unit. This document does not define effectiveness in any other way; it simply assumes that transfer is completed when the end user is as satisfied with the result as he was before the transfer. It may happen that for a particular case an incomplete transfer may give better overall results. That is, there may be a more complete definition of effectiveness. Also, this analysis makes no assumptions about whether cost/use is greater or less than value/use. Finally, it is assumed that the machine or system which is the destination of a transfer is itself no more and no less cost effective than the original, source machine or system. Management must assign its own weighing factors to the figures derived from this analysis.

In the following sections some simple formulas are presented along with tables containing estimates of relative values for substitution in the formulas. Some of these estimates are broken down into more detail later. Many of the suggestions presented include relative cost factors associated with system transfer and continued operation. In fact, the guidelines presented are those for which the cost factors are the most significant.

.ogram conversion is a combination of manual and automatic processes. For assembly language in particular, the automatic processing is done by a fairly expensive program which must be amortized over a large pool or program conversions. Existing conversion aids were designed to process a pool of programs which may not have been conditioned by application of the guidelines presented in this document; hence the conversion aids are more complex and expensive than they would need to be to process well-conditioned programs. Amortization accounting or pricing policies infer that programs that have been designed and coded for ease of transfer may not get the price break they deserve until there is a large enough pool of well-conditioned programs to support cheaper conversion aids. On the other hand, the cost of the manual part of the conversion process can be reduced immediately. Steps may be taken to reduce the amount of manual work (avoiding usages that the automatic process cannot handle) and documentation may be prepared to make the manual steps easier to follow. This may be of no benefit, however, when the price of the manual work is fixed at so much per card in the whole program and established for a work mix including ill-conditioned programs. Most commercial conversion services do offer an option under which the customer does the manual work. For well-conditioned programs this is probably advantageous.

The cost for each use of a program is the amortized program preparation cost plus the operating cost:

- $CU = P/N + \emptyset$

In this formula,

CU = Cost per use

N = The number of uses of the program, usually projected when estimating a preparation policy.

P = The preparation cost. This and operating cost will be broken down by method of preparation, by component programming task, by language and by application area.

$\emptyset$ = The cost of operating the program, both direct and indirect (e.g. requirement for additional core storage).

Three levels of program preparation for system transfer are distinguished for estimating purposes:

- Design and code from problem statement (complete re-write).
- Code from existing design (design transfer).
- Transfer the program largely intact (code transfer).

The following tables give general estimates of the relative costs associated with these techniques. For example, in Table 1A the least costly approach is to transfer a COBOL program from one system to another. Similarly, it is about 1-1/3 times as expensive to re-code a program in assembly language as it is in COBOL. All of these figures depend on the particular problem program, system being used, etc. They are intended only to indicate a general feeling for the relative costs most often incurred.

The 10/1 ratio for level 1/level 3 preparation of assembly language material, for instance, is simply the ratio $10/card (a widely used but unsubstantiated estimate) to a $1/card average charge for conversion services (see Appendix C). Note that the figures are related to each other only within a table and not between tables. Hence, the ratio of programming to operating costs is not estimated and is really peculiar to each program and machine.

TABLE 1A

Program Transfer Preparation Costs

	COBOL	FORTRAN	JOVIAL	ASSEMBLY
Re-write	75	60	60	100
Design Transfer	40	20	30	60
Code Transfer	1	2	5	10

The preparation cost of transferring code can be further analyzed:

- $\text{Cost} = A + O + (I + C + W) * N$

This includes automatic costs - the costs of use of a conversion aid are:

A = Amortized development cost
O = Operating cost

and manual costs - the costs of manual work are those of:

I = Identifying exceptions to automatic conversion
C = Comprehending exception code
W = Writing target machine equivalents

These are all multiplied by:

N = The number of exceptions.

It is important to note that there are three classes of exceptions:

- Those the converter can identify, but not deal with.
- Those the converter can flag as potential problems.
- Those that escape identification by the converter.

TABLE 1B

Program Operating Costs After Transfer

	COBOL	FORTRAN	JOVIAL	ASSEMBLY
Re-write	90	60	50	20
Design Transfer	90	65	50	30
Code Transfer	100	65	70	50

The excess cost of operating transferred code is a result of:

- Original code not needed in the new environment - usually because it is included in the operating system or in the new programming language library.

- Simulation of the methods by which the original machine performed a function, rather than duplication of that function.

1.2 MANAGEMENT POLICIES

1.2.1 Introduction

Several items to be considered in the formulation of a transferability policy were presented in the previous section with suggestions for an orderly weighing of alternatives. A concrete plan for the further formalization, application and enforcement of this policy must now be developed. Many questions remain to be resolved by management before such a plan can be implemented.

1.2.2 Policy Formulation

A complete transferability policy requires an initial statement of goals and an outline of the methods to be used in their fullfillment. Transferability considerations are first to be listed along with the basic project purposes, usually by the manager responsible for the programming budget. Realistic transferability depends not only upon the available standards and guidelines but also upon the judgment and experience of this individual.

The competitive elements of initial cost, operating cost, future cost, scheduling and general project feasibility must be carefully analyzed and balanced by the manager. Then, the methods proposed for attainment of the desired results must be carefully outlined. Initially, it is important to determine the availability and practicality of these techniques and to realize to what extent they will effect costs. Later, they may be refined as part of the ordinary design process. Explicit directions and constraints are thus laid down for subsequent workers. The level of detail should leave options, but not doubts. Uniformity across several program projects may be a very important subgoal in that it can spread the amortization of conversion aids and training. Hence, many of the basic aspects of program design must be delineated by management if an organized, coordinated transferability effort is to be realized.

The following questions should be considered in formulating a transferability policy. Although they are listed roughly in order of inquiry, not all of them may always be applicable.

- How important is transferability?

- Where does transferability rank among:

  - Meeting a tight deadline?
  - Meeting a tight budget?
  - Accomplishing a demanding development effort?

- What is the probability of future transfers?

- Do other decisions reflect the stated importance of transferability?

- Do schedules and budgets allow for the thorough, thoughtful job necessary?

  (Remember that if a program never attains executability, its other attributes are irrelevant. For this reason transferability (and maintainability and reliability) are at best secondary goals which will be squeezed out by unrealistically tight scheduling.)

- What effect will choice of computing system have upon transferability?

  - Regarding transfers to the system?
  - Regarding future transfers from the system?

- Does the system seem to be in the main stream of computing development? How similar are the probable transfer target machines?

- In choosing a language, where does transferability rank among:

  - Convenience of the language?
  - Applicability of the language?
  - Familiarity of the language?
  - Conformation with shop practice?

- What restrictions must be placed upon the usage of the language? Does a recognized standard exist and should it be used? Should extensions be used?

- Is the language still applicable with the restrictions?

- What methods will be used to effect a transfer?

- What aspects of the system lend themselves to program parameterization?

- Is the overall design of the program (or system of programs) suited to the selected transfer method?

- Should a library of system dependent routines be developed and what should it contain?

1.2.3 Policy Enforcement

In the programming environment, policy enforcement efforts may be focused on the following four areas:

- Task sequencing and emphasis
- Conflict between policy and practice
- Conflict between policy and technology
- Attention to details

There is a natural order for the performance of the subtasks in producing a program. A subtask is easier if performed while the information required is fresh from a previous or concurrent subtask. This is particularly true of documentation. The information contained in the documentation is substantially the same as that produced and required for design and code. In fact, projects large enough to be deemed difficult are probably best done by producing meaningful documentation at each step. Successive steps may then be guided by previous documentation.

In addition, postponed subtasks may seem anticlimatic and thus present morale problems when resumed. Hence, little should remain to be done after the program is checked out. Very few programmers care to work on a project after part of the staff have gone on to other things.

Once the subtasks are scheduled, each should be appropriately emphasized and carried to completion in its turn. (Some subtasks may be concurrent.) Whatever level of supervision is appropriate should be maintained.

Day-to-day management decisions constitute the actual management policy which may not in fact agree with the stated policy. Such decisions, therefore, should be examined for agreement with policy statements. One or the other may require change.

Policy may well be translated into statements of the expected performance rewards or penalties. In rewarding programmer performance, watch out for visible results achieved at the expense of less visible factors. If transferability is a goal, a piece of work may be finally evaluated only when transfer is attempted. If you reward a programmer for meeting a tight deadline, and penalize a maintenance programmer for finding the result difficult to deal with, morale suffers and maintainability/transferability policies do not in fact have management support.

It is best to reward the programmer whose work most accurately fulfills the goals set for the project.

Beware of trying to translate goals into a blanket set of rules once and for all. If the rules are not in fact the best way of meeting the goals, or if they become less than germane as time goes on, programmers will be frustrated and alienated. At least a periodic review of the rules is needed.

Moreover, defined requirements can be obeyed without actually fulfilling their purpose. An example is a program listing which contains the amount (percentage, number of lines, etc.) of comments and documentation required by management without conveying much understanding of the program. Documentation beyond the programmer's ability to communicate is wasted and harmful.

The coding of a program sometimes exposes weaknesses in its design. Hence, it may be found infeasible to code a program without certain transferability constraints, particularly if those constraints were established without consideration of the programming problem at hand.

If the transferability policy was established independently of a particular program or project, it can probably be redesigned and other methods found. If the policy was established for a larger system uniformly, the program in trouble may have to become an exception.

Even well documented transferability practices may get lost in a welter of details. This is particularly true if a programmer is inclined to take shortcuts or make ill-advised "improvements" on the policy. Unfortunately, examination of programmer output for compliance is tedious. Probably the best way to verify transferability is to transfer the program immediately, while problems can be more readily corrected and while the results can directly influence the programmer's future level of compensation and responsibility. Immediate transfer, of course, is not always possible or cost justifiable.

The programming language translator or precompiler can examine source code for factors that influence transferability. Whenever possible, compilers should flag usage of any non-standard language features that they implement. WATFOR (Siegel - see Bibliography and Synopsis) is an excellent example of fulfilling this function.

Presently, there are COBOL precompilers that enforce management-selected language usage constraints. Information Management Inc., for instance, has Magic Standards Enforcer which notes use of management-prescribed COBOL verbs. Arkay Computer Applications' MECCA flags about 200 management selected violations. Although largely oriented toward uniformity of the listings for the sake of readability, such products usually can flag use of any selected reserved words. These standard enforcement packages are generally machine specific, however, and thus non-transferable in themselves.

In the selection and use of standards enforcement products and techniques, it may be wise to consider one basic aspect of the computing environment. Namely, that programming originally demanded people who were clever and comfortable in an absence of established practice. The best of their innovations have become the current standards and recognized techniques. Some of their efforts, however, have left an aura of eccentricity which by now is somewhat misleading. It must be kept in mind, therefore, that policy enforcement is an ordinary management problem, not a manager-programmer adversary situation.

All in all, it is difficult to recommend a specific policy on transferability which will adequately suit every manager's needs, desires, budget constraints, and personnel relationships. Each project or installation executive must formulate a plan of action pursuant to his particular operating environment, personality and upper management dictates. For example, it was recommended that after initial coding an attempt should be made at program transfer as a test and possible measure of programmer ability and conformance to installation standards. In many instances complete transfer may be untimely, not cost-effective or in some other way undesirable. Hence, the manager may elect to inspect or re-compile a small routine - section (of his choice) or to discard the technique entirely.

Although a wide variety of management policies seems reasonable, a universal aspect has been shown to exist. Namely, some sort of plan must be formulated and enforced if any degree of program transferability is to be attained.

1.3 GENERAL PROGRAM DESIGN GUIDELINES

1.3.1 Introduction

This section presents general guidelines for program design approaches that will help minimize one or more of the transfer costs described in Section 1.1.2. These guidelines are usually applicable independent of the choice of coding languages.

Transferability is sometimes not the major consideration in program design. Fortunatley, transferability techniques seldom conflict with usual good design practices.

This following outline gives design steps to use in identifying transferability problems and chosing a method of solving them. This process goes on in parallel with the ordinary design process and must be repeated as often as necessary since there are transferability considerations at each level of detail. In a few cases previous decisions must be modified in light of new information, but this is not unique to transferability.

- Estimate the breakeven point for additional cost of a new program versus cost of transferring the original program.

- Identify the dependencies. List the factors which may effect the transfer attempt. These may include operating system facilities, hardware configuration, language and level of language, I/O paths, secondary and primary storage characteristics, data representation, instruction set, registers (if any), addressing, program language features, etc. If such a list is prepared for an installation, save it for future reference. The list can be made as the design progresses, but remember that it may overlook pertinent items in the early stages of design during which it is prepared. Many of the factors in the list are probably included in Appendix B.

- Categorize the items on the list in terms of their impact upon transferability. That is, if the item is absent or different on the next machine, one of the following applies:

  - The function being performed is no longer pertinent.
  - Redesign is required.
  - Recode is required.

- Code transfer is not precluded.
- Transferability effort is deferred until transfer time (for budget reasons, or if there is a reason to expect the problem will never arise).

- Select a way of dealing with each of the above categories. For example:

  - Loss of relevance becomes a system configuration constraint.

  - Redesign may be treated as a system configuration constraint, or it may be possible to isolate the problem area into a module slated for redesign.

  - Items for re-code should be isolated in modules to limit their effects on other areas (see modularity).

  - For untroublesome functions, parameterization and use of "standard" code are often enough to permit program transfer.

  - Deferred items should be isolated.

1.3.2 Modularity

Modularity is a formal way of dividing a program into a number of subunits each having a well-defined function and relationship to the rest of the program. There are well-known advantages in modularity; the one of most interest here is that those program functions which will need the most programmer attention upon transfer can often be isolated and functionally identified as distinct modules. The modules, if organized and documented properly, can be worked on with little reference to, or interference with, the rest of the program. Less critical areas of the program can then be converted using other, perhaps less expensive, methods.

In FORTRAN, modules may be subprograms which can be compiled independently and reconnected to the balance of the program at load time.

In JOVIAL and COBOL, the modules are subprograms and paragraphs respectively.

Assembly languages are also flexible in this regard although it is usually necessary to distribute some symbol definitions across modules that are assembled independently. This may be done either by preloading the assembly symbol table (the COMPOOL or DSECT concept) or by a load time symbol table (the external symbol concept).

An essential aspect of these modules is their isolation from the rest of the program. Each section should be a sequence of programming language statements having a well marked start and end. The function of the module and its interface with the rest of the program should be simple and well documented. A module performing several closely related functions may have several entry points. However, one entry point and a function parameter branch is usually better practice. Similarly, having program control enter a module in the middle simply to take advantage of a convenient piece of code should be avoided. That entry defines an additional function for the module which may be expensive to reproduce.

A program should be modularized by function and by nature of machine/system dependency. Modularization by function has been considered good practice in those languages that facilitate it.

The following is a list of modules into which a majority of programs may be organized. Each of them should be considered for splitting into two modules - one machine/system independent, the other containing the dependencies. In defining a machine dependent module, remember that the object is to render the environment of the independent modules invariant with respect to transfer:

<u>General Program Modules</u>

- Parameterization
  - Character sets, data masks, etc.
  - Machine architecture
- Initialization
  - Machine/System
  - Primary storage
  - Program
  - Files

- Processing

    - Interrupt processing
    - Data acquisition
    - Data editing
    - Data organization (primary storage)
    - Computation
    - Intermediate secondary storage organization
    - Data conversion/formatting
    - Output
    - Error checking and recovery

- Termination

    - Files
    - Normal end
    - Exceptional end

This modularity outline is, by the nature of this discussion, very general in concept. More specific suggestions on program organization are presented in the individual sections concerning FORTRAN, JOVIAL, COBOL, and Assembly languages.

1.3.3 Parameterization

Parameterization is a method by which a problem program may self-adjust to a program, hardware, or system modifications. Some numeric or symbolic items in the program may require alteration if the same function is to be performed in a new environment. These values may be written into the code explicitly or they may be parameterized. In the latter case, symbolic variables are used in place of actual values. These symbolic variables are set to the applicable values in an initialization phase of the program execution or during assembly/compilation. The values chosen may be directly initialized by programmer coding or they may be set up or calculated by the language processor.

If the value is written in by the programmer, it should be in a well marked statement, with all such statements collected in one place. The documentation for each such value should be explicit in how and when a new value is to be determined for the parameter. If the value is to be computed, the computation must be checked out for a range of values.

Parameterization may also be used to avoid usages which may not be supported by all compilers for a language. For example, not all FORTRAN compilers support variable array sizes in subroutines. The usage may be avoided by passing array sizes as parameters and calculating index values (if multi-dimensional) explicitly. Or, if there is a possibility that the organization of an array is not standard (stored by rows rather than columns) a switch can be used to select the appropriate calculation. The value of that switch can be determined by the program during initialization.

Some assemblers provide for conditional assembly in which the value of a parameter, defined near the start of the program, may be tested for skipping of alternative code sequences.

Although a transferred program usually performs an unaltered function on the new machine, parameterization may be used even more extensively when the program function is altered during transfer. This is the case for the parametric programming language processors mentioned in Section 1.1.3, Methods of Transfer.

Parameterization requirements depend on the details of the programs involved. Each actual value in the code must be considered for parameterization as it is being written. If the value and the representation of that value are independent of any reasonable change in machine applications, scope, or language, or if the programming language does not permit a symbolic value in that context, then parameterization does not apply.

In general, over-parameterization is rarely a problem, especially in the assembly or compilation process. Thus, the cost of parameterization is nontrivial only if the logic of the program or usage of the language must be perverted. The value of parameterization, on the other hand, can be considerable. Individual examples of this fact are discussed in depth in other sections.

1.3.4 Code Constraints

Program transferability can be greatly enhanced simply by avoiding code that is difficult to transfer. Finding and using alternative code, however, does involve additional programming costs which must be measured against the benefits obtained.

For higher level languages, the primary code constraints of concern involve the avoidance of language features which are apt to be missing or altered, or which will give different results when compiled on the transfer target. As discussed in Section 1.5, competitive advantage, implementor taste and standardizacion ambiguities contribute to lack of basic language compatibility.

The language features most likely to be missing or different in other compilers are the language extensions and the expensive or little-used standard features. Unless a language was specifically extended to suit an application, use of extensions can be avoided with only a limited loss of efficiency. In those cases where the standard language is inadequate, the unusual usages should be isolated in modules for recode.

Aside from subsets and extensions, two things in general make code difficult to transfer - techniques that bind to a particular representation of data structures and instructions, and those that bind to a particular sequence of operations. A binding technique, then, is any usage that takes advantage of or is cognizant of itself, when any portion of any instruction is used as an operand, or when any address is used which is not completely symbolic. Code may become cognizant of operation sequences because they define a function which must be reproduced when the program is transferred. In further discussions in Section 2, the effects of these techniques will be called code restraints.

Program code which modifies itself has proven to be bad practice for reasons of maintenance difficulty, error proneness, and non-re-entrant properties. Recent machines and languages have taken steps to render self modifying code both unnecessary (e.g. execute instructions) and difficult or impossible (e.g. write protect). Higher level language code does not directly allow dependencies upon instruction representation.

Generally, unless the machine is deficient in indexing and other dynamic address functions, representation-dependent code may be easily avoided. Furthermore, resource conservation realized through use of this type of code usually amounts to only an insignificant percentage.

For any particular machine, external data representations correspond to predictable internal storage bit patterns. It is seldom the case that those representations will be equivalent over a variety of different machines. If the external representation used does not correspond to the function of the item, the internal representation will become inappropriate upon transfer. An example of this type of code is character identification by comparison with small decimal integers.

Similarly, packing several data items in a word will bind the code with respect to word size, as a data item may be split by a word boundary upon transfer. For example, 3 items of 6 bits each do not repack well into a 16 bit word.

Operation sequence and data representation constraints can also be combined in ways which may seem of great value but which definitely hinder transferability.

It is often the case that an assembly or higher level language does not explicitly provide a facility needed for a particular application. The facilities that are provided can be used in combination to synthesize the required facility.

Such synthetic facilities become of transferability interest in several ways. It may be that one of the facilities used in the synthesis is not present in the transfer target. Or, it may be that the corresponding combination of facilities on the new machine does not have the same effect. (In this case the original synthesis is called idiomatic.) On one machine, for instance, a fixed point number can be converted to floating point by adding and subtracting special floating point constants. The technique is very convenient and very machine dependent.

In any case, converting a synthetic facility component-wise can be less efficient than finding or synthesizing the equivalent facility in the target machine or system. Nearly all machines provide some form of shift instruction for use in synthesizing facilities (usually bit-resolution or data addressing). Arithmetic and logical operations are defined and stabilized by widespread usage, but the shift concept is rather computer unique and is usually defined at the convenience of the computer architect. As a result the shift instruction set is often a transfer problem.

The difficulty of transferring synthetic facilities may be reduced by use of macros in assembly language. The macro invocation statement becomes equivalent to the use of an explicit rather than synthetic facility. The facility may be transferred by using a new synthesis or, if present, the explicit facility, in the macro definition. Fortunately, the number of facilities that need to be synthesized is less than the totality of all useful macros.

The principles described for modularity and parameterization applies as well to the synthetic facilities.

1.3.5 System Configuration Constraints

A program may be dependent upon aspects of system configuration in ways that make it infeasible to transfer the program to a system in which those aspects differ significantly.

Those system attributes which become severe constraints are usually either system resources which are heavily used or devices which provide facilities not qualitatively present on the new system.

Primary storage is likely to be the constraint most often felt. If a program is written for a machine with a large amount of core storage, it may be infeasible to move to a machine with less storage unless the program was initially written with this in mind. Similarly, programs often do not adjust well to an overlay/nonoverlay transition since basic algorithms may differ (overlay processing often requires a file pass for each phase, where this would be unnecessary with adequate primary storage). On the other hand, large primary storage may have been crucial to efficient operation of the program, and it would be wasteful not to use it.

One approach to this problem, then, is to consider (and document) primary storage size as a system constraint precluding code and perhaps design for transfer to a machine with less storage.

Similar considerations apply to random vs sequential access secondary storage devices.

In the area of input/output, efforts must be directed toward avoidance of explicit references to physical devices which can permanently bind a program to a particular configuration. Programmers should design and code with logical entities such as the system input device (rather than the card reader) or the object output file (rather than the tape punch). Physical devices and their associated characteristics (such as record length) must be parameterized and isolated from the main body of the program as much as possible.

Even stronger constraints apply to programs which are intended to exploit particular system aspects. It would be pointless, for instance, to transfer a graphics processor to a system without graphic I/O capability. This type of constraint can be considered (and documented) as categorically apart from others. In this case, no partial loss of inefficiency can eliminate the constraint.

1.3.6 Transferability Feasibility Analysis

In some cases it is not immediately clear whether program transfer will be cost effective. Hence, it is desirable to compare potential new machines with respect to their suitability for transfer of the existing library. A guide to making such decisions must take into account the cost of the machine and its effectiveness for running the transferred library.

If it is assumed that the system configuration constraints are met, and that an acceptable transfer method must be available (or at least is well enough specified for estimating purposes), then the cost to be dealt with becomes the operating cost of the transferred program. This in turn depends on the cost of the resources provided by the system and the burden that the program places on those resources.

The resources treated here are time, primary storage space, and peripheral device utilization.

The total operating cost is computed as:

Cost =

$$\left(\frac{\text{memory cost}}{\text{word.sec}} \text{ x words} + \frac{\text{CPU cost}}{\text{sec}} + \frac{\text{device cost}}{\text{device.sec}} \text{ x devices}\right) \text{ x secs}$$

Words, secs and devices represent the space, time and peripheral device support needed to run the program. The time and space units are arbitrary - minutes or hours may be more convenient.

CPU cost includes all those items without which the system cannot be used, including software.

Memory cost is treated separately. Additional primary storage is often optional or separately priced, and programs often become larger upon transfer.

The term

$$\left(\frac{\text{device cost}}{\text{device.sec}} \text{ x devices}\right)$$

should be estimated for each peripheral device or fraction thereof dedicated to the program and summed to obtain total peripheral device costs.

When considering a maximum amount of primary storage or a maximum complement of peripheral devices, the estimates can be made for the worst case deemed feasible.

Effect of Transfer Upon Size and Speed

Table 1B in Section 1.1.3 gives rough estimates of relative operating costs by level of transfer method (redesign, recode, convert code) and language. Those estimates may be revised appropriately for specific applications.

