UCID- 20332

CIRCULATION COPY
SUBJECT TO RECALL
IN TWO WEEKS

PRAXIS
RELEASE NOTES
VERSION 7.2

Frederick W. Holloway
Anthony J. De Groot
Timothy A. Sherman

January 18, 1985

Lawrence Livermore National Laboratory

This is an informal report intended primarily for internal or limited external distribution. The opinions and conclusions stated are those of the author and may or may not be those of the Laboratory.
Work performed under the auspices of the U.S. Department of Energy by the Lawrence Livermore National Laboratory under Contract W-7405-Eng-48.

DISCLAIMER

This document was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor the University of California nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial products, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or the University of California, and shall not be used for advertising or product endorsement purposes.

Printed in the United States of America
Available from
National Technical Information Service
U.S. Department of Commerce
5285 Port Royal Road
Springfield, VA 22161
Price: Printed Copy $; Microfiche $4.50

Page Range	Domestic Price	Page Range	Domestic Price
001-025	$ 7.00	326-350	$ 26.50
026-050	8.50	351-375	28.00
051-075	10.00	376-400	29.50
076-100	11.50	401-426	31.00
101-125	13.00	427-450	32.50
126-150	14.50	451-475	34.00
151-175	16.00	476-500	35.50
176-200	17.50	501-525	37.00
201-225	19.00	526-550	38.50
226-250	20.50	551-575	40.00
251-275	22.00	576-600	41.50
276-300	23.50	601-up[1]	
301-325	25.00		

[1]Add 1.50 for each additional 25 page increment, or portion thereof from 601 pages up.

```
PPP   RRR    AA   X   X  III   SSS
P  P  R  R  A  A   X X    I   S
PPP   RRR   AAAA    X     I    SS
P     R R   A  A   X X    I      S
P     R  R  A  A  X   X  III  SSS
```

Release Notes
Version 7.2

Fred W. Holloway
Anthony DeGroot
Tim A. Sherman

Controls and Analysis Group
Nova High Energy Laser Facility
Laser Program
Lawrence Livermore National Laboratory
P. O. Box 808, L-492
Livermore, CA 94550
(415) 422-5917

Generated on: January 18, 1985

file: disk_lib:[praxislib.documents.version72]version72.doc

Table of Contents

CHAPTER 1 INTRODUCTIONS

1.1 PURPOSE OF THESE NOTES 1-1
1.2 ORGANIZATIONAL CHANGES TO ACCOUNTS 1-1
1.2.1 Removal Of Non-Dependent Software 1-1
1.2.2 Separate Products For TEXTIO, OSI, RMS, And Conversion Routines 1-1
1.2.3 Praxis Account 1-3
1.3 INSTRUCTIONS FOR USE 1-3
1.3.1 Version 7.2 At The LIB Level 1-3
1.3.2 New Logical Symbols For Other Products 1-3
1.3.3 Praxis Logical Symbols, Libraries, And Global Symbols 1-6
1.3.3.1 Logical Symbols 1-6
1.3.3.2 Library Names 1-7
1.3.3.3 Global Symbols For Switching Between Versions 1-7

CHAPTER 2 IMPROVEMENTS TO THE PRAXIS COMPILER

2.1 DECLARATIONS 2-1
2.1.1 Size Of Enumerations 2-1
2.1.2 Enumerations As Conversion Functions 2-1
2.1.3 Easier Way To Declare Strings 2-1
2.1.4 Compile-time Constants 2-2
2.1.5 Array Constructor Improvements 2-2
2.1.5.1 Arrays Of 1 Character Strings 2-2
2.1.5.2 Multiple Dimensioned Array Run Time Constructors 2-2
2.1.6 Initialized Variant Static Structures 2-3
2.1.7 Structures And Other Objects With Holes 2-3
2.1.7.1 Compare Operations On Objects With Holes . . . 2-3
2.1.7.1.1 Strings With Odd Number Of Characters . . . 2-4
2.1.7.2 The Problem Of Control Of Holes 2-4
2.1.7.2.1 Possible Solutions 2-4
2.1.7.2.2 Implementation 2-5
2.1.7.3 Compiler Directives 2-5
2.1.7.3.1 Clear_dynamic_Holy_Types Compiler Directive 2-5
2.1.7.3.2 Warn_Dynamic_Holy_Types Compiler Directive . 2-6
2.1.7.4 Object With Holes Within The Compiler 2-6
2.1.8 Rules And Effects Of Packing 2-6
2.1.9 Table Constructor Size Checking 2-7
2.2 MATHEMATICAL STATEMENTS 2-7
2.2.1 32 Bit Integers On The LSI-11 2-7
2.2.1.1 Assignment Of Constants 2-7
2.2.1.2 Comparisons (<=, <, >, >=) 2-7
2.2.1.3 Sizeof 2-8
2.2.1.4 For Loops 2-8
2.2.1.5 IN VAL Parameters 2-8
2.2.2 <1 To 15> Bit Arrays On LSI-11 2-8
2.2.3 Multiply, Divide, Mod Of Small Sized Integers And Cardinals 2-9

