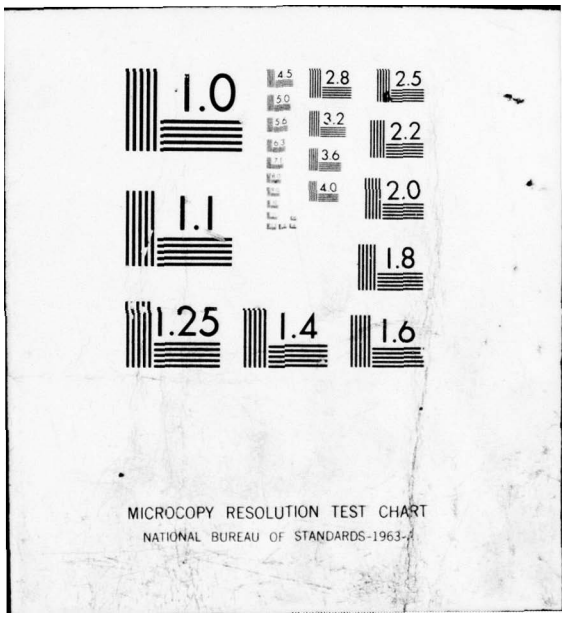
AD-A049 737 SOFTECH INC WALTHAM MASS F/G 9/2
COMMUNICATIONS HIGH-ORDER LANGUAGE INVESTIGATION. VOLUME I.(U)
OCT 77 R S EANES, J B GOODENOUGH, J R KELLY F30602-76-C-0306
UNCLASSIFIED RADC-TR-77-341-VOL-1 NL

1 OF 3
AD
A049737


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

AD A049737

(2)
SC

RADC-TR-77-341, Volume I (of two)
Interim Report
October 1977

COMMUNICATIONS HIGH-ORDER LANGUAGE INVESTIGATION

SofTech, Incorporated

AD NO. ____
DDC FILE COPY

Approved for public release; distribution unlimited.

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, New York 13441

DDC
RECEIVED
FEB 9 1978
B

COMMUNICATIONS HIGH-ORDER LANGUAGE INVESTIGATION, VOL I

This report has been reviewed by the RADC Information Office (OI) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

Because of the size of this report, it has been divided into 2 volumes.
Volume I contains Parts I and II and Appendices A and B.
Volume II contains Appendix C.

RADC-TR-77-341, Volume I has been reviewed and is approved for publication.

APPROVED: Douglas White

DOUGLAS WHITE
Project Engineer

APPROVED: Alan R Barnum

ALAN R. BARNUM
Assistant Chief
Information Sciences Division

FOR THE COMMANDER: John P. Huss

JOHN P. HUSS
Acting Chief, Plans Office

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (ISIS) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return this copy. Retain or destroy.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: RADC-TR-77-341, Vol I (of two)	2. GOVT ACCESSION NO. VOL-1
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle): COMMUNICATIONS HIGH-ORDER LANGUAGE INVESTIGATION, Volume I.	5. TYPE OF REPORT & PERIOD COVERED: Interim Report. 1 June 1976 - 30 June 1977.
	6. PERFORMING ORG. REPORT NUMBER: N/A
7. AUTHOR(s): R.S. Eanes, J.B. Goodenough, J.R. Kelly, D.K. Klotzbach, A.J. Nickelson, L.H. Shafer, L.M. Willoughby	8. CONTRACT OR GRANT NUMBER(s): F30602-76-C-0306
9. PERFORMING ORGANIZATION NAME AND ADDRESS: SofTech, Inc. 460 Totten Pond Road Waltham MA 02154	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 33125F 20220401
11. CONTROLLING OFFICE NAME AND ADDRESS: Rome Air Development Center (ISIS) Griffiss AFB NY 13441	12. REPORT DATE: October 1977
	13. NUMBER OF PAGES: 223
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Rome Air Development Center (ISIS) Griffiss AFB NY 13441 (12) 233p.	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE: N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

DDC RECEIVED FEB 9 1978 B

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer: Douglas White (ISIS)
Because of the size of this report, it has been divided into 2 volumes. Volume I contains Parts I and II and Appendices A and B. Vol II contains Appendix C.

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Programming Language, High Order Language, High Level Language, Communications, Requirements, Evaluation.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report contains data and conclusions that were used in selecting JOVIAL J73I as a base language for modifications that would result in a language suitable for programming communications applications. Volume I of this report contains a description of the process used to develop hypotheses on what features are required of a language suitable for programming communications applications. This volume also contains rationale for selection of JOVIAL/J73I, efficiency requirements and results of a questionnaire concerning high-

(over)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

i 405932 4B

cont

order languages and communications.

Volume II of this report contains an extensive features list showing the presence or absence of specific features from each of the languages evaluated.

ACCESSION for
NTIS White Section ☑
DDC Buff Section ☐
UNANNOUNCED ☐
JUSTIFICATION
BY
DISTRIBUTION/AVAILABILITY CODES
Dist. AVAIL. and/or SPECIAL
A

TABLE OF CONTENTS

	Page
INTRODUCTION	ii
PART I - HYPOTHESES ON COMMUNICATIONS HOL REQUIREMENTS	I-1
SECTION 1 - DEVELOPMENT OF HYPOTHESES	I-2
SECTION 2 - COMMUNICATION LANGUAGE DESIGN FACTORS	I-3
SECTION 3 - COMMUNICATIONS HOL FEATURES	I-8
PART II - RATIONALE FOR BASE LANGUAGE SELECTION AND PROPOSAL FOR MODIFICATIONS	II-1
SECTION 1 - LANGUAGE FEATURE AREAS	II-5
SECTION 2 - USABILITY, UNIFORMITY, AND RELIABILITY	II-12
SECTION 3 - MODIFICATIONS TO J73	II-15
SECTION 4 - SUMMARIES OF DIFFERENCE ANALYSES	II-24
APPENDIX A - REQUIREMENTS FOR COMPILER EFFICIENCY IN COMMUNICATIONS PROGRAMMING	A-1
SECTION 1 - INTRODUCTION	A-2
SECTION 2 - LANGUAGE LEVEL EFFICIENCY	A-4
SECTION 3 - MACHINE LEVEL EFFICIENCY	A-8
APPENDIX B - RESULTS OF COMMUNICATIONS HIGHER ORDER LANGUAGE QUESTIONNAIRE	B-1
SECTION 1 - INTRODUCTION	B-1
SECTION 2 - NUMBER OF RESPONDENTS	B-5
SECTION 3 - COMMENTS	B-7
SECTION 4 - RESPONSES TO QUESTIONS	B-9
SECTION 5 - WHAT IS TOUGH AND IMPORTANT ABOUT COMMUNICATIONS PROGRAMS	B-92
SECTION 6 - WHERE DO THE BUGS COME FROM	B-97
SECTION 7 - WHAT THEY EXPECT FROM A HOL	B-99
SECTION 8 - SPECIAL FACILITIES	B-104

TABLE OF CONTENTS (Continued)

	Page
ANNEX I - RAW DATA RESPONSES	B-I-1
ANNEX II - COMMUNICATIONS HIGHER ORDER LANGUAGE QUESTIONNAIRE (Unscrambled)	B-II-1
APPENDIX C - LANGUAGE FEATURES LIST	C-1

INTRODUCTION

A year long study of higher order languages (HOL's) has been performed to determine an HOL for use in the Communications Processing System (CPS). This study involved two aspects:

- definition of needed HOL capabilities;
- evaluation of five languages to determine their suitability for programming the CPS, including CS4, JOVIAL (both J3 and J73), PASCAL and PL/I.

The definition of HOL capabilities involved collaboration with experienced communications system designers and programmers, as well as extensive on-site interviews of Air Force communications experts. Hypotheses were formulated concerning desired features, and subsequently verified by programming selected communications algorithms in the candidate HOL's. The evaluation of the candidate languages involved describing each of them in terms of an exhaustive list of HOL features, and comparing each against an ideal language described in terms of the same list based on the verified hypotheses about communications programming.

The difference analyses were used to evaluate suitability of each of the languages for use in communications programming. The evaluations dealt with identifying which language:

- has the greatest number of required capabilities;
- most efficiently accomplishes communications functions;
- best promotes development of reliable software;
- provides the most consistent syntax or semantics when modified;
- is the most cost effective to modify and use in the near and long term.

The conclusions reached were that the extreme range of communications programming requirements makes all of the existing languages unsuitable without modification. One language, J3, was determined to be unsuitable both without modification and as a base language. All the languages required considerable modification, and the determination of which is most cost effective to use depends on balancing the near and long term. J73 was determined to be the most efficient and immediately usable. It also very clearly lacks important capabilities related to use over the life time of a communications system, in terms of possible savings in:

- reduced debugging and testing time, and
- reduced software maintenance costs.

The modification strategy was designed to incorporate language features that makes programs easier to understand (so maintenance programmers can familiarize themselves with the programs more easily), easier to modify, and that help in detecting errors introduced when programs are modified.

The formal study of programming requirements for communications involved interaction between experienced designers and implementers of HOL's and experienced designers and implementers of communications systems, in the combination of the HOL expertise of SofTech with the communications software experience of Data Systems Analysts. A joint analysis of the needs of communications programming was performed, to determine the set of hypotheses about communications programming needs. Then a set of program segments was selected as representative of communications software problems. These were reprogrammed in each of the candidate HOL's, and the experience used to confirm or deny the initial hypotheses.

An extensive list of high order language features, originally produced for the Army Electronics Command at Fort Monmouth, N. J., was used to describe the candidate languages. This list contains descriptions of over 2200 features, characterizing a wide variety of system programming languages, and analyzes each construct in terms of a complete set of evaluation factors and subfactors. The feature list ignores trivial syntactic differences among similar constructs in different languages, and hence permits an accurate evaluation of the essential similarities and differences among languages. New language features for communications programming were incorporated into the Fort Monmoth list to allow the candidate languages to be analyzed uniformly over all areas.

The results of the application analysis and reprogramming effort were used to verify the initial hypotheses and to produce an hypothesized ideal communications language. This was entered into the features list, as a way of defining the hypothesized language and as a way of facilitating comparing the hypothesized language with the candidates. Listings of the differences between the hypothesized language and each candidate were produced, each showing desired features present in each language, desired features not present (additions), and features present but not desired (deletions). The differences were analyzed and summarized, and are discussed in Part II.

Part I discusses the hypothesized communications language features as determined from the applications analysis and reprogramming study. Hypothesized needs for the following data types and control structures were verified:

- data types
  - integer
  - real
  - Boolean
  - character
  - status
  - procedure
  - pointer
  - bit
  - label

- data structures
  - tables
  - variant fields

- control structures or statements
  - assignment
  - goto
  - return
  - compound
  - conditional
  - iteration
  - case

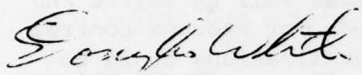
A number of questions about efficiency of procedure calls, and requirements for register usage within procedure bodies were not firmly resolved by the interviews or reprogramming. There was no clear requirement identified for particular parameter transmission methods, optional parameters, and multi-entry or exit capabilities. Requirements for reentrant as well as inline and inline assembly language procedures were verified. Needs for process control and real-time capabilities were identified, but no generalizations could be derived about the structure of process control notations. Interrupt handling and termination exception handling were other processing system characteristics determined to be needed in the HOL.

The proposed language, J73 with suitable modifications, represents a powerful communications programming language, combining very high level data structuring, storage management and storage control capabilities with very low level flexibility and high efficiency. It includes all the required types of values, with mechanisms for using them precisely and understandably, and a complete set of "structured programming" control structures, including notations for critical region support and exception handling. A simple procedure extension mechanism has been designed to enable special procedure types, with different mechanisms for argument transmission and special entry and exit actions, to be described and used. It will also be possible to make use of the J73 COMPOOL capability to collect declarations of defined types, access rights, and associated procedures to give the equivalent of encapsulation of extended types and monitors.

The language is described in terms of proposed modifications to J73 in Part II. The second year of the Higher Order Language Investigation for the CPS will involve finalizing the syntax of the modified language, specifying the full CPS HOL in both informal and formal terms, and specifying the CPS HOL compiler. The formal specification will be done using the SEMANOL language and specification system to describe precisely the syntax and semantics of the CPS HOL. The compiler specification will include correctness and performance tests as well as designs for major compiler modules.

EVALUATION

This effort with SofTech, Incorporated is an attempt to develop a high-order language (HOL) that can be used in communications applications programming where assembly languages are presently used. This is part of the RADC Technology Plan (TPO V/3.1) to create a HOL Environment, the objective of whcih is to reduce software costs and increase its performance. This particular effort will result in specifications for a language and its compilers to be used for programming communications applications.



DOUGLAS WHITE
Project Engineer

PART I

HYPOTHESES ON COMMUNICATIONS

HOL REQUIREMENTS

SECTION 1

DEVELOPMENT OF HYPOTHESES

This part of the report contains our final hypotheses about what HOL features are needed to effectively program communications applications. These hypotheses were derived from an extensive study involving:

- review of communications documents
- interviews and discussions with communications programmers and other communications experts
- a comprehensive review of potential HOL features in light of communications requirements
- use of existing HOLs to write selected portions of particular communications applications

Initial versions of the hypotheses were produced and submitted for critical review by groups interested in the projects. This final version represents the conclusions of our study.

Section 2 discusses the various factors that bear on the design of a communications HOL. It shows how the language design factors lead to the selection of particular language features.

Section 3 is organized by language area. It shows what features are needed in each area to support communications programming.

SECTION 2

COMMUNICATION LANGUAGE DESIGN FACTORS

There are a number of ways in which the characteristics of communications applications affect the design of an HOL to support these applications. This section sketches the major communications language design factors and illustrate how they motivate inclusion of particular language features.

2.1 Function to be Programmed

The most obvious factor affecting the design of a communications HOL is the functional requirements of the programs to be written. The communications HOL must clearly contain features powerful enough to express the required functions.

The communication functions to be programmed can be divided into three classes - on-line functions, off-line functions and development aids. Each of these classes of functions places slightly different requirements on the design of a communications HOL.

There are a number of advantages to using the same language for all of these areas. For instance, having a single language means less programmer training and fewer different support tools would be required. Taking the approach of providing a single communications HOL to support all of the areas implies that the functional requirements of each area must be provided for, regardless of the relative importance or difficulty of that area.

2.1.1 On-Line Functions

The first and most critical of the function classes are the on-line communication functions. These include the actual message or circuit switching functions together with on-line support function such as maintenance and diagnostics, startup, ledgering and recovery, and performance monitoring.

2.1.1.1 Security

Since military communications systems often carry secure or classified information it is critically important that security breaches not be allowed even in the presence of hardware or software errors. The security of a communications system must be carefully designed. The communication HOL alone cannot insure security. How the HOL helps to insure security is to increase the likelihood of correctly implementing a well designed system.

The most important HOL features contributing to reliable and, therefore, secure programming are strong typing together with the ability to define new data types and to limit access to particular data and procedures. This could allow, for example, creating signal data types for message buffers containing secure data. The programs having access to these buffers would be limited to a small number of carefully checked routines.

In a supposedly secure system the problem arises of verifying the system security. Note that it is not required that the reliability of the HOL compiler be completely established (this is not within the state-of-the-art). All that is required is to verify that the particular security related programs are correctly compiled. The verification can be done first at the HOL source level. Then the generated code can be verified by critically comparing against the already verified HOL source. This is a much simpler task than complete verification of a system at the machine language level.

2.1.1.2 Interaction with Specialized I/O Interfaces

An important characteristic of the on-line programs is the requirement to interact with specialized I/O interfaces. This implies that the communications HOL must provide some types of flexible low level I/O primitives. It also implies that the programmer must have control over data structure layout so that the data format appropriate to a particular I/O device may be setup. The ability to handle interrupts from various devices is also required.

2.1.1.3 Parallel Process Control

On-line programs almost always involve operations that are, at least conceptually, done in parallel. This implies the communication HOL must provide facilities for parallel process control. Since parallel activities may access common data some form of shared data, protection must be available. More detail on the parallel processing requirements of on-line programs is given in Section 3.3.6.1.

2.1.1.4 Control Over Program Size and Speed

On-line programs must often operate within critical time or space bounds. To allow programs to meet these bounds the HOL must allow control over program size and speed. This implies that the language should allow control over data structure layout so that the most appropriate time-space tradeoff can be selected.

In the most critical areas it may be necessary to rewrite a part of the programs in some type of assembly level language to allow the programmer to explicitly control the generated machine instructions. Since the use of explicit instructions can be dangerous, the HOL should impose some discipline on the use of assembly language inserts.

Other HOL features that contribute to programmer control include the ability to specify that procedures be expanded in-line and the ability to control the implementation of other features such as those for parallel process control. Appendix A provides a more complete discussion of HOL efficiency issues.

2.1.1.5 Other On-Line Requirements

Other communication HOL features required for programming on-line functions include the ability to deal with pointers and the ability to work with a variety of character sets. In special cases on-line function may be

more easily or efficiently programmed if it is possible to directly access the bit representation of a data item. This is especially true for operations on characters.

2.1.2 Off-Line Functions

In addition to the on-line function, most communication systems require some function to be done off-line, typically, by a backup processor. These functions may include preparation of reports on system operation, statistical analyses of operation, data retrieval from historical tapes, etc.

To meet the off-line functional requirements the communications HOL should provide for higher level I/O operations than are required for on-line programs. This includes formatted I/O for report preparation and possibly record I/O for reading off-line files. Note that this does not necessarily require that formatted I/O, for instance, be built directly into the language. It is perfectly acceptable to provide the capability as a subroutine package written using more primitive language features.

To carry out statistical analyses the HOL should include provisions for floating point calculations. Since the efficiency requirements of off-line programs are not high the floating point operations need not be highly optimized.

2.1.3 Development Aids

The final class of functions to be programmed in the communications HOL are program development aids. These are programs that are used to help develop or maintain other programs. They include, for instance, table generators which simplify the creation and maintenance of large, installation dependent tables. Perhaps the most extreme example of a program development aid is the compiler for the communications HOL itself.

The HOL requirements improved by the program development aids are the same as for any system programming task. Such tasks are characterized by the requirement to create and manipulate intricate data structures.

2.2 Programming Environment

Not so obvious as functional requirements but equally important to communications HOL design are the characteristics of the environment in which the communication programs are designed, written and maintained. The environmental factors to be considered include the program lifecycle and the characteristics and composition of the programming and maintenance groups.

2.2.1 Long Program Lifetime

Communications systems are designed to have a long lifetime, especially by the standards of the computer age. During the lifetime of a communications program it is highly likely to be fixed, modified or extended in some way. This implies that the programming should be done in an understandable and therefore maintainable way. It also implies that the ease of reading a program

is more important than the ease of writing. Any HOL features that contribute to making explicit the intent of the programmer are also useful.

These characteristics imply such HOL features as strong typing, range declaration, constant names, status data types and boolean data, all of which help in allowing the programmer to explicitly specify his intent. Other contributing features are explicit declaration and typed pointers. Support for structured programming must be an essential part of the communication HOL.

2.2.2 Programming Groups

Because communication programs are quite large it is necessary that a large group of programmers be applied to the program development task. This implies that the HOL should provide for separate compilations of individual modules so that the work can be effectively divided up. The HOL must also provide for intermodule sharing of data and procedures. Some means for creating and accessing libraries of definitions should be provided.

2.2.3 Program Reusability and Portability

New communications projects have many similarities with previous projects. The cost of developing a new communication system could be reduced if previously written programs could be reused in new applications. To support this the HOL should allow programs written in a general way to be specialized to a particular use. This will increase the chance that a program can be reused. The HOL should also make it possible to write largely machine independent programs.

These application characteristics translate into language requirements for conditional compilations to allow tailoring of a general routine, for constant names to allow parameterizing programs, and for facilities to encapsulate and set off unavoidable machine dependencies.

2.3 Language Design Principles

In addition to factors directly related to the application, there are a number of basic language design principles that must be considered when designing any HOL. First among these is safety. The language should be designed to detect errors or, better yet, make certain error logically impossible.

An HOL should be designed with the principles of simplicity, uniformity and orthogonality in mind. The simpler feature is preferred if both are functionally adequate (simplicity). A rule applying to one part of the language should apply in a similar way to analogous situations in other parts of the language (uniformity). As far as possible there should not be different ways of accomplishing the same function (orthogorality).

Finally, the language should be designed to support the state-of-the-art in programming techniques. This implies the language should encourage the use of modular, structured programming.

2.4 Project Constraints

The final major influence on the design of a communications HOL is the constraints on the language design project itself. The characteristic of the candidate base language will affect the final language. Other language design projects such as the DoD-1 effort also influence the recommended communications HOL. The communications HOL should differ from the DoD-1 requirements only when there is a good reason.

SECTION 3

COMMUNICATIONS HOL FEATURES

This section discusses the communications HOL requirements in each language area. It is intended to be used with the HOL features list in Appendix C. The features in the list marked CHOL refer to the hypothesized communications HOL. In many cases the statement in the features list stands alone as the statement of the communications HOL requirements. In other cases, the text in this section provides background information and arguments behind the selection of particular features.

3.1 Declarations and Storage Management

The purpose of declarations in a programming language is to describe the objects that the program deals with. These descriptions allow the compiler to select the code appropriate to the type of object being manipulated. The declarations also provide a redundancy that is useful in reducing the chance of error due to misspellings or misuse of objects. At each use of an object the properties of that object, as specified in the declaration, may be checked against the properties required for the use of that object. Any mismatch is a signal that something is wrong.

Because of the error checking capabilities of declarations, an explicit declaration should exist for each object used. There should be no implicit or contextual declarations because of the possibility that a misspelled identifier could, by implicit or contextual declaration, appear to be legitimate.

It is helpful to the reader of the program if all attributes of the object being declared are written explicitly in the declaration. This means there will be no misunderstandings between the author and reader as to what the attributes of the object are. There are some exceptions to this however. For attributes that are present in the vast majority of declarations and which are only overridden using explicit attributes it is reasonable to allow default specification. The attributes AUTOMATIC and INTERNAL are examples of reasonable defaults.

The HOL compiler should provide alphabetical cross reference listings of all declared objects. A listing showing where each object is used and modified would also be useful.

3.1.1 Scope Rules

The scope of a name refers to the part of a program in which the attributes of the name are defined by a particular declaration. In the communications HOL it is sufficient for the procedure body to be the unit of scope. Because procedures are expected to be short, there is no need for blocks to allow scopes smaller than a procedure body.

3.1.2 Variable Declarations

The communications HOL will clearly provide the ability to declare variables. The declaration will associate a type with the variable. Other attributes such as storage class or range may also be specified in the declaration.

3.1.3 Constant Declaration

The communications HOL will make it possible to associate a name with a constant value. The name may then be used in any context in which the constant could be used.

3.1.4 Type Declarations

One of the most important principles embodied in the communications HOL is the concept of strong typing in which the type of all expressions is known and checked at compile time. The value of strong typing is immensly increased by providing in the HOL the ability for the programmer to define new types and then declare variables of these types. The communications HOL will support the definition and checking of new user types.

3.1.5 Storage Allocation

The communications HOL will provide programmer control over the storage allocation for data by allowing specifications of a storage class for the data. Data may be allocated statically (STATIC storage), at procedure entry (AUTOMATIC storage) or dynamically on request (heap storage). The programmer must have control over the dynamic heap storage strategy for allocating and freeing storage. Relying on garbage collection for freeing heap storage is not acceptable.

3.2 Data Types and Operators

This section discusses, in detail, the issues concerned with the data types that should be provided by the communications HOL and the operations that should be allowed on those data types.

In general, there should be variables and constants of all types. The exact order of expression evaluations should not be specified to allow the compiler maximum freedom to optimize code and to discourage depending on a specified order. Anywhere a constant of some built-in type can appear a constant expression of that same type can appear. The evaluation of constant expressions will be done at compile time using the same type of arithmetic that would be done on the target machine. Then there is no difference in result between evaluation at compile time and at run time.

The communication HOL will, in general, do no implicit conversions between data of different types. Explicit conversions will, however, be available.

For some data types it will be possible to specify a range of values that variables of the type can take on. At the programmers option the compiler will generate code to check the actual value of a variable for the correct range whenever an assignment to that variable is made. Compile time checks will also be made for possible range conflicts. For instance, if the compiler can determine that the range of the source and the destination do not have any values in common a compile time error will be issued. If there are any values in common the assignment must be allowed. If the compiler can determine that the range of the source is a subset of the range of the destination, the assignment is known at compile time to be safe. When the assignment is not known at compile time to be safe, a run time check will be made if the programmer has selected that option.

3.2.1 Integer Data Type

Integer data is clearly basic to any application.

It must be possible to specify integers of different sizes up to at least 32 bits. It should be possible to specify the range of values a particular integer variable can assume. It should also be possible to specify the size (i.e., the number of bits allocated) for the integer. Either the size alone or the range alone can imply the other attribute, however, in some cases they should be specified separately (the specified size must, of course, be sufficient to hold numbers in the specified range). For instance, for an installation dependent parameter it may be useful to allocate a certain size field even though for some installations the range of values to be in that field does not require the full size. This can minimize the need for recompilation of subroutines accessing such a field.

The arithmetic operations of addition, subtraction, multiplication and division must be provided. Exponentiation is not required. A modulo or remainder operation would be useful. Algebraic comparison operators are required. The operations 2 to the Nth and log 2 (N) rounded up to the next integer (i.e., the number of bits required to hold value up to N) are useful in communications.

3.2.2 Real Data Type

Reals are not required in the vast majority of online, operational programs. They do, however, find use in offline programs. Since it is desirable to have a single language supporting both online and offline programs, reals should therefore be included. Including reals expands the range of application of the language and provides greater compatibility between the communications language and other languages.

Most computers used for communications applications do not have floating point hardware so reals must be handled by software. Efficiency is not of major importance in those cases where reals are useful in online or offline programs.

There are two ways of representing real numbers - floating point and fixed point. Fixed point requires simpler run time actions and so it is more efficient on a machine without special hardware. It is, however, harder to use

and to compile because of the need to deal with scaling factors. Floating point requires less thought by the programmer but requires more complicated run time actions and is therefore less efficient on a machine without special hardware. Since efficiency is not important in real number calculations it is sufficient to provide only floating point representation.

Again, because real operations are not critical in communications applications, tight programmer control over precision is not as important as in other applications. It is acceptable to provide a single floating point precision class. An implementation providing one word (about 32 bits) for the mantissa and one word for the exponent would be sufficient and also easy to implement.

The real operations provided should include addition, subtraction, multiplication, and division. Exponentiation can be done with a built-in function as can other operations such as trig functions, square root, etc. Relational operations are needed.

As for integers, it should be possible to specify a range of values that a real variable can take on. The range should not, however, be required.

There should be explicit conversion operations between integers and reals. Mixed mode arithmetic involving both integer and real variables is not needed. Automatic conversion of literals is useful and can be done at compile time.

3.2.3 Status Types

Status types (sometimes called enumeration types or defined scalar types) are required. (While they could be dispensed with functionally, the additional readability and reliability they provide is important enough to justify the requirement.)

Status types are defined by specifying a list of status constants that are to be the values of that status type. For instance, the status type color may be defined to hold values from the list (red, yellow, orange, green, blue).

In practice it is necessary to allow the same status constant name in different status types or to have other variables or constants with the same name as a status constant. These problems are often solved in particular projects by establishing conventions for naming status constants in such a way that the type name is made part of the constant name (e.g., by a prefix or suffix indicating the type name). This argues in favor of some type of qualification of status constants.

To allow for interface with other devices and to provide some type of optimization, it should be possible for the programmer to determine what representation is to be used for the status constants. This can be done by specifying an integer value to be used to represent the status value.

Some status types are used as ordered types while in other status types the order of the values is not important. Many status types (e.g., days of the week) may be thought of as being cyclic. It would probably be useful to allow the ordered-unordered distinction to be explicitly stated. The cyclic attribute is not useful enough to warrant inclusion.

Operations on status data include comparison for equality and inequality, ordering comparisons for ordered status types. Successor and predecessor operations are needed in the absence of special looping forms.

Status types should be allowed as array or table indices. They will be especially useful in conjunction with a CASE statement to select from among a group of possible actions.

Loop statements that cause an index to successively take on values from a status type may be useful.

Explicit conversion to and from integers should be provided as an escape hatch to allow for special operations.

3.2.4 Bit Data Type

Some means of manipulating groups of bits is required. The primary reason for this is to allow manipulation of the representation of other data (for example, to allow masking out certain bits from the representation of a character to map lower case to upper case).

There are two approaches to this. The first is to provide a bit string data type. The second approach, which was introduced in PASCAL, is to provide a set data type. A set data type may be thought of as a collection of bits which is associated with an underlying base type, such as a range of integers or a status type. There is one bit for each possible value in the base type. If the corresponding bit is a 1, a value is said to be in the set. Otherwise, it is not in the set. The set data type thus represents a set of 0 or more values drawn from the base type.

Since the purpose of allowing access to bits is to allow manipulation of representation, the more direct bit data type is preferable to sets. Experience has also shown that the concept of sets might possibly cause training difficulties.

Required bit string operations include union (OR), intersection (AND), complement (NOT), equivalence, and exclusive OR. Tests for equality and inequality are required. A test for the value of a single selected bit is required. Shift operations are required. The ability to specify bit constants is required. It should be possible to extract or change a selected substring of a bit string.

The length of a bit string will usually be known at compile time; however, there may be cases where the exact length will not be known until runtime. In any case, once the string is allocated, its size will be fixed.

There is no need to provide for strings of dynamically varying size. There is no need for a bit string concatenation operation.

3.2.5 Booleans

Boolean data is required. The only issue involved is whether or not booleans are to be a special case of a bit string containing only one bit. Since bit string data is an escape hatch to allow manipulation of representations, it is not appropriate to use bit data to represent boolean data. A separate boolean data type is needed.

The boolean operations of AND, OR, and NOT are required. An exclusive OR operation is required. The comparison operations for equality and inequality, which are the same as the boolean operations of equivalence and exclusive OR respectively, should be provided.

Boolean operations should be evaluated in short circuit mode (i.e., if the value of an operation is known after evaluating only the first operand, then the second operand is not evaluated). This is required to support the optimal use of structured programming.

3.2.6 Character Data Type

Character handling is one of the most important activities in message switching and to a slightly less degree in packet switching. Because of the importance of character handling, a communications HOL must provide character handling features that are well tailored to the requirements. Character string handling is often on the most time-critical path. It is therefore important that these operations be capable of being performed efficiently.

3.2.6.1 Operations on Strings of Characters

There must be a way to declare a variable or a field holding a string of characters up to a specified maximum length. Such a variable may be thought of as a buffer.

The number of valid characters in a buffer may vary from 0 up to the maximum size of the buffer. The decision about the number of valid characters in a string is up to the programmer. Sometimes a count of characters is used; sometimes a special terminating character is used to mark the end of the string. This implies that the idea of a varying length character string must not be built in into the language, but must be under control of the programmer. There is no requirement to support character strings with a dynamically varying maximum length.

It must be possible to handle a buffer of characters indirectly through a pointer. It is necessary to have arrays of character strings. It is reasonable for each character string in an array to be required to have the same maximum length.

It must be possible to move an entire string of characters. It must be possible to compare entire strings of characters for equality, inequality, and ordering relations. It is necessary to extract or replace a contiguous substring within a larger string.

An infix concatenate operator may be conceptually useful. However, its unrestricted use can be expensive since it may require generation of string valued temporaries. The safe use of concatenations is to move a series of strings into an already allocated buffer. This could be done with the substring replace operation mentioned above.

A variety of scanning and searching operations are required. It should be possible to scan a string for the occurrence of one of a set of specific substrings. It should be possible to verify that a string consists entirely of characters in some specified class such as alphabetic or numeric.

Since it is probably not possible to conclusively identify all possible scan and search operations required there should be some general mechanism to allow the creation of efficient new operations. Such a mechanism (sometimes called a driver) would do a character-by-character crawl through a character string carrying out some operation on each character.

3.2.6.2 Operations on Individual Characters

There must be a literal representation for individual characters. It must be possible to represent both printing and non-printing characters.

It is sometimes required to treat a character as if it were an integer (e.g., in hash coding applications or in computing special packed representations such as the DEC Radix 50 code). If ordering relations are not defined for characters, this same escape hatch may be used to make ordering comparisons when needed.

It must be possible to test for special classes of characters such as alphabetic, numeric, control, etc. The properties of the character set should be taken into consideration to make these tests efficient. It is necessary to obtain the equivalent numeric value of digit characters.

It is probably necessary to allow arbitrary bit manipulation of characters as an escape hatch to provide the flexibility to make optimal use of the properties of a particular character set. The need to use this escape hatch should be minimized by providing primitive operations (for instance by use of special functions) for the operations normally required such as the conversion from digit to numeric value mentioned above.

It is often required to translate characters from one character set to another. Sometimes this is a simple one-to-one translation but other times more complicated one to many, many to one, or even many to many translations are needed (e.g., converting fractional characters such as '1/2' available in some sets to the separate characters '1', '/', '2' or supplying or deleting case shift characters when translating to or from BAUDOT).

3.2.6.3 Character Set Issues

A character set is a specification of a set of printing and non-printing characters together with a binary representation for each character. In addition, there may be conventional meaning attached to particular sequences of characters (such as sequences involving BAUDOT case shift characters or ASCII escape characters).

There are a large number of different character sets in use currently. (E.g., several minor variations on ASCII, at least three modifications of the BAUDOT code, EBCDIC, and other modified BCD codes, etc.) It is not possible to interpret a particular bit pattern as a particular character (or vice versa) without knowing the character set.

A message or packet switch must be able to handle messages written in one of the large variety of character sets. In fact, one of the reasons for having a centralized message switch is to concentrate all the incompatibility in one place and provide a machine with sufficient power to deal with all the different possible translations required.

Often systems are built around a single system character set (most frequently ASCII) used internally for all purposes. Translation to and from this system character set is done as required on input and output. This simplifies the internal system at slight increase in input and output time.

Although the interpretation of a character representation depends on the character set, and several different sets must be handled, the ability to specify the character set to be used for any literal character string is not required. Because of the use of a system character set, as discussed in the previous paragraph, it is reasonable to do without the convenience of using literals in other character sets. Where it is required to deal with specific characters in another code the representation can be explicitly used.

In the simplest definition, ordering relations are determined by the binary representation of the characters in a character set. Thus, the usefulness of this simple ordering relation between two characters depends on the character set representation. Providing a representation independent ordering relation would be much more complicated.

It is sometimes important for efficiency to exploit the properties of the binary representation provided by a particular character set. For instance, in the ASCII character set the alphabetic characters form a contiguous sequence and are in alphabetic order. Thus, to test if a character is alphabetic all that is required is to test that the value lies between the value for 'A' and 'Z' inclusively. Other character sets do not have the same properties. For instance, in EBCDIC the alphabetic characters are not contiguous and in the BAUDOT set they are neither contiguous nor in alphabetical order.

The operations which depend on the properties of a particular character set should be encapsulated in some way so that the source text of a program will not depend on the character set of the data being processed.

3.2.7 Pointer Data Type

A communications language must support construction and manipulation of flexible linked structures where various data elements are linked by pointers. In addition there must be a way of indirectly referencing such things as character strings (buffers).

Pointers must be provided in the language to allow creation and manipulation of linked structures. In accordance with the ideas of strong typing there will be a type associated with pointers indicating the type of object pointed to. There must be an escape hatch for overriding this type checking in special circumstances.

The only required operations on pointers are comparisons. Comparison for equality and inequality are required. Ordering comparisons are also useful, for example, in minimizing search time in lists of pointers. The ordering is useful even though it is arbitrary.

It should not be necessary to do arithmetic on pointers in normal processing. There are cases (for instance in a free storage management routine) where pointer arithmetic is needed. This may be handled by use of an escape hatch to allow treating a pointer as an integer.

There are some dangers in using pointers. The most serious is that it sometimes happens that a pointer outlives the data item pointed to. In this case, a reference to the object via the pointer would have unknown and possibly serious results. A way of avoiding this is to insure that the lifetime of the pointer is not greater than the lifetime of the value pointed to. For dynamically allocated data objects it is not possible to check this condition if the programmer may explicitly free the dynamically allocated storage. Another problem with this restriction is that in some languages to allow access to a pointer variable from different modules it is necessary to make its lifetime be indefinite because of the coupling of variable lifetime with variable scope.

3.2.8 Label Data Type

Labels are required in any practical language both as the target of GO TO statements and as identifiers of program points and program sections.

All that are required are label constants. It is not required and would be very harmful to allow label variables. The use of label variables makes it impossible to understand the control flow through a routine without following the whole computation so it can be determined what the value of the label variable is.

For interface to programs written in assembly language it is necessary to be able to pass addresses within a program. This provides an escape hatch, allowing for example creation of special calling sequences. To provide this interface some type of conversion operation that turns a label into a machine address is required.

3.2.9 Procedure Data Type

Procedures in an HOL are the primary modularization tool. They determine the means by which a large system is broken into understandable pieces. Procedures in a communications HOL must be designed to make a positive contribution to security, reliability and understandability while not unduly sacrificing efficiency.

3.2.9.1 Procedure Usage

In addition to the traditional procedure usage in which procedures are constant objects called directly in a program, a communications HOL should allow the use of procedures as a data value. This allows the identification of a procedure to be assigned to a variable or stored in a data structure for use at a future time.

The ability to treat procedures as data values allows large applications to be almost entirely table driven by allowing the identification of procedures handling specific cases to be stored in the data structure. The driver interpreting the table can then retrieve and call the appropriate procedure without the identification of the procedure being built in to it. In communications, frequent use is made of state transition tables in which an action is associated with each combination of state and current event. These actions can be specified by entering the identification of a procedure to carry out the action as part of the state transition table.

Other uses of procedure data values are to store the identification of error handling or ledgering procedures as part of the data structure they are concerned with. Again, this allows decoupling of the application using the data structure from the details of the implementations.

The alternative to procedures as data values is to use integers or status values as codes for procedures and then switch on the code into a case statement calling the procedure corresponding to each case. This is not nearly as flexible as direct use of procedure values and is unduly time consuming.

Going along with the use of procedure identification as data values is the ability to pass procedures as parameters and to have them returned as values of functions. The usual operations of assignment and comparison for identify should be provided.

Procedures used as variables or parameters must not access global data in the automatic storage class. Only global static data or heap data may be accessed without passing the data as a parameter to the procedure. This is to ensure that the procedure variable can be implemented as a simple pointer, without an extra pointer specifying its dynamic environment.

The primary operation on a procedure is to invoke or call it. While the exact form of the invocation operation is not important it is important that there be checking for correctness of the call. The number and type of actual parameters must be checked against the corresponding formal parameters.

It is important that the communications HOL allow a choice between the traditional procedure invocation by jumping to a section of closed code and actually inserting the procedure code inline at the point of the call. The ability to specify the procedures be called inline allows all the advantages of modular program constructions without the overhead of the normal subroutine call and return. Coupled with the ability (discussed below) to define procedures written in explicit (assembly language) code inline procedures form a powerful extension mechanism to define new special operations for particular tasks. It is possible using this mechanism to gain access to special purpose machine instructions such as those for I/O or interrupt enable/disable.

Another advantage of inline procedures is that it is possible to apply constant expression evaluation as the inline call is being expanded. Any operations involving constant actual parameters can be done at compile time. This includes, for instance, the test in an IF statement, allowing for compiling only one branch in some circumstances. In this way a general routine may be automatically tailored to the needs of the particular call.

3.2.9.2 Procedure Definitions

It must be possible in a communications HOL to define reentrant procedures. There should be an attribute to mark reentrant procedures since it is not required that all procedures be reentrant. Recursive procedures are not so important but could be provided if the overhead were not high.

In special cases such as interrupt handling, for extreme efficiency in critical areas, or to force utilization of specific machine register to pass parameters, it must be possible for the programmer to control the procedure entry and exit sequence. This could be done either by defining the entry and exit sequences to be used or by specifying that no entry and exit sequence be produced by the compiler, allowing the programmer to insert his own by means of inline procedure calls at the entry and exit points.

As has been mentioned before there are cases in which the programmer must explicitly control the code generated for a procedure. This is required to get access to instructions not used by the compiler, such as I/O or special looping instructions, and for top efficiency in certain cases. The communications HOL must provide some means, such as assembly language, to allow the generation of explicit code.

The use of explicit code should be segregated from other parts of the program. This can be done by limiting the use of explicit code to the bodies of specially marked procedures. Note that the inline call mechanism can be used so that this segregation causes no loss of efficiency.

While support was noted for the idea of procedures with multiple entry points, such procedures need not be provided in the communications HOL. The functional equivalent can be provided by defining the actions common to two different entry points as a separate procedure or by calling the procedure with a status parameter indicating the conceptual entry point. Providing multiple entry points opens the way for errors such as accessing an argument of an entry point other than the one used in a particular call.

3.2.9.3 Parameter Handling

A procedure definition includes a specification of the interface between the procedure and the calling program in the form of a list of parameters. While there are some cases where a variable number of parameters would be useful, consideration of the type checking difficulties and possible sources of error with variable number of parameters leads to the conclusion that the language should only provide for procedures with a fixed number of parameters.

The procedure definitions should allow access restrictions to be specified for the parameters. It must be possible to limit parameters to read only access, read/write access or to write only access. These access limitations improve program understandability and aid in demonstrating the reliability and security of the system.

By specifying parameters access restrictions in the above form much programmer confusion about call by value, call by reference and call by name can be dispelled. Read only parameters can be passed by value or by reference at the compilers option (possibly under control of a programmer directive). Write only parameters may be handled either by result or by reference. Read/write parameters must be handled by reference.

The use of output parameters (read/write or write only) in a function opens the possibility of side effects which could cause the different results depending on the order of expressions evaluation. This can be avoided by forbidding output parameters (and all other write access to global data). This restriction is unduly severe since it disallows some useful programming techniques. It is sufficient in practice for the compiler to check for obvious dangers such as use of an actual output parameter in the same expression as a function call.

