stichting

mathematisch

centrum

$\sum$
MC

2e boerhaavestraat 49 amsterdam

AMS(MOS) subject classification scheme (1970): 68A30, 68A40, 68A74

ACM-Computing Review Categories: 4.20, 4.30

GUIDELINES FOR PROGRAM PORTABILITY [*]

Andrew S. Tanenbaum
Vakgroep Informatica
Wiskundig Seminarium
Vrije Universiteit
Amsterdam, The Netherlands

and

Paul Klint
Mathematisch Centrum
Amsterdam, The Netherlands

and

Wim Bohm
Mathematisch Centrum
Amsterdam, The Netherlands

ABSTRACT

The areas in which programs are most unlikely to be portable are discussed.   Attention is paid to programming languages, operating systems, file systems, i/o device characteristics, machine architecture and documentation.   Pitfalls are indicated and in some cases solutions are suggested.

KEY WORDS & PHRASES

Portability, program design, machine independence, machine architecture, operating system interface, documentation

---

[*] This report will be submitted for publication elsewhere.

1. INTRODUCTION

This document contains a number of problems that implementors of "portable" software should be aware of. The hope is that before unleashing a portable program on the world, the implementor will check to see that he has found satisfactory answers to all the difficulties mentioned below.

The problems are grouped under a variety of headings, but these are not entirely distinct since some problem areas overlap:

Programming Languages.
Real Numbers.
Files.
Physical Media.
Interactive Terminal I/O.
Operating System Problems.
Machine Architecture.
Documentation.
Miscellaneous.

Each point is described by a short phrase and an explanation or some examples below it.

We will use the following terminology throughout. The machine on which the program is already running is called the "host". The machine to which it is to be moved is called the "target". A "designer" produces a portable program on a host computer. An "installer" attempts to implement such a portable program on a target computer [1]. Portability is defined as follows (this definition is based on a definition given in Poole & Waite[2]):

Portability is a measure of the ease with which a program can be transferred from one environment to another; if the effort required to move the program is much less than that required to implement it initially, and the effort is small in absolute sense, then that program is highly portable.

Note that this definition does not exclude complete rewriting of small programs.

There are essentially two methods of writing programs intended to be moved to many machines: writing the program in a high level language, or writing it in a low level (assembly like) language for an abstract machine. In the former case it is tacitly assumed that a compiler for the high level language already exists on the target machine, while in the latter case the installer must write a processor for the abstract machine instructions to translate them to the target machine's instruction set (or equivalent). In most cases the low level language is so simple that the translator consists of a set of macro definitions for a macro processor or macro assembler. Most of the problems listed below

apply to both the high and low methods, however the section on Machine Architecture is most relevant to the abstract machine method, rather than the high level language method.

Some of the problems mentioned below do not occur or are not serious if one is trying to move a small program written in a high level language. However, if one is trying to move a large program or system (e.g. a data base management system) that makes heavy use of operating system calls, and consists of several programs linked together, and much data, the situation is very different.

These guidelines are intended to be used by designers and installers of portable software. Minimal attention has been paid to the problem of adapting existing (not portable) programs to other machines.

2. PROGRAMMING LANGUAGES

The specifications of most programming languages are either incomplete, ambiguous or both. As a consequence, each implementation of some programming language has to associate a meaning with language constructions which are left undefined by the standard language definition (if any). Hence undefined constructions have different meanings in various implementations of the same language. Many implementations tend to deviate from the standard language definition or add non standard language features. General advices are:
-    Do not use language extensions, use only standard language features.
-    Do not use "undefined" constructions, therefore
-    Read the standard language definition.
-    Use portability check options of your compiler (if available) or use a verifier (e.g PFORT [3]).

2.1 Dialects.
      Not every compiler accepts exactly the standard language, assuming there is a standard. Often there are restrictions, such as a requirement that all parameters in Algol 60 be fully specified. In other words, only a subset of the language may be implemented on the target machine. On the other hand, severe problems can arise when a superset of the language has been implemented on the host machine.

2.2 Length of Identifiers.
      Identifiers in Algol 60 and Pascal and some other languages may in principle be arbitrarily long. However, hardly any compiler allows this. Most of them have some restriction, and these differ from compiler to compiler.

2.3 Stropping.
The begin symbol in Algol like languages is printed in publications in bold face type. The computer representation is not standardized. The following are common representations: "BEGIN" "BEGIN 'BEGIN' 'BEGIN and .BEGIN. If different representations are used on the target and host machine, a conversion is needed.

