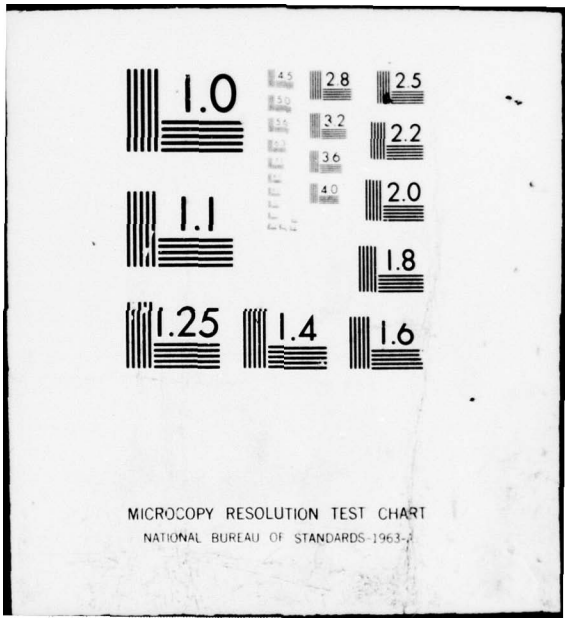
AD-A064 749 SOFTECH INC WALTHAM MASS F/G 9/2
HOL INVESTIGATION.(U)
JAN 79 J L FELTY, J R KELLY, J B GOODENOUGH F30602-76-C-0306
UNCLASSIFIED RADC-TR-78-269 NL

| OF |
AD
AD 64749

END
DATE
FILMED
4 -79
DDC


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

LEVEL #12

RADC-TR-78-269
Final Technical Report
January 1979

HOL INVESTIGATION

SofTech, Inc.

JAMES L. FELTY
JOHN R. KELLY
JOHN B. GOODENOUGH
LAWRENCE H. SHAFER

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

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.

RADC-TR-78-269 has been reviewed and is approved for publication.

APPROVED: Douglas White

DOUGLAS WHITE
Project Engineer

APPROVED: Wendell C. Baumann

WENDALL C. BAUMANN, Col, USAF
Chief, Information Sciences Division

FOR THE COMMANDER: John P. Huss

JOHN P. HUSS
Acting Chief, Plans Office

If your mailing 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-78-269	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE *(and Subtitle)*: HOL INVESTIGATION,	5. TYPE OF REPORT & PERIOD COVERED: Final Technical Report - June 1976 - September 1978
6. PERFORMING ORG. REPORT NUMBER: N/A	
7. AUTHOR(s): James L. Felty, John B. Goodenough, John R. Kelly, Lawrence H. Shafer	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: 33126F 20220401
11. CONTROLLING OFFICE NAME AND ADDRESS: Rome Air Development Center (ISIS) Griffiss AFB NY 13441	12. REPORT DATE: January 1979
	13. NUMBER OF PAGES: 50
14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)*: Same	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

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

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer: Douglas White (ISIS)

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

Compiler
Programming Language
JOVIAL
Communications
SEMANOL
Specification

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

The HOL investigation was a two year effort to determine communications high-order language requirements and define modifications to an existing language to be suitable for these requirements. Modifications to the JOVIAL $J_{73/I}$ programming language have been described as well as a SEMANOL formal specification of the modified language.

405 932

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

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

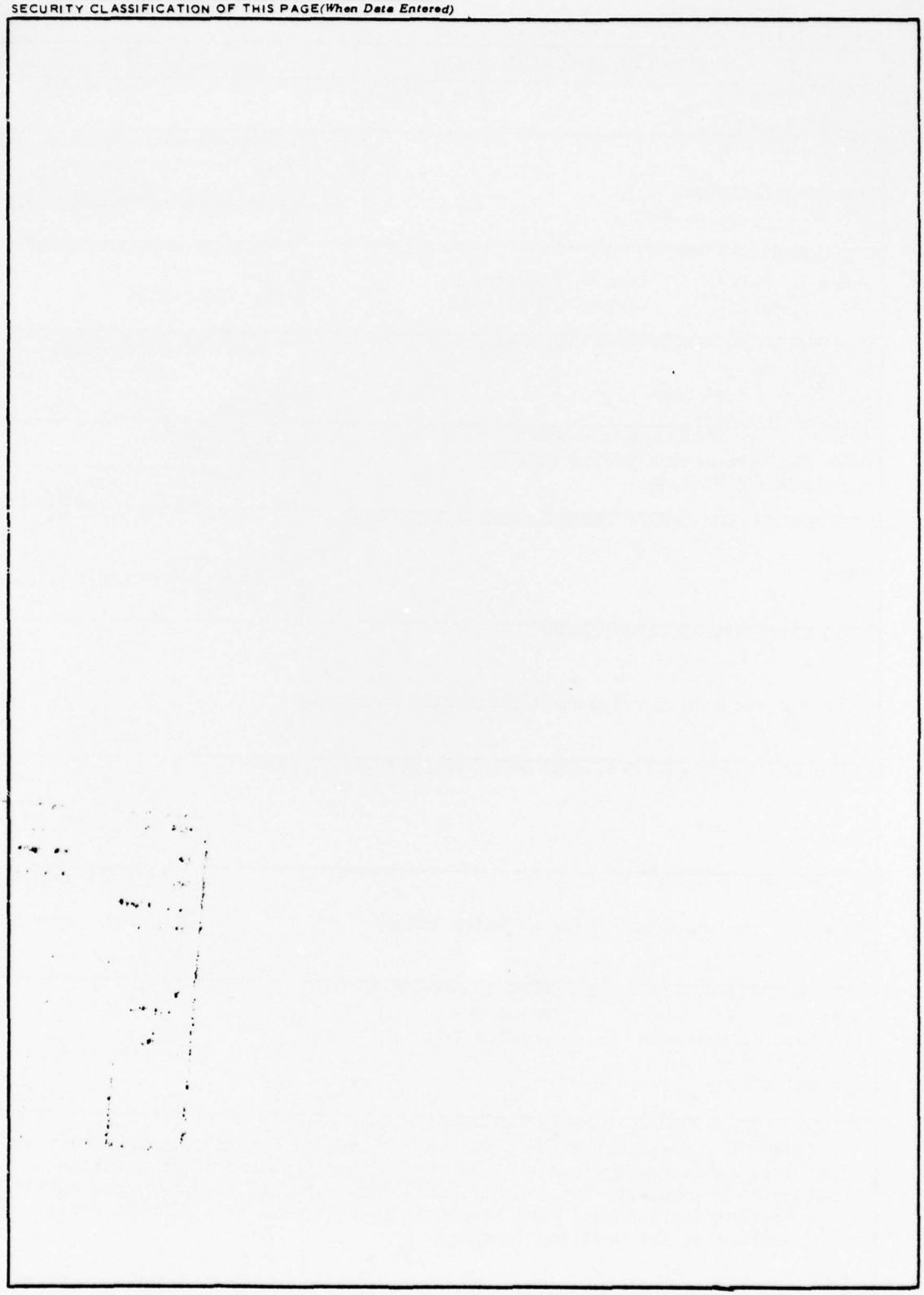
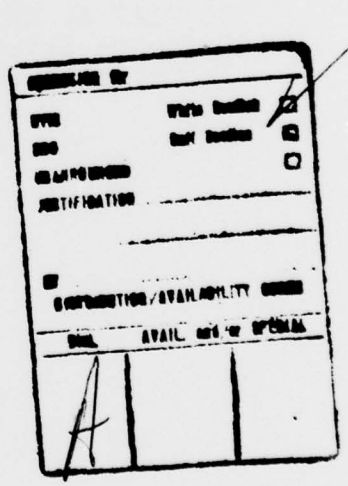
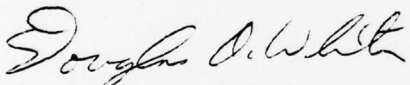


TABLE OF CONTENTS



EVALUATION

The HOL Investigation effort was undertaken to provide a programming language suitable for developing communications applications. This effort directly supports the RADC Technology Plan (V/3.1) by applying technologies which bring the programming environment closer to the system requirements and the human's logical thought process. This effort has resulted in the development of a language with features designed specifically to accommodate functions required by communications programming applications. The language also promotes the production of more readable and reliable software. These capabilities will aid in the overall Air Force goal of reducing software costs.



DOUGLAS A. WHITE
Software Sciences Section
Information Processing Branch

Section 1

INTRODUCTION

A two-year investigation of a high-order language (HOL) for use in communications applications has been completed. The study consisted of the following tasks:

- defining HOL capabilities needed in communications applications;
- evaluating five languages (CS4, JOVIAL J3, JOVIAL J73, PASCAL, and PL/I) to determine their suitability for programming communications applications;
- defining an appropriate HOL based on the results of the above evaluation;
- specifying a compiler for the HOL.

The first two tasks were completed in the first year of the study, and the results of that investigation are documented in Interim Report RADC-TR-77-341, Communications High-Order Language Investigation, Vols. I and II. That report concluded that none of the languages studied was suitable for communications applications without considerable modification. J73 was the most immediately usable of the five languages and was the one selected for use as the basis of a communications HOL. A list of proposed modifications to J73 was produced.

This report describes the second year of the investigation, which was devoted to defining the communications HOL and to specifying its compiler.

The task of defining the J73-based communications HOL consisted of two parts: (1) producing a reference manual for the language, which finalizes the syntax of the language and presents the semantics of each construct in English, and (2) producing a formal definition of the syntax and semantics of the language using the SEMANOL specification tool. The language has unofficially been called "J73 with extensions for communications," or for short, J73/C. The reference manual has been printed as SofTech report 3061-14.1, JOVIAL J73/C Communications HOL Reference Manual. The SEMANOL specification was partially completed and printed as SofTech report 3061-17, a SEMANOL (76) Specification of JOVIAL J73/C.

Section 2, below, describes the approach taken by SofTech in defining J73/C. Modification of the existing J73 (the MIL-STD-1589 version) was made in accordance with current Air Force policy requiring that all new versions of J73 be upward compatible with the existing language, i.e., all programs that are legal in current J73 must also be legal programs in J73/C. This requirement necessitated a few departures from the recommendations of the Interim Report. Nothing could be deleted from the language, and the task of adding the required new features in a uniform way was

complicated. Section 2.1 summarizes what was added to J73 to produce J73/C. The differences between J73/C and the modified J73 projected in the Interim Report are noted. Section 2.2 describes SofTech's approach toward producing a uniform language, upward compatible with the existing J73, while at the same time containing the features needed for communications applications.

Certain capabilities needed in communications applications are implementation dependent and/or impose strict efficiency requirements. For these reasons, they were omitted from the language presented in the reference manual. These include I/O operations, real-time processing capabilities, and certain string and character manipulation capabilities. The intent is that these will be provided as needed by implementation-dependent library routines. These routines are described in Section 2.3.

Section 3 describes the SEMANOL specification of J73/C. The approach and structure of the SEMANOL specification are described in Sections 3.1-3. Section 3.4 contains an evaluation of the SEMANOL specification technique, based on SofTech's experience in applying it to J73/C.

The compiler specification, the fourth task in the Communications HOL investigation, was presented in SofTech report 3061-15, JOVIAL J73/C Communications HOL Compiler Specification. In addition to presenting the architecture and functional characteristics of the various portions of the compiler, that report specifies error messages, documentation standards, and quality assurance provisions.