Halstead (loc. cit.) notes that assembly language programs increases in size by as much as 30% upon code conversion. As a first approximation it may be assumed that the number of instruction executions will increase in the same proportion. Hopefully, programs that were designed and coded with transfer in mind will not grow as radically.

Estimates of higher level transferred code efficiency may be refined by recording the relative size and speed figures of benchmark programs run for a new machine.

Similarly, assembly language code may be estimated from knowledge of conversion equivalents. An assembler which tabulates instruction usage counts is very helpful in this respect. For instance, if linear instructions are converted to loops, instruction execution counts will increase more than proportionately to the code size decrease.

1.4 GENERAL PROGRAM DOCUMENTATION GUIDELINES

1.4.1 Introduction

In most cases, conventional program documentation will require certain additions and changes to facilitate program transfer. This section describes the additional documentation by purpose, information content, and utilization.

This additional information is to be distributed over and incorporated in existing standard installation documentation formats. A completely new style might not be suitable in some circumstances. Particular sections, such as those for FORTRAN or JOVIAL programs, will contain comprehensive suggestions as to the content of the documentation. They will be especially important in regard to identifying system dependencies.

1.4.2 General Format

Documentation efforts toward more transferable programs can be divided into four parts:

- Independent aspects: These are descriptions that do not change when the program is transferred. This includes the purpose and overall functioning of the program as well as any details that are clearly machine/system independent.

- Program dependencies: This section covers those things that may change from one machine/system to another. In other words, it will describe how the program is machine dependent. This information will be used by the programmer in determining what form these aspects will take in future machines and systems.

- Specific implementation: This should describe the form taken by dependent aspects for a particular machine/system. The specific environment should be clearly identified.

- Transfer plan: This section describes the intended method of transfer for a program and a particular implementation. Note that several specific implementations of a program may not be equally transferable to a new machine. The transfer plan should be specific as to which version of the program is expected to be transferred.

Among the sections described above, those dealing with independent aspects and program dependencies can theoretically be written once and then remain valid for all specific implementations. In practice, aspects thought to be independent may be found to be dependencies when the actual transfer is attempted. Specific implementation material, however, will clearly need to be re-written for each new version. Similarly, the transfer plan may be partly specific to a particular implementation.

Depending on the volume of documentation required, the number of dependencies involved, the frequency of transfers, and similar factors, several physical formats may be used. Perhaps the most widely acceptable and convenient, however, would be the current installation standard. As necessary, each paragraph could be divided into independent, dependent and transfer plan subparagraphs. The specific implementation material most logically would appear in an appendix.

If the documentation is updated after all projected transfers are completed, the specific implementation material could be listed at the appropriate place in the main text, with each item identified as to pertinent implementation.

1.4.3 Additions to Conventional Documents

The following paragraphs review the usual organization of program documentation. Each level and type of document is identified by purpose and customary content, and additional transferability information appropriate to each type is described.

Overview/Introduction

This section should satisfy the needs of persons, typically in management, who need not become involved in the details of the program. These individuals may, however, need to review this material with regard to future efforts in the application area, cost predictions, proposals and similar management activities.

This documentation usually includes identification of a program by name, purpose, and position in a larger system. It should also contain:

- A description of capabilities which are system-independent.

- System configuration constraints.
- A description of optional, system dependent capabilities.
- A description of the expected transfer methods with, at least, relative transfer cost estimates.
- A description of the relative importance of the program to an overall system.

This additional information may be used to estimate the feasibility of transferring to various new systems.

Product or Functional Specifications

This portion of the documentation specifies in detail what the program does. It should also include any aspects of the program not covered in the introduction but that may be of interest to users and management. In particular, the following transferability information should be added:

- System configuration constraints.
- The relative independence or dependence of each program function-application.
- The nature of all system dependencies.
- The degree of effort directed toward transferability.
- Projected transfers and methods of transfer.
- The proportion in which total transfer effort is divided between intial development and actual transfer time.

Design Specification - Internal or Program Logic Manual

This portion of the documentation describes in complete detail how the program works. It includes narratives, flowcharts and program listings. It is a working document, successive elaborations of which result in the finished program. Also, it functions as a training aid to the maintenance/transfer programmers who may be required to become as familiar with the program as its authors were originally.

The following paragraphs describe the usual organization of the document along with the extra material required for transferability.

The design/logic document is usually replicated heirarchically. That is, there is a section for the program as a whole and similar sections for each component subroutine. These include narrative, flowcharts, and for some sections, listings.

It must be kept in mind that the design/logic section is a working document rather than simply an archival record of a completed design. The process of writing the first draft of this document is the process of design of the program. Difficulties encountered in describing the transferability considerations for a program or subroutine in this section should be taken as a warning of similar difficulties to be experienced in actual transfer of the program or subroutine.

The narrative included in design specifications generally contains information such as:

- Purpose of program/subroutine.
- Role of this routine in the program or system and entry/exit details.
- Data handled by this routine and the input/output details.
- Parameters required/generated.
- Tables built or used.

The narrative continues with a detailed description of the algorithms by which the primary function and subsidiary functions are performed.

The flowchart should be a pictorial representation of the narrative (rather than of the listing). For the program as a whole, most of the flowchart boxes will be subroutines. Each box should indicate, by subroutine name or listing label/line number, where detailed information may be found.

Transferability Information

Section 1.4.2 recommends breaking documentation into four subsections: machine independent section, dependency section, particular implementation and transfer plan. For a properly designed program, the machine independent section will be as similar and complete as ordinary documentation. In particular, if some function cannot be properly documented in the machine independent narrative, then the design should be modified. If that is not possible, a configuration constraint is indicated.

Items in the machine independent writeup for which transferability is a consideration should be flagged. Ordinary textual cross reference conventions are suitable for this; parenthesized note numbers are suggested.

The note numbers can then be used in the dependency, particular implementation, and transfer plan subsections to relate items.

The scheme outlined above is used for the program as a whole, and for each subprogram. Key information should be repeated in the introduction to each subroutine; in particular, a subroutine which constitutes a machine independent module should be so identified in the introduction to that subroutine.

The following example shows the mode of expression in each of the subsections:

- Machine independent writeup:

  ...the subroutine then finds the leftmost comma in the character string (5) and returns the next four characters...

- Dependency writeup:

  (5) The speed of the program depends heavily upon the speed of this search - a basic assembly subroutine is indicated.

- Particular Implementation for IBM S/360

  (5) Translate and test instruction is used.

- Transfer Plan

  (5) Since the speed of the transferred program is an expected problem, manual rewrite of the code paragraph labeled SEARCH in subroutine CHARSEL is anticipated for minimum execution time.

Flowcharts

The scale and information content of a proper flowchart should be such that the chart parallels the machine independent narrative. A flowchart more closely related to the code than to the narrative does little to bridge the intelligibility gap between narrative and code, and is more susceptible to change during program debug, maintenance and transfer.

Listings

The document should be organized and annotated so that it is easy to find code lines corresponding to flowchart blocks or narrative sentences.

Code lines may well be broken into logical 'paragraphs' by use of comment lines. Any line or 'paragraph' that is machine/system dependent should have comments explicating the dependency and detailing the transfer plan. Occasionally, alternative code exists which would be more transferable but which is for some reason not justified in this particular implementation. Such code may be displayed as comments, in the appropriate location.

Code of particular interest at transfer (or maintenance) time should be indicated by comment lines having scan value, i.e. readily seen when leafing through the listing.

Manuals

Usually, a separate document is prepared for the guidance of personnel operating the computer while the program is being run. It details the input, intermediate and output files by physical characteristic, source, disposition, and the operating system information requirements. It also lists error and exception conditions, and the dialogs between operator and operating system.

The machine/system independent portion of the document serves as an introduction to the functional purpose of operator actions and information responses.

Since a machine-independent operating system cannot be assumed, it is expected that the operator interface will be very machine-dependent. Thus, a dependency writeup would be limited then to a description of the operational differences of various systems among which regular transfers actually take place. That is, if the program is to be run interchangeably on several levels or versions of ostensibly the same machine/system, the dependency writeup may detail the differences which affect the operation of the program. Additionally, the manual should include detailed writeups for the particular system(s) on which the program is run regularly.

The transfer plan portion should describe the mutual program/system/operator interface. This is best done in functional terms such that operating procedures and job control language statements can be designed for a new system.

To avoid confusing operators, the individual machine requirements and the transfer plan should probably be bound separately.

A user oriented manual must also be prepared. It will be clear from the nature of the program whether or not the users will be professional programmers. In any case, the document should be complete enough for use by a stranger to the program.

The machine independent part of this document will include all facilities expected to be transferred. Functional terms should again be used in descriptions.

The dependency section must allow the user to foresee problems that may be encountered with particular features if the program is transferred. It is not intended that the user resolve or be responsible for program transfer. He is entitled, however, to an understanding of the status of the program with respect to the risks or delays that may be associated with its use.

The particular implementation section should fully describe all system dependent items, especially the actual data formats and conversational terminal protocols.

The transfer plan should describe in detail whether and how the user will be able to move his data files to a new system if necessary.

The recommended documentation plan, therefore, involves additions to current installation standards rather than major revisions to existing techniques. In this way, the amount of emphasis placed on transferability can vary not only by installation, but also by program.

1.5 OTHER TRANSFERABILITY TECHNOLOGIES

This section touches upon several subjects which are related to transferability but which are either of less general applicability or are covered elsewhere in the literature.

1.5.1 Microprogramming

A microprogram is a program to be executed by a rather simple computer for the purpose of simulating a more powerful computer, which in turn executes application programs. The hardware need only interpret and facilitate the micro instructions which are constrained for ease of execution rather than the ordinary non-micro instructions which are designed for the convenient expression of application algorithms. Often, as with the IBM 360, the user may not even be aware that the machine is microprogrammed.

Microprogramming has never accounted for a very significant percentage of total programming efforts. The percentage does appear to be on the increase as a result of emphasis on emulation and because of burgeoning mini-computer sales. Nevertheless, because of the small percentage of total programming that is in this area, and because of the nature of microprogramming, program re-write is the most general means of program transfer.

Generally, micro level instructions are very much oriented to the particular machine that they are a part of, and consequently not very amenable to being transferred to other computers. It seems to be the nature of microprogrammed applications that the requirement for program transfer occurs less frequently. Still, many of the techniques applied to assembly code factors in order to attain portability will be directly applicable to similar microprogramming factors. Hence, some of the assembly level conclusions may be applied to the microprogramming factors.

Microprogrammed computers can actually provide significant benefits in some transferability situations. These computers normally have built-in facilities for readily combining their basic instructions into more powerful instructions. In this manner, a specific microinstruction computer can be made to operate like certain other computers. When a set of regular assembly level programs are to be transferred to a microinstruction computer, this approach may be feasible. Often though, the creation of a correct emulator in the micro-computer can itself be a fair size task. Incompatibilities of word and character size, the absence in the new machine of internal data paths and registers not essential to its original architecture, and differing I/O devices all limit emulation efficiency. Similarly, operator interface facilities such as console switches cannot readily be simulated. Hence, microprogrammed machines are by no means a panacea for transfer problems.

1.5.2 Interpretive Languages

As with microprogramming, the total amount of interpretive programming represents only a small percentage of all programming activities. On the other hand, the employment of interpretive programming techniques seems to be on the upswing as can be evidenced by the increased usage of BASIC and APL, both of which are generally compiled and executed in an interpretive manner. Actually, in many instances utilization of interpretive instructions and languages may provide the most promising approach for strictly maximizing program transferability.

An interpretive language is essentially a series of subroutine calling sequences, and the interpreter a set of subroutines. One central subroutine, analogous to the instruction fetch circuits, handles subroutine linkage, selecting the next call in accordance with sequence logic (usually a location counter), isolating parameters, and branching to the designated subroutine. The pseudo-machine code can be quite compact and efficient sin the addressing facilities need not be general but can be accurately tailored to the application.

The major shortcomings of programming with interpretive code is that program execution speed will always be less on a specific computer than the equivalent program coded at the assembly level by a good programmer. In some cases, recent refinements in interpretive languages and interpreters, in conjunction with increased computer speeds, have reduced the execution speed difference to minor significance.

The major advantages of the interpretive approach are:

- Significantly reduced program transfer costs for programs or program libraries.
- Significantly reduced core requirements.
- Faster (than higher level language) execution speed.
- Facility for inclusion of debug aids and measurement tools.

An attribute common to the more successful interpretive languages is their approach to memory design, usually referred to as stack processing. The stack processing approach is to have those tables that are of variable length share the available memory space. That is, the tables are allocated initial starting locations, and as a table touches another, the other tables are moved to make space. Tables in this type of memory arrangement are called stacks.

When interpretive techniques are used, the transferability burden is centralized with the designers of the interpreter and the interpretive language. Usually, programmers using these languages have the fewest restrictions imposed by transferability. With a well designed and standardized interpretive language, the greatest transferability imposed restrictions on design and documentation will be in the area of peripheral equipment utilization, and even there the effect will be minimized.

By carefully designing and standardizing the interpretive language, transferability of a program library can be accomplished in a very small time frame. The essential task required to accomplish transfer of a set of interpretive programs is the implementation of the appropriate interpreter for the new computer. This can require from one man month to a few man years of effort, depending upon the particular language. Generally, the effort involved in creating an interpreter is only a small fraction of the effort of creating a traditional higher level language compiler.

It should be noted that many of the guidelines presented in Section 2.4 may be used for both interpretive language design and for interpreter design. These promote transferability of user programs, simplify creation of future interpreters for the same language, and reduce the deleterious effects that interpretive operations have upon execution speed.

Finally, many of the problems associated with standardizing traditional higher level languages also exist with interpretive languages. This is evidenced by all the extensions that manufacturers have made to BASIC and that virtually all APL implementations are for a special version of APL. Were these language standards more comprehensive and acceptable, the propagation of variations of these languages would probably be reduced.

1.5.3 Conversion Tools

The subject of source to source converters was discussed in Section 1.1. The information presented there substantiated the fact that there is a significant economy of scale in the development of conversion aids. The larger the pool of programs to be converted, the more the converter costs can be justified. Primarily for this reason, the number of commercial conversion aids currently available is relatively small. Similarly, the enormous possibilities of combinations of systems, machines and languages dictates that no comprehensive set of conversion tools will ever be developed.

From a transferability standpoint, therefore, automatic program conversion tools must be considered expensive, generally unapplicable and after-the-fact techniques on which to base future programming efforts. For the programs that already exist, however, there are a number of aids which may apply.

Decompilers/Disassemblers

A special problem is presented when the running object program is not identical with currently maintained source decks. This may result from extensive patching of assembled code or dropping, misplacing, or pirating from a source deck. There are products available which will disassemble the object code to produce source statements. Generally, they are limited to very few machines and are not completely effective. Decompilers, especially, would be enormously expensive if they were to handle (efficiently) all possible combinations of object code. Again, use of this type of conversion aid is generally the result of a lack of planning for transferability in original programming efforts.

Source to Source Conversion

A source-to-source converter is a program which can identify constructs in one source language and build an equivalent construct in another source langauge, transcribing the pertinent parameters. The equivalence is always programmer designed. Certain items may not have an equivalent in the other language, or the translation is too subtle or too inefficient to be worth the effort for developing converter capability for it. In these cases, the converter should flag the item for programmer attention.

Compatibility/Compliance Monitors

Higher level language programs are transferable only to the extent that they do not use language features that are different or missing from the compiler of the destination machine.

Programs are presently available which inspect COBOL source code for compliance with management-selected restrictions. By restricting usages which are apt to be incompatible, management can reduce future transfer difficulty.

Such monitors can be developed for any programming language. It is more efficient, however, to incorporate these features in the original compilers which must completely analyze source code anyway. The WATFOR FORTRAN compiler mentioned elsewhere is an example of this approach. Unfortunately, few compiler implementors seem willing to admonish customers against the use of competitively motivated language features.

Usage Reference

The cross reference listing produced by many assemblers and compilers can easily be made to include information on the use of particular language features. For example, the locations and frequency of use of instructions and addressing methods such as indirect would be a definite aid during program transfer.

1.5.4 Language Standards

Perhaps the most fundamental step toward transferability of programming languages is standardization. This opinion is reflec - ed by the existence of standards committees such as those of ANSI (American National Standards Institute) which have been organized for most popular languages. It has long been recognized that substantial savings in programmer retraining, manual preparation, and program transfer can be achieved through language compatibility. Unfortunately, there are many problems associated with this endeavor. Among these are:

- Inability to completely and clearly define the language
- Subsetting
- Extensions
- Dialects
- Enforcement

Most of the work in language definition has been in the area of syntax. The semantics and practical usage descriptions of most languages leave much room for improvement. There are often ambigious and generalized statements which must be interpreted by individual implementors. Similarly, some groups, motivated by efficiency or competitive advantage considerations (maybe even ego), will seek to "improve" upon standards by making changes to the rules.

In other instances, hardware characteristics such as characters per word may cause the imposition of non-standard conventions (such as name size) for the sake of operating efficiency. This may be encouraged by the common contract agreements to process a specified number of cards per minute or to operate within a given core partition, etc. This type of alteration is usually referred to as a dialect.

Another area where hardware plays an important part is in language extensions. Often the ability to utilize special hardware such as direct access devices can present a considerable short-term competitive advantage. Similar arguments and justifications can be given for application oriented additions to a language. This form of non-standardization is not readily controlled by the customer-user.

Hence, standards committees are faced with the ambitious task of concisely defining a language which will exactly suit everyone's needs. Of course, a conglomeration of all the opinions, techniques and desires is not a realistic goal.

Standards committee must usually omit some useful language features because not all compiler implementors can afford them. Even so, some features included in the standard may be left out of some implementations as too expensive. Thus, subsets of a language come into existence.

The COBOL standard seeks to resolve the subset problem by defining standard subsets as well. Despite this, few COBOL compilers actually conform even to a standard subset. A possible solution is to define a fundamental minimum language with a mechanism whereby extensions may be defined.

Even if all the individual requirements could be resolved, there would still be the problems of enforcement. More will be said about this aspect and the extent of current non-adherence to established standards in the sections on higher level languages.

1.5.5 Universal Machines

The transferability problem could be effectively eliminated by removing all the differences among machines, systems, and languages. It must be pointed out, however, that had blanket standardization of hardware been adopted when it was first suggested, the machines of the 1950's would still be in use.

Computer architecture is still being improved. Some features presently very common are demonstratably less cost effective than some newer developments.

The marketplace, itself, seems to impose the optimum constraint upon architectural change. Customer acceptance of a new machine depends, in part, upon the advantages of a new architecture outweighing the disadvantage of difficult program transfer. As program libraries grow, transferability becomes more important to the users, and hardware changes are inhibited unless they are clearly cost effective.

Furthermore, the ever growing variety of applications and requirements for computer systems will certainly continue to obviate complete hardware stagnancy.

The concept of the universal languages as related to hardware differences is discussed in Section 2.4.

1.5.6 Operating Systems

Operating system differences are often the major transfer problem when standard higher-level languages are used. The operating system and the language used to control it are usually more complex than any other element of the system. Further, there seems to be no constraint upon operating systems diversity.

Operating systems and job control languages are the subject of a study by Applied Data Research Corporation, and the reader is referred to their report.

1.5.7 Data Bases

A production program usually represents a continuing process with several files read regularly and updated periodically. If the program is transferred, and the process is to continue on the new system, the files must be moved also. This represents a problem in several ways:

- The organization of the data may reflect the organization of the original machine and be inappropriate to the new machine.

- The hardware determines physical format of the data and thus incompatibility with new hardware.

- The operating system usually has certain conventions about labels, retrieval index formats, tables of contents, record sequence, etc. Many of these conventions are considered transparent to the user and are ill-documented.

- The structure of the data may be implicit in the code, rather than explicitly described. This is really a program transfer problem,but it can be alleviated by some measures that alleviate the data transfer difficulties.

Data base considerations are covered in depth in other reports (ADR Study).

SECTION 2. PROGRAMMING LANGUAGE GUIDELINES

This section presents specific design guidelines for programs to be implemented in the higher level languages FORTRAN, JOVIAL and COBOL. Suggestions for assembly and macro language program preparation are also included in a separate chapter.

Each of the languages is treated independently. Once the implementation language is chosen for a particular program, the appropriate section may be reviewed prior to actual design and coding. In most cases, references to other portions of this document will not be necessary.

The following tables are referenced in the discussions of dialects. They represent a possible method of assignment of costs to the use of language extensions versus adherence to recognized standards.

Levels of Impact on Transferability-Rework Necessary to Conform to Standards

T1 - Readily available user routines may be substituted
T2 - Requires occasional coding changes or conversion tools
T3 - Requires minor design changes and re-coding
T4 - Requires extensive re-coding
T5 - Requires major re-design
U - Unlikely to be a problem

Inconvenience Costs of Adhering to the Standard

L1 - Minor additional programmer effort required
L2 - Additional coding required
L3 - Major additional programmer awareness/effort
L4 - Major additional coding required
L5 - May be no substitute - management decision

2.1 COBOL PROGRAMMING FOR TRANSFERABILITY

2.1.1 Introduction

The development of COBOL (COmmon Business Oriented Language) began as the result of a desire for a business oriented source language that was widely compatible among host computers. Formal work started in 1959 under the direction of language experts from several major computer manufacturers. Maintenance and organized changes to COBOL have since become the responsibility of a group called CODASYL (Conference on Data Systems Languages). The most significant aspect of this history is that the original design committee was composed of representatives from competing manufacturers who were attempting to create a language usable on many unrelated computers.

Achieving maximum compatibility on existing equipment has always been the goal of COBOL development. Application areas were intended to be confined to business data processing with no attempt to generalize the capabilities, though some implementations are tending toward expansion of scientific programming features.

COBOL users can encompass a wide spectrum of individuals. The relatively inexperienced programmer would enjoy the naturalness of COBOL, while the supervisor or business manager would find that the inherent readability of programs provides very adequate documentation. The machine level programmer, as a matter of fact, might be unhappy with the verbosity required; however, the ability to document programs while composing them and to understand what has been done when they are completed certainly is an asset to later machine transfer and re-coding.

COBOL is divided into four parts: IDENTIFICATION, ENVIRONMENT, DATA, and PROCEDURE DIVISIONS. The IDENTIFICATION DIVISION is really self-explanatory and for the most part transferable across many compilers. The ENVIRONMENT DIVISION is intended to be the focal point of the machine dependent qualities in COBOL programs. Thus, an attempt is made within the language to isolate compatibility problems. The DATA DIVISION, which describes records, files, and program data, experiences the most difficulty in system transfer. It is in this area that extensions, interpretations and more subtle compiler dependencies are concentrated. The PROCEDURE DIVISION, which handles the bulk of problem program execution, can be made relatively mobile if care is taken to parameterize implementation dependent attributes via the other divisions.

As with all languages, compatibility problems do exist in COBOL. The following sections will try to point some of the ways in which these difficulties can be minimized.

2.1.2 Dialects

The first formal definition of COBOL was released in 1960. That version is now referred to as COBOL-60. Subsequent versions are called COBOL-61, COBOL-62, etc. The ANSI standard COBOL description used in this document was published in 1968 and shall hereafter be referenced as "the Standard".

As in most modern languages, COBOL does not remain unchanged. Some aspects become obsolete while others are augmented with new words, clauses and usages. Often this evolutionary process results in several ways of doing the same thing. Since too much flexibility creates confusion and increases the burden of new compiler implementation and testing, many COBOL users have adopted subsets. Software products such as MECCA 360 have been developed to precompile programs and enforce this type of restriction. In an attempt to control and clarify this trend, the Standard has adopted the following elementary terminology which allows for three levels of capabilities. Specific functions within COBOL are referred to as modules. A particular module can attain one of three levels. 0 indicates no capability. In the following table, n represents the level supported by a particular compiler.