2.4 Pragmats.
Some languages (e.g. Algol 68) allow the programmer to provide hints and advice and certain commands to the compiler. The syntax and semantics of these so-called pragmats are machine dependent.

2.5 Mapping of Types onto Machine Words.
A compiler writer for a computer with byte addressing and a 16 bit word can decide to use 16 bit integers or 32 bit integers (or some other length). Thus two compilers for the same language and same hardware may have different length integers, reals, etc. This problem is particularly critical in Fortran programs with many COMMON variables, since changing the storage sizes of variables can make previously correct COMMON declarations incorrect, semantically.

2.6 Separate Compilation.
Some compilers allow procedures to be compiled independently, and others do not. If a program has been developed using separate compilation and is later moved to a machine where that is not possible, the program may not compile because it is too large. No assumptions should be made about the order in which separately compiled procedures are tied together by a linkage editor.

2.7 Maximum Array Sizes.
Some compilers put limits on the maximum number of elements in an array, the maximum subscript value, the maximum number of arrays, or the maximum number of subscripts.

2.8 Packing.
If a program makes assumptions about how many characters fit in an integer there are likely to be problems when it is moved. Also if the host compiler packed the elements of boolean arrays into words with one element per bit the program may not fit on the target machine if one element occupies a whole word there.

2.9 Garbage Collection.
Some systems perform garbage collection and others do not. If the program needs it and it is not available, it probably will not run.

2.10 Runtime checking.
Most languages require a certain amount of runtime checking (e.g. array bounds, computed goto's, uninitialized variables, scope and

type checking). In most systems these checks can be turned off to gain execution speed. Cases are known, however, where programs ran "perfectly" without these checks, but gave errors when the checks were enabled.

3. REAL NUMBERS (FLOATING POINT)

There are so many different ways of representing reals, that it is very likely that the set of allowable numbers will differ between the host and target machines. On most machines, the representation of floating point numbers consists of:
1. Mantissa (explicit)
2. Exponent (explicit)
3. Radix (implicit)

The arithmetic of a given machine can be parameterized by ([4]):
-   Radix: base of the floating point number system.
-   Mantissa length: number of base-radix digits in the mantissa of a stored floating point number.
-   Relative precision: the smallest number x such that $1.0-x < 1.0 < 1.0+x$ where $1.0-x$ and $1.0+x$ are the stored values of the computed results.
-   Overflow: the largest number x such that both x and $-x$ belong to the system of real numbers.
-   Underflow: the smallest positive real number x such that both x and $-x$ are representable as elements of the system of real numbers.
-   Symmetric range: the largest real number x such that the arithmetic operations o are correctly performed for all elements a, b of the system of real numbers, provided that a, b and the exact mathematical result of a o b do not have an absolute value outside the range $[1/x,x]$.

To the extent possible, reals should be avoided. If the problem demands the usage of reals, it also induces a set of parameters for the arithmetic. It should be checked that the machine parameters can meet the demands of the problem parameters.

3.1 Range of Reals.
The target machine's floating point numbers may have a smaller range for its numbers than the host's. The PDP-11, for example, can not handle numbers larger than $10^{**}39$, whereas the CDC Cyber can handle numbers up to $10^{**}322$. A Cyber program using numbers between $10^{**}39$ and $10^{**}322$ will not work when moved to a PDP-11.

3.2 Precision of Reals.
The smallest positive real number is highly machine dependent. Any program that makes use of it, for example, by iterating until

something is less than epsilon (where epsilon equals the smallest positive real), will behave differently when transferred. In particular, some problems in numerical analysis have the property that a series converges until a certain point, and then begins to diverge. If the target machine has more precision than the host, the series may be computed to more terms, and give totally meaningless answers as a result. Thus more precision does not always yield more accurate answers. The designer of a portable program may not be aware of the precision, which his program requires. It is possible that the host machine provides too much, unwanted precision (e.g. all floating point operations are implicitly performed in double precision). In such a situation the portable software may require more precision than was anticipated by its designer.

3.3 Normalization.
Different machines have different rules for when normalization is performed. On some machines (e.g. Cyber) it is not performed automatically and it must be explicitly programmed. On other machines (e.g. PDP-11) it is not even possible to express unnormalized real numbers. This can give problems if the program needs to deal with unnormalized numbers (e.g. for real-to-integer conversion).

3.4 Comparison of Reals.
Some machines (e.g. Cyber) do not have an explicit instruction to compare 2 reals. This means that statements of the form IF X<Y THEN ... must use explicit subtractions, with the attendant problems of overflow that must be checked for explicitly.