Section 4.1 describes the rationale for the compiler design presented in 3061-15, and summarizes its overall structure. The modular design is specified in rigorous detail, including all of the data interfaces and intermediate files.

Computer programs were written in J73/C to test the compiler's ability to translate the features of J73/C that were added to the existing language. Compiler verification programs already existed for J73 in the form of the JOVIAL Compiler Validation System (JCVS). New tests for the added communications features were written in the JCVS style and format, and have been merged with the existing JCVS tests to form a complete set of tests for the communications language. Section 4.2 describes the nature of those programs. The programs themselves reside on the HIS 6180 MULTICS system at RADC.

The appendix to this report is a list of errata that were discovered since the J73/C reference manual was printed.

Section 2

THE JOVIAL J73 COMMUNICATIONS HOL

2.1 Additions to Existing J73

The modifications needed to adapt J73 for communications applications were described and justified in the Interim Report. In this Section, we will summarize how the required additional features were integrated into the J73 language. A complete specification of the J73 communications HOL, with the syntax in BNF and the semantics explained in English, is presented in the JOVIAL J73/C Communications HOL Reference Manual (April 7, 1978). Various errata and additions to that document are given later in Appendix A of this report.

The additions to J73 that were recommended in the Interim Report were of three kinds: (1) "functional " additions, i.e., new language capabilities that are needed by communications programmers; (2) application-independent additions that reduce programming errors, foster readable code, etc.; (3) additions that make for more efficient object code.

The recommended functional additions to J73 are presented below with a discussion of how the recommendations were implemented. A key consideration in making these additions was Air Force policy that all changes be upward compatible with JOVIAL J73 as defined in MIL-STD-1589. Therefore, the new features were designed to be consistent with the syntactic and semantic conventions of J73 insofar as possible. Where we encountered conflicts between current language design principles and the existing language, we either resolved them in favor of the existing language or in the manner discussed later in Section 2.2.

The "functional" changes recommended in the Interim Report are given below. Each recommendation is followed by a brief discussion of how the recommendation was reflected in the J73/C design.

- Provide the ability to specify classes of table entries with limited access characteristics; provide the ability to specify programmer defined data types.

This recommendation was implemented by providing an "encapsulated" type definition capability. Such definitions have the form of a table definition, except they include procedure definitions as well as table component definitions. Access to all components of the definition is automatically provided to routines declared within the type definition, but outside the type definition, access is permitted only to those components whose names are explicitly exported from the definition. Consequently unexported table fields can be read or modified only by invoking a routine belonging to the encapsulation. In essence, this capability guards against unauthorized access to sensitive fields. As a simple example, consider the following definition of the STACK data type.

```
CONSTANT ITEM STACK'SIZE U = 100;
TYPE STACKTYPE S;
TYPE STACK RECORD;
BEGIN
      EXPORTS (PUSH, POP, READONLY TOP);
      ITEM TOP U = 0;  "number of items in STACK'REP"
      TABLE STACK'REP  [1:STACK'SIZE]  STACKTYPE;

      PROC PUSH (S1, I1);
            BEGIN
                ITEM I1 STACKTYPE;
                TABLE S1 STACK;
                S1.TOP = S1.TOP + 1;
                S1.STACK'REP [S1.TOP] = I1;
            END

      PROC POP (S1) STACKTYPE;
            BEGIN
                TABLE S1 STACK;
                POP = S1.STACK'REP [S1.TOP] ;
                S1.TOP = S1.TOP - 1;
            END
END
```

This example makes the TOP field of STACK type objects accessible only for reading outside the type definition; however, the PUSH and POP functions can modify TOP since they are local to the definition. The contents of STACK'REP are not accessible outside the STACK type definition (since STACK'REP is not exported), and so the only way these contents can be accessed is by calling the PUSH and POP routines. The example also illustrates the use of constant declarations and simple type declarations, e.g., STACKTYPE is equivalent to a signed integer type; it is only necessary to change STACKTYPE's declaration to obtain a stack whose elements are some type other than integers. It should be noted that the syntactic conventions of encapsulated type definitions follow the style of similar declarations in J73, e.g., the components of the declaration are enclosed in BEGIN-END brackets, a semicolon terminates that part of the declaration preceding the BEGIN, etc. In this way, the unity of the J73 language has been maintained.

The encapsulation capability includes the ability to define actions invoked when assigning to a variable of the defined type, actions for comparing two variables for equality, actions for allocating and deallocating heap storage (see later discussion), and actions for granting access to objects shared among concurrent processes (see later discussion). For example, if it was desired to provide a routine for comparing two stacks for equality, we would add the following definitions to the encapsulation.

```
PROC EQUAL (S1, S2) L;
     BEGIN
            TABLE S1 STACK;
            TABLE S2 STACK;
            ITEM RESULT L = TRUE;
            ITEM INDEX  1:STACK'SIZE;
            IF S1.TOP<>S2.TOP;
               RESULT = FALSE;
            ELSE
               FOR INDEX: 1 TO S1.TOP;
                   IF S1.STACK'REP[INDEX]<>S2.STACK'REP[INDEX];
                      BEGIN
                         RESULT = FALSE;
                         EXIT;   "exits FOR loop"
                      END
            EQUAL = RESULT;
      END
COMPARE EQUAL;
```

The last statement in this example says that when two stacks are compared using the "=" operator, the EQUAL function is to be invoked; the result returned by this function defines what equality means for this data type.

- Provide typed pointers to reference objects safely and efficiently.

The pointer type was introduced explicitly into the language to help reduce errors involving the use of pointers. Only pointers to aggregates (i.e., tables) are permitted. The J73 convention of using single letters for built-in types is followed here by using P to designate the pointer type. A pointer is declared:

```
ITEM PTR P aggregate:type:name ;
```

PTR can subsequently be used only to access tables and table components of the designated type. For example, if the following declaration exists:

```
ITEM P1 P STACK;
```

then @P1.TOP would access the number of items stored in the stack P1 points to. An expression of the form @P1.XX or @P1.STACK'REP would be illegal, since objects of type STACK have no XX or STACK'REP field that is exported.

- Provide programmer control over allocation strategies, to permit flexible and efficient storage management.

The ability to define special purpose storage management routines is important to permit efficient allocation and deallocation of data accessed through pointers. Separate storage allocation areas, called

zones, can be established with different storage management strategies. For example, one strategy might be tuned to allocate only blocks of a single fixed size. Another zone might be prepared to allocate blocks of three or four different sizes. It is also possible to create a storage area in which data blocks are allocated from the ends toward the middle. In general, a wide variety of suitable allocation strategies can be specified.

The following example indicates how a zone set up to allocate data of a single type could be defined:

```
ZONE ALLOC'TYPE'ZONE RECORD;
     BEGIN
          TABLE ZONE'SPACE[0:99]ALLOC'TYPE ;
                "reserves space for 100 objects"
          PROC ALLOCATE (ALLOC'SIZE) P ALLOC'TYPE;
                "finds unused space in ZONE'SPACE"

                ...
                END
          PROC DEALLOCATE (PTR, ALLOC'SIZE);
                "frees space identified by PTR"

                ...
                END
          ALLOC ALLOCATE;
          DEALLOC DEALLOCATE;
     END
```

The ALLOCATE and FREE procedures are invoked when NEW(ALLOC'TYPE) and FREE(PTR'TO'ALLOC'TYPE) are executed, assuming that the following declarations are in effect:

```
TYPE ALLOC'TYPE IN ALLOC'TYPE'ZONE...;
     "definition of ALLOC'TYPE indicating objects of this
     type are allocated in the zone called ALLOC'TYPE'ZONE
ITEM PTR'TO'ALLOC'TYPE P ALLOC'TYPE;
```

To allocate space in the special zone, the programmer would execute PTR'TO'ALLOC'TYPE = NEW(ALLOC'TYPE). To free an allocated object, he would execute FREE(PTR'TO'ALLOC'TYPE). These statements would invoke the special ALLOCATE and DEALLOCATE procedures defined for ALLOC'TYPE'ZONE. These allocation procedures can be particularly efficient, since only one size object is being invoked.

- Provide table entry declaration capabilities to support nesting of tables, tight packing, packed arrays, and variant fields.

The nesting capability was achieved by relaxing the restrictions on what a J73 TABLE element can consist of. In J73/C, table components can be declared to be TABLE (i.e., records and arrays); they are no longer restricted to simple items. In addition, although J73 TABLES must be indexed, the presence of an index is optional in J73/C, thus permitting the declaration of an unindexed table, i.e., a table with only one entry. The result is a table declaration capability whose syntax closely resembles the existing table syntax, yet has the increased versatility needed in communications programming.

Once tables can be nested within tables, the question arises of whether or not all table component names must be distinct. J73/C requires uniqueness of names only within a given level of nesting; disambiguation is permitted by qualifying the nested name by the name(s) of the outer table(s), using a "." between each of the names, as in PASCAL or PL/I.

Components within a table entry may be tightly packed -- i.e., allocated with no "filler" bits between items -- by using the allocation specifier T. Introducing an additional single-letter allocation specifier in the language is consistent with the existing use of single letters for this purpose (namely, N,M, and D) and reduces the need for programmers to resort to specified tables.

The final syntactic addition to table syntax is that of providing the ability to declare "variant" fields in tables. Variant fields allow run-time selection of one of several possible table component declarations or groups of declarations, based on the current value of a specially-designated "tag" field declared earlier in the same table. The syntax of variant fields parallels the syntax of the J73 SWITCH statement (with table components instead of statements), thereby maintaining uniformity with other parts of the language. For example, consider the following declaration:

```
TYPE DEVICE STATUS (V(DISPLAY),V(DISK),V(TAPE));
TYPE DEVICE'CONTROL TABLE
      BEGIN
      ITEM DEV DEVICE;  "tag field"
      SWITCH DEV
            BEGIN
            [V(DISPLAY)] BEGIN ... END
            [V(TAPE),V(DISK)] BEGIN ... END
            END
      END
```

DEV is the tag field. Its value indicates whether a DISPLAY table structure is present or a TAPE/DISK structure.

- Provide procedure variables, including the use of procedures as table components.

This capability was provided by permitting procedures and functions to be types whose values can be stored in variables and table components. For example,

```
ITEM ACCESS PROC (S);
```

declares the variable ACCESS to be a variable whose values are procedures taking a single signed integer input parameter. The declaration:

```
TYPE P'TYPE PROC (S);
```

declares a type name standing for PROC(S). Hence, we could also have declared the ACCESS variable with:

```
ITEM ACCESS P'TYPE;
```

The P'TYPE or PROC(S) form can also be used in declaring table components.