Another possible source of side effects is aliasing or linkages among parameters. For instance if the same variable is allowed as both an input and output parameters of a procedure the supposedly constant formal input parameter may change its value. This could cause behavior to depend on whether call by value or call by reference is used. When pointers are available it is not possible to determine if linkage is present since an input parameter could be pointed to by an output parameter. Because of the impossibility of complete checking, all that should be done is to check for the obvious cases of possible storage overlap among writable or read/write parameters. For instance, passing the same variable to two different writable parameters would be caught.

3.2.10 Table Data Type

The typical communications program, be it for a message, circuit, or packet switch contains many tables. Tables are data structures organized as arrays of identically structured entries, where each entry is in turn subdivided into named fields (see Figure 3-1). The fields of an entry are of constant length. The smallest possible field occupies a single bit, but fields occupying several words are also required. In overall concept, the form of tables used in communications programming is essentially identical to that of JOVIAL tables,

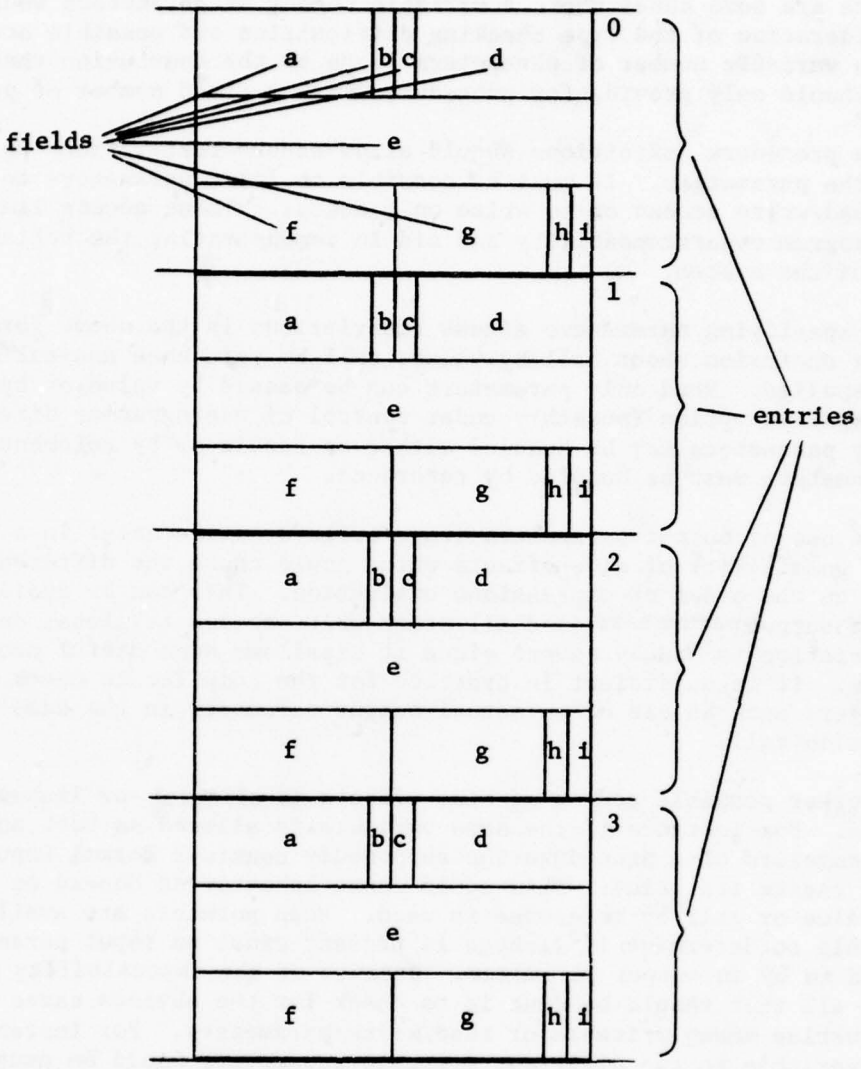


Table Structure

Figure 3-1

although capabilities not directly supported by JOVIAL are also exploited in current communications programs. We will discuss these capabilities later.

Most tables in a communication system consist of a single entry. Perhaps 25% of the tables may have multiple entries. For example, the line table of a small circuit switch might have 150 entries (one for each line) with 30 to 40 fields per entry, requiring 20 characters of storage.

In the TRITAC message switch, there are approximately 450 different tables. AUTODIN has about 200. Circuit and packet switches tend to have fewer tables. A small commercial PBX may have 100, a small packet switch might have 50. A military packet switch such as AUTODIN-II would probably have 200 different tables.

Tables are used to store permenent and temporary control data and constants, line status, device status, routing information, alternative routing information, line protocols, security information, priority, working queues, code conversions, format conversions, buffers, etc.

The present use of tables in communications systems is so all pervading that many of these systems are said to be "table driven". This structure has evolved as a direct result of trying to make communications software more easily maintainable. Wherever it has been possible to substitute tables of constants and parameters for specialized code it is done. A common control routine uses the combination of line state (stored in the line table) and the identity of the signalling event just detected (obtained through a signalling event translation table) as an index to a sequential jump table that produces the new state for the line and directs the program (via the jump table) to the subroutine that is to execute the response to the detected situation. Different signalling protocols are accommodated by different table entries. Should it be necessary to change the protocol, entries in the various control tables will be revised, rather than the code. A similar approach is used in message switching for code conversions and format conversions. This use of jump tables suggests the need for case statements or case expressions (as in ALGOL 68) or arrays of subroutines (called "entry arrays" in MULTICS PL/I, which permits arrays and variables of subroutine data types to be declared). The use of jump tables and their HOL implications will be discussed in the section on control structures.

3.2.10.1 Table Organization

A table is a combination of the ideas of arrays and structures found in many HOLs. Tables are a more general concept than arrays and structures (since both are special cases) and likely to be more familiar to communications programmers. On the other hand the separate concepts fit in more easily with the ideas of strong typing and user defined types. Either arrays and structures or a special table type would be acceptable for a communications HOL. The decision should be made in accordance with the chosen base language.

The remainder of this section is written in terms of tables. The points mentioned about table entries apply to structures as well. The points concerned with table indexing apply to arrays as well.

3.2.10.2 Table Entry Form

The structure of an entry is the identification and arrangement of the fields within the table. The size of an entry is determined by its structure. The entries in most tables have a constant structure which is known at compile time. Some tables, however, contain entries that may have one of a relatively small number of alternative structures. In these cases it is not known at compile time which alternative is present at any point in time. The maximum entry size is, however, known at compile time. In most cases it is acceptable to allocate the maximum size for the entry. In some cases it may be desirable to allow the programmer to specify a restricted set of one or more variants that will be assumed for a particular table and to allocate only enough space to hold the largest of these variants.

To handle tables with alternative structures the programmer must have some means of deciding at run time what alternative is present. This information is not necessarily contained in the entry itself.

Typically, entries with alternative structures contain some fields which are common to all variants together with several fields that are specific to the particular alternative. Since the alternatives are mutually exclusive, it is possible that fields specific to one alternative may overlap fields specific to another alternative in an arbitrary way.

It is possible that a specific alternative structure may itself contain alternatives. This leads to a tree structured set of alternatives.

A very few tables seem to require entries of varying size where the information about the actual size is contained in the entry itself or in some other runtime accessible location. Such tables cannot be indexed and must be accessed by pointers or by searching. There must be some algorithm known to the programmer for stepping from one entry to the next.

3.2.10.3 Table Fields

Table fields must be able to contain data of any type. In particular it must be possible for a table field to contain another table. This gives a hierarchial structure.

It is not necessary to allow fields to overlap. This is dangerous since data stored in one field would affect the value of another field. The use of variant structures will provide all the overlap that is needed.

It is sometimes necessary to allow fields to be packed into less than a word, although this is not as important as it once was. Field packing is defined by the position and size of that field. It should usually be sufficient to allow the compiler to select the exact packing for table fields under general guidelines from the programmer. There are cases where the exact packing

must be controlled by the programmer, for instance, when fields are defined by existing hardware or software. It should be noted that packing is a property of the field and not of the table as a whole. Often it is important to pack some fields even though others could be unpacked.

3.2.10.4 Table Indexing

Tables with 0, 1, 2 and 3 dimensions are required. (A table with 0 dimensions consists of a single entry which is not indexed. It is the same as a structure in languages with arrays and structures.) No requirement has been identified for tables with more than 3 dimensions.

To index into a table requires that all table entries have the same size. A table with varying length entries must be searched or accessed via pointers.

3.2.10.5 Table Operations

It should be possible to move entire table entries with a single operation. In some special cases it may be useful to allow other operations on tables. For instance, a table whose entries consist of a single character may be considered to be a character string and therefore, subject to the character string operations.

3.2.10.6 Table Storage Allocation

Some flexibility in allocation of storage to tables is necessary. In communications systems tables often contain fixed data and dynamic data. The dynamic data is periodically stored ("ledgered") against the possibility of a restart. The physical storage of the table is often arranged so that all data to be ledgered is grouped so a single I/O transfer will store the entire collection of dynamic data.

The minimal capability needed is the ability to specify that the table is to be stored serially (with each entry contiguous) or in parallel (with corresponding words of each entry contiguous). A more flexible scheme, such as is available in PL/I, in which the programmer could specify how the storage is arranged, would be useful.

3.3.1 Assignment Statements

There is clearly a need for an assignment statement in any HOL. Only the straightforward single assignment is required. It is not necessary that the assignment statement yield a value or allow multiple targets.

The issue of what can be assigned to what are discussed under the section for each data type. In general, strict type checking is done and no implicit conversions are allowed across an assignment.

3.3.2 Block/Compound Statement

Because modern programming practice dictates that procedures be small there is no need for scope-defining blocks below the procedure level. The only use of a compound statement is to group a number of statements into a single statement.

3.3.3 Conditional Statements

Conditional statements are clearly required in any application to allow a choice of action based on the occurrence of particular conditions.

There is no need for conditional expressions (which select a value depending on the occurrence of particular conditions). Conditional statements can adequately perform this function. While on the surface there may appear to be some efficiency advantages in conditional expressions, a moderately sophisticated compiler can compile just as good code for either case.

It should be possible to select one of two courses of action based on the value of a boolean condition (IF-THEN-ELSE statement). If the action to be taken in the false case is null, the ELSE clause can be omitted (IF-THEN statement).

It should also be possible to select one of a number of courses of action based on the value of a given expression (CASE statement). Each possible course of action should be marked to indicate those values of the given expression for which this action should be selected. There must be a course of action specified for each possible value of the given expression.

3.3.4 Iteration Statements

The iteration statement, or loop statement, is another fundamental control structure. It specifies that a body of code should be exercised repeatedly until some termination condition is met. There are two basic forms of iteration statements -- the definite and the indefinite. The definite form includes some explicit specification of how many times the loop should be repeated; the indefinite form does not. There are also variations which combine these two basic forms.

The communications HOL should contain both definite and indefinite loop forms. The test for loop termination of an indefinite loop should be allowed to appear anywhere in the loop -- at the top, at the bottom or anywhere in between. In normal cases, it should not be necessary to exit from the loop other than at the specified loop termination test point. An exception could cause unusual termination of a loop. A GO TO statement could also be used to terminate a loop, however, this use is not recommended.

It is often important that, in a communications environment, loops execute efficiently. It is worth some time and space to do loop setup operations in a way which would permit optimization of those operations which occur every time through the loop. This implies that increment and termination values be evaluated just once, prior to entering the loop.

3.3.5 Unconditional Transfer of Control

Although the use of unconditional transfers (GOTO statements) is discouraged in good programming, such a statement must be provided in the communications HOL. It is needed to allow synthesis of control structures not provided in the language and to improve efficiency in certain critical cases. The use of GO TO statements is potentially dangerous and should be put under administrative control.

Esoteric forms such as indexed or computed GO TO are not required since their function is taken over by the CASE type of conditional statement.

Nonlocal GO TOs are required only to support exception handling. All other GO TOs are strictly limited to the local procedure in which they occur.

3.3.6 Parallel Process Control

The term process control is used here to refer to those programming issues, features, problems and characteristics having to do with event scheduling, interrupt handling, resource sharing, process synchronization, timing and other related topics. Process control is central to the communications applications. It is also a most difficult HOL area. The problems involved in dealing with these issues have not been adequately addressed by language design researchers. Most languages incorporating process control features are academic rather than practical tools, e.g., MODULA, Concurrent PASCAL. Very little practical experience has been accumulated in the use of an HOL in these areas. Because of these difficulties the only reasonable recommendation that can be made about what process control feature should be in the communications HOL is for extreme flexibility so the solution can be tailored to the particular application.

The following section discusses the use of process control in communication. Following that is a discussion of the approaches to providing process control features. Finally, more detail is given about the recommended approach.

3.3.6.1 Process Control in Communications

Broad statements of required HOL capabilities can be deduced from the very nature (the functional requirements) of communications systems. More specific capability descriptions must be based upon the methods and techniques used to implement communications systems. This section describes, from both viewpoints, those communications systems characteristics which we believe will have the greatest impact on the process control capabilities of the HOL.

3.3.6.1.1 Functional Characteristics

Communications systems are concerned with transferring information from one or more receivers, with setting up connections to support direct communications, and with controlling the activities of various devices. Often there are reliability and timing constraints. These may require that faults be detected and that alternate or parallel paths, devices or techniques be selected. Sensitive information may be transmitted, thus imposing data access restriction requirements of a complex variety, since all combinations of secure, private and

nonsensitive data must be dealt with. Finally, communications systems must be capable of receiving from and transmitting to a wide variety of devices. These devices vary in transmission rates, character sets, signalling protocols, message formats, and any number of other ways. Some must be polled, others will demand attention. Some will be interfaced to the processor directly, others will be tied to a common bus. The possibilities are staggering. The communications HOL must deal with this general problem.

These characteristics have serious implications for the HOL. For example, the timing constraints, the varying transmission rates, and the variety of interfaces all point to an HOL facility for dealing with clocks and timers. Similarly, that same variety of device interfaces indicates the HOL must provide ways of dealing with interrupts. Data access restrictions suggest sophisticated means for handling shared data and synchronizing the processes which can access it. To accomplish the required levels of reliability and accountability, communications systems periodically save the system state so they may backup to a recent checkpoint if a restart becomes necessary. Besides aggravating time and memory problems, while not actually furthering accomplishment of the system functions, restarting requires knowledge of what comprises the system state, what has changed since the previous checkpoint, etc.

3.3.6.1.2 Design Characteristics

Communcations systems are frequently distinguished by being placed in one of two broad categories -- those which are cyclic in nature and those which are interrupt-driven. It is our contention that the differences are not really basic. Certainly as far as HOL implications are concerned, the similarities are of much greater interest than the relatively minor differences. The distinction usually made between cyclic and interrupt-driven systems is that cyclic systems save up processing tasks to be done on a periodic basis while interrupt-driven systems process requests as they occur. This does not appear to be a very useful distinction, however, since there are many variations on these themes and a great deal of overlap. Some parts of a system may be cyclic and others interrupt-driven. Another distinction which may be more enlightening is that cyclic systems maintain queues of work to be processed by regularly scheduled activities while interrupt-driven systems maintain queues of activities to be executed. This distinction will be clarified in the following descriptions of cyclic and interrupt-driven systems. The descriptions are oversimplified to emphasize the significant features.

3.3.6.1.2.1 Cyclic Systems

In a cyclic system, an executive program repeatedly steps through, or cycles through, a predetermined, and ordered, list of activities which must be accomplished, calling the corresponding programs in turn. Each activity has a queue of work to be done. On any given activation, the program or activity completes as much of the work in its queue as is reasonable. What is reasonable is a design consideration and one of the variables for each activity. Activities can add work to the queues for other activities.

A cycle consists of two broad categories of activities: those which are mandatory and must be activated every cycle, and those which are activated upon demand. In fact, the queue for a given mandatory activity might be empty on a given cycle, in which case the activity would immediately terminate. Mandatory activities include such things as scheduling output lines (SOL), line supervision, header verification, and message routing. Operator/system dialogues would come under the heading of "on demand" activities.

Typically, then, a cycle would begin with cycle initialization performed by the executive, proceed to the mandatory and then the supervisory activities, and end with ledgering activities which would record all dynamic core which could have changed during the cycle. Then the cycle is repeated.

While all this is going on, interrupts are occurring asynchronously. The cycle runs at a relatively low priority level and is interrupted to process events occurring at higher levels. Interrupt handling is brief, consisting primarily of responding to the interrupt and adding work to the queues to be processed by various activities during the cycle.

There are many variations on the basic cyclic system. There can be cycles within cycles; ledgering can occur incrementally during the cycle rather than at the end; "on demand" activities can be initiated at the first opportunity after they are recognized; and, as alluded to earlier, the amount of work done by a given activity on a given activation can vary. For example, a routine, or activity, may be restricted to whatever it can do in the time of 5 drum revolutions, or 50 ms, or to x items in its work queue. The fact that the normal cycle activities can consume varying amounts of time leads to another system variation; minimum times are associated with each activity in the cycle and, whenever even these minimum times may be not required, the "holes" may be filled with processing on some other activity. Maximum times are also included as checks on system viability. Whenever the maximum time is exceeded, this is an exception.

The following example will clarify the structure of a basic cyclic system. The program to schedule output lines (SOL) is a mandatory part of the cycle. SOL examines the line tables to determine if any line has met its schedule criteria. The schedule criteria concern:

1. Whether there is traffic for the line;

2. What the priority of the traffic is;

3. The speed and protocol of the outgoing line;

4. Whether any blocks have yet to be transmitted;

5. System state information such as the buffer block threshold condition and the queue block threshold condition.

If it is determined that there is a need to schedule data for output, SOL fetches a queue block and builds a work order. It then checks the next line, continuing until all lines have been scanned or the output queue has reached a specified threshold.

SOL then passes control back to the exec which determines which function to initiate next. The next step is typically a cyclically invoked disc scheduler, which has been gathering read requests from several sources -- SOL, exec, interrupt level routines, human operator, input processing functions, etc. The disc scheduler builds a read schedule list or matrix. This can be done as a queue of commands, in a one dimensional table, a two dimensional table, etc. The specific details depend upon the device, the system, the application, etc. When the table has been built, including the construction of the I/O commands, control is passed back to the exec. The exec initiates the first read operation by properly activating the disc handler. In some systems, the disc handler integrates read and write commands. The disc handler operates on at least four priority levels:

1. The base sequence level (cycle level)

2. Exec (which is typically not at the base level)

3. The disc interrupt level

4. An exception condition level -- also interrupt driven

The first command in the command chain is initiated by the exec via the disc handler, at the exec level or the base level. When the interrupt occurs signifying that the first read has been accomplished, it may be treated at two priority levels. The first priority level fields the interrupt, validates it, types it, queues the contents to the next lower level, and unblocks interrupts.

The second priority levels examines the type and transfers control to the disc handler. The disc handler issues the next command in the command queue, does some housekeeping, and typically allocates another buffer block or group of blocks. Note that the storage allocation routine is also in operation at several levels.

When the entire set of queued operations has been completed, the interrupt level handler will note the fact in a control table used by exec. Meanwhile, exec has been continuing the cycle. If the system is well tuned, within a few milliseconds after the completion of the last read operation, the exec will get to the point in the cycle where it checks to see if the read has been done. If it has, it will initiate the next phase in the cycle.

3.3.6.1.2.2 Interrupt-Driven Systems

Rather than maintaining one work queue for each of several activities, interrupt-driven systems maintain several priority based queues of activities to be performed. Interrupts from various sources, indicating requests for service, cause designated programs to be scheduled by placing an entry on the appropriate queue. Each program, in turn, schedules the next program which must process the request, and so on until the request has been completely handled. Programs may be scheduled for processing immediately, after a specified delay, or on the occurrence of a particular interrupt or event. On each interrupt or program termination, the executive program selects the highest

priority program which is ready to run (which may not necessarily be the interrupted program). Interrupt-driven systems vary in the strategies used to schedule programs, or to select the next item in the queue to be acted upon.

If the example described above referred to an interrupt-driven system, operation would not be significantly different. The termination interrupt on the output line (i.e., termination of the block currently being transmitted) would be fielded at a high priority level and would cause the output line handler to examine the line state to see if there was anything more for this line. If there were, it would construct the disc read orders and queue them to the disc handler and then quit that level. The disc handler, or perhaps an intermediary optimizer would arrange the I/O order in a list, table, etc., which would depend, among other things, on the current position of the disc. Thereafter, the operation is similar to that of a cyclic system in that the interrupts are fielded at a higher priority level and result in the construction of work queues and state changes.

3.3.6.1.3 Time-Sense

A factor which plays a major role in communications systems performance is the need to detect system failures of various sorts and take corrective action. Many such faults are detected or confirmed by noting failure to respond in a given amount of time. For example, if an intercharacter message delay should exceed a predetermined threshold, then a line would be considered to be functioning abnormally. The abnormal condition would then be monitored and if its duration exceeded a given value, the line would be put in an "open" state. Monitoring would continue until the line was known to be good for a certain amount of time, when it would then be put back in service. Or failures can be detected by actually polling the equipment and timing how long it takes to receive the acknowledgement. A timeout would indicate some type of failure.

Monitoring of this nature may require as many as 2 or 3 interval timers per line or circuit. These are not hardware timers which actually issue an interrupt when the timeout occurs. Rather, they are software timers. These may be implemented in a variety of ways. For example, one may save both the current clock value and desired interval value when the timer is set and later compare actual elapsed time with the desired interval to see if timeout has occurred. Or, one can set a count to the desired interval value and decrement it every cycle (or N cycles) and check for less than or equal to zero.

The first of these techniques requires access to an ordinary clock. A clock will also be required for timestamping of messages or connections and for timing various events such as the duration of a call.

3.3.6.2 Aspects of Process Control

Process control can be divided into two major areas -- processor management and resource management. Processor management deals with flow of control issues such as starting and stopping tasks, scheduling, interrupt handling and switching the processor from one task to another.

Resource management is concerned with insuring that resources (such as data or I/O devices) shared among more than one task are properly protected. It is vital to insure that a task which can change the state of the shared resource has exclusive access to the resource. Otherwise, timing dependent errors can occur if one task happens to read the state of the resource when another task is in the process of changing it.

3.3.6.3 Approaches to Process Control

The approaches to process control can be roughly divided into low level approaches and high level approaches. The low level approach is characterized by "close to the machine" primitives such as saving and restoring registers, enabling and disabling interrupts, etc. The high level approach provides more abstract "close to the problem" features such as processes or tasks, monitors or critical regions, etc.

In general the low level approach provides greater flexibility and closer control of the available resources. The high level approach, on the other hand, is closer to the problem and generally safer and more readable.

There is also a compromise between the high level and low level approaches in which the language contains high level features, but allows the implementation of these features to be controlled by the programmer. This approach combines some (but not all) of the safety and readability of the high level approach with some (but not all) of the flexibility of the low level approach.

The compromise approach can be implemented by defining a pattern of procedure calls that will be generated by the compiler for a particular high level feature. The implementation of the procedures is then left up to the programmer. In this way the programmer retains much of the flexibility and efficient access to machine resources that is available in the low level approach.

3.3.6.4 Process Control Recommendations

3.3.6.4.1 Processor Management

For the processor management aspect of process control the communications HOL should provide a low level approach. The low level approach can be implemented using the in-line assembly language procedures that are required in the language for other purposes. This approach insures that the communications HOL will provide all the power required to do any communications application or executive development.

The low level facilities, in conjunction with the rest of the language especially the strong typing, can be used to build up a library of safer, closer to the problem facilities. The development of such a library should be part of the development of a communication operating system.

This approach to processor management provides the experienced programmer the flexibility necessary to build todays advanced communications systems. It is feasible because access to these facilities is required only by the most experienced communication system programmers. Application programmers would have little need to use the low level primitives directly. They would use only packages tailored for their application by the system programmers.

3.3.6.4.2 Resource Management

For the resource management aspect of process control the communications HOL should provide a compromise approach. This implies the language should provide facilities such as monitors or critical regions, but should allow the detailed implementation of the facility to be specified by the programmer.

The resource management aspect of process control differs from processor management in that access to shared resources cannot be limited to system programmers. Almost every application must access some shared data. The low level approach would not provide enough protection to the application programmer. The compromise approach allows the provision of high level features to the applications programmer while reserving to the more experienced system programmer the ability to determine the appropriate implementation of the resource protection scheme for the project.

3.3.7 Input/Output

I/O issues are at the very heart of any study of communications systems. The reason for this is apparent: by definition, communications systems, particularly message and packet switches, accept inputs from and transmit outputs to a wide variety of devices. Not quite so obvious is the need to generate frequent messages and reports, do ledgering for reliability reasons, and respond to operator inputs. Communications systems are like operating systems in this respect. A third I/O issue, related more to the design than to the function of communications systems, has to do with paging or overlaying techniques. This, too, is a typical operating system concern.

Clearly, the basic communications function requires great flexibility in the I/O support provided by the HOL, for not only is the variety of devices large, it is changing. At the other extreme is the report generation requirement. This implies the HOL I/O capabilities should be powerful, meaning that substantial processing can be invoked through use of very concise commands. Unfortunately, power and flexibility seldom go hand-in-hand. In the communications HOL we must arrive at a satisfactory compromise. The recommended approach is to provide sufficient flexibility for the programmer to create his own powerful I/O capability. The following sections discuss communications I/O requirements. Following this the recommended communications HOL approach is discussed in more detail.

3.3.7.1 Current I/O Requirements

The following paragraphs discuss communications I/O requirements in more detail.

I/O is interrupt-driven and is overlapped both with regular processing and with other I/O. When an I/O operation is completed, the device, or channel, requests a processor interrupt. I/O interrupts are probably the most common form of interrupt in a communications system. I/O, therefore, is intimately related to the process control facilities provided in the HOL. The process control features of the language adequately address those aspects of I/O and so they are not dealt with here.

Actual I/O commands have many parameters such as parity, character code, rate, start/stop bits, turn on/off carrier, maintain/monitor crypto_synch, etc. Frequently, these control settings are established by several commands, each doing one control function. The I/O command structure is very rich and some believe it would be a mistake to depend upon constructing command skeletons because there would be so many of them. On the other hand, it has been possible to devise common I/O facilities which are used by large parts of the system. For instance, communications systems typically do device handling by macros, and I/O is done by common routines which consist of the specific command (read, write, status request, etc.), logical device address, buffer address, lengths, etc. Failure is indicated either by a transfer to an error address or by lack of a positive response in a specified time; the crux of the I/O problem would seem to be in this discrepancy in the ability to use common I/O facilities and the need for rich command structures.

It is also necessary to specify a list of records to be read or written. That is, a data chaining capability is needed. This facility would be useful in optimizing disc accesses, for example. If the architecture permits, a chain of I/O commands may be built up to process the entire linked record list.

It is necessary to output the same information to several devices, e.g., a message may be addressed to several recipients, so that the same message must be sent over different lines to different devices. Similarly, ledgering requires a "shadow" file capability wherein the same information is written to two devices for reliability reasons.

Ledgering is a necessary online activity which does not directly support the communications function. Thus, it should be extremely efficient. Ledgering is a frequent online activity and only minimal formatting is done.

Messages are normally made up of variable length records inside fixed length blocks. Records are frequently delineated by framing characters, but can also have a record length indicator or a pointer to the last valid character. Records are not necessarily packed tightly in a block. That is, non-full blocks can occur inside the message as well as at the end. Non-full blocks have a non-full indicator set.

There is a need to produce offline reports whose format changes as a system evolves. At least two things can be inferred from this. One, that some type of formatting or editing facility would not go unused, and two, that it is not necessary that this facility be extremely efficient since the reports are generated offline.

The system must support interactive dialogues with the operator. Currently, online reports, such as messages to the operator, are handled by getting core, reading in a message skeleton, converting data to character code format, filling in blanks in the skeleton, and issuing the output command.

3.3.7.2 I/O Recommendations

The communications HOL process control features will handle the following I/O issues: allowing a program to "wait" for I/O completion while another program proceeds, or to continue processing, periodically testing for I/O completion; fielding I/O interrupts; and allowing different I/O activities to proceed in parallel.

To handle the remaining I/O issues the communications HOL should allow construction of a set of low level I/O primitives that can be used to create more complicated sequences as HOL subroutines. For instance, in the CPS architecture, I/O commands (and in fact, all inter-unit communication commands) may be treated as pieces of data passed from one unit to another. The low level primitives can be built using inline assembly language procedures. To provide for low level interaction, the basic data structuring facility of a language can be used to build up a data structure representing an I/O command sequence. All that remains is to pass this data structure to a primitive "issue inter-unit command" facility. Having this capability, together with the capability to handle asynchronous events (such as interrupts), the programmer may write programs in the HOL to carry out arbitrarily complicated I/O interactions. In particular, stream and record I/O subroutine packages can be built from the low level primitives and formatted I/O capabilities can be based upon these packages or built directly from the primitives. The linkage and other overhead associated with using subroutine packages will present no particular problems since the programs which would use record, stream, or formatted I/O are, for the most part, low priority and/or low frequency programs. And, if linkage becomes a problem, there is always the possibility of compiling them as inline procedures.

It is important that the HOL provide the necessary primitive facilities in a fashion which would not require recourse to the compiler implementers when new I/O devices are made available. For example, a primitive of the type "issue inter-unit command" or "issue I/O command" which has a pointer to a data structure containing the bit pattern representing the actual hardware command to be executed is a low level primitive. It is obviously extremely flexible and would not require recourse to the compiler implementers to add new devices. On the other hand, commands such as "read" and "write", which in some sense seem fairly basic, would require parameters indicating device type, parity, I/O type, blocking factors, etc. It may be impossible to anticipate all the things which would eventually control the execution of such commands, and thus, addition of a new device would almost certainly require changes in the compiler. Rather than choosing to implement commands which would require many parameters, we've chosen to implement primitives from which the user may build as many subroutines and subroutine packages as necessary to accomplish his I/O tasks.

3.4 Program Development Aids

Several of the topics discussed elsewhere in this paper, such as inclusion of assembly language code, could reasonably be considered as program development aids. This section discusses only those items which did not warrant a separate section. For example, there is a definite requirement to be able to structure systems as independently compiled modules. This is essential for maintainability.

An area which has not yet been addressed is that of conditional compilation. This is a requirement for three reasons. First, this is a powerful tool for customizing general routines; second, it is in keeping with the way many communications systems have been implemented; and third, it is a useful debugging and maintenance aid.

Some means must be provided for isolating common data declarations and macro definitions. That is, it should not be necessary for each programmer using a given piece of data to write the declaration in his program. As a minimum, a copy capability permitting text to be copied into the compiler source stream from a designated external file should be supplied.

It is important that commenting facilities be natural and easy to use. Comments form a prominent part of the documentation for a program and programmers should in no way be discouraged from their use. It should be possible to place comments just about anywhere. As much as possible, errors in comments should not cause errors in the program, for example, by causing code to be interpreted as part of the comment. Comment design should support the use of other tools, such as a listing formatter and a tool which would extract comments for documentation purposes.

Compile time procedures or macros are not required in the communications HOL since user defined types and inline procedures provide the essence of what macros are typically called on for.

PART II

RATIONALE FOR BASE LANGUAGE SELECTION AND PROPOSAL FOR MODIFICATIONS

Selection of a base language for the communications HOL was performed on a dual basis. One basis was technical. The five candidate languages -- CS4, J3, J73, PASCAL, PL/I -- were thoroughly analyzed using the Fort Monmouth Language Features List, an exhaustive encyclopedic summary of features of many existing higher level languages. This characterization of each of the languages was compared with a standard hypothesized as a result of a year-long study of communications programming requirements. The deficiencies of the languages were analyzed and modifications to each studied. The technical conclusion reached was that the extreme range of communications programming requirements makes all of the existing languages unsuitable without modification. One language, J3, was determined to be unsuitable both without modification and as a base language.

Selection then proceeded on the second basis, or on managerial grounds. This considered factors such as the short-term usability of one of the unmodified languages and the feasibility of performing required modifications purely as additions to the existing language. J73 was determined to be the most usable of the unmodified languages and found to be amenable to change strictly by addition. Additional requirements releated to long-term usability of the modified language, and its contribution to the reliability and maintainability of systems programmed in the modified language, were also analyzed and determined to be satisfied by the strategy of additions to J73.

The following report documents the language analyses and technical aspects of the considerations of short and long-term usability of the communications language.

Section 1 discusses the candidate languages using a cross-section of feature areas:

- data structuring and storage management
- process handling
- I/O control
- system integration
- control structures
- data type usage

Language features in each of these areas are evaluated according to three factors:

- capability -- ability to perform necessary functions
- efficiency -- ability to perform these functions efficiently
- control -- ability to specify desired restrictions

For each of the areas and factors, particular communications requirements are given special attention. In data structuring, the flexibility to do both low-level storage allocation and very high level typing is required. In process handling, the ability to control the mechanization of synchronization must be combined with facilities that logically group cooperating processes and shared data. In I/O handling, low-level features such as buffering and device synchronization are essentially all that is required in the on-line environment. (Off-line I/O is not truly a communications language issue since minimal efficiency requirements enable this function to be performed by closed procedural extensions.) In system integration, the ability to specify common bindings across separate compilation units and control the use of data and procedures over large systems are required. The control structures analysis reflects the need for a complete set of "structured programming" constructs and special control forms for analyzing character valued objects. Finally, the analysis of data type usage notes the hypothesized need for eight specific value types and their operations. The operations of assignment and parameter passing are isolated in order to make important comments on their efficiency characteristics.

Section 2 summarizes considerations related to management factors, such as usability of the current languages and projections of the reliability and uniformity of the modified languages. The basic conclusions are that while PL/I contains the most functional capabilities, it lacks crucial machine dependent features and contains inherent inefficiencies which make it currently unusable for communications. J73 is almost as powerful functionally as PL/I, is by far the most efficient of the languages, and is certainly the most appropriate of the unmodified languages for communications usage. CS4 and PASCAL are unusable in their current state, CS4 containing the most features related to reliability and PASCAL providing the best model of uniformity and simplicity. For the long term, the factors of uniformity and consistency are surely the most important in terms of controlling life cycle costs of systems programmed in the Communications HOL. All the languages can be modified to contain the desired functional and reliability-fostering features. While CS4 is itself an "extension" of PASCAL, it makes some crucial decisions, such as the abolition of pointers and variant structures, which are opposed to the efficiency needs of communications programming. For the long term, then, PASCAL provides the best basis for a communications language.

As a way of dealing with both the short and long-term needs of communications programming, then, a compromise approach is to modify J73, improving both its capabilities and its consistency. Section 3 describes this strategy. The approach will be to add the (few) functional and (many) control-related features required to meet the extreme range (very high level security and very low level flexibility) of communications requirements. In addition, duplicate features for existing inconsistent constructs will be added when the parallelism does not produce ambiguities. A way of restricting programs to use the more consistent set of constructs will be provided (perhaps by an additional keyword preceding programs in the consistent subset).

Table 1 summarizes the evaluation, the language feature areas and usability ratings. Each factor of each feature and usability area contains a ranking of the candidate languages, with the most satisfactory language occurring first. Some languages are not rated in some of the areas, where no support occurs or a

rating is inappropriate. When language names occur grouped in parentheses, the languages are judged to be equivalent on the factor in question.

Table 1

SUMMARY OF EVALUATION FACTORS

Data Structuring and Storage Management		
	capability	(J73, PL/I), PASCAL, CS4, J3
	efficiency	J73, PL/I, PASCAL, CS4 (J3 not rated)
	control	Cs4, PASCAL, PL/I, J73, J3
Process Handling		
	capability	CS4, PL/I, PASCAL (J73, J3 not rated)
	efficiency	PL/I, CS4, PASCAL (J73, J3 not rated)
	control	PASCAL, CS4, PL/I (J73, J3 not rated)
I/O Control		
	capability	(PL/I, J73, CS4), PASCAL, J3
System Integration		
	capability	(J73, CS4), PL/I, J3 (PASCAL not rated)
	efficiency	J73, PL/I, J3 (CS4, PASCAL not rated)
	control	CS4, J73, J3, PL/I (PASCAL not rated)
Control Structures		
	control	CS4, PASCAL, J73, PL/I, J3
Assignment and call		
	efficiency	PASCAL, CS4, J73, J3, PL/I
Data type usage		
	capability	PL/I, J73, PASCAL, CS4, J3
	control	PASCAL, CS4, J73, PL/I, J3
	efficiency	J73, PL/I, PASCAL, J3 (CS4 not rated)
Usability		
	existing languages	J73, PL/I, J3, PASCAL (CS4 not rated)
Uniformity and simplicity		
	existing languages	PASCAL, CS4, PL/I, J73, J3
Reliability		
	existing languages	CS4, PASCAL, J73, PL/I, J3

SECTION 1

LANGUAGE FEATURE AREAS

In the data structuring area, functional capability has to do with the ability to perform storage management to any required level of detail. Heap storage with programmer controlled allocation is one way of achieving a large amount of this kind of flexibility. Based storage, or the ability to interpret any appropriately large section of memory as having a specified structure, is another way. The ability to base a structure is the most general form of data structuring and can be used to implement the heap form of dynamic storage management. J73 and PL/I have based storage and so must be listed as having the most flexibility. PASCAL has heap storage without programmer control, and so has inherently less flexibility. What is missing is a way to perform the lowest level of storage allocation; an untyped pointer and the ability to associate a type with a tailored allocation action are needed. J3 has neither heap nor based storage.

The efficiency requirement in data structuring has to do with the ability both to express all possible storage management requirements and to incur the minimum required overhead. J73 comes closest to meeting the overall objective of this requirement. Its mechanization of based storage as a uniform property of all data structures, including blocks and procedure local data spaces, allows storage of every kind to be managed efficiently. PL/I has somewhat less capability, since it does not allow machine specified packing. PASCAL has minimal efficiency in this sense, since its heap storage management is supported by built-in routines. CS4 has an unknown amount of flexibility, since its heap storage characteristics are not specified or implemented. J3 has no dynamic storage capability.

The rating of storage efficiency is done primarily for dynamic storage, since all languages have functionally adequate structuring of static storage, in the sense of flexibility to express any static configuration (possibly with programmed allocation); and the efficient implementation of access to them is essentially a compiler issue. Some intermediate issues concern storage which is dynamically allocated on a stack or formal parameters whose length is determined by that of their corresponding actual parameters. In these cases, including essentially character strings and arrays (possibly embedded in structures), the important efficiency consideration is that the mechanisms for handling the variable length forms not affect the handling of fixed length character strings and arrays. Thus, a compiler which uses dope vectors, for example, to access arrays whose bounds are not known at compile time should not use them to access arrays whose bounds are known. Again, this is a compiler rather than a language issue.

The control issue in storage allocation has to do with the ability to insure correct access to data (i.e., avoid meaningless data references) as well as to protect data from being read or written without authorization. This is essentially the ability to restrict data usage, and so in some ways is the reverse of the capability and efficiency issues. This is true of all the candidate languages because in each of them one approach -- heap storage with typed pointers or based storage with untyped pointers (or integers as in J73) -- is

taken exclusively. CS4 and PASCAL emphasize control, CS4 to the extent that pointers do not even appear explicitly. CS4 also allows access and write operations to be defined on structured data, which gives both more control and more flexibility, since such procedures can both restrict and tailor the meaning of comparison and assignment. The ranking for control, then, is the reverse of the flexibility and efficiency rankings, except for CS4 rated both more flexible and more controlled than PASCAL.

It is important to note that no language is adequate in the data structure area, primarily because all take a single approach -- emphasizing efficiency or emphasizing safety. One of the significant conclusions of the communications programming study is that both approaches are necessary. Extended types provide desirable control and access restrictions, while the ability to program heap storage allocation using untyped pointers provides lower level flexibility and efficiency.

An interesting view of the possibilities of the HOL capabilities is provided by the special purpose data structuring facilities of PL/I: controlled storage and the use of offset pointers with based storage. Controlled storage is a built-in stack data type. It provides evidence of the possibilities to be achieved by enabling special purpose data structures with specifiable access functions to be defined. However, it is only one mechanization of one common special purpose type. Real flexibility would involve the ability to define somewhat different stack operations, perhaps with assignment interpreted as copying a DEC-10 style stack pointer and with specific push and pop operations. More flexibility would involve the ability to define lists, queues, buffer pools, and so on in the language. Offset pointers are another example of potential flexibility. Essentially these can be achieved by read and write functions defined for each component of a data structure, which add a base pointer to the value of a typed offset pointer as part of accessing the specified component.

Process control features of the reference languages are difficult to evaluate. For these features, functional capabilities and notational appropriateness are eclipsed by performance factors. Flexibility, having to do with the ability to perform a wide variety of real-time functions, is meaningless without efficiency. CS4 has a full set of high-level primitive operations (schedule, terminate, seize, release) on processes, and somewhat lower-level synchronization operators (shared data attribute, event data type). PASCAL, in the concurrent form used in this study, includes a monitor process handling cluster, which is the highest level concept available. PL/I has a more fully tested set of essentially lower level operations, including task initiation procedures, event variables, and wait statements. The ability of any of these features to achieve acceptable performance in communications programming is not known. In general, the lower the level, the more confidence is available that the operation is appropriate, so the flexibility and efficiency ranking on existing capabilities would rate PL/I higher than CS4 and PASCAL with J73 and J3 unranked. The control ranking would be the reverse.

The hypothesized communications language capability involves low-level operations achieved by machine language encapsulation, along with an ability to collect sets of procedures using similar operations and sharing data into

types of cooperating processes. CS4 comes closest to providing this capability, since it supports a shared variable attribute and encapsulation of machine code. On the other hand, the ability to collect sets of cooperating procedures resembles the monitor feature of Concurrent PASCAL, modified to provide some degree of flexibility in the mechanization of synchronization. The designation of procedures as tasks is similar to the PL/I capability, and the hypothesized low level of synchronization primitives is also similar to PL/I (though inherently more flexible than the single event type of PL/I). In short, the desired process handling capability is a collection of characteristics of all the candidate languages. The ratings on process characteristics are not particularly significant, since all the languages require modification to achieve the hypothesized efficiency, flexibility, and control.