2.2.4 Logical Right Shift 2-9
2.3 EXCEPTIONS 2-9
2.3.1 Exceptions In Detached Processes 2-9
2.3.2 Guard Block Alignment 2-10
2.3.3 Rangechecking 2-10
2.3.3.1 Compile Time Range Checking On Assignments . 2-10
2.3.3.1.1 No Range Check On Large Object Assignments 2-10
2.3.3.2 Rangechecking Of Flexible Arrays 2-11
2.3.3.3 ARM X_RANGE_ERROR 2-11
2.3.3.4 Rangechecking Compiler Directives 2-11
2.3.3.5 DISARM Explanation 2-11
2.3.3.6 Global Symbols For Exceptions On PDP11 . . . 2-11
2.3.3.7 RANGE Statement Modification 2-12
2.3.3.8 Some Subrange Statements Crashed The Compiler 2-12
2.3.4 Alternative To Integer Overflow Exceptions . 2-13
2.4 ALLOCATE AND FREE 2-13
2.4.1 One Word Object With Initial Value On LSI-11 . 2-13
2.4.2 Free Of Large Items 2-13
2.4.3 Protection From Use Of Nil Pointers 2-14
2.5 FLEXIBLE ARRAYS 2-14
2.5.1 Allocation Of Flexible Arrays On The LSI-11 . 2-14
2.5.2 Assignment Of Flexible Arrays 2-14
2.5.3 Comparison Of Flexible Arrays 2-14
2.5.4 HIGH And SIZEOF Functions Of Flexible Arrays . 2-14
2.5.5 Local Variable Declared For High Of Flexible Array 2-15
2.6 RUN TIME SUPPORT, RSX AND STAND-ALONE LSI-11 . . 2-15
2.6.1 Heap Support 2-15
2.6.2 Raising And Catching Exceptions 2-15
2.6.3 Hardware Exceptions 2-15
2.6.4 Boot Time Diagnostics For Stand-alone LSI-11 . 2-16
2.6.5 Segment Control On The LSI-11 2-16
2.7 IMPORT/EXPORT IMPROVEMENTS 2-16
2.7.1 Initialization Of Static Imported Types . . . 2-16
2.7.1.1 USE Statement Must Be Used 2-16
2.7.2 Encode External Names 2-17
2.8 PROCEDURE/FUNCTION INTERFACE IMPROVEMENTS . . . 2-17
2.8.1 Clarification Of Size Check On Ref Parameters 2-17
2.8.2 Procedure Parameter Combinations 2-18
2.8.2.1 General Rules For Parameters 2-18
2.8.2.2 Unworkable Parameter Combinations 2-19
2.8.2.3 All Possible Combinations 2-19
2.8.3 Functions Returns For Flexible Parameters . . 2-21
2.9 BETTER INTERFACE TO VMS AND RMS 2-21
2.9.1 Fortran Procedure Parameter Changes 2-21
2.9.2 Variadic Parameters 2-21
2.9.2.1 Variadic REF GENERAL WITH SIZE 2-22
2.9.2.2 Variadic Flexible Arrays 2-22
2.10 OPTIMIZATION 2-23
2.10.1 Constant Expresions Evaluated At Compile Time 2-23
2.11 DEBUGGING IMPROVEMENTS 2-23
2.11.1 Better Diagnostic Messages 2-23
2.11.2 Unencoded And Encoded Names In PDP11 M11 File 2-24
2.11.3 Wider Symbols In Listings 2-24
2.12 BATCH AND INTERACTIVE MODE INCONSISTENCIES . . . 2-25

2.13 INTERNAL COMPILER DIAGNOSTICS 2-25
2.13.1 Trace Messages 2-25
2.13.2 Printing The Forest 2-25

CHAPTER 3 EXAMPLE PROGRAMS

3.1 SIZE AND CONVERTERS OF ENUMERATIONS 3-2
3.2 ARRAYS OF 1 CHARACTER STRINGS 3-3
3.3 ARRAY CONSTRUCTORS 3-4
3.4 THREE DIMENSIONAL ARRAY CONSTRUCTORS 3-8
3.5 ARM AND USE OF RUN-TIME RANGECHECKING 3-10
3.6 RANGECHECKING OF SUBRANGES 3-11
3.7 EXPORT OF LARGE STRUCTURE WITH INITIAL CONDITIONS 3-12
3.8 IMPORT OF LARGE STRUCTURE WITH INITIAL CONDITIONS 3-12
3.9 ALLOCATION SYNTAXS 3-13
3.10 PROTECTION FOR USE OF NIL POINTERS 3-15
3.11 FUNCTIONS RETURNS FOR FLEXIBLE PARAMETERS . . . 3-16
3.12 HIGH OF VARIADIC FLEXIBLE ARRAY 3-17
3.13 OPTIMIZATION OF CONSTANT EXPRESSIONS 3-18
3.14 SEGMENT CONTROL ON THE LSI-11 3-20

CHAPTER 4 SUGGESTED FUTURE IMPROVEMENTS

4.1 SETS 4-1
4.1.1 Initially Of Sets 4-1
4.2 MATH ISSUES 4-1
4.2.1 Unsigned Comparisons 4-1
4.2.2 Real Exponentiation 4-1
4.2.3 Long Real Constants 4-2
4.2.4 Long Real On PDP Not Implemented. 4-2
4.2.5 Round, Floor, Ceiling For The PDP-11 4-2
4.3 ADDITIONAL COMPILE-TIME RANGECHECKING 4-2
4.4 INTEGER AND CARDINAL CONSTANTS SHOULD BE EQUIVALENT 4-2
4.5 SELECT PROBLEMS 4-2
4.5.1 Tag Field Inside A Variant 4-2
4.5.2 Large Range Of Cases 4-3
4.6 SILLY DECLARATIONS 4-3
4.7 IMPORT / EXPORT ISSUES 4-3
4.7.1 Status Of "module_name . Imported_from" Syntax? 4-3
4.7.2 WALKER 4-3
4.7.3 CROSS (Phebus) 4-3
4.8 DYNAMIC ALLOCATION ISSUES 4-4
4.8.1 Allocation Of Objects With Holes 4-4
4.8.2 No Type Check Of Allocate Size Variable 4-4
4.9 PROCEDURE INTERFACE ISSUES 4-4
4.9.1 Confusing N On Multiple Flexible Arrays 4-4
4.9.2 OUT Flexible Array - What Is It? 4-4
4.10 SUGGESTIONS FOR BETTER DEBUGGING 4-4
4.10.1 Detection Of Comparisons Of Objects With Holes . 4-5
4.10.2 Source Lines Overs 132 Characters In Length . . 4-5
4.10.3 More Accurate Line Number Of Errors 4-5
4.10.4 Line Numbers In M11 And Listing Files 4-5

4.10.5 Create_Unique_Global_Symbols Compiler Directive 4-5
4.10.6 Diagnostice When SIZEOF Result Will Not Fit. . . 4-5
4.11 RUN TIME SUPPORT OF BETTER DIAGNOSTICS 4-6
4.11.1 Run Time Line Numbers 4-6
4.11.2 Interface To Version 4.0 Vax Debugger 4-6
4.12 OPTIMIZATION IDEAS 4-6
4.12.1 Array Referencing On LSI-11 4-6
4.12.2 Initial Conditions Of Arrays, Structures 4-6
4.12.3 Register Allocation 4-6
4.12.4 CG_Table Pattern Improvements 4-6
4.12.5 Library Functions For String Operations 4-6
4.13 VMS VERSION 4.0 ISSUES: 4-7
4.13.1 Module Name Limit Of 9 Characters 4-7
4.14 SUPPORT ISSUES 4-7
4.14.1 32 Bit Integer In TEXTIO 4-7
4.14.2 Holes In The Compiler 4-7
4.14.3 Example Programs 4-7
4.14.4 User Analyzer 4-7

CHAPTER 5 PREVIOUSLY FIXED BUGS

5.1 CONVERSION FUNCTIONS 5-1
5.2 INTEGER OF REAL NUMBER 5-1
5.3 PROCEDURE/FUNCTION ISSUES 5-1
5.3.1 OUT Or INOUT By VAL On PDP-11 5-1
5.3.2 RETURN S:Structure Or Array 5-2
5.4 ASSIGNMENT OF IMPORTED STRINGS 5-2
5.5 SELECT ON OUT-OF-RANGE ENUMERATION 5-2

Acknowledgement

We wish to acknowledge the valuable consulting assistance of James M. Duffy (C. G. A. Computer Inc., 1011 East Touhy Avenue, Des Plaines, Illinois, 60018 , Phone:(312) 296-9660) during the beginning and organizational phase of this improvement to the Praxis compiler.