	Specific	Level
Module	Implementation	Range
Nucleus	n NUC 1,2	1-2
Table Handling	n TBL 1,3	1-3
Sequential Access	n SEQ 1,2	1-2
Random Access	n RAC 0,2	0-2
Sort	n SRT 0,2	0-2
Report Writer	n RPW 0,2	0-2
Segmentation	n SEG 0,2	0-2
Library	n LIB 0,2	0-2

The full standard involves complete implementation of all modules. The Air Force has defined subsets A, B and C as follows:

Subset	Description
A	1 NUC, 1 TBL, 1 SEQ
B	2 NUC, 2 TBL, 2 SEQ, 2 RAC, 1 SEG, 1 LIB
C	2 NUC, 2 TBL, 2 SEQ, 2 RAC, 1 SRT, 2 RPW, 1 SEG, 1 LIB

Basic differences between module levels is summarized in this table:

1 NUC	Basic math operators such as ADD and SUBTRACT
2 NUC	COMPUTE verb, options to internal verbs such as ROUNDED, and DATA DIVISION enhancements such as level 66 in PICTURE
1 TBL	One level subscripting
2 TBL	Three level subscripting
3 TBL	SEARCH, SET, KEY, ASCENDING/DESCENDING, INDEXED BY
1 SEQ	Basic file processing
2 SEQ	Control of storage allocation and file labels
1 SRT	Linkage to generalized sort program and access to records before and after sorting
2 SRT	Multiple sorts
1 SEG	Overlays
2 SEG	Variable limit assignments
1 LIB	COPY
2 LIB	COPY REPLACING

Random access and report writing seem self-explanatory; they are either full implementation or null. By this time nearly all major manufacturers have implemented at least Subset B on large scale machines.

There are also many factors which contribute to extensions of the COBOL standard. Disk files and random access, a desire for more computational ability via floating point number representations and built-in functions, international currency considerations, debug aids, buffer allocation control, differences in collating sequences, a desire to abbreviate, flags to the system, and modularity in subroutine compilation are examples of such influences.

As mentioned elsewhere, the use of extended capabilities can be severely detrimental to program transferability. The prime uniformity which exists among these "improved" versions is their choice of unique names and methods of implementation. In other words, an extended COBOL compiler can not guarantee that all other augmented implementations will be compatible. Clearly, many features which are quite useful for a specific application or facility must be discarded if program and programmer portability are to be maintained.

The orderly updating of the Standard and adherence to current Standard features (which hopefully will become subsets of the the improved Standard) is strongly recommended.

In the meantime, several large scale computers maintain highly extended COBOL compilers. The following charts attempt to present some of the more obvious deviations from the Standard. The COBOL compilers referenced (IBM 360, CDC 6600, UNIVAC 1108, Burroughs B5500) are all medium to large scale. Refer to the introduction to Section 2 for an explanation of the transferability and avoidance cost symbols.

STANDARD DIALECT DEVIATIONS CHART 1

Input/Output Organization	Computers	Transferability Cost	Avoidance Cost
Debug Aids: TRY EXHIBIT TRACE MONITOR DEBUG CLUSTER-DUMP	IBM 360 B5500 UNIVAC 1108	T3	L2
Inherent Printer Controls: LINES-PER-PAGE LINES-AT-TOP LINES-AT-BOTTOM LINE-SPACING CHANNEL IS PRINT-SWITCH APPLY TO FORM-OVERFLOW	UNIVAC 1108 CDC6600 B5500 IBM360	T3	L2
Direct Access Clauses: ORGANIZATION IS DIRECT,RELATIVE REWRITE RECORD KEY IS MD (disk file description) SYMBOLIC KEY IS ACTUAL KEY - track number SYMBOLIC KEY - record in track number TRACK-AREA IS APPLY RESTRICTED SEARCH OF n TRACKS	IBM360 B5500	T4	L3-L5
Buffer Allocation: APPLY POOL,DEMAND, STANDBY SIZE IS n BLOCKS BLOCKS CONTAINS	UNIVAC 1108 CDC 6600	T3	L1

STANDARD DIALECT DEVIATIONS CHART 1

(Continued)

Input/Output Organization	Computers	Transferability Cost	Avoidance Cost
Labeling Clauses:	UNIVAC 1108	T3	L1
VALUE OF ID IS	CDC6600		
PAD CHARACTER IS	B5500		
EDITION-NUMBER IS	IBM360		
DATE-WRITTEN IS			
REEL-NUMBER IS			
RETENTION-CYCLE IS			
ENDING-LABEL IS			
MULTIPLE FILE ID (MFID) IS			
SAVE-FACTOR IS			
CLOSE WITH PURGE			
ON LABEL ERROR			
Record Classification:	B5500	T3	L1
TECHNIQUE A	CDC6600		
RECORDING MODE IS $\begin{pmatrix}\text{BINARY}\\ \text{DECIMAL}\\ \text{XS3}\end{pmatrix}\begin{pmatrix}\text{HIGH}\\ \text{LOW}\\ \text{HYPER}\end{pmatrix}$ DENSITY	UNIVAC 1108		
RECORD CONTAINS n CHARACTERS DEPENDING ON RECORD-MARK			
APPLY WRITE-ONLY ON FILE			
CONTROL WORDS ARE			
FILE CONTAINS ABOUT			
DISPLAY-ST (sterling)			
Inherent Device Names:	B5500	T3-T4	L3
DISK	CDC6600		
KEYBOARD	IBM360		
CONSOLE			
PAPER-TAPE			
PHYSICAL TAPE			

STANDARD DIALECT DEVIATIONS CHART 1

(Continued)

Input/Output Organization	Computers	Transferability Cost	Avoidance Cost
Inherent Device Names: (Continued)	B5500	T3-T4	L3
SWITCH n	CDC6600		
MOD (model of disk)	IBM360		
SYSIN			
SYSOUT			
SYSPUNCH			

STANDARD DIALECT DEVIATIONS CHART 2

Data Representation and Organization	Computers	Transferability Cost	Avoidance Cost
Additions to PICTURE Structure:	UNIVAC 1108	T4	L3
H (computational type)	IBM360		
E (floating point)	B5500		
C D /£ : s d (sterling currency)	CDC6600		
J (sign overpunch)			
Hyphen (insertion character)			
OCCURS n TIMES DEPENDING ON			
Non-PICTURE Data Definition and Editing:	UNIVAC 1108	T4	L3
POINT LOCATION (PT) IS n PLACES	B5500		
SIZE IS n DIGITS,CHARACTERS			
CLASS (CL) IS NUMERIC SIGNED			
ZERO SUPPRESS (ZS) LEAVING n PLACES			
FLOAT DOLLAR-SIGN (FS)			
CHECK PROTECT (CP)			
MOVE A TO A-B 28,8,20 (packing specifications)			
Collating Sequence and Character Set:	IBM360	T3-T4	L2
TRANSFORM FROM TO	CDC6600		
UPPER-BOUND(S)			
LOWER-BOUND(S)			
Unusual characters < > ≤ ≥ ¬ ≠ % & : @ [ ] ←			
Abbreviations:			
CMP (COMPUTATIONAL)	IBM360	T4	L1
PC (PICTURE)	CDC6600		
SY (SYNCHRONIZED)	B5500		
VA (VALUE)	UNIVAC 1108		
ZS (ZEROES)			
JS (JUSTIFIED)			
SZ (SIZE)			

STANDARD DIALECT DEVIATIONS CHART 3

Procedural Differences	Computers	Transferability Cost	Avoidance Cost
Conditional Statements: Relationals: GR,LS,GQ,LQ,EQ,NQ,NGR,NLS UNEQUAL,EXCEEDS IF OTHERWISE THEN clauses IF IF clauses	CDC6600 UNIVAC 1108 B5500	T4	L1
Separately Compiled Subroutines: LINKAGE SECTION ENTER LINKAGE CALL SUBROUTINE USING COMMON-STORAGE SECTION USE FOR ENTRY POINTS ENTER FORTRAN ROUTINE REFERENCING EXECUTE EXITING	IBM360 UNIVAC 1108	T3-T5	L5
Built-In Functions: SIN,COS,ARCTAN,EXP,SIGN,SQRT,LN,ABS	B5500	T1	L3
System Flags: PERFORM WITH (multiprocessing) PERFORM (progl, prog2) (multiprocessing) ASSIGN TO MEMORY (multiprocessing) DEFINE (define user verbs) INCLUDE (paragraph inclusion) SECTION PRIORITY RERUN EVERY END-OF-REEL RERUN ON TAPE EVERY n RECCRDS END-OF-JOB	B5500 UNIVAC 1108	T4	L2-L5

STANDARD DIALECT DEVIATIONS CHART 3

(Continued)

Procedural Differences	Computers	Transferability Cost	Avoidance Cost
Additions to Computational Capabilities COMPUTE - FROM EQUALS EXPONENTIATED BY MULTIPLIED BY MINUS PLUS X (multiplication) DIV (integer division) MOD (modulo division) JUSTIFIED LEFT (illegal)	UNIVAC 1108 B5500 CDC6600	T4	L3
Subscripting (more than 3)	B5500	T3-T5	L3

A secondary result of extensions to a language is the proliferation of compiler reserved words. Even though a program may be coded in strict adherence to the Standard, it may produce errors when run on another system if programmer defined names conflict with the new compiler's extensions. An attempt to delineate all possible reserved words seems unreasonable. The following partial list, however, may give the COBOL programmer some insight intc name formation techniques to avoid. In particular, the avoidance of expansions or abbreviations of standard reserved words is a good practice.

Extended List of COBOL Reserved Words (Non-Standard)

ABOUT
ABS
AN
APPLY
ARCTAN

B-5500
BEGINNING
BEGINNING-FILE-LABEL
BEGINNING-TAPE-LABEL
BINARY
BIT(S)
BLOCK-COUNT
BLOCKS
BOTTOM
BOTTOM-OF-PAGE
BZ

CALL
CARD-PUNCH
CARD-READER
CHANGED
CHANNEL
CHECK
CL
CLASS
CLUSTER-DUMP
CMP
CONSOLE
CONSTANT
CONVERSION
COS
CP
CREATION-DATE
CYCLE

DEBUGG
DECIMAL
DEMAND
DENSITY
DESTRUCTIVE
DIAGNOSTIC-FILE
DIGIT(S)
DIRECT
DIRECT-ACCESS
DISK
DISPLAY-ST
DIV
DOLLAR
DRUM(S)
DUMP

EDITION-NUMBER
END-OF-FILE
END-OF-JOB
END-OF-TAPE
EQ
EXCEEDS
EXECUTE
EXHIBIT
EXITING
EXP
EXPONENTIATED
EXTENDED

FASTRAN
FORMAT
FORM-OVERFLOW
FORTRAN
FS

GQ
GR
GREATER-EQUAL

HYPER

IBM-360
INCLUDE
INTERNAL

JS

KEYBOARD

LEAVING
LESS-EQUAL
LIBRARY

Extended List of COBOL
Reserved Words (Non-Standard)

(Continued)

LINKAGE
LN
LOAD
LOCATION
LOW
LOWER-BOUND(S)
LP
LQ
LS

MASS-STORAGE
MD
MEMORY-DUMP
MFID
MINUS
MOD
MONITOR
MULTIPLIED

NAMED
NGR
NLS
NQ

OC
OH
ORGANIZATION
OTHERWISE
OV
OVERFLOW

PAD
PAPER-TAPE-PUNCH
PAPER-TAPE-READER
PC
PER
PHYSICAL-TAPE-NUMBER
PLACE(S)
PLUS
POINT(S)
POOL
PREPARED

PRINTER(S)
PRINT-SWITCH
PRIORITY
PROTECT
PT
PUNCH
PUNCHB
PURGE

RANGE
READY
RECORD-COUNT
RECORDING
RECORD-MARK
REEL-NUMBER
REFERENCING
RELATIVE
RENAMING
RESTRICTED
RETENTION
REWRITE
RG

SAVE-FACTOR
SENTINEL
SEQUENCED
SIN
SN
SORT-FILE
SPACING
SQRT
STANDBY
SUBROUTINE
SUPERVISOR
SUPPRESS
SWITCH
SY
SYMBIONT
SYMBOLIC
SYSIN
SYSOUT

Extended List of COBOL Reserved Words (Non-Standard)

(Continued)

SYSPUNCH
SZ

TAPE-PUNCH
TAPE-READER
TECHNIQUE-A(Z)
TEST-PATTERN
THEN
TRACE
TRACK-AREA
TRACKS
TRANSFORM
TRY

UNEQUAL
UNISERVO
UPPER-BOUND(S)
UTILITY

WORD
WRITE-ONLY

XS3

ZS

2.1.3 System Environment

As mentioned earlier, a great deal of emphasis was placed on machine independence in tne developm.nt of COBOL. Certain attributes of the operating environment will always be unique however. Many of these aspects that depend on the physical characteristics of . specific system are isolated in the ENVIRONMENT DIVISION of the COBOL program.

Since it was intended to be machine-oriented, a significant part of this division must be re-coded each time a system transfer (or configuration change) is made. The other divisions are nc completely relieved of dependency problems, however.

System characteristics do have some effect on the methods of data description in the DATA DIVISION. The file description, for example, contains references to recording techniques such as tape labeling and record and block sizes. Similarly, the results of SYNCHRONIZED clauses depend on boundary alignment and word organization features of a particular machine.

To a large extent, the PROCEDURE and IDENTIFICATION divisions require little change during system transfer provided that dialect extensions have been avoided.

Perhaps the most troublesome aspect of COBOL transfers in general is the input/output function. The SPECIAL NAMES clause of the ENVIRONMENT DIVISION can be used to equate implementor defined device mnemonics to programmer defined names used in the PROCEDURE DIVISION, but other less obvious problems are not so easily resolved.

The IOCS (Input/Output Control System) requirements for each machine are unique and produce a wide variation in record formatting and file organization. These differences may not be of concern to the problem programmer while he is coding for one particular system, but they will become apparent when transfers are attempted.

Each operating system has its own unique file control block (fcb) to which information is supplied by the FD statement and other sources. Often sub-parameters are left null to be filled in by the system at execution time. This group of language features is therefore somewhat system dependent. Many computers have more than one way to record information. RECORDING MODE, for instance, is mandatory for IBM and unimplemented by UNIVAC. DATA RECORD and RECORD CONTAINS are sometimes ignored and treated as comments while other compilers check their syntax even though the clauses cause no specific action. In any case, the words used and the physical effects generated are unique to each manufacturer.

An example of a problem which may arise is the transfer of a program written for a word-oriented computer to a character-oriented machine. Sometimes, the number of characters in a record need not be a multiple of the number of characters in a machine word. However, on many computers, the smallest unit that can be handled is a word. In this case, if a data entry does not use all the characters of the last word on a data record, the whole word is output. When read back into the same computer, the extra characters are ignored. However, when the same record (assuming constant density, blocking, etc.) is read into the character-oriented machine, the extra characters become the start of the next record. Hence, when a transfer is contemplated, it would be wise to re-code the record description to include a FILLER ENTRY to receive the extra characters.

More serious conversion problems do exist. Consider the compatibility of BCD variable length and multiple internal representation fixed word length systems. Clearly, such situations are hard to understand let alone resolve. A suggested technique to minimize these nightmares is to use DISPLAY numerics (which are the most standard) for all files that may experience machine transfer. If computational fields appear in the program, they must be converted via a MOVE to an intermediate DISPLAY file before output.

Because some record formats may be extremely uncompatible (different character sizes) it may be necessary to code intermediate assembly language utilities.

The general problem of input/output errors is best left to the system, but first time read errors after OPEN should be considered by the programmer. They are most likely record length problems, and the program should be able to tell if correct data is present before self or involuntary abortion (this can be handled with DECLARATIVES).

Another suggestion is to place input/output in a SECTION by itself and PERFORM it. This method is best for ease of maintenance.

A list of miscellaneous factors pertaining to the system environment follows.

- REVERSED input is not universally available.

- Integer or identifier names are preferable to implementor supplied mnemonics.

- CLOCK-UNITS differ from machine to machine.

- FILE LIMITS are hardware dependent.

- LABELS are implementor defined.

- ACTUAL KEY clauses are hardware/implementor dependent.

- DATE and TIME may be a single word or require an input card.

- The effect of OPEN and CLOSE on internal operations is determined by the individual system.

- BLOCK, RECORDING MODE, and RECORD CONTAINS vary greatly.

- Special characters ≠ ≮ : < > ≤ ≥ and others are not implemented on many computers.

- Card column punches and their internal representations are not universal.

- Quotation marks may consist of either ' (IBM) or " (RCA).

- APPLY may be required for all, some, or none of the files.

- Optimization specifications (SAME AREA) will differ.

- File identification within multi-file and multi-reel tapes may be a problem.

- The collating sequence may no longer be valid.

- The precision available in numeric calculations may no longer be adequate.

- Core space may be less.

- The number of peripherals available may change.

- The printer line size may be reduced.

- Sense switches may not exist.

- Console interface may not be permitted.

- STOP may not be allowed.

- Report headings should be kept under 120 characters or, if necessary, coded as 120 characters and 12 characters combined.

2.1.4 Modularity

COBOL users may justifiably claim that it is one of the most "modularly organized" higher level languages in current use. A survey of the syntax makes this obvious. The segregation of many system dependent attributes to the ENVIRONMENT DIVISION and further separation of hardware oriented aspects of the problem program into DATA and PROCEDURE divisions is evidence of this inherent modularity.

Any programmer will testify to the rigidity of style necessary to produce acceptable DIVISION, SECTION, PICTURE and paragraph structures. Though somewhat of a nuisance to the application programmer, this division of functions within a COBOL program is a great aid to clarity, maintenance and overall portability.

Although the dummy argument-subroutine capability of FORTRAN and the desirable feature of modular compilation are not available in standard COBOL, most business applications do not seem to suffer greatly. The use of the PERFORM verb and its attendant options allow for great flexibility of design. Many machine dependent functions such as I/O may be assigned to independent paragraphs and utilized from various points within the program.

The principle example of this technique would be the isolation of OPEN, CLOSE, READ, WRITE, ACCEPT, DISPLAY and associated format conversion routines.

Other languages called up via the ENTER directive should also be subroutinized in this fashion.

Also, some non-standard features, because of their power and ease of use, may find their way into a program. If this is the case, the user should be aware that special treatment will be necessary at conversion time. This would be greatly facilitated by the preparation of a special area of usage and a documentation package entitled "Non-standard features utilized - minimum subset necessary". The chances are that someone other than the original author will do the conversion.

Another area where suspected machine dependent items may be grouped is WORKING-STORAGE (put these items first). Even an old character set/collating sequence may be entered here eventually for data translation.

The COBOL feature with the greatest potential for alleviating transferability re-work is the COPY directive. If many programmers can use standardized (within the installation) descriptions in the ENVIRONMENT DIVISION, SPECIAL NAMES, FILE CONTROL and I/O CONTROL SECTIONS, a great deal of redundant coding could be eliminated. These standards could be done once, entered into the library and called via the COPY option. For example,

```
FD PAYROLL-HISTORY COPY LIB-01
```

would constitute a complete file description. Even though sometimes the formats do not show it, library calls may be utilized by almost every section of a source program. PROCEDURE library modules could contain a majority of the I/O discussed earlier.

A typical summary of the forms of COPY available is included below.

In the Environment Division:

SOURCE-COMPUTER.	COPY library-name.
OBJECT-COMPUTER.	COPY library-name.
SPECIAL-NAMES.	COPY library-name.
FILE-CONTROL.	COPY library-name.
I-O-CONTROL.	COPY library-name.

In the File Section:

FD file-name	COPY library-name.
SD sort-file-name	COPY library-name.
01 data-name	COPY library-name.

In the Working-Storage Section:

```
    01 data-name            COPY library-name.
```

In the Report Section:

```
    RD report-name          COPY library-name.
    01 data-name            COPY library-name.
```

In the Procedure Division:

```
    procedure-name          COPY library-name.
```

2.1.5 Parameterization

Almost all computer programs possess one common attribute; they will require change. Modifications to design or programming techniques, revisions to customer or application specifications and hardware/system transfer all contribute to this problem. Particularly susceptible to these events are table sizes, hardware device names, arithmetic and format literals, the structure of files, printer control and the scaling and precision of numeric data.

Needless to say, it would be extremely unwise to ignore these possibilities. The judicious use of compile time parameterization, on the otherhand, can eliminate much of the individual analysis and modification usually necessary in a system change.

COBOL does not contain an over proliferation of parameterization possibilities, but the careful programmer can produce a more transferable product if he makes full use of what is available.

The following list should be considered (and augmented) before beginning a new program.

- Define all literals that are used often in the PROCEDURE code in the WORKING-STORAGE SECTION via level 01 entries. This should also be done for I-O CONTROL statements.

- Avoid literals in arithmetic statements using ADD, SUBTRACT, MULTIPLY, and DIVIDE.

- Use data names rather than literals in loop controlled PERFORMs.

- Parameterize your own carriage control characters and use them in ADVANCING clauses.

- Make full use of the SPECIAL NAMES paragraph to assign mnemonic names to hardware devices and switch states.

- Recognize the FILE-CONTROL paragraph as a parameterization of hardware device names.

- Quote marks are often non-compatible - check for compiler control card input to adjust for this.

Apparently missing from COBOL's parameterization capabilities are provisions for OCCURS phrases (table size) and PICTURE structure (size, scaling and precision). These items require re-coding for each change.

2.1.6 Code Restraints

Every COBOL compiler implementation will contain individual author and machine eccentricities. It is very important that knowledge of these idiosyncracies is not used to produce unobvious or "trick" results. There is no guarantee that successive compilers will exhibit the same results or that the "clever" programmer will be around to patch things up.

In addition, unexperienced programmers may not be aware of the existence of alternate interpretations and mechanizations which could be applied to their coding. For these reasons, the following list is included. It is a brief summary of techniques which may create unpredictable results when transferred.

- Implied subjects in IF statements, though usually acceptable, may cause ambiguous meaning when used with NOT. For example:

  ```
  IF X NOT LESS THAN 6 OR GREATER THAN 9 GO TO PAR1
  ```

  may be interpreted as

  ```
  IF X NOT (LESS THAN 6 OR GREATER THAN 9)
  ```

  in Burroughs implementations, or as

  ```
  IF X (NOT LESS THAN 6) OR GREATER THAN 9
  ```

  in IBM compilers.

- Omission of terms in conditional compounds may be acceptable, illegal or unpredictable. For example,

  IF ANNUM EQUAL 1,4,5, OR 9

  is permissible in some compilers.

- Multiple conditions IF IF should be avoided.

- Do not assume that if the GIVING phrase is omitted in MULTIPLY X BY Y that the result will replace Y.

- Use ON SIZE ERROR whenever numeric data may be close to 0.

- Be careful about utilizing very long records in I/O.

- READ statements executed after execution of an "AT END" clause may produce bad results.

- SYNCHRONIZE clauses are hardware dependent regarding the format of external media (tape) containing items of this type and the generation of implicit FILLER when the preceding item does not terminate on a word boundary.

- No implied background characters should be assumed; un-initialized areas may result in random printer control characters on output.

- No rules are given to control the way in which arithmetic is done; thus the compiler may perform calculations in binary, BCD, packed demical, etc. Only the result is required to be in the proper form. This implies the following:

  - Limit arithmetic operations to two operands when possible to avoid intermediate result differences.

  - Do rounding by adding 5 X 10^n where n is one greater than desired accuracy.

  - If there are more places to the left of the assumed decimal point than in the data item assigned to the answer, extra digits will be dropped or execution may stop. Therefore, use SIZE ERROR options when this is a possibility.

. COMPUTATIONAL and COMPUTATIONAL-n internal representations vary greatly. For example, Honeywell COMP-1 is binary and COMP is packed decimal while IBM is just the opposite. The programmer may well try to use binary items exclusively.

- Comparison of data of unequal type and size produces non-uniform results.

- Do not MOVE groups to elementary elements due to varying justification and size problems.

- Do not MOVE blanks to numeric items and expect 0's.

- MOVE ALL to a field containing a decimal point may or may not terminate the MOVE at the decimal point.

- LABEL processing is not standard.

- Automatic end of reel and end of file processing is done in many ways; take care in these situations when transferring programs.

- Do not assume that the IF name GREATER THAN "Z" statement will be a safe numeric test even if it does save time and core. A standard COBOL collating sequence has not been defined. Hence the following holds true:

  Special characters < A-Z < 0-9 for IBM,CDC,RCA,XDS

  while

  0-9 < Special < A-Z for GE and Honeywell.

- Differences in collating sequence produce differences in LOW-VALUE and HIGH-VALUE computations.

- The card codes for + = " ' ( ) ; @ may have differences between hardware while * - / $, . are common.

- All numeric fields used in computations should be defined as signed.

- If a numeric field is to be used mathematically, define it as COMPUTATIONAL.