I/O in communications is essentially an amalgamation of storage control (buffering), process handling (synchronization) and machine dependent data structuring abilities (ability to specify any type of packing, and to perform bit level operations on words and subwords). These are capabilities that are dealt with elsewhere. Note that all of the reference languages have serious deficiencies here:

	Storage Control	Process Handling	Machine Dependent Structure	Bit Operations
PL/I	Yes (pointer)	Yes	No	Yes
CS4	Yes (files)	Yes	Yes	No
PASCAL	Yes (files)	Yes (read/ write operations)	No	No
J73	Yes (pointer)	No	Yes	Yes
J3	No	No	Yes	Yes

Essentially, then, PL/I, J73, and CS4 must be rated as giving approximately an equal amount of capability, with PASCAL and J3 having successively somewhat less capability. Again, as with the process control area, the ratings are not highly significant, since none of the languages provides the complete set of features needed.

In the system integration area, functional adequacy has to do with the ability to perform a variety of bindings and specify relationships between modules at compile time. Efficiency has to with the ability to avoiding unneeded processing in performing bindings. Control has to do with the ability to limit access to bindings that are needed, as opposed to having access to all information in a separate unit, even information that should be protected. The ability to construct a system using separate compilation units is central to communications programming and occurs in CS4 (ACCESS), J73 and J3 (COMPOCL), PL/I (EXTERNAL and %INCLUDE). The ratings for CS4 and J73 are high, since they support separate compilation units as well as controlled access. In CS4, binding of subroutines and data across separate compilation units is based on complete parameter and attribute matching, so the control rating for CS4 is the higher of the two. The efficiency aspects of the CS4 method are not known,

since no implementation exists. In principle, a symbol table output from compiling a CS4 module could be arranged in a form suitable for efficient use by other compilation units. The J73 COMPOOL mechanism is a significant model for efficiency in providing these integration mechanisms. Of the other reference languages, J3 and PL/I have essentially functionally equivalent capabilities, both supporting the ability to bind enclosing declarations without rigid control. PASCAL provides no separate compilation unit capability, and no special aids for compile time module integration (which would provide the basis for some rating in this area).

In the area of control structures, the rating is primarily an historical one. Since essential agreement in the field exists about desirable capabilities, at least from the semantic point of view, the ranking is quite simple. CS4 has a complete set: case statement, loop statement with all common control forms including a leave feature, conditional statements, and compound statements. PASCAL has less loop control. J73 has a less well-structured case statement and less loop control. (Its conditional statement is somewhat eccentric in not having a THEN keyword for the first alternate, but that is a minor syntactic point.) PL/I and J3 have no case statement and less loop control. The ranking is strictly on control, since all the languages have an equal amount of functional capability. Efficiency is primarily a compiler issue (the differences occur in areas such as allocation of loop control variables to registers). The ability to program loops using character variables, as required in communications, is supported only by PASCAL.

Assignment and parameter transmission are properly thought of as operations associated with data types to which their actions apply. However, the assignment operators for the various data types and aggregates have properties in common that are worth factoring out. Similarly parameter matching has common characteristics among all the data types of a language, many of them in common with assignment (at least in languages with that degree of consistency and uniformity -- this will be dealt with in the uniformity section). Conversion is one important difference -- with implicit conversion and strict matching of types complementary aspects. As a reliability issue this properly belongs under the control of each data type, since it has to do with limiting the values a variable can assume. As an efficiency issue it can be extracted and isolated for rating purposes. The ability to invoke extra conversion actions implicitly is an undesirable characteristic of both J73 and PL/I. In PL/I these conversions can be quite complex. For aggregates, a significant efficiency issue on assignment and parameter passing is the extent to which copying of large amounts of information can be implicitly invoked. Passing array parameters by value is one way this can be done; none of the candidate languages do this (except CS4 under specific programmer control). Compatible array assignment and BYNAME array assignment in PL/I however have similar properties; they are special purpose multi-word assignments potentially invoking indirectly quite complex selection and assignment code. PL/I then ranks lowest in terms of efficiency in this area. Requirements in CS4 for performing run-time range checks on assignment and parameter passing, without any mechanisms to control their use, have the same type of implicit effort.

Another difference is in the area of control of parameter transmission. CS4 allows specification of the transmission mechanism of each parameter.

PL/I on the other hand passes all parameters by reference. This is inherently inefficient for single word data items on a machine with enough registers to allow passing such parameters by value. J73 passes these parameters by value, but requires assignments to and from input and output parameter locations at procedure boundaries, even when the value may never be referenced other than in a register. This is particularly inefficient on machines with a limited number of registers (so that some parameters have to be passed in an argument list) and with an indrect mode available for the instructions used to address the parameter. Particularly for output parameters with a limited number of references, passing by reference is more efficient. Implicit conversion affects this issue also, since conversions of output parameters cause code to be implicitly generated in the context of the call. In PL/I and PASCAL, no mechanism is provided for distinguishing between input and output parameters. This means possible efficiencies related to the ability to save storing an input parameter which cannot be modified inside the procedure body cannot be realized. J73, of course, also does not strictly allow these optimizations to be performed, since the language specification requires stores, and input parameters can be modified. However, the possibility exists for an implementation to keep values in registers, using information the compiler gathers itself, and achieving the same semantic effect. PASCAL can perform the same optimization, without violating its language definition document.

In the area of data type usage, the types of values required are integer, floating point, status, bit, boolean, character, procedure, and pointer. None of the languages has all the types. The following table indicates which languages have which type (P indicates partial support).

	In-teger	Float-ing	status	bit	boo-lean	charac-ter	pro-cedure	pointer
J73	+	+	P	+		+	P	P
CS4	+	+	+		+	+	P	
PL/I	+	+		+		+	+	P
PASCAL	+	+	+		+	+	P	P
J3	+	+	P		+	P	P	

CS4 and PASCAL have the set type instead of a bit type. PL/I has integers as a subset of fixed point, which has undesirable effects for integer division. CS4 and PL/I have varying length character strings, which represent excess capability. The status types of J73 and J3 are incomplete in that only status constants are supported. Status variables are actually integers and can assume any integer values. PASCAL has the most control related features, all in essentially desired form. CS4 and PASCAL support a range attribute for numeric data, CS4 somewhat more consistently. None of the languages support the multiple character set facilities required for communications.

The ranking for capability essentially emphasizes the basic computational abilities. PL/I and J73 both allow all operations, by virtue of their numeric types (which all the languages have) and their bit and untyped pointer types.

PL/I includes a full procedure type, and so ranks higher than J73. The partial status type possessed by J73 actually adds nothing in capability over PL/I, since everything (functionally) that can be done with status variables can be done with integers. PASCAL and CS4 rank lower primarily due to lack of low level facilities connected with absolute pointers and the bit data type; the expectation is that considerable circumlocutions are required to perform with the set data type all the functions for which the bit data type is currently used. The lack of a boolean type in PL/I and J73 is not a serious problem functionally, since the bit data type can (less clearly) perform the same functions.

The efficiency of data usage has mainly to do with allocation and this is a complex area to use as a basis for evaluating languages. One of the simplest types of difference has to do with local data. The area is worthy of comment since J73 allocates local storage for internal procedures, at least in the case of based procedures, in the enclosing non-internal procedure. A language feature supporting this directly would be an "INTERNAL" attribute. The same capability is supported, without an explicitly declared attribute, as an optimization in MULTICS PL/I.

Other factors than local data space allocation include implicit register usage, such as for the procedure call stack and for global data reference. The one disadvantage of the J73 approach to based local data is that the register used for the base must be explicitly loaded at the point where the enclosing non-internal procedure is called. This loading may often be superfluous, since the likelihood is that the pointer variable used for the local data is the same as the one used in the calling program (except for being incremented, which can usually be done most efficiently in a register anyway). This register is, for reasons of minimizing the number of registers used for special purposes, often the same as the call stack, and is the basis for a PL/I-type automatic storage mechanism. The use of this type of mechanism is important to efficiency, in both time (in terms of the efficiency of a single dedicated register) and space (which is the primary reason for non-static storage of any type).

The efficiency issue for global data has to do with overhead at procedure entry and exit for establishing addressability to both static and dynamic global data. In the case of static data, this has to do with base and/or relocation registers. Language features which support partitioning of global data are important here. Traditional features are common block and named block capabilities. Another type of capability, not available in any of the candidate languages, is the ability to declare all data in a desired page, say, using attributes associated with each data declaration.

The issue for global automatic data has to do with the amount of support provided for reference to variables in enclosing scopes. All the reference languages support such nesting. A language without separate compilation units, such as PASCAL, can provide a complete modeling of the static environment of a procedure at run time, in terms of a display maintained at procedure entry and exit. This is inefficient in a situation with very few references to global data, since the overhead of updating the display occurs at every procedure call. The MULTICS version of PL/I maintains a local variant of the display for separate compilation units, and minimizes actual need for it by identifying internal

procedures and allocating their data globally, so that all procedures in a compilation unit potentially can use the same stack pointer. J73 supports the ability of the programmer to control this by use of based storage. Another way to control the overhead associated with global variable access is to limit the ways of referring to it. None of the reference languages do this, but it is optionally supported for the communications HOL, both for clarifying global data needs and assisting the compiler to optimize references to them.

SECTION 2

USABILITY, UNIFORMITY, AND RELIABILITY

Usability of the unmodified languages is a complex subject, since all the languages require significant deletions and additions. The functional aspects of communications programming are of course achievable in some sense in all the languages, provided some powerful low level extension mechanism such as assembly language inclusion is available. J73 and PL/I both provide general data structuring capability and essentially all required control capabilities, in the purely functional sense. J73 has slightly higher efficiency ratings (other than in the process control area, where PL/I is not completely adequate) and so should probably be rated the most usable. J3 and PASCAL are much less usable in the basic sense, lacking the bit data type and an absolute pointer. PASCAL lacks support for separate compilation units, and so may be the least usable. CS4 would perhaps rank ahead of J3 and PASCAL if it were implemented, particularly since it already supports assembly language encapsulation. However, being unimplemented and hence untried, CS4 must be considered essentially unusable in its current state. A more appropriate way of dealing with CS4 is simply not to rank its usability.

The issue of life cycle costs related to use of one of the modified languages is similar for them all. The modified languages are essentially defined by their meeting the requirements of the communications language hypotheses. They all contain the features which support program reliability, modifiability, and understandability. These include defined types, constant names, strict type checking in all operations, no implicit conversion except numeric constants, and so on. Due to the large number of changes required to all the languages, the main difference in life cycle costs will be a consequence of the uniformity of the modified languages. The primary differences in observed error rates should occur due to difficulty in learning the language, writing code in it, and reading produced programs.

The language most likely to give a uniform set of syntax and semantics after modification is probably the language with the most consistency unmodified. The process of adding data types, data structuring facilities, and process handling capabilities can and should be done within the framework of the base language. Since the uniformity of the modified languages is primarily a function of the existing languages, the lesser lifecycle costs should be incurred by the most uniform languages.

The uniformity and simplicity factors related to this requirement are really quite straight forward to apply. PASCAL is the simplest and most uniform of all ALGOL variants. It has been the subject of very few complaints or problems. Samples include: the use of square brackets for array indexing, the inconsistent treatment of integer subranges as types, its numeric labels, and its nonuniform handling of structures (not allowed as values of functions). CS4 has an unknown amount of inconsistency, since very little experience has been gained with it. It was designed with many of the known problems of PASCAL in mind, and so presumably should be better in some sense than PASCAL, in areas where its features are derived directly from those of PASCAL. On the other hand, many CS4 features are not so derived, and at the very least contribute to an increased complexity

that makes the language potentially harder to use. PL/I has a number of semantic inconsistencies, mostly related to its implicit conversion philosophy and essentially arbitrary meanings attached to some of its operations (e.g., array operations and comparisons). PL/I also tends to confuse with its large variety of special purpose operations, built-in functions, factored declaration capabilities, and treatment of integers as fixed point numbers with zero scale. J73 has a number of syntactic anomalies and non-uniformities in the way it performs qualification (indexing, component selection, procedure call). Its storage allocation capabilities and argument transmission semantics are overly elaborate and somewhat arbitrary. As opposed to PL/I, the inconsistencies of J73 occur in areas that must be carried through in the modified language.

Reliability is the final area of language evaluation. It is related to the control capabilities of each of the language feature areas. The ability to specify storage, program control, data structuring and usage exactly is tantamount to the ability to detect or avoid errors. The control features of CS4 and PASCAL include the range attribute for numeric variables, the enumeration type for exact treatment of finite sets, a mostly complete set of control structures, and fairly uniform control of operations defined on each data type. Some features that even CS4 lacks which should be included in the communications language are the ability to specify that global variables are read only in a scope, and to express read/write access restrictions on components of a structure (as part of an extended type). This is not as strict an approach as that taken in the Ironman DOD1 language requirements and in Euclid, where access to global variables must be requested in each scope. The communications language approach is more that desired restrictions can be expressed, but they are not required. No language can provide complete protection; languages can only promote reliability.

Safe ways of effectively and efficiently doing low level storage management, i/o functions and process communication are not known; the goals of safety and efficiency are perhaps more intrinsically opposed in these areas than others. Also, truly structured ways of dealing with exceptions to normal control sequences are in some ways inherently opposed to the principles of regularity in existing programming languages. Extensible control structures have often been proposed to assist in handling some of the less global aspects of special control situations, but this approach presents extraordinary optimization problems. The procedure, of course, is the basic extensible control mechanism, and the problems of optimizing over procedure invocation (i.e., open expansion and allocating data in enclosing non-internal procedures) are well-understood. What is lacking from a "control" point of view (both in the sense of enabling the programmer to express exactly what is intended and in allowing errors of misuse to be detected) is the ability to characterize a procedure as a special type. To some extent, this is supported by encapsulation devices and monitors, where procedures are identified as operators on specified data structures and as requiring special enabling actions or entry conditions. The CS4 approach to exception handling has some of the same flavor, where procedures are identified as exception handlers and invoked by special call "signals". The communications HOL will take a similar approach to true exceptions, and will allow situations which are simply exceptions to normal local control sequences to be expressed by procedures with other sorts of type identification. This is a potentially highly usable and appropriate way to promote reliability in areas where none of the candidate languages provide support.

The actual ranking of support for reliability among the candidate languages is essentially a summary of the "control" ratings in the language feature areas. CS4 has more "first" control ratings than PASCAL. PASCAL has the remaining "first" and all the "second" ratings in those areas where CS4 is first. J73 ranks ahead of PL/I in control structures and data types, and is almost equivalent to PL/I in the data structuring area. J3 is last in all control categories.

SECTION 3

MODIFICATIONS TO J73

The extension of J73 as a communications language first involves providing basic data and procedure extension mechanisms to allow specification of properties important to communications programming. It also involves purely functional additions to existing character and procedure data types, to handle multiple character sets, character loop control, and procedure dispatch tables. A final aspect of the modification strategy is to improve the reliability and usefulness of existing capabilities. Such additions will improve the readability of programs (by using named constants and restricting where declarations can occur), minimize the possibility of errors (by identifying potentially dangerous transfer of control and potential misuse of data), and improve the usefulness and generality of compile-time evaluation capabilities.

The principal functional additions to J73 can be summarized by the following list:

- ability to define named types which identify classes of table entries with limited access characteristics
- typed pointers to reference typed objects safely and efficiently
- heap storage with programmer control over allocation functions, to permit flexible and controlled storage management
- new capabilities for table entry declaration to support complete description of structure, including variant fields, nesting of tables, tight packing, and packed arrays
- procedure variables to allow use of procedures as table entries
- procedure attributes to allow specification of properties such as reentrancy, recursion, interrupt handling, task initiation, exception handling and synchronization at entry and exit
- new capabilities for the character data type, including the ability to use more than one character set, to represent special characters, to treat the length of an actual parameter as the length of the corresponding formal, and to specify the length of a dynamically allocated character string when it is allocated

- ability to specify the transmission mechanism for a parameter, including whether by value, reference or result and which target machine register is used

- ability to do open or inline calls of procedures, and to specify procedure bodies in assembly language

- ability to use character variables in case indexing and loop control

- absolute allocation attribute for procedures

- ability to nest COMPOOL's (use COMPOOL directives in COMPOOL modules) and to collect type definitions, shared variable declarations and related procedure declarations into clusters in COMPOOL's

- shared variable attribute and access blocks for support of exclusive access to shared resources and for implementing critical regions

- ability to use the bounds of an actual parameter for those of the corresponding formal parameter

- ability to specify the bounds of a dynamically allocated table by expressions evaluated when the table is allocated

- ability to specify a function used to convert from an integer to a status representation

The remaining recommended additions can be thought of as application independent but as helping to realize a number of valid programming goals. These features assist in the following areas:

- improving readability of programs;

- assisting in detecting or avoiding common programming errors;

- allowing machine characteristics to be parameterized, so that programs are more portable;

- simplifying and controlling modifications to programs so that they can be reused.

Constant names are the best example of this sort of feature. They help in all the application independent areas. They improve readability by allowing meaningful mnemonics to be used instead of unexplained or magic numbers. They control errors by defining a value in one place and associating a single complete

set of attributes with it. As possible machine dependent values, they enable machine dependencies to be characterized. As common values defined in a single place they enable changes to be automatically propagated to all statements using a value, and so foster modifiability.

A complete list of such recommended additions is the following:

- constant names for all built-in types
- Boolean data type to replace use of bit values in conditionals
- range attribute for numeric data
- ability to specify a variable read-only in a scope
- ability to perform the termination test during iteration anywhere in a loop body (at the end, corresponding to a REPEAT-UNTIL capability; or in the middle, corresponding to a LEAVE or EXIT capability)
- a complete set of built-in machine dependent parameters
- reserved words to refer to all the declared attributes of a variable of any built-in type (similar to LBOUND, UBOUND, and SIZE)
- constant evaluation for built-in operations on all built-in types
- reserved words for all hardware exception conditions and for optional software conditions such as range violations, length differences, and conversion errors
- a uniform notation for enabling or disabling software exceptions at the statement and program module level

Promoting reliability is a final facet of improvements to J73. An approach to obtaining the error detection properties that correspond to restricting a typeless language such as J73 is to provide controllable warnings. This mechanism can syntactically be similar to the notation used to control exceptions, and should be capable of controlling warnings at either the statement or module level. Such warnings would achieve the effect of detecting statements using implicit conversions, restricting where declarations can occur, and avoiding potential errors in referencing data or modifying control. In modules or statement where an error prone construct was desired, a programmer could use a control, identifying warnings or classes of warnings to be detected or ignored.

Some of the restrictions that can be dealt with in this way are:

- declarations must be grouped

- declaration before use is required
- external declarations can occur only in COMPOOL
- blocks can occur only in COMPOOL
- the following must match exactly in type (except numeric literals):

  operands of an expression

  source and target of assignment

  external name with declaration in COMPOOL

  actual parameters of a procedure call with corresponding formals (where the procedure is local or declared in COMPOOL)

- an integer declared with a status type name is the the target of an assignment statement where the source is not an integer identified with the same status type name, or such an integer is a formal parameter where the actual is not of the same status type
- a constant is too large to be a substring or subscript index, or is outside the range of the target of an assignment or the corresponding formal parameter
- no references can occur to labels inside loops and case statements
- a DEFAULT is required on SWITCH statements if the range of possible index values is not covered
- no fall through is allowed between SWITCH alternatives
- LOC cannot be applied to automatic storage
- no duplication of actual parameters is allowed
- structure of table entries must match on assignment and comparison
- assignments to static storage are not allowed in reentrant procedures
- SWITCH index values must be explicitly specified
- loop increment value must be constant

- substring index must be constant
- reference to global labels or label parameters is not allowed
- address formulas can be used only for explicit dereferencing of based storage
- named RETURN is restricted to exception handling procedures
- an integer used as a pointer for based storage is multiplied or divided by a non-constant expression
- initialization of variables is required

Finally, some language features aid in the ability of a compiler to optimize the generated code. Some of these give information that is not otherwise available to the compiler or is very difficult to construct. Examples of this type of feature are:

- a TO form of loop which implies the value of the loop control variable is read-only and is not used outside the loop;
- procedure attributes indicating that special forms of call can be employed (such as an INTERNAL attribute to allow local data to be allocated in the enclosing stack frame, or a TERMINAL attribute to allow not saving the return address or to enable use of an execute capability); other procedure attributes such as INLINE and INTERRUPT, also improve a compiler's optimization possibilities.
- specification of parameter transmission methods, such as a "BY REFERENCE" attribute to allow "inout" parameters to be efficiently implemented in J73.

Other efficiency related features are actually compiler capabilities, but should be part of the requirements for a language. These include:

- constant expression evaluation for all built-in operations on all built-in types
- performing conditional compilation as a consequence of constant expression evaluation of boolean expressions in IF statements and substitution of INLINE procedure bodies.
- in general, completely specified compiler optimization capabilities.

The ability to make use of a language in a real time critical area such as communications will ultimately depend greatly on the quality of code generated by its compiler. When these kinds of capabilities are not part of every implementation, they are likely to be less used in those implementations that do support them. Programs tend to be distorted in the direction of the minimal capabilities of a language, and methods used on one implementation to circumvent poor quality of code will generally be used on others. The tendency of non-uniform optimization is to degrade readability and maintainability of programs.

Complete discussion of all these features is not possible within the scope of this effort. A language reference manual is one of the prime outputs of the second year of effort, and will include complete syntax and semantics for the language.

The modifications to J73 discussed here describe essentially requirements for the final language and indicate the source of the requirement. This is only a summary list. Further justification for some of the more important features are given below, associated with each feature or group of features.

- Ability to define structures using named types - referenced using typed pointers, allocated using heap storage with programmer controlled allocation, and accessed using programmer defined read and write operations. Tables are used as the primitive structuring notation and attributes for limiting access to objects and their representations are available.

  This is the basic mechanism for describing and controlling dynamic storage use. The current capabilities of J73 are functionally adequate, but are inherently less reliable and secure. Because J73 pointers are integers, they provide no safeguards on use and access to dynamic storage. Because structure cannot be fully described in one place and mechanization of access can (and sometimes must) be dispersed, an implementation cannot be developed, understood, maintained or modified as easily as with defined types. The proposed set of capabilities provides a minimal and well-understood capacity for the special storage management strategies needed for communications programming, such as buffer pools and queues, and provides a framework for maximal protection.

- Optional range attribute for numeric variables.

  Knowledge of the range of values of a variable is important both for representing it (its size) and for controlling its use. Range checking can be rigidly enforced for debugging and testing purposes, or can be suspended for efficient run-time operation with much meaningful

range checking still performed at compile time. Ranges are naturally associated with subscript and loop control uses of integers, as well as being important information about application data. Size alone as an attribute simply gives less information.

- Boolean data type

  It occurs implicitly in J73, in that the use of the right-most bit in conditional statements insures minimal semantic impact from interpreting relationals as Boolean values. Booleans are easily added upward compatibly, and constitute an improvement in precision and clarity. Short-circuited evaluation of Boolean expressions should be required of any implementation, as an optimization and as an efficient way of controlling evaluation to avoid undefined evaluations or references.

- Completion of the procedure data type

  Procedures can occur only as parameters in J73. Procedure variables, primarily as table entries, are required in communications, for example, in dispatch tables.

- Additional attributes for procedures

  A complete set of attributes for procedures, from well-understood ones such as reentrancy to less commonly definable procedural properties such as interrupt handling and process initiation, should be expressible in a communications language. Semantics should include control over the implementation of argument transmission, local storage allocation, call mechanization and entry/exit actions. The based allocation attribute of J73 procedures provides a notation only for storage allocation, and provides no control over its use in connection with specification of the type of operations which procedures with this attribute implement. The point is that a base for a procedure local data space is a second-level property such as interrupt level or owning process.

- Control over parameter transmission

  The transmission of actual parameter values is one of the most time consuming aspects of procedure invocation. In J73, input parameters are stored in local storage, whether the action is significant or not. Output parameters are returned by result and

may be converted implicitly, requiring a store action in the context of the call, after procedure completion. There should be in a communications language a mechanism for passing by reference, to save having to store parameters, when the action is not semantically significant. There should certainly be a default transmission convention but one that provides a more reasonable compromise of efficiency and safety, depending on the target machine and possibly on what is required for "most" programs in a particular application system. (J73 provides the basis for a reasonable treatment of overlaps of parameter space and global data, in terms of the INTERFERENCE directive.) There should also be a way of overriding the default, controlled by procedure attributes, such as ones designating sets of application and operating system procedures, with similar parameter transmission requirements. In addition, there should be a way of describing the transmission actions (by reference, by value or by result, in a register or in an argument list) for every argument, in cases where tuning might be required by means of special calling sequences, or where the ultimate implementation of a procedure is in assembly language.

- A unified exception handling mechanism to replace global label references and label parameters

  The use of labels which are parameters or are global to a procedure containing a branch to them is to terminate a processing sequence when the sequence should not continue. An exception handling facility can thus replace these uses of labels. Exception handling notations provide more information than the label method. Knowledge of both the reference and desired recovery and resumption actions must be present at the point where the label is defined. Encapsulating this knowledge in a special exception handling procedure is a way of documenting its function, and additionally provides a natural mechanism for passing parameters (otherwise, global variables are required) and for returning control to the invoker or enabler via special return actions.

- Unions within structures, in the form of variants at the end of structure representations, discriminated by programmer specified tag fields.

  This is the minimal support for discriminated unions. Different spellings should not be required for all field names of variants.

- Ability to nest tables within tables.

  This would give some expressive capabilities that do not currently exist. It would provide the ability to include arrays as structure components.

- Built-in exception names for all hardware exception conditions, and subscript range violations, type and range incompatibilities, and conversion errors.

  Programmers should be able to control any software exceptions and cause or avoid run-time testing, by specifications at the statement and program module level.

- Inline assembly language procedures

  These are required for performing low level operations efficiently. Their use should be controllable, possibly by restricting their declaration to COMPOOL.

- Capabilities for shared variable protection and critical region support.

  These can be achieved by access blocks and support for an access attribute. For shared variables, the attribute is connected with each data declaration. For critical regions, the attribute is connected with potentially several variable declarations.

- Requirement that constant expression evaluation to use target machine arithmetic.

  This insures that optimization performed by constant expression evaluation is valid, in that it is the same as run-time evaluation.

- Machine dependent parameters for maximum and minimum values of numeric data types and other built-in type attributes.

  This provides a way of parameterizing initializations, loop bounds, range checks, and other constructs, in a portable way.

SECTION 4

SUMMARIES OF DIFFERENCE ANALYSES

The descriptions of each of the candidate languages in the language feature data base were compared, feature by feature, with the description of the hypothesized ideal communications language. These listings were used to isolate characteristics of the candidate languages in the feature analyses of Section I. They also provide evidence of the pervasive assumption that all the candidate languages require significant changes. Since the difference listings are quite lengthy, they are summarized here to clarify their main points.

I. JOVIAL/J3

Additions to J3 include pointer, bit, and procedure data types, procedure attributes, more loop control, and a number of refinements to tables. Major additions include dynamic storage, data type extension, and support for process control and exception handling. Deletions from J3 include implicit conversions, the index form of the switch statement, fixed point real values, label parameters, and files.

The following list summarizes the detailed changes to J3:

add named constants
delete implicit conversions
add automatic storage
add heap storage
require grouping of declarations
add ability to specify read-only access in a scope
require initialization
add range attribute for numeric data
delete fixed point data type
add explicit conversions to integer and real
add ability to define status representation
add programmer defined status conversion functions
add pointer data type
add bit data type
delete label parameters
add ability to compare single characters
add ability to handle multiple character sets (> 2)
add procedure variables
add procedure attributes, such as reentrant
add special entry and exit sequences for procedures
add full parameter checking for procedure call
add inline procedures
add ability to choose from multiple procedure bodies
add ability to specify array bounds at allocation
add packed arrays
use bounds of actual parameter for formal
add array assignment and comparison

add variant structures with programmer specified tags
add tight packing
add array and structure literals
add read/write control on structure components
add explicit conversion for variant components
delete file data type
add constant evaluation on conditional statements
restrict transfers of control into loops and case alternatives
delete index switch
add case default, and require it if range of case index is not covered
add indefinite iteration form with test at the beginning and in the middle of loop body
add exception handling capability
add critical region and shared variable support
add conditional compilation
add data type extension facilities
add inline assembly language procedures
add constant expression evaluation with target machine arithmetic
delete direct code

II. JOVIAL/J73

Modifications to J73 are fully described in Section 3. The following list summarizes the output of the comparison between the hypothesized communications language and J73, and is included for completeness.

Detailed modifications to J73 are:

add constant names
add heap storage
delete implicit conversions
require grouping of declarations
require initialization
add ability to specify read-only access in a scope
add range attribute for numeric data
add status variables
add programmer-defined status conversion functions
add boolean data type
add ability to handle unprintable characters and multiple character sets
add ability to specify length of character string at allocation
add ability to compare single characters
add typed pointer
add procedure variables other than parameters
add procedure attributes, such as reentrant
add special entry and exit sequences for procedures
add full parameter checking for procedure call
add inline procedures
add ability to choose from multiple procedure bodies
add packed arrays
add array and structure literals
use bounds of actual parameter for formal

add ability to specify bounds at allocation
add tight packing
add read/write control on structure components
add variant structures with programmer specified tag
add explicit conversions on variant structures
delete real loop control variable
add character loop control variable
restrict transfers of into loop body and case alternatives
require else in case statement if range not covered
add constant evaluation for conditional statements
add ability to perform loop termination test anywhere in the loop body
add exception handling capability
delete label parameters
add critical region and shared variable support
add inline assembly language procedures
add data type definition capabilities
delete allocation specifier for anything but tables
require constant evaluation to use target machine arithmetic

III. PL/I

Additions to PL/I include boolean and status data types, procedure attributes, a case statement, and the ability to specify packing and variant fields in structures. Major additions include typed pointers, heap storage, and flexible support for process control and exception handling. Deletions from PL/I include implicit conversion, the fixed point data type, varying length bit and character strings, most array operations, label variables, and current methods of process control and exception handling.

Detailed modifications to PL/I are:

add constant names
delete block scope
delete controlled storage
add heap storage
delete implicit declarations
add ability to specify read-only access in a scope
require initialization
add range attribute for numeric data
add explicit conversions to integer and real
delete fixed point (except integer)
add status data type
add boolean data type
delete label variables
add ability to compare single characters
add ability to handle unprintable characters and multiple character sets
delete character string conversions and operations
delete varying length bit and character strings
add typed pointers
add procedure attributes, such as reentrant
add special entry and exit sequences for procedures

add full parameter checking for procedure call
add call by value for simple variables
delete alternate entry points for procedures
add inline procedures
add ability to choose from multiple procedure bodies
change procedure value to not include environment
delete file data type
add packed arrays
add array and structure literals
limit array assignment and comparison to simplest form
add variant structures
add explicit conversions on variant structures
add machine independent and machine dependent packing
add read/write control on structure components
add multiple target assignment
restrict transfers of control into loops and case alternatives
delete switch statement
add case statement
add character loop control variable
allow loop termination test to occur in the middle of a loop
    or at the end
add exception handling procedures
delete ON units
add critical region and shared variable support
delete event variables and wait statement
add conditional compilation capabilities
add inline assembly language procedures
add data type extension facilities
add constant expression evaluation with target machine arithmetic

IV. PASCAL

Additions to PASCAL include bit and character data type, variable initialization, untyped pointers, procedure attributes, programmer specified packing and separate compilation units. Deletions include the set data type, subranges as types, the file data type, and the WITH statement.

Detailed modifications to PASCAL are:

add static allocation
add constant names for all built-in types
add variable initialization
add control over heap allocation
add ability to specify read-only access in a scope
delete ability to associate range with a type
add optional size specification for integers
add explicit operators for conversion to integer and real
add controllable range checks on assignment
add overlap requirement on parameter range matching
add range attribute on real data
add ability to define status representation
add programmer-defined status conversion

add bit data type
add ability to handle unprintable characters and multiple
    character sets
add character data type
add ability to compare single characters
add untyped pointer, LDC function, and untyped pointer operations
add short-circuited boolean evaluation
add procedure attributes for reentrancy, etc.
add special entry and exit sequences for procedures
add procedure variables other than parameters
add ability to select from alternate procedure bodies
add machine independent and machine dependent packing
add ability to determine bounds at allocation
use bounds of actual parameter for formal
add array and structure literals
add read/write control over structure components
delete set data type
delete file data type
add multiple target assignment
restrict transfers of control into loops and case alternatives
add character loop control variable
add ability to perform loop termination test in the middle of
    the body
require a case else if range of the index is not covered
add exception handling procedures
add critical region and shared variable support
delete WITH statement
add inline assembly language procedures
add conditional compilation capabilities
add separate compilation unit capabilities
add support for access to declarations in a named file

V. CS4

Additions to CS4 include bit and pointer data types, procedure attributes, programmer control over type checking in unions, and heap storage. Deletions include the set and fractional data types, varying length character strings, and built-in matrix operations.

Detailed modifications to CS4 are:

delete block scope
add ability to specify read-only access in a scope
add heap storage
add optional size attribute for integers
delete set data type
add ability to handle unprintable characters and multiple
    character sets
add ability to compare single characters
add pointer data type
add procedure variables
delete file data type

add packed arrays
delete matrix operations
delete varying length character strings
add variant structures with programmer control over tag field
delete union data type
add read/write control over structure components
add machine independent packing attributes for structures
add default argument transmission conventions
add procedure attributes, for interrupt procedures and others
add control over procedure entry and exit sequences
add multiple target assignment
add pointer and character loop control variables
make loop exiting capabilities more uniform
add requirement for constant expression evaluation to use
    target machine arithmetic
add conditional compilation capabilities
add ability to define type definitions separate from operations

APPENDIX A

REQUIREMENTS FOR COMPILER EFFICIENCY IN COMMUNICATIONS PROGRAMMING

Section 1

INTRODUCTION

Efficiency is defined as the production of desired effects without waste. An efficient compiler produces object code that does not contain unnecessary instructions and gives its user the ability to allocate global resources properly among HOL coded programs.

Example: An efficient compiler will not produce code to load a register that already contains the desired value. It may allow the user the option to specify that a register be "frozen" to contain the address of a certain data block.

Many factors affect the degree of efficiency obtained from a compiler. These include the features of the language to be compiled, the application for which the language is used, and the architecture of the target machine. A well designed compiler which utilizes state-of-the-art optimization techniques and a carefully chosen language can produce good quality, efficient code. The communications HOL will be designed to produce such code, and this report discusses the methods to be used.

However, utilization of good internal design techniques is not sufficient to produce a compiler that meets all the efficiency needs of real, in the field communications applications programmers. One must also give consideration to the needs and skills of the intended users, and provide for interaction between the compiler and the programmers. Or, in other words, the compiler must be an efficient tool. Specifically:

- High order languages have not previously been used for communications applications. Once the communications HOL is in use, data will be available to improve and refine the original design. Provision should be made in the compiler to provide information to its users to aid the refining process. The compiler designers should be very critical of their own assumptions about the totality of their knowledge, and should provide for incremental change in areas of the compiler that may later be improved.

- Even the best optimization techniques cannot compete with human insight into the most efficient way to structure certain algorithms. Therefore, the compiler should provide directives which give the programmer the ability to control how it performs certain data allocation and code generation decisions.

- Programmers will do their best to write source code that exploits the compilers optimization strengths. Therefore, the language should be designed so that readable source code produces optimized object code. User documentation should contain information about the efficiency implications of

various coding choices. The compiler should provide special listing data to be used by programmers attempting to tune their programs.

This paper discusses the methods to be used to create an efficient compiler for the communications HOL. These methods will be discussed in two categories -- language level methods and machine level methods. Language level efficiency is obtained by the choice of language features and by the optimizations that can be specified for all implementations of a compiler. Machine level efficiency is obtained by the code generation methodology for a specific target machine and by the features that enable the programmer to control code generation when necessary.

Section 2

LANGUAGE LEVEL EFFICIENCY

Language level efficiency is obtained by choosing language features that allow generation of good object code in combination with the machine independent optimizations included in the compiler.

Example: If the language contains conditional expressions one can write I=IF pred THEN A ELSE B. If only conditional statements are available, one must write IF pred THEN I=A ELSE I=B. However, if the compiler has an optimization which looks for a common left hand variable in a conditional statement, it can generate code as efficient as that generated for a conditional expression.

2.1 Language Features

One of the important requirements for the communications HOL is the efficiency of the compiler. Therefore, a positive contribution to efficiency will be one of the criteria for selecting language features. Other factors such as a readability and understandability are also important considerations for the selection of language features. If a feature that impairs efficiency is chosen, methods of correcting for its effects will be designed in the compiler.

One feature of the language will be a uniform syntax for directives -- statements which are used to control the generation of code by the compiler rather than to generate code themselves. One of the uses of this language level feature will be to obtain certain machine level efficiencies. These uses for directives will be discussed in detail in Section 3.2.

2.2 Language Level Optimizations

The term language level optimization refers to analyses performed by the compiler on the source code in order to generate more efficient object code. There are many language level optimizations that can be implemented in a compiler, both in the funding of its implementation and in the constraints in its run time environment. Therefore, a careful choice must be made of which optimizations are included in a production compiler.

One of the factors in choosing an optimization is whether it can be specified in a machine independent fashion. If the compiler will eventually generate code for multiple target machines, then machine independent optimizations are important. They assure that the compiler will optimize uniformly across all its implementations. In addition, since these optimizations are contained in the machine independent portion of the compiler, they need only be implemented once.

Another factor to consider when choosing optimizations is the expected frequency of the statement or operation type to be optimized. For example, one would expect a high frequency of character manipulation operations in communications programs. Therefore, an optimization which improved these operations would be considered very valuable to the communications HOL compiler.

A third factor to consider when selecting optimizations to be included in any language is the complexity of the optimizer. This is partially a cost consideration, but primarily a design consideration. There is a limiting point beyond which the interactions among the various optimizations are so complex they they exceed human comprehension. This factor must be taken into account along with the first two factors in order to make good judgments about the inclusion of optimizations in the compiler.

Some of the language level optimizations that may be specified for the communications HOL compiler are:

- Common subexpression extraction. When the same value is calculated more than one time, the compiler can recognize this, and generate code appropriately.

  Example: L1=A+B ... L2=A+B

  Assuming no assignments have been made to A or B then "A+B" is a common subexpression. This concept can be extended to be more inclusive.

  Example: L1=A+B ... C=A ... L2=C+B

  Example: IF A<B ... IF A>B ...

  The compiler can recognize A+B and C+B as the same expression, and can treat comparisons as a common expression viewed as "A compare B".

  Common subexpressions also occur in generated object code, and can be recognized by the compiler. In fact, recognition of generated common expressions, especially for addressing, is a very important optimization to include for the generation of good object code.

  Example: in the statement: IF A<B THEN C(X+1)=A ELSE C(X+3)=B ... addressing calculations should be extracted as common subexpressions as much as possible.

- Loop optimizations. There are several optimizations that can be applied to generate efficient code for loops. Loop control variables can be kept in incrementable registers. Reduction in strength can be performed, for example, to compute the address of the next object to be accessed by

addition rather than multiplication when possible. Sometimes more efficient code can be generated for a special case by rearranging the logic for a loop. One might want to split a loop into separate loops, combine separate loops into a single loop, or unroll loops.

Example: If the statement:

FOR I STEP 1 ... A(I) ...

is operating on an array of characters packed two per word, better code might be generated for

... FOR I STEP 2 ... A(I) ... A(I+1) ...

While it would be difficult for the compiler to independently determine when such optimizations should be performed, it could perform them by directive. Thus, efficient object code could be generated without sacrificing source code readability.

- Support for register management. Although most of register allocation strategy is machine dependent, the machine independent portion of the compiler must gather the data to support good register management. Usage counts should be computed for source variables and for generated temporaries. Flow analysis should be performed to determine the basic blocks over which variables can be kept in registers. Language features supporting structured programming simplify the flow analysis quite a bit.

- Optimizations based on range data. The compiler can optimize code by utilizing range information. If, for example, the range of A is declared to be between 0 and 10, the compiler might treat the expression A<1 as A=0 if that could generate more efficient object code.

- Short circuited boolean evaluation. When the compiler encounters a statement such as IF A<B AND C<D, it need not evaluate C<D if A<B is false. If short circuited boolean evaluation is specified, however, the programmer must be aware that code in the second part of a boolean evaluation may not be executed. For example, in the statement IF A<B AND PROC(C)<D, if A<B is false, the call to procedure PROC will not take place.

- Miscellaneous local optimizations. Various local optimizations -- optimizations on a single calculation -- can be performed.

Example: calculate 2*A as A+A or shift left A by 1.

Example: generate code for A(X+1) using the address +1 of A offset by X.

Good choices of optimizations will be made at compiler design time. However, as experience is gained in use of the language, optimization by compiler can be improved. Therefore, the compiler design should allow for incremental addition of optimizations. In addition, provision might be made for the compiler to provide cumulative information about statement and data type usage frequency.

Section 3

MACHINE LEVEL EFFICIENCY

A big cause of efficiency in HOL generated code is a lack of correspondence between the form of the high order language and a given machine architecture. In other words, the very qualities of a high order language that insulate the user from the machine in order to provide program understandability and maintainability are a source of inefficiency. This is because the compiler cannot always recognize when a segment of code can be restructured to exploit a special machine characteristic.

Historically, high order languages were designed for use on large mainframe machines where resources allowed somewhat inefficient use of machine characteristics. Recently, however, the value of high order languages for mini-computer applications has been recognized, and the need for more effective use of target machine characteristics has arisen. This need can be met by code optimization, by the inclusion in the language of directives to assist the compiler in generation of code for specific cases, and by the ability to explicitly specify the code to be generated in portions of a program written primarily in the high order language.

3.1 Compiler Code Generation

There are several methodologies available for generating high quality code for a specific target machine. All of these should be employed by the communications HOL compiler.

Addressing inefficiencies are a common problem in compiler generated code and should be avoided in the communications HOL. A good register assignment strategy that recognizes the frequency of use of variables and that takes advantage of special machine characteristics is important for efficient addressing. However, the compiler cannot always make the best choice of addressing strategy for every situation. Therefore, its default algorithm must be modifiable by programmer supplied directives.