3.5 Hidden Digits.
Extra digits that are stored in some parts of the machine but not all parts (e.g. registers vs. memory) can be a problem. For instance, tests for zero may depend on where the test is performed, and whether the extra digits are all zero or there can be a difference between computed values and stored results.

4. FILES

For the purpose of this discussion the following terminology will be used (see [5]). An elementary data item is a piece of information that is stored, retrieved or processed. A collection of one or more data items that describes some object is called a record. For handling convenience, records are usually grouped into logical units called files. The accessibility of the various records in a file is determined by its file organization (sequential, random, list). A file system provides primitive operations on files, such as naming, creating, deleting,

protecting and sharing. It may provide several file organizations and ways to structure a number of files in a database. Every system uses a somewhat or completely different file concept. To improve portability of file manipulations:
- Use sequential files.
- Use character files. Avoid binary files.
- Use a standard character code (ASCII, EBCDIC).
- Avoid read and write operations on the same file.

4.1 File Names.

Different operating systems have different conventions for naming files. The number of characters in a name, and which ones are allowed as first letter and as subsequent character differ. In UNIX, a file name may consist of almost any character sequence of up to 14 characters. For example, a carriage return is a valid one character file name, although one that virtually no other system will accept. Some systems have default file name extensions, which may cause trouble on machines that have different default extensions, or no extensions.

4.2 File Directories and Directory Operations.

Some operating systems make a distinction between local and permanent files. In some cases, a permanent file must be explicitly attached before it can be used (e.g. CDC SCOPE). On other systems (e.g. UNIX) this is not necessary. If a program needs to read a data file, the method by which the data file is accessed may cause trouble when moved. Some systems require an explicit "rewind" operation to reset a file pointer before a file can be read. On other systems, the file pointer is automatically positioned at the beginning of the file, when the file is opened. Likewise, on some systems a file may have to be cataloged in order to allow it to survive longer than the current job or session. Other file manipulation operations such as opening files, renaming, etc. may be difficult to port.

4.3 File protection.

There are several schemes for file protection in use. Some involve passwords (e.g. CDC SCOPE), some involve explicit lists of who may access what and how (e.g. MULTICS), and some may grant permission based on some attributes of the requester, such as a distinction between the owner, members of the owners group, and everybody else (e.g. UNIX). If a program creates files or if files are to be shared the protection mechanism will have to be used.

4.4 Restrictions on Files.

An operating system may distinguish between different classes of files such as text files vs. binary files (e.g CDC SCOPE). Other systems may place minimum or maximum sizes on files, or require files to consist of a certain number of fixed size blocks.

4.5   Random Access.
      Some operating systems do not allow random access files (especial-
      ly simple minicomputer systems) and those that do may provide for
      it in a variety of ways. Common methods are addressing information
      by its position within a file, or addressing information by having
      the system search for a record containing a user specified key.

4.6   Special Files.
      Some systems may treat certain files (e.g. INPUT, OUTPUT) dif-
      ferently than normal files (e.g. you cannot rewind INPUT).

4.7   Record Length.
      There are often minimum and maximum record lengths that can be
      handled, and these may differ from source files to binary files
      when such a distinction is made.

4.8   Reproducibility of Data Written to a File.
      On some systems data written on a file may not come back exactly
      as it went out due to changes inflicted by the operating system.
      Certain characters may be systematically replaced (e.g. expansion
      of tab characters). Sometimes lines are filled out with spaces.

4.9   Blocking of Records.
      Different systems have different requirements concerning how short
      records are collected together into longer ones. In particular,
      the degree of user transparency may differ.

4.10 Maximum Open Files.
      Most systems restrict the users to a maximum of N open files. If N
      on the target machine is too small, the program will not work.

4.11 End of File Conventions.
      End of File is signalled in a variety of ways. Sometimes the end
      of file indication is returned along with the last record. Some-
      times it is returned only on the read subsequent to the last
      record.   Sometimes repeated attempts to read past end of file will
      continue to give end of file indications.

4.12 Multifile Tapes.
      The relation between a tape file and a copy of that file on disk
      may be system dependent, especially when the tape contains several
      files.