Also, Christian Visconti of the Phebus Project was very helpful in the separation of Praxis from several other software products, and the organization of the new product accounts.

References

A complete list of references to the reports on Praxis is contained within the "Praxis Language Environment Distribution Package Description," UCID-30196 as revised January 1985.

CHAPTER 1

INTRODUCTIONS

1.1 PURPOSE OF THESE NOTES

These release notes are intended as a guide to those responsible for Nova software. They assume extensive knowledge of the present Praxis language.

Each improvement made in Praxis is described. The descriptions are organized as shown in the table of contents. Many of the improvements have example programs which are contained within the chapter on example programs. For completeness, details are also included on the specific areas within the compiler which were modified. These will only be useful to those working on the compiler itself.

1.2 ORGANIZATIONAL CHANGES TO ACCOUNTS

1.2.1 Removal Of Non-Dependent Software

Over time and for development convience, considerable software had accumulated within the Praxis account which Praxis as a language does not depend upon. Four major sets of this software are now mature enough to be considered as independent products and have therefore been moved out of the Praxis account and into their own accounts.

1.2.2 Separate Products For TEXTIO, OSI, RMS, And Conversion Routines

All the software related to the TEXTIO, Operating System Interface, RMS files interface, and the conversion routines were moved into separate Nova-standard 3-level products as shown in figure 1. Only the SYS level of these new products was loaded. They were loaded from the directories and levels shown in the figure.

There was some confusion regarding where to get the best versions of these products because various people have made modifications of them while they were buried within Praxis at the 2 levels that Praxis existed on our on-line disks and within one or more of the 7 (seven!) versions of Praxis on the off-line Praxis disk!

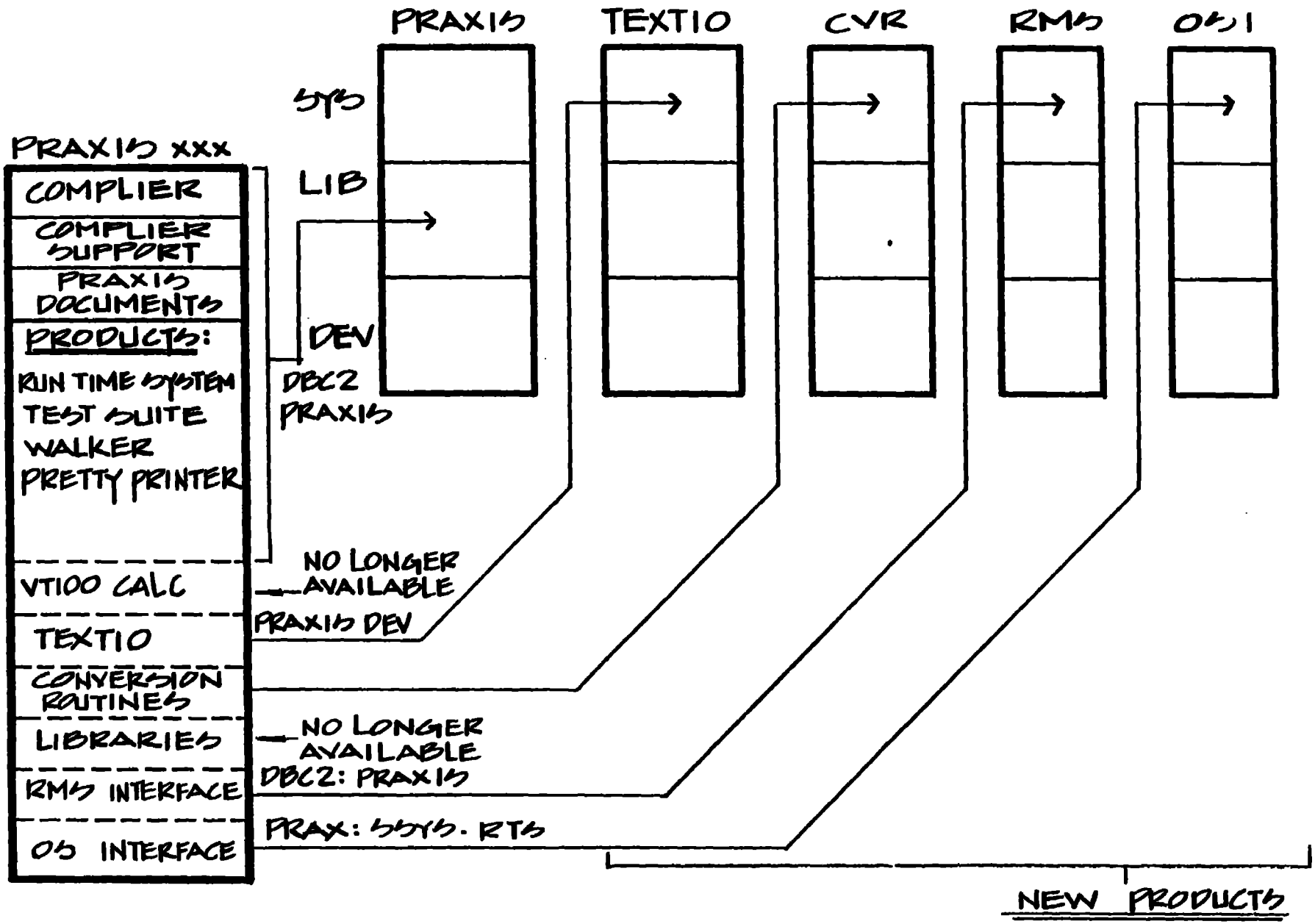

Figure 1: Organization and Derivation of New Products

1.2.3 Praxis Account

The Praxis account now contains only those tools and products that are needed to support the compilers and the langauage. These include the run-time support, manuals, debugging aids, etc.

The Praxis run-time libraries contain routines that implement, in an efficient fashion, certain features of the language. Calls to these routines are generated by the compilers. The details of the routines are instruction-set as well as operating system dependent. Therefore there are versions for VAX/VMS and each of three PDP-11 operating systems (RSX, RT-11 and Stand-Alone).