Variable code pattern selection by a compiler increases the efficiency of generated code. That is, the same code pattern is not always generated for the same operation. For example, when the compiler generates code to perform the operation "add A1 to A2" it queries its simulated machine state to determine conditions such as if A1 or A2 are contained in registers, and generates code according to the answers. The more questions that are asked, the better the potential for high quality code. This approach has its limitations, however, since each new question asked doubles the number of cases, and too many cases become unmanageable.

Local optimizations can be performed on both single and multiple instruction sequences. Single instruction optimizations delete instructions such as shifts or adds of zero. Multiple instruction optimizations use peephole optimizations to recognize patterns of code that can be replaced by more efficient patterns. For example, the sequence LOAD A, ADD 1, STORE A might be replaced by a single INCREMENT A instruction.

When allocating resources for generation of high quality code, the projected use of the compiler for its application should be taken into account. For example, in the communications HOL, more resources will be allocated to move and compare operations than to arithmetic calculations.

3.2 Code Generation Directives

The programmer can recognize aspects of programs and of the system that cannot be recognized by the compiler. Because of this, directives should be provided to allow the programmer the option to control certain aspects of code generation.

Directive control is definitely necessary for register assignment. There are special considerations in individual compilation units and there are system wide efficiency considerations that cannot be deduced by the compiler. For example, on a system wide basis it may be desirable to "freeze" a register to the location of a certain variable, while the importance of this variable may not be obvious in a specific compilation unit. Or, the default strategy may be to assign a register to point to the literal pool, but some programs may produce more efficient code if that register is free for general use. Both specific assignment directives (e.g., freeze register A to variable B) and general algorithm directives (e.g., use register management strategy number 2) would be useful for the communications HOL.

Directives will also be necessary to assign data to specific locations and to specify the link among various data items. There will definitely be specific directives -- those that assign one variable to a register or storage location. It may be possible to implement a more general set of directives which give the compiler additional information, but do not explicitly specify the code.

Example: a directive that says "make X easily addressable by Y"

Example: a BASE directive which assigns a variable to a base register without specifying which register.

Directives that cause or suggest certain optimizations might also be implemented. For example, some of the loop optimizations suggested in Section 2.2 might best be controlled by directives. Or, a directive might state the probability of execution of the THEN and ELSE branches of a conditional statement.

Directives may also be used for special procedure calling and entry sequences and to support encapsulated or embedded assembly language. This usage will be discussed in more detail in Section 3.4.

3.3 Programmers Use of Directives

The use of directives can significantly enhance the ability to generate efficient, tailored compiler code without resorting to explicit code (assembly language). Some consideration should be given to how the programmer can use these directives and understand their effect without reading the assembly language generated for each compilation unit.

One aid for code tailoring would be an optional compiler listing containing code generation information. This listing would print space and time estimates for various sections of code. It would also flag source code lines to indicate what code generation actions that they caused. For example, flags could indicate such actions as:

- common subexpression extraction
- base register loading
- literal creation
- temporary location creation
- NO-OP creation for alignment
- long instruction generation

During detailed compiler design, the types of flags to be printed and their meaning would be determined. The documentation of these flags would suggest possible actions to the programmer. For example, unused space in data areas due to alignment might be removed by reordering data declarations. Or, an excessive amount of subexpression extraction might indicate that an algorithm could be recorded in a more compact fashion.

One could envision the following program development/debugging/tuning process:

- Programs are written, following good coding practices, and compiled.
- The system is linked, and functional testing begins. Functional testing can continue while the tuning activities take place.
- Information from the system link and testing identify programs that are not small or not fast enough.
- "Problem" programs are recompiled to get code generation data.
- If the problems cannot be solved by the previous process, then the generated assembly code is listed for detailed efficiency analysis.

3.4 Explicit Code Generation

All hypotheses about the communications HOL have assumed that it would allow the programmer to explicitly specify the code to be generated if necessary. At times, this capability has been referred to as embedded assembly language, but the term explicit code generation is more accurate.

Explicit code generation is the ultimate escape from the generality of the HOL. Its use, however, has adverse affects on other areas. It impacts the safety/reliability afforded by the use of an HOL, and for this reason should be avoided when possible. Use of explicit code can also hamper effective optimization by the compiler in a compilation unit in which it appears. The use of directives to minimize this impact is the major concern of this report.

The compiler should have as much information as possible about the effects of an explicit code segment on the machine environment. For example, the compiler should know which registers will be changed by explicit code in order to set up efficient register allocation for the rest of the compilation unit. In addition, the compiler should know which variables will be modified by the explicit code so that it doesn't have to assume loss of control of all variables when explicit code is encountered. In other words, the introduction of explicit code to improve efficiency in one area of a compilation unit should not degrade the efficiency of the remaining HOL generated code.

One area in which explicit code generation will be used is in procedure linkage. Compilers normally generate one or a few standard call and return code sequences which make standard assumptions about the necessity to save and restore registers. Default assumptions are also made about the manner of passing parameters. There are times when significant efficiency can be gained by the use of special case procedure linkages. It may be that the link-editor can generate data to indicate subroutine overhead information in a form to aid procedure linkage tailoring when needed.

The same mechanism used to direct special procedure linkage will be used to support embedded explicit code. Embedded code will be surrounded by a procedure type header and trailer. Directives will be used to specify exact linkage. This will contribute to the uniformity of the language and to the simplicity of the compiler design. It is doubtful that default assumptions should be provided for embedded code as they are for procedure linkages, but otherwise the same mechanisms should be applicable.

Section 4

CONCLUSION

All current state-of-the-art optimization techniques will be judiciously used in the communications HOL compiler. New directions will be taken in integrating recognition, control and use of machine oriented capabilities into the language in a consisten- and usable fashion. Provision will be made when possible for incorporating improvements into the compiler as experience has gained via field use of the language.

APPENDIX B

RESULTS OF COMMUNICATIONS

HIGHER ORDER LANGUAGE QUESTIONNAIRE

1. INTRODUCTION

1.1. General

This report is a summary and interpretation of the results of the questionnaire conducted under RADC's Communications HOL investigation.

This questionnaire was developed by DSA under SOFTECH's guidance to provide insight to various aspects of a communications HOL. Questions were included for the following reasons:

1. They concerned an issue whose resolution would have bearing on the structure of the language and for which there were contradicting opinions.
2. They had bearing on superficial features of the language (i.e., semantic sugaring), which might influence the language's acceptability.
3. They served as a check on the respondent (e.g., Was he awake?).
4. They probed gut-level response to non-technical barriers to the use of any HOL.

The questionnaire consisted of 277 questions, in the form of assertions. The respondents were asked to state their degree of agreement with the assertion by writing a number from one to five (total disagreement to total agreement). They were also asked to judge the relevance of the question to communications programming on a similar scale of one to five with one irrelevant and five as crucial. The order of questions was randomized to avoid leading the respondent. Questions concerned with status types (Q257 - Q277) were not randomized. Appendix I (raw data responses) and Appendix II (questionnaire) are in unscrambled order.

1.2. Notes

1.2.1. Strength of Responses

We need to explain our usage of the terms "strong," "moderate," "weak," "disfavor," etc. While we provided a scale of 1-5 in which to express opinions, most individuals do not feel that strongly about anything. We examined the questions and the responses in detail. This suggested that the entire scale of 1-5 was effectively mapped on a range of approximately 1.7 to 4.5. On this revised scale, a mean response of 2 would be an extremely strong rejection of the statement, similarly, a mean response of 4.5 would be an enthusiastic agreement with the statement. Therefore, we will use terms like "strong" for 3.5 or 2.5, as appropriate.

ACTUAL RESPONSE	1	2	3	4	5
EFFECTIVE RESPONSE	0	1.5	3	4.5	6

1.2.2. Consistency

To judge whether an individual's response to correlative questions were consistent, we ask that the responses be within $\pm$ 1 of each other except for "3" response. Thus pairs of responses as in the following table would be considered consistent:

11	12	13	14	15
21	22	23	24	25
31	32	33	34	35
41	42	43	44	45
51	52	53	54	55

If the two questions were contradictory, the match was made on a complementary scale as in the following table:

11	12	13	14	15
21	22	23	24	25
31	32	33	34	35
41	42	43	44	45
51	52	53	54	55

To maintain a balance interpretation, 33 responses to a pair of statements were ignored - this is justified since the individuals didn't really care one way or the other about the issues. Consistency evaluation was done only for individuals who responded to both statements. This notion of consistency is a bit subjective since questions were rarely directly contradictory or supplementary.

1.2.3. Agreement

The tone of most of the statements was positive. That is, we tended to structure them to expect a positive (agreement) response, rather than a negative response. There were reasons for this, the most important being that a preponderance of negative statements (that is, statements to which we would expect a mean negative response) would have antagonized the respondent and further eroded what motivation he may have had toward the completion of the questionnaire. Conversely, we could not overbias the questionnaire in the positive direction since individuals tend to agree with the written word - go along with what they think is the "correct" answer; the statement, since it is a printed statement, must at some subconscious level, be the "correct" answer to the "question." To counteract this trend we inserted some statements to which we expected overwhelming disagreement. This served to keep the respondent off-guard and avoided mechanical responses. The mean agreement was 3.29, with a standared deviation of .52. We also suspect that in many cases, where the respondent did not understand the question, he or she wrote "3" rather than leave it blank. This was mentioned by

some respondents. Recent graduates, who tended to view the questionnaire (at an emotional level, rather than intellectually) as a test, did this most often. In some cases, our conclusions were obtained by removing the "3" responses to see what remained.

1.2.4. Relevance

There was a conscious attempt to make the overwhelming majority of the statements relevant. The mean relevance was 3.44 with a standard deviation of .32. Caution should be exercised in the interpretation of relevance responses.

1. There was a bias in the introduction to the questionnaire which told them the questions were relevant, besides which the mere fact that the statement had been included automatically made it relevant in their mind. They would have been reluctant to say something was irrelevant. For example, the most unimportant thing of all, questions related to the required range of real numbers, were given a 2.42 and 2.48. The ability to insert non-printing characters into a comment was given a 2.48 also. Conversely, the statement relating to the message accountability and avoidance of misrouting was given a 4.73 (about right).
2. The respondents tended to judge relevance in accordance to their responses. That is, relevance was correlated highly with agreement or extreme disagreement. That is,,"If I agree (strongly disagree) with it, it must be important." An unbiased approach would have used a blind experiment, in which respondents were asked to comment on agreement, separate respondents on relevance, and yet a third group, on both. The three questionnaires would have had to be different, to avoid yet another kind of bias.

   The correlation between agreement and relevance is evidenced by the high proportion (particularly among some respondents) whose answers were of the form "11," "22," "33," "44," "55," or alternatively, "15," "24," etc., if they happened to disagree with the statement.

In view of the above bias, we do not recommend attaching too much significance to the relevance responses outside of the context of this study. In conjunction with the agreement responses, it has enabled us to delve a little deeper and extract something we might have missed otherwise - it has proven useful.

2. NUMBER OF RESPONDENTS

Approximately 200 copies of the questionnaire were distributed to various telecommunications software groups in the United States, Europe and South America. The groups were chosen on the basis of the following criteria:

1. All individuals had personally participated in the development of at least one real telecommunications system - at a level that required them to spend a significant part of their time producing source code.
2. Most respondents had significant experience in military communications systems (approximately 90%), and were familiar with U.S. and similar documentation standards.
3. Level of respondents ranged from relative novices (under two years of experience) to highly qualified lead programmers and managers. A fair balance was sought, reflecting the typical programming organization structure.
4. We attempted to diversify the sample so as to avoid biasing the results to one population or another.

There were a total of 58 responses to the questionnaire at the time the final tabulation was done. The average number of responses to any question was 50.3, with a standard deviation of 3.4. The typical respondent answered 249 questions out of a total of 277, or 90%. The typical respondent complained about the length of the questionnaire. The number of responses to the relevance issue was somewhat smaller - 49.12 respondents per questions with a standard deviation of 3.7 and 243 questions or 88%.

The program used to collate the data also did a calculation of the rate at which the answers were converging. Several convergence criteria were used. At present, it would require a drastic population shift to change the mean values by more than one or two percent. While additional responses might have been comforting, they should not be statistically necessary. A concerted effort to reach a wider population might reach the entire base of 3,000 to 5,000 elig ble programmers. Drawing on such a random population, with which we have no contact,the

number of respondents would be materially decreased. We would expect no more than 150 to 250 responses (of which we already have 60). The cost of reaching these would be approximately $15,000 to $20,000, for what would be marginal returns.

3. COMMENTS

Respondents were encouraged to comment on the questionnaire and any HOL issue that moved them. The most common comment was to point out that Questions 1 and 16 were identical; this was due to a typographical error. The second most common comment was that the questionnaire was too long. The remaining general comments are given below. Specific comments on questions are listed with the questions or question groups to which they belong. Other comments relating to the structure and organization of the questionnaire are not included since they have no bearing on HOL issues.

1. "I would not like a compiler to insert anything for me unless the programmer controlled the insertion."
2. "Real numbers are not used often. Time can be represented as an integer multiple of some known unit. The XYZ (computer) has no real numbers. The advantages of HOL would be in documentation, speed of implementation, and the elimination of bugs caused by 'mechanical' errors (typing, forgetting a restriction, etc.). As programmed by an 'average' programmer, a HOL program would probably be less efficient than an assembly program. But this is not so important as the potential gains. Loop control is usually very simple and increments by word or block size. Because of high hardware dependence, much coding will still have to be done at assembly level even with HOL."
3. "Almost all conditions should have a documented default condition to alleviate the defining process. Most HOL's don't allow enough direct control of I/O functions, which adds enormous data shuffling and error processing code. That is, input of raw data into accessible bit strings, handling variable length records access somehow, to hardware status after operation, etc. There should be two distinct sets of functions (verbs) and documentation; a) the HOL does it all, for normal use in systems, b) the HOL does nearly nothing for special subroutines to handle off-line tasks, i.e., READ, WRITE, MOVE - type a; DEVICE-IO (DEV, FUNC, STATUS) - type b."
4. "There is too much knowledge, no single most common reason for bugs. Lack of programmer knowledge or attention to detail, lack of understanding of specifications, hurrying (lack of research) etc., are all contributing factors. HOL appears more inflexible to me than low level assemblers. Debugging, maintenance, and enhancements become more difficult with HOL. Higher order languages such as NCA-CDP contribute to inefficient coding in the reference of instructions status (personal experience)."

Elementary operation coding takes much less execution time. Simple, single action for single instruction assembler language is easier for me to read and understand."

5. "What is this HOL really supposed to accomplish for communications software?"

6. "In language design, I feel simplicity to be more important than elegance. The rules should be stated clearly and explicitly. Further, many features of the language should require a user to take explicit action. Languages for communications system design are akin to (if not equivalent to) standard system programming languages. Many of the principles followed in better SPL's would serve as a good model."

7. "Be very careful about compilers generating code. The programmer does not necessarily want the compiler to generate multiple machine instructions most of the time. The more trouble the programmer has with the compiler generated code, the faster he resorts to one [source] instruction makes one machine instruction. Problem varies with quality of compiler implementation and hardware. I feel that the communications programmers (especially assembly programmers) do not want fancy or exotic things from a HOL. He wants functional, efficient, simplicity. He does not want to be forced into anything or method. He likes the freedom of assembly."

8. "I think ink blots are prettier."

9. "The ability to define a table search type instruction which repeats, auto-increments table pointers, and updates iteration counts or interrogates for a physical table terminator has been omitted from this questionnaire."

10. "IF HOL > ASSEMBLER THAN DO HOL ELSE DO ASSEMBLER."

4. RESPONSES TO QUESTIONS

4.1. Declarations

Implicit/explicit (Q1-Q5).

4.1.1. Purpose of Questions

The purpose of these questions was to determine what the language should reflect with respect to implicit or explicit declarations. Should declarations be explicit, should implicit declarations be allowed, or should some third course be implemented?

4.1.2. Comments

1. "Yes! Default values."
2. "If you do this [insert missing declarations] - need compile time error messages."

4.1.3. Questions and Responses

Q1 - "Declarations of all properties of variables shall be made explicitly."

A	3	16	7	19	4	49	3.10	1.13
R	0	1	19	24	4	48	3.65	.66

There was slight agreement, but a clear cut bimodal distribution. The distribution cuts across all groups and does not correlate with anything obvious. The best we can come up with on this one is that agreement with the statement correlates with HOL experience. Pure assembly programmers tended to disagree - this correlation is weak.

Q2 - "The language will allow default declarations of properties where obvious and safe."

A	4	13	11	21	4	53	3.15	1.13
R	0	5	20	22	5	53	3.51	.79

Another, although weaker bimodal distribution with a slight favoring of default declarations. The answer is inconsistent with the first question's answer. Approximately half of the respondents gave consistent answers. The consistent answers were dominated by those who favored default declarations.

Q3 - "It will not be necessary to make explicit declarations of loop control variables."

A	4	21	6	14	2	47	2.77	1.10
R	0	0	25	18	3	46	3.52	.62

Another bimodal distribution with opinion running against default declarations in this case. We cannot find any common attributes to the two populations. The answers were extremely consistent with those given to Question 2.

Q4 - "The compiler shall recognize missing declarations and insert them in accordance to a well documented scheme."

A	4	20	6	20	4	54	3.00	1.16
R	1	7	16	24	5	53	3.47	.90

The bimodal distribution continues. Surprisingly this proposal is less acceptable than Q2. Consistency with Q2 was high (30 out of 50 respondents). No clear-cut pattern here either.

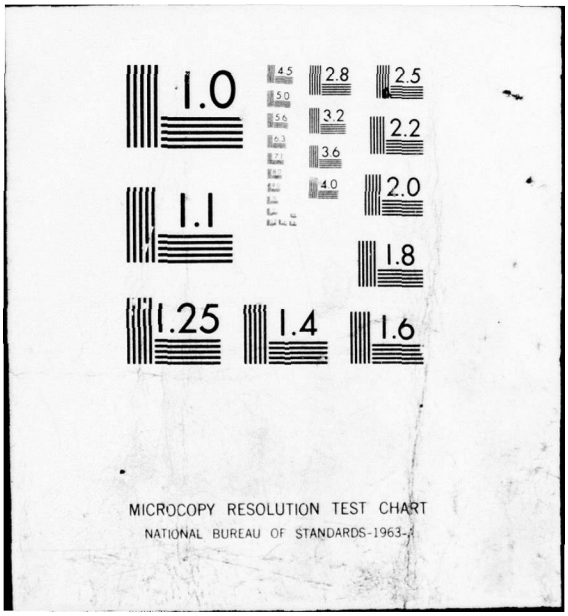
Q5 - "The compiler shall insert missing declarations which will require explicit confirmation by the programmer. Once confirmed, these default declarations shall have the same status as those supplied by the programmer."

A	3	16	9	15	2	45	2.93	1.06
R	0	3	19	22	1	45	3.47	.65

Almost no difference in response to this question compared to Q4. Extremely consistent responses to Q4 and Q5 (39/45 for those who answered both questions). Those who objected tended to be more junior than those who agreed, except that those who objected the most were the most experienced and similarly, those who favored it the most were also among the most experienced.

AD-A049 737 SOFTECH INC WALTHAM MASS F/G 9/2
COMMUNICATIONS HIGH-ORDER LANGUAGE INVESTIGATION, VOLUME I.(U)
OCT 77 R S EANES, J B GOODENOUGH, J R KELLY F30602-76-C-0306
UNCLASSIFIED RADC-TR-77-341-VOL-1 NL

2 OF 3
AD
A049737


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
7.1
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

4.1.4. Analysis

There was no obvious pattern to the answers, at least none sufficient to explain the bimodal distributions that dominated the answers to this issue. If anything can be surmised from this, it is that there is a slight bias in favor of explicit declarations, and a fear of default declarations.

4.2. Declarations - Order of Appearance (Q6-Q8)

4.2.1. Purpose of Questions

These questions were intended to probe the order in which declarations should occur. We wanted to get a confirmation that it was important to allow functional groupings of declarations, which meant that declarations had to be capable of being intermixed. The question related to mixing declarations with code was intended primarily to serve as calibration of the respondent.

4.2.2. Comments

None.

4.2.3. Questions and Responses

Q6 - "All declarations of a given type must be grouped with others of the same type."

A	9	31	6	6	1	53	2.23	.92
R	1	5	22	19	4	49	3.39	.84

Opinions were strongly opposed. No clear cut pattern to the favoring group and the opposing group.

Q7 - "Declarations may be intermixed with code."

A	10	20	10	11	1	52	2.48	1.08
R	0	5	24	15	6	50	3.44	.83

We expected and got strong disagreement to this statement. The respondents who favored this approach were generally experienced and talented.

Q8 - "Declarations can appear in any order in the declaration area and can be intermixed as to type."

A	3	11	9	25	8	56	3.43	1.12
R	0	4	27	19	5	55	3.46	.76

General agreement with this approach, but with a clear minority opinion. The distributions were completely consistent with the answers given to Q6. That is, those that liked the Q6 approach did not like the Q8 approach and vice versa. The answers are for the most part reflections. No clear cut pattern to those opposed.

4.2.4. Analysis

There was a general willingness to accept some restrictions on the location of declarations (i.e., not allowed in code). Furthermore, the language should allow functional groupings of declarations.

4.3. Declarations - Scope of Q9-Q13

4.3.1. Purpose of Questions

Actually, this is one of several groups of questions relating to the treatment of scope in general.

4.3.2. Comments

None.

4.3.3. Questions and Responses

Q9 - "All declarations shall apply to the entire module in which they are made; properties of objects remain constant in a given module."

A	0	5	8	26	8	47	3.79	.85
R	0	3	17	23	3	46	3.56	.71

Generally strong agreement with this statement. This was the clearest and most understood statement of this group. No clear cut characteristics of the minority group.

Q10 - "The declarations of calling module can override the declarations of the module it calls."

A	10	15	6	16	6	53	2.87	1.33
R	2	4	15	25	5	52	3.54	.91

We expected and got general disagreement with this statement. The distribution however, is bimodal. The experience level of those opposed generally was higher than those who favored it. When we removed those questionnaires which were inconsistent to the answer of Q9, the opinion swung overwhelmingly opposed to allowing an override. The revised mean was 2.08, which would make it one of the most decisive answers in the entire questionnaire. This time, the bimodality appears to be due to confusion and incomplete understanding of the issues and/or questions.

Q11 - "Local declarations can be overriden by common system wide declarations."

A	4	17	7	16	1	45	2.84	1.07
R	1	1	21	18	3	44	3.48	.75

There is general disfavor towards override. Experienced programmers tended to cluster in disfavor of this question. Removal of inconsistent reponses did not materially change the result.

Q12 - "The declarations of a submodule can override the declarations of the modules that call it."

A	5	21	5	18	3	52	2.87	1.16
R	2	3	20	25	2	52	3.42	.82

Stronger disfavor here with regards to the submodule overriding. We would have expected this option to be favored because often, subroutines are created to encapsulate a complex process, which for security/accountability reasons must be done properly. To assure that this is done, the subroutine overrides the calling routine. If parameters do not check, the error is noted and flagged at run time. The consistent results outweighed the inconsistent when compared to Q11. There was no clear-cut distinction within the consistent group (those who gave the same responses

to Q11 and Q12) with respect to favoring and not favoring. Within the inconsistent group, those who favored override by the subroutine were distinctly more experienced than those who opposed override.

Q13 - "The highest level declarations override the lower level declarations unless specific instructions are given to the contrary."

A	1	6	8	23	2	40	3.47	.89
R	1	4	15	17	2	39	3.38	.84

The agreement is definite, and is indicative of a general desire to have control over things - not to close out options. The reduced response to this statement was not due to lack of understanding of the issues, but merely due to the fact that this question came toward the end of the questionnaire (Q193 scrambled version) as the respondents saw it. Since they were urged to submit partial questionnaires, this was one of the ones that was sacrificed.

4.3.4. Analysis

There was some confusion over the intrepretation of these questions and the respondents did not have a clear idea, or the time to figure out the implications. They did not consider these to be burning issues, as judged from the relevance. What we should probably do is to adopt a posture with respect to declaraction scope and override, and furthermore allow modification of the built-in policy by explicit statements to that effect. Of course, all declaration conflicts should be detected and flagged.

4.4. Access and Security - (Q14-Q18)

4.4.1. Purpose of Questions

The purpose of these questions was to examine the approach to be taken with respect to controlling access in the interest of security and software reliability. Unfortunately, Q15 and Q16 were

identical in the scrambled version; this was picked up by many respondents.

4.4.2. Comments

1. "Access by routines to data base elements and to other routines and/or groupings [inserted by respondent] shall be allowed unless explicitly forbidden by the programmer."
2. "I would not like a compiler to insert anything for me [compiler inserted run time checks for access control] unless the programmer controlled the insertion."

4.4.3. Questions and Responses

Q14 - "Read/write/execute restrictions can be associated with routines and data base elements."

A	0	0	9	31	5	45	3.91	.55
R	1	5	12	23	3	44	3.50	.87

Overwhelming agreement with this proposition

Q15-Q16 - "Read/write/execute restrictions can be associated with routines and data base elements and the compiler inserts the proper run time checks."

A	1	9	21	59	9	99	3.66	.83
R	4	14	34	37	8	97	3.31	.97

Very strong agreement with this proposition

Q17 - "Access by routines to data base elements and other routines shall be allowed unless explicitly forbidden by the programmer."

A	2	6	10	25	6	49	3.55	.99
R	2	3	16	24	5	50	3.54	.90

Strong agreement, but note the reduction as the implied work is increased.

Q18 - "Read/write/execute restrictions can be associated with routines and data base elements. The compiler performs proper compile time checks, but inserts run time check code only if directed by the programmer."

A	0	6	12	27	4	49	3.58	.81
R	2	3	20	21	1	47	3.34	.81

Strong agreement as before. We would have hoped for more agreement since this is the real proposal.

4.4.4. Analysis

We examined the responses to these questions, particularly the negative responses. We found the following on the basis of examining responses that had objected to at least one of these four questions:

1. One respondent who gave inconsistent answers to two identical questions.
2. Of the negative respondents, among those which we had calibration, three were from very sophisticated programmers, whose disagreement was over whether the run time checks should be automatic unless forbidden by the programmer, or inserted only if asked for by the programmer.
3. The remaining responses were from uncalibrated or junior sources.

Based on the pattern of these answers, we suggest that restrictions be part of the declarations and that some kind of warning is given if restrictions have not been specified. Both forms "forbidden unless allowed" and "allowed unless forbidden" should be employed. The programmer should have the ability to insert run time checks. Compile checks should always be made. Run time checks should be the default option. Elimination of run time checks should be by explicit declaration and the routine or data base element should be tagged in all listings.

4.5. Cross Reference List (Q19)

4.5.1. Purpose of Question

The primary purpose of this question was to see if the respondent was awake.

4.5.2. Comments

None.

4.5.3. Questions and Responses

Q19 - "The compiler shall generate a complete cross reference index of accesses to the data base, subroutine calls, macro calls, indicating the allowed access (read/write/execute) the identification of the caller, and the type of accesses attempted.

A	0	2	1	22	27	52	4.42	.72
R	0	0	4	28	19	51	4.29	.60

As expected, an overwhelming agreement with this proposition.

4.5.4. Analysis

The only interesting result here is the two negative responses. Both were from very experienced and sophisticated programmers, who had extensive telecommunications programming experience, and subtantial HOL experience, not only as users, but as participants, in compiler development. We were able to follow up one of the respondents; his concern was with the implied complexity of the compiler, that extensive cross referencing was desirable from a user's point of view, was not an issue.

4.6. Variant Data Structures (Q20-Q24)

4.6.1. Purpose of Questions

These questions were aimed at determining the incidence and type of usage of variant data structures, and delimiting the kinds of facilities that would be necessary to support them.

4.6.2. Comments

None.

4.6.3. Questions and Responses

Q20 - "When the same data structure (e.g., a queue block) is used with different interpretations, the interpretation is generally constant within the subroutine or module."

A	1	5	6	36	7	55	3.78	.85
R	2	2	18	27	5	54	3.57	.85

The respondents felt this was a moderately important issue and agreed strongly with the statement. The minority group are experienced in telecommunications and represent a fairly broad spectrum of experience. Most have been intensely involved in assembly language programming of switching systems. They are not distinguishable from the majority group. This is just how these individuals perceived the world.

Q21 - "The interpretation of the format of a data base element (e.g., queue block) cannot be changed within a subroutine. Control must be transferred to a different procedure."

A	2	15	2	26	1	46	3.20	1.06
R	1	2	15	25	2	45	3.56	.75

We expected and got a controversy with respect to this proposition. There are no obvious distinguishing characteristics between the dissenting group and the majority group.

Q22 - "To change the interpretation of a data element (e.g., a queue block) an explicit statement which identified the format to be used next must be supplied (this entails no run time code)."

A	1	3	11	23	6	44	3.68	.87
R	0	3	15	19	5	42	3.62	.78

Since this was a more resaonable approach to the treatment of variant structure we expected and got more agreement with the proposal. The shift from a bimodal distribution is completely accounted for by a change of mind of the minority group in Q21, who objected to an enforced non-variant interpretation within a module and enthusiastically supported a mechanism that would

allow variant structure. The shifts were from (1,2) to (5,4). There was only one reverse shift, who appears to object to variant structures. His answer was consistent with what he saw as the incidence of variant structures.

Q23 - "The interpretation of data structures can be changed at will - i.e., different formats may be used, different masks, etc. The compiler will make no checks and no prohibitions."

A	7	26	3	10	2	48	2.46	1.10
R	0	3	13	19	10	45	3.80	.86

We had hoped for rejection of this proposition and got it. The proposal was intended to reflect the situation as it now exists in assembly language. Most of the minority group felt that the data structures were essentially constant; only three of the minority group felt that the data structures were not constant (Q20) and presumably needed the freedom this proposition implied.

Q24 - "The interpretation of a data structure can be changed at will as long as all variations in the interpretation have been declared. Compile time checks will be performed, but no run time checks will be inserted, unless specifically dictated by the programmer."

A	1	9	13	15	4	42	3.29	.98
R	1	7	15	13	3	39	3.26	.93

Based on the shift in responses, the dissent was due to individuals who wanted fairly rigid controls. The freedom lovers stuck to their guns. Responses were generally consistent.

4.6.4. Analysis

Data structures variants do occur and it is convenient to be able to switch to a variant interpretation within a routine. Control over such shifts (such as an assert statement) which implies no run-time code would be acceptable. Complete freedom is not deemed essential. To force a change of process in order to use a variant structure will not be acceptable.

4.7. Re-Entrant Code (Q25-Q29)

4.7.1. Purpose of Questions

To determine the attitude and default conditions related to the implementation of re-entrance. To determine if the compiler-managed stacks (i.e., code generated by the compiler) in support of re-entrance were acceptable.

4.7.2. Comments

None.

4.7.3. Questions and Responses

Q25 - "The management of dynamically allocated stacks in support of re-entrant code will be done by compiler generated code."

A	1	2	9	33	7	52	3.83	.78
R	0	2	19	24	5	50	3.64	.71

Very strong support for this proposal. No clear cut characterization of the dissenters except that they have strong assembly language experience.

Q26 - "The management of dynamically allocated stacks in support of re-entrant code will be done by compiler generated code unless the programmer has explicitly specified that some other process be used."

A	0	2	8	35	8	53	3.93	.67
R	0	3	22	20	4	49	3.51	.73

Two of the dissenters of the last question became enthusiastic supporters of this proposition. One stuck to his guns. He was joined by what had been a supporter of proposition 25 and now objected to this proposition because he felt that such freedom should not be had by the programmers.

Q27 - "An explicit declaration must be made to identify a routine as re-entrant."

A	1	13	4	26	7	51	3.49	1.07
R	0	4	16	25	6	51	3.65	.79

We expected and got support for this approach. The dissenting group as a whole was unexperienced compared to the other respondents.

Q28 - "All routines will be re-entrant unless explicitly declared not to be."

A	2	9	4	26	15	56	3.77	1.12
R	0	1	7	33	13	54	4.07	.66

This is effectively the converse to Q27. We expected disagreement with this proposition, but surprisingly, got more agreement than dissent. The dissenters in this case were all individuals who had supported Q27. The supporters of Q28 also included six individuals whose opinion on Q27 is not known. (Q27 came towards the end of the scrambled questionnaire.)

Q29 - "All routines will be re-entrant."

A	6	22	9	9	6	52	2.75	1.21
R	0	3	17	20	10	50	3.74	.84

We expected and got strong opposition to this proposal, especially since there are instances in which a routine should <u>not</u> re-enter (e.g., resource allocation). Sophisticated proponents of this statement offered the point of view that most of what we wanted in a non-re-entrant routine could be accomplished by forcing the user to get his resources by the executive program or by means of some other centralized facility which took care of the interlocks. Recognizing that this would increase processing overhead, it was worthwhile nevertheless. Contradicting this point of view, a respondent cited interrupt processing which involved manipulation of the same data base segment. Automatic and complete re-entrance would force this to go through a centralized exec also, and that the overhead caused by this would be odious.

4.7.4. Analysis

Routines can be made re-entrant by default, with an explicit declaration for non-re-entrant routines. These should be grouped (i.e., a global declaration that names which routines are not re-entrant). Compiler managed stacks in support of re-entrant code is probably acceptable and an option here is not necessary.

4.8. Storage Allocation (Q30-Q42)

4.8.1. Purpose of Questions

To determine whether or not there should be compiler generated code management of dynamically allocated storage pools and the extent of the programmer's control and options over this.

4.8.2. Comments

1. "The number of elements in use or available [for use, although not in use] in a compiler managed dynamic storage pool can be queued."

4.8.3. Questions and Responses

Q30 - "Compiler generated dynamic storage allocations shall be provided (and managed by compiler generated code) for any process that requests it by means of suitable declarations."

A	0	4	8	32	4	48	3.75	.72
R	0	4	21	19	3	47	3.45	.74

Q31 - "Dynamic storage allocations can be supplied at programmer's option for character strings."

A	0	2	7	33	14	56	4.05	.72
R	1	3	15	26	10	55	3.75	.88

Q32 - "Dynamic storage allocation can be supplied at programmer's option for task control queues."

A	3	2	6	35	6	52	3.75	.92
R	0	3	16	25	7	51	3.71	.77

Q33 - "Dynamic storage allocation will be supplied at programmer's option for all array or structures whose size has not been declared at compile time."

A	1	5	12	30	2	50	3.54	.80
R	0	5	22	17	4	48	3.42	.79

Q34 - "Dynamic storage allocation will be supplied by request for any definable data type."

A	1	5	10	36	2	54	3.61	.78
R	1	4	17	26	3	51	3.51	.80

Q35 - "Included in the specification of a data structure such as a queue block or buffer block is the specification of the programmer defined storage allocation and return routines to be used for the management of that data element."

A	2	10	14	19	0	45	3.11	.90
R	3	4	21	12	3	43	3.19	.95

There is general agreement with the idea of compiler generated code for dynamic storage allocation. This should not be too surprising since this is effectively what is happening now from the individual programmer's point of view. That is, storage request is now made via macros and subroutines and the storage management is effectively transparent to the application programmer. The dissenters tend to be inconsistent, agreeing in general, but not in specifics, or agreeing to specifics but not in general. The objection to Q35 is based on the problem of having to make numerous declarations. All of the above dealt with programmer's options and did not imply a forced usage of a built-in dynamic storage allocation scheme.

Q36 - "The programmer can specify the storage management routine to be used with any pool of data types. If no such specification is provided, the compiler will generate such facilities."

A	1	6	18	23	2	50	3.38	.82
R	2	8	22	15	2	49	3.14	.88

Moderate agreement. Part of the objection is based on not trusting the built-in capability. This is confirmed by correlation with Q39.

Q37 - "One and only one storage allocation and management scheme (routine) can be used with a given pool."

A	2	12	12	15	5	46	3.20	1.08
R	0	4	16	20	5	45	3.58	.80

The disagreement is based on numerous instances in which a given pool is subdivided in several different ways, in which variants are defined for the pool, and/or in which pool elements may be further split and allocated dynamically.

Q38 - "Any number of different storage allocations and management schemes (routines) can be defined for use with any given pool."

A	2	14	6	18	5	45	3.22	1.13
R	1	2	20	15	6	44	3.52	.87

Upon eliminating inconsistent responses to Q37/Q38, including those that did not answer both, we obtain a revised mean for Q37 of 3.03 and a preponderance of more experienced programmers objecting to the Q37 proposition. For Q38, the consistent group rejected the proposal. This was due mainly to the elimination of fence sitters. The mean was 2.9 with a decidedly bimodal distribution. There was no easy characterization of either group in response to Q38.

Q39 - "Any storage pool declared to be dynamic must have an associated allocation/management scheme or routine. Failure to declare one will be treated by the compiler as a bug."

A	3	10	4	30	2	49	3.37	1.04
R	0	6	13	27	3	49	3.55	.78

This seemed like a reasonable proposal. Of those objectors we queried, the lack of default was the cause.

Q40 - "The number of elements in use in a compiler managed dynamic storage pool can be queried."

A	0	0	13	28	14	55	4.02	.70
R	2	5	16	19	12	54	3.63	1.04

Q41 - "Compiler managed storage pools (e.g., stacks for re-entrant code) can be given attributes like any other data structure (e.g., read/write/execute privileges, security tags, special processing, etc.)."

A	0	7	6	33	2	48	3.63	.78
R	1	6	19	18	3	47	3.34	.86

No clear-cut characterization of the dissenting group here.

Q42 - "Compiler managed storage pools will transfer control to programmer specified routines upon detection of illogical conditions or when they run out of space."

A	0	5	2	38	11	56	3.98	.77
R	0	0	11	34	10	55	3.98	.62

The dissents here were sophisticated. We were not able to obtain the crux of their objections.

4.8.4. Analysis

There is general agreement with having compiler generated code for dynamic storage allocation as long as it is optional and does not entail too much writing in the form of declarations. It is enthusiastically supported for character strings, less so for task control queues, and meets with a general distrust as it becomes applied to more and more data types. Furthermore, there seems to be an awareness that all these goodies may require extra work to invoke or reject. They want to know, by some mechanism, when they have run out of space and want to be able to query the available space for each declared pool. There is some resistance to a lack of default. We are coming to the opinion that strong typing and explicit declarations, while desirable in the abstract, will meet with significant resistance because of the perceived extra labor it entails. (We say "perceived" because while it does entail extra work in source code production, it is hopefully paid for in reduced debugging.)

4.9. Initialization (Q43-Q51)

4.9.1. Purpose of Questions

To probe the kinds of initialization facilities that should be built-in, the kind of default necessary, or conversely, the resistance to explicit declarations.

4.9.2. Comments

1. "Initialization of memory is specified as part of the declaration [optional, I hope]."

4.9.3. Questions and Responses

Q43 - "Initial values must be explicitly declared for every data item, data structure, pools, etc. Reasonable abbreviations are available."

A	6	17	13	11	2	49	2.71	1.07
R	0	5	21	18	4	48	3.44	.79

General rejection of this proposal. This is possibly due to the use of the word "must" which tends to drop the agreement by several fractions. The issue is judged to be moderately important. The proponents of this proposal were generally more senior than the opponents. When responses inconsistent with Q43/Q44 were removed, the mean for Q43 changed to 2.69.

Q44 - "Initial values for data items, structures, pools, etc., need not be declared explicitly. The compiler will initialize to a default value of logical Ø."

A	5	7	5	30	9	56	3.55	1.16
R	0	3	21	25	6	55	3.62	.75

This is essentially the opposite proposition to Q43, which received significant backing. The dissidence reversed itself and was generally consistent. The mean for the consistent subgroup was 3.65. The shift was partly due to a consistent attitude regarding these two

contradictory propositions, but more important, it was due to fence sitters for Q43 who had a definite opinion in Q44. This is noticeable in the raw distributions, but even more noticeable in the sub-distributions of consistent answers.

Q45 - "When the compiler initializes a memory location to a default value, it will make an explicit statement to that effect to bring the fact to the programmer's attention."

A	1	4	6	31	9	51	3.84	.87
R	2	4	15	23	7	51	3.57	.95

This is an eminently reasonable proposal, and we would have expected overwhelming support, as we did get. The dissidents universally rejected Q43 and explicit declarations of initial values, and, with one exception, supported implicit default values consistently. Objection may be based on having too long a printout with excessive warnings. Perhaps the approach is to make this category of diagnostics optional.

Q46 - "Initialization of memory can be specified by means of a process which is executed at compile time (this does not preclude other methods of initialization)."

A	0	5	7	35	4	51	3.75	.74
R	1	6	19	21	1	48	3.31	.80

Strong support for this proposal. The dissidents are very experienced in communications. They felt the issue was not very important, along with the other respondents, although the dissidents felt it was less important th n the majority. The idea is that actual initialization must be done by a complex program. To have the data base initialized to anything but zero merely complicates the issue and the language. Members of this group were all strong supporters of initializations (by default) to zero.

Q47 - "Initialization of memory can be accomplished by a programmer specified run time process (does not preclude other methods of initialization)."

A	2	1	16	25	13	57	3.81	.93
R	1	13	20	14	9	57	3.30	1.04

Enthusiastic support for this proposition. Dissidents are very experienced in telecommunications. The basis of the objection is not known.

Q48 - "Initialization of memory is specified as part of the declarations."

A	1	8	16	26	6	57	3.49	.92
R	0	8	21	21	5	55	3.42	.85

Willingness to go along with this, but no strong support. Objections are based on the idea that in almost all cases, the proper initialization is to zero or to the complex, effectively arbitrary, values loaded by the data base support modules which are specific to the system, therefore, why have the complexity? The comparison with Q43 is instructive. This statement is essentially the same thing. In Q48 we adopted a lackadaisical tone and did not spell things out explicitly. In Q43, we adopted a stronger posture which did not differ in substance. Q43 looked like a lot more work than Q48.