2.1.7 Policy

When some reasonable adherence to a standard is enforced, COBOL source programs can be coded in such a fashion that transferability problems between various machines and compiler implementations are minimal. Much of this power is due to the inherent separation of DATA and PROCEDURE coding and the relative word/byte size independence of the data description. Compiler restrictions on ENVIRONMENT descriptions, when supplemented by programmer efforts, also contribute to the recognition and isolation of system dependent features.

Frustratingly often, it is individual compiler interpretations and extensions and not the hardware that introduces idiosyncracies which interfere with smooth, rapid program transfers. Operating system interfaces (which are not dealt with here) also provide conversion difficulty even when running on the same hardware. In order to overcome such restraints, a small loss in execution efficiency must be accepted.

In general, the use of a compiler's extensions will result in incompatibility to any other compiler. Any shorthand or non-use of Standard required syntax should be considered an extension and avoided. This includes abbreviations such as PC for PICTURE and implied subjects in IF statements. In addition, special character shorthands such as >for GREATER THAN should not be used since some machines don't have these characters. Similarly, even though a number of compilers accept * in the continuation column as a NOTE, many do not.

Another general suggestion is to always use at least one alphabetic character or hyphen between the first and last characters of a name (some implementors reject all-numeric procedure names for instance).

A form of abbreviation which may not be overlooked is the COMPUTE verb. Arithmetic computations can be done with less space and time if COMPUTE is used, but intermediate result handling (especially truncation) is implemented in a variety of ways. Many of the techniques are very machine dependent and final results may vary from machine to machine. If consistency of results is important, the basic ADD, SUBTRACT, MULTIPLY, and DIVIDE verbs with two operands should be utilized.

In one of the previous sections, the use of DISPLAY characters for final output (to another compiler/machine) was recommended since this form of data representation is the most compatible. Unfortunately, problems do exist in transferring DISPLAY generated tapes between some machines such as UNIVAC and IBM. In these situations assembly level utilities will probably be necessary. If possible, the use of data records which are a multiple of 24 bits (3 or 4 characters) may aid in this conversion process.

Many installations concerned with transferability problems have adopted additional techniques not previously mentioned. In some firms, for instance, name qualification has been ruled out and "uniqueness" is considered to be part of the standard. This helps avoid confusion when maintenance must be performed by someone other than the original author.

Another approach is suggested by products which perform precompilation of COBOL programs with a pre-defined library of standards to be checked. If the COBOL program in question violates any of the rules set up by the standards group, appropriate error flags and severity codes are issued. Some of the more popular restrictions include separate cards for paragraph names, minimum name lengths, ACCEPT is not allowed, qualification is not allowed, REVERSED is illegal, and many more.

Fortunately, listings of COBOL source statements provide fairly good documentation and ease the burden often experienced in changes or additions to other higher level languages such as FORTRAN. This facility can be furthered by writing COBOL statements such that phrases common to most compilers are separated from those that must be varied to satisfy individual requirements. In the example:

```
SELECT file-name ASSIGN TO
device-name
```

only the card with the device name would require change. This technique, though seemingly trivial, can do alot to isolate and identify probable changes and will even save a little keypunch time.

Some other commonly used ideas to increase self documentation and facilitate maintenance of COBOL programs are:

- Final source decks should be carefully sequenced by at least an increment of 10 per card.

- Do not qualify fields. It is better to say

  "MOVE IN-PAYMENT TO OUT-PAYMENT"

  than to use

  "MOVE PAYMENT OF IN-RECORD TO PAYMENT OF OUT-RECORD".

- Define subscripts in a descriptive way. For example: TABLE-SUBSCRIPT or ARRAY-INDEX.

- Data structures should be incremented in a fixed amount larger than one to facilitate change.

```
01 FIRST-STEP
 05 SECOND-STEP
  10 NEXT-STEP
```

- Isolate verbs and nouns in some fixed field format for ease of reading.

```
ADD     DOLLARS CENTS    GIVING    TOTAL
MOVE    TOTAL            TO        OUT-T
PERFORM ROUTINE1
WRITE   OUTPUT-BUF
```

- Notes may be more easily recognized when set off by dashes.

```
NOTE-----THIS IS IMPORTANT
```

- When an error occurs, print all pertinent information and branch to the end of job if possible.

- Perform a paragraph name by specifying beginning and ending names. For example:

```
PERFORM PARAG1 THRU PARAGI-EXIT
        .
        .
        .
PARAG1.
        .
        .
        .
PARAG1-EXIT.
       EXIT.
```

- Use indentation whenever applicable. For example:

```
IF TODAYS-DATE IS LESS THAN DEADLINE
   MOVE DOLLARS TO PAYMENT
   GO TO NEXT-STEP
MOVE XSIZE TO TAX
```

In perspective, it is reasonable to expect many additions and improvements to be made to Standard COBOL. It may also be assumed that some companies will find extensions and dialect changes worth the additional transferability problems they create. Every effort must be made to insure that these improvements and deviations are incorporated in an orderly fashion and that potential problems are identified, documented and isolated whenever possible. This requires implementor awareness (IBM currently flags most extensions in their COBOL manuals) and user diligence.

2.2 FORTRAN PROGRAMMING FOR TRANSFERABILITY

2.2.1 Introduction

In November of 1954, International Business Machines introduced the "IBM Mathematical FORmula TRANslation System, FORTRAN", a language designed to permit users a facility for solving complex mathematical problems on a computer without intimate familiarity with that computer. Today, FORTRAN has achieved world wide user acceptance and has been implemented on nearly every major computer. Although originally intended for mathematical/engineering applications, FORTRAN is now used for a variety of other data processing activities.

With wide user acceptance, however, comes a multitude of extensions of what is considered standard FORTRAN. This proliferation of dialects has hampered FORTRAN's universal transferability. Since FORTRAN IV is the most widely used version and a standard has been developed for it, all further references will be to FORTRAN IV.

In the following sections, information will be presented on dialect differences and their impact on transferability, unique system environment attributes such as character sets, machine dependent practices to avoid, program organization to maximize transferability, parameterization possibilities, compiler implementation technique variations and the ensuing code restraints, and finally, examples of alternatives to our suggestions and justification of the warnings given.

2.2.2 Dialects

Unfortunately, almost as many dialects of FORTRAN IV exist as do machines/systems on which the language has been implemented, if not more (witness IBM 360 levels F, G, H). Some versions contain simple logical extensions of the basic computational capabilities such as mixed mode arithmetic involved expressions for subscripts, or conditional looping. Other implementations involve system environment extensions such as disk-oriented I/O instructions (FIND, DEFINE FILE). For obvious purposes then, we will reference all discussion of extensions, mutations and their relative costs to transferability to the ANSI-X3.9-1966 Standard FORTRAN. Although this document seems to represent the majority opinion of what constitutes standard FORTRAN, it should be noted that the ANSI version is in itself an extension of early FORTRAN IV dialects.

The ramifications of using extended or "non-standard" dialects can be numerous and far reaching. Consider, for example, a company which purchases a computer and associated software including a large FORTRAN system. After a few years of full utilization of their "improved" FORTRAN, it becomes desireable to switch to a new system with more speed, additional storage, better peripherals, etc. Upon consideration of several alternatives, it is decided that System Z (a different manufacturer) is by far the superior product and that a changeover should be initiated immediately. In choosing the new computer, it may at first seem that additional costs will be incurred only for minor job control interfaces and the new hardware. It is soon discovered, however, that the new FORTRAN does not support the same deviations and extensions as the current system and that re-programming costs will be a significant percentage of the original development outlay. The obvious result is that many have become "locked-in" to a particular manufacturer. In this case it would have been wise to avoid extensive use of non-standard features, or at least to have recognized and isolated exceptions.

To fully document all deviations ever conceived or realized seems an ambitious task. For this reason, it was decided to illustrate the most common dialect differences by reviewing several popular implementations. Specifically, IBM 7094, OS 360, CDC 6600, XDS Sigma 5/7, and Burroughs B5500 computer systems were investigated. The tables that follow include examples of notable exceptions to the standard and estimates of their cost impact on transferability as well as the expected increased effort necessary to avoid their use. These symbols used to represent the relative import of each attribute are defined in the introduction to Section 2.

STANDARD DIALECT DEVIATIONS CHART 1

Program Procedures	ANSI Standard	Notable Exception	Transferability Cost	Avoidance Cost
Continuation Cards	19	B5500 (unlimited)	T2	L1
Mixed mode arithmetic	None	IBM 7094 (not allowed)	T4	L2
Complex subscript expressions (C and K are scalars)	C*V+K C*V-K C*V V+K V-K V	OS 360 (anything)	T4	L2
Number of subscripts	3	B5500 (unlimited)	T4	L3
Statements per line	1	XDS Sigma 7 (>1)	T4	L1
Definition strings ex. X=Y=Z=4	None	CDC 6600	T4	L1
Assembly language inclusion	None	XDS Sigma 7	T5	L3-L4
Use of special logical characters < ≤ > ≥ ≠ =	None	B5500	T2	L1
Pure integer complex variables	None	B5500	T5	L4
Additional comment indicators /*,$,Δ	None	XDS Sigma 7	T2	L1
Subprograms defined within other programs with access to all parent program variables	None	XDS Sigma 7	T4	L3
IMPLICIT Statements	None	OS 360	T5	L3
Apostrophe literal definition	None	IBM 7094 (not allowed)	T4	L1
Multiple ENTRY and RETURN	Allowed	IBM 7094 (not allowed)	T3	L1
Division by 0.0	None	IBM 7094, IBM 360 (different results)	T2	L1

STANDARD DIALECT DEVIATIONS CHART 2

Control Statement	ANSI Standard	Notable Exception	Transferability Cost	Avoidance Cost
Number of nested DO loops	None	CDC 6600 (50)	U	L2
Termination of DO Loops	Not allowed on IF, GO TO, etc.	B5500 (anything)	T3	L2
Transfer into DO loops	None	XDS Sigma 7	T2	L2
REPEAT form of DO loops	None	XDS Sigma 7	T4	L2
Conditional DO loops	None	XDS Sigma 7	T4	L2
Real variables in ASSIGNed GO TO	None	B5500	T3	L2
Real variables as DO loop parameters	None	B5500	T3	L2
Literals or integers after PAUSE	None	OS 360, B5500	T2	L1
GLOBAL (COMMON by name)	None	XDS Sigma 7	T3	L2
FREQUENCY (optimize feature)	None	XDS Sigma 7	T2	L1
ABNORMAL (optimize feature)	None	XDS Sigma 7	T2	L1
LABEL, END LABELS (local statement numbers)	None	XDS Sigma 7	T3	L2
OVERFLOW, SENSE, DVCHK, etc.	None	XDS Sigma 7	T4	L4-L5
ACCEPT (typewriter I/O)	None	XDS Sigma 7	T3-T5	L3-L4
Arithmetic IF statements with less than three branches	None	CDC 6600	T3	L1
DEBUG, TRACE, DISPLAY, etc.	None	OS 360	T4	L3
Extended intrinsic functions ex. DTAN, ASIN, CDABS	None	XDS Sigma 7	T1-T4	L1-L4

STANDARD DIALECT DEVIATIONS CHART 3

Input/Output	ANSI Standard	Notable Exception	Transferability Cost	Avoidance Cost
PRINT, PUNCH	None	OS 360	T2	L1
Alternate action flags	None	OS 360	T5	L4-L5
ex. READ(N,ERR=101,END=102)				
Device oriented instructions	None			
FIND		OS 360	T5	L5
DEFINE FILE		OS 360		
READ TAPE		CDC 6600		
READ DISC		CDC 6600		
READ DRUM		CDC 6600		
ENCODE		CDC 6600		
DECODE		CDC 6600		
READ ECS		CDC 6600		
READ MS		CDC 6600		
READ N		B5500		
READ (N=4)		B5500		
Extended I/O Supervision	None		T4	L2
BUFFER		CDC 6600		
CLOSE		B5500		
LOCK		B5500		
PURGE		B5500		
Expressions for unit members	None	B5500	T2	L1
ex.READ (2*Y+1,10)				
Explicit record identifica-	None		T3	L2
tion ex.READ(8'16,10)		OS 360		
READ (8=4)		B5500		
INPUT,OUTPUT (NAMELIST)	None	XDS Sigma 7	T3	L2
Decimal data input in I format	None	XDS Sigma 7	T3	L2

2.2.3 System Environment

It is difficult to control or predict the machines/systems on which a program may someday be compiled. There are, however, several actions that can be taken to avoid system environment impact on transferability. An awareness of the many items which are likely to be unique in a given environment can be of great importance in planning programs to minimize and isolate system dependencies. The following two lists show attributes to consider in system changes and practices to avoid in general.

Unique Attributes to Consider

1. Word size and precision.

2. Character size and the effect on core space and character oriented coding. Example:

   DATA MX/6HCHARAC,6HTER---/ versus

   DATA MX/4HCHAR,4HACTE,4HR---/

3. Logical unit assignments.

4. Peripherals available (number and kind).

5. BCD, EBCDIC card code differences.

6. Time controls.

7. Block and segment command words affixed to binary tape records.

8. Maximum and minimum record size constraints.

9. Printer and typewriter line size and character sets.

10. Incremental compiler restrictions (order of statements).

11. Internal representation (hex, octal, binary).

12. Core size.

Practices to Avoid

1. Multiple file I/O. (OS 360, not available on other machines.)

2. Special disc or drum oriented instructions.

3. Literal unit assignments.

4. Special system routines such as sense switch options.

5. Typewriter I/O.

6. Use of hex and octal literals (internal representations).

7. Special characters (other than A-Z, 0-9, +, -, *, /, (,), ,, $ and, ANSI standards).

8. Format controlled records of more than 120 characters.

9. Assembly language interfaces.

10. STOP and PAUSE (differing system response).

11. Assuming pre-initialized memory.

12. Character tests with machine dependent coding (input the character set).

13. Excessive use of labelled COMMON (insufficient blank COMMON may result in the inability to load in OS 360 due to loader sharing of that space).

14. Word, byte, character, or sign implementation dependent coding.

2.2.4 Modularity

The ability to divide machine/system dependent and independent attributes into separate entities via sub-programs greatly enhances the transferability of programs. Furthermore, the ability to compile these entities individually aids in the speedy checkout of transferred modules. FORTRAN is exceptional in the ease to which it facilitates modularity. The FORTRAN programmer may write his program as a collection of small routines, compile and test them individually, and put them together in a building block fashion into large assemblages without recompilation. Only subroutines that are modified or corrected ever need be recompiled. In this way, machine dependent code may be economically isolated into separate modules.

Assuming that the isolation of system dependencies is feasible, the dependent segment should be separated into relatively homogenous sub-segments. This is necessary because some aspects will not be as dependent as others (formatted versus unformatted input/output, for example).

Similarly, it may not be possible to completely purge an independent section of transfer problems. Thorough checks should be made for obscure problems connected with machine characteristics such as precision and round-off. Parameterization may be of help in this area.

Programs written in FORTRAN usually contain the following grouping:

Dependent	Independent
Special input/output	Program logic
Error checking and recovery	Calculations (except for precision)
Operating system interface	Data movement
Character set	
Data masks	

With this functional grouping in mind, a typical program can be designed (divided) into the following modules.

- Main control (independent)
  - Calls computational routines
  - Calls dependent routines
  - Calls main control loop
  - Calls independent exit routine
  - Capable of compilation on very basic system
- Dependent initialization
  - Sense lights, overflow indicators, etc.
  - Character set (read in)
  - Format statements (read in)
  - Parameters (word size, character size, etc.)
  - Calls ERROR routine

- Main control loop (independent)
  - Calls independent initialization
  - Defines blank COMMON
  - System independent tests and calculations to drive the job
  - Calls input routines
  - Calls ERROR routine
  - Calls output routines
  - Loop control
  - Calls general program
  - Returns to main control

- Operating system interface (dependent)
  - Mechanism to exit system and dump
  - ERROR routine
  - Unformatted I/O routine (entered via computed GOTO)

- Independent initialization
  - Usual program initialization process
  - Calls to operating system interface

- ERROR routine (dependent)
  - Handles sense lights, overflow check, etc
  - Handle recoverable errors (parity, end files, etc.)
  - Calls dump routines

- Formatted input/output (independent)
  - All formatted input/output
  - Controlled by computed GOTO

- General program (independent)
  - All calculations, data handling and program logic not present in separate modules.
  - Calls dependent routines

2.2.5 Parameterization

Specification of machine/system dependent aspects by way of coded or card input parameters would indeed aid program transferability. Unfortunately, many of the features such as word size, byte size, core size, record length and type, are deeply imbedded in individual compiler designs or have simply been overlooked. For instance, the number of characters per word is inherent in hollerith definitions such as DATA/X/6HABC DEF/. Similarly, there is no easy way to reduce the dimensions of arrays in COMMON. Of course, it has always been good programming technique to parameterize such things as DO loop indices or logical unit assignments, but much more is needed in this area. Although there have been a few contributions such as variable dimensions in subroutines, transferability of application programs must be taken more into account in future FORTRAN compiler implementations and revisions of the Standard.

2.2.6 Code Restraints

Many commonly utilized techniques or "tricks" in FORTRAN programs are severly detrimental to program portability because of machine/implementation differences inherent to individual compilers. The following list is a sample taken from knowledgeable users of such tricks.

Coding Techniques With
Unpredictable Variations

1. Testing for 0. Example: IF(X.EQ.0.0) GO TO 10

2. One terminal statement for a nested DO loop exit.

3. Termination of a DO with a complex statement such as an IF.

4. Literal strings of excessive length (>255 characters).

5. Altering a DO parameter within the loop.

6. Ambiguous statements such as

   N=N+FUNC(N)*N where FUNC(N) alters N.

7. Double exponentiation: A**B**C.

8. Assumption that some finite value will be substituted for a division by 0.

9. Recursive subroutines.

10. Variable length argument strings in subroutine CALLS.

11. Scattered specification statements.

12. Shifting by multiplication.

13. Using large numbers as DO indices. Example:

    DO 10 I=1, N where N = 2^{17}

14. Assuming a loop is always executed once. Example:

    DO 10 I=4,N where N = 3

15. Transferring into a DO range.

16. Assuming the value of an index outside a DO loop.

17. Returning from a subprogram via an assigned GO TO.

18. Assuming an incorrect computed GO TO variable will result in a default condition such as "falling through".

2.2.7 Policy

Regardless of individual arguments against the Standard FORTRAN, it seems that only strict adherence to this definition will minimize transferability problems. Although intended for convenience and versatility, extensions of the Standard base will eventually lead to severe restriction and user dependence on particular suppliers' products. In addition, programmers working in an environment which allows broad variations of FORTRAN will experience re-adjustment difficulties when attempting to code on other systems; hence, *programmer* transferability cost is also a factor.

If an organization is committed to non-standard dialects, every effort should be made to utilize the Standard recommendations as a documentation guideline. All deviations should be listed along with estimates of their diversity and importance to the individual program (see Section 1.4).

If possible, compilers that are non-standard should include diagnostics pointing out these extensions/modifications as potential problems. An example of this approach is the WATFOR FORTRAN compiler developed by the University of Waterloo. WATFOR flags its own extensions and also attempts to point out (via diagnostics) system dependent code such as uninitialized variables.

Many general policies which will aid in program design are summarized in the following list.

General Guidelines

1. Avoid usage of extensions to the Standard whenever possible.

2. Document extensively all dialect variations and machine/system dependent code, functions, general design, etc.

3. Modularize programs into machine/system dependent and independent sections.

4. Avoid assembly language code interfaces.

5. Do not use programming tricks dependent on machine idiosyncracies.

6. Design magnetic tape outputs for general compatibility. Avoid complicated blocking or binary (non-formatted) final outputs.

7. Lay out blank COMMON in one central routine and treat variables there as if they were global. This avoid duplication and is a form of documentation.

8. Always initialize memory even if your present system does it for you.

9. Always assume the character set is variable. Do not make programs dependent on the internal character representation of a particular machine.

10. Avoid operator inter-action (typewriter I/O).

11. Plan to end up with a test package which can be used in future transfers.

12. Estimate the range of data values and document same. The precision of integer and floating arithmetic is machine and software dependent. If future systems have fewer bits assigned to the characteristic in floating point representations, for example, the current data may generate over/underflows (which may go undetected).

After the program is complete and a system change occurs, the following checklist may help detect potential problems.

Checklist for Transferring a Program
From One Machine to Another

1. Does FORTRAN exist on the new machine/system?

2. Is the new machine's core size adequate?

3. Are all intrinsics available?

4. Are all library routines available?

5. Is the FORTRAN standard?

6. Are word and byte size the same?

7. Is the precision adequate?

8. Is internal representation the same? (Octal, hex, binary, etc.)

9. Is the character set the same? Must card decks be converted?

10. Are necessary peripheral devices available and adequate?

11. Does operating system interface necessitate changes in the code? (i.e. are the systems CALLs the same?)

12. Can assembly language routines be re-written easily?

13. Can the machine characteristics be defined in an initializing subroutine?

14. How will machine speed affect the programs? Must some be re-written for more efficiency?

15. Are all necessary FORTRAN statements available?

16. Does the machine/system dependence/independence modularity still hold?

17. What about data? Can some be used without reformatting? Do tapes have to be converted?

18. Will possible record size changes cause any storage problems?

19. Can rapid change and checkout be made? If not, why not?

20. Recompile independent modules first. Any errors which are encountered may signify some basic problems.

Many of the suggestions made in previous sections involve trade-offs which perhaps should be individual management decisions. In modularity, for example, design of relatively independent subprograms greatly enhances transferability, but it also will increase execution time in most cases. If the project leader feels that modularization will severely hamper the development/ efficiency of the program and also that the application is short term and will surely not need to be transferred, then obvicusly the suggestions made in 2.2.4 should be discarded.

With regard to extensions of the Standard, a general policy of avoidance is a good one. However, the programming manager may feel that the use of particular additional capabilities present in his dialect is of such importance that it outweighs the obvious transfer problems it will create. Again, as suggested in 1.4, at least document and attempt to isolate these extensions.

Also, it is apparent that some programs simply cannot be transferred. An in-core sort based on at least 128K bytes of storage for tables may be impractical on a machine with only 32K bytes. Similarly, if a scientific analysis program currently takes one hour to run and a transfer is proposed to a machine with 1/10 the speed, it may no longer be economically feasible to utilize this routine. This brings out another basic policy consideration in program design. The manager should influence design decisions based on speed, accuracy and core space keeping in mind how the particular computer now being utilized relates to other possible future machines. For instance, if the current application computer has 36 bits and at least 34 bits are required for accuracy of computations, perhaps double precision should be used anyway since a transfer to a computer with 32 bits may involve extensive reprogramming. Again, this may be a management trade-off (speed versus reprogramming) decision. The same reasoning applies to speed versus table size considerations.

So that there is no misunderstanding about code restraint implications listed previously, the following justifications are included:

1. Due to inherent inaccuracies of floating point representations, it may happen that a test for a specific number will fail when actually it should pass. For example:

   A = 1. - 2.0/2.0 may come out to be .0000001 and a subsequent test for (A.EQ.0.) would fail.

2. Some compilers put special restrictions on nested DO loops that terminate on one statement. XDS Sigma 5/7, for instance, only allows the inner most DO to transfer directly to its termination point.

3. Compilers which allow termination of DO loops on IF statements disclaim the predictability of results (XDS Sigma).

4. Compilers may have inherent table size restrictions on character string length (CDC).

5. Altering a DO parameter within a loop may produce varying results depending on the mechanization of the DO (pre test, post test, index in core, index in register, etc.)

6. Ambiguous statements such as

   ```
   N=N+FUNC(N)*N
   ```

   depend on the mechanization of the scanning algorithm. If the original value of N is not stored in another temporary location FUNC(N) may destroy it.

7. Double exponentiation A**B**C is also not defined in the Standard and is dependent on the mechanized algorithm

   $(A^B)^C$ or $(A)^{B^C}$

8. Division by 0 may result in values 0, 10^{39}, etc. depending on the system.

9. Most compilers do not allow recursive subroutines.

10. Some compilers will not deal successfully with missing arguments (variable length) in a subroutine CALL (mechanized at the RETURN and assumes the number of arguments is fixed.

11. Incremental compilers cannot handle scattered type, dimension, and DATA statements. Yet, this form of compiler is desirable for a speed/user interface stand-point.

12. Shifting algorithms are word-size and hardware (multiplication, sign representation) dependent.

13. Compilers may implement DO loops in index registers (IBM 7090) and hence set an upper limit on index values.

14. Loops of the form DO 10 I=K, J where J<K may or may not be executed once depending on where the test for completion is made.