1.3 INSTRUCTIONS FOR USE

1.3.1 Version 7.2 At The LIB Level

The new version of Praxis (Version 7.2) was built initially at the LIB level of the present 3-level account structure for Praxis. This version contains all the sources of the compiler, so that the need for supporting an off-line disk in parallel is diminished.

1.3.2 New Logical Symbols For Other Products

Logical symbols were set up for the new products in *LOG.COM files. See the following table.

While studying the existing use of logical symbols, we found several that did not appear to meet existing standards, and we recommend that their use be curtailled, but we did not remove them.

At the present time, one must envoke the new logical symbols by executing the appropriate *LOG.COM file, because we did not want to disturb the existing system logical symbols.

```
!-----------------------------------------------------------------
!                LOGICAL SYMBOLS FOR NEW PRODUCTS
!-----------------------------------------------------------------
! TEXTIO:
!
! Pointers to account and TEXTIOLOG.COM file and documentation:
!
!               textio$com:
!               textio$doc:
!
! Prefered symbols for Vax/VMS use:
!
!               txtvms$src:
!               txtvms$sps:
!               txtvms$lib:
!
! Prefered symbols for PDP/RSX or LSI/RSX use:
!
!               txtrsx$src:
!               txtrsx$sps:
!               txtrsx$lib:
!
! Prefered symbols for LSI Stand Alone Integer only use:
!
!               txtsai$sps:
!               txtsai$lib:
!
! Prefered symbols for LSI Stand Alone Real and Integer use:
!
!               txtsar$sps:
!               txtsar$lib:
!
! Following will continue to be supported - apply to VAX/VMS:
!
!               textio$src:
!               textio$lib:
!
! Old symbols still supported, FOR NOW:
!
!               txtvax$sps:
!               txtpdp$sps:
!               txtlsi$sps:
!               txtint$sps:
!               txtint$src:
!               txtlsi$src:
!               txtpdp$src:
!
! Following symbols DO NOT MEET OUR STANDARDS AND ARE DROPPED:
!
!               textio$pdp:
!               textio$int:
!               textio$lsi:
!               textio$vax:
!-----------------------------------------------------------------
```

```
!-------------------------------------------------------------
! OSI (Operating System Interface)  (J. Smart's routines)
!
!       VMS Operating system:
!               OSIVMS$com      Command files
!               OSIVMS$src      Sources
!               OSIVMS$spc      Synopsis
!               OSIVMS$lib      Objects and Libraries
!       RSX Operating system:
!               OSIRSX$com      Command files
!               OSIRSX$src      Sources
!               OSIRSX$spc      Synopsis
!               OSIRSX$lib      object files and libraries
!       RT-11 Operating system:
!               OSIRT$com       Command files
!               OSIRT$src       Sources
!               OSIRT$sps       Synopsis
!               OSIRT$lib       object files and libraries
!-------------------------------------------------------------
!
!  CVR (Conversion Routines)
!               CVR$com         Pointer to CVRLOG.COM file
!               CVR$doc         Documenation
!
!       VMS Operating system:
!               CVRVMS$com      Command files
!               CVRVMS$src      Sources
!               CVRVMS$spc      Synopsis
!               CVRVMS$lib      Objects and Libraries
!       RSX Operating system:
!               CVRRSX$com      Command files
!               CVRRSX$src      Sources
!               CVRRSX$spc      Synopsis
!               CVRRSX$lib      object files and libraries
!       Rebel/Basic Operating system:  (a good reference for LSI-11 code)
!               CVRREB$com      Command files
!               CVRREB$src      Sources
!               CVRREB$sps      Synopsis
!               CVRREB$lib      object files and libraries
!-------------------------------------------------------------
!
!  RMS  (RMS files interface from Praxis, written by BBN)
!
!       VMS Operating system:
!               RMSVMS$com      Command files
!               RMSVMS$src      Sources
!               RMSVMS$spc      Synopsis
!               RMSVMS$lib      Objects and Libraries
!       RSX Operating system:
!               RMSRSX$com      Command files
!               RMSRSX$src      Sources
!               RMSRSX$spc      Synopsis
!               RMSRSX$lib      object files and libraries
!-------------------------------------------------------------
```

1.3.3 Praxis Logical Symbols, Libraries, And Global Symbols

1.3.3.1 Logical Symbols -

Some of the logical symbols for Praxis were modified in order to keep track of the various machine/operating system combinations and the new run time libraries. The PRAXISLOG.COM file contains all the logical symbols as is general practice on the Nova control system. It also contains the redefinition of the symbol USEPRXLIB to point to the correct level of Vax Praxis library.

Of particular note is the addition of 3 logical symbols that point all the way to the library files for support under the various operating systems.

The logical symbols for Praxis are shown in the following table.

LOGICAL SYMBOLS FOR PRAXIS, Version 7.2

Praxis command files:	
PRAXIS$COM	command files (praxislog & prxsps)
VAX Praxis compiler:	
PRXVAX$lib	executable
Praxis Vax Library:	
USEPRXLIB	
VMS Operating system:	
PRXVMS$com	Command files
PRXVMS$src	Sources
PRXVMS$spc	Synopsis
PRXVMS$lib	Objects and Libraries
PRXVMS$library	Library (including library 'name'.olb)
PDP-11 Praxis compiler:	
PRXPDP$lib	executable
RSX Operating system:	
PRXRSX$com	Command files
PRXRSX$src	Sources
PRXRSX$spc	Synopsis
PRXRSX$lib	object files and libraries
PRXRSX$library	library (including library 'name'.olb)
RT-11 Operating system:	
PRXRT$com	Command files
PRXRT$src	Sources
PRXRT$sps	Synopsis
PRXRT$lib	object files and libraries
Stand Alone (no operating sytem) on LSI-11	
PRXSA$com	Command files
PRXSA$src	Sources
PRXSA$sps	Synopsis
PRXSA$lib	object files and libraries
PRXSA$library	Library (including library 'name'.olb)

1.3.3.2 Library Names -

The names and contents of the run-time libraries have changed. As described under the section on logical symbols, new symbols have been added which show the names of the libraries, and can be directly used in link command files. These symbols have also been added to the system for the older version of praxis (7.1).

The library names can be optained by 'show logical PRXooo$Library' where 'ooo' is VMS, RSX, or SA.

1.3.3.3 Global Symbols For Switching Between Versions -

The following system-wide global symbols were established to demonstrate switching from version 7.1 to version 7.2. These symbols can be used directly or traced to their respective command files to get the kernal of the important information from which to write custimized files.