Q49 - "The programmer can specify initialization processes to be executed on every entrance to a subroutine."

A	1	2	18	27	2	50	3.54	.73
R	1	12	17	18	1	49	3.12	.87

This was not deemed to be a very important issue. There was fairly strong support nevertheless. They'll go along with it and might use it if they had it, but they don't think it is essential.

Q50 - "The programmer can specify initialization processes that are executed by the loader at load time."

A	1	5	11	37	3	57	3.63	.79
R	1	9	18	26	2	56	3.34	.85

Stronger support for this, but no overwhelming enthusiasm. The dissident group has a fairly broad base and is not readily characterizable. None of them thought the issue was important.

Q51 - "Run time initialization routines can be given dummy names, with the actual process determined at run time as part of the calling sequence."

A	1	5	18	17	1	42	3.29	.80
R	1	7	23	9	0	40	3.00	.71

This had a lukewarm reception and was considered terribly unimportant. The dissidents are most of the same as those for Q50.

4.9.4. Analysis

They want a simple, implicit approach to initialization. They want a default initialization to 0 and do not want to be bothered with a whole lot of printouts that bring this to their attention. They want the ability to specify, explicitly, values other than zero, but do not want to be forced into it. Overall, initialization is not a burning issue.

4.10. Storage of Integers (Q52-Q55)

4.10.1. Purpose of Questions

To determine how much flexibility is required in the storage of integers, how much the programmer will allow the compiler to do, and how many restrictions they will accept regarding the storage of integers.

4.10.2. Comments

None.

4.10.3. Questions and Responses

Q52 - "The size of storage allocated to integers is determined by the compiler."

A	4	19	11	18	4	56	2.98	1.11
R	1	6	24	21	4	56	3.38	.84

We expected rejection of this proposition and got it. The negative group was slightly more experienced than those in favor. It is important to recall that most of the respondents would equate integers and pointers. This rejection is to a great extent due to the possibility that they would not know how the compiler would do the assignment and a desire to save space.

Q53 - "The programmer can specify the number of characters to be used to store any given integer."

A	1	6	9	28	3	47	3.55	.87
R	0	5	18	18	5	46	3.50	.83

Note the marked increase in approval when they are given some measure of control. There were few reversals. Almost all of the support came from individuals who had objected to lack of control and were happy to get some measure of control. The strongest dissident had wanted no control (answered 5 to Q52 and 1 to Q53). Most of the fence sitters stayed on the fence and those who didn't, generally moved in the direction of approval. Almost all of the objections came from individuals who had wanted the compiler to do the job in the first place.

Q54 - "The programmer can specify the number of bits to be used to store any given integer."

A	1	16	17	20	1	55	3.07	.89
R	1	9	23	20	1	54	3.20	.80

The idea of this question was to find out if we can have too much of a good thing. We expected that they would want some control, but did they feel that it was necessary down to the bit level? Those who had objected to character level control did not become

enthusiastic with the prospect of bit level control. The fence sitters moved slightly in the direction of favor to bit level control. No one got turned on by this offer - that is, there was no upward movement. The support to this proposition was entirely due to those who had supported character level control. The reduced enthusiasm was due to those who approved of character level control but did not want bit level control. The remaining supporters of bit level control were consistent with their dislike of lack of control as evidenced by their response to Q52.

Q55 - "The programmer must specify the number of characters to to be used to store integers."

A	6	24	6	16	0	52	2.62	1.04
R	2	6	21	20	2	51	3.28	.87

We offer this response in support of our contention that the word "must" is a trigger for disagreement. We phrased this in an obnoxious manner and got an expected result. Most of the supporters of this proposition (the minority) had favored character level control over the length of the integer. The major part of the erosion of support came from fence sitters who were now moved to voice an objection.

4.10.4. Analysis

Programmer control over the length of integers, at least to the specification of the number of characters, is essential. Finer grained control is not necessary. A default specification to the machine word length is probably desirable. This is not one of the burning issues in language design.

4.11. Arithmetic Overflow (Q56-Q60)

4.11.1. Purpose of Questions

To determine what (if anything) should be automated as regards the detection of arithmetic operation overflow.

4.11.2. Comments

None.

4.11.3. Questions and Responses

Q56 - "The compiler automatically inserts the code needed to check for arithmetic operation overflow."

A	7	14	9	19	6	55	3.06	1.24
R	3	7	19	20	5	54	3.31	1.00

Lukewarm support here as expected - they don't like the compiler inserting anything that is not under their control.

Q57 - "The programmer can specify whether run time code for arithmetic overflow detection should be inserted by the compiler."

A	0	2	13	32	3	50	3.72	.63
R	1	7	26	14	1	49	3.14	.76

Strong support for this proposition.

Q58 - "The programmer must specify the arithmetic overflow control routine to be used for reacting to integer overflow as part of the declaration of that integer variable. Suitable abbreviations exist."

A	6	19	13	14	0	52	2.67	.99
R	1	6	25	16	2	50	3.24	.79

Rejected, mostly for "must" and partially for having to write additional things. Some of the rejection may have been due to the interpretation of the statement, that is, it may have been interpreted as suggesting that some kind of overflow control routine was mandatory. A different phrasing of the statement might have elicited a more favorable response. The responses were unusual in that more inconsistencies in answers were found here than in any other question thus far. Approximately half of those who objected also objected to Q56, supporting the contention that some of them did see the two statements as equivalent.

Q59 - "The programmer can specify which arithmetic overflow control routine should be used for reacting to integer overflow, if none is specified, the compiler will use its own."

A	2	9	12	24	2	49	3.31	.95
R	1	9	26	8	2	46	3.02	.79

Comparing Q58 and Q59, we find that the issue becomes more palatable as soon as a default is given. Compare also to Q60.

Q60 - "The programmer can specify which arithmetic overflow concontrol routine should be used for reacting to integer overflow, if none is specified, the compiler will treat this as a bug."

A	3	14	17	19	1	54	3.02	.95
R	3	13	22	14	1	53	2.94	.90

4.11.4. Analysis

Arithmetic overflow is not one of the burning issues in language design. As usual, the flexibility is desirable; the additional code or specifications are not desirable, and defaults are valued.

4.12. Range of Integers (Q61-Q65)

4.12.1. Purpose of the Questions

To resolve the conflict between requiring a range specification for integers or a size specification. DSA and SOFTECH were of opposite opinions on this, with DSA favoring a size specification and SOFTECH favoring a range specification.

4.12.2. Comments

1. "Specification of range checks should be controllable and not automatic."
2. "Range checks at run time by compiler generated code should be an option and easily deleted."
3. "Might be OK [range attribute checking at run time by compiler generated code] if it were optional and could easily be changed after the debug phase.
4. "[...range attribute checking at run time by compiler generated code is...] wasteful, but maybe if optional."

4.12.3. Questions and Responses

Q61 - "The programmer must specify the range of values expected for integers as part of the declaration (e.g., between 47 and 96)."

A	6	16	10	18	0	50	2.80	1.06
R	1	5	21	17	3	47	3.34	.83

Q62 - "The programmer can specify both the range of expected integer values and the amount of storage (say, in bits) to be allocated to the integer. The compiler checks compatibility of the two declarations."

A	6	17	12	15	1	51	2.76	1.06
R	0	10	19	16	3	48	3.25	.85

Q63 - "The programmer specifies range of values for each integer and/or size of the field. The compiler performs all compatibility checks at compile time and flags conflicts."

A	3	3	8	30	10	54	3.76	1.00
R	2	4	14	24	9	53	3.64	.97

Q64 - "If the programmer has specified a valid range attribute for a parameter, the actual value of the parameter will be checked by compiler generated code at run time."

A	3	5	8	35	4	55	3.58	.95
R	1	7	18	25	4	55	3.44	.87

Q65 - "If the programmer has specified a valid range attribute for a parameter, the actual value of the parameter will be checked at run time by compiler generated code if, and only if, directed to do so by the programmer."

A	1	12	5	29	4	51	3.45	1.00
R	2	3	23	19	3	50	3.36	.84

4.12.4. Analysis

There was strong positive support for all proposals except Q61 and Q62. The issue was considered moderately important. A comparison of Q52,Q53,Q54,Q55 and Q61,Q62,Q63,Q64,Q65 shows

Q63 which allowed the programmer to specify range and/or size met with the greatest approval. Q64 which implied a default option also met with considerable approval. Requiring a specification of both range and value was rejected. The dissenters to Q65 had generally less experience than the proponent group. Our recommendations for this are:

1. Programmer can specify either range and/or size or neither. Programmer must have option to allow or forbid run time check code on range.

4.13. Miscellaneous Operations (Q66-Q69)

4.13.1. Purpose of Questions

To get reaction to specific arithmetic operation proposals.

4.13.2. Comments

1. "If this is all [exponentiation of the form 2^N, where N is an integer] you plan to do, why bother? Shifting can accomplish the same thing."
2. [Different symbols for integer and real multiplication is a] "Damned nuisance for nothing gained."

4.13.3. Questions and Responses

Q66 - "The language provides exponentiation of integers for powers of 2. For example 2^N, where N is an integer."

A	1	3	27	16	3	50	3.34	.76
R	2	14	23	7	2	48	2.85	.87

Considered to be a singularly unimportant issue, which met with general approval. How would they have reacted had we said that there would be no shift instruction? I suspect they did not realize that this was instead of a shift. Clearly, one or the other is essential and a shift is more natural.

Q67 - "The language provides logarithms rounded to next lowest whole number to base 2 for integers."

A	0	8	29	7	0	44	2.98	.58
R	8	17	14	5	0	44	2.36	.91

Even greater indifference. There was so much apathy to this question that the answer could have gone either way. If there is a means for detecting the most significant bit in an integer (efficiently) we can probably do without this.

Q68 - "The language provides ABSOLUTE VALUE operator for integers."

A	0	3	20	26	4	53	3.59	.71
R	4	10	20	16	2	52	3.04	.98

The absolute value operator met with greater approval and was considered slightly more important.

Q69 - "Arithmetic operations symbols for integers and floating point numbers are different. Example: "*" vs "x."

A	7	15	13	12	2	49	2.74	1.10
R	5	8	20	12	3	48	3.00	1.04

Another ho-hum issue, which met with some disapproval. Those who approved were a relatively experienced group taken as a whole. No simple characterization of their reasons. I suspect that it is mostly a matter of individual proclivities.

4.13.4. Analysis

Shift operations and most significant bit operations for integer must be provided. This can be done by exponentiation and logarithms. It would probably be acceptable, but would seem unnatural to the programmers. Unique symbols for operations between integers and floating point, etc. are undesirable and will probably cause more bugs than they eliminate.

4.14. Integer/Real Conversions (Q70-Q71)

4.14.1. Purpose of Questions

To examine attitude toward real/integer conversions.

4.14.2. Comments

1. [In converting from real numbers to integers, the compiler shall discard the fractional part of the number.] "ALWAYS?"

4.14.3. Questions and Responses

Q70 - "In converting from real numbers to integers, the compiler shall discard the fractional part of the number."

A	3	14	12	19	3	51	3.10	1.05
R	2	11	22	12	2	49	3.02	.89

Q71 - "In converting from real numbers to integers, the compiler shall round up or down to the nearest whole number."

A	1	9	14	22	1	47	3.28	.87
R	3	7	20	14	1	45	3.07	.90

4.14.4. Analysis

This was a ho-hum issue also. Rounding to nearest whole number met with more general approval than did truncation.

4.15. Representation Range of Floating Point Numbers (Q72-Q77)

4.15.1. Purpose of Questions

To find range and accuracy requirements for floating point numbers to test specific propositions in which floating point numbers are stored as two words of 32 bits each (mantissa and exponent).

4.15.2. Comments

1. [Five decimal digits of accuracy are sufficient for floating point...] "Not in common programs!"

2. [Real numbers shall be constrained to lie between $10^{\pm 38}$] "This spread is highly extreme!"

4.15.3. Questions and Responses

Q72 - "Five (5) decimal digits of accuracy are sufficient for floating point real numbers. There are no double precision operations."

A	7	9	15	14	6	51	3.06	1.21
R	8	11	16	9	5	49	2.84	1.20

Ho-hum response to an unimportant question.

Q73 - "Seven (7) decimal digits of accuracy are sufficient for floating point real numbers. There are no double precision operators."

A	3	11	16	10	2	42	2.93	.98
R	6	9	21	3	2	41	2.66	.98

Slightly more negative response of no statistical significance. Since this question came later in the scrambled version, not as many individuals answered it.

Q74 - "Nine (9) decimal digits of accuracy are sufficient for floating point real numbers. There are no double precision operators."

A	4	12	15	14	5	50	3.08	1.11
R	6	12	21	6	3	48	2.75	1.03

About the same response and enthusiasm.

Q75 - "There are significant number of instances in which more than 9 digits of accuracy are required for floating point real numbers."

A	9	22	9	3	2	45	2.27	1.00
R	8	10	16	9	1	44	2.66	1.06

This confirms the unimportance of the issue. Those who felt that there was a need for more than 9 digits were distinctly less experienced than the majority, who felt no such need.

Q76 - "Real numbers shall be constrained to lie between $10^{\pm 38}$."

A	2	8	28	10	2	50	3.04	.82
R	8	14	24	2	0	48	2.42	.81

Q77 - "Real numbers shall be constrained to lie between $10^{\pm 620}$."

A	8	8	17	6	6	45	2.87	1.24
R	7	16	13	5	2	43	2.51	1.04

4.15.4 Analysis

There will be no strong objections if double precision is not provided. The two word representation, one word for the mantissa and one word for the exponent will probably be acceptable.

4.16 Real Number Range Checking as Specification (Q78-Q81)

4.16.1 Purpose of Questions

To find the attitude towards compiler inserted code for real number range checking. To compare that attitude to integer range checking.

4.16.2 Comments

1. [The possible range of real numbers must be specified as part of the declaration.] "For math subroutines also???"

4.16.3 Questions and Responses

Q78 - "The compiler shall perform compile time range checks on the validity of the range of real numbers."

A	0	5	17	25	2	49	3.49	.73
R	3	10	19	12	1	45	2.96	.92

An issue of significant indifference when compared to the reaction vis-a-vis integers. They seem to be willing to let the compiler do the range checking and are willing to do the work to specify it.

Q79 - "The compiler will insert run time range checks on the validity of real numbers if directed to do so by the programmer."

A	1	3	19	24	10	57	3.68	.88
R	6	13	18	11	7	55	3.00	1.18

As usual, given additional control there is additional approval.

Q80 - "The possible range of real numbers must be specified by the programmer as a part of the declaration."

A	5	13	10	23	4	45	3.15	1.14
R	4	13	23	13	2	55	2.93	.95

As usual, a "must" dampens the enthusiasm.

Q81 - "The possible range of real numbers can be specified by the programmer as part of the declaration. If such a specification is made, the compiler will insert appropriate run time checking code."

A	2	4	12	28	2	48	3.50	.87
R	2	9	21	14	1	47	3.06	.86

This question generated the most enthusiasm. As expected, a default and a measure of control is what they are looking for.

4.16.4 Analysis

They agree, in general, with the principle of range checking for real numbers with compiler inserted run time check code if such insertions can be controlled by the programmer. Furthermore, defaults are desirable. The whole issue however, is considered to be very unimportant.

4.17. Boolean Flag Setting (Q82)

4.17.1 Purpose of Question

To determine the desirability of the proposed statement.

4.17.2 Comments

1. "Never necessary!"

4.17.3 Question and Responses

Q82 - "In addition to the appearance of the relation operators equal, not equal, greater, greater or equal, etc., in the context of a control statement such as "IF A GRT B THEN ... ELSE...." The language provided Boolean operator whose value is TRUE, if the relation is satisfied. Example: SET A TRUE IF B = C."

A	2	3	17	27	6	55	3.58	.89
R	3	4	18	26	3	54	3.41	.91

4.17.4 Analysis

Fairly strong approval to what was judged to be a moderately important issue. We could not find the basis for disapproval.

4.18 Range Checking Operations - Three way Conditional Branch (Q83-Q85)

4.18.1 Purpose of Questions

To determine the desirability of the three way range check statement for integers, reals and for all sensible data types.

4.18.2 Comments

1. [The range check instruction is...] "Not provided for integers?"

   Note: The order of the questions presented the proposal for reals and then much later for integers.

2. [The range check instruction is provided for reals only.] "BAD!!!"

3. [The range check instruction is provided for integers only.] "BAD!!!"

4.18.3 Questions and Responses

Q83 - "The language provides a range checking three way conditional statement of the form:

```
IF A ≤ X ≤ B IS LOW THEN DO...
             IS BETWEEN THEN DO...
             IS HIGH THEN DO...
```

This is provided for integers only."

A	2	13	14	18	2	49	3.10	.97
R	1	10	20	12	4	47	3.17	.93

Mild approval and mild interest.

Q84 - "The language provides a range checking three way conditional statement of the form:

IF A $\leq$ X $\leq$ B IS LOW THEN DO...
IS HIGH THEN DO...
IS BETWEEN THEN DO...

This is provided for real numbers."

A	0	4	14	32	5	55	3.69	.73
R	1	9	22	18	4	54	3.28	.89

Stronger approval, which is surprising, except that Q83 was restrictive and Q84 while equally restrictive, did not have a restrictive tone.

Q85 - "The language provided conditional statements of the form shown below for any data type which makes sense.

IF A $\leq$ X $\leq$ B IS LOW THEN DO...
IS BETWEEN THEN DO...
IS HIGH THEN DO...."

A	0	2	13	32	5	52	3.77	.67
R	1	6	21	21	2	51	3.33	.81

Strong approval to the general case. Dissenters were on the junior end of the scale.

4.18.4 Analysis

A generalized range checking statement would be considered helpful by the majority of the respondents, however, it is not considered among the more burning issues in communications programming.

4.19. Strong Typing (Q87)

4.19.1 Purpose of Question

To determine acceptability or resistance to strong typing.

4.19.2 Comments

None.

4.19.3 Question and Responses

Q87 - "The language does not permit implicit conversions from one data type to another (example, converting integers to real). All conversions must be explicit. This does not usually result in run time code. Mixed operations are forbidden."

A	2	19	6	18	3	48	3.02	1.09
R	1	6	16	19	4	46	3.41	.90

Strong division on an issue of moderate importance.

4.19.4 Analysis

We did not expect enthusiastic response to this proposition. The opponents and proponents are indistinguishable as a group. Strong typing will be accepted, albeit with considerable grumbling.

4.20. Specification of Bit Constants (Q88-Q93)

4.20.1 Purpose of Questions

To determine the most acceptable approach to the specification of bit field constants.

4.20.2 Comments

None.

4.20.3 Questions and Responses

Q88 - "Bit field or bit string constants can be specified in binary."

A	1	1	4	35	16	57	4.12	.75
R	1	3	15	27	9	55	3.73	.86

Overwhelming support for the ability to specify bit strings in

binary, also viewed as a fairly important issue. We were not able to find the basis for dissent.

Q89 - "Bit field or bit string constants can be specified in octal."

A	0	3	7	31	9	50	3.92	.74
R	1	7	15	20	5	48	3.44	.93

Strong support for non-restrictive octal specification, but noticeably less than that for binary specification. Octal is considered less important than binary.

Q90 - "Bit field or bit string constants must be specified in hexadecimal."

A	21	20	8	4	0	53	1.91	.92
R	3	6	16	21	6	52	3.40	1.02

There seems to be a hatred for hex. This was considered mildly important. The negative response is in part probably due to the word "must."

Q91 - "Bit field or bit string constants can be specified in binary, octal, or hexadecimal."

A	0	4	7	25	14	50	3.98	.86
R	0	1	17	24	7	49	3.75	.72

Giving them the freedom to use binary, octal or hex is not as enthusiastically supported as binary alone.

Q92 - "Bit field or bit string constants can be specified in any base (2,4,8,16, etc.) except 10."

A	8	14	13	15	4	54	2.87	1.19
R	3	10	20	16	3	52	3.12	.97

Given extreme freedom, they seem to balk at the additional complexity implied. Opinions are much further spread, but in gen eral, they do not like this.

Q93 - "Bit field or bit string constants are specified in decimal."

A	7	26	12	7	0	52	2.37	.88
R	2	6	19	20	2	49	3.29	.88

This was an outrageous proposal and we really did expect a more violent reaction.

4.20.4 Analysis

Whatever other mode of specifying bit field constants are provided, the ability to specify such fields in binary is considered very important. The objection to hex may be due in part to lack of familiarity by the group of respondents. Most of them have more experience with octal. Actually, things should be specified in a natural manner, and binary specification of bits is natural, far more natural then octal, hex, or decimal. There also appears to be a rational attitude regarding excessive freedom.

4.21 Operations on Incommensurate Bit Strings (Q94-Q98)

4.21.1 Purpose of Questions

To determine the desired approach to the handling of comparison logical operations on bit strings of different lengths.

4.21.2 Comments

1. [I am] "Not indifferent, but unknown way to proceed." [With respect to operations on bit strings of different lengths.]
2. "Why not [pad the shorter string on the] left [with zero bits]?"
3. [Padding of the shorter string is] "Usually a hardware function."
4. [The language will not permit bit field operations... of different lengths.] "[You mean] mandatory! Even with assembly inserts?"
5. "You keep trying to pad my updates of queue block elements and print strings!"

4.21.3 Questions and Responses

Q94 - "In operations involving two different bit fields where the fields are not of the same length, the compiler will pad the shorter operator on the left with zero bits."

A	3	16	10	21	3	53	3.09	1.07
R	0	7	19	22	3	51	3.41	.80

Q95 - "In operations involving two different bit fields where the fields are not of equal length, the compiler will pad the shorter string on the right with zero bits."

A	13	16	10	15	3	57	2.63	1.24
R	5	8	10	24	11	58	3.48	1.19

Q96 - "The language will not permit bit field operations (such as masks, AND, OR, etc.) on fields of different lengths."

A	18	23	5	8	0	54	2.06	1.01
R	3	2	12	24	13	54	3.78	1.03

Q97 - "In bit field operations where the fields are of different lengths, padding will be on the right or left of the shorter string, with zeros or ones as the programmer directs."

A	2	9	13	21	3	48	3.29	.98
R	1	4	18	18	6	47	3.51	.90

Q98 - "The compiler will not permit bit string operations on fields of different lengths. The programmer will use appropriate shift and pad operations to convert the strings to equal length strings."

A	10	32	3	11	0	56	2.27	.97
R	0	4	11	35	6	56	3.77	.73

4.21.4 Analysis

These were considered moderately important issues. The reaction was generally negative. I suspect that many of the readers did not really see the issue, but treated the whole thing as it would be done in assembly language. That is, they did not realize that some type of convention must be established with respect to operations on bit strings of different lengths. Proposition 97, which

must appear to be a lot of extra work, was given grudging approval. It is probably the way to go.

4.22 Occurrence of Bit String Operation (Q99-Q103)

4.22.1 Purpose of Questions

To determine the importance and frequency of different situations met with in bit string operations. There were no language proposals among these questions.

4.22.2 Comments

1. [There are a significant number of cases in which it is convenient to...give a bit substring a different name.] "If implemented properly!"

4.22.3 Questions and Responses

Q99 - "There are a significant number of cases in which there are bit string operations carried out on strings whose length is not known until run time."

A	1	12	12	21	6	52	3.37	1.02
R	2	5	17	17	7	48	3.46	1.00

Moderate support for this assertion. A lot of this depends upon the programmer's specific experience; not all programmers will have worked in areas requiring this. The key word, "significant" was objected to, it left them with a value judgement they were not willing to make.

Q100 - "There are a significant number of cases in which it is convenient to treat a portion of a bit string as a bit string in its own right, i.e., to give a substring a different name."

A	1	4	11	29	11	56	3.80	.90
R	1	6	16	21	9	53	3.59	.96

Strong support for this proposal/assertion - moderate importance.

Q101 - "There are a significant number of cases in which it is desired to operate on a substring of a bit string, but in which the length of the substring and its position within the larger string is not known until run time."

A	1	13	14	21	2	51	3.20	.93
R	2	7	21	16	3	49	3.22	.91

Again the wishy-washy agreement and lack of enthusiasm when the issues are not clear and based on specific experience.

Q102 - "There are a significant number of cases in which it is desirable to operate on a substring of a bit string, but in which the position of the substring within the larger string is not known until run time."

A	1	15	12	22	2	52	3.17	.95
R	1	15	12	21	1	50	3.12	.93

Almost the same answer as in Q101. This was probably one of the sets of questions in which they felt the same questions were being repeated.

Q103 - "There is a significant number of cases in which it is desirable to operate on a substring of a bit string where the position of the leading bit is known at compile time, but the length of the string is not known until run time."

A	2	11	16	17	3	49	3.16	.98
R	1	10	21	13	1	46	3.06	.82

Same type of answer here.

4.22.4 Analysis

The issue was probably obscure. The statements required a value judgement they were not prepared to make. The relevance was unknown to many of the respondents; more respondents left the relevance entry blank than for similar groups of questions. I don't think the questionnaire resolved anything here.

4.23 Character String Operations (Q104)

4.23.1 Purpose of Question

To determine the importance of character string operations.

4.23.2 Comments

None.

4.23.3 Question and Responses

Q104 - "The language will support operations on character strings, such as COMPARE, MOVE, INSERT, etc."

A	0	0	2	23	31	56	4.52	.57
R	0	0	6	26	24	56	4.32	.66

4.23.4 Analysis

The agreement with this proposition was overwhelming. The issue was felt to be extremely important.

4.24. Character String Length (Q105-Q107)

4.24.1 Purpose of Questions

To determine the need for dynamic storage allocation for character string operands.

4.24.2 Comments

1. "What happened to dynamic storage???" [In response to the proposition that character string lengths can be set at compile time.]

4.24.3 Questions and Responses

Q105 - "The length of all character strings is known at compile time."

A	6	26	5	10	1	48	2.46	1.02
R	0	2	18	23	5	48	3.65	.72

Strong disagreement here. The question was ambiguous. Those who thought that character strings referred to (among other things)

messages, disagreed. Those who felt that we were talking about constant strings, and that dynamic storage allocation was available for messages, agreed with this proposition. The question was therefore a washout. Those who agreed with the proposition were very experienced.

Q106 - "The maximum length of all character strings can be set at compile time."

A	5	7	9	29	1	51	3.28	1.05
R	1	4	23	19	4	51	3.41	.82

We expected and got stronger agreement with this proposition. The same ambiguity as in Q105 operated here also. The dissenters represented a fair cross section of the respondent set.

Q107 - "The programmer can specify if a character string is to be fixed or variable length."

A	0	1	7	39	6	53	3.94	.56
R	0	4	16	29	4	53	3.62	.73

Strong support for a programmer's option for dynamic allocation of character strings.

4.24.4 Analysis

The language should provide facilities for defining fixed length character string constants, variable length, dynamically allocated areas, and fixed length character string buffers. The issues were judged to be fairly important.

4.25 Code Set Selection (Q108-Q114)

4.25.1 Purpose of Questions

To determine which, if any, code(s) should be built into the language.

4.25.2 Comments

1. [The programmer can specify the code to be used...by declaring a table. ...code will be used in all character operations, particularly ordering tests.] "Within reason (e.g. alpha, numeric, etc.)."

4.25.3 Questions and Responses

Q108 - "The language will support ASCII character codes directly."

A	0	1	4	30	21	56	4.27	.67
R	1	0	6	27	22	56	4.23	.78

Only one dissenter who claimed he made an error, felt the language should support ASCII.

Q109 - "All operations involving characters will be based on the assumption that the characters are represented in ASCII."

A	8	15	5	21	3	52	2.92	1.24
R	0	2	13	25	11	51	3.88	.78

Slight negative reaction to this proposition. Fairly balanced between those who feel they can live with built-in ASCII and those who feel they cannot. The comparison with Q108 is instruction. Q109 implies a restriction to ASCII, while Q108 is ambiguous.

Q110 - "The language will directly support ITA-2 codes."

A	1	4	12	20	7	44	3.64	.93
R	1	7	12	16	7	43	3.49	1.02

Strong disagreement with ITA-2 code. The point is that in most modern systems, an attempt is made to convert to ASCII for internal processing. If they had a code conversion mechanism, they could live without ITA-2 support built in.

Q111 - "The language will directly support ITA-5 code."

A	1	6	10	19	8	44	3.61	1.00
R	2	5	12	15	10	44	3.59	1.09

More support for ITA-5 than for ITA-2, but nowhere near as much for ASCII. Note that ITA-5 differs from ASCII only in insignificant details, being the international telecommunications union version of the universal code. The opponents to ITA-5 were distinctly less experienced as a group than the proponents.

Q112 - "The programmer can specify the character code to be used in character related operations by declaring a table. The programmer specified code will be used in all character operations, particularly ordering tests."

A	0	7	11	29	5	52	3.62	.84
R	0	4	22	22	3	51	3.47	.72

Several respondents who had objected to this proposal had interpreted the statement to mean that either the code set specification would have to be made for every operation, or that the code set specification would have to be made for every routine or program. Given that the code set specification could be made system wide, globally, they reversed their position. On this basis, we have strong support for this proposition.

Q113 - "The compiler will assume character operations to be in ASCII unless specifically declared to be in some other character code.

A	1	6	10	30	5	52	3.62	.88
R	1	7	14	25	5	52	3.50	.91

Strong support for a default of ASCII.

Q114 - "The compiler will assume character operations to be in a programmmer specified default character set if no specification is made. Character operations can be directed to be in terms of any character set, but this takes additional writing."

A	0	6	6	32	3	47	3.68	.77
R	0	5	19	20	2	46	3.41	.74

This is the clearest statement of the lot, and met with the greatest approval. Objections are based on just how this it to be done. They would not like to have to make the specification of alternate

code sets for every operation, but would like to be able to assign a scope.

4.25.4 Analysis

A code set should not be built-in. The compiler should assume a default code set which is defined as part of the declarations. It must be possible to override the use of the system wide default code set, by specific declarations to that effect. This would allow the definition of subroutines that operated in other than the main code set. Building in a code set would reduce the applicability of the language; ASCII, while dominant in the United States, is not universal, and there are other applications which require other codes. Interface with international networks in particular require ITA-2, ITA-5, etc.

4.26 Character Packing (Q115-Q116)

4.26.1 Purpose of Questions

To determine whether packing should or should not be built-in.

4.26.2 Comments

None.

4.26.3 Questions and Responses

Q115 - "Characters will be packed into 8 bit fields aligned on physical character, or word boundaries."

A	0	6	11	28	6	51	3.67	.83
R	0	6	17	22	4	49	3.49	.81

Q116 - "Characters will be packed into the tightest packing allowed by the character set. The compiler will automate all packing, unpacking and extraction instructions."

A	8	13	13	18	4	56	2.95	1.19
R	1	6	20	22	6	55	3.47	.89

4.26.4 Analysis

Clear-cut support for non-packing of characters and treating them as 8 bit fields. A few respondents reversed themselves upon being confronted with their answer to Q115. There was nevertheless support for Q116. What they would probably like to have is a default in which characters are placed in accordance to "natural" boundaries as well as the option for involving packed operations. This will become increasingly unimportant during the language's period of deployment. I expect that eventually, at least by the mid-80's, they will no longer attempt to pack characters than they would attempt to pack bits.

4.27 Character String Search Operations (Q117-Q121)

4.27.1 Purpose of Questions

To determine what should be offered for string search operations.

4.27.2 Comments

1. [The language should contain an operation that searches a character string for the appearance of a given character sequence and ...] "Returns the position [of the string] and the number of characters skipped [in the test string]."
2. [The language searches for the appearance of a specified substring. It returns either the position at which the substring was found, or an indication that no such string was found.] "And how far the search progressed." [If it failed.]

4.27.3 Questions and Responses

Q117 - "The language has a character string operation that scans a specified string in reverse order for the appearance of a given substring."

A	1	8	13	25	3	50	3.42	.90
R	0	10	18	20	2	50	3.28	.83

General support for what they felt was an operation of dubious value.

Q118 - "The language contains an operation that searches a character string for the appearance of a specified substring. It returns either the position at which the substring was found, or indication that no such string was found."

A	0	2	1	38	13	54	4.15	.62
R	0	1	8	31	14	54	4.07	.69

Very strong support for an important issue. Objections may have been due to misinterpretation of the statement.

Q119 - "The language contains an operation that searches a character string for the appearance of a given character sequence and returns the character string leading to the first appearance of the searched for string."

A	1	9	12	26	2	50	3.38	.89
R	1	7	15	25	1	49	3.37	.83

This is viewed as less important, but generally useful.

Q120 - "The language contains an operation that searches a character string for the appearance of a given character sequence and returns the character string following the first appearance of the searched for string.

A	1	4	4	33	15	57	4.00	.88
R	0	2	16	26	13	57	3.88	.80

Strong support here again.

Q121 - "The language contains an operation that searches a character string for the appearance of a given character sequence starting with a specified number of characters past the beginning of the searched string.

A	1	4	9	34	5	53	3.72	.81
R	1	7	12	27	5	52	3.54	.91

The proposition again received strong support and was considered reasonably important.

4.27.4 Analysis

The language should incorporate as a minimum an operation such as was proposed in Q118. This should be augmented with sufficient

string manipulation primitives to allow the efficient construction of other optional strings, e.g., BEHEAD, CURTAIL, APPEND, ABEND, etc.

4.28 String Comparisons (Q122-Q126)

4.28.1 Purpose of Questions

To determine what rules should be built in for comparing incommensurate strings.

4.28.2 Comments

1. [In operations involving character strings the compiler will generate code that pads the short strings with logical blanks on the right.] "Or left, or none?" [by programmer option].
2. [Padding of character strings for comparisons and move is...] "Usually done by hardware."
3. [The compiler will generate code that will pad the short string...] "Maybe overlays are needed for print lines of text."
4. [The compiler will generate code...] "BAD!"

4.28.3 Questions and Responses

Q122 - "In operations involving character strings of different lengths, such as a move operation, truncations and padding is never allowed."

A	10	37	3	3	0	53	1.98	.69
R	1	1	16	30	5	53	3.70	.74

Q123 - "In operations such as a move involving character strings, the compiler will generate code that will execute a branch to a specified location if the source string is longer than the target string."

A	3	12	13	21	2	51	3.14	1.01
R	1	4	24	18	3	50	3.36	.79

Q124 - "In operations such as a move involving character strings, the compiler will generare code that will pad the shorter strings with logical blanks on the right.

A	4	11	9	27	4	55	3.29	1.09
R	0	5	26	20	5	56	3.45	.78

Q125 - "In operations such as a move instruction involving character strings of different lengths, the compiler will generate code that will pad the short string on the right with logical zeros."

A	7	24	7	10	1	49	2.47	1.03
R	0	3	20	21	5	49	3.57	.76

Q126 - "In moving a long string into a short string area, the compiler will generate code that will cause a branch to a specified location if the truncation of the longer string requires the removal of non-blank characters. Otherwise, it will truncate characters on the right and perform the move."

A	1	10	20	17	0	48	3.10	.80
R	0	7	17	21	2	47	3.38	.79

4.28.4 Analysis

The strong opposition to the proposal for bit strings are repeated here. We have at best grudging approval. We are not doing things right, or not presenting them correctly, or both. Perhaps we should specify options with respect to bit and character move operations, or better yet, distinguish between a MOVE and a MERGE.

MOVE - Inserts the source string into the destination string from stated (bit or character) position.

OPTIONS - curtail source string tail if source is longer.

- expand destination string if source is longer.

MERGE - Inserts the source string into the destination string from stated (bit or character) position.

OPTIONS - blank remainder of source string, if source is shorter.

- blank initial source string if source is shorter

- blank destination tail, if source is shorter.

I do not propose the above in detail or as final, but we need to rethink this bit and character string operation.

4.29 Pointers (Q127-Q134)

4.29.1 Purpose of Questions

To probe the whole idea of pointers, pointer restrictions, strong typing of pointers, checking, indexing, etc.

4.29.2 Comments

1. "Who decides?" [...when it is necessary to have the compiler insert run time check code on the validity of pointer values.]

2. "Should the compiler care?" [...if incrementing a pointer results in pointing to the next item rather than the next word or character in memory.]

4.29.3 Questions and Responses

Q127 - "Pointers to data base structures and fields will be *distinguished from integers and other data types.*"

A	2	4	12	24	12	54	3.74	1.00
R	2	5	14	25	6	52	3.54	.95

Some controversy, but general agreement with the concept. No obvious characterization of the opponents.

Q128 - "The same pointer to data base elements may be used to point to fields having different characteristics, or may be functionally incompatible. That is, to a bit field or to an integer, say."

A	4	11	7	19	7	48	3.29	1.21
R	1	8	18	15	5	47	3.32	.95

More divisiveness here with only moderate agreement.

Q129 - "A given pointer to a data base element always points to one and only one kind of data base element or field."

A	2	17	11	18	2	50	3.02	1.01
R	2	5	17	20	5	49	3.43	.95

This is a strong typing issue - it is generally rejected.

Q130 - "Incrementing a pointer should result in pointing to the next item of the same kind in the table rather than to the next word or character in memory."

A	2	10	4	31	8	55	3.60	1.06
R	1	6	12	27	9	55	3.67	.94

Strong support for this proposition. Some persons want to point to physical rather than logical things. This is considered a moderately important issue.

Q131 - "Pointers to data base elements will be packed to conserve space."

A	4	9	20	12	0	45	2.89	.90
R	0	9	19	15	1	44	3.18	.78

Massive indifference bordering on disapproval for a "who-cares" issue.

Q132 - "Pointers to data base elements will be packed to conserve space if so directed by the programmer."

A	0	7	15	30	4	56	3.55	.80
R	3	5	22	21	4	55	3.33	.94

That woke them up!

Q133 - "The programmer specifies the permissible range of values of pointers to the data base. The compiler performs compile time checks (as possible) and inserts run time check codes as necessary."

A	0	9	13	26	1	49	3.39	.80
R	0	8	23	15	1	47	3.19	.73

Q134 - "The insertion of run time check code to check the validity of a pointer (to a data base element) is controlled by the programmer."

A	1	13	15	16	4	49	3.18	.98
R	1	3	21	20	3	48	3.44	.79

Just when you think you understand the issues, they throw you for a loop. One would have expected stronger support for programmer control, but we have just the opposite. Perhaps it was the fact that Q133 implied more explicit work.

4.29.4 Analysis

There is grudging acceptance of pointers as a type with the usual conflict between safety and ease of programming. The entire issue is not considered crucial. They want, as usual, control.

4.30 Subroutine Optional Parameters (Q135-Q145)

4.30.1 Purpose of Questions

To help select one of several alternative approaches to the treatment of optional and mandatory parameters.

4.30.2 Comments

None.

4.30.3 Questions and Responses

Q135 - "Every parameter that appears in a subroutine definition must appear in every call to that subroutine."

A	5	27	8	9	1	50	2.48	.96
R	0	5	14	26	4	49	3.59	.78

Strong opposition to lack of optional parameters.

Q136 - "Parameters of a subroutine call can be labeled as optional or mandatory. A call must have all mandatory parameters."

A	0	6	5	32	11	54	3.89	.85
R	0	4	11	30	7	52	3.77	.77

Strong support to optional parameters.

Q137 - "If parameters are omitted in subroutine calls, their place is denoted by means of a special character, such as:

CALL SUBROUTINE SAME (TOM,*, HARRY)."

A	3	17	10	19	1	50	2.96	1.02
R	2	8	21	16	2	49	3.16	.89

Q138 - "If paramters are omitted in a subroutine call, and the parameters in question are optional, they are left out of the call, as in the following example:

CALL SUBROUTINE SAME (TOM,,HARRY)."

A	1	6	11	32	5	55	3.62	.86
R	0	6	21	23	2	52	3.40	.74

Dislike of special characters to denote missing parameter, approval of leaving it blank with position denoted by space between commas.

Q139 - "There are instances in which it is desirable to define subroutines in which some parameters are optional and others are mandatory."

A	1	1	6	36	5	49	3.88	.69
R	0	6	15	23	5	49	3.55	.83

Additional support for optional parameters.

Q140 - "There are instances in which it is desirable to define subroutines in which the number of parameters is variable."

A	2	3	8	35	7	55	3.76	.87
R	1	6	21	22	5	55	3.44	.87

Strong support for the idea of a variable number of parameters. We do not concur. I believe that the issue has been misunderstood, that there is some confusion between parameter values and number of parameters. This should be followed up with requests for specific examples and instances.

Q141 - "There are instances in which it is desirable to define subroutines in which the number of parameters is variable and cannot be determined until run time, that is, the number of parameters may itself be a parameter of the call."

A	2	8	7	25	4	46	3.46	1.02
R	1	8	22	11	4	46	3.20	.90

There is still moderate support for the proposition of variable number of parameters. But note that when the issue is made clearer, some support is eroded.

Q142 - "There are instances in which it is desirable to define subroutines in which the maximum number of parameters cannot be determined and the actual number of parameters is not known until run time."

A	1	14	12	22	2	51	3.20	.95
R	3	9	20	16	2	50	3.10	.94

Continuing along the same line with a more explicit statement, we get a further erosion of the support.

Q143 - "The number of parameters to a subroutine call need not be constant nor be specified until run time. However, an explicit declaration to this effect is required. Subroutines with fixed number of parameters can of course be written."

A	4	3	14	28	8	57	3.58	1.03
R	3	13	18	19	3	56	3.11	.99

Assuming that the number of parameters can be variable, they are willing to treat it as a special case.

Q144 - "If and when a subroutine is defined which has a variable number of parameters, some of which are optional and some of which are mandatory, the programmer must specify how the deleted parameters are to be handled."

A	3	14	16	14	6	53	3.11	1.09
R	1	7	25	16	3	52	3.25	.83

But not if they have to work too hard at it.

Q145 - "If and when a subroutine is defined which has a variable number of parameters, some of which are mandatory and others of which are optional, the handling of linkages for deleted optional parameters shall be controlled by the compiler."

A	1	15	7	20	3	46	3.20	1.03
R	1	5	23	12	4	45	3.29	.86

But on the other hand, they don't really trust the compiler.