4.13 Automatic File Opening
      Some systems automatically open a file the first time it is refer-
      enced.

5. PHYSICAL MEDIA

The designer of a portable program is confronted with physical media in two ways:

- The distribution of the program.
  For increasing program size, the following media can be used:
  1 Listing (if the program is not larger than one page)
  2 Cards or paper tape (Several thousands of lines)
  3 Magnetic tape (larger programs)

  A character set, suiting the medium must be chosen. For listings, the character set will not give rise to problems. For cards, the 64 ASCII subset or the 64 BCD character set are reasonable candidates. For paper tapes and magnetic tapes 128 ASCII can be used. For further details see [6].
- The I/O carried out on the target machine.
  On many systems I/O operations are device dependent. But even if the devices can be handled uniformly, they still have their own physical limitations.

5.1 Line and Page Sizes.
The number of characters on a print line is system dependent. Sizes of 80, 128, ... characters per line have been used. Also the number of lines per page will be different on a system that provides automatic page headings and one that does not.

5.2 Carriage Control.
At least two radically different schemes for carriage control are in use. One system uses column 1 to indicate page eject, double space, etc. The other system depends on explicit control characters such as form feed and vertical tab. The effect of carriage control characters may also depend on the "carriage control tape" used on the printer.

5.3 Columns 73-80.
It is common practice to use columns 73-80 of card input for program identification. These columns may or may not be allowed for programs or data by the operating system.

5.4 Binary Cards.
Binary cards may be expected in a certain manufacturer defined format, with sequence numbers and checksums in certain columns. It may be very difficult, if not impossible, to read a binary card not in this format.

5.5 Restrictions on Input.
Certain combinations of characters may cause trouble when punched on data cards. For example, // in columns 1 and 2 may be intercepted by the operating system on the IBM 370. Likewise, :: may cause trouble under CDC SCOPE if punched in columns 9 and 10. This

problem is particularly annoying when trying to read in programs which allow free form input, and have // and :: as valid operators.

5.6 Card Size.
Cards with 51, 80, and 96 columns are in use. There is a possibility that the target machine will have the wrong kind of card reader.

5.7 Magnetic Tape Width.
There are two common tape widths: 7 track and 9 track. They are completely incompatible.

5.8 Magnetic Tape Density.
A variety of densities are in common use (e.g. 200, 556, 800, 1600 bpi). There is the possibility that the target machine will not be able to read the tape at all.

5.9 Parity.
Both magnetic tape and paper tape have parity, and there are four standards, even, odd, parity bit always 0 and parity bit always 1.

5.10 Magnetic Tape Labels and Formats.
Many operating systems refuse to read a tape that does not have labels and formats that it recognizes.

5.11 Magnetic Tape Record Structure.
Tapes often use file marks, trailers and other system dependent markings.

6. INTERACTIVE TERMINAL I/O

Interactive terminal I/O presents problems which are related to the points mentioned in the previous section. The problems in this area are caused by:
- different terminal characteristics, such as screen size, cursor and screen addressing, character set, special function keys, full and half duplex communication etc.
- interaction between terminal and program, such as escape and interrupt conventions, inspection of terminal status, I/O is done either character-wise or line-wise, etc.

6.1 Terminal Access.
In some systems the terminal is accessed as a file; in others it is accessed in a way distinct from files. If the terminal is seen as a file, it may nevertheless have properties very different than normal files, such as how it handles rewinds, etc.

6.2 Buffering.
Batch systems often collect file output in secret buffers, and actually perform I/O only when the buffer is full. This strategy can cause great difficulty with interactive users, since the program generally assumes that the user can see a given line as soon as it has been printed. If the user has not yet seen it, he can hardly be expected to take action based on it. To get around this problem, it is sometimes necessary to add special "buffer flushing" commands to a program. These commands are rarely portable.

6.3 End of Line Conventions.
The ASCII characters CR (carriage return) and LF (line Feed) are used in different ways to denote end of line. Four possibilities exist: CR, LF, CR+LF, LF+CR. All four are in use, which complicates both input and output. On top of that, some systems accept line feeds and convert them to CR+LF or LF+CR. These problems arise not only in interactive terminal usage.

6.4 Tabs.
Tabs are peculiar characters in that their semantics depend on their context. If the system automatically converts tabs to the appropriate number of spaces upon input at a low level, the effect of erasing a tab may be unpredictable. If tabs are not converted to spaces, the effect of outputting a file containing tabs to a device that does not recognize them (e.g. a line printer) may be unpredictable. Also, the output may depend upon the exact position of the tab stops on the terminal, which can cause difficulty if they can be reset by the user.

6.5 Erase Characters.
All time sharing systems provide some way for users to correct typing errors. Often this consists of a character to erase one character and another character to erase the current line. These functions may or may not be performed by a part of the operating system. If the system does not work with internal line buffers, but rather passes characters to the user program as it receives them (to allow cueing etc.), then user programs must be prepared to handle intraline editing. Also, some allow the users to define their own erase characters; programs should not assume this is possible.