These symbols switch all logical symbols, symbols, and library names so that if a users command files are written to use the library name logical symbols as described above, a complete switch between the versions of the compiler can be accomplished by typing one of these symbols, then recompiling and linking with the normal command files.

PRAXIS_V72	- reassigns all logicals and symbols to point to version 7.2
PRAXIS_V71	- reassigns all logicals and symbols to point to version 7.1
PRAXIS_V	- print version number of Praxis now assigned to

CHAPTER 2

IMPROVEMENTS TO THE PRAXIS COMPILER

Unless otherwise stated, all changes apply to both the PDP-11 and VAX compiler. Most improvements have example programs that were used for testing, and the names of these examples are given.

2.1 DECLARATIONS

2.1.1 Size Of Enumerations

The ability to specify the size of an enumerated type has been added so that one can control the size of packed structures. The syntax is:

```
A is [red, green, blue]
B is 5 bit A
```

See BITENUM.PRX in the chapter of examples.

It is not possible to allow the specification of the size within the statement which defines the enumeration (ie A is 5 bit [red, green, blue]) because that is not within the present bounds of the language parser tables.

2.1.2 Enumerations As Conversion Functions

The ability to use a defined enumeration as a conversion function has been added. See BITENUM.PRX in the chapter of examples.

2.1.3 Easier Way To Declare Strings

To declare an array of strings without having to fill in the entire length of each string:

```
declare
    ErrMsgMax = 60                          // max length of string
    Error_Char is char initially $<SP>
    Error_Type is packed array[1..ErrMsgMax] of Error_Char
    ErrMsgType is array [0..2] of Error_Type
    ErrMsg = ErrMsgType (
        [bad_attribute]:        "not segment attribute.",
        [bad_directive]:        "bad syntax in a compiler directive",
        [bad_general]:          "invalid use of general")
enddeclare
```

This method was found in the praxis compiler itself! IT WAS NEVER MENTIONED IN THE MANUALS, nor discovered by most users.....

2.1.4 Compile-time Constants

Any expresion, including those composed of elements of constant array constructors or constant structure constructors, that can be resolved as an integer, cardinal, or logical constant at compile time is now compiled to a literal constant in the code. Older versions of the compiler would do much of the work of resolving expressions of constants at run time. This improvement results in a saving of space and speed at run time.

2.1.5 Array Constructor Improvements

2.1.5.1 Arrays Of 1 Character Strings -

Initialization of arrays of 1 character strings by either table or array constructors now works properly. In previous versions after each string was stored in memory (P_data) the memory address was rounded up to an even word boundary, which did not match the storage rules for packed arrays of items of less than a word in size. This was changed to round up only for strings larger than 2 in length, which is compatible with the indexing and addressing rules for objects. See ONECHAR.PRX in the chapter of examples.

2.1.5.2 Multiple Dimensioned Array Run Time Constructors -

Previous versions of the compiler generated erroneous data for multiple dimensioned array run time constructors. This was reported by the Phebus project, and has been fixed in this version of the compiler.

Note that array constructors of packed arrays of any size integer (ie with holes) now work correctly also. A good example of the various types of array constructor and initially constructs is shown in TARCONST.PRX (see chapter of examples), which was developed as a through test of this concept. Also see ARCONST3.PRX which show 3 dimensional constructors.

Details: The problem was caused by 2 errors in create_array in WALKEXP.PRX. Indexes of types which were arrays themselves were incorrectly calculated. Then after creating temparary working space on the stack, and filling the temporary space with the appropriate values, a "pop assign" operation was performed with the wrong type of CCode operand. See comments in "temp_output" procedure in WALKSUB.

The CG Pattern for "X pop assign" is tricky. It is not correct for the case where the source operand increments or decrements the stack and the destination operand depends upon the stack.

This coding error was probably a direct result of poor internal documentation within the compiler. Someone just forget the proper operand procedure to use, and there is little documentation about the purpose or restrictions of most of the procedures or variables within the compiler. We have added documentation within the sources of the compiler in those areas which were modified.

2.1.6 Initialized Variant Static Structures

Variants within a statically declared structure with initial conditions caused the structure not be be initialized properly in previous versions of the compiler. This has been fixed. A good example of packing a complex variant structure with several fields all into 1 word is shown in PPACKONE.PRX within the Praxis account test package.

details: Variants with any item in them of size equal to or larger than a word had to be aligned to the next word boundary to meet the packing rules of the language. Also, when initializing any item in a variant that is smaller than the largest item in the variant, extra zeros had to be added to fill out to the size of the largest item in the variant. The SC procudure in WALKCON was approximately 40% rewritten to do this.

2.1.7 Structures And Other Objects With Holes

2.1.7.1 Compare Operations On Objects With Holes -

Praxis provides the ability to efficiently compare (=, <>) two objects of the same type even though the types may be complex and require large amounts of memory. For speed optimization the two objects are compared word-by-word in a tight loop. However, due to the rules of packing the parts of a large object, many such objects have holes in them (ie. bits or bytes which are inaccessible to the user program). Comparisons of these objects may give inaccurate results if the contents of the holes are not carefully controlled.

2.1.7.1.1 Strings With Odd Number Of Characters -

The most common example of this is a packed array of an odd number of characters which packs 2 characters per word and leaves a hole which is one character in size in the last word. These comparisons now give correct results in version 7.2.

2.1.7.2 The Problem Of Control Of Holes -

Control of the contents of holes can be accomplished at the point of declaration of the object. The previous versions of the compiler (prior to 7.2) accomplished this for permanent (static) variables, constants, and in some cases temporary (dynamic) variables with initially clauses, however it did not properly handle dynamic variables without initial conditions.

The initialization of permanent (STATIC) objects is done by the loading the defined parts of the object over a field of zeros. Temporary (dynamic) objects however were created incorrectly by taking the required length on the machine stack and writing any intial conditions into the defined parts of the object. Undefined parts (holes) were left with whatever junk remained from some previous programs use of the stack. Thereafter, compares of these objects were unpredictable. The problem grows deeper when one considers copying a dynamic object with holes into a static object with holes. Now, comparisons of the static objects also would give unpredictable results.

The above problem with odd length charater strings is an example of a minor visible problem leading to a major flaw. It is suspected that the problem of holes in structures may explain some mysterious results we have had with Praxis. By trial and error, we learned to declare nearly everything static even though such declarations may use more memory, they worked. It is now clear that holes were not a problem with static declarations.