15. Most compilers do not guarantee the results of transfers into a DO loop.

16. The addressing scheme of some machines/compilers allows for passing of statement label addresses for an assigned GO TO to subroutines.

17. Some compilers generate code for testing the computed GO TO arguments. The last statement label or next statement becomes the error condition default.

In summary, the use of FORTRAN can unquestionably aid in program mobility especially when applications are modularized into machine/system dependent and independent sections. Dialect variations should be avoided. Although most compilers contain diagnostics which will flag illegal code, serious design problems may result when machine/system transfer is attempted. Undetected errors will often occur when special tricks or machine dependent algorithms are used. Parameterization of crucial restraints such as core size, and word length are needed though FORTRAN is somewhat deficient in this area. It is the responsibility of the programmer to follow recommended guidelines and carefully document exceptions.

2.3 JOVIAL PROGRAMMING FOR TRANSFERABILITY

2.3.1 Introduction

Although JOVIAL has been developed primarily by System Development Corporation (SDC), its history of modifications and varying implementations has been as complex as FORTRAN's. The basic objective of "Jule's Own Version of the International Algebraic Language" was to create a language for use by professional programmers in solving large complex information processing problems. In route to the fairly successful fulfillment of this objective, JOVIAL has been involved with all the problems associated with efforts to avoid dialects, control extensions and subsets, maintain compiler independence and attain wide usage.

Originally used by SDC for Air Force projects, it has also been adopted by the Navy and Army in many of their programming efforts and has enjoyed wide acceptance within SDC. In June 1967, after several "official" JOVIAL specifications had been established by SDC, the Air Force issued its own specifications of JOVIAL (J3) as the standard programming language for Air Force command and control applications. This document, Air Force Manual 100-24, is considered "the standard" in the preparation of the following material.

2.3.2 Dialects

Standardization of languages has been said to be important for two reasons: It allows the transfer of programs from one computer system to another. It allows the transfer of programmers from one application to the other. The consensus of opinion on JOVIAL is that the first reason, program transfer, is possibly less important than the second, maintaining a staff of mobile programmers. The assumption is that JOVIAL applications, largely in support of command and control systems, tend to be much more application specific than routines written in other higher level languages such as FORTRA . Hence transfer may be impractical or meaningless in many instances. Transferring programmers between applications, on the other hand, is deemed extremely important with great emphasis having been placed on programmer convenience and usability. This concept seems largely responsible for the adoption of a standard JOVIAL (J3) by the Air Force.

From a transferability standpoint, we must assume that the Air Force standard is generally acceptable and that it will form the basis for subsequent recommendations that may be made. With this in mind, it is the purpose of the following tables, compiled from descriptions of widely used J3 dialects, to point out common extensions and subsets of the Air Force standard. Furthermore, an attempt is made to assign relative cost factors to these extensions in terms of programmer convenience of initial coding and transfer re-work (see tables at the beginning of Section 2). The compiler implementations used for this study were: Burroughs D-825, Hughes 3118M, UNIVAC 1108, Control Data 6000, IBM AN/FSQ-7.

EXTENSIONS OF THE STANDARD (J3) DIALECT

Program Procedure and Syntax	Notable Exception	Transferability Cost	Avoidance Cost
Use of single letter variable names for simple items outside a "FOR" loop.	D-825	T4	L1
Multi-character variable names as "FOR" indices.	D-825	T4	L1
Optional number specification preceding hollerith literals 3H(ABC).	D-825	T4	L1
Boolean operations on fields of bits (allow values greater than 1 as a result of Boolean operations).	D-825	T2	L2
Use of special symbols: $ ( ) ; : @ #	H3118	T4	L2
Use of dual variables in indexing by conversion of left half to single.	Q7	T3	L2
Status variables operating on the right of relational expressions or in a string of relationals such as V(POOR) LQ QUALITY LQ V(GOOD).	D-825	T4	L2-L3
Re-declaration of items with subsequent code relating to the new declaration and previous code relating to the previous declaration.	D-825	T5	L3
ITEM ORIGIN specifying the location of a particular item.	Q7	T3	L1
Use of CHAR and MANT functions on non-floating items.	D-825	T3	L1

EXTENSIONS OF THE STANDARD (J3) DIALECT
(Continued)

Program Procedure and Syntax	Notable Exception	Transferability Cost	Avoidance Cost
ODD function applied to Boolean formulae.	D-825	T3	L2
TABLE ADDRESS structure which allows specification of exact core locations for tables.	Q7	T3-T5	L1-L5
Equate ENTRY's of tables of different structure (parallel and serial).	CDC 6000	T3	L2-L3
Automatic (compiler) initialization of non-set items of a partially pre-set entry.	Q7	T2	L2
Combination of one-factor "FOR" clauses and only special compounds (BEGIN-END).	Q7	T3	L2
Multiple entry points in a PROCEDURE.	UNIVAC 1108	T3	L3
Input/Output Statements			
No type agreement necessary between actual and dummy arguments in PROCEDURE calls.	Q7	T4	L3
Initialization constants in FILE statements which are written on the file whenever an OPEN is executed for that file.	UNIVAC 1108	T3	L2
Debugging outputs: MONITOR 'DEBUG,'ENDBUG	CDC 6000 UNIVAC 1108	T3	L2

EXTENSIONS OF THE STANDARD (J3) DIALECT
(Continued)

	Notable Exception	Transferability Cost	Avoidance Cost
<u>Input/Output Statements</u>			
Implicit file status procedures to b executed if a file status in the range 16-23 occurs at any time (placed on OPEN,INPUT,OUTPUT statements).	UNIVAC 1108	T4	L4
Input/Output of STATUS items.	CDC 6000	T3	L2
Use of all primitive functions such as POS, BYTE, etc. in I/O.	CDC 6000	T3	L2
Use of FORTRAN formatted I/O (PRINT,LIST, PRINTF,ENDL directives).	CDC 6000	T4-T5	L4
WAIT: halt in execution until typewriter carriage returns.	H3118	T1-T5	L2
<u>Procedures and Functions</u>			
Pattern function nP(L) which sets up a bit pattern of n bits per digit in number L.	UNIVAC 1108	T3-T4	L2-L3
External PROCEDURES which can be compiled separately and brought in at load time.	UNIVAC 1108	U	L1
System-oriented user routines (BTOD,DECODE, ENCODE,etc.)	CDC 6000	T4	L3
Type conversion functions (FLOAT,FIX,ENTER).	D-825	T3-T4	L3
Mathematical subroutines (built-in SIN,COS SQRT).	CDC 6000	T1	L1
Sensing functions (DIVCK,OVERCK,SENSE)	H3118	T3-T5	L3-L5

It was almost always the case in FORTRAN that a particular implementation had numerous extensions but virtually no deletions from the standard. JOVIAL compilers, on the other hand, seem to divide changes equally between additions and subsets.

Extensions may be dealt with in advance of system transfers by simply avoiding them. Trying to "guess" what will be left out or changed in the next system, however, is no easy task. Indeed, in the event that current JOVIAL programs are to be transferred to "smaller" systems, it appears that recoding cannot be avoided. Actually, no meaningful or predictable common subset seems to exist among popular implementations. Rather, each compiler tends to drop or alter different features. The following list of unavailable or changed aspects of JOVIAL was produced from the small but highly utilized computers IBM Q7, Burroughs D-825 and Hughes H3118.

It is not intended to suggest that these features be avoided since they are only a representative sampling. Avoidance of all facets of JOVIAL which may not be present in "some" compiler implementation may very well result in a null capability. It is hoped, however, that this list may at least indicate differences to look for in the event of a pending system change.

Subsets of the Air Force Standard and Miscellaneous Differences

Attribute	System
1. No variable (ITEM) range specification.	Q7
2. Floating point not available.	Q7
3. Transfer around CLOSE declarations must be provided by the programmer.	Q7
4. Storage of arrays by rows rather than by columns.	D-825
5. Absolute addresses not permitted.	D-825
6. The (* *) form on exponentiation is not allowed.	Q7
7. Functions CHAR,MANT,STRING,'LOC are not present.	Q7
8. Exponentiation takes precedence over negation: -B**2=$-B^2$.	D-825

9. Variables assigned to items with fewer bits are truncated from the right. D-825

10. Variables assigned to items with more bits are left-justified with trailing 1 bits. D-825

11. The default mode is A [illegible] S. Q7

12. CLOSE names are not allowed as inputs to a PROCEDURE. Q7

13. DUAL variables are not permitted. D-825

14. Rounding (R) is not available. D-825

15. Literal items must be less then 3 or a multiple of 3 characters. H3118

16. Medium packing (M) is not available. H3118

17. Tables and arrays may not be the output parameters of a PROCEDURE. H3118

2.3.3 System Environment

Many of the factors which were recognized as system environment sensitive in Section 2.2.3 (FORTRAN) apply to JOVIAL programs as well. The impact of different word and byte sizes, character representations, peripheral devices and other attributes unique to each system is universal.

Actually, JOVIAL programs are generally more dependent on the current operating environment than are programs coded in many other higher level languages such as FORTRAN. This is not to say that careful choice of the JOVIAL capabilities utilized and generous parameterization will not be effective in maximizing transferability. Rather, it must be emphasized that failure to recognize potential system dependent situations can seriously complicate subsequent machine transfers. The range of possible outcomes is due in large to the number of environment sensitive specifications which are left to the JOVIAL programmer. Some of these include:

- the number of bits per variable (ITEM)
- the number of words per table entry

- literals of "unlimited" length packed in tables
- fixed point precision - the number of bits in the fraction
- rounding or truncation
- programmer packed tables
- formatting of I/O
- device oriented I/O

Many of these features can be specified parametrically. When no convenient method of parameterization exists, however, isolation of necessary machine dependent functions into easily recognizable and hopefully changeable modules is recommended.

Perhaps the most likely candidate for isolation in JOVIAL programs is input/output. One of the reasons for the system dependence of JOVIAL I/O commands is that file manipulation is at the device (rather than logical) level. Only physical records in the file are addressed and after initialization, no JOVIAL monitoring takes place. Hence, error checking by means of status items tests (which vary from system to system) and even data conversions (such as formatted I/O in FORTRAN) are left entirely to the programmer. Needless to say, a great deal of machine dependent coding and duplicated effort is generated in this process.

A possible approach to minimizing this problem is the creation of a system library. A standard set of routines could be made up from an outline of the various types of input/output generally performed and included with each application. These PROCEDURES would include FILE,OPEN,SHUT,INPUT,OUTPUT declarations and status constant tests for end of file, parity errors, etc. The programmer would then call the appropriate routine and pass information on status (in terms of program defined constants), number of words, array addresses, etc. This approach would most likely follow the lines of inherent I/O used by FORTRAN with intermediate buffering, fixed record lengths and similar standardizations. By giving up some freedom, the programmer would produce routines which could be transferred to other machines by re-writing of I/O procedures rather than extensive re-coding of every application.

Some of the more subtle forms of system environment effect on transferability involve the basic JOVIAL commands and functions.

The following list is a sample taken from the Air Force Standard. It is not intended to be an exhaustive accumulation of do's and don't's, but is intended to point out the types of statements in JOVIAL which should be used with caution to insure a minimum of system dependence.

Machine (Environment) Dependent
Aspects of the JOVIAL Language

1. The use of the BYTE modifier in conjunction with non-literal items assumes the character size and the number of characters per word.

2. The BIT modifier assumes word size.

3. The 'LOC modifier in many cases assumes fixed addresses for variables.

4. CHAR and MANT results depend directly on machine characteristics and floating point implementations.

5. When STOP is used, the computer may take alternate action (including bombing out) depending on the computer system.

6. Literals exceeding the host computer word size or packed partially in multiple core locations will encounter machine dependent restrictions such as byte boundary alignment and compiler restrictions such as starting byte (in the word) specification. Thus, literal packing by the programmer should be avoided, and code referencing such items should avoid word length dependence.

7. Octal constants modifying or setting literals such as BLANKS = 0(6060) are word size and character set dependent.

8. The ODD function may produce varying results with regard to the negative number representation of the machine. AFM 100-24 specifies "ODD designates the value true when the least significant bit is one and false when it is zero. ODD is true, therefore, when the modified variable, considered an integer, is odd." Note what happens in one's complement notation: -3=1....1100. Therefore, -3 is even! If ODD is to be used on negative numbers, perhaps ODD(ABS(X)) should be substituted.

Some of the items to be considered in transferring I/O routines to new systems include:

- FILE declarations contain unique status constants to indicate errors, tape marks and device ready. The number, order, and meaning of these constants is guaranteed to vary.
- Device names are assigned by the compiler implementor.
- Card column usage is system dependent.
- Physical device characteristics vary greatly (number of tracks, density, parity).
- Record size restrictions may exist. Some systems override FILE declarations and make their own assignments.
- Records may contain system reference marks such as end-of-line, carriage control, beginning of record, etc.
- Timing controls (one manual specifies that JOVIAL programmers must be sure that 40 milliseconds elapses between rewinds!)
- I/O can often be done only in word increments making physical record length a variable.

Core size, record size, instruction set, and myriad other considerations effect the method of compiler implementation. Because of this, compilers will be forced to apply restrictions on programs they process. Clearly, no well-defined prediction of these restrictions can be made without specific knowledge of the particular machine to be utilized. A brief summary of some of the more commonly restricted items and representative size limits may nonetheless provide some insight into when JOVIAL code may become "uncompilable". This in turn may help the JOVIAL programmer to avoid "way out" designs.

Compiler Implementation Restraints

Item		Restraint	Example
1.	Status constants	6 characters	UNIVAC 1108
2.	Array dimensions	5	Q7

Item	Restraint	Example
3. Number of constants/ names	2000	UNIVAC 1108
4. Nested parentheses	25	UNIVAC 1108
5. Nested PROCEDURE calls	25	UNIVAC 1108
6. Number of PROCEDURE calls	500	UNIVAC 1108
7. Number of PROCEDUREs	250	UNIVAC 1108
8. Number of external references	1025	UNIVAC 1108
9. Number of JOVIAL signs per statement	200	UNIVAC 1108
10. Length of hollerith constants (characters)	250	CDC 6000
11. Length of a name (characters)	30	CDC 6000
12. Number of words per table entry	32	CDC 6000
13. Nested BEGIN-END brackets	25	CDC 6000
14. Nested IF statements	50	CDC 6000
15. Nested IFEITH-ORIF statements	20	CDC 6000
16. Nested FUNCTION calls	20	CDC 6000
17. Time between I/O rewinds	40ms	Q7
18. Number of DEFINE statements	75	Q7
19. Number of OVERLAY declarations	128	Q7

Item		Restraint	Example
20.	Number of variables per OVERLAY	32	Q7
21.	Size of an array (product of first 2 dimensions)	2048	Q7
22.	Size of index in a FOR loop	15 bits	Q7
23.	Items per densely packed TABLE	50	Q7
24.	COMPOOL Restrictions		Q7
	ITEM name	4 characters	
	STATUS name	3 characters	
	TABLE name	3 characters	
	ENTRIES per TABLE	9,999	
25.	Number of diagnostic messages generated	200	CDC 6000
26.	Items in compiler packed TABLE	300	CDC 6000
27.	Literals' length	8 characters	D-825
28.	Only the first six characters of names are used by the compiler		H3118

2.3.4 Modularity

The JOVIAL programmer can write his program as a collection of small routines and compile, debug and modify them individually. Assembling these routines into a larger program, however, involves re-compilation of the entire set. Further modification or correction also implies total recompilation in most systems. This difficulty could seemingly be avoided by insertion of checked-out subprograms into the system library. Since most JOVIAL compilers produce non-relocatable code, however, this can not be recommended as normal practice.

This undesireable restriction evidently has led a few programmers to express their doubts regarding JOVIAL modularity as an aid to program transferability. Some feel that the inability to update individual portions may inhibit efforts to modularize machine dependent code. Isolation intc subprograms (which may introduce "extra" interface bugs) for the purpose of facilitating re-coding may increase check-out costs since "good" code must be re-compiled many times as well.

On the whole, this lack of independence in JOVIAL seems to be more of a deterrent to efficiency than to portability. The basic elements of modular design are nonetheless available to the programmer. Furthermore, what JOVIAL lacks in relation to FORTRAN independence of compilation, it may make up for in its ability to utilize the macro-like CLOSE structure within individual routines. Isolation of system dependent code to PROCEDUREs (subroutines) and further division into CLOSE structures within these PROCEDUREs is an important aid to transferable design.

Specification of general input/output routines with individual I/O functions defined as CLOSE areas, for example, might enable this highly machine oriented activity to become a series of CLOSE calls. Hence in a new system, much of the structure of the I/O function could be retained while the necessary CLOSE's are re-coded.

Another feature worth mentioning is COMPOOL. In large command and control applications, data referenced in several programs need be declared only once. Thus, where transfer becomes necessary, machine dependent declarations must be rewritten only once. If proper parameterization is used, a relatively few statements need ever be re-coded. It is felt, however, that the J3 Standard should specify more precisely the kind and form of declarations which may appear in COMPOOL.

2.3.5 Parameterization

The most predictable aspect of large-scale system development is change. Modifications to the requirements, revisions to the specifications, and changes in design or programming techniques all result in changes to the program data. Particularly susceptible to these events are the dimensions of data structures, the explicit coding of numeric data, and the scaling, packing and precision of fixed point items. If specified in exact numeric terms, as in FORTRAN, any changes in these characteristics will require individual analysis and modification of each of several references.

JOVIAL presents the programmer with an opportunity to isolate many system design and machine dependent features into a relatively small number of statements. Thus, the careful programmer can produce a large percentage of transferable code.

This facility can be realized through judicious use of compile time parameterization. On the other hand, overlooking possibilities of using parametric representations for crucial system oriented factors can result in very dependent programs. Likewise, poor technique or over proliferation of explicit parameters can also be a hindrance. For instance, a parametric variable for the number of records on an output tape may reduce re-coding during system transfer, while explicit delineation of the precision required for specific variables (precision is a JOVIAL parameter) may cause extensive recoding of ITEM declarations.

Fortunately, JOVIAL is flexible in this area and even parameters may be parameterized. For example, DEFINE a variable "PRECISE" to be "5". Then use "PRECISE" as the number of bits to the right of the binary point in a particular set of items. The statements:

```
DEFINE  VLENGTH "16"
DEFINE  PRECISE "5"
ITEM    X  VLENGTH S PRECISE
```

would generate ITEM X 16 S 5. Of course, even the sign designation "S" should be parameterized if any change is possible. If a machine change is made and 16 bits are no longer available, then only VLENGTH may need to be changed. All ITEM declarations using VLENGTH would be left alone. These simplistic techniques, though quite effective, seem to be overlooked or ignored by most JOVIAL programmers.

Perhaps the most crucial area with regard to data structure declarations is table packing. JOVIAL allows the programmer to specify the number of bits allocated to each ITEM and further to specify which bits of each word these ITEMs are to occupy. This is indeed a powerful tool which could produce highly optimized table storage allocations. To use this ability explicitly, however, is to hamper attempts at transferability. A new machine is very likely to have a different word size. Hence, all table packing would be redone by hand. A better technique may be to parameterize the number of bits per ITEM and allow the compiler to pack the data. Most compilers will pack from left to right as ITEMs are encountered.

The programmer may still participate in storage allocation by listing ITEMs in the order in which they will best pack into words. Furthermore, some compilers will automatically scan all ITEM declarations and choose those which "fit" together best (use most number of bits in each word).

The following is a summary of items which should be parameterized in most JOVIAL programs:

- number of bits per variable
- number of fractional bits in a variable
- sign specification
- packing factor
- rounding or truncation
- maximum number of entries in a table
- number of words per entry
- range of values for a variable
- array size
- I/O device names
- number of records per file
- device status constants
- default mode(s)

The last entry in the list deserves additional mention. The Air Force Standard allows for the presence of more than one MODE directive in JOVIAL programs. Hence, by careful ordering of initialization statements, much of the explicit item declaration can be avoided. For instance, if a program is to have two basic types of variables, say 16 bit integers and 16 bit fixed point with 5 fractional bits, two MODE declarations could possibly take the place of many complex ITEM declarations.

Consider the example:

```
MODE I 16
   AB=3+XA
      .
      .
      .
      .
   AZ=5

MODE  A  16 S 5 P 5.1
   ITEM XX
   ITEM YY
   ITEM ZZ
      .
      .
      .
      .
```

All variables initialized between AB and AZ would be integers and all variables after AZ (XX,YY,...) would be fixed point with initial value 5.1. Hence, only the MODE directive may require changing during system transfer (it actually should be parameterized also).

2.3.6 Code Constraints

Care must be exercised to avoid the use of tricks dependent on particular machine or system idiosyncracies. So many variations in structure, type conversion and algorithm implementation are possible that use of specific compiler characteristics can result in virtually untransferable code. Worse yet, programs may appear machine independent while producing several results when executed on different machines. If often happens that compilers do not reject certain illegal or undefined structures but compile them instead, giving results the programmer "knows" are appropriate.

It is universally recommended that programmers avoid exploitation of this kind of special knowledge. There is no guarantee that a new version of the compiler will exhibit the same eccentricities so cleverly discovered by a programmer who may no longer be present.

Some instances where this type of coding may be encountered are listed below.

- Floating point comparisons for equality may fail due to hardware algorithm inaccuracies (absolute value of difference should be substituted).

- Binary point alignment required for comparison of fixed and integer operands may cause loss of high order bits.

- Avoid hollerith comparisons other than EQ,NQ since no collating sequence is defined generally.

- Unpredictable results may result from FILE declarations within PROCEDURES.

- Mixing programmer allocated and compiler allocated (N,D,M) packing is unpredictable or unallowed.

- Do not assume that the value of a function is passed in the accumulator, register 0, etc.

- Do not try to inhibit rewind by overlaying a DUAL item on a file name, OPEN file and setting DUAL = 1.

- Do not expect 2^N to be implemented by a shift (it may be done by logorithms).

- Avoid use of mixed type expression evaluations such as numeric=hollerith which expect hollerith to be treated as a straight numeric value (no conversion) or hollerith-numeric with truncation of the numeric.

- The assumption that output parameters not referenced in a particular PROCEDURE execution (skipped) maintain previous values (last time PROCEDURE was executed) is sometimes not warranted.

- The assumption that A**B**C is evaluated left to right

  $$(A^B)^C$$

  may not always hold true.

- Shortening status constant lists by specifying too few bits and assuming remaining constants are ignored is bad practice.

- Automatic code conversion between hollerith and transmission code for comparisons may not occur.

- Entry operations on TABLE entries of varying length may produce varying results.

- The assumption that indexing is performed by taking the absolute value of an expression is not warranted. For example X(9-17)=X(8) in some implementations and is an error in others.

- Although convention specifies a left to right scan, some compilers could conceivably produce different results for expressions containing functions which modify variables in the expression. For example:

  A=A+SFUNC(A)*A is usually $A_2=A_0+SFUNC(A_0)*A_1$

  where A_0,A_1 and A_2 represent 3 states of the variable "A" which is altered by function SFUNC. Other results could exist especially if optimization of code is performed by the compiler. (See Appendix A.)

- Do not assume that core is pre-set by the system (to 0 or other values).

- Do not assume that the NENT of a variable length table is pre-set.

2.3.7 Policy

Many of the recommendations that were made in Section 2.2 are directly applicable to JOVIAL programs. A review of 2.2.7 should prove valuable to JOVIAL programmers as well.

Adherence to an acceptable standard remains the most necessary step toward assuring program and programmer transferability. Similarly, careful segmentation of programs into system oriented and machine independent modules is another basic goal. A third fundamental principle, parameterization, is especially important. JOVIAL has distinct advantages over other higher level languages in its ability to parametrically define system dependent attributes. This may balance the seemingly restrictive necessity for device oriented input/output which is absent in languages such as FORTRAN.

These three concepts, standardization, modularity, and parameterization cannot be over emphasized. There is another aspect of JOVIAL, however, which also is an aid to transferability. The lack of format restrictions and the ability to intermix comments among the symbols in JOVIAL statements provides the ingenuous programmer an opportunity to create programs which are largely self-documenting. This facilitates easy maintenance and revision. A policy of one JOVIAL statement per line and meaningful comments on the same card where possible will help maximize the readability of any JOVIAL routine.