6.6 Null, Delete, Backspace and Other Control Characters.
These characters may or may not be thrown away or treated in some special way by the operating system.

6.7 Fill.
Lines typed on interactive terminals are not restricted to any fixed size. The operating system may or may not pad out short lines to some fixed length (e.g. 72 or 80 characters). This will affect programs that do not expect (or do expect) trailing blanks

on input lines.

6.8 Partial Lines.
It may or may not be possible for a program to type some output, wait for input from the user, and then continue typing on the same line.

6.9 Lower case to Upper Case Mapping.
If the target machine uses only upper case letters internally, the difference between "name" and "NAME" will be lost even though the program is correctly read in. For programs that use both, there will be trouble.

6.10 Incompatable File Operations.
Some programming languages provide file operations that are incompatable with interactive terminal I/O (e.g. to read a line in PAS-CAL, the first character of the next line must already be present).

7. OPERATING SYSTEM PROBLEMS

An operating system creates an environment for program execution, it may hide certain features of the underlying machine architecture and it may add other features. Hence, porting problems may even arise when the hardware is identical but the operating systems differ. An operating system manifests itself in:
- Job control language
- Job structure
- System calls
- Resource management
- Default options

All operating systems impose restrictions on program size, file size and the like, which the installer may have to face. Since no two operating systems are alike, the designer of a portable program should:
- only use the simplest operating system facilities
- completely document the facilities used, and indicate all error situations
- use no defaults (they differ from installation to installation)
- make the operating system interface as adaptible as possible

7.1 Job Control Language.
No two manufacturers have even vaguely similar JCL´s. If the system to be transported consists of several programs with some JCL commands to put the pieces together, there will almost assuredly be some changes needed.

7.2   System Log.
Some operating systems produce a log (dayfile on CDC machines)
which may or may not be readable or writable by a program in exe-
cution. The formats and information provided vary widely from sys-
tem to system.

7.3   Overlays.
Large programs are often handled by breaking them up into over-
lays, but the overlay structure may be different on the host and
target machines. Some systems regard overlays as a linear chain
of successively loaded modules, while others envision the overlays
built up as a tree. Communication between overlays is done in
different ways as well.

7.4   Job Dependency.
There is sometimes a way to insure that a collection of jobs sub-
mitted to a batch system simultaneously will be run in a specific
order. However, this possibility is not always present.

7.5   Operator Communication.
A program may have to communicate with the system operator for
some reason (e.g to request mounting a special disk pack, or to
put green paper in the printer). The possibilities and conventions
are strongly system dependent.

7.6   Checkpointing.
It is common practice to have very long jobs periodically save
their status on disk or tape so that a system crash does not force
rerunning the entire job. The methods of generating and restarting
checkpointed jobs are highly system dependent.

7.7   Initialization of Memory.
Some operating systems or compilers explicitly initialize all of
memory to 0 or -0 or some other value before giving a user program
control. It is dangerous to rely on any such initialization.

7.8   Execution Time, Memory, Output, Tapes.
If no limits are specified explicitly, most operating systems give
each job a maximum time limit, and perhaps a maximum memory allo-
cation, a maximum output page count, and a maximum number of tapes
that can be mounted simultaneously. A program that worked on the
host machine may not work on a target machine whose default limits
are more stringent.

7.9   File Protect Ring
Magnetic tapes have a plastic ring on the back side which can be
removed to prevent the tape from accidently being overwritten. At
some installations the default is ring in and at others it is ring
out, and at still others the tape is simply mounted the way it
comes out of the tape rack, with or without ring.

7.10 I/O Buffer Space.
If a program is moved to a target machine on which the default option for buffer space is larger than on the host, the amount of memory remaining for programs may be too small, and the program may not fit.

7.11 Availability of I/O Devices.
If the program requires n tapes and the target machine has only n-1, there will be difficulty.

7.12 Memory Allocation.
On some computers, the user program must explicit request and release memory via system calls. If the program was developed on a system where this is not necessary, changes may have to be made on computers where it is necessary. The problem is particularly severe for jobs that consist of a number of programs with dynamically changing memory requirements.

7.13 Resource Requests.
To avoid deadlocks, some systems require users to specify in advance all the resources they will need during the entire run. Others are less stringent. The trouble comes when moving from a system in which a certain resource need not be requested at all, to a system where it must be requested explicitly.