2.1.7.2.1 Possible Solutions -

Two solutions to this problem were identified:

1. Change the code generated for compare operations to only compare the declared parts of an object with holes.

2. Clear the holes of dynamic objects whenever they are declared.

Solution #2 was chosen because it would have less effect on run time efficiency, and it has some advantage in detail debugging sessions when one is studying the contents of the stack. Also, solution #1 may not be sufficient in that there may be other places where junk left in a hole would lead to mysterious results.

2.1.7.2.2 Implementation -

To implement the solution, an internal function was written within the compiler which can detect holes in any (however complex) object. A side benefit of this effort was the discovery of the exact rules for packing which had been incompletely described in the manual, and are now described in a separate section below.

The following language features may generate dynamic objects:

1. Declaration block

2. Envoking procedures or functions which expect out val parameters

3. Function return variables

4. Run time constructors for structures, arrays, and sets.

Code was added to the compiler in its handling of each of the syntax's to perform the following operations:

```
if dynamic_declaration do
    if "clear_dynamic_holy_type" do
        if holes do
            if "warn_dynamic_holy_type" do
                    issue appropriate diagnostic
            endif
            get space for object on stack
            clear space for object
            store initial conditions into object
        otherwise
            get space for object on stack
            store initial conditions into object
        endif
    otherwise
        get space for object on stack              // same as versions
        store initial conditions into object       // prior to 7.2
    endif
endif
```

2.1.7.3 Compiler Directives -

2.1.7.3.1 Clear_dynamic_Holy_Types Compiler Directive -

For backward compabiblity a compiler directive was added ("clear_dynamic_holy_types") with a default of true. This can be set false for any module that does not do comparisons of dynamic objects with holes and thereby save the code space and time required to clear these objects during

their declaration. Note however that no diagnostic will be given if a compare of these objects does exist or is added later.

2.1.7.3.2 Warn_Dynamic_Holy_Types Compiler Directive -

For optimization, it is better to declare objects without holes in them. A second compiler directive ("warn_dynamic_holy_types") was added with a default of false. By setting this directive true, one can search for dynamic objects with holes and redefined them.

Details: New compiler directives were added to the OPTION and OPTNAM modules. New diagnostics were added to the ERROR and ERRMSG modules. A declared_size function was added to the SIZE module. Logic to detect holes and deal with new compiler directives was added to the CG_routine, formal_push, and create_space procedures in the WALKEXP module, and in the dynamic_storage case of the CG_State procedure in the WALK module. A "multiple push of 0's" pattern was added to the CGTABLES module for both the VAX and the PDP11.

2.1.7.4 Object With Holes Within The Compiler -

The recognition and solutions to the problem of dynamic holes lead to the location of several potential bugs within the compiler itself, since it would no longer compile with the "warn__dynamic__holy__type" directive set true. (ie therefore, it is suspected that the earlier versions were not handling some things correctly). Changes were made to the following rotines so they would pass this new diagnostic.

The make_operand function in MCODE used a 3 character string for register naming. Then it proceeds to compare these strings. We Nova engineers all know that this didn't work reliably in previous versions of the compiler. These names were changed to 4 characters so the would run faster. The new version of the compiler would handle them correctly but a little slower because of having to clear the arrays on the stack.

Numerous other modules deal with the OPERAND structure, which has holes in it. No further attempt has been made to identify possible bugs in earlier versions of the compiler due to improper handling of objects with holes.

2.1.8 Rules And Effects Of Packing

The rules for packing have always been veiled in mystery. The rules are:

1. Only arrays or structures may be specified as packed, and the specification applies only to the next level down within the array or structure.

2. Any item longer than 16 bits must start on a 16 bit word boundary and take up an integral number of 16 bit words. Any item in a packed array or structure which will not fit within what remains of a word after previous items are

assigned, will be assigned to the next word, possibly leaving a hole.

3. The packing rules are consistent between the Vax and the PDP, so that structures defined on one are the same on the other.

Note: Be careful of packed structures. They can cause much more code space to be generated due to do all the shifting and masking necessary, than is saved in the data space.

Also Note: Unless the user has specifically declared the size of enumerations (see section on Size of Enumerations) in packed arrays or structures they take up only the actual number of bits needed to express the range of the enumeration. Then extra code is needed to mask these bits during accesses.

2.1.9 Table Constructor Size Checking

The size of Table constructors are now checked as the default on both the PDP and VAX compiler. The previous versions of Praxis always warned if there were too many element in a table, but only warned about too few if the module had %set Warn_Table_Size = true. This was changed such that Praxis always warns of either too many or too few items in a table, but the warning about too few can be disabled by %set warn_table_size = false. This now matches the design of the language and the purpose of the table constructor. { It should be noted that the compiler, itself, must be compiled with the switch set to false, due to the compiler implementors' method of using tables..}

2.2 MATHEMATICAL STATEMENTS

2.2.1 32 Bit Integers On The LSI-11

The implementation of 32 bit integers on the PDP was incomplete in previous versions of compiler. This caused the following problems which have been fixed.

2.2.1.1 Assignment Of Constants -

Details: by masking out the high and low parts of the literal in procedure Read_Ccode in CGACTION for the ccode_LO_ccode case and the ccode_HI_ccode case respectively. The mask is done only if the mode is auto_inc and the register is PC (ie immediate constant).

2.2.1.2 Comparisons (<=, <, >, >=) -

Details: Some forms of comparisons of 32 bit integers (<=, >=, <, >) were never implemented in the code generator tables (CGTABLES.DAT) for the LSI-11.

The missing forms were added so that all forms of 32 bit integer comparisons now work (=, <>, <=, >=, >, <). The PDP11 opcodes (BHI and BCS) which are required for correct 32 bit compares were not in the available list of opcodes for the compiler, so they were added to the TABLET module.

2.2.1.3 Sizeof -

The sizeof(1234567) now works correctly.

2.2.1.4 For Loops -

Both upward for loops (for I := 1234567 to 1234568 do) and downward for loops (for I := 1234567 downto 1234555 do) now work correctly. Note however that since these FOR statements uses 32 bit math they will run slower than 16 bit math versions. (ie FOR loops should never be used for timing anyway..)

Also, the design of the language is that the indexing variable takes on the type and precision of the FROM operand. There is no diagnositic at present to indicate that the FROM operand is of less precision than the TO operand. So for example,

FOR I := 1 to 1234568 will compile but will not assemble correctly. This same operation can be achieved correctly by:

```
declare (J : 32 bit integer initially 1)
FOR I := J to 1234568 do
```

Details: The declaration of "target_long_integer_size = 32" was added to to TARGET. Added dcl_long_integer to NAMES, and initialized in SEMINIT

2.2.1.5 IN VAL Parameters -

Passing an 8 bit or 16 bit integer to a procedure which expects an IN VAL 32 BIT INTEGER now works correctly.