4.30.4 Analysis

Subroutines may have mandatory and optional parameters. The mandatory parameters should be the default case. A declaration of optionals should be required. Missing parameters can be left out of the call, but no special symbol should be used. A value convention for missing parameters must be decided upon.

While there is still a claim for a need for subroutines with variable number of parameters, I believe that this is not the case. We must find specific examples. They do not think that this is a terribly important issue anyhow.

4.31 Subroutine Entrances and Exits (Q146-Q150)

4.31.1 Purpose of Questions

To determine if multi-exit and/or multi-entrance subroutines are necessary; how much objection would there be if they were not provided.

4.31.2 Comments

1. [Subroutines must have a single entrance and exit. Multiple entrances and exits are simulated by an entrance and/or exit parameter.] "ALWAYS?"

2. [Subroutines may be defined as having any number of entrances and/or exits.] "Within reason!"

3. [Subroutines may be defined as having any number of entrances.] "Within reason."

4. [The use of entry and/or exit parameters to simulate multi-entry multi-exit subroutines is...] "VERY BAD!"

4.31.3 Questions and Responses

Q146 - "Subroutines must have a single entrance and exit. Multiple entrances and exits are simulated by an entrance and/or exit parameters."

A	2	21	10	15	7	55	3.07	1.14
R	0	3	21	24	7	55	3.64	.77

Very grudging agreement here on an important issue.

Q147 - "Subroutines may be defined as having any number of entrances."

A	8	15	5	19	4	51	2.92	1.27
R	1	2	15	27	5	50	3.66	.79

Q148 - "Subroutines may be defined as having any number of exits."

A	11	11	5	21	9	57	3.10	1.40
R	0	1	13	26	16	56	4.02	.77

Q149 - "Subroutines can be defined as having any number of entrances and/or exits."

A	8	15	5	22	2	52	2.90	1.21
R	1	2	16	24	9	52	3.73	.86

Q150 - "If multiple entrances and exits can be defined for subroutines, there is no need to have subroutines with a variable number of parameters in the call."

A	9	23	9	8	0	49	2.33	.96
R	2	7	13	24	3	49	3.39	.94

4.31.4 Analysis

There is a lot of controversy here and no clear cut answers. The shifts from approval to disapproval are small. The only thing they seemed to agree on was to disagree with the relation between multiple entrances and a variable number of parameters in the call. This last statement was inserted on the basis of the first round discussions and interviews, where it appeared that this proposal would be acceptable. I suspect that this issue is still not clear and the questionnaire has not helped to resolve it. The violent opposition to Q150 was totally unexpected. Therefore, we followed up the entire issue with additional informal interviews. The following resulted:

There was confusion with parameters and parameter values. We were thinking that pointers could be parameters; they thought that it was not possible to pass pointers to dynamic areas. We asked for specific examples of variable numbers of parameters in a call and were given things like, "the entries of a queue," "the RI's in a message," "the blocks of a message," etc. Upon further explanation they reversed their stand. Consequently, as long as reasonable objects, such as pointers can be parameters of a call (we assume that all objects can be call parameters, including lists) then there should never, never, be a need for variable number of parameters in a subroutine definition.

4.32 Moving Structures (Q151-Q152)

4.32.1 Purpose of Questions

To determine if it is desirable to be able to move arrays and structures, rather than just their names.

4.32.2 Comments

None.

4.32.3 Questions and Responses

Q151 - "The language must have the ability to pass arrays (bit strings, tables, character strings, vectors, etc.) as parameters. Not to be confused with passing the name of an array."

A	0	9	16	21	3	49	3.37	.85
R	1	6	18	18	3	46	3.35	.87

Q152 - "Arrays of bits or characters, words, etc., can be moved assigned, passed and operated in much the same manner as individual words, characters or bits."

A	0	0	5	41	12	58	4.12	.53
R	0	0	15	33	9	57	3.90	.64

4.32.4 Analysis

It is possible that the negative replies to Q151 stem from an implication that it would not be possible to pass the name of an array, but only the array itself.

4.33. Indexing Character Strings (Q153)

4.33.1 Purpose of Question

To determine desirability of allowing character string indexing.

4.33.2 Comments

None.

4.33.3 Question and Responses

Q153 - "Character strings can, at the programmer's option, be treated as arrays and indexed as such."

A	0	0	5	39	7	51	4.04	.48
R	0	4	17	25	5	51	3.61	.77

4.33.4 Analysis

Very strong support for this proposal - no dissenters.

4.34 Indexing (Q154-Q158, Q162)

4.34.1 Purpose of Questions

To probe various aspects of indexing and index variables.

4.34.2 Comments

1. [The language will not have indexing modes - access is provided by a compound name of the form "TABLENAME.TABLEENTRY."] "A la Pascal???"

2. "If you insist [...names of the form TABLENAME.TABLEENTRY instead of indexing] at least allow either [upper or lower] case."

4.34.3 Questions and Responses

Q154 - "Considering index variables used to access elements of an array: it is necessary to be able to have constant value indexes."

A	1	8	13	18	2	42	3.29	.91
R	1	3	24	12	2	42	3.26	.76

This was not considered particularly important. I think the dissenters are mistaken. They would like to be able to avoid the circumlocution of spurious assignment statements that would come about if they had not the ability to allow constants in an address term.

Q155 - "Considering index variables used in access elements of an array: it is necessary to be able to add such variables."

A	0	3	10	28	5	46	3.76	.73
R	0	2	22	16	5	45	3.53	.75

Stronger support when we ask for index addition.

Q156 - "Index arithmetic can be complicated and may require general arithmetic operations or expressions, such as A+B*C."

A	0	2	13	31	9	55	3.85	.72
R	2	3	21	24	4	54	3.46	.85

Even stronger support for a more general index arithmetic.

Q157 - "Indexing operations (for arrays, say) can be complicated and could require the execution of a function or subroutine."

A	0	5	9	31	3	48	3.67	.74
R	1	3	28	12	3	47	3.28	.76

Damping of enthusiasm for indexing subroutines.

Q158 - "Indexing operations (for arrays, say) can be complicated and could require the execution of one of several functions or subroutines whose identity in any specific instance is not determined until run time."

A	0	8	12	23	4	47	3.49	.87
R	1	8	20	14	2	45	3.18	.85

We suspect they are hedging their bets and that this functional capability is not really necessary.

Q162 - "The language will not have indexing modes of address. Instead, individual entries in a table are named and access is provided by a compound name of the form:

"TABLENAME.TABLEENTRYNAME"

A	15	25	6	9	0	55	2.16	1.00
R	0	4	9	26	14	53	3.94	.86

We were surprised that anybody would go along with this "modest" proposal. Those who agreed with the proposal included a PASCAL buff, a few COBOL buffs, a sprinkling of individuals whose problems and proclivities were not discernable. I suspect here an ability to "live with" the proposal rather than to like it.

4.34.4 Analysis

Moderate index arithmetic facilities should be provided; probably no more than would come about in the interest of uniformity. It does not appear to be a burning issue. The compound name instead of indexing is clearly rejected.

4.35 Common Parts of Variant Structures (Q159-Q160)

4.35.1 Purpose of Questions

To determine if it is possible to group common parts of variant structures.

4.35.2 Comments

1. [Grouping of common parts of variant structures, followed by variable parts] "Implies [variable part] entry size differences."

4.35.3 Questions and Responses

Q159 - "In defining a table, such as a queue table in which there is likely to be several alternative formats and interpretations, it is always possible to arrange things so that the common parts (among these variations) are grouped at the beginning of the entry followed by the variable parts."

A	6	9	8	18	15	56	3.48	1.32
R	3	8	15	19	11	56	3.48	1.12

General agreement to this statement.

Q160 - "In tables whose entries can be interpreted in a variety of ways depending on use (such as a queue block) the common parts of these variant structures may appear in the beginning, middle, or end for good reasons."

A	6	5	15	19	8	53	3.34	1.18
R	4	2	20	20	7	53	3.45	1.02

They agree with this too.

4.35.4 Analysis

This pair of statements yielded the most inconsistent set of answers thus far. Eighteen responses were clearly inconsistent. Only sixteen responses were consistent. Among the consistent responses those in favor of keeping common parts at the top averaged 3.44, while those who sought the flexibility averaged 2.26. The problem was probably with Q160, which in part was interpreted as if the compiler would assign the common parts in accordance with its own whim without giving the programmer any say in the matter.

4.36 Names of Things (Q161-Q163)

4.36.1 Purpose of Questions

To determine attitude towards the construction of names.

4.36.2 Comments

1. [Building names of the form IMA.YANKEE] "Do it carefully."

2. "Listings [that contain names like IMA.YANKEE] tends to lose readability.

4.36.3 Questions and Responses

Q161 - "In addressing table names of entries can be built up in a simple manner. For example, if the name of a table is 'IMA' a specific entry in that table might be named 'YANKEE' while another might be named 'JKL.' This would result in names in the form of 'IMA.YANKEE' or 'IMA.JKL.' This feature is in addition to the typical array indexing features of most language."

A	0	2	12	35	2	51	3.72	.60
R	0	7	21	21	1	50	3.32	.73

Strong support for a moderately interesting feature.

Q163 - "A simple scheme for building up names of things will be provided. Of the form, NNNN.AAAA.BBBB, resulting in names such as: THIS.IS.TABLE1"

A	1	2	17	23	3	46	3.54	.77
R	2	7	18	15	2	44	3.18	.91

Moderate interest in this feature.

4.36.4 Analysis

It is clear from the response to Q162, in the context of indexing that if this were the only way of getting at components of structures, they would reject it violently. Since it is an additional mode of naming, and does not imply restrictions, they will go along with it for the ride. The point is, that this should probably not be implemented if this is its only purpose. To get a better gauge of this proposal would require giving the respondents some specific examples and advantage. The present statement was too vague to elicit a response.

4.37 Length of Arrays, Number of Entries in Structures (Q164-Q168)

4.37.1 Purpose of Questions

To determine if automatic, dynamic allocation, not under the programmer's control, is necessary for structures in general.

4.37.2 Comments

1. [The length of all arrays and structures can be specified at compile time...] "Can be, but requires variable length storage if not."

2. [The maximum length of all arrays and structures can be specified at compile time.] "I WANT DYNAMIC STORAGE!!!"

4.37.3 Questions and Responses

Q164 - "The length (e.g., number of entries) of all arrays and structures can be specified at compile time."

A	2	8	7	33	7	57	3.61	.99
R	1	5	14	31	6	57	3.63	.85

General agreement with this statement.

Q165 - "The length (e.g. number of entries) of most arrays and structures can be specified at compile time."

A	3	5	6	34	2	50	3.54	.94
R	1	0	24	17	6	48	3.56	.79

Although less agreement with the more general statement, it's surprising. There were seven more answers to Q164 than Q165, because of the placement of the questions in the scrambled Questionnaire. Ten respondents had clearly inconsistent answers. Based on the consistent answers, the mean for Q164 was 3.37 and for Q165, 3.93.

Q166 - "The maximum length (e.g., number of entries) of all arrays and structures can be specified at compile time."

A	3	5	6	34	3	51	3.57	.95
R	0	5	23	18	5	51	3.45	.80

Q167 - "The maximum length (e.g., number of entries) of most arrays and structures can be specified at compile time."

A	1	6	6	35	5	53	3.70	.86
R	0	1	22	22	6	51	3.65	.71

Here the number of respondents were closer and the results more consistent.

Q168 - "There are arrays and structures whose length (e.g., number of entries) can only be determined at run time."

A	4	7	6	29	7	53	3.53	1.11
R	2	3	18	23	7	53	3.57	.92

4.37.4 Analysis

This is a set of inconclusive results. The issues appear to be of moderate interest. It is probably safe to establish upper bounds to all arrays and structures. After all, this is what they are living with now in assembly language.

4.38. Operations on Data Structures (Q169-Q173)

4.38.1 Purpose of Questions

To examine the need for complex MOVE, LOAD, STORE and COMPARE operations on general data structures.

4.38.2 *Comments*

1. [...control over which fields should be moved, if they are of different types, should be...] "Optional (re: Q230)."

2. [In complex moves of data structures involving fields of different types] "Moving from real to integer, etc., compiler should do conversion."

4.38.3 Questions and Responses

Q169 - "It is possible to perform assignment operations (e.g., move, load) on entire structures or elements thereof (e.g., queue blocks, loaded with a constant entry in one statement."

A	1	0	10	32	5	48	3.83	.69
R	1	4	18	20	4	47	3.47	.85

Although they felt this was only a moderately interesting feature, they liked the general idea. We do not know the basis of the lone dissenter's objections.

Q170 - "In complex move operations involving a data structure (say from a queue block to another queue block) the move will be allowed if and only if the target structure and the source structure are of the same type."

A	3	18	5	24	4	54	3.15	1.13
R	0	2	19	27	5	53	3.66	.70

Notice the objection to a restriction. This was also considered to be important.

Q171 - "In a complex move operation (say, moving a queue block into another queue block) only those fields which are of the same type and name will be moved, the rest will not participate in the update process."

A	1	14	10	20	2	47	3.17	.98
R	0	4	18	22	2	46	3.48	.71

Offering to move the type matched elements did not materially remove the objections. They were less sure of their position and considered this question significantly less important than Q170.

Q172 - "The language provides a comparison operator for structures that allow a specific queue block to be compared to a stored skeleton."

A	0	2	16	24	6	48	3.71	.73
R	3	6	16	19	1	45	3.20	.93

The general proposal is accepted, but this is not considered particularly important.

Q173 - "The language provided a comparison operator that allows a specific queue block to be compared to a stored skeleton. The comparison is done only for those fields that are compatible and have the same name. The rest of the fields do not participate in the comparison."

A	1	4	9	33	2	49	3.63	.77
R	1	5	14	26	2	48	3.48	.82

Some damping of enthusiasm when restrictions and strong typing is implied. The question becomes more important, also.

4.38.4 Analysis

There is not too much enthusiasm for complex structure move and compare operations. Furthermore, what interest there is becomes significantly dampened when realistic contraints vis-a-vis type checking, etc., must be imposed. I suspect that we shall have to come up with specific proposals if a more definitive answer is to be gotten. It is probably best to let such complex operations be a spin-off in the interest of structural uniformity, rather than an overt characteristic of the language.

4.39 Nested IF THEN Statements (Q174)

4.39.1 Purpose of Question

To determine the desirability of nested IF-THEN-ELSE statements.

4.39.2 Comments

None.

4.39.3 Question and Responses

Q174 - "It is necessary to nest conditional branch statements as in the following example:

IF A=B THEN [IF C=D THEN DO...ELSE...] ELSE DO...."

A	3	14	5	26	5	53	3.30	1.13
R	1	3	18	26	3	51	3.53	.78

4.39.4 Analysis

Split opinion with moderate support for a moderately important function. Not too many fence-sitters. The opposing group was

on an average more experienced than the favoring group. It is never really necessary to construct such nestings, merely convenient. Nestings of this kind are contrary to clean coding and legibility. I would recommend not putting in this feature, unless it happened to come out as a spin-off.

4.40. Control Structure Delimiters (Q175-Q177)

4.40.1 Purpose of Questions

To determine the desirability of explicit control structure delimiters and their desired format.

4.40.2 Comments

1. "Termination symbol is a good idea."
2. "Be careful." [with explicit termination symbols]
3. [The control symbol written backwards as a terminator makes the...] "Program becomes unreadable."

4.40.3 Questions and Responses

Q175 - "Control structures such as loops will be terminated with an explicit termination symbol such as BEGIN/END, or CALL/RETURN."

A	1	2	6	28	19	56	4.11	.86
R	4	2	13	28	10	57	3.67	1.03

Very strong support for a moderately important issue. Of the three dissenters, two objected to the specific form of terminator given in the example and not to an explicit terminator, that is they took the question as specific rather than generic. Similarly two of the fence-sitters took the question as specific. Correcting for these, we obtain a mean of 4.23, which is overwhelming support for explicit delimiters.

176 - "Control structures such as loops will be terminated with an explicit termination symbol which is the control symbol written backwards. For example:

DO.....
.....OD"

A	16	15	19	6	1	57	2.32	1.05
R	7	11	18	16	4	56	2.98	1.13

Strong opposition to the backward spelling for the control delimiter. With one exception, all supporters of this idea were equally supportive of the other choice (or choices, depending upon their interpretation of Q175). In this respect, we can say that these individuals were actually indifferent to the proposal, and took the statement as somewhat generic rather than as purely specific. Accordingly, the opposition to the proposal is stronger than the distribution would indicate, being closer to 2.2 than 2.3. I hope that this puts this proposition to bed.

Q177 - "Control structures such as loops will be terminated with an explicit termination symbol which is the control symbol followed by 'END,' such as, 'DO.....DOEND,' 'CALL.....CALLEND,'.....'IF.....IFEND.'"

A	4	7	19	20	2	52	3.17	.98
R	5	12	15	15	3	50	2.98	1.09

Generally more support for this proposition, but not enough to warrant its adoption.

4.40.4 Analysis

There was strong support for the idea of explicit control structure delimiters, but no agreement on the format that such delimiters, particularly terminators, should use. The following criteria should be applied: the terminator should be formed by a simple uniform rule. Pairs such as CALL/RETURN, BEGIN/END, DO/CONTINUE, should be avoided since they are not uniform and require learning special rules. I don't think that extensive polling will give us any except grudging approval for a specific proposal.

4.41. Increment Point of Loop Control Variables (Q178-Q183)

4.41.1 Purpose of the Questions

To determine what options should be provided with regards to the increment point of a loop control variable.

4.41.2 Comments

1. "Allow for programmer's option here." [As to placement of iteration point.]
2. [Testing of loop termination conditions] "Either before or after [iteration] is O.K., just so you know when it's tested."

4.41.3 Questions and Responses

Q178 - "Iterations of loop control variables can occur at the beginning of the loop."

A	1	9	15	26	2	53	3.36	.87
R	2	9	26	14	1	52	3.06	.82

Q179 - "Iteration of control variables in loops can occur at the end of the loop."

A	1	2	11	30	4	48	3.71	.76
R	2	5	27	10	2	46	3.11	.81

Q180 - "It might be necessary to iterate the control variable in a loop someplace in the middle of the loop."

A	4	18	11	22	2	57	3.00	1.06
R	0	5	26	21	4	56	3.43	.75

In all three of these questions there was confusion on the part of the respondents as to whether we were describing a characteristic of the compiler/language, or a characteristic of the application. Opposers in all three cases tended to have interpreted the statements as a language characteristic, while approvers felt it was referring to the application. These were not considered burning issues in any case. Q180 was more explicitly slanted to application and harder to misinterpret as a language characteristic. In

Q180 we find considerable disagreement. There is a strong group that favors not allowing this degree of freedom, but as a group they were less experienced than those favoring flexibility. It would probably be a good idea to allow both initial and terminal iteration, and to let the exceptional cases be handled by less automatic, explicit statements (e.g., IF...THEN...ELSE).

Q181 - "Loop termination conditions are always tested prior to iteration of the control variable."

A	3	17	13	12	4	49	2.94	1.08
R	2	6	21	17	3	49	3.26	.90

Q182 - "Loop termination conditions are tested after iteration of the control variable."

A	1	7	18	28	1	55	3.38	.80
R	1	7	21	23	3	55	3.36	.84

Q183 - "The relation between the point in the loop at which control variables are iterated and the point at which the control variable is tested for satisfaction of termination conditions is effectively arbitrary and cannot be done in any single uniformed way."

A	9	21	4	14	0	48	2.48	1.10
R	0	3	16	22	5	46	3.63	.76

Again a considerable confusion over whether this was a language characteristic or an application characteristic. For example, Q180 was interpreted as: "The compiler won't tell me where it tests for satisfaction in relation to iteration." That would have been truly abominable.

4.41.4 Analysis

Because of semantic confusion and interpretation of the questions, we cannot give too much credence to the answers. If the issue is really important (it was not felt to be so) then further questions and/or reactions to specific proposals should be sought. I recommend the following approach:

Allow both begin and end of loop iteration, and pre and post iteration testing. All other variations to be done by means of built-up

structures. This should be enough flexibility.

4.42. Storage and Access of Loop Control Variables (Q184,186,187,189-Q192)

4.42.1 Purpose of Questions

To determine how loop control variables should be stored, what kind of access should be provided, and how loop control variable values should be saved.

4.42.2 Comments

1. [The contents of the loop control...] "MUST be available after exits!!!"
2. "In HOL's the same [loop control] variable is usually reused [outside of the subroutine or in other loops] by the programmer."

4.42.3 Questions and Responses

184 - "Control variables in a loop may be subscripted or indexed variables, that is, they may be elements of an array or table."

A	2	4	7	31	12	56	3.84	.96
R	0	5	13	29	9	56	3.75	.83

General strong support for this statement; felt to be moderately important issue. No apparent pattern to the dissenters.

Q186 - "The contents of the control variable of a loop (i.e., the loop count) can only be accessed within the body of the loop."

A	10	29	3	10	3	55	2.40	1.14
R	0	4	8	38	5	55	3.80	.70

Q187 - "The contents of the memory location that stores the value of the loop control variable in a loop is arbitrary when the program exits the loop."

A	5	25	8	9	1	48	2.50	.98
R	0	5	14	25	3	47	3.55	.77

Q190 - "Loop control variables are usually local to the routine in which they appear and rarely have significant value outside of that routine."

A	7	18	1	23	3	52	2.94	1.25
R	2	2	15	26	6	51	3.63	.88

Q192 - "The contents of the memory location that stores the loop count is not accessible once the loop has been exited."

A	23	21	6	7	1	58	2.00	1.07
R	3	4	9	29	12	57	3.75	1.03

Extremely strong rejections of the idea that the loop control variable value is not accessible or is arbitrary upon exiting the loop. Many of the objectors to these statements nevertheless agreed with Q190, their position being that while it is rare, it is essential. Seventeen respondents agreed with one or more of the proposals in Q186, 187, and 192. Q186 and 192 were essentially identical. Eight of them gave inconsistent answers to these questions; judging from the response to Q187, we would put them down as opposed to all three propositions.

This left seventeen respondents who were willing to forego access to the loop control variable (either directly or by default). This group was distinctly less experienced in telecommunications than the average and had less assembly language experience than the average of the sample. They were split evenly as to whether or not the loop control variable had significance outside the loop (Q190).

Q189 - "Most loops are controlled by counters or indexes which must be safe stored for recovery purposes."

A	7	19	8	17	0	51	2.69	1.08
R	2	8	16	21	2	49	3.26	.92

General disagreement with this proposition. Agreement was generally inversely proportional to experience. We should have asked the question with the phrase "...may be controlled...." It would have been more revealing.

Q191 - "Loop control variables (e.g., loop counters) are generally not kept in common tables."

A	1	5	11	29	9	55	3.73	.90
R	5	6	22	17	5	55	3.20	1.05

General agreement with the statement.

4.42.4 Analysis

Whether or not we have found the real reason (if there is a single reason) why loop control variables must be accessible outside of the loop and may not be destroyed, it is clear that there are strong feelings about it among experienced programmers.

4.43. Loop Increment Value Changes (Q185,Q196,Q197)

4.43.1 Purpose of Questions

To determine the degree of flexibility required for the iteration value, its method of control and limits of complexity.

4.43.2 Comments

None.

4.43.3 Questions and Responses

Q185 - "If the control variable of a loop is subscripted, that is, the control variable is stored in an array, the subscript which designates the control variable will be evaluated only once per entrance to the loop. This means that the identity of the control variable may not be changed within the loop it controls."

A	0	7	9	24	8	48	3.69	.92
R	2	4	20	19	1	46	3.28	.83

Strong agreement with the idea of keeping the same control variable in the loop.

Q195 - "The increment value for a loop is constant."

A	2	22	6	16	3	49	2.92	1.08
R	1	3	18	25	2	49	3.49	.76

Q196 - "The increment value for a loop may be variable, but will remain constant within the specific call (for a given entrance to the loop)."

A	1	4	8	31	11	55	3.85	.88
R	1	2	20	23	8	54	3.65	.84

Q197 - "The value of the increment for a loop control variable may be modified within the loop."

A	8	16	4	17	4	49	2.86	1.28
R	2	5	18	19	5	49	3.41	.95

4.43.4 Analysis

These issues were only of moderate importance. An examination of those who felt the need for modifiability of the loop control variable or increment value, showed a fair cross section of experience. The disagreement with Q195 stems from the ambiguity of the statement. Many thought that the increment value was a constant of the loop definition and could never be varied from call to call. This was not intended. On the basis of these responses we would recommend not allowing the control variable to change, and not allowing the increment value to change once the loop had been entered.

4.44. Increment Value by Default (Q193-Q194)

4.44.1 Purpose of Questions

To determine attitude towards default increment value.

4.44.2 Comments

None.

4.44.3 Questions and Responses

Q193 - "If an increment value is not explicitly specified for a loop, the compiler will assume a value of 1 was intended."

A	5	12	3	29	5	54	3.31	1.18
R	0	3	22	25	4	54	3.56	.71

Q194 - "If the increment value for a loop is not explicitly provided, the compiler will assume that there has been a bug."

A	6	21	7	21	2	57	2.86	1.13
R	1	10	18	25	3	57	3.33	.88

4.44.4 Analysis

This is the same old story of defaults/explicit - safety/ease of programming dichotomy, splitting along much the same lines that this issue has split before. Sixteen of the respondents provided inconsistent answers either agreeing with both statements (12 respondents) or disagreeing with both (4 respondents). Defaulters outweighed explicitors by about 2 to 1.

4.45 Increment Value Functions (Q198-Q199)

4.45.1 Purpose of Questions

To determine if it is necessary to have a function for an increment value.

4.45.2 Comments

None.

4.45.3 Questions and Responses

Q198 - "The increment value for a loop may be a function or a simple algebraic expression."

A	2	4	13	31	3	53	3.55	.86
R	1	8	26	13	4	52	3.21	.86

Q199 - "Increment value for a loop may be determined by a procedure."

A	1	5	12	29	5	52	3.62	.86
R	2	10	26	12	2	52	3.04	.85

4.45.4 Analysis

Definite agreement with a very ho-hum issue. "Procedure" is a more acceptable term than "function" or "algebraic expression." Maybe they do indeed prefer the generality implied by "procedure," finding it difficult to cast the procedures they envision into an algebraic or functional mold. The answers were extremely consistent. Of the four inconsistent responses, two were based on the difference between functions and procedures. One of these would accept generalized procedures, but saw no particular use for more restricted algebraic functions; the other favored functions, but rejected procedures because of the implied complexity in the compiler. The basis of the remaining objections is not known.

This is probably one of those things that should be allowed to fall out. It is not really essential and there should be many ways of doing this if needed. If it falls out as a result of uniformity considerations, then it could be implemented, but there is no point of going out of your way to do so.

4.46 Loop Termination (Q188,Q200-Q202)

4.46.1 Purpose of Questions

To determine mode of termination and termination options.

4.46.2 Comments

1. [In exiting a set of nested loops, it is possible to exit several levels simultaneously.] "If this question is??? COBOL (example) - number of internal 'PERFORMS' in a given loop - you can be BADLY burned."

4.46.3 Questions and Responses

Q188 - "A loop may have more than one simultaneous control variable, the first variable to meet the loop termination conditions will cause the termination."

A	3	5	8	37	4	57	3.60	.93
R	0	7	28	17	4	56	3.32	.78

Fairly strong agreement, but the issue is not terribly important. The objectors as a group were less sophisticated than the average and had more standard data processing experience than communications experience.

Q200 - "It is possible to specify a loop by indicating only the termination value. In such cases, the compiler will assume that the loop is started with a value of 1 and is incremented by 1 each time through."

A	0	8	12	31	4	55	3.56	.83
R	1	10	25	17	2	55	3.16	.83

Moderately strong support for default option.

Q201 - "In exiting a set of nested loops, control must pass through each level in order."

A	6	16	2	19	5	48	3.02	1.28
R	0	2	18	24	4	48	3.63	.70

Strong split with no fence sitting.

Q202 - "In exiting a set of nested loops, it is possible to exit several levels simultaneously."

A	4	12	6	25	7	54	3.35	1.17
R	0	1	19	27	5	52	3.69	.67

Stronger support for this contention. Seven inconsistent responses.

4.46.4 Analysis

We expect that the ability to exit several nested loops simultaneously is concerned primarily with exception conditions and could be satisfied by an appropriate exception handler. I think they are worried about detecting conditions which require a crash escape. This might be followed up when specific proposals are brought forth for the language.

4.47 Case Statements and Jump Tables (Q203-Q205)

4.47.1 Purpose of Questions

To determine acceptability of the case statement as a means of implementing jump tables.

4.47.2 Comments

1. [Case statements] "EAT UP CORE!"
2. [A case statement with value ranges, as in Q204 is...] "Not needed if a form [of three way conditional branch] is available."

4.47.3 Questions and Responses

Q203 - "A statement of the form shown below is useful:

```
DO CASE I OF:
   L1:  BLAH BLAH
   L2:  DO ETC.
    :
   LN:  BLAH + BLAH...
```

CASEEND, where L1, L2, etc., are statements or procedures."

A	2	4	9	25	3	43	3.52	.92
R	1	8	15	13	3	40	3.22	.94

Moderate support, but the issue was not felt strongly.

Q204 - "In constructing a jump table controlled by variable i, the language permits ranges of the variable to be specified as follows:

```
DO CASE i
   i = 1-34      L56
     = 35-76  ,  L87 etc."
```

indicating that statement L56 will be executed if the control variable is between 1 and 34."

A	3	9	12	35	2	53	3.60	.83
R	1	6	18	27	0	52	3.37	.76

Since this is more flexible, there is more support.

Q205 - "In constructing a jump table, controlled by variable i, the language allows specifications of the form:

```
DO CASE i          STATEMENT

1                  L56
2,4,7              L17
1,2,4              L90
                   CASEEND
```

The program will execute all statements in the order of appearance corresponding to the value of the control variable. In the above example, a value of 1 will cause statements L56 and L90 to be executed. A value of 2 will cause L17 and L90 to be executed, while a value of 7 will cause L17 to be executed."

A	1	6	11	28	3	49	3.53	.86
R	3	1	21	20	3	48	3.40	.88

Some support eroded, probably because of the complexity of the statement and the length of the questionnaire. This question was toward the end.

4.47.4 Analysis

Case statements can probably be used to substitute for jump tables if the implementation is as efficient as a jump table. Q204 and Q205 proposals, while handy, are conducive to bugs and not essential. The whole issue was not enthusiastically received and simple approaches will probably be acceptable.

4.48 Decision Tables (Q206-Q210)

4.48.1 Purpose of Questions

Should the language provide direct implementation of decision tables and if so, how?

4.48.2 Comments

1. [In executing decision table specifications, the program...] "Should [execute] some statements on some conditions." [???]

4.48.3 Questions and Responses

Q206 - "The language has facilities for the direct specification of decision tables."

A	0	1	11	34	3	49	3.80	.57
R	1	4	11	30	1	47	3.55	.77

Strong support for the general idea of direct implementation of decision tables. However, not viewed as one of the burning issues.

Q207 - "In executing decision table specifications, the program will always execute all statements that satisy the conditions."

A	3	9	10	17	11	50	3.48	1.19
R	0	6	16	20	7	49	3.57	.88

Q208 - "In executing decision table specifications, the program will execute only the first statement that satisfies the condition."

A	6	19	14	7	5	51	2.72	1.14
R	1	6	13	24	6	50	3.56	.92

Q209 - "In executing decision table specifications, the statements can be labeled (in the decision table) as to whether or not they should be executed if some previous statement that satisfies the same conditions has already been executed."

A	0	4	13	26	0	43	3.51	.66
R	0	4	25	12	0	41	3.19	.59

Q210 - "In implementing decision table processing, the programmer can direct the decision table processor to pick up the next statement that satisfies the conditions of the table if there is more than one statement corresponding to a particular set of conditions."

A	0	3	10	28	3	44	3.71	.69
R	0	3	21	19	0	43	3.37	.61

4.48.4 Analysis

As usual, the more flexibility provided, the more the proposition

is liked. The more restrictive it appears to be, the less it is liked. Actually, decision tables should be left out altogether, and in the interest of uniformity, a more complete truth table processor should be provided.

4.49 Jump Table Validation (Q211-Q213)

4.49.1 Purpose of the Questions

To determine if and how, the range of jump table control variables should be checked.

4.49.2 Comments

None.

4.49.3 Questions and Responses

Q211 - "A range check is always inserted by the compiler to check the validity of the control variable of a jump table."

A	1	14	9	22	6	52	3.35	1.05
R	0	4	20	21	4	49	3.51	.76

Again not a burning issue, but surprising agreement about the automatic insertion of jump table variable range check code.

Q212 - "The compiler inserts run time range validation check code for all jump tables."

A	0	10	14	25	5	54	3.46	.90
R	1	6	15	28	3	53	3.49	.84

We repeated the question to make sure. Q212 came first and they liked it better. There were only six inconsistent answers; all of them objecting to run time checks automatically inserted. When the inconsistent answers were removed from Q211, the results were almost identical to Q212. That is, there were no inconsistencies in the opposite direction.

Q213 - "Run time validation of the control variable in a jump table will be done by compiler inserted code if so directed by the programmer."

A	0	6	7	32	3	48	3.67	.77
R	1	7	16	21	2	47	3.34	.86

As expected, there was stronger support for a programmer option of the insertion of run-time check code for jump table range validity.

4.49.4 Analysis

Automatic insertion of run time check code should be the default option; an explicit declaration should be made to avoid the run code insertion. There is, after all, no simple way for the compiler to tell just how, when and where a jump table is being used. It would probably end up inserting too many spurious checks.

4.50. Comments (Q214-Q218)

4.50.1 Purpose of Questions

To provide guidance to the format for comments.

4.50.2 Comments

1. "How about using a 'C' at the end of a line." [to delimit comments]

4.50.3 Questions and Responses

Q214 - "It is essential to be able to insert non-printing characters into comments."

A	7	19	12	13	1	52	2.65	1.05
R	11	17	12	8	3	51	2.51	1.16

Not too much agreement with the proposition. The strongest supporter for this proposition entered his "5" as a joke. A few of the other supporters claimed they had made a mistake. Some had confused this with non-printing characters in a print statement.

The rest of the support can be classified as reflecting a situation in which the comments are used by some other processor such as a flowcharter, and in which the non-printing character serves a control function. Another alternative could be covered by suitably implemented conditional compilation. This is not necessary.

Q215 - "Comments are distinguished by a pair of terminator symbols such as '[' and ']' or '#' and '#' or '¢' and '¢'."

A	5	17	15	13	2	52	2.81	1.04
R	5	7	19	15	3	49	3.08	1.05

Q216 - "Comments are distinguished by a single initial symbol such as '¢' and terminated by the end of the line."

A	0	6	20	20	11	57	3.63	.91
R	2	8	20	23	2	55	3.27	.88

Q217 - "Comments must appear in certain columns of a formatted source."

A	12	19	6	15	2	54	2.56	1.21
R	4	8	14	21	4	51	3.25	1.06

Q218 - "It is desirable to be able to insert executable statements within a comment, for example, '¢ DO .....DOEND¢.'"

A	13	14	6	17	2	52	2.63	1.27
R	3	7	18	17	5	50	3.28	1.02

4.50.4 Analysis

Non-printing characters in comments are neither desirable nor essential. The specific mechanism for delimiting comments is the use of an explicit initial delimiter and an implicit end delimiter (End of Line). A lot of this reflects individual proclivities. The specifics of treatment of comments was considered a ho-hum issue all the way.

5. WHAT IS TOUGH AND IMPORTANT ABOUT COMMUNICATIONS PROGRAMS

5.1. General

The purpose of this section is to explore what, in the minds of the programmers, are the difficult aspects and crucial issues in the design of telecommunications software. Each question will be treated separately. There is not much to analyze here, other than presenting the raw results and comments. This section should serve to display the myths of telecommunications software design.

5.2. Questions, Responses, Comments and Analysis

Q219 - "The most difficult part of communications programming is understanding the system specifications."

A	1	19	10	17	4	51	3.08	1.05
R	0	3	19	20	6	48	3.60	.78

Q220 - "The most difficult part of communications software is the communications channel interface software."

A	4	23	16	10	1	54	2.65	.93
R	4	8	14	19	8	53	3.36	1.13

"That depends on the hardware."

Q221 - "Timing considerations are crucial to communications software."

A	0	1	0	27	24	52	4.42	.60
R	0	0	2	28	22	52	4.38	.56

Q222 - "Hardware characteristics are the most troublesome aspect of communications programming."

A	2	20	5	20	2	49	3.00	1.07
R	1	3	15	23	6	48	3.63	.86

[Hardware characteristics are...] "A." [...troublesome aspect of communications programming.]

"Defining the requirements (concisely) is the most troublesome aspect of communications programming."

Those who disagreed tended to have more assembly language experience than those who agreed with this proposition. The way a person responds to this question depends very much on what that person is doing at the moment; the last machine he worked on; how long that machine has been in production; how long he has been on that machine, etc.

Q223 - "The most difficult part of communications software is related to recovery and restart."

A	3	10	7	17	16	53	3.62	1.25
R	2	5	9	20	17	53	3.85	1.09

"As with all [everything], debugging is more important and difficult [than anything else] on the project."

A strong response for a fairly important issue. Half of the dissenters have been intimately involved in the design of recovery and restart related software. They tended to be senior assembly language programmers with considerable experience (a mean of seven years in the telecommunications programming) - they did not see it as a problem because of their familiarity with it. The remaining group could not be characterized.

Q224 - "Message accountability and the avoidance of misrouting are crucial issues in the design of communications software."

A	0	0	1	13	44	58	4.74	.48
R	0	1	1	10	46	58	4.74	.57

The only interesting thing about this question is that it gives us a calibration of the respondents as a group. 4.74 is probably the highest agreement we could get to anything, including a statement such as "The sky is blue," or "Software is important." The low standard deviation is also a calibration. Note also the identity of the agreement and relevance mean value.

Q225 - "Process control, executive structure, and intermodule communications dominate the design issues in communications software."

A	1	5	7	28	9	50	3.78	.92
R	1	1	11	27	9	49	3.86	.81

"No. Only the customer's spec." [Specification dominates the design issues in communications software.]

No particular pattern to the dissenters.

Q226 - "More effort is spent on the design of the data base than any other single thing."

A	2	14	12	20	4	52	3.19	1.04
R	1	5	15	22	9	52	3.63	.94

[More effort is spent on the design of the data base than *any other single thing.*] "Except documentation!"

Substantial disagreement with this all inclusive statement was, of course, expected. The reason for stating it this way was to prove a point. Those who agreed with the proposition felt that it was indeed the most labor intensive area. Those who disagreed, did not disagree regarding the importance, but with the characterization as "most important." There should be no further doubt regarding the importance of strong HOL support to data base definition and maintenance.

Q227 - "Interrupt processing is a major design issue."

A	0	4	3	24	20	51	4.18	.86
R	0	1	1	27	20	49	4.35	.62

[Interrupt processing is a major design issue...] "Along with...." [Input processing, output processing.]

This statement is, of course, culturally loaded, and it takes a lot of guts to disagree. Of the four dissenters, one can be identified as an extremely experienced assembly programmer who appears to have transcended such problems; the others cannot be calibrated.

Q242 - "Large projects employ a data base specialist (e.g., one individual who is responsible for the design and/or maintenance of the data base).

A	5	10	6	27	8	56	3.41	1.19
R	4	4	8	30	10	56	3.68	1.07

"Maybe they should, but...."

"Varies."

"How I wish this were true!"

Q243 - "The data base design is generally done by a few senior individuals."

A	0	4	4	36	8	52	3.92	.73
R	1	5	10	25	9	50	3.72	.94

There is probably no substantial disagreement over the desirability of employing either full or part time data base specialists and the need to concentrate data base design responsibilities into a small group. The dissent here is with the statement that this is the way it is being done. We should have asked another question of the form "...should be done by a few senior individuals...." Note the stronger agreement with the design team for the data base. The implications in the HOL design is that one could almost have a separate rich complicated declaration language. This would allow a lot of flexibility and control over the data base design, and would abet the concentration of the design and maintenance to a few individuals or specialists. There should be no apprehension over undue complexity here; this is one area where we can tolerate the (programming) complexities.

Q246 - "Running time efficiency is the most important aspect of communications software."

A	1	18	9	19	7	54	3.24	1.10
R	1	1	15	23	13	53	3.87	.87

[Running time efficiency is the most important aspect of communications software.] "Definitely so for a HOL!"

Q247 - "Program space is generally more important than running time."

A	7	32	5	7	0	51	2.24	.85
R	1	0	11	31	8	51	3.88	.73

"Depends on application."

"On what level? System? Module? Subroutine?"

Q248 - "Tight packing is employed frequently to conserve data base space even though this may increase processing time."

A	1	8	17	23	7	56	3.48	.95
R	1	3	20	24	8	56	3.63	.86

There were five more respondents to Q248 than to Q247. Of these two answered "3" and three answered "4" to Q248. Nineteen of those who disagreed with Q247 contradicted that disagreement by agreeing with Q248 and the need for packing. Others went from disagreement with Q247 to uncertainty regarding Q248. Sixteen respondents were consistent, eight insisting that time was less important and that packing should be done, and eight holding to the opposite opinion. The large number of inconsistent answers here is due to several possible causes:

1. The question was in some sense the last one.
2. The word "time" usually precedes "space." There might have been a mental reversal.
3. The respondents are confused with regard to the issue.
4. The two statements are not really contradictory, since space reduction can cause time reductions and vice versa.

At any rate, it is clear that automatic sacrifices in favor of time at the expense of space should not be made; optimization is more complex and should not be based on any single, simplistic space or time criterion.

6. WHERE DO THE BUGS COME FROM

6.1. Purpose of the Questions

The purpose of these questions was to determine the subjective evaluation of the source of bugs.

6.2. Questions, Responses, Comments and Analysis

Q249 - "The single most common source of bugs are due to misunderstanding the specifications."

A	2	20	8	14	7	51	3.08	1.17
R	0	5	10	28	7	50	3.74	.82

"The single most common source of bugs is due to carelessness!"
"What specifications?"