One area where a qualification of general policy regarding extensions might be beneficial is external PROCEDUREs. Most compilers allow only internal PROCEDUREs which must be compiled with each program and become part of that program. External PROCEDUREs, which are compiled separately, are very useful when a routine is to be used by several programs that operate in core memory together. With this facility, only one copy of the routine is in core memory and each program refers to it either by an external PROCEDURE declaration or by COMPOOL definitions. Although this is not a standard feature, it is quite useful and does not seem to inhibit future transferability to a great degree.

There are also features in JOVIAL which should be avoided even though they are included in the Standard. The "DIRECT" statement is an example. "DIRECT" is a means of breaking out of JOVIAL within a program and writing instructions in the basic assembly language of the target computer and hence is generally non-transferable. Obviously, then, DIRECT code should be eliminated or, if absolutely necessary, isolated, simplified and exhaustively documented.

Similarly, the specification of absolute addresses is inflexible and prohibits relocation of data or program blocks. Relocation is highly desirable in that it allows for the assembly of programs in a variety of ways insensitive to the operating system environment. Generally, it would be invalid to construct programs that assume some order or sequence in storage. The possibility exists that the operating system may segment arbitrarily and load programs into whatever space is available in memory. Furthermore, in a time-sharing environment each "turn" may result in a newly allocated memory environment.

Undoubtedly, many of these aspects require management decisions: is absolute location necessary; how much direct code is required; should time-sharing and multi-programming be ruled out; etc.

Finally, there are some facets of JOVIAL which are noticeably deficient. The precision of numeric variables (number of bits), for instance, cannot exceed the number of magnitude bits in in one memory word. Frequently, greater precision is needed and the hardware may even support multiprecision arithmetic. JOVIAL seems curiously lacking in this area considering the myriad of sub-word capabilities that do exist. Such word size dependence may seriously inhibit the transfer of JOVIAL programs to "smaller" machines.

2.4 ASSEMBLY LANGUAGE PROGRAMMING FOR TRANSFERABILITY

2.4.1 Introduction

It is in assembly level programming that the problems associated with program transfer become the most acute and costly. The least significant progress made in regards to portability has been in this area. Yet, because of the large volume of programming still done at the assembly level, the greatest requirements for transferability solutions still exist.

Assembly languages, by definition, are designed for a particular machine to provide some way of specifying every function and combination of functions that the machine may perform. Assembly languages are most often used when no compiler exists for the machine; to specify functions not contained in higher level languages; to avoid some highly general and hence inefficient code generated by compilers; or to avoid interference between the compiler output and the desired program.

These conditions are most frequently encountered in the military and scientific environments because of the absolute need for maximum program efficiency, because many special purpose computers (usually not possessing FORTRAN or other higher level compilers) are utilized, and because of the frequent computer changeovers necessitated by new weapon systems.

Generally, when a programming language makes reference to the unique hardware of a specific computer system (sense switches, drums, discretes, analog I/O, etc.), it is not possible that programs written in this language can be directly transferred to another machine unless simulation techriques are utilized. Inherent characteristics such as word length, internal representations (binary, packed decimal, etc.), and character size produce similar difficulties when machine change is attempted. Word sizes, for instance, affect the results of numerical calculations, the storage space allocated to literal data, address sizes, direct value magnitudes, and many other less tangible aspects of the problem program. Often, even transfers to new systems on the same machine involve problems such as assembler instruction repertoire and structure. Apparently then, there is no way to write assembly code to make it automatically transferable. This unfortunate conclusion has caused many programmers to label assembly language transferability as "impossible" or "ridiculous" and to abandon all attempts toward achieving such a goal. Hence, some programs are written in a manner which dictates complete re-design, re-coding, and re-check out for each new system.

Clearly, the problem is one of degree. There are techniques for improving the transferability of assembly language programs. The following sections will deal more specifically with the factors involved and will present examples of methods currently utilized by groups actively involved in this problem. Perhaps as these concerned organizations develop workable techniques, skepticism will wane and more individuals will come forward with meaningful new ways to increase the portability of "machine oriented" programs.

2.4.2 Assembly Language Justification

There are many applications in the military, systems and scientific areas where assembly level programs are highly economical and unlikely to be transferred intact (ideas and techniques may be transferable while specific hardware is unique and short-term). In these cases, the use of assembly level coding needs no justification. In other more general applications, however, serious consideration should be given to the use of procedure and problem oriented languages.

The non-professional programmer is often more concerned with expressing and solving his problem rapidly than with obtaining maximum efficiency from the machine being utilized. For this type of individual, the facility by which he can write, modify, understand, and demonstrate his program is of foremost importance. In many instances, the actual code may be used as explanation and documentation of the problem being solved. Before such a program is implemented, a review should be made of the justifications most commonly given for the use of assembly languages.

- No higher level or macro language is available for the target computer and the cost/time of development of such a tool is prohibitive.

- Acceptable I/O and inner-loop speeds can be obtained in no other way.

- The problem program requires code complexity which cannot be obtained in any other way.

- Insufficient space requires extensive bit packing, machine dependent algorithms, etc.

- A level of security must be provided - programs must not be readily discernable by "outsiders".

- Unique machine features such as discretes, sense switches, and interrupts cannot be handled in another language.
- Timing and instruction counts must be exactly calculable.
- Extensive "patching" of assembled code may be necessary.
- Dynamic alternation of code is necessary - execution of the program will alter the instructions themselves.
- System or library routines with heavy usage should be as efficient as possible.

Many of the items listed above may not be relevant upon further investigation. Code complexity on most commercial machines, for instance, is seldom an excuse since FORTRAN and ALGOL type languages will handle a large percentage of these problems. Similarly, state-of-the-art techniques in overlays, paging, and storage allocation have alleviated many space problems. Assembly language "security" has really never been valid since obscure code can always be overcome by an experienced programmer with a strong desire to do so. Finally, patched code and dynamic source alteration are considered dubious programming techniques in most installations.

There are still instances when only assembly level and "machine-oriented" code is acceptable. A complex time-sharing system, an 8K airborne computer, a 9-bit process control monitor and myriad other applications require peak efficiencies in time and space utilization usually obtained only by assembly level languages.

It is in this area, programs that must be coded at the machine level, that efforts for increased transferability are most needed. Subsequent sections will present suggestions for the improvement of this situation.

2.4.3 Hardware Design Constraints

The computer designer has considerable freedom of choice in determining the final configuration of a computer system. The assembly languages ultimately provided for use of these systems will be largely pre-determined and severely constrained by the array of hardware features selected. It follows, then, that assembly language transferability is directly effected by basic hardware compatibility.

From a portability standpoint, one of the most difficult machine dependent areas to overcome is that of the computer's register architecture. This is especially true, when fully effective employment of the computer's particular register scheme is to occur. Currently, common computers range from having 16 general registers to one very limited register. This wide variance is further compounded by difficulties arising from differing register arrangements. Possibilities involved are: operations between general registers or registers and memory, transfers of data between registers of different lengths, registers altered by external devices, registers that count up, down or both, registers with distinct sign bits, and more.

Another basic incompatibility that must be dealt with is the current wide variety of addressing schemes available. Some of the more common methods include direct, base, relative, indirect and indexed. Virtually every combination of these types has been incorporated by some computer manufacturer. Even within a particular address type, there may be wide variations. For example, different quantity of index registers, single level or multiple level indirect, pre-indexing, post-indexing, etc. This area represents one of the most difficult transferability stumbling blocks since taking full advantage of existing addressing schemes is fundamental to program efficiency. Virtually all areas of a computer program will be effected.

An additional factor that often hampers the transfer of programs is the computer's internal method for representing negative numbers. If the computer uses a binary number system internally, number representation may be 2's complement, 1's complement, or magnitude and sign; similar choices may be made if the computer is decimal or byte oriented. One particular problem that arises is that some programs may make use of a negative zero, which does not exist in a 2's complement number system. Similarly, a programmer may use a negative constant as a logical mask. When transferring to another computer with a different negative representation, the mask usage may cause an error.

In yet another area, control of peripheral devices, both the design and use of assembly languages necessarily effects the original configuration of hardware. The equipment associated with a computer may vary widely in speed, capacity and mode of use. A disk memory may have fixed or multiple heads. Magnetic tape units may be seven or nine track, recording may be of different densities and speeds, and the tape unit may read only forward or may read both forward and backward. The number of printer columns may differ from one installation to another, or perhaps an output device may be capable of both printing and pictorial output. Input-output device priorities may be hardware or software determined. Communication with peripheral devices may be by means of busses, or direct wiring; in one case each device must be addressed and connected before data transfer occurs, in the other, these steps may be omitted. Input-output communications may be through the central processor, through a controller that steals memory cycles, or by a satellite computer that accesses the same memory devices.

In the very worst case, unconventional hardware may be introduced, with the result that either program libraries must be rewritten or else the capabilities of the computer will be very inefficiently employed. Unconventional hardware that may have to be considered includes microprogramming, associative memories, parallel processors, and pipeline processors. Microprogramming capabilities may permit the user to configure instruction codes to his own needs. Associative memories may greatly affect look-up or search procedures. Parallel processing, exemplified by Illiac IV, requires an entirely different type of operation. A computer with a software-alterable pipeline processing system would necessitate new programming and system operation techniques for effective use.

After a brief investigation into some of the factors involved in the design and use of an assembly language, it becomes clear that many computer programs cannot be effectively transferred without massive re-coding. This, of course, is without even considering basic differences in language mnemonics and formats. Upon elimination of the more extreme computer architectures, a list of basic machine differences may be compiled.

- Instruction repertoires - number, type and format
- Word size
- Page size
- Core size
- Address size
- Character and byte size

- Register architecture - general, base, index
- Cycle time
- Input/output characteristics
- Interrupt types, mechanisms and priorities
- Internal number representation - binary, packed decimal, etc.
- Floating point format
- Arithmetic - fixed point fractional, integer, etc.
- Multiplication and division characteristics - double, single, remainder, etc.
- Condition codes, overflows, sense lights, etc.
- Logical and magnitude operations
- Half, full, and double length instructions
- Half, full, variable length data reference capabilities
- Character set
- Negative number representation
- Shifts - sign excluded, sign propagated, single, double, variable length, arithmetic, logical, rotational, etc.
- Peripherals - speed, capacity, modes of use.
- Addressing - direct, base-displacement, indirect, relative, indexed, etc.
- Memory - scratchpad, drum, core, ROM, etc.

This list should be consulted (and augmented) whenever a system transfer is anticipated. In the event a new computer is to be obtained, a quantitative analysis of architectural differences weighted by their effect on program library transferability could be compiled for each prospective machine. This in turn may be weighed against the cost, power and general applicability of each candidate. Unfortunately, these weighing factors will vary considerably with the applications and machines involved. Hence, they cannot be rigidly defined for all machines in general.

2.4.4 Parameterization

In the previous section, a number of machine dependent characteristics were identified as probable hindrances to the transfer or assembly language programs. Fortunately, some of these variables can be effectively handled by parameterization techniques.

Frequently, an assembly level statement will be non-transferable simply because the programmer unnecessarily utilized machine-oriented knowledge in performing an application-oriented function. For instance, to left justify a character in a register (contending that this is a reasonable thing to do independent of a particular machine) he may have coded:

```
     SHIFT,LEFT    REGO,30
```

Embedded in this statement is a knowledge of the register (REGO) and character size (say 36 and 6 bits respectively). A more transferable approach would be to write:

```
     SHIFT,LEFT    REGO,WDSIZE-CSIZE
```

where WDSIZE and CSIZE had been parametrically defined (via an "EQU") at the beginning of the program.

Almost all programmers have been asked to modify certain aspects of their programs as a result of changes to specifications and requirements of the problem being solved or function being performed. These programmers rapidly learned to plan ahead for such events by parameterizing application dependent attributes such as table sizes, number of fields on a card, etc. Few will disagree that this practice has saved considerable effort when changes are finally requested. The same techniques should be encouraged for computer-dependent aspects of assembly language programs.

Depending on the application, many of the following machine features will appear in the code and thus will require parameterization.

- condition codes
- characters per word
- word size
- character size - bits per character
- address size
- page size
- right halfword (mask)
- left halfword (mask)
- character mask
- character set
- registers
- parity (0 for even, 1 for odd)
- -1 or other basic negative number
- sense devices (switches, lights)
- buffer sizes
- record sizes
- I/O devices
- origins (instruction counter values)
- key bit or character positions (for packed data)
- file marks

2.4.5 General Coding Techniques

The previous section contained a list of machine dependent items which are the most likely candidates for parameterization. In conjunction with this list, a set of general assembly language coding guidelines has been prepared. Many of the suggestions are based on composite characteristics of the currently most popular machines.

- When defining condensed data structures, no more than n bits of information should be packed into a single word, when n is the minimal word size of a computer to which the programs might be transferred.

- No immediate (direct) values should exceed 50% of the minimum word size or be relocatable.

- All shift instructions that are justifying or positioning data should use word-size and character-size parameters.

- Data should be declared on word boundaries.

- Logical masks should not be defined as negative numbers.

- Data constants and EQU parameters defined at a definite place in the program should be used in place of literals.

- Address and instruction modifications during execution (dynamic code replacement) should be avoided.

- Test characters and values against data constants defined earlier.

- Use symbolic names for all hardware such as registers, indexes, etc.

- Avoid relative code such as:

  ```
  JUMP $+5
  ```

  If necessary for some reason, parameterize the addresses as $+N5. This is especially important in transferring between byte address and word address machines.

- Explicitly state each equation, function, or decision tree in comments.

- List all system calls, their purposes and locations.

- Avoid complex floating point manipulations.
- Note any bulk I/O non-standard usage.
- Limit printed output to 120 characters per line.
- Use data masks to extract sub-units of a word (left half-word, second character, etc.).
- Avoid tests which are sign and arithmetic dependent. For example, whenever possible use "transfer on non-zero" rather than "transfer on plus".
- Do not assume pre-initialized memory.
- Avoid sign manipulation such as "set sign plus" or "complement and add" rather than "subtract".
- Use macros or subroutines for double precision operations.
- Don't depend on absolute core location knowledge - assume relocatability.
- Avoid indexed - indirect addressing.
- Use macros or subroutines for I/O operations.
- Note micro coded instructions - relying heavily on a micro coded search (B5500) or decimal (IBM 360) instruction may cause poor results when transferring to a new machine without these facilities.
- Be familiar with hardware trends. In particular, recognize engineer designed and programmer designed machines. For instance, code may be transferred more readily between two machines that are programmer oriented.
- Take advantage of special timing knowledge (external device, instruction execution, etc.) only if necessary.
- Try to avoid complex machine oriented scaling.
- Prepare a section (module) frequency chart. This will aid in general understanding of your program and may be used to estimate the time/cost effects of transfer

to new machines. Bascially, this chart should contain:

- function performed
- section of module involved
- estimated frequency per run
- estimated time per iteration
- estimated percent of total run time

- Prepare a register usage chart. This is of great benefit to other programmers who may be required to alter your program and it also helps to insure that the minimum number of registers is used at all times. This chart is usually subdivided by program section or subroutine and register. Each register in the module is then flagged by symbols indicating its use. For example,

  S - special purpose
  M - multi-purpose
  L - loop control
  X - index
  G - global
  A - arithmetic
  H - logical masks
  F - shifting

Additional guidelines of a more complex nature will be presented in Sections 2.4.6, Modularity, and 2.4.7, Macro Coding.

2.4.6 Modularity

It was pointed out in the discussion of higher level languages that the ability to divide machine dependent and independent attributes into separate entities greatly enhances the transferability of computer programs. This concept also pertains to assembly level coding. Programs can be modularized so as to render the environment of relatively independent sections invariant while identifying and isolating the more machine oriented areas.

Hence, program functions which will require a great deal of programmer attention upon transfer should be isolated in distinct modules. These modules, in turn, should be organized and documented such that they can be worked on with little reference to, or interference with, other modules. The rest of the program can then be automatically transferred, converted or re-coded with less effort.

Individual assembly of modules further aids the check out of transferred programs. Assembly languages are usually flexible in this regard, although it is generally necessary to distribute symbol definitions among modules by pre-loading the assembly table or by load-time resolution.

A theoretical modularity design specific enough to be effective yet general enough to cover a wide range of application/systems is a somewhat abstract concept. A review of assembly language programs, however, indicates many common features. The following outline, which reflects these similarities, is proposed as a basis for the modularization of assembly level programs. Although it seems functional in nature, this design serves to separate highly system dependent code (inner loop, interrupts) from less machine oriented activities (main body, data generation).

Modular Design of Assembly Programs

- Main Control
  - Prologue
  - Main Body
  - Epilogue
- Initialization
- Data Generation
- Outer Loop
- Inner Loop
- Output Data Routine
- Clean-Up
- Interrupts
- Error Handling

The purpose and content of these sections is defined on the following pages:

The main control section is intended to be the basis for the transfer of the program. It must contain sufficient information to allow other programmers to perform the re-coding, substitution of parameters, and other procedures necessary to update the program for the new machine. In particular, the prologue should include:

- storage layout
- program modules
- section frequencies
- constants and masks
- register definitions, macros, universal instruction definitions (see 2.4.7)
- character set
- common storage
- documentation as to purpose of the program
- comments regarding any of the above items which occur in other sections and the relationships involved

The main body consists of the main control logic and calculations. It will call upon the other sections including the system and termination procedures.

- main program
- calls to routines
- system service requests
- exit

The epilogue will summarize the machine and program characteristics not mentioned elsewhere. Generally, this will include:

- miscellaneous machine characteristics
- code anomolies
- system requirements

- eccentricities, idiosyncracies, tricks, etc.
- pointers to key areas where word, byte and other machine factors are of special importance

The initialization section sets up starting program data, initializes memory, restart parameters, path setting information, and other factors not handled in the prologue.

The data generation section either generates (via program algorithms) or inputs data to be used in other sections. This data should be well insulated from external devices. Hence, it must be presented in a format which is source device independent. Macros or, if possible, higher level languages can be utilized in many instances.

The outer loop section is somewhat artificial in that its primary function is to control the inner loop. It may be thought of as an interface between the main body (application oriented) and the inner loop (specific machine functions necessary to perform calculations).

The inner loop can be defined as those machine specific functions which must be re-coded for every machine. Candidates for this module(s) would be: functions which are very critical in terms of time, space, or accuracy; algorithms which are very difficult to parameterize but which utilize hardware characteristics; micro-coded instruction routines; and unusual external device interfaces. Elements which are placed in the inner loop are often responsible for the choice of assembly language as the implementation tool. By definition, the inner loop will probably have to be re-organized and re-coded for every machine. For this reason, it must be minimized by full utilization of parametric and macro coding techniques. Similarly, the inner loop must be carefully commented as to purpose, precisions, frequencies, machine oriented techniques (dynamic word modification, etc.), and application functions included.

The output data routing section will be the counterpart to the data generating section. It will convert data from a basically machine independent format to device oriented structures and perform the actual output commands. It may include preparation of data for handling by other programs written in different languages.

The clean-up section will prepare and make dumps for future restarts if necessary. It will also act as the counterpart to the initialization section.

The interrupt section will detect and process all interrupts (I/O, system, error, etc.). It may pass control to the error section for handling of recovery procedures if necessary. This section is usually very machine oriented and may require extensive re-organization during system transfer.

The error section will handle problems encountered in other sections and may be further sub-divided into software and hardware errors. With a great deal of care, this section can be made largely general in nature and will require varying degrees of re-work.

Of course, in practice it is often difficult to distinguish and separate machine dependent attributes. Especially in assembly language, machine independence is often only a matter of degree. Nonetheless, some efforts at modular design, when combined with parameterization and macro coding techniques, can produce noticeable reductions in the costs of program transfer.

2.4.7 Macro Coding

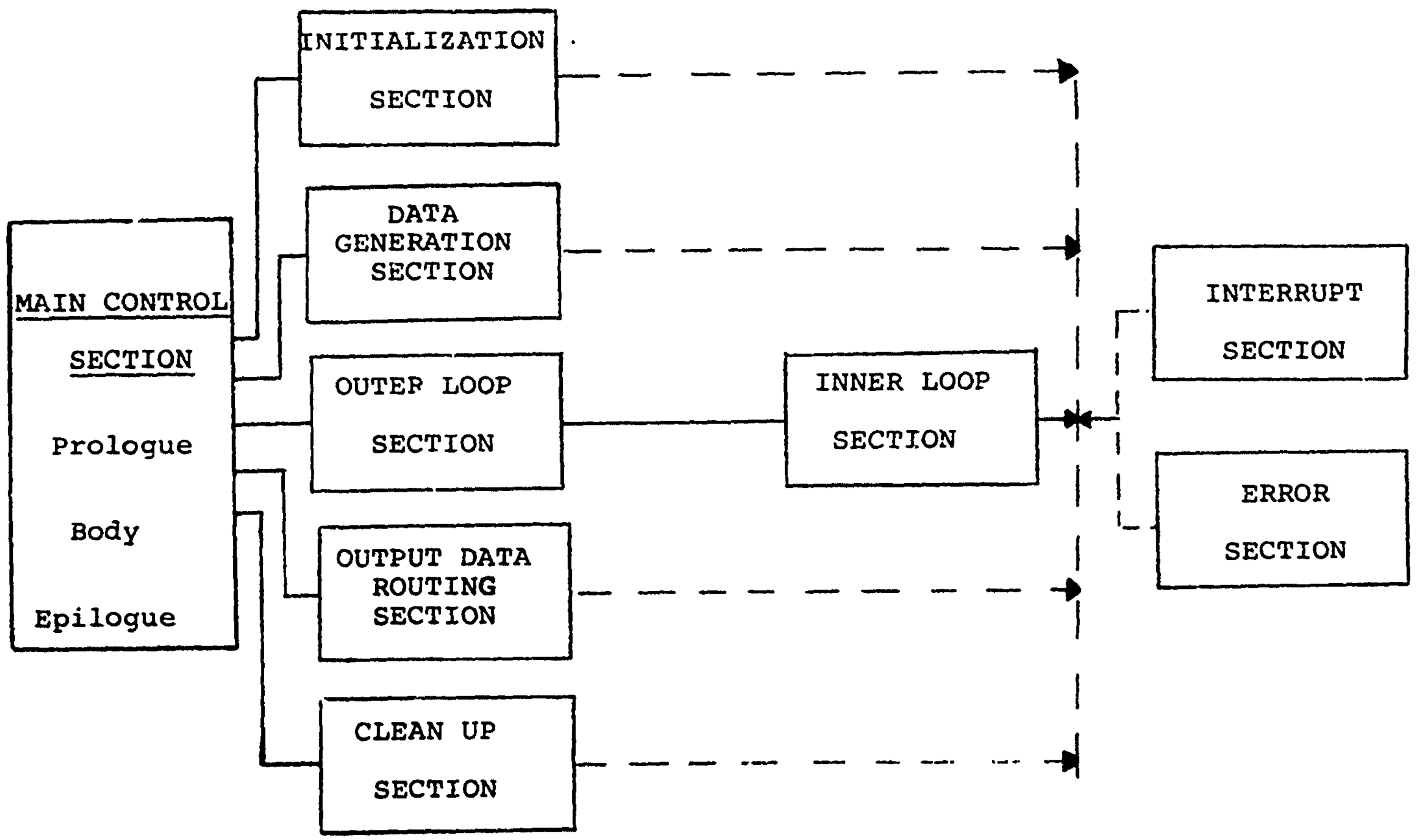
Many of the transferability advantages of higher level languages can be realized with the use of macro concepts. To the extent that existing macro assemblers allow for a common invocation format, a program written as a sequence of macro instructions can be transferred to machines of similar architecture by simply rewriting macro definitions. The costs involved in program transfer can be greatly reduced in this manner. In many instances, programs may be coded which are nearly as transferable as those written in higher level languages. Very little operating efficiency is sacrificed since the macro definitions are machine oriented.

For example, a triple address computer may be utilized with the macro:

```
ADD   AUGEND   ADDEND   RESULT
```

In one machine this may expand to:

```
CLA   AUGEND
ADD   ADDEND
STO   RESULT
```



SAMPLE LAYOUT OF ASSEMBLY LANGUAGE PROGRAM
Figure 2.4A

In another computer the result could be:

```
L     R1,AUGEND
A     R1,ADDEND
ST    R1,RESULT
```

Hence, simple macros may be developed which can be used as higher level language statements and which display noticeable machine independent characteristics.