This extension of the language to include procedure variables is quite straightforward. However, the existence of procedure type names introduces an additional problem if uniform language capabilities are to be provided. The problem stems from the fact that in J73, procedure names can be declared using the form:

```
PROC P'NAME (PARAM) BEGIN ITEM PARAM S; END;
```

This declares P'NAME to be a procedure taking a single signed integer argument. The body of this procedure is defined elsewhere. Extending this ability to make use of procedure type names suggests providing a form:

```
(1)  PROC P'NAME P'TYPE;
```

where P'TYPE is declared as before. However, this PROC declaration can also be used to declare a parameterless function returning a value of type P'TYPE. There is no semantic ambiguity, since functions cannot return routines as values, but it is not psychologically acceptable to have the significance of P'TYPE (i.e., whether it defines P'NAME's return type or P'NAME's type) depend on the meaning of P'TYPE. Consequently, instead of introducing the declaration form illustrated by (1), we provide the form:

```
PROCNAME P'NAME P'TYPE;
```

for use when P'TYPE is a procedure type, and limit the form (1) to

implying P'NAME is a function returning a value of type P'TYPE. (Hence, (1) is illegal if P'TYPE is a procedure type name.) For uniformity with the rules involving the use of type names elsewhere in the language, the definition of P'TYPE can also be used directly, e.g.,

```
PROCNAME P'NAME PROC (S);
```

- Provide procedure attributes to allow specification of properties such as reentrancy, recursion, interrupt handling, task initiaion, and exception handling.

The specification of recursive or reentrant procedures is indicated by preceding the procedure declaration with the keywords RECURSIVE or REENTRANT. Procedures that are used for interrupt handling are indicated by following the parameter list with the sequence:

```
INTERRUPT compile:time:character:formula
```

The significance of the character:formula is implementation dependent; it causes the linker to place the interrupt routine so it is invoked by the target machine interrupt hardware described by the character string. For example, the routine that responds to an interrupt from a card reader might be declared:

```
PROC HANDLE'CARD INTERRUPT 'CHN 5';...
```

Interrupts received on the target machine's "channel 5" will initiate execution of the HANDLE'CARD routine.

No mechanism for task initiation was provided since we felt that efficiency considerations precluded defining this concept in the language (see Section 2.3.2).

The exception handling procedures that have been provided are very limited and are intended only to help terminate procedures that have encountered a fatal error. It is anticipated that exception handlers will be defined at a global level (i.e., in the COMPOOL). When an exception is raised, the handler is invoked directly -- there is no search up the invocation chain to find a handler defined in some dynamically enclosing scope. A handler returns by specifying termination of some procedure whose invocation led to the exception. The handler body must determine which procedure is to be terminated.

- Provide the ability to specify transmission mechanisms for parameters, including whether by value, reference, or result and which target machine register is used.

The keywords BYREF, BYVAL, or BYRES can be included before any formal parameter specification to indicate passage by reference, value, or result, respectively. BYVAL may be used only for input parameters, and BYRES, only for output parameters. If the parameter transmission mode is not explicitly specified, the normal J73 conventions are followed, i.e., BYVAL for input parameters that are not tables or blocks, BYREF for table and block input parameters, and BYRES for output parameters.

The register used to pass a particular parameter can be explicitly specified, e.g., the declaration

```
PROC P1 (BYVAL PARAM REGISTER R1);
```

specifies that P1 takes a single input parameter passed by value in register R1 (register names are, of course, implementation dependent). The ability to specify register passing conventions permits special purpose procedure linking conventions to be established.

- Provide the ability to specify inline calls of procedures and to specify procedure bodies in assembly code.

An INLINE procedure is specified by preceding its declaration with the keyword INLINE. Similarly, an assembly code procedure is specified by preceding its declaration with the keyword CODE. The semantics of INLINE procedures are the same as the semantics of closed procedures except that BYREF parameters are substituted for the formal parameters in the expanded body.

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

If a variable is declared to be the status type ALPHA, then its value can be used like the values of any status type. In particular, usage in case indexing and loop control is permitted. If it is desired to use a variable declared as a character string of length one in case indexing, then it must be converted to a status type, e.g., ALPHA(V1) treats the value of V1 as the status value for a single character. Similarly, integer values in loops can be converted to status values representing a single character.

- Provide an absolute allocation attribute for procedures.

The ability to specify an absolute target machine location is provided by the ABSOLUTE option:

```
PROC P1 (...) ABSOLUTE 5574;
```

indicating that P1 is to be located at decimal address 5574. An

octal address could be specified as INT(3B'5574') (this converts the octal bitstring 5574 to an unsigned decimal integer).

- Provide the ability to use COMPOOL directives in COMPOOL modules and to collect type definitions, shared variable declarations and related procedure declarations into COMPOOLs.

The J73 definition of a COMPOOL was extended to permit declaration of all the J73/C types in COMPOOLs and to permit COMPOOLS to contain references to other COMPOOLs.

- Provide a shared variable attribute and access blocks to support exclusive access to shared resources and to implement critical regions.

To ensure exclusive access to variables shared among concurrently executing processes a variable is declared to be SHARED and accessed within an UPDATE block. For example, if V1 and V2 are declared to be SHARED variables, then:

```
UPDATE V1, V2;
   ...
END
```

provides a block within which exclusive access to V1 and V2 is guaranteed. The programmer can provide special purpose locking and unlocking procedures that are invoked on entering and exiting update blocks. Such procedures are defined by specifying the type of V1 and V2 to be an encapsulated type, e.g.,

```
TYPE V'TYPE RECORD; BEGIN
     ...
     PROC ENTRY'PROC; BEGIN ... END
     PROC EXIT'PROC; BEGIN ... END
     ENTRY ENTRY'PROC;
     EXIT EXIT'PROC;
END
TABLE V1 SHARED V'TYPE;
TABLE V2 SHARED V'TYPE;
```

Now when the update block for V1 and V2 is entered, ENTRY'PROC will be invoked, and when it is exited, EXIT'PROC will be invoked. Thus efficient special purpose locking and unlocking procedures can be defined.

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

A formal parameter may be declared with asterisks in place of all its bounds specifications, indicating that the bounds of the actual parameter are to be used.

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

This capability was not provided since the impact on run-time support for the language did not seem commensurate with its usefulness.

- Provide expanded 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.

Of these capabilities, only the last (specifying the length of a dynamically allocated character string) was not implemented because additional analysis convinced us that the added capability was not worth the cost in increased language and implementation complexity.

The ability to specify that character strings are represented in a particular character set is specified with the !CHARACTERS directive. Only one such directive is permitted in a single complete program, although more than one character set may be supported in a given implementation.

To represent special characters there is a built-in status type ALPHA that is defined as a sequence of status literals whose representation is that associated with the characters in the set indicated by the !CHARACTERS directive. Each of the literals in ALPHA is of the form V(x), where x is either a printing character of the character set or an implementation-defined mnemonic spelling corresponding to a non-printing character. For example, the default ASCII values of the ALPHA constants include V(BEL) for the BEL character, V(A) for upper case A, V($A) for lower case A, etc.

A formal character string parameter can be declared with an asterisk to indicate that it is to assume the length of the corresponding actual parameter. (An asterisk is used similarly to indicate that the bounds of formal table parameters are to assume the values of the bounds of corresponding actual parameters.) The length of the actual parameter can be determined by applying the BYTESIZE function to the parameter.

The following features were recommended to improve program readability, modularity, and reliability by restricting the need for GOTO statements, providing additional mnemonic capabilities such as constant names, etc.

- constant names for all built-in types

This capability was provided in a simple and uniform way by permitting the keyword CONSTANT to precede any item or table declaration. This means that the value of the designated item or table cannot change at run-time and must be specified via a preset. In addition, the constant name may be used in any way that a literal of that type may be used, e.g., in presets or in other contexts requiring compile-time values, e.g., formulas evaluated at compile time, or in specifying the size of a signed integer type:

```
CONSTANT HALF'WORD U = 15;
ITEM V1 S HALF'WORD;  "equivalent to ITEM V1 S 15"
```

- Boolean data type

The Boolean data type was introduced so the rules for evaluating Boolean formulas could specify short-circuit evaluation. The specification of short-circuit evaluation cannot be considered just an optimization. For example, the loop

```
WHILE PTR<>NULL AND @PTR.VALUE = 0; BEGIN ... END
```

cannot be expressed so easily if short-circuit evaluation is not guaranteed, since @PTR.VALUE is ill-defined if PTR is NULL.

- Range attribute for numeric data

An optional specification of range attributes is permitted in place of the normal J73 numeric type specifications, e.g.,

```
ITEM V1 0:100;
```

indicates that V1 is an integer variable with range 0 through 100.

- The ability to specify a variable read-only in a scope

The read-only declaration, e.g., READONLY V1; indicates that the variable V1 cannot be assigned to in the scope of the read-only declaration. A read-only declaration in a block declaration or in the definition scope of a variable declares a name with a constant

value at run time. Such names differ from compile-time constants in that they are not used in compile-time formula evaluation. The programmer or compiler can often make use of these items, however, to ensure that literals used frequently in enclosed scopes have only one storage location assigned to them. The ability to provide programmer-defined pools of literals can permit more space-efficient object code to be generated on some minicomputers.

- Ability to terminate by a loop exiting statement.

This capability is provided by a loop exiting statement. If the exit statement references a label, the loop named by that label is exited. If no label is referenced, the innermost loop containing the exit statement is exited.

- A complete set of built-in machine dependent parameters

The following constant names specifying machine-dependent values have been defined: BITSINBYTE, BITSINWORD, LOCSINWORD, BYTESINWORD, ADDRESSSIZE, MAXBITSIZE, MAXCHARSIZE, MAXINTEGER, MININTEGER, MAXFLOAT, MINFLOAT, MAXFLOATPREC, MAXSTATUS, MINSTATUS. These values can be used to assist in parameterizing programs for use on different target computers, or for explicitly indicating machine dependencies, e.g., the range of a signed integer is MININTEGER:MAXINTEGER

- Ability to refer to all the declared attributes of a variable of any built-in type

The size of a variable can be determined using the BITSIZE, BYTESIZE, and WORDSIZE functions. The dimensions of a table can be determined with the LBOUND and UBOUND functions. The minimum and maximum values of a numeric variables range can be determined with the MIN and MAX functions.

- Constant evaluation for built-in operations on all built-in types

- Names for all hardware exception conditions and for optional software conditions such as range violations, length differences, and conversion errors.

Constant expression evaluation has been provided for all built-in data types except for expressions that dereference variables. The only predefined exception condition names are SIZE'ERROR and RANGE'ERROR. These exceptions are raised on assignment. If non-padding bits are truncated on assignment, SIZE'ERROR is raised. If

the value being assigned lies outside the range of the target variable, RANGE'ERROR is raised. Range and size checks are enabled or disabled for a given scope by the !ENABLE and !DISABLE directives. Other condition names may be defined for a given implementation depending on what conditions the hardware checks for.

- A uniform notation for enabling or disabling exceptions at the statement and program module level.

To minimize the number of extensions to the language, only a minimal exception handling capability was provided. The only exceptions that can be enabled or disabled are those that are language or implementation defined, and such exceptions can be enabled or disabled only for entire scopes, not for simple statements. This capability was considered to be minimally sufficient.

The features recommended for increased efficiency that have not already been discussed are:

- A TO form of loop that implies that value of the loop control variable is read-only and is not used outside the loop

- Conditional statements whose controlling predicate is a constant Boolean expression will initiate conditional compilation of the THEN or ELSE branch.

Both these capabilities have been provided. In addition, constant expression evaluation, parameter transmission specification, and INLINE procedures contribute to object code efficiency.