Strong division of opinion here, probably based on individual experience.

Q250 - "Keypunch and typographical errors are the source of most bugs."

A	16	26	6	6	1	55	2.09	1.00
R	5	8	15	18	6	52	3.23	1.14

[Keypunch and typographical errors are the source of most...]
"INITIAL." [bugs]

The comment is important; many programmers do not regard keypunch and typographical errors as bona fide bugs, but rather tend to treat them as mere gnats.

Q251 - "Most bugs occur in calling sequences."

A	2	37	6	5	0	50	2.28	.69
R	1	13	16	15	1	46	3.04	.88

"So?"

Strong disagreement with this one.

Q252 - "Most bugs result from a misinterpretation of the data base."

A	5	29	9	13	0	56	2.54	.94
R	5	8	18	23	3	57	3.19	1.03

"Other than initial bugs?"

Again a strong disagreement. It does not seem to be possible, on the basis of these questions, to pin down just where the bugs come from - vague terms such as "carelessness," or "misunderstanding" would have been confusing and somewhat provoking.

7. WHAT THEY EXPECT FROM A HOL

7.1. Purpose of the Questions

The purpose of these questions was to get an idea of the expectations of the respondents regarding a HOL.

7.2. Questions, Responses, Comments and Analysis

Q228 - "A HOL will increase a programmer's productivity."

A	3	8	7	23	16	57	3.72	1.17
R	2	5	6	22	21	56	3.98	1.08

"Yes. One bad statement executes more code to disguise the bug better."

"Depends wholly on his background."

"In writing, but not necessary in debugging [because of the...] need to go to assembler code to debug. [where] 1 HOL statement = N assembler instructions."

"Only because of its self-documenting attribute."

Strong support for an important issue, but equally strong dissent from an experienced group.

Q229 - "Programming with a HOL will probably be more difficult than assembly language."

A	14	31	5	2	0	52	1.90	.71
R	2	6	11	23	10	52	3.63	1.04

"I hope not."

"Depends on how good the HOL is."

It is sad to see such ardent hopes dashed. We may be expecting too much in this respect, and confusing the applications that are programmed in HOL's with the HOL's. After all, the question is really like saying that Russian is harder than Turkic - it depends on where you come from and what you have to talk about.

Q230 - "HOL program will be easier to debug."

A	4	8	10	23	10	55	3.49	1.16
R	2	2	11	21	19	55	3.96	1.01

"Caution." [...that depends on the target CPU peculiarities.]

"If not, forget it!"

Q231 - "There will be fewer, but more subtle bugs in a HOL program."

A	3	17	7	20	5	52	3.13	1.14
R	0	3	15	24	8	50	3.74	.80

"Very true, that's why the programmer <u>is required</u> to know assembly language."

"A bug is a bug is a bug!"

Q232 - "HOL programs will be easier to maintain."

A	2	5	6	29	16	58	3.90	1.01
R	2	5	6	26	18	57	3.93	1.04

"FALSE!"

Strong support and belief in the issue.

Q233 - "It will be more difficult to learn the HOL than to learn how to program a new machine in assembly language."

A	14	29	2	6	1	52	2.06	.97
R	5	3	14	18	11	51	3.53	1.18

I didn't really expect to say this to a bunch of assembly programmers and not get a strong reaction.

Q234 - "The skill level for HOL programming of communications systems will generally be lower than that needed for assembly language."

A	5	21	6	15	6	53	2.93	1.23
R	3	2	15	17	14	51	3.72	1.09

"HOL programmers better know the assembly language to debug." [Paraphrased!]

"Even more [skill] I think."

Major diversion of opinion over what must be an emotional issue.

Q235 - "HOL communications software will provide machine independence for application software."

A	6	12	7	17	11	53	3.28	1.32
R	2	0	3	25	22	52	4.25	.87

"That will be the day."

Q236 - "HOL programs will require more space than assembly programs."

A	0	5	8	29	15	57	3.95	.87
R	2	7	15	19	14	57	3.63	1.09

"IBM COBOL INDEX LOOP - 37 instructions,
IBM ASSEMBLER INDEX LOOP - 10 instructions"

Q237 - "HOL programs will (on an average) execute faster than routines written by the typical assembly language programmer."

A	8	30	8	3	3	52	2.29	.99
R	2	3	17	18	11	51	3.65	1.01

The response must have had a lot to do with whether the respondent identified himself as "typical" or not. The statement was, unfortunately, too provocative.

Q238 - "HOL programs can be brought on-line in less time than assembly language programs."

A	3	8	9	22	9	51	3.51	1.13
R	1	2	14	20	14	51	3.86	.93

"Providing the design is right!"

"Depends on code generated."

Q239 - "HOL source programs will have fewer initial bugs than assembly language programs."

A	5	2	8	25	18	58	3.84	1.16
R	2	6	8	25	16	57	3.82	1.06

"Just different types." [of bugs]

"Fewer syntax errors."

"All programmers have bugs - talk about debugging a HOL!"

Q240 - HOL debugging will be more difficult since you will have to get down to assembly language anyhow, eventually."

A	17	27	5	6	3	58	2.16	1.11
R	2	1	17	27	11	58	3.76	.90

"Not if the assembly listing provided with the HOL is usefully laid out!"

"Only if [other] HOL or programmer"[is unreliable]

Q241 - "It will be more difficult to integrate a HOL package than an assembly language package."

A	14	30	1	5	2	52	2.06	1.01
R	0	4	16	18	13	51	3.78	.91

"Do you mean: Debug? Answer = Yes
Write? Answer = No
Combined? Answer = Yes?"

"Why?" [will it be more difficult to integrate]

Q244 - "A language that does not support communications I/O operations might as well not exist."

A	7	11	6	16	17	57	3.44	1.40
R	4	3	9	17	21	54	3.89	1.20

"For our purposes."

Moderate agreement with this, but sharp split.

Q245 - "The variety of communications related I/O operations is so great that it would not be possible to construct a single approach that was not equivalent to assembly language programming."

A	2	27	6	9	4	48	2.71	1.08
R	1	1	12	24	9	47	3.83	.83

This is a good indication of expectations with regard to I/O handling; they think it can be done and they expect it to be done. This is probably going to be a strong source of user criticism.

Q253 - "The documentation of programs in a HOL will be easier to understand."

A	1	11	10	22	7	51	3.45	1.03
R	1	2	14	24	10	51	3.78	.87

"What documentation? The code edit?"

"The documentation of programs in a HOL will absolutely exist."

Q254 - "A HOL source program will obscure much of what is happening the documentation will have to be increased to compensate for this."

A	13	24	2	13	1	53	2.34	1.15
R	0	4	14	24	12	54	3.81	.86

Q255 - "A HOL will promote better programming style and will eliminate many dubious practices."

A	10	7	5	24	12	58	3.36	1.39
R	4	5	5	27	15	56	3.79	1.14

Q256 - "Reprehensible programming practices will neither be abetted nor averted through the use of a HOL."

A	4	18	9	18	7	56	3.11	1.19
R	3	10	14	18	9	54	3.37	1.13

"Dubious practices are just different in a HOL."

"When structured programming methods are employed."

7.3. Comments

They have high hopes with regard to the reduction of labor contents for HOL's, but express the expected concern over time and space. They expect all the things that HOL propaganda has led them to expect. A good language, which abets security, documentation, maintainability, and other functional and life cycle considerations at the expense of efficiency, ease of programming skill level, etc., would be rejected and probably for the wrong reasons.

8. SPECIAL FACILITIES

8.1. Purpose of Questions

These questions, except for Q86, were included in an unscrambled order to determine the desirability of status types or an equivalent extension mechanism and other specific language facilities.

8.2. Questions, Responses, Comments and Analysis

Q86 - "The programmer can specify data types of his own in addition to those, such as integers, reals, pointers, which are build into the language. For example, you could specify SECURITY as a data type."

A	0	3	15	31	2	51	3.63	.66
R	2	6	19	23	2	52	3.33	.87

This found a lot of favor, but the issue was not considered terribly important.

Q257 - (A1) "It is planned to introduce a data type called 'Status data type.' A status type is used to define a set of properties of an element, which thereafter can be used for control purposes. For example, the declaration of a status type might have the following

```
TYPE IS SECURITY = SECRET, UNCLASSIFIED, NATOCONF, CON-
                   FIDENTIAL
```

In use, it would appear in the form of a statement such as:

```
IF CLASSIFICATION IS SECRET THEN.....ELSE.....
```

Having made such a declaration, the compiler would spot the bug such as:

```
IF CLASSIFICATION IS URGENT THEN.....ELSE....."
```

A	1	1	6	36	8	52	3.94	.72
R	1	6	13	26	6	52	3.58	.91

Note the increased support with a fuller explanation.

Q258 - (A2) "Status types are encoded as consecutive integers in the order of their appearance in a declaration."

A	3	10	23	12	2	50	3.00	.92
R	1	6	25	14	2	48	3.21	.79

No support for something enforced.

Q259 - "The programmer can specify the encoding of status types
(A3) (e.g., SECRET = 1, UNCLASSIFIED = 17, NATOCONF = 3, etc.)."

A	0	3	12	31	6	52	3.77	.72
R	0	7	19	21	4	51	3.43	.82

Major increase in support when option is given.

Q260 - "If no encoding is specified, the compiler will use con-
(A4) secutive integers as a default."

A	2	4	11	28	3	48	3.54	.89
R	0	5	28	12	2	47	3.23	.69

Fair support for default.

Q261 - "No status type requires more than 256 cases (e.g., MONDAY,
(A5) TUESDAY, WEDNESDAY, ...)."

A	2	8	14	24	2	50	3.32	.93
R	0	4	27	15	2	48	3.31	.68

Balking at a limitation - grudging approval.

Q262 - "The number of bits used to represent a status type should
(A6) be left to the compiler to optimize."

A	1	15	15	16	4	51	3.14	.99
R	0	11	18	16	3	48	3.23	.87

Taking away an option - grudging approval.

Q263 - "The programmer must specify how many bits are to be used
(A7) for representing each status type."

A	3	20	16	11	1	51	2.75	.93
R	0	5	27	15	1	48	3.25	.66

But don't make me do all that work!

Q264 - "The ability to handle more than 256 cases for a given status
(A8) type is important."

A	3	25	12	9	2	51	2.65	.97
R	0	7	26	13	2	48	3.21	.73

"Depends on application."

They will accept the limitation.

Q265 - "It is desirable to allow a given status value name to take
(A9) on different meanings in different contexts, for example:

```
TYPE IS JUNK = TOM, DICK, HARRY, BILL,
.......
TYPE IS HELP = TIM, TOM, JACK, JILL.
```

If the two declarations are in different scopes, then the second TOM is permitted and will not be confused with instances of the first."

A	3	10	7	23	5	48	3.35	1.11
R	1	3	18	21	3	46	3.48	.80

"Why not IF JUNK is TOM as in QA1?"

"Why not as in A1?"

Q266 - "As in Question 265, but the distinction will be made in
(A10) the call, for example:

```
IF TYPE = JUNK.TOM          THEN .......ELSE.......
and
IF TYPE = HELP.TOM          THEN .......ELSE......."
```

A	2	5	11	22	5	45	3.51	.98
Q	1	5	15	18	4	43	3.44	.90

"A la Pascal."

Q267 - "Ordering related operations for status types are provided.
(A11) Example: IF TYPE IS GREATER THAN SECRET THEN ....ELSE....."

A	0	0	9	32	7	48	3.96	.58
R	0	4	17	21	4	46	3.54	.77

Some enthusiasms here.

Q268 - "In comparing two types, it is always assumed that ordering
(A12) is possible. The ordering is that which results from the integer encoding of the values."

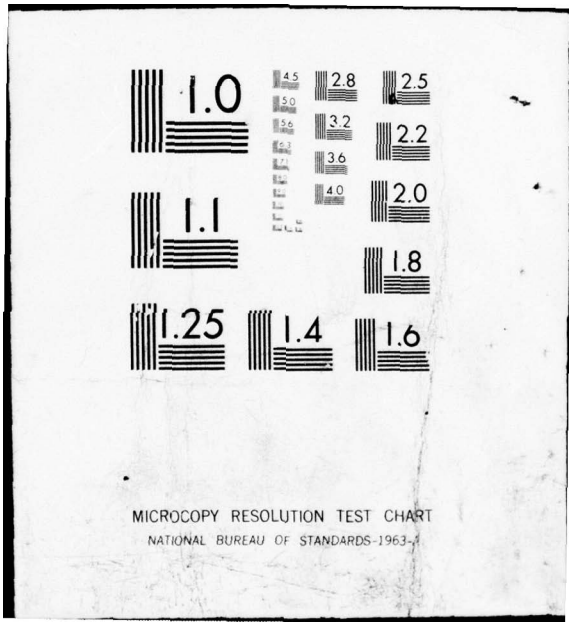
A	0	4	8	33	4	49	3.75	.72
R	0	7	19	18	3	47	3.36	.81

Reluctance when presented with specifics.

AD-A049 737 SOFTECH INC WALTHAM MASS F/G 9/2
COMMUNICATIONS HIGH-ORDER LANGUAGE INVESTIGATION. VOLUME I.(U)
OCT 77 R S EANES, J B GOODENOUGH, J R KELLY F30602-76-C-0306
UNCLASSIFIED RADC-TR-77-341-VOL-1 NL

3 OF 3
AD
A049737

END
DATE
FILMED
3-78
DDC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

Q269 - "Types are assumed to be ordered by their encoding unless
(A13) explicitly declared to be unordered."

A	1	8	14	25	1	49	3.35	.85
R	1	8	17	18	3	47	3.30	.90

Q270 - "Types are assumed to be unordered unless explicitly de-
(A14) clared as ordered."

A	3	21	9	15	1	49	2.80	1.01
R	0	4	22	19	2	47	3.40	.70

Q271 - "The compiler will prohibit the use of ordering operators
(A15) (e.g., greater than) for types not declared as ordered."

A	0	11	11	21	5	48	3.42	.95
R	0	5	20	18	3	46	3.41	.77

Q272 - "Types can be declared as being cyclically ordered (e.g.,
(A16) MON, TUE, WED, THU, FRI, SAT, SUN, MON, ...)."

A	2	3	15	26	2	48	3.48	.84
R	1	8	22	13	2	46	3.15	.83

Q273 - "Successor and predecessor operators for cyclic ordered
(A17) types are available:

e.g., SUCC(MON) = TUE ; PRED(MON) = SUN

A	1	4	17	22	2	46	3.44	.80
R	1	10	23	9	1	44	2.98	.78

Q274 - "If the programmer supplies the encoding (See Question 259,
(A18) 260) different elements can have synonymous codes. For example:

DUPLEX = 2, HDX = 3, SATELLITE = 2, SIMPLEX = 4,
TRUNK = 1, etc."

A	1	2	9	32	4	48	3.75	.75
R	0	6	25	12	3	46	3.26	.76

Q275 - "A number of bit field operations are being considered. When
(A19) they appear in the context of an assignment statement, they result in a truth function value (TRUE/FALSE), when in the context of an IF statement, they provide a branching operation. Of particular interest are 'greater than or equal' comparisons for bit strings or bit fields. For example A is greater than B if every bit of A is greater than or equal to very bit of B. If however, some bits of A are greater than corresponding bits of B, while some bits of B are greater than corresponding bits of A, then A and B are incompatible. Is a statement of the following form useful?

IF A GRT B THEN .....ELSE....."

A	2	8	16	19	2	47	3.23	.93
R	2	10	18	13	1	44	3.02	.89

Q276 - "How about: IF A LTE B THEN .....ELSE....."
(A20)

A	1	7	17	19	3	47	3.34	.88
R	2	9	19	12	2	44	3.07	.91

Q277 - "How about: IF A GRT B THEN .....IF NOT.....IF
(A21) INCOMPARABLE THEN....."

A	4	6	17	19	0	46	3.11	.94
R	2	9	17	15	0	43	3.05	.86

"IF A GRT B DO X, IF A LTE B DO Y, IF A INCOMPATIBLE DO Z."

8.3. Comments

There was a general approval of the proposed facilities with dissension relating to specifics along the typical and familiar lines of safety/simplicity, explicit/implicit, flexibility/declarations, etc. Informal comments related to these proposals generally indicated a willingness to try something new, but that we should not expect enthusiastic response regarding any proposed feature until it has been tried under fire.

ANNEX I

RAW DATA RESPONSES

DISTRIBUTION FOR AGREEMENT

QUEST	1	2	3	4	5	NUMB	MEAN	DEVI
1	3	16	7	19	4	49	3.10	1.13
2	4	13	11	21	4	53	3.15	1.11
3	4	21	6	14	2	47	2.77	1.10
4	4	20	6	20	4	54	3.00	1.16
5	3	16	9	15	2	45	2.93	1.06
6	9	31	6	6	1	53	2.23	0.92
7	10	20	10	11	1	52	2.48	1.08
8	3	11	9	25	8	56	3.43	1.12
9	0	5	8	26	8	47	3.79	0.85
10	10	15	6	16	6	53	2.87	1.33
11	4	17	7	16	1	45	2.84	1.07
12	5	21	5	18	3	52	2.87	1.16
13	1	6	8	23	2	40	3.47	0.89
14	0	0	9	31	5	45	3.91	0.55
15	0	5	10	31	5	51	3.71	0.77
16	1	4	11	28	4	48	3.63	0.83
17	2	6	10	25	6	49	3.55	0.99
18	0	6	12	27	4	49	3.59	0.81
19	0	2	1	22	27	52	4.42	0.72
20	1	5	6	36	7	55	3.78	0.85
21	2	15	2	26	1	46	3.20	1.06
22	1	3	11	23	6	44	3.68	0.87
23	7	26	3	10	2	48	2.46	1.10
24	1	9	13	15	4	42	3.29	0.98
25	1	2	9	33	7	52	3.83	0.78
26	0	2	8	35	8	53	3.93	0.67
27	1	13	4	26	7	51	3.49	1.07
28	2	9	4	26	15	56	3.77	1.12
29	6	22	9	9	6	52	2.75	1.21
30	0	4	8	32	4	48	3.75	0.72
31	0	2	7	33	14	56	4.05	0.72
32	3	2	6	35	6	52	3.75	0.92
33	1	5	12	30	2	50	3.54	0.80
34	1	5	10	36	2	54	3.61	0.78
35	2	10	14	19	0	45	3.11	0.90
36	1	6	18	23	2	50	3.38	0.82
37	2	12	12	15	5	46	3.20	1.08
38	2	14	6	18	5	45	3.22	1.13
39	3	10	4	30	2	49	3.37	1.04
40	0	0	13	28	14	55	4.02	0.70
41	0	7	6	33	2	48	3.63	0.78
42	0	5	2	38	11	56	3.98	0.77
43	6	17	13	11	2	49	2.71	1.07
44	5	7	5	30	9	56	3.55	1.16
45	1	4	6	31	9	51	3.84	0.87
46	0	5	7	35	4	51	3.75	0.74
47	2	1	16	25	13	57	3.81	0.93
48	1	3	16	26	6	57	3.49	0.92
49	1	2	18	27	2	50	3.54	0.73
50	1	5	11	37	3	57	3.63	0.79
51	1	5	18	17	1	42	3.29	0.80
52	4	19	11	18	4	56	2.98	1.11
53	1	6	9	28	3	47	3.55	0.87
54	1	16	17	20	1	55	3.07	0.89
55	6	24	6	16	0	52	2.62	1.04

56	7	14	9	19	6	55	3.06	1.24
57	0	2	13	32	3	50	3.72	0.63
58	6	19	13	14	0	52	2.67	0.99
59	2	9	12	24	2	49	3.31	0.95
60	3	14	17	19	1	54	3.02	0.95
61	6	16	10	18	0	50	2.80	1.06
62	6	17	12	15	1	51	2.76	1.06
63	3	3	8	30	10	54	3.76	1.00
64	3	5	8	35	4	55	3.58	0.95
65	1	12	5	29	4	51	3.45	1.00
66	1	3	27	16	3	50	3.34	0.76
67	0	8	29	7	0	44	2.98	0.58
68	0	3	20	26	4	53	3.59	0.71
69	7	15	13	12	2	49	2.74	1.10
70	3	14	12	19	3	51	3.10	1.05
71	1	9	14	22	1	47	3.28	0.87
72	7	9	15	14	6	51	3.06	1.21
73	3	11	16	10	2	42	2.93	0.98
74	4	12	15	14	5	50	3.08	1.11
75	9	22	9	3	2	45	2.27	1.00
76	2	8	28	10	2	50	3.04	0.82
77	8	8	17	6	6	45	2.87	1.24
78	0	5	17	25	2	49	3.49	0.73
79	1	3	19	24	10	57	3.68	0.88
80	5	13	10	23	4	55	3.15	1.14
81	2	4	12	28	2	48	3.50	0.87
82	2	3	17	27	6	55	3.58	0.89
83	2	13	14	18	2	49	3.10	0.97
84	0	4	14	32	5	55	3.69	0.73
85	0	2	13	32	5	52	3.77	0.67
86	0	3	15	31	2	51	3.63	0.66
87	2	19	6	18	3	48	3.02	1.09
88	1	1	4	35	16	57	4.12	0.75
89	0	3	7	31	9	50	3.92	0.74
90	21	20	8	4	0	53	1.91	0.92
91	0	4	7	25	14	50	3.98	0.86
92	8	14	13	15	4	54	2.87	1.19
93	7	26	12	7	0	52	2.37	0.88
94	3	16	10	21	3	53	3.09	1.07
95	13	16	10	15	3	57	2.63	1.24
96	18	23	5	8	0	54	2.06	1.01
97	2	9	13	21	3	48	3.29	0.98
98	10	32	3	11	0	56	2.27	0.97
99	1	12	12	21	6	52	3.37	1.02
100	1	4	11	29	11	56	3.80	0.90
101	1	13	14	21	2	51	3.20	0.92
102	1	15	12	22	2	52	3.17	0.95
103	2	11	16	17	3	49	3.16	0.98
104	0	0	2	23	31	56	4.52	0.57
105	6	26	5	10	1	48	2.46	1.02
106	5	7	9	29	1	51	3.28	1.05
107	0	1	7	39	6	53	3.94	0.56
108	0	1	4	30	21	56	4.27	0.67
109	8	15	5	21	3	52	2.92	1.24
110	1	4	12	20	7	44	3.64	0.93
111	1	6	10	19	8	44	3.61	1.00
112	0	7	11	29	5	52	3.62	0.84
113	1	6	10	30	5	52	3.62	0.88
114	0	6	6	32	3	47	3.68	0.77
115	0	6	11	28	6	51	3.67	0.83
116	8	13	13	18	4	56	2.95	1.19
117	1	8	13	25	3	50	3.42	0.90
118	0	2	1	38	13	54	4.15	0.62
119	1	9	12	26	2	50	3.38	0.89
120	1	4	4	33	15	57	4.00	0.88
121	1	4	9	34	5	53	3.72	0.81

122	10	37	3	3	0	53	1.98	0.69
123	3	12	13	21	2	51	3.14	1.01
124	4	11	9	27	4	55	3.29	1.09
125	7	24	7	10	1	49	2.47	1.03
126	1	10	20	17	0	48	3.10	0.80
127	2	4	12	24	12	54	3.74	1.00
128	4	11	7	19	7	48	3.29	1.21
129	2	17	11	18	2	50	3.02	1.01
130	2	10	4	31	8	55	3.60	1.06
131	4	9	20	12	0	45	2.89	0.90
132	0	7	15	30	4	56	3.55	0.80
133	0	9	13	26	1	49	3.39	0.80
134	1	13	15	16	4	49	3.18	0.98
135	5	27	8	9	1	50	2.48	0.96
136	0	6	5	32	11	54	3.89	0.85
137	3	17	10	19	1	50	2.96	1.02
138	1	6	11	32	5	55	3.62	0.86
139	1	1	6	36	5	49	3.88	0.69
140	2	3	8	35	7	55	3.76	0.87
141	2	8	7	25	4	46	3.46	1.02
142	1	14	12	22	2	51	3.20	0.95
143	4	3	14	28	8	57	3.58	1.03
144	3	14	16	14	6	53	3.11	1.09
145	1	15	7	20	3	46	3.20	1.03
146	2	21	10	15	7	55	3.07	1.14
147	8	15	5	19	4	51	2.92	1.27
148	11	11	5	21	9	57	3.10	1.40
149	8	15	5	22	2	52	2.90	1.21
150	9	23	9	8	0	49	2.33	0.96
151	0	9	16	21	3	49	3.37	0.85
152	0	0	5	41	12	58	4.12	0.53
153	0	0	5	39	7	51	4.04	0.48
154	1	8	13	18	2	42	3.29	0.91
155	0	3	10	28	5	46	3.76	0.73
156	0	2	13	31	9	55	3.85	0.72
157	0	5	9	31	3	48	3.67	0.74
158	0	8	12	23	4	47	3.49	0.87
159	6	9	8	18	15	56	3.48	1.32
160	6	5	15	19	8	53	3.34	1.18
161	0	2	12	35	2	51	3.72	0.60
162	15	25	6	9	0	55	2.16	1.00
163	1	2	17	23	3	46	3.54	0.77
164	2	8	7	33	7	57	3.61	0.99
165	3	5	6	34	2	50	3.54	0.94
166	3	5	6	34	3	51	3.57	0.95
167	1	6	6	35	5	53	3.70	0.86
168	4	7	6	29	7	53	3.53	1.11
169	1	0	10	32	5	48	3.83	0.69
170	3	18	5	24	4	54	3.15	1.13
171	1	14	10	20	2	47	3.17	0.98
172	0	2	16	24	6	48	3.71	0.73
173	1	4	9	33	2	49	3.63	0.77
174	3	14	5	26	5	53	3.30	1.13
175	1	2	6	28	19	56	4.11	0.86
176	16	15	19	6	1	57	2.32	1.05
177	4	7	19	20	2	52	3.17	0.98
178	1	9	15	26	2	53	3.36	0.87
179	1	2	11	30	4	48	3.71	0.76
180	4	18	11	22	2	57	3.00	1.06
181	3	17	13	12	4	49	2.94	1.09
182	1	7	18	28	1	55	3.38	0.80
183	9	21	4	14	0	48	2.48	1.10
184	2	4	7	31	12	56	3.94	0.96
185	0	7	9	24	8	48	3.69	0.92
186	10	29	3	10	3	55	2.40	1.14
187	5	25	8	9	1	48	2.50	0.98

188	3	5	8	37	4	57	3.60	0.93
189	7	19	8	17	0	51	2.69	1.08
190	7	18	1	23	3	52	2.94	1.25
191	1	5	11	29	9	55	3.73	0.90
192	23	21	6	7	1	58	2.00	1.07
193	5	12	3	29	5	54	3.31	1.18
194	6	21	7	21	2	57	2.86	1.13
195	2	22	6	16	3	49	2.92	1.08
196	1	4	8	31	11	55	3.85	0.88
197	8	16	4	17	4	49	2.86	1.28
198	2	4	13	31	3	53	3.55	0.86
199	1	5	12	29	5	52	3.62	0.86
200	0	8	12	31	4	55	3.56	0.83
201	6	16	2	19	5	48	3.02	1.28
202	4	12	6	25	7	54	3.35	1.17
203	2	4	9	25	3	43	3.53	0.92
204	3	1	12	35	2	53	3.60	0.83
205	1	6	11	28	3	49	3.53	0.86
206	0	1	11	34	3	49	3.80	0.57
207	3	9	10	17	11	50	3.48	1.19
208	6	19	14	7	5	51	2.72	1.14
209	0	4	13	26	0	43	3.51	0.66
210	0	3	10	28	3	44	3.71	0.69
211	1	14	9	22	6	52	3.35	1.05
212	0	10	14	25	5	54	3.46	0.90
213	0	6	7	32	3	48	3.67	0.77
214	7	19	12	13	1	52	2.65	1.05
215	5	17	15	13	2	52	2.81	1.04
216	0	6	20	20	11	57	3.63	0.91
217	12	19	6	15	2	54	2.56	1.21
218	13	14	6	17	2	52	2.63	1.27
219	1	19	10	17	4	51	3.08	1.05
220	4	23	16	10	1	54	2.65	0.93
221	0	1	0	27	24	52	4.42	0.60
222	2	20	5	20	2	49	3.00	1.07
223	3	10	7	17	16	53	3.62	1.25
224	0	0	1	13	44	58	4.74	0.48
225	1	5	7	28	9	50	3.78	0.92
226	2	14	12	20	4	52	3.19	1.04
227	0	4	3	24	20	51	4.18	0.86
228	3	8	7	23	16	57	3.72	1.17
229	14	31	5	2	0	52	1.90	0.71
230	4	8	10	23	10	55	3.49	1.16
231	3	17	7	20	5	52	3.13	1.14
232	2	5	6	29	16	58	3.90	1.01
233	14	29	2	6	1	52	2.06	0.97
234	5	21	6	15	6	53	2.93	1.23
235	6	12	7	17	11	53	3.28	1.32
236	0	5	8	29	15	57	3.95	0.87
237	8	30	8	3	3	52	2.29	0.99
238	3	8	9	22	9	51	3.51	1.13
239	5	2	8	25	18	58	3.84	1.16
240	17	27	5	6	3	58	2.16	1.11
241	14	30	1	5	2	52	2.06	1.01
242	5	10	6	27	8	56	3.41	1.19
243	0	4	4	36	8	52	3.92	0.73
244	7	11	6	16	17	57	3.44	1.40
245	2	27	6	9	4	48	2.71	1.08
246	1	18	9	19	7	54	3.24	1.10
247	7	32	5	7	0	51	2.24	0.85
248	1	8	17	23	7	56	3.48	0.95
249	2	20	8	14	7	51	3.08	1.17
250	16	26	6	6	1	55	2.09	1.00
251	2	37	6	5	0	50	2.28	0.69
252	5	29	9	13	0	56	2.54	0.94
253	1	11	10	22	7	51	3.45	1.03
254	13	24	2	13	1	53	2.34	1.15

255	10	7	5	24	12	58	3.36	1.39
256	4	18	9	18	7	56	3.11	1.19
257	1	1	6	36	8	52	3.94	0.72
258	3	10	23	12	2	50	3.00	0.92
259	0	3	12	31	6	52	3.77	0.72
260	2	4	11	28	3	48	3.54	0.89
261	2	8	14	24	2	50	3.32	0.93
262	1	15	15	16	4	51	3.14	0.99
263	3	20	16	11	1	51	2.75	0.93
264	3	25	12	9	2	51	2.65	0.97
265	3	10	7	23	5	48	3.35	1.11
266	2	5	11	22	5	45	3.51	0.98
267	0	0	9	32	7	48	3.96	0.58
268	0	4	8	33	4	49	3.75	0.72
269	1	8	14	25	1	49	3.35	0.85
270	3	21	9	15	1	49	2.80	1.01
271	0	11	11	21	5	48	3.42	0.95
272	2	3	15	26	2	48	3.48	0.84
273	1	4	17	22	2	46	3.44	0.80
274	1	2	9	32	4	48	3.75	0.75
275	2	8	16	19	2	47	3.23	0.93
276	1	7	17	19	3	47	3.34	0.88
277	4	6	17	19	0	46	3.11	0.94

DISTRIBUTION FOR RELEVANCE

QUEST	1	2	3	4	5	NUMB	MEAN	DEVI
1	0	1	19	24	4	48	3.65	0.66
2	0	5	21	22	5	53	3.51	0.79
3	0	0	25	18	3	46	3.52	0.62
4	1	7	16	24	5	53	3.47	0.90
5	0	3	19	22	1	45	3.47	0.65
6	1	5	22	19	4	51	3.39	0.84
7	0	5	24	15	6	50	3.44	0.83
8	0	4	27	19	5	55	3.46	0.76
9	0	3	17	23	3	46	3.56	0.71
10	2	4	15	26	5	52	3.54	0.91
11	1	1	21	18	3	44	3.48	0.75
12	2	3	20	25	2	52	3.42	0.82
13	1	4	15	17	2	39	3.38	0.84
14	1	5	12	23	3	44	3.50	0.87
15	2	7	18	19	4	50	3.32	0.95
16	2	7	16	18	4	47	3.32	0.97
17	2	3	16	24	5	50	3.54	0.90
18	2	3	20	21	1	47	3.34	0.81
19	0	0	4	28	19	51	4.29	0.60
20	2	2	18	27	5	54	3.57	0.85
21	1	2	15	25	2	45	3.56	0.75
22	0	3	15	19	5	42	3.62	0.78
23	0	3	13	19	10	45	3.80	0.86
24	1	7	15	13	3	39	3.26	0.93
25	0	2	19	24	5	50	3.64	0.71
26	0	3	22	20	4	49	3.51	0.73
27	0	4	16	25	6	51	3.65	0.79
28	0	1	7	33	13	54	4.07	0.66
29	0	3	17	20	10	50	3.74	0.84
30	0	4	21	19	3	47	3.45	0.74
31	1	3	15	26	10	55	3.75	0.88
32	0	3	16	25	7	51	3.71	0.77
33	0	5	22	17	4	48	3.42	0.79
34	1	4	17	26	3	51	3.51	0.80
35	3	4	21	12	3	43	3.19	0.95
36	2	8	22	15	2	49	3.14	0.88
37	0	4	16	20	5	45	3.58	0.80
38	1	2	20	15	6	44	3.52	0.87
39	0	6	13	27	3	49	3.55	0.78
40	2	5	16	19	12	54	3.63	1.04
41	1	6	19	18	3	47	3.34	0.86
42	0	0	11	34	10	55	3.98	0.62
43	0	5	21	18	4	48	3.44	0.79
44	0	3	21	25	6	55	3.62	0.75
45	2	4	15	23	7	51	3.57	0.95
46	1	6	19	21	1	48	3.31	0.80
47	1	13	20	14	9	57	3.30	1.04
48	0	8	21	21	5	55	3.42	0.85
49	1	12	17	18	1	49	3.12	0.87
50	1	9	18	26	2	56	3.34	0.85
51	1	7	23	9	0	40	3.00	0.71
52	1	6	24	21	4	56	3.38	0.84
53	0	5	18	18	5	46	3.50	0.83
54	1	9	23	20	1	54	3.20	0.80
55	2	6	21	20	2	51	3.28	0.87
56	3	7	19	20	5	54	3.31	1.00

57	1	7	26	14	1	49	3.14	0.76
58	1	6	25	16	2	50	3.24	0.79
59	1	9	26	8	2	46	3.02	0.79
60	3	13	22	14	1	53	2.94	0.90
61	1	5	21	17	3	47	3.34	0.83
62	0	10	19	16	3	48	3.25	0.85
63	2	4	14	24	9	53	3.64	0.97
64	1	7	18	25	4	55	3.44	0.87
65	2	3	23	19	3	50	3.36	0.84
66	2	14	23	7	2	48	2.85	0.87
67	8	17	14	5	0	44	2.36	0.91
68	4	10	20	16	2	52	3.04	0.98
69	5	8	20	12	3	48	3.00	1.04
70	2	11	22	12	2	49	3.02	0.89
71	3	7	20	14	1	45	3.07	0.90
72	8	11	16	9	5	49	2.84	1.20
73	6	9	21	3	2	41	2.66	0.98
74	6	12	21	6	3	48	2.75	1.03
75	8	10	16	9	1	44	2.66	1.06
76	8	14	24	2	0	48	2.42	0.81
77	7	16	13	5	2	43	2.51	1.04
78	3	10	19	12	1	45	2.96	0.92
79	6	13	18	11	7	55	3.00	1.18
80	4	13	23	13	2	55	2.93	0.95
81	2	9	21	14	1	47	3.06	0.86
82	3	4	18	26	3	54	3.41	0.91
83	1	10	20	12	4	47	3.17	0.93
84	1	9	22	18	4	54	3.28	0.89
85	1	6	21	21	2	51	3.33	0.81
86	2	6	19	23	2	52	3.33	0.87
87	1	6	16	19	4	46	3.41	0.90
88	1	3	15	27	9	55	3.73	0.86
89	1	7	15	20	5	48	3.44	0.93
90	3	6	16	21	6	52	3.40	1.02
91	0	1	17	24	7	49	3.75	0.72
92	3	10	20	16	3	52	3.12	0.97
93	2	6	19	20	2	49	3.29	0.88
94	0	7	19	22	3	51	3.41	0.80
95	5	8	10	24	11	58	3.48	1.19
96	3	2	12	24	13	54	3.78	1.03
97	1	4	18	18	6	47	3.51	0.90
98	0	4	11	35	6	56	3.77	0.73
99	2	5	17	17	7	48	3.46	1.00
100	1	6	16	21	9	53	3.59	0.96
101	2	7	21	16	3	49	3.22	0.91
102	1	15	12	21	1	50	3.12	0.93
103	1	10	21	13	1	46	3.06	0.82
104	0	0	6	26	24	56	4.32	0.66
105	0	2	18	23	5	48	3.65	0.72
106	1	4	23	19	4	51	3.41	0.82
107	0	4	16	29	4	53	3.62	0.73
108	1	0	6	27	22	56	4.23	0.78
109	0	2	13	25	11	51	3.88	0.78
110	1	7	12	16	7	43	3.49	1.02
111	2	5	12	15	10	44	3.59	1.09
112	0	4	22	22	3	51	3.47	0.72
113	1	7	14	25	5	52	3.50	0.91
114	0	5	19	20	2	46	3.41	0.74
115	0	6	17	22	4	49	3.49	0.81
116	1	6	20	22	6	55	3.47	0.89
117	0	10	18	20	2	50	3.28	0.83
118	0	1	8	31	14	54	4.07	0.69
119	1	7	15	25	1	49	3.37	0.83
120	0	2	16	26	13	57	3.88	0.80
121	1	7	12	27	5	52	3.54	0.91
122	1	1	16	30	5	53	3.70	0.74

123	1	4	24	18	3	50	3.36	0.79
124	0	5	26	20	5	56	3.45	0.78
125	0	3	20	21	5	49	3.57	0.76
126	0	7	17	21	2	47	3.38	0.79
127	2	5	14	25	6	52	3.54	0.95
128	1	8	18	15	5	47	3.32	0.95
129	2	5	17	20	5	49	3.43	0.95
130	1	6	12	27	9	55	3.67	0.94
131	0	9	19	15	1	44	3.18	0.78
132	3	5	22	21	4	55	3.33	0.94
133	0	8	23	15	1	47	3.19	0.73
134	1	3	21	20	3	48	3.44	0.79
135	0	5	14	26	4	49	3.59	0.78
136	0	4	11	30	7	52	3.77	0.77
137	2	8	21	16	2	49	3.16	0.89
138	0	6	21	23	2	52	3.40	0.74
139	0	6	15	23	5	49	3.55	0.83
140	1	6	21	22	5	55	3.44	0.87
141	1	8	22	11	4	46	3.20	0.90
142	3	9	20	16	2	50	3.10	0.94
143	3	13	18	19	3	56	3.11	0.99
144	1	7	25	16	3	52	3.25	0.83
145	1	5	23	12	4	45	3.29	0.86
146	0	3	21	24	7	55	3.64	0.77
147	1	2	15	27	5	50	3.66	0.79
148	0	1	13	26	16	56	4.02	0.77
149	1	2	16	24	9	52	3.73	0.86
150	2	7	13	24	3	49	3.39	0.94
151	1	6	18	18	3	46	3.35	0.87
152	0	0	15	33	9	57	3.90	0.64
153	0	4	17	25	5	51	3.61	0.77
154	1	3	24	12	2	42	3.26	0.76
155	0	2	22	16	5	45	3.53	0.75
156	2	3	21	24	4	54	3.46	0.85
157	1	3	28	12	3	47	3.28	0.76
158	1	8	20	14	2	45	3.18	0.85
159	3	8	15	19	11	56	3.48	1.12
160	4	2	20	20	7	53	3.45	1.02
161	0	7	21	21	1	50	3.32	0.73
162	0	4	9	26	14	53	3.94	0.86
163	2	7	18	15	2	44	3.18	0.91
164	1	5	14	31	6	57	3.63	0.85
165	1	0	24	17	6	48	3.56	0.79
166	0	5	23	18	5	51	3.45	0.80
167	0	1	22	22	6	51	3.65	0.71
168	2	3	18	23	7	53	3.57	0.92
169	1	4	18	20	4	47	3.47	0.85
170	0	2	19	27	5	53	3.66	0.70
171	0	4	18	22	2	46	3.48	0.71
172	3	6	16	19	1	45	3.20	0.93
173	1	5	14	26	2	48	3.48	0.82
174	1	3	19	26	3	51	3.53	0.78
175	4	2	13	28	10	57	3.67	1.03
176	7	11	18	16	4	56	2.98	1.13
177	5	12	15	15	3	50	2.98	1.09
178	2	9	26	14	1	52	3.06	0.82
179	2	5	27	10	2	46	3.11	0.81
180	0	5	26	21	4	56	3.43	0.75
181	2	6	21	17	3	49	3.26	0.90
182	1	7	21	23	3	55	3.36	0.84
183	0	3	16	22	5	46	3.63	0.76
184	0	5	13	29	9	56	3.75	0.83
185	2	4	20	19	1	46	3.28	0.83
186	0	4	8	38	5	55	3.80	0.70
187	0	5	14	25	3	47	3.55	0.77
188	0	7	20	17	4	56	3.32	0.78

189	2	8	16	21	2	49	3.26	0.92
190	2	2	15	26	6	51	3.53	0.88
191	5	6	22	17	5	55	3.20	1.05
192	3	4	9	29	12	57	3.75	1.03
193	0	3	22	25	4	54	3.56	0.71
194	1	10	18	25	3	57	3.33	0.88
195	1	3	18	25	2	49	3.49	0.76
196	1	2	20	23	8	54	3.65	0.84
197	2	5	18	19	5	49	3.41	0.95
198	1	8	26	13	4	52	3.21	0.86
199	2	10	26	12	2	52	3.04	0.85
200	1	10	25	17	2	55	3.16	0.83
201	0	2	18	24	4	48	3.63	0.70
202	0	1	19	27	5	52	3.69	0.67
203	1	8	15	13	3	40	3.22	0.94
204	1	6	18	27	0	52	3.37	0.76
205	3	1	21	20	3	48	3.40	0.88
206	1	4	11	30	1	47	3.55	0.77
207	0	6	16	20	7	49	3.57	0.88
208	1	6	13	24	6	50	3.56	0.92
209	0	4	25	12	0	41	3.19	0.59
210	0	3	21	19	0	43	3.37	0.61
211	0	4	20	21	4	49	3.51	0.76
212	1	6	15	28	3	53	3.49	0.84
213	1	7	16	21	2	47	3.34	0.86
214	11	17	12	8	3	51	2.51	1.16
215	5	7	19	15	3	49	3.08	1.05
216	2	8	20	23	2	55	3.27	0.88
217	4	8	14	21	4	51	3.25	1.06
218	3	7	18	17	5	50	3.28	1.02
219	0	3	19	20	6	48	3.60	0.78
220	4	8	14	19	8	53	3.36	1.13
221	0	0	2	28	22	52	4.38	0.56
222	1	3	15	23	6	48	3.63	0.86
223	2	5	9	20	17	53	3.85	1.09
224	0	1	1	10	46	58	4.74	0.57
225	1	1	11	27	9	49	3.86	0.81
226	1	5	15	22	9	52	3.63	0.94
227	0	1	1	27	20	49	4.35	0.62
228	2	5	6	22	21	56	3.98	1.08
229	2	6	11	23	10	52	3.63	1.04
230	2	2	11	21	19	55	3.96	1.01
231	0	3	15	24	8	50	3.74	0.80
232	2	5	6	26	18	57	3.93	1.04
233	5	3	14	18	11	51	3.53	1.18
234	3	2	15	17	14	51	3.72	1.09
235	2	0	3	25	22	52	4.25	0.87
236	2	7	15	19	14	57	3.63	1.09
237	2	3	17	18	11	51	3.65	1.01
238	1	2	14	20	14	51	3.86	0.93
239	2	6	8	25	16	57	3.82	1.06
240	2	1	17	27	11	58	3.76	0.90
241	0	4	16	18	13	51	3.78	0.91
242	4	4	8	30	10	56	3.68	1.07
243	1	5	10	25	9	50	3.72	0.94
244	4	3	9	17	21	54	3.89	1.20
245	1	1	12	24	9	47	3.83	0.83
246	1	1	15	23	13	53	3.87	0.87
247	1	0	11	31	8	51	3.88	0.73
248	1	3	20	24	8	56	3.63	0.86
249	0	5	10	28	7	50	3.74	0.82
250	5	8	15	18	6	52	3.23	1.14
251	1	13	16	15	1	46	3.04	0.88
252	5	8	18	23	3	57	3.19	1.03
253	1	2	14	24	10	51	3.78	0.87
254	0	4	14	24	12	54	3.81	0.86

255	4	5	5	27	15	56	3.79	1.14
256	3	10	14	18	9	54	3.37	1.13
257	1	6	13	26	6	52	3.58	0.91
258	1	6	25	14	2	48	3.21	0.79
259	0	7	19	21	4	51	3.43	0.82
260	0	5	28	12	2	47	3.23	0.69
261	0	4	27	15	2	48	3.31	0.68
262	0	11	18	16	3	48	3.23	0.87
263	0	5	27	15	1	48	3.25	0.66
264	0	7	26	13	2	48	3.21	0.73
265	1	3	18	21	3	46	3.48	0.80
266	1	5	15	18	4	43	3.44	0.90
267	0	4	17	21	4	46	3.54	0.77
268	0	7	19	18	3	47	3.36	0.81
269	1	8	17	18	3	47	3.30	0.90
270	0	4	22	19	2	47	3.40	0.70
271	0	5	20	18	3	46	3.41	0.77
272	1	8	22	13	2	46	3.15	0.83
273	1	10	23	9	1	44	2.98	0.78
274	0	6	25	12	3	46	3.26	0.76
275	2	10	18	13	1	44	3.02	0.89
276	2	9	19	12	2	44	3.07	0.91
277	2	9	17	15	0	43	3.05	0.86

ANNEX II

COMMUNICATIONS HIGHER ORDER LANGUAGE QUESTIONNAIRE

(Unscrambled)

Dear Collegue:

SOFTECH Inc. and Data Systems Analysts, Inc. are investigating the requirements for a higher order language (HOL) intended for applications in communications software. To this end we have taken the unusual approach of asking experienced communications programmers what they feel about the subject. Your answers will play a significant role in determining the structure and facilities of such a language.