7.14 Console Switches.
If the program assumes that the operator can communicate with the program via the console switches, and the target machine has no switches or they are addressed by a different mechanism, problems can be expected.

7.15 Facilities used.
Different operating systems provide different facilities to their users. If a feature does not exist on the target machine, this will cause trouble. Typical examples are:
-    a system call to provide status about a file (length, mode, creation date, etc.)
-    hooks for making measurements
-    the possibility to execute a program from another program
-    interprocess communication
-    dump/restart facilities

7.16 Run Time Error Handling.
The ways that overflow, underflow and stack overflow are handled are dependent on the operating system. In particular, if the portable program assumes that there is a way to get control back after a trap and there is not, the program may not work. Note that exception handling is sometimes used to implement run time checks.

7.17 Environment information, Time and Date.
Whether and how system and environment information can be inspect-
ed and/or modified is extremely system dependent. Do not use these
facilities, unless the information is badly needed.   The formats
which systems use to return the time and date (assuming they re-
turn the time and date) are not standard. Furthermore, some sys-
tems have a time of day clock, others have an interval timer, and
some have both. Time may also be measured in clock pulses, which
depend on the line frequency (60 Hz. in the United States, 50 Hz.
"elsewhere").

7.18 Effects of Spooling.
If tapes are spooled to disk before being read, the file structure
on the tape will have to be converted to disk format, which may be
incompatible with tape format. As a result, information about the
hierarchical structure of the records may be lost. If the host
system did not spool tapes, these problems may only arise when the
program is moved. Furthermore, if large quantities of data must
be spooled, disk overflow may occur on the spooled system, but not
on the unspooled on.

8. MACHINE ARCHITECTURE

The difference in machine architecture between the host and target
machine is a main cause of portability problems. It is not uncommon
that the architecture of the host machine influences the program to be
ported.   This influence may either be explicit (word length) or be im-
plicit (the algorithms that are chosen). The problems in this area are
much more severe for the case of low level transport via abstract
machines, than for high level language programs.

8.1 Word length.
Different machines have different word lengths, which usually
means that the largest integer available is machine dependent. A
program whose integer variables can take on values above 2**N will
usually cause problems on a machine whose word length is N bits.

8.2 Byte vs. Word Addressing.
Programs should not have any implicit assumptions about consecu-
tive words having consecutive addresses, since a word may consist
of 2, 4 or some other number of consecutively numbered bytes.
Furthermore, on some byte addressable machines, (e.g. IBM 370) a
word is addressed by its leftmost byte, whereas on other machines
(e.g. PDP-11) it is addressed by its rightmost byte. On some byte
addressable machines, certain operations require their operands to
be aligned to certain byte or word boundaries.

8.3 Data Sizes.
The program should not make any assumptions about how many storage
units are occupied by integers, reals, pointers, etc. The first
version of the so called portable Pascal P compiler assumed that
integers, reals, pointers, and characters all occupied the same
number of storage units. On machines with very long word lengths
(e.g. the Cyber) this may be true, but on byte addressable
machines it is not true.

8.4 Character Code.
Different machines have character codes requiring different
numbers of bits. Six, seven and eight bit codes are all common.
Among other problems, characters available on the host machine may
not exist on the target machine. Also, the collating sequence may
be different. If a portable program makes the assumption that it
can check to see if a character it has just read in is a letter by
seeing if its code is in between that of "A" and "Z", it may not
work on a machine whose letters are not both consecutive and in
ascending sequence.

8.5 Special Bit Patterns.
Many machines have bit patterns with peculiar meanings. On one's
complement machines -0 is a problem. On two's complement machines
1000...000 is a problem since it is a negative number that is its
own complement i.e. x = -x. The Cybers have special bit patterns
for infinite and indefinite, with their own special algebra. On
the 360 some addresses are invalid, because they adddress out of
memory. These addresses are used by the ALGOL-W compiler for run
time checks. When the ALGOL-W compiler was moved to a 370, these
checks did not work anymore because on the 370 every address is a
valid one (virtual memory).

8.6 Tag Bits.
On some machines (e.g. Burroughs B6700) the hardware can tell
whether a given word contains an integer or a real by examining
its tag bits. A Fortran program that equivalences integers and
reals and then tries to use a real as though it were an integer
will have problems here. To make this point clearer, consider a
Fortran program in which a certain word can be accessed as the
real X or the integer I. The statement I=I/2 will perform an in-
teger division (i.e. a right shift of 1 bit) on I, even though it
may contain a real. When accessed as X, the new value may be ap-
proximately the square root of the old one, since the exponent has
been halved. Although this trick may be useful to give starting
values for iterative square root routines on some machines it will
not work on a tagged target machine.