2.2 Upward Compatibility with Existing J73

The interim report recommended that numerous restrictions be made to J73 and that certain existing J73 capabilities be deleted from the J73 communications HOL. Most of these changes were recommended in the interest of increasing the reliability of the language. Some of the recommended restrictions were as follows (asterisks denote those restrictions supported in J73/C):

* 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)

* a constant that 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, cannot be used in those contexts.

* 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 overlap of actual parameters passed by reference 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 after loop entry

• 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

The J73 communications HOL was designed in accordance with the current Air Force policy requiring that all language extensions be upwardly compatible with the base language. This requirement implies that, properly speaking, nothing in the existing language can be deleted. Restricting the class of legal programs is difficult, because all programs legal under existing J73 must also be legal in the new language.

The solution offered in the communications language is to provide a number of "restrictions directives" that programmers may use to enable or disable certain capabilities. For example, the directive

```
!DISABLE IMPLICIT'CONVERSIONS;
```

causes J73/C's strict type matching rules to be enforced. Once disabled by the above directive, implicit conversions are effectively removed from J73/C. They may be re-introduced by an "enable directive", viz.,

```
!ENABLE IMPLICIT'CONVERSIONS;
```

which reverses the effect of the preceding disable directive. The default setting for implicit conversions is "enabled"; hence, upward compatibility is maintained.

For the sake of simplicity, the number of restrictions directives introduced into the language was kept small. Hence, of the list of language restrictions and deletions recommended for reliability in the Interim Report, only those deemed essential were introduced. In addition to IMPLICIT'CONVERSIONS, the following restrictions directives were introduced:

UNRESTRICTED'GO'TO -- if enabled, control transfers out of the current scope and update statements or into loops, switches, and update statements are permitted. Transfers to label parameters are considered transfers out of the current scope (UNRESTRICTED'GO'TO is enabled for compatibility with J73.)

CASE'FALL'THROUGH -- if enabled, permits specifying that when control reaches the end of a switch alternative, control can continue on to the next alternative instead of passing to the statement after the switch statement. (CASE'FALL'THROUGH is enabled for compatibility with J73.)

CASE'DEFAULT'OPTIONAL -- if enabled, specification of a DEFAULT alternative in SWITCH statements is never required; moreover, default specification of indices associated with switch alternatives is permitted. If CASE'DEFAULT'OPTIONAL is disabled, a DEFAULT alternative may be omitted only if each possible value of the selector is associated with a switch alternative either explicitly or by inclusion in a closed range of values. Moreover, the default specification of an index value is prohibited. (CASE'DEFAULT'OPTIONAL is enabled for compatibility with J73.)

RANGE'CHECK -- if enabled, the RANGE'ERROR exception will be raised if the value:

- of the source formula in an assignment statement, or
- of an actual parameter passed by value, or
- of a formal parameter passed by result, or
- of an index in a table element reference

is not in the range of values associated with

- the target variable's range,
- the corresponding formal parameter's range,
- the corresponding actual parameter's range, or
- the corresponding dimension range,

respectively. Moreover, assignment of an out of range preset value is considered a compile time error. If RANGE'CHECK is disabled, no checks are made for the presence of a RANGE'ERROR and if such an error is present, the effect is undefined. (RANGE'CHECK is disabled for compatibility with J73.)

SIZE'CHECK -- if enabled, the SIZE'ERROR exception will be raised if the number of bits in the:

- source formula of an assignment statement, or
- an actual parameter passed by value, or
- a formal parameter passed by result

exceeds the number of bits associated with the

- target variable,
- the corresponding formal parameter, or
- the corresponding actual parameter,

respectively, and nonpadding bits or blanks of the assigned value must be truncated. Moreover, truncation of nonpadding elements in a preset assignment is considered a compile-time error. If SIZE'CHECK is disabled, the value is truncated without a check being made to see if only padding components are lost. (SIZE'CHECK is disabled for compatibility with J73.)

COMPOOL'REFS'ONLY -- if enabled, REF specifications and procedures without statements are prohibited. In addition, DEF PROCNAME declarations are permitted only in COMPOOL modules. (COMPOOL'REFS'ONLY is disabled for compatibility with J73.)

COLLECTED'DECLARATIONS -- if enabled, requires that all declarations except for procedure and function declarations appear at the start of their scope. (COLLECTED'DECLARATIONS is disabled for compatibility with J73.)

An additional restrictions directive was added in the interest of increasing the efficiency of compiled code:

LOOP'REEVALUATION -- if enabled, the increment:phrase of a loop is evaluated before every loop iteration. If disabled, the increment:phrase constituents are evaluated only once, prior to the first execution of the loop, and these values are used for every loop iteration. (LOOP'REEVALUATION is enabled for compatibility with J73.)

Restrictions directives were used to solve one other compatibility problem. Some of the added features discussed earlier in Section 2.1 provide capabilities that overlap or parallel some of the capabilities in J73. For example, based storage and integer pointers can be used in J73 to obtain the effect of typed pointers in J73/C, but without the protection of type-checking. Based storage integer pointers cannot be removed without sacrificing upward compatibility, so a restrictions directive has been added to make the old forms and semantics unavailable for communications programming.

BASED'STORAGE -- if enabled, the use of based allocation: specifiers is prohibited in declarations, and the use of integers as pointers is prohibited in variable references. If BASED'STORAGE is disabled, only typed pointers can be used. (BASED'STORAGE is enabled for compatibility with J73.)

A similar parallelism occurs in the definition of status types. There are two ways of defining status values:

in J73:

```
STATUS NETWORKS V(CBS), V(NBC), V(ABC), V(PBS);

ITEM NET1 U NETWORKS = V(CBS);
```

using new facilities:

```
TYPE NETWORKS STATUS (V(CBS), V(NBC), V(ABC), V(PBS));

ITEM NET1 NETWORKS = V(CBS);
```

The first status type is just a form of integer declaration, while the second is a true status value. In the first case, NET1 can be assigned any unsigned integer value fitting in the number of bits associated with NET1 and V(CBS), for example, is considered just a special form of integer constant. In the second case, NET1 can only be assigned the specified status literal values; attempting to assign an integer value would be an error and V(CBS) is not considered an integer value. To ensure only the new status capabilities are used in communications programming, the following restrictions directive is provided:

INTEGER'STATUS -- if enabled, permits the declaration and use of integer status lists as defined in J73. If disabled, only the new status types can be used. (INTEGER'STATUS is enabled for compatibility with J73.)

2.3 Features to be Provided by Extension

Some language capabilities needed for communications applications were intentionally left out of the J73/C language as specified in the Reference Manual. The intention was that these features should be added as extensions at a later date, at which time they could be tailored efficiently toward a specific implementation and/or application. These features fall into three categories:

- I/O capabilities
- Process control capabilities
- Additional character-handling capabilities

The principal reason these were omitted from the base specification is one of efficiency. I/O and process control entail interfaces with an operating system, and to specify a general form for these features would restrict their implementation in ways that would be efficient under some systems and machines and inefficient under others. Furthermore, the capabilities needed, particularly

for I/O, are highly application-dependent.

What the J73/C specification does provide, however, is language features to support application-dependent, low-level extensions for those capabilities. The following subsections discuss how those extensions might be realizable for each of the three above mentioned kinds of capabilities.

2.3.1 Recommended I/O Extensions

Extensions for I/O may be realized via either HOL code or assembly code, and can be done by either inline or closed procedures. Assembly code may be inserted in a J73/C procedure by means of the keyword DIRECT. Inline expansion of a procedure may be forced by specifying a procedure attribute INLINE. For high-level I/O, a standard library in the form of closed procedures through the logical I/O level, is highly recommended.

The following paragraphs describe the levels of I/O and their possible implementation. The related process control methods that field I/O interrupts are addressed in the next subsection.

High level I/O would consist of such procedures as

```
     PRINT'STRING('-----'),
     PRINT'INT(integer),
     SPACE(n)          (to print blanks),
and  END'LINE          (to terminate a line).
```

High level I/O is implemented using low-level logical I/O (see below).

Formatted I/O would use a procedure such as

```
     FORMATTED'PRINT(format, value1, value2,...).
```

The formatted print routine would have type codes in a tabular format for each type of value expected in the remainder of the argument list. Such type codes would be used to select appropriate high-level print routines to output the argument values.

Since J73/C does not allow passing of type information, high-level routines to decompose each structure (table) to be output would also have to be programmed. Additional information in the format could include field sizes and type of literal to be printed, as well as various output device controls (tab, new line, column no.). The FORMATTED'PRINT routine would be implemented using the high-level procedures described above.

Formats could be hand-coded constant tables. A FORMAT data type to facilitate preprocessing of formats encoded as strings could be added to alter versions of the language, or supported separately by a preprocessor.

Logical I/O includes such procedures as

```
FILE = OPEN(DEVICE)
PUT'STRING(FILE,STRING)
```

and includes the use of returned conditions codes or software exceptions to interface with physical I/O buffering routines. Logical I/O is implemented using physical I/O.

Physical I/O involves transmission of data to a device and often requires synchronization with other users of the device and with the device timing. Typical operations are:

```
WRITE'CHAR(device,character)
WRITE'WORD(device,value)
WRITE'BLOCK(devide,buffer,length)
```

These occur often in communications programming and basically issue the device I/O commands. Device condition testing is by device-specific testing functions, interrupt procedures, and exception-handling mechanisms. Synchronizing with other users of the device and for device timing can be achieved using UPDATE blocks, or by lower-level synchronization methods implemented as part of an executive extension library.

Writing a J73/C I/O procedure containing an arbitrary number of arguments is made possible by the J73/C procedure attribute VARIADIC.

2.3.2 Recommended Process Control Extensions

Examples of the definition of synchronizing methods using UPDATE blocks follow. Two types of waiting are described, busy waiting (which puts each process trying to access a device into a tight loop until a previous process has completed execution of the critical region for device access, consistent with a dedicated multiprocessor situation) and a FIFO queueing of access, which can be combined with busy waiting or single-processor multi-tasking, and insures that access occurs in the order of requests.

Busy waiting may be accomplished with an encapsulated type, using a single flag accessible only to all processes trying to enter a critical region. A primitive operation, TEST'AND'SET, is

assumed to provide testing and setting a bit in a single instruction; the value of the TEST'AND'SET operation is the read value of the bit.

```
TYPE BUSY'WAIT RECORD;
      BEGIN
      ITEM BUSY L = FALSE;
      PROC WAIT;
            BEGIN
            WHILE TEST'AND'SET(BUSY);
                BEGIN
                END
             END
      PROC SIGNAL;
            BEGIN
            BUSY=FALSE;
            END
      ENTRY WAIT;
      EXIT SIGNAL;
      END
```

Then a SHARED variable declared with this type, i.e.,

```
TABLE variable SHARED BUSY'WAIT;
```

could be used with an UPDATE block of the form,

```
UPDATE variable
code for device access
END
```

to insure non-interfering access. The WAIT procedure is called at the entry of the UPDATE block, and locks the process attempting to execute the block until a process already executing is completed. Completion is indicated by the SIGNAL procedure, which by disabling the BUSY flag allows the first process to read the disabled value to continue into the critical region.