The issues probed here include the design process, software transportability, software maintenance, reliability, security, privacy, system operation and maintenance, system resource requirements, language efficiency, characteristics of the language, the nature of communications programming, and other comments about it. Furthermore, any ideas or "pet peeves" you have regarding HOL's or communications programming that impinge on the design of a HOL for this application, would also be welcome.

We have not asked all possible questions, only those to which we feel the answer will be significant or involves issues which we cannot resolve without your help. For example, we have not asked if assembly language inclusions should be allowed, since it is obvious to us that this is mandatory. Similarly, we have not asked about the need for bit level control, etc. Nevertheless, there still are many questions to be answered and we beg your indulgence in this.

We would like this questionnaire to be answered at a "dead run." If you do not feel you can answer the question, skip it rather than try to figure out what was intended. Please do not research anything. A few seconds of reflection should suffice in most cases and your initial "gut" level feelings are closer to what we want rather than the result of any long intellectual agonizing. Besides, it takes less of your time.

Thank you for your help.

INSTRUCTIONS

All questions have the form of positive assertions such as, "The moon is made of green cheese." We ask you to put down two numbers from 1 to 5. The first number indicates your degree of agreement or disagreement with the assertion according to the following scheme:

1. Strongly disagree
2. Disagree
3. Indifferent
4. Agree
5. Strongly agree

The second number reflects your opinion regarding the assertion and your estimate of the relevancy of that assertion to the communications design. Use the following to indicate your answer.

1. Of no importance at all
2. Marginal interest
3. Moderate relevance
4. Quite important
5. Crucial issue/fundamental problem

We would prefer that you try to give both numbers to each question, but if you cannot, perhaps because you have an opinion but you do not know how important the issue is, or you know the importance of the issue, but really have no basis for an opinion, you could skip one or the other numbers. However we urge you to give us both numbers since your assessment of the relevancy is as valuable as your opinion regarding the assertion.

	Agreement (1-5)	Relevance (1-5)
1. Declarations of all properties of variables shall be made explicitly.	_____	_____
2. The language will allow default declarations of properties where obvious and safe.	_____	_____
3. It will not be necessary to make explicit declarations of loop control variables.	_____	_____
4. The compiler shall recognize missing declarations and insert them in accordance to a well documented scheme.	_____	_____
5. The compiler shall insert missing declarations which will require explicit confirmation by the programmer. Once confirmed, these default declarations shall have the same status as those supplied by the programmer.	_____	_____
6. All declarations of a given type (e.g. integers, literals, pointers, tables, etc.) must be grouped with others of the same type.	_____	_____
7. Declaration may be intermixed with code.	_____	_____
8. Declarations can appear in any order in the declaration area and can be intermixed as to type.	_____	_____
9. All declarations shall apply to the entire module in which they are made; properties of objects remain constant in a given module.	_____	_____
10. The declarations of a calling module can override the declarations of the modules it calls.	_____	_____
11. Local declarations can be overridden by common system wide declarations.	_____	_____
12. The declarations of a submodule can override the declarations of modules that call it.	_____	_____
13. The highest level declaration overrides lower level declarations unless specific instructions are given to the contrary.	_____	_____

NOTE: "1" demotes strong disagreement or low relevance; "5" denotes strong agreement or high relevance.

	Agreement (1-5)	Relevance (1-5)
14. Read/write/execute restrictions can be associated with routines and data base elements.	_____	_____
15. Read/write/execute restrictions can be associated with routines and data base elements and the compiler performs the necessary compile time checks.	_____	_____
16. Read/write/execute restrictions can be associated with routines and data base elements and the compiler inserts the proper run time checks.	_____	_____
17. Access by routines to data base elements and to other routines shall be allowed unless explicitly forbidden by the programmer.	_____	_____
18. Read/write/execute restrictions can be associated with routines and data base elements. The compiler performs proper compile time checks, but inserts run time check code only if directed by the programmer.	_____	_____
19. The compiler shall generate a complete cross reference index of accesses to the data base, subroutine calls, macro calls, indicating the allowed access (read/write/execute) the identification of the caller, and the type of accesses attempted.	_____	_____
20. When the same data structure (e.g. a queue block) is used with different interpretations, the interpretation is generally constant within the subroutine or module.	_____	_____
21. The interpretation of the format of a data base element (e.g. queue block) cannot be changes within a subroutine. Control must be transferred to a different procedure.	_____	_____
22. To change the interpretation of a data element (e.g. a queue block) an explicit statement which identifies the format to be used next must be supplied (this entails no run time code).	_____	_____
23. The interpretation of data structures can be changed at will - i.e., different formats may be used, different masks, etc. The compiler will make no checks and no prohibitions.	_____	_____

NOTE: "1" denotes strong disagreement or low relevance; "5" denotes strong agreement or high relevance.

	Agreement (1-5)	Relevance (1-5)
24. The interpretation of a data structure can be changed at will as long as all variations in the interpretation have been declared. Compile time checks will be performed, but no run time checks will be inserted, unless specifically dictated by the programmer.	_____	_____
25. The management of dynamically allocated stacks in support of re-entrant code will be done by compiler generated code.	_____	_____
26. The management of dynamically allocated stacks in support of re-entrant code will be done by compiler generated code, unless the programmer has explicitly specified that some other process be used.	_____	_____
27. An explicit declaration must be made to identify a routine as re-entrant.	_____	_____
28. All routines will be re-entrant unless explicitly declared not to be.	_____	_____
29. All routines will be re-entrant.	_____	_____
30. Compiler generated dynamic storage allocations shall be provided (and managed by compiler generated code) for any process that requests it by means of suitable declarations.	_____	_____
31. Dynamic storage allocations can be supplied at programmer's option for character strings.	_____	_____
32. Dynamic storage allocation can be supplied at programmer's option for task control queues.	_____	_____
33. Dynamic storage allocation will be supplied at programmer's option for all array or structures whose size has not been declared at compile time.	_____	_____
34. Dynamic storage allocation will be supplied by request for any definable data type.	_____	_____
35. Included in the specification of a data structure such as a queue block or buffer block is the specification of the programmer defined storage allocation and return routines to be used for the management of that data element.	_____	_____

NOTE: "1" denotes strong disagreement or low relevance; "5" denotes strong agreement or high relevance.

	Agreement (1-5)	Relevance (1-5)
36. The programmer can specify the storage management routine to be used with any pool of data types. If no such specification is provided, the compiler will generate such facilities.	_____	_____
37. One and only one storage allocation and management scheme (routine) can be used with a given pool.	_____	_____
38. Any number of different storage allocation and management schemes (routines) can be defined for use with any given pool.	_____	_____
39. Any storage pool declared to be dynamic must have an associated allocation/management scheme or routine. Failure to declare one will be treated by the compiler as a bug.	_____	_____
40. The number of elements in use in a compiler managed dynamic storage pool can be querried.	_____	_____
41. Compiler managed storage pools (e.g. stacks for re-entrant code) can be given attributes like any other data structure (e.g. read/write/execute privileges, security tags, special processing etc.).	_____	_____
42. Compiler managed storage pools will transfer control to programmer specified routines upon detection of illogical conditions or when they run out of space.	_____	_____
43. Initial values must be explicitly declared for every data item, data structure, pools, etc. Reasonable abbreviations are available.	_____	_____
44. Initial values for data items, structures, pools, etc., need not be declared explicitly. The compiler will initialize to a default value of logical 0.	_____	_____
45. When the compiler initializes a memory location to a default value, it will make an explicit statement to that effect to bring the fact to the programmer's attention.	_____	_____
46. Initialization of memory can be specified by means of a process which is executed at compile time (this does not preclude other methods of initialization).	_____	_____

NOTE: "1" denotes strong disagreement or low relevance; "5" denotes strong agreement or high relevance.

	Agreement (1-5)	Relevance (1-5)
47. Initialization of memory can be accomplished by a programmer specified run time process (does not preclude other methods of initialization).		
48. Initialization of memory is specified as part of the declarations.		
49. The programmer can specify an initialization process to be executed on every entrance to a subroutine.		
50. The programmer can specify initialization processes that are executed by the loader at load time.		
51. Run time initialization routines can be given dummy names, with the actual process determined at run time as part of the calling sequence.		
52. The size of storage allocated to integers is determined by the compiler.		
53. The programmer can specify the number of characters to be used to store any given integer.		
54. The programmer can specify the number of bits to be used to store any given integer.		
55. The programmer must specify the number of characters to be used to store integers.		
56. The compiler automatically inserts the code needed to check for arithmetic operation overflow.		
57. The programmer can specify whether run time code for arithmetic overflow detection should be inserted by the compiler.		
58. The programmer must specify the arithmetic overflow control routine to be used for reacting to integer overflow as a part of the declaration of that integer variable. Suitable abbreviations exist.		

NOTE: "1" denotes strong disagreement or low relevance; "5" denotes strong agreement or high relevance.

	Agreement (1-5)	Relevance (1-5)
59. The programmer can specify which arithmetic overflow control routine should be used for reacting to integer overflow, if none is specified, the compiler will use its own.	_____	_____
60. The programmer can specify which arithmetic overflow control routine should be used for reacting to integer overflow, if none is specified, the compiler will treat this as a bug.	_____	_____
61. The programmer must specify the range of values expected for integers as part of the declaration (e.g. between 47 and 96).	_____	_____
62. The programmer must specify both the range of expected integer values and the amount of storage (say, in bits) to be allocated to the integer. The compiler checks compatibility of the two declarations.	_____	_____
63. The programmer specifies range of values for each integer and/or size of the field. The compiler performs all compatibility checks at compile time and flags conflicts.	_____	_____
64. If the programmer has specified a valid range attribute for a parameter, the actual value of the parameter will be checked by compiler generated code at run time.	_____	_____
65. If the programmer has specified a valid range attribute for a parameter, the actual value of the parameter will be checked by at run time by compiler generated code if, and only if, directed to do so by the programmer.	_____	_____
66. The language provides exponentiation of integers for powers of 2. For example 2^N, where N is an integer.	_____	_____
67. The language provides logarithms rounded to next lowest whole number to base 2 for integers.	_____	_____
68. The language provides ABSOLUTE VALUE operator for integers.	_____	_____
69. Arithmetic operations symbols for integers and floating point numbers are different. Example: "*" vs "x."	_____	_____

NOTE: "1" denotes strong disagreement or low relevance; "5" denotes strong agreement or high relevance.

	Agreement (1-5)	Relevance (1-5)
70. In converting from real numbers to integers, the compiler shall discard the fractional part of the number.	_____	_____
71. In converting from real numbers to integers, the compiler shall round up or down to the nearest whole number.	_____	_____
72. Five (5) decimal digits of accuracy are sufficient for floating point real numbers. There are no double precision operations.	_____	_____
73. Seven (7) decimal digits of accuracy are sufficient for floating point real numbers. There are no double precision operators.	_____	_____
74. Nine (9) decimal digits of accuracy are sufficient for floating point real numbers. There are no double precision operators.	_____	_____
75. There are significant number of instances in which more than 9 digits of accuracyare required for floating point real numbers.	_____	_____
76. Real numbers shall be constrained to lie between $10^{\pm 38}$.	_____	_____
77. Real numbers shall be constrained to lie between $10^{\pm 620}$.	_____	_____
78. The compiler shall perform compile time range checks on the validity of the range of real numbers.	_____	_____
79. The compiler will insert run time range checks on the validity of real numbers if directed to do so by the programmer.	_____	_____
80. The possible range of real numbers must be specified by the programmer as a part of the declaration.	_____	_____
81. The possible range of real numbers can be specified by the programmer as a part of the declaration. If such a specification is made, the compiler will insert appropriate run time checking code.	_____	_____
82. In addition to the appearance of the relation operators equal not equal, greater, greater or equal, etc., in tne context of a control statement such as "IF A GRT B THEN ...ELSE...". The language provides Boolean operator whose value is TRUE, if the relation is satisfied. Example: SET A TRUE IF B = C.	_____	_____

NOTE: "1" denotes strong disagreement or low relevance; "5" denotes strong agreement or high relevance.

	Agreement (1-5)	Relevance (1-5)
83. The language provides a range checking three way conditional statement of the form: IF A ≤ X ≤ B IS LOW THEN DO.... IS BETWEEN THEN DO.... IS HIGH THEN DO.... This is provided for integers only.	_____	_____
84. The language provides a range checking three way conditional statement of the form: IF A ≤ X ≤ B IS LOW THEN DO.... IS HIGH THEN DO.... IS BETWEEN THEN DO.... This is provided for real numbers.	_____	_____
85. The language provides conditional statements of the form shown below for any data type which makes sense. IF A ≤ X ≤ B IS LOW THEN DO.... IS BETWEEN THEN DO.... IS HIGH THEN DO....	_____	_____
86. The programmer can specify data types of his own in addition to those, such as integers, reals, pointers, which are built into the language. For example, you could specify SECURITY as a data type.	_____	_____
87. The language does not permit implicit conversions from one data type to another (example, converting integers to real). All conversions must be explicit. This does not usually result in run time code. Mixed operations are forbidden.	_____	_____
88. Bit field or bit string constants can be specified in binary.	_____	_____
89. Bit field or bit string constants can be specified in octal.	_____	_____
90. Bit field or bit string constants must be specified in hexadecimal.	_____	_____
91. Bit field or bit string constants can be specified in binary, octal, or hexadecimal.	_____	_____
92. Bit field or bit string constants can be specified in any base (2,4,8,16, etc.) except 10.	_____	_____

NOTE: "1" denotes strong disagreement or low relevance; "5" denotes strong agreement or high relevance.

	Agreement (1-5)	Relevance (1-5)
93. Bit field or bit string constants are specified in decimal.	_____	_____
94. In operations involving two different bit fields where the fields are not of the same length, the compiler will pad the shorter operator on the left with zero bits.	_____	_____
95. In operations involving two different bit fields where the fields are not of equal length, the compiler will pad the shorter string on the right with zero bits.	_____	_____
96. The language will not permit bit field operations (such as masks, AND, OR, etc.) on fields of different lengths.	_____	_____
97. In bit field operations where the fields are of different lengths, padding will be on the right or left of the shorter string, with zeros or ones as the programmer directs.	_____	_____
98. The compiler will not permit bit string operations on fields of different lengths. The programmer will use appropriate shift and pad operations to convert the strings to equal length strings.	_____	_____
99. There are a significant number of cases in which there are bit string operations carried out on strings whose length is not known until run time.	_____	_____
100. There are a significant number of cases in which it is convenient to treat a portion of a bit string as a bit string in its own right, i.e. to give a substring a different name.	_____	_____
101. There are a significant number of cases in which it is desired to operate on a substring of a bit string, but in which the length of the substring and its position within the larger string is not known until run time.	_____	_____
102. There are a significant number of cases in which it is desirable to operate on a substring of a bit string but in which the position of the substring within the larger string is not known until run time.	_____	_____

NOTE: "1" denotes strong disagreement or low relevance; "5" denotes strong agreement or high relevance.

	Agreement (1-5)	Relevance (1-5)
103. There is a significant number of cases in which it is desirable to operate on a substring of a bit string where the position of the leading bit is known at compile time, but the length of the string is not known until run time.	_____	_____
104. The language will support operations on character strings, such as COMPARE, MOVE, INSERT, etc.	_____	_____
105. The length of all character strings is known at compile time.	_____	_____
106. The maximum length of all character strings can be set at compile time.	_____	_____
107. The programmer can specify if a character string is to be fixed or variable length.	_____	_____
108. The language will support ASCII character codes directly.	_____	_____
109. All operations involving characters will be based on the assumption that the characters are represented in ASCII.	_____	_____
110. The language will directly support ITA-2 codes.	_____	_____
111. The language will directly support ITA-5 code.	_____	_____
112. The programmer can specify the character code to be used in character related operations by declaring a table. The programmer specified code will be used in all character operations, particularly ordering tests.	_____	_____
113. The compiler will assume character operations to be in ASCII unless specifically declared to be in some other character code.	_____	_____
114. The compiler will assume character operations to be in a programmer specified default character set if no specification is made. Character operations can be directed to be in terms of any character set, but this takes additional writing.	_____	_____
115. Characters will be packed into 8 bit fields aligned on physical character, or word boundaries.	_____	_____

NOTE: "1" denotes strong disagreement or low relevance; "5" denotes strong agreement or high relevance.

	Agreement (1-5)	Relevance (1-5)
116. Characters will be packed into the tightest packing allowed by the character set. The compiler will automate all packing, unpacking and extraction instructions.	_____	_____
117. The language has a character string operation that scans a specified string in reverse order for the appearance of a given substring.	_____	_____
118. The language contains an operation that searches a character string for the appearance of a specified substring. It returns either the position at which the substring was found, or indication that no such string was found.	_____	_____
119. The language contains an operation that searches a character string for the appearance of a given character sequence and returns the character string leading to the first appearance of the searched for string.	_____	_____
120. The language contains an operation that searches a character string for the appearance of a given character sequence and returns the character string following the first appearance of the searched for string.	_____	_____
121. The language contains an operation that searches a character string for the appearance of a given character sequence starting with a specified number of characters past the beginning of the searched string.	_____	_____
122. In operations involving character strings of different lengths, such as a move operation, truncations and padding is never allowed.	_____	_____
123. In operations such as a move involving character strings, the compiler will generate code that will execute a branch to a specified location if the source string is longer than the target string.	_____	_____
124. In operations such as a move involving character strings, the compiler will generate code that will pad the shorter strings with with logical blanks on the right.	_____	_____
125. In operations such as a move instruction involving character strings of different lengths, the compiler will generate code that will pad the short string on the right with logical zeros.	_____	_____

NOTE: "1" denotes strong disagreement or low relevance; "5" denotes strong agreement or high relevance.

	Agreement (1-5)	Revelance (1-5)
126. In moving a long string into a short string area, the compiler will generate code that will cause a branch to a specified location if the truncation of the longer string requires the removal of non-blank characters. Otherwise, it will truncate characters on the right and perform the move.	_____	_____
127. Pointers to data base structures and fields will be distinguished from integers and other data types.		
128. The same pointer to data base elements may be used to point to fields having different characteristics, or may be functionally incompatible. That is, to a bit field or to an integer, say.	_____	_____
129. A given pointer to a data base element always points to one and only one kind of data base element or field.	_____	_____
130. Incrementing a pointer should result in pointing to the next item of the same kind in the table rather than to the next word or character in memory.	_____	_____
131. Pointers to data base elements will be packed to conserve space.	_____	_____
132. Pointers to data base elements will be packed to conserve space if so directed by the programmer.	_____	_____
133. The programmer specifies the permissible range of values of pointers to the data base. The compiler performs compile time checks (as possible) and inserts run time check codes as necessary.	_____	_____
134. The insertion of run time check code to check the validity of a pointer (to a data base element) is controlled by the programmer.	_____	_____
135. Every parameter that appears in a subroutine definition must appear in every call to that subroutine.	_____	_____
136. Parameters of a subroutine call can be labeled as optional or mandatory. A call must have all mandatory parameters.	_____	_____

NOTE: "1" denotes strong disagreement or low relevance; "5" denotes strong agreement or high relevance.

	Agreement (1-5)	Revelance (1-5)
137. If parameters are omitted in subroutine calls, their place is denoted by means of a special character, such as: CALL SUBROUTINE SAME (TOM,*, HARRY).	_____	_____
138. If parameters are omitted in a subroutine call, and the parameters in question are optional, they are left out of the call, as in the following example: CALL SUBROUTINE SAME (TOM,,HARRY).	_____	_____
139. There are instances in which it is desirable to define subroutines in which some parameters are optional and others are mandatory.	_____	_____
140. There are instances in which it is desirable to define subroutines in which the number of parameters is variable.	_____	_____
141. There are instances in which it is desirable to define subroutines in which the number of parameters is variable and cannot be determined until run time, that is, the number of parameters may itself be a parameter of the call.	_____	_____
142. There are instances in which it is desirable to define subroutines in which the maximum number of parameters cannot be determined and the actual number of parameters is not known until run time.	_____	_____
143. The number of parameters to a subroutine call need not be constant nor be specified until run time. However, an explicit declaration to this effect is required. Subroutines with fixed number of parameters can of course be written.	_____	_____
144. If and when a subroutine is defined which has a variable number of parameters, some of which are optional and some of which are mandatory, the programmer must specify how the deleted parameters are to be handled.	_____	_____
145. If and when a subroutine is defined which has a variable number of parameters, some of which are mandatory and others of which are optional, the handling of linkages for deleted optional parameters shall be controlled by the compiler.	_____	_____

NOTE: "1" denotes strong disagreement or low relevance; "5" denotes strong agreement or high relevance.

	Agreement (1-5)	Relevance (1-5)
146. Subroutines must have a single entrance and exit. Multiple entrances and exits are simulated by an extrance and/or exit parameters.	_____	_____
147. Subroutines may be defined as having any number of entrances.	_____	_____
148. Subroutines may be defined as having any number of exits.	_____	_____
149. Subroutines can be defined as having any number of entrances and/or exits.	_____	_____
150. If multiple entrances and exits can be defined for subroutines, there is no need to have subroutines with a variable number of parameters in the call.	_____	_____
151. The language must have the ability to pass arrays (bit strings, tables, character strings, vectors, etc.) as parameters. Not to be confused with passing the name of an array.	_____	_____
152. Arrays of bits or characters, words, etc., can be moved, assigned, passed and operated in much the same manner as indivudual words, characters or bits.	_____	_____
153. Character strings can, at the programmer's option, be treated as arrays and indexed as such.	_____	_____
154. Considering index variables used to access elements of an array: it is necessary to be able to have constant value indexes.	_____	_____
155. Considering index variables used to access elements of an array: it is necessary to be able to add such variables.	_____	_____
156. Index arithmetic can be complicated and may require general arithmetic operations or expressions, such as A+B*C.	_____	_____
157. Indexing operations (for arrays, say) can be complicated and could require the execution of a function or subroutine.	_____	_____

NOTE: "1" denotes strong disagreement or low relevance; "5" denotes strong agreement or high relevance.

	Agreement (1-5)	Relevance (1-5)
158. Indexing operations (for arrays, say) can be complicated and could require the execution of one of several functions or subroutines whose identity in any specific instance is not determined until run time.	_____	_____
159. In defining a table, such as a queue table in which there is likely to be several alternative formats and interpretations, it is always possible to arrange things so that the common parts (among these varations) are grouped at the beginning of the entry followed by the variable parts.	_____	_____
160. In tables whose entries can be interpreted in a variety of ways depending on use (such as a queue block) the common parts of these variant structures may appear in the beginning, middle, or end for good reasons.	_____	_____
161. In addressing tables names of entries can be built up in a simple manner. For example, if the name of a table is "IMA" a specific entry in that table might be named "YANKEE," while another might be named "JKL." This would result in names in the form "IMA.YANKEE" or "IMA.JKL." This feature is in addition to the typical array indexing features of most language.	_____	_____
162. The language will not have indexing modes of address. Instead, individual entries in a table are named and access is provided by a compound name of the form: "TABLENAME.TABLEENTRYNAME"	_____	_____
163. A simple scheme for building up names of things will be provided. Of the form, NNNN.AAAA.BBBB, resulting in names such as : THIS.IS.TABLE1	_____	_____
164. The length (e.g. number of entries) of all arrays and structures can be specified at compile time.	_____	_____
165. The length (e.g. number of entries) of most arrays and structures can be specified at compile time.	_____	_____

NOTE: "1" denotes strong disagreement or low relevance; "5" denotes strong agreement or high relevance.

		Agreement	Relevance
166.	The maximum length (e.g. number of entries) of all arrays and structures can be specified at compile time.	_____	_____
167.	The maximum length (e.g. number of entries) of most arrays and structures can be specified at compile time.	_____	_____
168.	There are arrays and structures whose length (e.g. number of entries) can only be determined at run time.	_____	_____
169.	It is possible to perform assignment operations (e.g. move, load) on entire structures or elements thereof (e.g. queue blocks, loaded with a constant entry in one statement).	_____	_____
170.	In complex move operationsinvolving a data structure (say from a queue block to another queue block) the move will be allowed if and only if the target structure and the source structure are of the same type.	_____	_____
171.	In a complex move operation (say, moving a queue block into another queue block) only those fields which are of the same type and name will be moved, the rest will not participate in the update process.	_____	_____
172.	The language provides a comparison operator for structures that allow a specific queue block to be compared to a stored skeleton.	_____	_____
173.	The language provides a comparison operator that allows a specific queue block to be compared to a stored skeleton. The comparison is done only for those fields that are compatible and have the same name. The rest of the fields do not participate in the comparison.	_____	_____
174.	It is necessary to nest conditional branch statements as in the following example: IF A=B THEN [IF C=D THEN DO...ELSE...] ELSE DO....	_____	_____
175.	Control structures such as loops will be terminated with an explicit termination symbol such as BEGIN/END, or CALL/RETURN.	_____	_____

NOTE: "1" denotes strong disagreement or low relevance; "5" denotes strong agreement or high relevance.

	Agreement (1-5)	Relevance (1-5)
176. Control structures such as loops will be terminated with an explicit termination symbol which is the control symbol written backwards. For example: DO .... OD	_____	_____
177. Control structures such as loops will be terminated with an explicit termination symbol which is the control symbol followed by "END," such as, "DO .....DOEND," "CALL .....CALLEND," ....."IF .....IFEND."	_____	_____
178. Iterations of loop control variables can occur at the beginning of the loop.	_____	_____
179. Iteration of control variables in loops can occur at the end of the loop.	_____	_____
180. It might be necessary to iterate the control variable in a loop someplace in the middle of the loop.	_____	_____
181. Loop termination conditions are always tested prior to iteration of the control variable.	_____	_____
182. Loop termination conditions are tested after iteration of the control variable.	_____	_____
183. The relation between the point in the loop at which control variables are iterated and the point at which the control variable is tested for satisfaction of termination conditions is effectively arbitrary and cannot be done in any single uniformed way.	_____	_____
184. Control variables in a loop may be subscripted or indexed variables, that is, they may be elements of an array or table.	_____	_____
185. If the control variable of a loop is subscripted, that is, the control variable is stored in an array, the subscript which designates the control variable will be evaluated only once per entrance to the loop. This means that the identity of the control variable may not be changed within the loop it controls.	_____	_____

NOTE: "1" denotes strong disagreement or low relevance; "5" denotes strong agreement or high relevance.

		Agreement	Relevance
186.	The contents of the control variable of a loop (i.e. the loop count) can only be accessed within the body of the loop.	_____	_____
187.	The contents of the memory location that stores the value of the loop control variable in a loop is arbitrary when the program exits the loop.	_____	_____
188.	A loop may have more than one simultaneous control variable, the first variable to meet the loop termination conditions will cause the termination.	_____	_____
189.	Most loops are controlled by counters or indexes which must be safe stored for recovery purposes.	_____	_____
190.	Loop control variables are usually local to the routine in which they appear and rarely have significant value outside of that routine.	_____	_____
191.	Loop control variables (e.g. loop counters) are generally not kept in common tables.	_____	_____
192.	The contents of the memory location that stores the loop count is not accessible once the loop has been exited.	_____	_____
193.	If an increment value is not explicitly specified for a loop, the compiler will assume a value of 1 was intended.	_____	_____
194.	If the increment value for a loop is not explicitly provided, the compiler will assume that there has been a bug.	_____	_____
195.	The increment value for a loop is constant.	_____	_____
196.	The increment value for a loop may be variable, but will remain constant within the specific call (for a given entrance to the loop).	_____	_____
197.	The value of the increment for a loop control variable may be variable and may be modified within the loop.	_____	_____
198.	The increment value for a loop may be a function or a simple algebraic expression.	_____	_____

NOTE: " 1" denotes strong disagreement or low relevance; "5" denotes strong agreement or high relevance.

	Agreement (1-5)	Relevance (1-5)

199. Increment value for a loop may be determined by a procedure. _____ _____

200. It is possible to specify a loop by indicating only the termination value. In such cases, the compiler will assume that the loop is started with a value of 1 and is incremented by 1 each time through. _____ _____

201. In exiting a set of nested loops, control must pass through each level in order. _____ _____

202. In exiting a set of nested loops, it is possible to exit several levels simultaneously. _____ _____

203. A statement of the form shown below is useful:

```
DO CASE I OF:
   L1:  BLAH BLAH
   L2:  DO ETC.
   :
   LN:  BLAH + BLAH...
CASEEND,   where L1, L2, etc., are statements or procedures.
```

_____ _____

204. In constructing a jump table controlled by variable i, the language permits ranges of the variable to be specified as follows:

```
DO CASE i
   i =1-34     L56
     =35-76  , L87 etc.
```

indicating that statement L56 will be executed if the control variable is between 1 and 34. _____ _____

205. In constructing a jump table, controlled by variable i, the language allows specifications of the form:

```
DO CASE i        STATEMENT
  1               L56
  2,4,7           L17
  1,2,4,          L90
                  CASEEND
```

The program will execute all statements in the order of appearance corresponding to the value of the control variable. In the above example, a value of 1 will cause statements L56 and L90 to be executed. A value of 2 will cause L17 and L90 to be executed, while a value of 7 will cause L17 to be executed. _____ _____

NOTE: "1" denotes strong disagreement or low relevance; "5" denotes strong agreement or high relevance.

	Agreement (1-5)	Relevance (1-5)
206. The language has facilities for the direct specification of decision tables.		
207. In executing decision table specifications, the program will always execute all statements that satisfy the conditions.		
208. In executing decision table specifications, the program will execute only the first statement that satisfies the conditions.		
209. In executing decision table specifications, the statements can be labeled (in the decision table) as to whether or not they should be executed if some previous statement that satisfies the same conditions has already been executed.		
210. In implementing decision table processing, the programmer can direct the decision table processor to pick up the next statement that satisfies the conditions of the table if there is more than one statement corresponding to a particular set of conditions.		
211. A range check is always inserted by the compiler to check the validity of the control variable of a jump table.		
212. The compiler inserts run time range validation check code for all jump tables.		
213. Run time validation of the control variable in a jump table will be done by compiler inserted code if so directed by the programmer.		
214. It is essential to be able to insert non-printing characters into comments.		
215. Comments are distinguished by a pair of terminator symbols such as "[" and "]", or "#" and "#" or "¢" and "¢".		
216. Comments are distinguished by a single initial symbol such as "¢" and terminated by the end of the line.		
217. Comments must appear in certain columns of a formatted source.		

NOTE: "1" denotes strong disagreement or low relevance; "5" denotes strong agreement or high relevance.

		Agreement (1-5)	Relevance (1-5)
218.	It is desirable to be able to insert executable statements within a comment, for example, "¢ DO .....DOEND¢."	_____	_____
219.	The most difficult part of communications programming is understanding the system specifications.	_____	_____
220.	The most difficult part of communications software is the communications channel interface software.	_____	_____
221.	Timing considerations are crucial to communications software.	_____	_____
222.	Hardware characteristics are the most troublesome aspect of communications programming.	_____	_____
223.	The most difficult part of communications software are related to recovery and restart.	_____	_____
224.	Message accountability and the avoidance of misrouting are crucial issues in the design of communications software.	_____	_____
225.	Process control, executive structure, and intermodule communications dominate the design issues in communications software.	_____	_____
226.	More effort is spent on the design of the data base than any other single thing.	_____	_____
227.	Interrupt processing is a major design issue.	_____	_____
228.	An HOL will increase a programmer's productivity.	_____	_____
229.	Programming with an HOL will probably be more difficult than assembly language.	_____	_____
230.	HOL program will be easier to debug.	_____	_____
231.	There will be fewer, but more subtle bugs in an HOL program.	_____	_____
232.	HOL programs will be easier to maintain.	_____	_____

NOTE: "1" denotes strong disagreement or low relevance; "5" denotes strong agreement or high relevance.

	Agreement (1-5)	Relevance (1-5)
233. It will be more difficult to learn the HOL than to learn how to program a new machine in assembly language.	_____	_____
234. The skill level for HOL programming of communications systems will generally be lower than that needed for assembly language.	_____	_____
235. HOL communications software will provide machine independence for application software.	_____	_____
236. HOL programs will require more space than assembly programs.	_____	_____
237. HOL programs will (on an average) execute faster than routines written by the typical assembly language programmer.	_____	_____
238. HOL Programs can be brought on-line in less time than assembly language programs.	_____	_____
239. HOL source programs will have fewer initial bugs than assembly language programs.	_____	_____
240. HOL debugging will be more difficult since you will have to get down to assembly language anyhow, eventually.	_____	_____
241. It will be more difficult to integrate a HOL package than an assembly language package.	_____	_____
242. Large projects employ a data base specialist (e.g. one individual who is responsible for the design and/or maintenance of the data base).	_____	_____
243. The data base design is generally done by a few senior individuals.	_____	_____
244. A language that does not support communications I/O operations might as well not exist.	_____	_____
245. The variety of communications related I/O operations is so great that it would not be possible to construct a single approach that was not equivalent to assembly language programming.	_____	_____

NOTE: "1" denotes strong disagreement or low relevance; "5" denotes strong agreement or high relevance.

	Agreement (1-5)	Relevance (1-5)
246. Running time efficiency is the most important aspect of communications software.	_____	_____
247. Program space is generally more important than running time.	_____	_____
248. Tigh packing is employed frequently to conserve data base space even though this may increase processing time.	_____	_____
249. The single most common source of bugs are due to misunderstanding the specifications.	_____	_____
250. Keypunch and typographical errors are the source of most bugs.	_____	_____
251. Most bugs occur in calling sequences.	_____	_____
252. Most bugs result from a misinterpretation of the data base.	_____	_____
253. The documentation of programs in a HOL will be easier to understand.	_____	_____
254. A HOL source program will obscure much of what is happening; the documentation will have to be increased to compensate for this.	_____	_____
255. A HOL will promote better programming stype and will eliminate many dubious practices.	_____	_____
256. Reprehensible programming practices will neither be abetted nor averted through the use of a HOL.	_____	_____
257. It is planned to introduce a data type called "Status data type." A status type is used to define a set of properties of an element, which thereafter can be used for control purposes. For example, the declaration of a status type might have the following appearance: TYPE IS SECURITY = SECRET, UNCLASSIFIED, NATOCONF, CONFIDENTIAL In use, it would appear in the form of a statement such as: IF CLASSIFICATION IS SECRET THEN ..... ELSE ..... Having made such a declaration, the compiler would spot the bug such as: IF CLASSIFICATION IS URGENT THEN.....ELSE.....	_____	_____

NOTE: "1" denotes strong disagreement or low relevance; "5" denotes strong agreement or high relevance.

	Agreement (1-5)	Relevance (1-5)
258. Status types are encoded as consecutive integers in the order of their appearance in a declaration.	_____	_____
259. The programmer can specify the encoding of status types (e.g. SECRET = 1, UNCLASSIFIED = 17, NATOCONF = 3, etc.).	_____	_____
260. If no encoding is specified, the compiler will use consecutive integers as a default.	_____	_____
261. No status type requires more than 256 cases (e.g. MONDAY, TUESDAY, WEDNESDAY, ...).	_____	_____
262. The number of bits used to represent a status type should be left to the compiler to optimize.	_____	_____
263. The programmer must specify how many bits are to be used for representing each status type.	_____	_____
264. The ability to handle more than 256 cases for a given status type is important.	_____	_____
265. It is desirable to allow a given status value name to take on different meanings in different contexts, for example: TYPE IS JUNK = TOM, DICK, HARRY, BILL, ... TYPE IS HELP = TIM, TOM, JACK, JILL. If the two declarations are in different scopes, then the second TOM is permitted and will not be confused with instances of the first.	_____	_____
266. As in Question 265, but the distinction will be made in the call, for example: IF TYPE = JUNK.TOM THEN .....ELSE..... and IF TYPE = HELP.TOM THEN .....ELSE.....	_____	_____
267. Ordering related operations for status types are provided. Example: IF TYPE IS GREATER THAN SECRET THEN .....ELSE.....	_____	_____
268. In comparing two types, it is always assumed that ordering is possible. The ordering is that which results from the integer encoding of the values.	_____	_____

NOTE: "1" denotes strong disagreement or low relevance; "5" denotes strong agreement or high relevance.

	Agreement (1-5)	Relevance (1-5)
269. Types are assumed to be ordered by their encoding unless explicitly declared to be unordered.	_____	_____
270. Types are assumed to be unordered unless explicitly declared as ordered.	_____	_____
271. The compiler will prohibit the use of ordering operators (e.g. greater than) for types not declared as ordered.	_____	_____
272. Types can be declared as being cyclically ordered (e.g. MON, TUE, WED, THU, FRI, SAT, SUN, MON, ...).	_____	_____
273. Successor and predecessor operators for cyclic ordered types are available: e.g. SUCC(MON) = TUE ; PRED(MON) = SUN	_____	_____
274. If the programmer supplies the encoding (see Questions 259, 260) different elements can have synonymous codes. For example: DUPLEX = 2, HDX = 3, SATELLITE = 2, SIMPLEX = 4, TRUNK = 1, etc.	_____	_____
275. A number of bit field operations are being considered. When they appear in the context of an assignment statement, they result in a truth function value (TRUE/FALSE), when in the context of an IF statement, they provide a branching operation. Of particular interest are "greater than or equal" comparisons for bit strings or bit fields. For example A is greater than B if every bit of A is greater than or equal to every bit of B. If however, some bits of A are greater than corresponding bits of B, while some bits of B are greater than corresponding bits of A, then A and B are incompatible. Is a statement of the following form useful? IF A GRT B THEN .....ELSE.....	_____	_____
276. How about: IF A LTE B THEN .....ELSE.....	_____	_____
277. How about: IF A GRT B THEN .....IF NOT.....IF INCOMPARABLE THEN.....	_____	_____

NOTE: "1" denotes strong disagreement or low relevance; "5" denotes strong agreement or high relevance.