8.7 Stack Instructions.
If a portable program is written using stack instructions, these
can usually be simulated on the target machine, except when the

target machine itself has incompatible stack instructions (e.g. PDP-11 stacks grow from high to low addresses). If the portable program breaks up its stack operations into individual operations to advance the stack pointer, check for overflow, and copy the data, etc., the person implementing the program on the target machine may not be able to recognize where stack operations are being performed, and may have to simulate in software what his own hardware is perfectly capable of doing directly. Problems may arise when objects of different type are placed on the same stack. On some machines (e.g. Burroughs B6700) the amount of stack space required by objects of different type is taken into account by the hardware. On other machines, the programmer has to be aware of the stack requirements of various objects (e.g. double length integer on PDP-11). A comparable problem occurs when an abstract machine is implemented via a high level language. Someone implementing an abstract machine in, say ALGOL 60, is really in trouble if objects of type integer and real should fit in his ´integer´ ´array´ stack.

8.8 Array Instructions.
If a portable program makes unwarranted assumptions about how array elements are stored and accessed, difficulties may arise on target machines with special array access instructions. For example, if the portable program makes assumptions about whether matrices are stored by rows or by columns, this may force an organization incompatible with how the target machine´s array instructions work. Again, the implementer may have to simulate in software operations that his hardware can perform much faster.

8.9 Operations on Different Datatypes.
Different target machine instructions may be required to perform "similar" operations on objects of different datatypes (e.g. integer, real or pointer comparision operations). The datatype of the operands of all operations should be made as explicit as possible (or desirable). Pointers in different address spaces should be considered as being of different type (see below).

8.10 Separate Instruction and Data Spaces.
Some computers (e.g. PDP-11/45) have distinct address spaces for programs and data. It is not possible for an ordinary (unprivileged) program to read the instruction space. This can cause havoc with portable programs whose calling sequence places the parameters (or pointers to them) in the words following the procedure call instruction because the called procedure on such a machine will have no way to get its parameters.

8.11 Multiple Data Spaces.
Some machines have a segmented virtual memory which presents itself to the user in the form of multiple independent address spaces, each beginning at address 0. If a portable program assumes

that there is a single linear address space whose words are num-
bered consecutively, it may be difficult or impossible to use most
of the available address space. This problem is illustrated by the
portable Pascal P compiler, which uses a stack (starting at ad-
dress 0 and growing upward) and a heap (starting at some high ad-
dress and growing downward). If stack pointer > heap pointer, an
overflow is assumed and the job is aborted. This model of storage
is deeply imbedded in the entire compiler design, making it impos-
sible for the implementer with two data segments at his disposal
to put the stack in one and the heap in the other.

8.12 Parallel Processing.
A program that uses parallel processing primitives (e.g. in Algol
68 or PL/I) but tacitly assumes that there is only one real CPU
that is multiplexed among the tasks may run into race conditions
when moved to a target machine with multiple CPU's. The order of
expression evaluation may depend on the number of multiple CPU's.
A portable program should not make assumptions about evaluation
order.

8.13 Total Address Space.
For large programs, the amount of address space may be a problem,
especially when moving programs to minicomputers whose address
space is rarely more than $2**16 = 65536$ bytes.

8.14 Device Registers.
On some computers (e.g. PDP-11, TI 990) I/O devices are addressed
as the top locations of memory. In addition to the obvious prob-
lems of converting to machines with explicit I/O instructions,
these I/O device registers reduce the available address space. On
the TI 990, the device registers take up 1K of the address space,
while on the PDP-11 they take up 4K. A program that filled up the
entire usable address space on the TI 990 will not fit into the
PDP-11's available address space.

9. DOCUMENTATION

No matter how "portable" a specific program may be, without a
sound documentation it will have a very low change to be used by others
then the designer. Good documentation has the following properties:
- It is complete but manageable. All information about the program
  should be made explicit.
- Debugging aids used by the designer may be valuable for the in-
  staller and should therefore be completely documented too.
- It contains a recipe for the installation of the program.

- It indicates all places where changes may be required. The meaning of all machine dependent parameters should be pointed out.
- A log of a successful run of all (portable) test programs is included.
- Experience of other installers (if any) and known installation problems are described.
- Hints are given for the optimization of the program, by providing run time statistics and the like.

9.1 Nomenclature.
The program may work perfectly on the target machine, except that its new owner may not understand what it does or how to make it work. One problem is the lack of standard nomenclature, e.g. "block" means something different on every system.