The second example supports access in the order of requests to a device, and is usable for a single processor. It uses primitives to SUSPEND and RESUME a task; these primitives may interact with a scheduler or do busy waiting. The example assumes a data structure,

```
TYPE Q'BLOCK TABLE;
      BEGIN
      ITEM NEXT P Q'BLOCK;
      ITEM TASKID PROCESS'ID;
      END
```

to create FIFO queues. Again, it is defined as an encapsulated type with ENTRY and EXIT functions.

```
TYPE FIFO'ACCESS RECORD;
      BEGIN
      ITEM BUSY L = FALSE ;
      ITEM HEAD P Q'BLOCK ;
      ITEM TAIL P Q'BLOCK ;
      PROC WAIT;
            BEGIN
            ITEM CURRENT'BLOCK IN P Q'BLOCK ;
            IF HEAD = NULL AND NOT BUSY ;
                        BUSY=TRUE ;
            ELSE BEGIN
                        CURRENT'BLOCK = NEW(Q'BLOCK) ;
                        IF HEAD <> NULL ;
                            NEXT@TAIL.TAIL = CURRENT'BLOCK ;
                            ELSE HEAD.TAIL = CURRENT'BLOCK ;
                        SUSPEND(TASKID@CURRENT'BLOCK);
                        HEAD = NEXT@CURRENT'BLOCK ;
                        FREE(CURRENT'BLOCK) ;
                        END
            END
      PROC SIGNAL ;
            BEGIN
            IF HEAD = NULL ;
                        BUSY = FALSE ;
                        ELSE RESUME(TASKID@HEAD) ;
            END
      ENTRY WAIT ;
      EXIT SIGNAL ;
      END
```

For a single processor architecture, SUSPEND could get the current task id, save it in a field in the CURRENT'BLOCK, and suspend the task. RESUME would then look up the task id and reschedule the task. A possible implementation of a multi-task system consistent with this method is given below.

For a multi-processor architecture, SUSPEND could just set a bit in the CURRENT'BLOCK and go into a busy wait testing it, until RESUME resets it. Or SUSPEND could save its return and registers in appropriate fields of the CURRENT'BLOCK and do a processor release, in a system with assignable multi-processors. RESUME would then reassign a processor and restart the path.

The following is a very simple example of process control assuming a single processor with a fixed number of tasks. The process data is encapsulated, and includes an index for the executing process and a table of data for each process, including priority, state, and registers. None of this data is accessible

outside the PROCESS'CONTROL structure. Three operations STARTUP, SUSPEND, RESUME are defined for manipulating the process data, and have appropriate meanings.

An encapsulated data type PROCESS'REGISTERS is assumed to describe the machine registers and to include an uninterruptible ASSIGN function for copying to and from the constant CURRENT'REGISTERS (of this type) identifying the physical registers. Registers include a stack pointer for each process (taken as the first register, arbitrarily, and in the initialization of the registers); the stack is assumed to contain the control address on startup and procedure call/return. The return from the SWITCH'PROC procedure which switches tasks causes the selected task to begin or continue execution.

Three status types are assumed to describe priority, process state, and process identification:

```
TYPE PRIORITY'LEVELS STATUS(V(LOW), ... , V(HIGH)) ;
TYPE PROCESS'STATES STATUS (V(DISABLED),V(ENABLED)) ;
TYPE PROCESS'ID STATUS ( ... ) ;
```

Two constants useful in initialization,

```
CONSTANT ITEM NO'PROCESSES U = no of processes ;
CONSTANT ITEM NO'REGS U = no of registers ;
```

are used. Initialization of the process data also assumes some properly initialized stacks (STACK1,STACK2,STACK3, ...) declared and allocated elsewhere.

The first process is assumed to be the lowest level background task which is never disabled. Only one additional priority level is assumed in the initialization of the processes, but intermediate levels could be easily added.

```
TABLE PROCESS'CONTROL RECORD ;
      BEGIN
      EXPORTS(SUSPEND , RESUME , STARTUP) ;
      ITEM EXECUTING'PROCESS PROCESS'ID ;
      TABLE PROCESSES[PROCESS'ID];
            BEGIN
            TABLE PRIORITY [NO'PROCESSES] PRIORITY'LEVELS
                        =V(LOW),NO'PROCESSES-1(V(HIGH));
            TABLE STATE [NO'PROCESSES] PROCESS'STATES
                        =NO'PROCESSES(V(ENABLED)) ;
            TABLE REGS PROCESSOR'REGISTERS
                        = LOC(STACK1), NO'REGS-1(0),
                          LOC(STACK2), NO'REGS-1(0),
                          LOC(STACK3), NO'REGS-1(0);
            END
```

```
PROC STARTUP ;
      BEGIN
      EXECUTING'PROCESS = MIN(PROCESS'ID);
      CURRENT'REGISTERS = REGS[MIN(PROCESS'ID)] ;
      SWITCH'PROC:
      END
PROC RESUME(PROCESS NO) ;
      BEGIN
      ITEM PROCESS'NO PROCESS'ID ;
      STATE[PROCESS'NO] = V(ENABLED) ;
      SWITCH'PROC;
      END
PROC SUSPEND (BYREF PROCESS'NO) ;
      BEGIN
      ITEM PROCESS'NO PROCESS'ID;
      PROCESS'NO = EXECUTING'PROCESS ;
      STATE[EXECUTING'PROCESS]= V(DISABLED) ;
      REGS[EXECUTING'PROCESS] = CURRENT'REGISTERS ;
      STARTUP ;
      END
PROC SWITCH'PROC;
      BEGIN
      ITEM PROCESS'NO PROCESS'ID ;
      ITEM PROCESS'IX PROCESS'ID ;
      PROCESS'NO = EXECUTING'PROCESS ;
      FOR PROCESS'IX : MIN(PROCESS'ID) TO MAX(PROCESS'ID) ;
          IF STATE[PROCESS'IX] = V(ENABLED)
             AND PRIORITY[PROCESS'IX]>= PRIORITY[PROCESS'NO];
             PROCESS'NO = PROCESS'IX ;
      IF PROCESS'NO <> EXECUTING'PROCESS;
          BEGIN
          REGS[EXECUTING'PROCESS] = CURRENT'REGISTERS ;
          CURRENT'REGISTERS = REGS[PROCESS'NO];
          EXECUTING'PROCESS = PROCESS'NO ;
          END
       END


END
```

Section 3

THE SEMANOL SPECIFICATION OF J73/C

The SEMANOL specification of J73/C that was produced under this contract is described in a separate report. Our purpose here is to summarize the results of that effort by extracting from that report.

Since insufficient time was available to produce a complete specification, we concentrated on producing an overall design and on coding in SEMANOL just the more complex portions of J73/C. This approach maximizes the value of the current effort in terms of what it contributes both to an understanding of J73/C and to a better understanding of the strengths and weaknesses of SEMANOL itself. In fact, an important goal of our effort was to investigage how the understandability of SEMANOL specifications could be improved.

The two advantages claimed for SEMANOL as a formal specification method are understandabiiity and executability. Executability, however, is the goal to which the two previous major efforts -- the full J73 and J3 specifications -- seem to have been principally addressed. To whatever extent executability is achievable, SofTech is not in as good a position as TRW (the developer of SEMANOL) to provide useful information. However, the understandability of the previous SEMANOL specification efforts is their main disadvantage in comparison with other formal specification methods. Other operational methods, particularly versions of VDL (the Vienna Definition Language), are more understandable primarily due to their suppression of low level detail. SEMANOL requires low level detail in part because of the simplicity of SEMANOL's control and data structures. That simplicity contributes to understandability and provability within the unit semantic #DF level, but leads to a complex structuring of semantic units that detracts enormously from both understandability and provability of the interaction among units. The consequence is that SEMANOL definitions are longer and require many more semantic definitions than other operational methods. In comparison with higher level formal specification methods, such as denotational and axiomatic methods, the differences are even greater.

The main advantage of the SEMANOL specification method, that it produces an executable description, also tends to compromise the understandability goal. In practice, executability requires an enormous amount of low-level effort. This effort can be reduced and the specification cut down to reasonable magnitude by restricting the language being described, in particular, by leaving out machine dependent features and restricting the language to a "safe" and correspondingly more easily described subset. This subsetting can only go so far, however, because much of the inherent interest in modeling a language is to characterize precisely the complicated and abnormal features that are more difficult to describe, both in informal specifications and with formal techniques such as SEMANOL.

We feel the primary barriers to the understandability of previous SEMANOL specifications are:

1. the modeling of too much of the language, including machine dependent features, such as overlaying, whose effects are easily understood,

2. the mixing of compile-time context sensitive type and attribute checking with control and evaluation semantics,

3. the attempt to use a concrete syntax as a basis for interpretation,

4. the use of implicit or inadequate methods for handling abnormal control flow,

5. the requirement that the specification be executable at the expense of understandability.

We have dealt with the first of these problems by suitably restricting what parts of J73/C are to be specified in SEMANOL, and with the second, third, and fourth by a substantial design effort to find solutions to essentially intrinsic SEMANOL limitations. With less emphasis on executability of the SEMANOL code than on clarity, we feel our partial SEMANOL specification provides a realistic basis for appraising the understandability of SEMANOL descriptions.

In subsequent sections we:

1. describe which portions of J73/C are omitted from our specification design,

2. give an overview of our design approach,

3. summarize which portions of the design have been coded in SEMANOL during the current effort, and

4. identify the principal problems we have encountered in our use of SEMANOL (76).

The current SEMANOL code is given in a separate report.

3.1 The Subset of J73/C Covered by the SEMANOL Design

The language covered by the J73/C SEMANOL design is a "safe" subset of the full J73/C capabilities. In general, the design omits certain machine-dependent or implementation-dependent features and certain

capabilities that could be used to violate type-checking and program modularity. In particular, the major omissions are:

1. The SEMANOL design omits all J73/C facilities dependent on directly specifying machine locations (relative or absolute) -- viz., REGISTER binding, CODE and INLINE procedures, INTERRUPT and ABSOLUTE procedure options, LOC, POINT, NWDSEN, and DSIZE functions, overlays, specified tables, table structure and packing specifiers, based allocation-specifiers, arithmetic operations on pointer values, zones, and the ALLOC and DEALLOC operations. The design does, however, include the machine dependent characteristics that are controlled by the J73/C machine-parameters.

2. The SEMANOL design omits all of the J73/C compiler directives except compool, define, and skip processing. Accordingly, the language does not support user-supplied J73/C restrictions-directives that control compilation capabilities of the J73 compiler. Rather, the SEMANOL design is for a language in which these directives are in effect "preset" so as to maximize program reliability and modularity. Specifically:

IMPLICIT' CONVERSIONS	is disabled
LOOP' REEVALUATION	is disabled
UNRESTRICTED'GOTO	is disabled
CASE'FALL'THROUGH	is disabled
CASE'DEFAULT'OPTIONAL	is disabled
BASED'STORAGE	is disabled
INTEGER'STATUS	is disabled
RANGE'CHECK	is enabled
SIZE'CHECK	is enabled
COMPOOL'REFS'ONLY	is enabled
COLLECTED'DECLARATIONS	is enabled

3. The SEMANOL design omits update blocks, SHARED allocation-specifiers, REENTRANT procedures, and the ENTRY and EXIT operations. These features have little meaning in the absence of multi-processing capabilities; although they provide the basis for a multi-processing implementation.

Except for certain minor omissions in addition to these, all J73/C capabilities can be specified using our SEMANOL design approach; not all capabilities have actually been coded in SEMANOL, however, due to lack of time.