Details: The convert_word_to_long (cvt_swl) and convert_byte_to_long (cvt_sbl) were added to the code generator tables.

2.2.2 <1 To 15> Bit Arrays On LSI-11

Packed arrays of integer, logical, or cardinal type of any precision from 1 bit to 15 bit are now handled correctly.

1 bit, 2 bit, 4 bit, 5 bit integer arrays may have some space advantage if large number of elements are required and they have sufficient precision for a particular application. The code generated for accessing these arrays is rather long however, so one should do the actual array indexing operation in as few places as possible.

Caution: One must be smart about declaring multi-dimensional packed arrays of unusual field size. The code will work, but if the least significant dimension doesn't fit evenly in one word, the code will be extremely inefficient (but still correct) and unfortunately, the same amount of space will be allocated for the data as if the whole word was used (this is a bug that may not be worth fixing). Its best to stick with powers of 2 for the ranges of dimensions of unusual field sizes.

Details: The fix for this bug required modification of the procedures for general assignment of fields [VRass in CGSPECIAL] and general loading of fields [Vload in CGSPECIAL].

Note: There is a bug in the register allocation which was patched in a non-general fashion to fix this problem. In the future, register allocation should be studied - its not clear how its done or if there may be other errors.

2.2.3 Multiply, Divide, Mod Of Small Sized Integers And Cardinals

The previous version of the compiler did not implement multiply, divide or mod (ie remainder) of integers or cardinals of 8 bits or less in size on the LSI-11. On the Vax, multiply or mod of cardinals of 8 bits or less in size, and divide of cardinals of 16 bits or less in size was not implemented. These have all been implemented in this version.

2.2.4 Logical Right Shift

Right shift of a logical variable does a sign extended shift. This is in fact a bug but it works just fine. The user can do a mask operation afterword. In fact, thats what is done inside the compiler code! Lets just mark up the language reference manual.

2.3 EXCEPTIONS

2.3.1 Exceptions In Detached Processes

An exception that was raised and not caught in a detached process failed to write the exeption message and name into the error file. This caused much grief looking for a problem of running out of heap space. This has been corrected.

Details: PRXLIB.MAR was modified to Create the error file instead of attempting to open it. Multiple problems existed in the previous open statement. The open was coded for read only, not write. Also, if the same file has already been

opened by the users use of TEXTIO, the PRXLIB module cannot open it because it doesnt have access to the fab and rab blocks. Also, opening a closed file might be opening a file that was written during a previous run of the process. The safest thing to do is create a new file. Also the file must be closed prior to doing the breakpoint trap so that the Debugger can open it again and write the breakpoint data into it.

2.3.2 Guard Block Alignment

The data tables generated to handle guard blocks and exceptions were not word aligned in previous versions of the compiler. Therefore if they happend to be coded such that they followed any constant data (P_data) that was not an even number of words, the resulting tables would have been misaligned. This was fixed.

2.3.3 Rangechecking

The concept of rangechecking of variables and indexes into arrays is a powerful debugging aid which to this date has not be used in the Nova control system because it didnt work in practical situations. With the following improvements it is expected that rangechecking will become a practical debugging tool.

2.3.3.1 Compile Time Range Checking On Assignments -

Assignment of constants, obtained from any object, to an integer or cardinal variable with declared ranges is now checked at compile time. Previous versions of the compiler only did range checking on very simple statements. Version 7.2 implements rangechecking even when the constant being assigned is coming from a field of an array constant or a field of a structure, even multilevel structures.

Note however that 32 bit cardinals are not range checked either at compile time or run time because at the present unsigned comparisons are not implemented.

2.3.3.1.1 No Range Check On Large Object Assignments -

No compile or run time rangechecking occurs on assignments of a object greater in size than one word to another like object. Such objects are generated as arrays or structures. These assignments are made in a tight word-by-word copy operation which ignores the data within the words.

2.3.3.2 Rangechecking Of Flexible Arrays -

Rangechecking of the index into flexible arrays is now supported. This was the problem that had stopped all meaningful use of rangechecking.

Details: The fix for this was to consider the low of a formal (flexible) array to be a descrete item - which it is.

2.3.3.3 ARM X_RANGE_ERROR -

Rangechecking is performed in any block where the ARM X_RANGE_ERROR statement occurs. Rangechecking was further divided into 2 kinds so that the space and speed overhead of total rangechecking can be avoided.

2.3.3.4 Rangechecking Compiler Directives -

Compiler directives can be used to select either (or both) of the following when X_RANGE_ERROR is armed:

a. Subscript_check - array references

b. Subrange_check - every assignment

By default, when the compiler is envoked, both of these are selected and the only thing the user needs to do is ARM X_RANGE_ERROR

See TESTARM.PRX in the chapter of examples.

2.3.3.5 DISARM Explanation -

DISARM now works - and always did work! We just didnt understand it, and the manual is wrong! Either Arm or Disarm (one but not both) can only be stated once in any block of code and it controls what happens over the scope of the entire block of code. One can always add block - endblock statements to control when exception checking is on or off at a finer level.

2.3.3.6 Global Symbols For Exceptions On PDP11 -

The symbolic global symbols output by the compiler for exceptions on the PDP11 were shortened to be unique within 6 characters to match the linker restrictions. The programmer still uses the names as described in the manual.

2.3.3.7 RANGE Statement Modification -

In earlier versions of the compiler, the RANGE statement did some sneeky things with the size of the variable that the user was unaware of. The stated range would override the stated or default precision of an integer or a cardinal. Then the size determined would be modified to fit the packing rules. For example, "A is 16 bit integer range 1..7" within a packed structure would make A 4 bits in size! In an unpacked structure "A" would be 8 bits in size. This seriously restricted the value of range checking in actual applications because the size of structures would change based upon the range of the variable. And even worse, the user did not know of the change.

RANGE statements now only determine the size of integer or cardinal variables which had not had its size specifically set during declaration. Of course the packing rules are still applied. The example programs SIZES.PRX and TSIZE.PRX were coded and give the following results:

	packed size	unpacked size
predefined integers:		
A is 1..2	3	8
A is integer range 1..2	3	8
user defined integers:		
A is 5 bit integer range 1..2	5	8
A is 16 bit integer range 1..2	16	16
A is 32 bit integer range 1..2	32	32

CAUTION: This change may cause the size and layout of a user's structures to be changed. It was decided not to change the synopsis version which would force everyone to recompile every module imported from, but this may cause some difficulty. It seems justified because it is not believed that anyone has used the user defined integers with range restrictions. At least they were not of any practical use because in the past range checking in general didnt work.