9.2 Lack of documentation.
If there is incomplete documentation, ambiguous documentation or no documentation at all, the installer or user of the portable program will surely have trouble.

9.3 Failure to Indicate Places Where Changes are Needed.
It is fairly common that the writer of a portable program realizes that a certain statement or parameter must be changed for each new machine (e.g. a statement like WORDLENGTH:=32). These statements should be clearly marked.

9.4 Failure to Parameterize machine dependent features.
It is fairly common that a designer of portable software considers some machine dependent property of the host machine as an universal truth, e.g. an address increment one is silently assumed, which is obviously incorrect on some target machines.

9.5 Failure to Provide Printed Documentation.
It is common practice to distribute the documentation on the same tape as the program. However, if the installer cannot figure out how to read the tape, he cannot extract the documentation from it telling him how to do so. PLAP is a text processor distributed by the portable Bill Waite himself. The program is accompanied by a very clearly written documentation. The only problem is that this documentation is written in the format of PLAP. So to get a readable documentation, the program has to run already on the target machine.

9.6 Lack of Source Code.
Distributing object programs is nearly hopeless. Invariably differences in I/O configuration, local operating system patches, and other installation peculiarities will make it necessary for the installer to to recompile or assemble the program. In addition to the source code itself, there should be documentation describing how to recompile the program, and how to install it.

9.7   Portable Test Programs.
The distributer of a portable program should provide portable test programs so the installer can tell whether he has in fact got the program working properly.

9.8   Maintenance.
A portable program is supposed to be delivered instead of abandoned on the target machine. The first version of a portable program may contain errors or there are other reasons for changing a first version. An (also portable) editor to get an updated version of the program may be of great help.

10. MISCELLANEOUS

10.1 Libraries.
Procedure libraries may be different from installation to installation, even if the machine and operating system are the same. For example, both systems may have a procedure RANDOM, but with different calling sequences, different seed values or different properties. The search order in libraries may be important.

10.2 Reproducibility of Random Numbers.
Many computers use a Markov algorithm for generating random numbers. This means that it is possible to reset the sequence. If a program counts on this fact, it will not work on a machine where random numbers are generated by amplifying and digitizing the quantum noise (Johnson noise) in a resistor.

10.3 Bootstrapping Procedures.
Given a bare machine one has to get started somehow. The procedure for getting on the air differs from machine to machine. This may cause trouble. For example, a machine may insist that the first record on a magnetic tape contains a bootstrap program written in its own machine language. It may be necessary to port a loader along with the program.

10.4 Data Bases.
Moving programs is totally trivial compared to moving large data bases.

10.5 Optimality is not Portable.
A program may be carefully tuned to the peculiar characteristics of a given machine. Even if it works on other machines, it may be so slow as to be useless. Note that optimality also depends on the accounting procedures used by the operating system. This means that an optimal program can become suboptimal when the ac-

counting procedures of the operating system are changed. On the other hand, portability does not imply inefficiency.

10.6 Adaptability.
Sometimes portability is not the only thing that is desired. A compiler running on the X machine when moved to the Y machine will continue to produce code for the X machine. What is actually desired is a different program, one that produces code for the Y machine. Both portability and adaptability are needed here.

10.7 Bugs in Target Machine Software.
Portable programs are either large or exercise system programs of the target machine (assemblers, loaders, editors) in a way which differs from the "normal" usage of such programs. It is not uncommon that during the installation of a portable program, (until that moment unknown) bugs are detected in the software of the target machine.

11. ACKNOWLEDGEMENTS

These guidelines are a result of discussions in the "Working Group on Program Portability", organized by the Mathematical Centre in the period 1976/1977. Many individuals made contributions to a draft version of this list, which was compiled by Andy Tanenbaum.

12. REFERENCES

[1] Druseikis, F. C. The design of transportable interpreters, Doctoral dissertation, University of Arizona, S4D49
[2] Poole, P.C. & Waite, W.M., Portability and Adaptability, in Software engineering, an advanced course, Lecture notes in computer science, Springer Verlag, 1975
[3] Ryder, B. G., The PFORT verifier, Software - Practice and Experience, Vol. 4, p359-378 (1974).
[4] Parameters for transportable numerical software, IFIP working group on Numerical Software (WG 2.5), 1976
[5] Roberts, C. R. File Organization Techniques, in Adv. in Computers, 12 (1972)
[6] Waite, W. M. Hints on Distributing Portable Software, Software - Practice and Experience, Vol. 5, 295-308 (1975)