3.2 The Design of the J73/C SEMANOL Specification

The SEMANOL specification of JOVIAL (J73/C) processes J73/C source programs in two phases. (1) the first phase (a compiler) parses the source program (called the "concrete program"), computes the values of formulas whose values must be known at compile time, determines type equivalences for nodes in the parse tree, and produces an "abstract" form of the source program for subsequent processing. (2) The second phase (an interpreter) parses the abstract program according to a corresponding set of syntax definitions and performs the remainder of the semantic analysis, i.e., it executes the abstract program. Phases one and two correspond roughly to compile-time processing and run-time processing, respectively. The advantage of this approach is that it separates the compile-time aspects of the language from the run-time aspects, simplifies the specification of each, and improves the understandability of the specification. This organization of a SEMANOL specification differs from previous uses of SEMANOL because we feel it provides a more understandable formal language specification.

Each major phase consists of two subphases. The compilation phase consists of lexical and translation subphases. The interpretation phase consists of load and execution subphases. The compilation phase first invokes a lexical subphase that processes DEFINE declarations and !SKIP directives. It then invokes a translation subphase that produces a type table and an abstract program and performs compile-time error checking. The lexical subphase does not create a context-free parse tree. Instead, a modified source program string is created and passed to the translation subphase. During the lexical subphase, J73/C DEFINE declarations are processed, the appropriate expansions are made, and program text is deleted as indicated by J73/C !SKIP directives.

The output of the lexical subphase includes:

1. A "concrete" J73/C program string reflecting the above changes.

2. Error messages citing any illegal uses of define declarations or skip directives, or any incorrect code in those constructs.

The translation subphase of the SEMANOL processing parses the J73/C concrete program string and performs three major kinds of activities: (1) it resolves data types of syntactic constructs, e.g., <formula>s; (2) it evaluates expressions whose values must be known at compile time; and (3) it transforms the concrete program into an abstract program. There are two reasons for performing the type resolutions in a separate phase (as opposed to doing the checking as the program is interpreted). First, it permits the checking to be performed only once, rather than repeatedly each time each syntactic construct is encountered during

execution. Second, type-checking requires global information that may not be easily derived at the required times or places during interpretation. Compile-time expression evaluation is performed in this phase because values are sometimes needed to determine type information (e.g., when expressions are used to specify table bounds) and because it is more economical to evaluate such expressions only once -- in this phase -- and replace them with their values, much as a compiler would.

SEMANOL makes it necessary to use two phases to achieve this economy by virtue of the fact that it is not generally possible in SEMANOL to add information to nodes of an already-produced parse tree or to replace a group of nodes with a different (semantically-equivalent) group. If type information could be associated with nodes of the concrete program parse tree, or if nodes corresponding to a compile-time expression could be replaced with a single node corresponding to the expression value, it would not be necessary to have separate parse trees in order to avoid re-evaluation of compile-time information each time a name or formula is re-encountered during interpretation. SEMANOL does provide a very limited form of this kind of capability in that certain semantic definitions may be specified as "syntactic" in a #DECLARE-SYNTACTIC-COMPONENT statement; however, general J73/C type determination requires the use of global variables, which may not be referenced in semantic definitions declared as "syntactic".

The translation subphase outputs a new source string, called the "abstract program". This string is somewhat similar to the concrete J73/C source program. Modifications are introduced to simplify later interpretation. For example, type codes are introduced to distinguish different types, field references are fully qualified by adding any implied field or table names, default specifications are written out explicitly, etc. Various errors concerning type mismatches, undefined names, improper goto's, etc. are detected at this time.

The interpreter phase of the SEMANOL definition parses the abstract program produced by the compilation phase and performs the remainder of the semantic processing. Many of the semantic definitions take as a parameter a state variable whose value corresponds to the state of dynamic memory (static, automatic, and heap storage) of an abstract J73/C machine at any given time during execution.

The load subphase corresponds to a linking loader. Its primary function is to construct the initial state for execution. This involves allocating and presetting the static-store state-component. The execution subphase executes (interpretively) the loaded abstract program and successively transforms the initial loaded state through a sequence of intermediate states to a final state (provided the abstract program terminates).

The compilation and interpretation phases are encoded as separate SEMANOL programs. Maintaining the two phases as separate SEMANOL programs simplifies the process of encoding and incrementally testing the SEMANOL code in that it keeps SEMANOL executions smaller and faster. However, names for syntactic entities, variables, and definitions have been kept unique across the two phases so that they may be easily combined into one larger SEMANOL program at a later date.

3.3 The Portions of the Design Currently Encoded

A complete context free grammar for concrete programs (as output by the lexical subphase), called the concrete syntax, has been coded. The lexical subphase has not been coded since it can be adapted from the existing J73 SEMANOL specification.

The translation subphase has not been coded, nor should it be until most of the interpretation phase has been coded. We expect that completing the coding of the interpretation phase will result in some minor modifications to the abstract program syntax and hence place some additional requirements on the translation subphase. For example, type-codes may need to be inserted in additional places in the abstract program.

The context free syntax for abstract programs and the type table, called the abstract syntax, is completely coded. This syntax was developed by successive transformations of the concrete syntax and thus has a similar structure and nomenclature. This approach should make the translation subphase easier to code and to understand.

Only a portion of the semantic definitions for the interpretation phase have been coded. We have incorporated several concepts from previous VDL definitions into this specification. In particular, states and values are modelled by tree structured objects (that are represented as nested sequences in SEMANOL).

SEMANOL coding for the semantics of the following J73/C features remains to be completed:

1. Procedure and function invocations.
2. All executable statement forms (including all forms of abnormal flow of control).
3. For integer, logical, and table types, all forms of declarations and operators.
4. Type-equivalence semantics for all types.
5. Heap storage.
6. Abstract data types.

7. For the remaining types, all basic operations (including assignment and parameter passage).

8. All intrinsic functions.

It should be emphasized that the design of the SEMANOL (76) specification of J73/C is such that all of the language described in Section 3.1, above can eventually be encoded. Subsequent inclusion of an omitted feature will rarely entail re-design or deletion of existing code. Where possible, "dummy" semantic definitions, or "stubs" have been provided for the omitted semantic definitions.

3.4 A Critique of SEMANOL (76)

SofTech has some initial experience in the use of SEMANOL, through coding a portion of the J73/C subset and through studying the other JOVIAL models. The principal problems we have encountered are:

1. The entirely context-free nature of the parsing function provides no way of using type information to disambiguate use of names with different properties in the same contexts.

2. The lack of information yielded when a parse is ambiguous makes it extremely hard and time consuming to construct an unambiguous syntax. SEMANOL's parsing function accepts any context free grammar and hence it is theoretically impossible to decide whether the grammer is ambiguous. Had SEMANOL's parser been restricted to, say, LALR (1) grammars, the ambiguity problem would be theoretically and practially decidable.

3. The absence of operations for modifying a parse tree or associating computed information with a node requires either:

   a. a complicated interpreter with type checking incorporated into many semantic units and in general a complicated relationship between parse tree nodes and associated semantic routines, or

   b. an extra pass which maps the parsed program into an intermediate form with additional information inserted, using string concatenation operations and an additional parse.

   The first corresponds to the method used in the other JOVIAL models, while the second corresponds to the analysis/synthesis phases of a compiler as discussed in our design above.

Note that one can indirectly associate information with a node by constructing maps or tables such as the static-store-map in our interpretation phase. We expect that the translation subphase will have to construct several such maps, in particular ones for node types and manifest-formula values. This will complicate the translation subphase, but simplifies the interpretation phase by having the translation subphase incorporate the information directly into the abstract program.

4. The limitations on control flow, as well as the lack of local variables in semantic #DFs, complicates the modeling of abnormal control flow (goto's out of control structures and exception handling mechanisms); our design incorporates detailed information on exits or abnormal returns at each parse tree node associated with a significant control action. As has been observed in TRW course notes, having to define the semantics of abnormal control flow significantly increases the complexity of the definition. We have considered several ways to model the abnormal control flow. We feel that the following two ways are most appropriate for an operational method such as SEMANOL.

   a. The first way is to have a modifiable control component such as the control tree in a VDL definition. The tree contains the instructions for the semantic actions that remain to be done. Normal control flow changes only the leaves of the tree (deleting nodes or replacing them with subtrees). Abnormal control flow causes a subtree to be replaced with another subtree. This approach has been successfully used in a VDL definition of Algol 68. However, VDL's notation and primitives are designed to make this approach easy and understandable, whereas SEMANOL's primitives are not. Furthermore, simulating the VDL approach in SEMANOL is not easy since SEMANOL lacks VLD's tree structured objects with named selectors. The result would be a definition that is harder to write, read, and understand. Hence, we have rejected this approach.

   b. The second way is the approach we have taken. Here, each semantic definition that needs to be aborted due to abnormal control flow must return an additional component that indicates normal or abnormal flow. The invoking semantic definition must test this returned component and take appropriate normal or

abnormal action. This approach has been successfully used in a functional VDL definition of Algol 60 and blends better with the functional nature of SEMANOL's semantic definitions. However, as the Algol 60 definition shows, the tests for abnormal control flow occur throughout a large portion of the definition, thereby significantly increasing its complexity.

In the '73 version of SEMANOL, it was possible to shortcut the additional interpretation by using the #RESUME function for clearing the SEMANOL control stack to a desired level. However, this required an implicit arrangement of semantic unit levels to keep the correspondence between the SEMANOL control stack and the modeled environment "in step". The problem with this use of #RESUME was that it complicated the control model in implicit and therefore subtle ways.

Although our functional approach to modeling abnormal control flow is compatible with SEMANOL's functional definitions, one still must write an excessively large number of #DF's when compared with, for example, the functional VDL model of Algol 60. There seem to be two reasons for this:

a. SEMANOL does not permit one to declare and use local abbreviations in a #DF. Instead, one must invoke another #DF and, in effect, use its formal parameters as abbreviations.

b. SEMANOL does not permit short-circuit evaluation of boolean expressions. Given that one is working with highly structured objects in the forms of parse trees and nested sequences, it would be helpful to be able to write conditional results such as

```
=> result #IF x #IS #SEQUENCE #AND
                some-sequence-predicate (x);
```

when x may be either a sequence or a node. A related problem is SEMANOL's lack of an #ELSE alternative for its #IF statements (in #PROC-DFs).

It appears to us that these capabilities could be easily added.

We would also like to eliminate the requirement that global and local names be distinct. Given the large number of

#DFs, it is hard enough when writing a new #DF to keep its name distinct from the others, but having to make it distinct from the formal parameter names of the other #DFs is unreasonable.

5. The lack of VDL-like tree structured objects with nameable selectors is an inconvenience that has forced us to define a large number of selector, constructor, and update #DFs in order to cope with the dynamically changing tree structure (nested sequences) of states and values. SEMANOL does provide a large assortment of operations for parse trees and for the top-level elements of sequences; in fact, so many that it is difficult to remember exactly which forms are available and where they stand in the syntactic hierarchy of SEMANOL expressions.