There are many difficulties to be overcome in a macro processor approach, however. Assembler implementations cover a wide range of capabilities. Some are widely employed because of their power while others are so weak as to be virtually useless.

The most reasonable solution to this non-standardization problem is the development of an essentially machine independent assembler. This software should include powerful macro capabilities and extensive target computer parameterization facilities. Such products, usually called meta assemblers, do currently exist (for example, International Computer Systems' DUAL meta assembler is operational on numerous host computers generating code for a variety of targets).

A meta assembler allows for the description of pertinent target computer characteristics including the symbolic assembly format of the instructions. The macro facilities include means of conditionally generating different sequences of code based on the arguments of the calling lines. With such a tool, assembly languages may be developed which are capable of producing object code for a wide variety of machines. Additional benefits could be realized if the meta assembler itself was readily transferable (via self-assembly and interpretive techniques).

Some of the factors which are provided by the user to the meta assembler include:

- core size
- word size
- character size
- character set
- floating point format

- fixed point format
- negative number representation
- address size
- page size
- base registers
- object structures
- command mnemonics and formats

Usually, this information can be maintained in libraries along with the macro definitions.

Cross-assembly and source to source translation are additional transferability applications inherent in a meta assembler. The commonly used conversion technique of translation, for instance, can be implemented by macros. Source input language mnemonics become macro names while the expansions consist of equivalent output language statements. Any unresolvable ambiguities can be readily flagged as part of the macro process. Similarly, when two or more input statements must be combined, the assembler can allow for saving input lines in working parameter lists.

Assuming the availability of such a tool, plans may be made for the development of a "universal" assembly language for use on a variety of machines but maintaining machine oriented capabilities. This concept of course, is complicated by the fantastic variety of computer architectures presently in use. The definition of a universal assembly language must include methods of defining register implementation, addressing schemes, and less salient features of the computer architecture. Otherwise, highly inefficient and lengthy programs may result.

It is difficult to imagine an assembly language designed for efficient use of every machine in existence. Since many programs would not be coded in assembly languages if efficiency was not of primary concern, this facet of the problem cannot be ignored.

Keeping these conflicting factors in mind, the most acceptable compromise seems to be the maximum utilization of the original target computer's architecture. The degree of efficiency of code produced for new targets must become a function of the "likeness" of the new machine. This is entirely reasonable since one of the factors involved in the purchase of new equipment must be its compatibility with existing and future hardware implementations.

A most important facet of the development of a universal assembly language is the efficient use of the current register-address schemes in a manner readily adaptable to other computers. As yet, no solutions have been presented which promise to completely satisfy this requirement. One possible approach, however, is described below.

- A base register machine is assumed. The assembler should do as much of the base-displacement manipulation as possible and the programmer should use symbolic addressing. If the target computer is directly addressable, the base address defaults to 0.

- The user will declare registers and code with a particular architecture in mind. For example, in a machine with 16 general registers, instructions such as the following may occur.

  ```
  ADD  R1 R2 R2
  ```

  If this implies "add the contents of register R2 to register R1 and leave the result in R2", it assumes a knowledge of the previous contents of R1 and R2 and makes efficient use of the machine's architecture. In a single accumulator (R0) machine, the same statement might be processed (by the ADD macro) as "add the contents of memory location R1 to the contents of memory location R2 and store in memory location R2". It follows, then, that the efficiency of the program with respect to machine design is directly related to the "likeness" of the computers involved. Note that upon transfer of such a program, a new register declaration and set of macros must be generated.

- Registers can be used as for arithmetic, shifting, indexing, and other "normal" operations.

- Registers and memory should be considered interchangeable for most instructions.

- Indirect addressing and indexing can be used for address modification.

- A basic address operand format of:

  ```
  *register,index
  *address,index
  ```

  is assumed where * indicates indirect addressing.

- The basic instruction format is:

  ```
  INSTRUCTION,m a b c
  ```

  where a and b are inputs, c is the result, and m is an instruction modifier (see condition codes). With some instructions, a, b, c and m will be optional. a, b, c, will be considered addresses or direct values where appropriate.

- Condition codes will be handled by the macro processor such that mnemonics may be substituted for actual values. For example, assuming X, Y, and Z are addresses and V is a value,

  ```
  LOAD                   X
  COMPARE                Y
  BRANCH-ON-CONDITION    V,Z
  ```

  may be coded as:

  ```
  CW,GT     X Y Z
  ```

  which reads "compare words X and Y and branch to Z if X is greater".

- All macro (universal assembly language command) mnemonics will contain a $ (or any special character) to avoid conflicts with actual machine instructions.

With these ground rules in mind, a set of universal assembly language commands can be formulated. In the following instruction set definition, a few descriptive symbols are used.

- The "Type" column contains four items; A,I,R, and M. These subcolumns indicate some of the characteristics of each instruction. The column explanations are as follows:

A - The number of address type operands associated with the instruction.

I - A check in this column indicates that an immediate value is presented as part of the instruction.

R - A check in this column indicates a relational is presented as part of the instruction. The relationals that may be associated with the "compare" and "jump if index" commands are:

EQ - Equal
GT - Greater than
GQ - Greater than or equal to
LT - Less than
LQ - Less than or equal to
NQ - Not equal

The relationals that may be associated with the JUMP command are:

P - Positive
N - Negative
V - Overflow
Z - Zero
NZ - Non zero

M - A check in this column indicates that the number of words, bytes, or bits operated upon is supplied as part of the instruction. For example,

```
MW$,3     A B
```

is interpreted as "move 3 words from A to B".

- In the function column, a, b, and c are used to represent the operands involved in the command. The letter v is used to represent immediate values.

UNIVERSAL COMMAND REPERTOIRE

COMMAND		TYPE				FUNCTION
Mnemonic	"English-like"	A	I	R	M	
Arithmetic:	(Fixed Point)					
A$	ADD$	3				$b + a \rightarrow c$
S$	SUBTRACT$	3				$b - a \rightarrow c$
M$	MULTIPLY$	3				$b \times a \rightarrow c$
D$	DIVIDE$	3				$b \div a \rightarrow c$ remainder → $c + 1$
AV$	ADD$VALUE	2	*			$a + v \rightarrow b$
SV$	SUBTRACT$VALUE	2	*			$a - v \rightarrow b$
MV$	MULTIPLY$VALUE	2	*			$a \times v \rightarrow b$
DV$	DIVIDE$VALUE	2	*			$a \div v \rightarrow b$ remainder → $b + 1$
Logical:						
AND$	AND$LOGICAL	3				$a . b \rightarrow c$
ORE$	OR$EXCLUSIVE	3				$a \oplus b \rightarrow c$
ORI$	OR$INCLUSIVE	3				$a + b \rightarrow c$
CA$	COMPLEMENT$ADD	2	*			$a + v \rightarrow b$
ANV$	AND$VALUE	2	*			$a . v \rightarrow b$
OEV$	OR$EXCL$VALUE	2	*			$a \oplus v \rightarrow b$
OIV$	OR$INCL$VALUE	2	*			$a + v \rightarrow b$
Manipulative:						
MW$	MOVE$WORDS	2			*	$a \rightarrow b, \ldots, a+n-1 \rightarrow b+n-1$
MWV$	MOVE$VALUE	1	*		*	$v \rightarrow a, \ldots, v \rightarrow a+n-1$

COMMAND		TYPE				FUNCTION
Mnemonic	"English-like"	A	I	R	M	
Manipulative:						
MWR$	MOVE$REPEATEDLY		2		*	a → b,...,a → b+n-1
MB$	MOVE$BYTES		2		*	Byte 1 → b,...,Byte n → b+n-1
MLB$	MOVE$LEFT$BITS		2		*	n leftmost bits are moved from a to b
MRB$	MOVE$RIGHT$BITS		2		*	n rightmost bits are moved from a to b
MBR$	MOVE$BYTE$RIGHT		2		*	Specified byte of a is moved to rightmost byte of b.
Shifts:						
SLA$	SHIFT$LEFT$A	1	*			The specified registers are shifted left arithmetically.
SRA$	SHIFT$RIGHT$A	1	*			The specified registers are shifted right arithmetically.
SLL$	SHIFT$LEFT$L	1	*			The specified registers are shifted left logically.
SRL$	SHIFT$RIGHT$L	1	*			The specified registers are shifted right logically.
SLC$	SHIFT$LEFT$CIRC	1	*			The specified registers are shifted left circularly.
SRC$	SHIFT$RIGHT$CIRC	1	*			The specified registers are shifted right circularly.

COMMAND		TYPE				FUNCTION
Mnemonic	"English-like"	A	I	R	M	
Shifts:						
SRP$	SHIFT$RIGHT$PROP	1	*			The specified registers are shifted right arithmetically with the sign bit propagating.
Sequence Control:						
CW$	COMPARE$WORDS	3		*		Execution control transfers to c if a "relation" b is true.
CBS$	COMPARE$BYTES	3		*	*	Execution control transfers to c if Byte string a "relation" Byte String b is true.
CLB$	COMPARELEFTBITS	3		*	*	Execution control transfers to c if Bit String a "relation" Bit String b is true
CRB$	COMPARERIGHTBITS	3		*	*	Execution control transfers to c if Bit String a "relation" Bit String b is true.
CWV$	COMPARE$VALUE	2	*	*		Execution control transfers to c if a "relation" v is true.
CL$	COMPARE$LIMITS	3		*		Execution control transfers to c if a "relation" b and b+1 is true.
J$	JUMP$	1				Unconditional transfer to a.
J$,V	JUMP$,OVERFLOW	1		*		Execution control transfers to a if the overflow indicator is set.

COMMAND		TYPE				FUNCTION
Mnemonic	"English-like"	A	I	R	M	
Sequence Control:						
J$,P	JUMP$,POSITIVE	2		*		Execution control transfers to a if b is positive.
J$,N	JUMP$,NEGATIVE	2		*		Execution control transfers to a if b is negative.
J$,Z	JUMP$,ZERO	2		*		Execution control transfers to a if b is zero.
J$,NZ	JUMP$,NONZERO	2		*		Execution control transfers to a if b is non-zero.
C$	CALL$	3				Call subroutine a; put return address in b; calling sequence is at c.
E$	EXIT$	1	*			Returns v cells beyond subroutine return address a.
JXD$	JUMP$IF$DECR	2	*			Execution control transfers to a if register b is decremented.
JXI$	JUMP$IF$INCR	2	*			Execution control transfers to a if register b is incremented.
JII$	JUMP$IF$INDEX	2	*	*		Execution control transfers to a if b "relation" v is true.

COMMAND		TYPE				FUNCTION
Menmonic	"English-like"	A	I	R	M	
Miscellaneous:						
I$	INPUT$	3				The specified quantity of data is input from the selected input device to the specified destination.
O$	OUTPUT$	3				The specified quantity of data is output onto the selected output device from the specific location.
XCH$	EXCHANGE$	2			*	The specified number of words starting at a and b are interchanged.
XEC$	EXECUTE$	2				The instructions starting at a down to b are executed.

Instructions of less general applicability which may be included are:

- Floating point arithmetic
- Peripheral equipment controls (rewind, backspace)
- Stack oriented instructions (PUSH,PULL)
- Conversion operations
- Search operations

In all instances, the suggestions made in previous sections regarding parameterization and general coding techniques may be incorporated into programs written in macro languages.

Some of the benefits which are to be expected from this approach include:

- An awareness of transferability considerations before and during coding.
- Relatively efficient use of original target computer architecture.
- Programs which are "very" transferable to machines of like architecture.
- Feasible transfer of programs between machines of different architectures (assume some loss or tradeoff of efficiencies).
- Standardization of assembly mnemonics and formats to reduce re-training costs.
- A basis for quantitative analysis of costs of transfer versus efficient use of machine characteristics (especially when considering the purchase of new hardware).
- Possible check-out of programs for "un-implemented" or currently unavailable machines on another computer.
- A recognizable starting point on which to base additional transferability effort.

2.4.8 Policy

Major hardware advances in a relatively short period of time have magnified and confused the problems of program transferability. Basic assembly languages, because of their inherent references to unique hardware designs, have presented the most acute obstacles to these changes.

The urgency associated with this problem seems to have been responsible for the after-the-fact attributes common to many current efforts. These range from source-to-source translation through complete re-coding. Generally, these approaches have been costly, machine specific, and only partially effective. It seems desireable, therefore, to redirect efforts at assembly language program transfer to the pre-program design stage.

This implies that assembly level programmers must be made aware of the potential for system transfer and the effect of their coding techniques on the success of such ventures.

Similarly, each programming organization must develop standards for design and coding of assembly language programs as they have done for COBOL, FORTRAN, and JOVIAL. The programmers should be encouraged to help develop these standards.

Additional areas where standardization might be imposed include the actual assembly languages and the equipment to which they are so closely related. It seems restrictive and unwise to require manufacturers to structure their hardware to be consistent with a standard assembly language. On the other hand, it does seem possible to design a standard pseudo-assembly language. Although this was discussed in Section 2.4.7, some additional comments seem appropriate at this time.

An unquestionably important factor in the success of assembly program transfer is the programmer's individual awareness and involvement. The use of universal assembly language with machine independent aspects is a possible method of solidifying these feelings. The knowledge that the language being used can be modified and improved by the programmer may engender such efforts (at least many have expressed this desire).

Similarly, the wide spread and often costly pre-occupation with "tricky", "shorter" or "faster" coding algorithms could conceivably be diverted to the development of "independence" algorithms. Dedication to the invention of clever ways to perform seemingly machine dependent functions in a universal or parameterized manner is certainly desireable.

Hence, the development and use of a standard assembly language, when combined with an awareness of transferability oriented coding techniques, could result in significant improvements. A recognizable effort in this area could influence machine manufactures toward partial standardization of character sizes, object formats, machine code mnemonics, etc.

Unfortunately, there will undoubtedly always be programs to which these techniques do not apply. Airborne computer applications, for instance, are often so limited in resources and application dependent that efforts at transferability would be wasted. There is considerable area for improvement, however. Elegant, abstruse programs written by the experienced coder to save a little time or space and unnecessarily complex concoctions produced by the inexperienced programmer through ignorance can both be eliminated. In the area of assembly level transferability, any improvement will be a major one.

APPENDIX A

FORTRAN Program Transfer

A small FORTRAN program was prepared, compiled and executed on several machines as part of this study. Each of the statements in the program would, theoretically, compile and execute "as expected" on at least one major computer's FORTRAN System. All statements which would not compile properly or which were rejected by the compiler were removed from the program before recompilation and execution.

It should be noted that other compiler versions of many of the computer systems utilized would display different results. It is also interesting that no particular patterns can be realistically formed from these results. Even more advanced compilers rejected some statements found acceptable to other more "primitive" systems.

A summary of the features of this transfer which would pose obvious transferability difficulties follows. In the first chart, Compilation Characteristics, several definitions are used.

F - The compiler flagged this statement or function as a probable error.

R - The compiler rejected this statement and produced no output for it.

A - The compiler accepted (compiled) this statement without special notes, errors, warnings, etc.

I - The compiler literally ignored the non-standard aspects of this statement. For instance, INTEGER*2 was read as INTEGER.

FORTRAN Program Transfer - Compilation

FORTRAN Characteristic	UNIVAC 1108	IBM 360/65	CDC 6400	IBM 7044
Un-initialized variables	F	A	A	A
IMPLICIT REAL	A	A	R	R
Four dimensions in an array	A	A	R	R
Halfword variables	I	A	R	R
Mixed mode arithmetic	A	A	A	F
Expressions for subscripts	R	A	R	R
Real-valued subscripts	R	A	R	R
Multiple assignments (X=Y=Z)	R	R	A	R
Special Characters (>)	R	R	R	R
Literal strings enclosed by apostrophes	A	A	A	F
Division by a zero-valued expression	F	A	A	A
An IF statement as the terminus of a DO loop	A	R	R	A
Real variable in assigned GO TO	R	R	R	R
Real variable as loop index	R	R	R	R
Incomplete arithmetic IF (2 arguments)	A	R	A	R
Arithmetic expression for a logical unit	R	R	R	R

FORTRAN Program Transfer - Compilation

	UNIVAC 1108	IBM 360/65	CDC 6400	IBM 7044
Transfer into a DO loop	A	R	A	A
Alteration of a DO index inside the loop	F	A	R	F
6H hollertith data	A	I	A	A
Octal constants	A	R	R	A
Hex constants	R	A	A	R
Test for equality to 0.0	F	A	A	A
Multiple ENTRY	A	A	A	R
RETURN in the main program	A	A	R	A
X**Y**Z	A	A	A	F
Function which alters its input parameters	A	A	R	A

FORTRAN Program Transfer - Execution

Characteristic	Execution Results			
	UNIVAC 1108	IBM 360/65	CDC 6400	IBM 7044
Division by zero	result=0	result=1	result=0	result variable
Ending value of a DO index	terminus	terminus	terminus+1	terminus+1
Last index of a DO loop with inexact terminus EX: DO 10 I=2,9,2	terminus-1	terminus-1	terminus-1	terminus+1
Input of real variable via an I format (value=2)	$.5\text{x}10^{-38}$	$.1\text{x}10^{-82}$	0.0	0.0
"Shift" of -1 to the left then right by 33 "places" using multiplication and division	-1	0	-1	-1
(-1)*2**3**2	$(-1)*(2^3)^2$	$(-1)*(2^3)^2$	$(-1)*(2^3)^2$	$(-1)*(2^3)^2$
Value of "zero" computation and test for 0.0	$-.1\text{x}1\bar{0}^{7}$(failed)	$-.2\text{x}1\bar{0}^{6}$(failed)	0.0(passed)	$-.7\text{x}1\bar{0}^{8}$(failed)
Value of real subscript		lowest integer		
Value of a random cell	0	0	flagged	0
Transfer into a DO	no problem	no problem	no problem	no problem

FORTRAN Program Transfer - Execution

(continued)

Characteristic	Execution Results			
	UNIVAC 1108	IBM 360/65	CDC 6400	IBM 7044
Alteration of a DO index within the loop	no problem	no problem	nc problem	no problem
Terminal value of a large index	0	terminus	terminus+1	terminus+1 modulo 100,000
Mixed mode expressions	OK	OK	OK	nonsense
Value of X=X+FUNC(X)*X where FUNC(X) changes X	$X_2=X_0+F(X_0)*X_1$	$X_2=X_0+F(X_0)*X_1$	$X_2=X_0+F(X_0)*X_1$	$X_2=X_1+F(X_0)*X_1$

APPENDIX B

Transferability Factors

- Operating system interfaces
- Dialects, extensions, subsets and abbreviations in higher level languages
- Code restraints imposed by interpretation of language standards
- Character set differences - internal, tape, and card representations and collating sequences
- Internal clock units
- Recording modes - record sizes, block sizes, densities, parities, control words, file limits, volume sizes, carriage controls, end of the line marks, etc.
- Computer speed
- Core size
- Peripherals - number and type
- Sense devices - switches, lights, etc.
- Operator interface
- Printer speed and line size
- Negative number representations
- Internal number representation (packed decimal, binary, etc.)
- Interrupts - types, mechanisms and priorities
- Boundary alignment - halfword, doubleword
- Condition codes, overflow bits, and other indicators
- Channel processing
- Memory types - ROM, scratchpad, etc.

- Instruction repertoires
- Variable word length instructions
- Division/multiplication algorithms
- Rounding
- Shifting algorithms
- Absolute addressing
- Relative addressing
- Indirect addressing (with/without index, etc.)
- Dynamic instruction modification
- Address size
- Page size
- Direct value size
- Word size
- Character size
- Object format
- Indexing
- Register schemes
- Pre-initialized memory
- Micro-programs
- Input/output timing controls

APPENDIX C

Software Products Synopses

These synopses are presented as a sample guide to the general nature of commercial conversion services and products. Capabilities and limitations are listed by way of example and are not intended to reflect the exact attributes of the associated products. A choice of conversion aids should not be limited to or influenced by this material.

AED Cross Compilers
SoftTech, Inc.

AED languages feature a high degree of modularity with target language modules treated as off-the-shelf software components.

BASIC to FORTRAN Translator
International Conversion Systems

Translates BASIC language to FORTRAN on a timesharing system to provide the facilities of BASIC to IBM 360 users.

BCD to EBCDIC
International Business Machines

Converts BCD punched cards to EBCDIC format.

BIAS - Basic Iterative Assembler for DEC PDP-11
COMPATA, Inc.

"Can be used on any host computer which supports ASA standard FORTRAN" with 23 bits plus sign precision, card reader, card punch, printer and one tape.

BIRS - Basic Indexing and Retrieval System
Information System Laboratory, Michigan State University

This collection of 150 FORTRAN routines includes the TRAC debug tool. TRAC is written in FORTRAN and is a pre-compiled debug tool for FORTRAN programs. This highly machine independent program provides sophisticated loop detection printouts and diagnostics.

BUFF40X
Nanodata Corporation

A special fast FORTRAN compiler with enhanced printouts of storage allocation, expanded diagnostics, traces, timing, etc. Operates on IBM 360/25 and up.

CATALIST - Autocoder to BAL Translator
International Business Machines

A conversion aid for transition from IBM 1401 Autocoder to IBM 360 BAL.

COM/FORT Programming Aids
ADAPT, Inc.

Subroutines designed to interface commercial applications with FORTRAN to extend the standard. They can also be invoked via BAL, COBOL and PL1 and increase speed and remove input/output restrictions such as record sizes. They run under DOS or TOS.

COBOL Language Conversion Program for IBM 1401
International Business Machines

Facilitates transition to IBM 360 by converting COBOL source written for IBM 1401 to System 360 Level E or Level F.

DECIBLE III
Independence Computing and Software

General decision table processor written in COBOL.

DSI-11
Decision Science, Inc.

Cross assembler for PDP-11 written in FORTRAN.

DUAL
International Computer Systems, Inc.

A procedure oriented macro processor with extensive cross-assembly and language generation facilities. Largely host and target machine independent.

DUO 360/370
Computer Technology, Inc.

The DUO package enables the user to run IBM 360/370 DOS object programs, without modification, under OS and to use the OS features via JCL.

EXODUS I and EXODUS II
Computer Sciences Corporation

Respectively translate IBM 1410 and 1401/40/60 AUTOCODER or SPS source to IBM System 360 BAL.

HELP - Highly Extendable Language Processor
Advanced Computer Techniques Corporation

Allows programmer to define extensions to FORTRAN, COBOL and PL1 via macro definitions. Translates extended statements to conventional language forms in a pre-compile phase.

IBM 7040/7044 Emulator Program
International Business Machines

The emulation is performed on the System 360 and consists of the emulator program and special machine additions called the 7040 compatibility feature.

MECCA 360/370
Arkay Computer Applications

"Monitor and Enforce Conformance to COBOL Administrative" standards. Inspects COBOL source for conformance to 200 management selected standards and lists corresponding severity codes and messages. Reserved words (abbreviations) can be prescribed.

MASTRAN
National Information Systems, Inc.

1401 AUTOCODER to System 360 BAL conversion service. "No charge for less than 90% conversion."

METAPLAN
Programmatics, Inc.

A procedure oriented macro language assembler with machine independent attributes.

MINI-SIM
Trippe Systems Incorporated

A 360 BAL program for simulating a variety of mini computer programs including control panels. Allows use of 360 peripherals by mini program for rapid checkout.

OPTIMAIL
Clasco Systems, Inc.

COBOL source program optimization by mail. Promises 25% reduction in computer time.

OPTIMIZER
CAPEX Corporation

Global analysis of IBM 360/370 COBOL compiler object output to minimize object code (time/core).

PASSPORT
Creative Computing Techniques

A general software-assisted program conversion service based on pattern matching of source/destination templates for various languages and machines. Conversion templates are tailored to customer habits, and CCT personnel assist in manual work.

PL1 Checkout Compiler
International Business Machines

PL1 program modules translated by this product run more efficiently and can be link-edited with output modules from OS FORTRAN and COBOL.

REPRIEVE
National Information Systems

A software package which allows RCA TDOS users to convert to IBM 360/370 without modification or system regeneration.

SIFT
SHARE

Operates on FORTRAN II source decks to produce basic FORTRAN IV source.

STAGE II
Optimization Sciences, Inc.

COBOL source program optimizer to reduce run time and core size by suggesting programmer changes to the code.

TACOS
Management Systems Corporation

Translates AUTOCODER 76 to IBM 360 COBOL for OS. Handles about 90% of source statements.