These problems are not new. They have apparently been discussed since the beginnings of SEMANOL. To some extent, they reflect fundamental questions about the nature of a formal specification tool. For instance, the requirement that the initial parse be entirely context free reflects the notion that programming languages should be as context free as possible. Also, the lack of operations on trees and the applicative nature of semantic units reflects the notion that semantic definition systems should be as simple as possible in order to provide an inherently simpler characterization of the language being described than the language itself. The result of these characteristics, however, is that the SEMANOL specification itself becomes inordinately large and complicated in order to cope with the context-sensitivities, complex concrete-to-abstract-syntax relationships, and abnormal control structures of the language being specified. The value of the SEMANOL specification of the language becomes severely diminished.

As a final comment, it is interesting to note that the difficulties that appear most intractable in writing and understanding SEMANOL specifications, namely,

1. context-sensitive characteristics of programming languages, including type descriptions and environment representations,

2. abnormal control flow,

3. complex relationships between concrete and abstract representations of programs,

represent the fundamentally hard problems in any formal specification method. In providing a framework for dealing with the essentially representational difficulties in solving these "hard problems" in SEMANOL, SofTech has provided important data on the relative merits of SEMANOL as a specification method.

Section 4

THE COMPILER SPECIFICATION

A specification for J73/C compilers was developed for use in a competitve procurement of J73/C compilers. This specification was written as SofTech document 3061-15. In addition, a set of J73/C compiler acceptance tests was produced and added to the set of J73 tests produced for use with the Jovial Compiler Validation System (JCVS). In this section of the Final Report, we briefly describe the information contained in the compiler specification and the acceptance tests.

4.1 The Functional Specification for a Compiler

A specification was produced for a J73/C compiler operating on both the Honeywell 6180 under MULTICS and the Interdata 7/32 and 8/32. The H6180 compilers were specified to produce object code for the H6180 and the Interdata computers. In addition, the Interdata compilers were specified to produce code for the computer on which they executed. These compilers are to be used in the experimental evaluation of the J73/C language.

In accordance with the statement of work for this effort, we specified a compiler architecture that separated the generaion of code from the other phases of compilation, so the development of new code generators and the re-hosting of a J73/C compiler would be easier. In addition, the specification describes the form of the source program listing, environment listing, set-used listing, symbol table output for use by symbolic debuggers, and about 200 error messages, exclusive of syntactic errors.

The functions performed by the major modules were specified in terms of their inputs, outputs and processing. The modules were decomposed into submodules specified in the same fashion. A requirement of the design activity specified in the same document is to continue the decomposition in the same fashion to the level of individual procedures. This will lead to a complete and understandable design document as a basis for implementation and a framework for program documentation.

Interface requirements for the compiler were also specified. They included:

- invocation of the compiler by the standard mechanism for initiating processors on the host systems,
- a complete list of compiler options,
- complete description of all compiler listing outputs,
- compatibility of relocatable module format with the standard module format for the host system,

- provision for special procedure calls to interface with FORTRAN and PL/I compiled procedures on the host systems,

- support for statistics collection and for use of symbolic debugging capabilities of the host systems.

Finally, compiler performance was required to include:

- throughput of 2000 lines per minute, under exactly described testing conditions,

- optimization to perform specified global machine independent optimizations using a prescribed decomposition of optimizer functions,

- overall optimizer/code generator performance such that object code is within 10% of equivalent assembly language.

The overall compiler design is such that the compiler will support all of these objectives.

4.2 The Acceptance Tests

In addition to the compiler specification, test programs were written in J73/C to test just those features that are not in J73. These tests were merged with the existing JCVS tests for J73; together with these tests, they exercise every feature of the language. The J73/C tests have the format and style of the existing J73 JCVS tests.

The JCVS for J73/C produced in this contract contains most of the code for the J73/I version of JCVS. That code consists of the class 1 tests (for correct processing of J73/I features) and the class 5 tests (for compool processing and accessing capabilities).

The J73/I JCVS tests adapted for J73/C are divided into separate files, and each file (or in some cases, each group of files) may be compiled and executed as a separable part of JCVS. Files consist of "modules", each of which contains code to test for a given language feature or capability (actually, a module boundary is merely a comment followed by the assignment of a new module number to a global variable.) The tests are "self-checking" in that execution of the code for the tests causes a message to be printed for each test, indicating whether the compiler "passed" or "failed". Because no I/O is defined in standard J73/I, printed output is obtained by assigning to a global character string variable for which a !TRACE directive is active.

In adapting the J73/I tests for J73/C, the data base and the output reporting mechanism routines were used directly.

To augment the Class 1 and Class 5 tests for J73/C, additional "modules" were written for the extended features. These modules follow the existing J73/I JCVS style and format, and they were numbered so that they could be interspersed with existing modules according to the kinds of features they test. (i.e., tests for extended table capabilities were placed among the J73/I table tests; tests for extended procedure features were placed with J73/I procedure tests, etc.) The J73/I module numbering left more than enough "gaps" for new module numbers could be inserted between existing ones.

Each added module contains, in the right margin of its first line, a comment specifying "J73/C 6-78". Any line of existing J73/I code that was modified in any way was similarly commented. Usually, modifications in existing lines of code were necessary only in !COMPOOL and !COPY directives, which refer to file names.

The existing error tests (Class 2) in JCVS for J73/I were completely replaced by new tests for J73/C. These new error tests were written according to the requirements for error detection that are stated in the J73/C compiler specification. Because the Class 2 tests did not entail interspersing new tests with old, the style of coding in the J73/C version is somewhat different. Identifiers have a spelling that corresponds to section numbers in the J73/C reference manual. Due to the nature of the error tests, they are not intended for execution.

JCVS for J73/C also contains Class 4 (capacity and efficiency) tests that verify that the J73/C compiler meets each of the various capacity requirements stated in the J73/C Compiler Specification. These tests, like the Class 2 tests, consist entirely of new code and do not include any code from the J73/I JCVS (except for the output reporting routines accessed via !COPY directive). These tests, too, are "self-checking" in the manner of the Class 1 and Class 5 tests.

The JCVS Class 5 testing, according to the J73/C compiler specification, is more stringent than Class 1 testing. Therefore, for J73/C Class 5 testing, it is intended that all of the Class 1 tests be passed as a prerequisite for passing Class 5 tests. That is to say, Class 1 tests should be used for Class 5 testing in addition to Class 1 testing. Those tests indicated as "Class 5" tests in the J73/C JCVS include only those tests that are not also Class 1 tests.

Every effort was made to assure that the new code for the JCVS tests is as free of errors as possible. However, because no J73/C compiler was available, the new tests had to be verified by visual examination only. When the tests are compiled and executed for the first time, some amount of debugging will be required.

APPENDIX A

The following changes are to be made in The J73/C Communications HOL Reference Manual that is dated April 7, 1978, and amended by the change pages of June, 1978:

PAGE	CHANGE
1-3	Add the word "signed" to the text describing MAXINTEGER and MININTEGER (middle of page), so that MAXINTEGER is the "Maximum value of signed integer values" and MININTEGER is the "Minimum value of signed integer values."
1-6	Add the following sentences as a new, additional paragraph between the first paragraph and the second paragraph: All of the *compool:modules* constitute one outer scope. Each *program:module* forms an inner scope relative to the compool scope.
1-9	Add the following sentence at the end of the fifth paragraph: The allocation permanence of SHARED data is the same as that of reserve.
1-14	Add the following sentence to the end of the second paragraph in 1.6.2.2: There is only one *program:declaration* in a *complete:program*.
1-15	The last sentence in the second paragraph of 1.7 should read, "non-encapsulated program-defined types (Sections 2.2.1 and 2.2.2) have the same rules as (i.e., are identical to) their underlying types."
1-15	In the first sentence of the last paragraph, encapsulated types should be *encapsulated:types*; also add the colon and italics for status types in the same sentence.
1-15	Add the following text to the end of Section 1.7: The attributes of routine types are the in/out-ness, binding, and type of each parameter, and the result type. The attributes of table types are the lexical order of component fields and variants, the component types, the variant label values (but not their order within a single label-list), the type of each dimension index, the tag field, and the structure and packing specifier.

Example:

Given TYPE AA t;
TYPE BB t;

where both t's are the same legal J73/C construct, then

AA, BB, and t are all identical types except when t is a *status:type* or an *encapsulated:type*. In these cases, AA and the first t are identical types, and BB and the second t are identical types, but the types of AA and BB are not identical to each other.

1-19 6th line from the bottom of the page: change the word "nonzero" to "nonpadding."

2-5 Replace the second occurrence of MAXINTEGER with 2**MAXINTEGER+1

2-5 Replace the last sentence on this page with the text:

If the size is less than that required to represent the specified S or U range, the range is taken to be the subrange of the specified range which can be represented using the given size. Otherwise, for explicit subranges, a compile-time error results.

2-8 Replace the fourth and fifth paragraphs with the following:

The *primary:status:sublist* establishes the representation of the first *status:literal* in the sequence. If a *status:list:index* is present, the first *status:literal* is represented with the index's integer value; if no *status:list:index* is present, a value of zero is assumed. The representation of the next *status:literal* in the sequence is the value of the index plus one, etc.

Insert after the sixth paragraph, the following paragraph:

The size of the representation of a status value is the minimum required to represent its specified values or differentiate the values in its type.

2-10 In the production for *input:description* (middle of the page) replace the / *aggregate:type* alternative with the following two alternatives:

/ TABLE *ordinary:table:description* (2.1.8)

/ TABLE *specified:table:description* (2.1.8.7)

2-17 Change the third alternative right-hand side of the *table:description* production so that it reads as follows:

```
/ dimension:specifier  aggregate:type:name   (2.1.8.3)(2.2)
  preset:part ;                              (2.1.10)
```

2-17 Replace the last sentence on the page with the following text:

> In the third alternative of *table:description*, the information describing the table is taken from the *type:definition* having the specified *aggregate:type:name*, except that (1) a non-nil *preset:part* in the *table:description* overrides all *preset:parts* in the corresponding *type:definition*, and (2) any arrayness specified in the *dimension:specifier* preceding the *aggregate:type:name* is in addition to whatever arrayness is specified in the corresponding *type:definition*.

2-18 After the last sentence on the page, add the following sentence as a separate paragraph:

> If the *preset:part* is not *nil*, then it overrides all *preset:parts* in the *table:body* (including those associated with *aggregate:type:names*).

2-20 In the middle of the long paragraph, change "...the value of the formula must be greater than zero..." to read "...the value of the formula must not be less than zero...".

2-24 In the right-hand-side of the *ordinary:variant* production, *ordinary:table:component:declaration* should be immediately followed by a raised asterisk.

2-25 Add the following paragraph after the last paragraph on the page:

> The first level field names of an *ordinary:table:body* must be mutually distinct. The first level names include those of the common part and each *ordinary:variant*.

2-26 In the right-hand-side of the *specified:variant* production, *specified:table:component:declaration* should be immediately followed by a raised asterisk.

2-27 Add the following paragraph after the last paragraph on the page:

> The first level field names of a *specified:table:body* must be mutually distinct. The first level names include those of the common part and each *specified:variant*.

2-29 In the right-hand-side of *block:body:part*, change *data:declaration* to *item:declaration* and add

> / *block:declaration*
>
> / *table:declaration*

Also, make these changes in the right-hand-side of *block:body:options*.

2-30 Add the following sentence as a separate paragraph after the last paragraph:

> Blocks may be passed as parameters only if IMPLICIT'CONVERSIONS is enabled.