TOTALTRAN
CPU Management Advisory Corporation

Translates 1401 object decks to System 360 assembly source. Includes disassemblers to produce AUTOCODER source from patched 1401 or 7070 object.

TRANSEMBLER
Code, Inc.

Cross assembly for the Varian 620/f written in FORTRAN.

Universal Assembly System for One-Address Minicomputers
Jack R. Guskin, International Business Machines
(Not an IBM Product)

Uses a well-defined fixed syntax and dynamically modified semantics dependent upon target machine characteristics.

UPGRADE
Information Management, Inc.

De-compile or translate IBM 1401, 1440, or 1460 source to COBOL source. Includes manual pre-translation, machine translation with flags, manual post-translation.

APPENDIX D

Document Synopses and Bibliography

Synopses

BALZER, R. M.
Dataless Programming

All data and function references are expressed in a single canonical form so that the alteration of a data representation has no syntactic effect on the program. A PL/1 extension.

BARTON, M. E.
SWAP: The Macro Assembler

Examples of features included in a macro assembler. Recursive function definition, ellipsis notation for arbitrary number of arguments, character string partitioning and naming are mentioned. Includes a 3½ page listing of "A SWAP Program to Interpretively Process a FORTRAN Like Language". Points out that macro facilities do not have to be out of reach of the average user programmer.

BELL, C. GORDON AND ALLEN NEWELL
Computer Structures: Readings and Examples

...presents many examples of computer systems...in enough detail (for) meaningful...study and analysis...using the original descriptions...in the technical literature. Collected comparisons of machine features, some of which are pertinent to transferability. (Those not transparent to the program.)

BEMER, ROBERT W.
Program Transferability

Program Transfer is complicated by each element which is different. Programs must be planned for transfer...(has) not yet been successful mechanical translation of machine language program. Lists information needed to transfer (run) a program. Suggests tools for conversion checkout. Suggests that compiler should capture information and describe its own peculiarities.

DELLERT, GEORGE T. JR.
A Use of Macros in Translation of Symbolic Assembly Language of One Compiler to Another

A set of macros used to help translate IBM 7090 Assembly Language to IBM 7040 Machine Language.

DONOVAN, JOHN J. AND HENRY F. LEDGARD
A Formal System for the Specification of the Syntax and Translation of Computer Languages

This paper deals with the use of established methods of recursive definition to present a single method to:

1. Specify the syntax of computer languages.

2. Specify the translation of programs in one computer language into programs in another language.

ELSPAS, BERNARD, MILTON GREEN AND CARL LEVITT
Software Reliability

We examine...approaches to improving the art of computing, debugging and verifying programs,...improvements in programming languages,...debugging techniques that incorporate semantic analysis in the...compilation,...possibilities for the logical proof of the correctness of a program.

ENGLANDER, WILLIAM R.
How Standard are COBOL Compilers?

COBOL compiler validation system (Air Force) used to audit computers for:

B2500/B3500 COBOL
GE 625/635 COBOL
S/360 COBOL F
UNIVAC 1108 COBOL EXEC II

Test failures are reported without identifying which compiler. Suggests using CCVS to select both a compiler and those features to be avoided as not compatible.

ERICKSON, BEVERLY
A Program Disassembler

The concept of object program disassembly is presented as the final step in a series of three utility programs. Flowcharts and examples are included.

FALOR, KEN
Survey of Program Packages

Lists source, name, description, computer, compiler and price for software items in the categories conversion, translation and simulation programs (19 items); debugging and testing programs (23 items); JCL compilers (6 items); documentation and flowchart producing programs (12 items); languages (17 items); preprocessors for decision tables, shorthand languages, etc. (19 items); other general purpose programming aids (14 items).

FERGUSON, DAVID E.
Evaluation of the Meta-Assembly Program

A generalized meta assembler is described. Evolution of the assembly program and its implication to compiler design are discussed.

FRITZ, W. BARKLEY
Computer Change at Westinghouse Defense and Space Center

Management guidelines for transferability.

FUJII, ATSUSHI
Software in Japan: Supported Growth

The Japanese Government is encouraging and supporting software development in several ways. A new organization will buy software packages from developers and lease them to users. The government thus assumes the majority of the risk. The government gives examinations for information processing engineers. In the test for programming, FORTRAN, COBOL or Assembly Language could be chosen for solving the problems. The Japanese Government has launched a 5-year project (large Industrial Engineering development) to upgrade the technology of the Japanese information processing industry. One program is for the development of common, hardware independent, software. FORTRAN, COBOL and PL/I compilers written in a subset of PL/I, translate to a common hardware independent intermediate language. Hardware dependent compilers translate the intermediate lanquage.

GAINES, R. STOCKTON
On the Translation of Machine Language Programs

Four classes of problems in translation:

1. Untranslatable segments, e.g. a core memory test for a particular machine.

2. Idiomatic expressions, e.g. the conversion of an integer to floating point on the CDC 1604 by doing an integer add and floating subtract of a particular constant.

3. Program self-modification, e.g. altering an address or an operation code in an instruction.

4. Differences in machine design, e.g. different word sizes, different instruction details.

Discusses simulation and automatic translation (of a programmer pre-edited source deck followed by programmer debugging). The problem statements suggest a partial answer: Syntax should accurately reflect semantics, e.g. 'Float' vice 'integer add, floating subtract'. Such can best be done with macros.

GODSEN, JOHN A.
Software Compatibility: What Was Promised, What We Have, What We Need

In very general terms, this paper defines compatibility; defines kind of, degree of, extent of and direction of compatibility; lists zones of compatibility ranging from one installation of a computer to all computers in Air Force Logistics Command; lists seven implicit promises made by producers of higher level languages and the extent to which the promises were kept.

GODSEN, JOHN A.
Software Compatibility

The subject of this paper is data compatibility.

GRAHAM, MARVIN LOWELL AND PETER ZILAHY INGERMAN
An Assembly Language for Reprogramming

Describes an early Meta-assembler.

GRIES, DAVID
Compiler Construction for Digital Computers

Theory and examples of higher level language compilers design including syntax, semantics, optimization, interpretive techniques and macros.

GRUENBERGER, FRED
Problems and Priorities

The problems facing the computer industry as of 1972 are listed in order of importance. Language standardization is given the lowest priority.

GUSKIN, JACK R.
A Uni-Language for a General Purpose Assembly Translator System for Single Address Machines

A discussion of translators capable of accepting a variety of source languages.

HALSTEAD, MAURICE H.
Machine Independence and Third-Generation Computers

An example of the use of a decompiler, converting to NELIAC.

HALSTEAD, MAURICE H.
Using the Computer for Program Conversion

Discusses decompilers which accept an absolute binary program, analyze it in great detail, translate it into an equivalent program in a higher level language and flag ambiguities. Discusses the economics of writing and using a decompiler.

HARPER, WILLIAM L.
Documentation for Application Programming

The function of documentation is defined in general terms.

HICKS, HARRY T. JR.
ANSI Cobol

Discusses the levels of ANSI COBOL, compiler validation, and the transition from prestandard COBOL. The arrival of the ANSI COBOL compilers will require many COBOL users to convert their existing programs. The organization can adopt a standard dialect to facilitate programmer training and make available a richer choice of future computers and compilers.

HOPPER, GRACE MURRAY
Standardization of High-Level Languages

The common element essential to mobility is the establishment of standards for programmers, programs, and documentation.

JONES, RONALD W.
Generalized Translation of Programming Languages

A generalized model for the computer translation of both programming and natural languages has been developed at the Linguistics Research Center of the University of Texas. The theory says there are two structures underlying a language - surface structure and basic structure.

LEE, JOHN A. N.
The Anatomy of a Compiler

An author who has written many practical language processors communicates his experience, choosing FORTRAN as an example. The exposition is concrete (thoroughly exemplified), a blend of theory and practice.

MEALY, GEORGE H.
Another Look at Data

Proposes theoretical model for data and data processing.

MILLS, HARLAN D.
Syntax-Directed Documentation for PL 360

The language PL 360 with its phrase structure grammar is used as a concrete basis for illustrating an idea called Syntax-Directed Documentation. The idea is:

1. To use the phrase structure of a program to define the structure of a formal documentation for that program.

2. To use the syntactic types and identifiers in the resulting structure to trigger the automatic formation of questions to the programmer whose answers will become part of that documentation.

3. To provide automatic storage and retireval facilities.

A small PL 360 program is worked out as an example.

MORENOFF, EDWARD
The Transferability of Computer Programs and the Data on Which They Operate

History and overview of RADC interest in transferability.

MULHOLLAND, K. A.
Software to Translate Telcomp Programs into KDF9 Algol

Lists the KDF9 Algol statements chosen to correspond to Telcomp statements. Not a particularly successful effort.

ORGASS, RICHARD J. AND WILLIAM M. WAITE
A Base for a Mobile Programming System

Describes an algorithm for a macro processor which has been used as the base for an implementation, by boot strapping, of processors for programming languages. This algorithm can be easily implemented on contemporary computing machines.

RECTOR, DR. ROBERT W. AND DR. C. J. WALTER
The Fourth - Another Generation Gap?

Some anticipated characteristics of fourth-generation computer systems are briefly discussed. The view is presented that computer designers do not understand the needs of the user; software production is a significantly different effort from software use and designers tend to optimize for software production.

REENSKAUG, TRYGVE
Some Notes on Portable Application Software

Notes based on experience with the AUTOKON system for design of ship's hulls. Three methods were tried - reprogramming - time consuming, expensive, boring, error prone. Generative coding via SLEUTH II; dropped because producing relocatable object decks for ill-documented operating systems was difficult and decks not customer-maintainable. High level language now method of choice, becoming restricted to USASI BASIC FORTRAN. Equivalence between real and integer arrays, low precision real have caused problems. Peculiar restrictions on FORTRAN unit numbers, paper tape not well supported. Operating systems have strong opinions about disc data physical organization.

SABLE, JEROME D.
Transferability of Data and Programs Between Computer Systems

The problem of transferring data and interpreting it can be approached by constructing an adequate range of data structure types and devising a standard way of describing these types (Data Description Language).

SATTLEY, KIRK, ET. AL.
On Program Transferability

Suggestions on transferability enhancements effecting the total program transfer problem with emphasis on operating systems. Prepared for RADC.

SHAW, CHRISTOPHER J.
A Comparative Evaluation of JOVIAL and FORTRAN IV

Side-by-side outline of the advantages and capabilities of each language. Identity, applicability, origin, dialects, structures, input/output and modularity are discussed with suggested uses of each language. Low-key comments regarding portability are included.

SAMMET, JEAN E.
Programming Languages: History and Fundamentals

An overall view of higher level languages. This book brings together in one place fundamental information about programming languages, including history, general characteristics, similarities and differences, as well as detailed technical descriptions. The book contains basic information and provides perspective on 120 higher level languages, including all the major, and most of the minor, languages developed in the United States. Major languages are described in a consistent way, following a fundamental outline and general discussion of technical and non-technical characteristics of programming languages.

SCHAEFER, MARVIN
DBL: A Language for Converting Data Bases

Describes a program to transfer data from one data management system to another. It is not clear from the article whether it will handle transfers from one operating system or one machine to another.

SIEGEL, STAN
WATFOR...Speedy FORTRAN Debugger

WATFOR is a FORTRAN compiler developed by the University of Waterloo, Canada. It provides diagnostics on usages that some FORTRAN compilers overlook and, by chance handle "properly". It shows definite transferability oriented design in the areas of code restraints and standardization.

VAUGHN, PETER H.
Can Cobol Cope?

The COBOL environment is fundamentally hostile to effective modular design. Business data processing applications are less rigorously designed than are scientific applications. Both considerations make COBOL programs hard to maintain and, although the paper does not mention it, hard to transfer unless the destination compiler is fully compatible.

WALSH, DOROTHY A.
A Guide for Software Documentation

A series of outlines or models is given for the standard manual and references written to describe software. The outlines give a basic minimum content list and provide a basic minimum features checklist for the software itself. Text gives directions for adapting an outline for a general document type to the specific software item to be described.

WARD, JAMES A.
Program Transferability

An overview of the need for and progress toward easier transfers.

Bibliography

ACM SIGPLAN, USASI COBOL STANDARD ANSI X3-23
April 1968.

ACM SIGPLAN, USASI FORTRAN STANDARD ANSI X3.9, 1966.

ALT, FRANZ L. AND MORRIS RUBINOFF
Standards for Computers and Information Processing
advances in computers, vol. 8, academic press, new york 1967.

BALZER, R. M., RAND
Dataless Programming
fjcc 1967.

BARBOUR, KENNETH, R., AND RICHARD E. LITTLE
Liberation and the Easytran Approach to Reprogramming
honeywell computer j. 2.2 (summer 68), pp. 39-48.

BARTON, M. E., BELL TELEPHONE LABS
SWAP: The Macro Assembler
proc afips 1970, fjcc.

BELL, C. GORDON AND ALLEN NEWELL
Computer Structures: Readings and Examples
mcgraw-hill, new york 1971, p. 668.

BEMER, ROBERT W.
Program Transferability
proc afips 1969 sjcc, afips press, montvale, new jersey,
1969, pp. 605-612.

BENNET, RICHARD K.
The Design of Computer Languages and Software Systems:
A Basic Approach
computers and automation, 19.2 (feb 69), pp. 28-33.

BIRKHOFF, GARRETT AND SAUNDERS MACLANE
A Survey of Modern Algebra
the macmillan company, new york, 1953.

BROMBERG, HOWARD AND STANLEY M. NAFTALY
Cobol Standards (USASI)
data processing, vol. XIII, data processing management assoc.,
park ridge, ill. 1968.

BURROUGHS CORPORATION
B5500 Information Processing System Cobol
#PCN 102424-001, may 1969.

BURROUGHS CORPORATION
B5500 Information Processing System FORTRAN Compiler
reference manual AA945-756, burroughs corporation, detroit,
michigan, december 1968.

CHEATHAM, THOMAS E., JR., JEAN E. SAMMETT, THOMAS B. STEEL JR.
AND PETER WEGNER
Long Range Goals and Programming Languages
panel at 70 fjcc.

CONTROL DATA CORPORATION
6600 Computer Systems COBOL Reference Manual
september 1967, control data corporation, palo alto, california.

CONTROL DATA CORPORATION
CDC 6600 FORTRAN Reference Manual #60174900B
control data software documentation, st. paul, minnesota, 1967.

CONTROL DATA CORPORATION
6000 Computer Systems JOVIAL Compiler
june 1971, no. 17302500.

DELLERT, GEORGE T. JR.
A Use of Macros in Translation of Symbolic Assembly
Language of One Compiler to Another
cacm 8, 12, dec. 1965.

DEPARTMENT OF THE AIR FORCE, U.S.A.
Standard Computer Programming Language for Air Force
Command and Control Systems (JOVIAL (J3))
air force manual no. 100-24, department of the air force,
washington, d. c., 15 june 1967.

DICKSON, K. G.
Users Manaul AN/FSQ-7 JOVIAL (J3) Language
tm-32581001/00, system development corporation, december 1966.

DONOVAN, JOHN J. AND HENRY F. LEDGARD
A Formal System for the Specification of the Syntax and
Translation of Computer Languages.
fjcc 1967.

ELSPAS, BERNARD, MILTON GREEN AND CARL LEVITT
Software Reliability
computer 4.1 (jan/feb 71), pp. 21-27, ieee computer society.

ENGLANDER, WILLIAM R.
How Standard are Cobol Compilers?
business automation 17.6 (june 70), pp. -70.

ERICKSON, BEVERLY A.
A Program Disassembler
sp-1807, february 1965, system development corporation, santa
monica, california.

FALOR, KEN
Survey of Program Packages
modern data 3.3 (march 70), modern data 3.8 (august 70).

FALOR, KEN
The Case for Programming Standards
modern data 3.12 (december 71).

FERGUSON, DAVID E.
Evolution of the Meta-Assembly Program
communications of the acm, volume 9, number 3, march 1966.

FREDERICKS, D. S.
Across Machine Lines in COBOL
communications of the acm, vol. 8, #12, december 1965.

FRITZ, W. BARKLEY
Computer Change at Westinghouse Defense and Space Center
fjcc 1967.

FUJII, ATSUSHI
Software in Japan: Supported Growth
datamation 17.4 (feb 15, 71), pp. 26-29.

Gaines, R. STOCKTON
On the Translation of Machine Language Programs
comm acm 8.12 (dec 65), pp. 736-741.

GAVRILOV, M. A. AND A. D. ZAKREVSKII (EDS)
LYaPAS: A Programming Language for Logic and Coding Algorithms
academic press, new york, 69.

GODSEN, JOHN A.
Software Compatibility: What Was Promised, What We Have,
What We Need
proc afips 1968 fjcc.

GODSEN, JOHN A.
Software Compatibility
proc afips 1969 sjcc, afips press, montvale, new jersey, 1969,
pp. 605-162.

GORN, SAUL
Advanced Programming and the Aims of Standardization
comm acm 9.3 (march 66).

GRAHAM, MARVIN LOWELL AND PETER ZILAHY INGERMAN
An Assembly Language for Reprogramming
comm acm 8.12 (dec 65), pp. 769-773.

GRAY, MAX AND KEITH R. LONDON
Documentation Standards
brandon/system press, princeton, new jersey, 1969.

GRIES, DAVID
Compiler Construction for Digital Computers
john wiley and sons, new york, 1971.

GRUENBERGER, FRED
Problems and Priorities
datamation, march 1972.

GUSKIN, JACK R.
A Uni-Language for a General-Purpose Assembly Translator
System for Single Address Machines
rapporto n-3, bologna-viale risorgimento 2, settembre 1970.

HALSTEAD, MAURICE H.
Machine Independence and Third-Generation Computers
fjcc 67.

HALSTEAD, MAURICE H.
Using the Computer for Program Conversion
datamation 16.5 (may 70), pp. 125-129.

HARPER, WILLIAM L.
Documentation for Application Programming
modern data 2.9 (sept 69).

HICKS, HARRY T. JR.
ANSI Cobol
datamation 16.14 (nov 1, 70), pp. 32-36.

HOPPER, GRACE MURRAY
Standardization of High-Level Languages
proc afips 1969 sjcc, afips press, montvale, new jersey, 1969, pp. 605-612.

HUGHES AIRCRAFT COMPANY
H4118 Jovial Compiler
hughes aircraft company, fullerton, california, 1970.

IBM
System/360 Operating System Cobol Language
form c28-6516, november 1970, new york.

IBM
IBM System 360 Fortran IV Language
form c28-6515, ibm programming publications, new york, new york, 1971.

JONAS, RONALD W.
Generalized Translation of Programming Languages
proc afips 1967 fjcc, anaheim, california (nov 1967), pp. 570-80.

JONES, GILBERT E.
The Impact of Standards
computers and automation 18.5 (may 69), pp. 38-39.

KEESE, W. M. AND W. H. WATTENBURG
The Single Document Approach to Computer Program Documentation
case 22, technical memorandum tm-65-1031-4, 1965, bellcom, inc.

LEE, JOHN A. N.
The Anatomy of a Compiler
reinhold publishing, n. y., 1967.

LENIHAN, JOHN F.
Are the Scales Tipping Toward Reprogramming?
data management, february 1971, vol. 9, #2.

MCCRACKEN, DANIEL
A Guide to Cobol Programming
second edition, john wiley & sons publisher, 1970, new york.

MEALY, GEORGE H.
Another Look at Data
fjcc 1967.

MILLS, HARLAN D.
Syntax-Directed Documentation for PL 360
comm acm 13.4 (april 70), pp. 216-222.

MORENOFF, EDWARD
The Transferability of Computer Programs and the Data on Which They Operate
proc afips 1969 sjcc, afips press montvale, new jersey, 1969, pp. 605-612.

MULHOLLAND, K. A.
Software to Translate Telcomp Programs into KDF9 Algol
computer j. 12.3 (aug 69), pp. 221-224.

O'BRIEN, W. M.
Jovial Evaluation Project
data dynamics, inc., los angeles, california, report #esd-tr-68-452.

ORGASS, RICHARD J. AND WILLIAM M. WAITE
A Base for a Mobile Programming System
comm acm 12.9 (sept 69), pp. 507-510.

PERSTEIN, MILLARD H.
The Jovial (J3) Grammar and Lexicon
system development corporation, santa monica, california, 26 may 1966.

PHILLIPS, B. E.
The Importance of Data Processing Standards
computer bulletin 12,4 (aug 68), pp. 144-146.

POOLE, P. C. AND W. M. WAITE
Machine Independent Software in Second Symposium on Operating Systems Principles, Princeton University, Princeton, New Jersey
brandon/systems press, princeton, new jersey, october 1969.

RECTOR, DR. ROBERT W. AND DR. C. J. WALTER
The Fourth - Another Generation Gap?
modern data 2.3 (march 69).

REENSKAUG, TRYGVE
Some Notes on Portable Application Software
datamation 16.4 (april 70), pp. 104-106.

RICHARDS, MARTIN
BCPL: A Tool for Compiler Writing and System Programming
proc afips 1969 sjcc, pp. 557-566.

ROSEN, S.
Programming Systems and Languages
mcgraw-hill book co., new york, 1967, cr 10.1 (jan 69).

SABLE, JEROME D.
Transferability of Data and Programs Between Computer Systems
proc afips 1969 sjcc, afips press, montvale, new jersey, 1969, pp. 605-612.

SAMMET, JEAN E.
Programming Languages: History and Fundamentals
prentice-hall, englewood cliffs, new jersey, 1969.

SATTLEY, KIRK, ET AL.
On Program Transferability
massachusetts computer associates, ca-7011-2411, november 24, 1970, wakefield, massachusetts.

SCHAEFER, MARVIN
DBL: A Language for Converting Data Bases
datamation 16.6 (june 70), pp. 123-130.

SCIENTIFIC DATA SYSTEMS
SDS Fortran IV Reference Manual 900956A
scientific data systems, santa monica, california, april 1966.

SHAW, CHRISTOPHER J.
A Comparative Evaluation of Jovial and Fortran IV
automatic programming information no. 22, college of technology, brighton, england, august 1964.

SHAW, C. J.
An Outline for Describing and Evaluation Procedure-Oriented Programming Languages and Their Compilers
automatic programming information, october 1962, nos. 15-16, brighton, england.

SIEGEL, STAN
WATFOR...Speedy Fortran Debugger
datamation, november 15, 1971.

SIMMONS, JOHN W.
Fortran Loop Detecting Trace
software age, march 1970.

SIMPLIFIED MANAGEMENT SYSTEMS
Cobol Standards and User's Manual
internal document, november 1971, los angeles.

SPERRY RAND
Univac 1106/1108 Cobol
#up-7626, rev. 2, 1970.

VAUGHN, PETER H.
Can Cobol Cope?
datamation 16.10 (sept 1, 70), pp. 42-46.

WALSH, DOROTHY A.
A Guide for Software Documentation
mcgraw-hill inter-act, new york, new york, 1969.

WARD, JAMES A.
Program Transferability
proc afips 1969 sjcc, afips press, montvale, new jersey, 1969,
pp. 605-612.

00050#
Generalized Translation of Programming Languages

0710#
The Ecology of Change in a Major Computer Conversion

12734#
Standardization as a Prerequisite for Efficient Industry-Wide
Data Reduction

00063#
The Undesireability of Rigid Standards for Programming Languages

13723#
Standards for Computers and Information Processing
j. accountancy 124.3 (sept 1967), pp. 52-57.

19,080#
Data Base Task Group Report to the CODASYL Programming
Language Committee (oct 69 report)
acm, new york, 1969.

17173#
Standards in Information Processing
elektronische daten verarbeiting 10.9 (1968), pp. 458-459,
(German).

9175#
Standardization in the Field of Computers and Information
Processing

#Numbers above are Computing Reviews review number.

APPENDIX E

International Computer Systems, Inc. gratefully acknowledges the assistance of the following individuals in the preparation of this report.

Robert Abbott
Helen Buchanan
Andrew Edwards
Larry Rieder
Aaron Spinak

In addition, contributions to the material presented were received from:

Erwin Book
Richard Bougetz
Richard DeSantis
Sam DiCarlo
J. W. Feller
Judith Ford
Phil Hammond
Harry Keit
H. Fred Kreetch
Pat Langendorf
Warren Loper
Ken Manire
Nancy Mathis
Jim McNeely
Hisa Ogushi
Henry Shapiro
Bill Soss