2-33 Replace the first sentence in the second paragraph with the following sentence:

> If the data structure being preset is not an aggregate, a non-nil *preset:part* must consist of an equal sign followed by a single *preset:formula*.

2-35 In the 4th alternative right-hand-side of *aggregate:type*, the word FORWARD should be followed by a semi-colon.

2-38 Replace the first sentence of the first paragraph with the following:

> *Encapsulated:types* are a subclass of table types that enable tables to be defined with extra protection.

2-38 Add the following sentence to the end of the second paragraph:

When referenced, component names must be qualified with the name of a table declared to be of the encapsulated type. Typically, this table will be a formal parameter of the procedure or function in which the component is referenced.

2-39 Add the following text as an additional paragraph:

> The operations are not part of the record allocated for each object of an encapsulated type. Instead, they are common routine-valued constants.

2-46 In the last sentence of the first paragraph, replace "actual parameter" with "formal parameter."

2-46 Add the following text after the first sentence in the second paragraph:

> In these cases, the executable body must be provided in a separate *procedure:declaration*. A *procedure:name:declaration* (or *procedure:declaration* having only *parameter:declarations* in its body) is associated with the "bodied" *procedure:declaration* having the same *procedure:name* with the same scope.

2-46 Add to the end of the second paragraph:

> If COMPOOL'REFS'ONLY is enabled, PROC declarations containing only parameter declarations are prohibited, external PROCNAME declarations are permitted only in COMPOOL modules (see Section 2.7), and redeclared parameter attributes and presets must be identical to the attributes specified in the preceding PROCNAME declaration.

2-49 Add the following text after the first sentence in the last paragraph:

> In these cases, the executable body must be provided in a separate *function:declaration*. A *function:name:declaration* (or *function:declaration* having only *parameter:declarations* in its body) is associated with the "bodied" *function:declaration* having the same *function:name* with the same scope.

2-49 Add to the last paragraph:

> If COMPOOL'REFS'ONLY is enabled, PROC declarations containing only parameter declarations are prohibited, external PROCNAME declarations are permitted only in COMPOOL modules (see Section 2.7), and redeclared parameter attributes and presets must be identical to the attributes specified in the preceding PROCNAME declaration.

2-50 In the third line of the first paragraph, add the words "PROCNAME or" after the word "using."

2-50 In the first paragraph under 1., BYVAL and BYRES parameters, replace the first sentence with the following sentence:

> Formal BYVAL and BYRES parameters can be declared as any other local name.

Replace the last sentence in the same paragraph with the following sentences:

> BYVAL and BYRES parameters can be overlaid and, if RESERVE, can be specified "external." CHAR * parameters, tables with * indices, and all based parameters cannot be preset; all other BYVAL and BYRES parameters can be preset.

2-50 Replace the last paragraph under 1. with the following:

> Item input parameters are BYVAL by default and cannot be bound BYRES. Output parameters are BYRES by default and cannot be bound BYVAL. Tables and blocks cannot be declared as output parameters of BYVAL input parameters and are bound BYREF by default.

2-50 After the first sentence in the first paragraph under 2., BYREF parameters, add the sentence "If the BYREF parameter declaration contains an *aggregate:type:name*, any presets in the corresponding *aggregate:type:definition* are ignored."

2-50 Two-thirds of the way down the page: delete the sentence "Values of BYVAL and BYRES parameters are copied on entry and exit of a procedure or function."

2-50 Next-to-last line: replace the word "Otherwise" with the clause, "If IMPLICIT'CONVERSIONS is enabled."

2-61 Add the following paragraph to the end of 2.7.1.2:

> If COMPOOL'REFS'ONLY is enabled, a *program:module* containing a DEF PROC must reference a *compool:module* containing a corresponding DEF PROCNAME.

2-66 In the first line of the third paragraph, change the word "may" to "must."

3-2 After the second sentence in the next-to-last paragraph, add the following parenthesized sentence:

> (It is, however, the user's responsibility to avoid situations that could lead to deadlock.)

3-3 Add the following sentence to the end of the next-to-last paragraph in 3.3.2:

> The system-supplied table assignment is a bit-for-bit copy of the source into the target, with padding or truncation as necessary.

3-4 Replace the second paragraph with the following:

> If UNRESTRICTED'GO'TO is disabled, the *goto:statement* may not be used to transfer control out of a name scope or UPDATE *compound:statement* or into a *controlled:statement*, *switch:statement*, or UPDATE *compound:statement*.

3-9 In the first paragraph of 3.8.2, delete the second and third sentences.

3-11 In the fourth paragraph, replace the second sentence with the following:

> "Such a *switch:point:index:group* is legal only if (1) it is not the first *switch:point:index:group* in the *switch:statement*, (2) the type of the *switch:point:selector* is integer, and (3) CASE'DEFAULT'OPTIONAL is enabled."

3-11 In the fifth paragraph, delete "the *switch:point:selector* is a status or integer subrange or logical value and."

3-11 In the next-to-last paragraph, replace the second, third, and fourth sentences with the following:

> If the comma is omitted, control passes to the textually succeeding *switch:point*. After execution of the final *switch:point*, control passes to the statement following the *switch:statement*.

3-13 In the second alternative of the production for continuation close up that space between "*replacement:*" and "*phrase*".

3-19 Add the following sentences to the end of Section 3.12.2 as an additional paragraph:

> The handler procedure is located by applying the static namescope rules, not by examining the dynamic calling sequence. The *return:statement* in the handler procedure must include a *return:name* in order to specify the point at which execution is to resume.

4-5 Add the following text as a new paragraph between the fourth and fifth paragraphs:

A *logical:formula* is evaluated in short-circuited mode. This means the formula is evaluated from left to right, with only as many of the constituent *logical:expressions* evaluated as are necessary to determine the value of the complete *logical:formula*.

4-6 Add to the end of the second semantics paragraph:

> The AND and OR operations are evaluated in short-circuit mode.

In the second example, "AA and BB" should read "AA AND BB."

Add a third example:

```
P <> NULL AND V@P        V@P is not evaluated if P = NULL
```

4-10 Last sentence in second paragraph: change the period to a comma and append the phrase, "without regard to type."

4-10 Add "dot qualification" to the list of operations that have precedence level 9.

4-19 The right-hand-side of the bitee production should be *"bit:formula"*, not just *"formula"*.

4-29 At the end of the next-to-last sentence on the page, add the phrase "and cast its result to type P *aggregate:type:name*."

4-36 Add the following as a fourth alternative right-hand-side of the *rangee* production:

```
/ status:type:name                                    (2.1.8.3)
```

4-36, 4-37 In both 4.12.21.2 and 4.12.22.2, change the phrase "specified numeric variable or numeric subrange type" to read "specified numeric variable, numeric subrange type, or status type."

4-38 Beginning with the production for table, replace the remainder of the page with the following:

```
table                    ::= non:subscripted:table
                             non:subscripted:table[index+,]

non:subscripted:table    ::= table:name              (2.1.7)
                           / @ pointer:formula       (4.8)
                           / table:name @ pointer:formula
                           / non:subscripted:table.
                             table:name

table:item   ::= non:subscripted:table:item
                 / non:subscripted:table:item [index+,]
                 / non:subscripted:table [index]

non:subscripted:table:item ::= table:item:name      (2.1.8.6)
                             / non:subscripted:table
                               . table:item:name
                             / table:item:name @
                               pointer:formula
```

4-40 In the third paragraph, precede the first sentence with the following additional text:

> A "." in a *table* or *table:item* denotes qualification.

After the last sentence in the same paragraph, append the following:

> There is no tag field protection when addressing variant fields, even in switch statements.

5-9 In the first right-hand-side alternative of *fractional:form*, make *digit** be *digit+*.

6-3 Remove the blank between ! and COPY.

6-4 In the second line of the next-to-last paragraph, add the word "unmatched" between the words "preceding" and *"begin:directive"*.

6-14 In LOOP'REEVALUATION (2.), replace the second sentence with "If LOOP'REEVALUATION is disabled, the *increment:phrase* is evaluated once at the start of the loop and the initial value(s) used for every loop iteration."

6-14 Replace paragraph 3 with the following:

> 3. UNRESTRICTED'GO'TO enables the use of branches out of the current scope and update statements or into loops, switches, and update statements. Branches to label parameters are included in branches outside the current scope. (UNRESTRICTED'GO'TO is enabled for compatibility with J73/I.)

6-14 Add at the end of the first sentence of paragraph 5, "and *switch:point:indexes*". In the second sentence, delete the phrase "the switch index is of status, integer subrange, or logical type and".

6-15 Add at the end of the first sentence of paragraph 10:

> and redeclared parameter attributes and presets must be identical to the attributes in any preceding PROCNAME declaration.

6-15 In the second line of paragraph 11, after the word "declarations," add the phrase "except *procedure:declarations* and *function:declarations*."

7-1 Add to the end of the first item 3:

or the combination is a *logical:formula* of the form

TRUE OR operand

or the form

FALSE AND operand

Add to the end of the second item 3:

except for *logical:formulas* of the form

TRUE OR operand

or

FALSE AND operand

A-7 In *table:description*, make the syntax change described above for page 2-17.

A-7 In *block:body:part* and *block:body:options*, make the syntax changes described above for page 2-29.

A-8 In *aggregate:type*, make the syntax change described above for page 2-35.

A-16 In *input:description*, make the syntax change described above for page 2-10.

A-19 In *ordinary:table:table:declaration*, make the syntax change described above for page 2-24.

A-25 In continuation, close up the space as described above for page 3-13.

A-27 Replace the productions for *table*, *table:qualifier*, and *table:item* with the productions given above for page 4-38.

A-37 In *bitee*, make the syntax change described above for page 4-19.

A-40 In *rangee*, make the syntax change described above for page 4-36.

A-46 In *fractional:form*, make the syntax change described above for page 5-9.

B-2 For *aggregate:type*, remove 2.1.3.5 under REFERENCES.

B-3 For *bit:formula*, add 4.12.4 under REFERENCES.

B-3 For *block:declaration*, add 2.1.9 under REFERENCES.

B-6 For *data:declaration*, remove 2.1.9 under REFERENCES.

B-6 For *dimension:specifier*, add 2.1.7 under REFERENCES.

B-8 For *formula*, remove 4.12.4 under REFERENCES.

B-10 For *item:declaration*, add 2.1.9 under REFERENCES.

B-14 Add (after *nolist:directive*):

	DEFINITION	REFERENCES
non:subscripted:table	4.13	4.13
non:subscripted:table:item	4.13	4.13

B-15 For *ordinary:table:description*, add 2.1.3.5 under REFERENCES.

B-19 For *specified:table:description*, and 2.1.3.5 under REFERENCES.

B-20 For *status:type:name*, add 4.12.21 under REFERENCES.

B-21 For *table:declaration*, add 2.1.9 under REFERENCES.

B-21 For *table:description*, add 2.1.8.6 under REFERENCES.

B-21 Remove *table:qualifier* (and its DEFINITION and REFERENCE numbers).

MISSION
of
Rome Air Development Center

RADC plans and conducts research, exploratory and advanced development programs in command, control, and communications (C^3) activities, and in the C^3 areas of information sciences and intelligence. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.