2.3.3.8 Some Subrange Statements Crashed The Compiler -

Some subrange statements caused the compilers before version 7.2 to infinite loop. This has been fixed. See SUBRANGE.PRX.

2.3.4 Alternative To Integer Overflow Exceptions

It is recommended that range checking be employed as an alternative to integer overflow exceptions. Overflow exceptions should not be used unless further work on the compiler makes it dependable. Range checking works better and is more tunable.

Overflow testing in the present compiler may not be correct in some situations. Testing of overflows on a PDP-11 or a VAX within a language is very tricky, because the overflow flags are destroyed upon the next instruction. Many times in the compiler there are other instructions that must be executed before it is convienent to test the overflow.

Some minor improvements were made within version 7.2 before the futilty of complete overflow testing was realized. Previous versions of the compiler did not check for integer overflow on a self_assignment statement (ie. I *= +3). This was fixed. Note that overflow checking occurs only when X_OVERFLOW is ARMed.

2.4 ALLOCATE AND FREE

A number of improvements were made in allocation and the use of allocated variables. See the example ALLOCATEX.PRX in the chapter of examples.

2.4.1 One Word Object With Initial Value On LSI-11

Allocation of 1 word objects with an initial value on the LSI-11 now works.

2.4.2 Free Of Large Items

Free of a large item (over 100 bytes) on the VAX caused a breakpoint trap in the previous version of the compilers runtime support package. Also, it is suspected that allocation of large blocks did not function consistently.

This has been fixed. It is important for efficiency to note that allocate and free of small items is handled in a very fast manner within the Praxis runtime support package by getting large blocks as needed from VMS then dividing them into pieces for small blocks as needed by the user. All the pieces are threaded transparently to the user. If a piece is freed, it is available for the next allocate but is not returned to VMS. Large blocks are now allocated and freed by direct calls to VMS for each operation. This portion was not coded in previous versions of the runtime support package. One would assume that allocation and free of larger blocks is much slower than small blocks which are optimized within the Praxis run time support package, but this has not been quantified. In this version of Praxis, the boundary between 'small' and 'large' blocks is 200 bytes. This was increased from the previous versions boundary of 100 bytes.

Note that the "X_heap_empty" exception does occur when a process attempts to allocate more space than its quota is set for. The space in the heap comes from the process virtual space.

2.4.3 Protection From Use Of Nil Pointers

The X_NIL_POINTER exception can be ARMed to check for accesses through nil pointers. This worked correctly on the previous version (7.1) of the compiler for the LSI-11 stand-alone, but was not suportted on the Vax. It has now been added to the run time support for the Vax, and works correctly. See XNILPOINT.PRX.

2.5 FLEXIBLE ARRAYS

Extensive additions were made in version 7.2 in handling flexible arrays (ie arrays of unknown size at compile time).

2.5.1 Allocation Of Flexible Arrays On The LSI-11

Support for allocation of both initialized and non-initialized flexible arrays from a Heap on the LSI-11 was added.

2.5.2 Assignment Of Flexible Arrays

Flexible arrays can now be assigned as single statments either to another flexible array of the same length or to a fixed length array of the same length.

2.5.3 Comparison Of Flexible Arrays

Version 7.2 implements the ability to compare flexible arrays. If the arrays are of unequal length or do not compare in every element, they are not equal.

2.5.4 HIGH And SIZEOF Functions Of Flexible Arrays

The HIGH and SIZEOF function now generate code when necessary to determine the highest index or size of a flexible array at run time.

2.5.5 Local Variable Declared For High Of Flexible Array

In explaination of the syntax of flexible arrays, the 'NN' of 'in ref packed array [1..?NN] of..' is actually a locally declared variable that can be modified if desired. For example, it can be decremented to take out trailing blanks of a text line.

2.6 RUN TIME SUPPORT, RSX AND STAND-ALONE LSI-11

The run time support for RSX and stand-alone LSI-11 systems has been extensively modified and the documentation has been improved. Separate command files, modules, and libraries now exist for RSX, Stand-alone (SA) versions and for a special version for the Power Conditioning system.

The following improvements were made:

2.6.1 Heap Support

Dynamic memory allocation on a heap is now provided. The amount of heap space to be provided in the LSI-11 is set by an instruction to the linker (for example "EXTSCT=$HEAP:2048").

2.6.2 Raising And Catching Exceptions

Note that one must catch every possible exception within any interrupt routine (ie a default case is suggested), or the results of the stack unwind are unpredictable.

Details: the external names of exceptions and faults on the LSI-11 were renamed so that they are unique within the 6 character limit of the Macro-Assembler and Linker. The logic for catching exceptions was cleaned up and documented. The ability to end in a controlled manner with diagnostic printout when an exception is raised and not caught was added. The saving and restoring of the processor status word in handling exceptions was added.

2.6.3 Hardware Exceptions

Guard block catching of the hardware exceptions on the LSI-11 was added. The following hardware exceptions are now handled correctly:

```
X_Stack_Full
X_Machine_Instruction
X_Machine_address
X_Real_Zero_Divide
X_Real_Overflow
X_Real_Underflow
X_Real_Uninitialized
X_Range_Error
X_Failed_Assertion
X_Nil_Pointer
X_Bad_Pointer
X_Overflow              (overflow testing not reliable)
X_Value_Not_in_Set      (but sets dont work)
X_Heap_Empty
X_Failed_Free
```

2.6.4 Boot Time Diagnostics For Stand-alone LSI-11

A simple (skeleton) diagnostic routine for the stand-alone LSI-11 to be run at boot time was added to the library. It only works on un-mapped LSI-11's at present.

2.6.5 Segment Control On The LSI-11

It was discovered that the segment attribute works well in controlling the location of both variable and entire procedures. See SEGTEST.PRX in the of examples chapter

2.7 IMPORT/EXPORT IMPROVEMENTS

2.7.1 Initialization Of Static Imported Types

Large, static objects which are declared of a type which is imported, and has initial conditions within the imported type, now are declared with the correct initial conditions. The use of Array constructors and structure constructors within the type definition now works. Dynamic (ie not static) objects of this sort always did work, and still do.

2.7.1.1 USE Statement Must Be Used -

Note however, that a "use" statement must be used for importing everything from the module with the declared type with initial conditions. There are items which are not specifically "exported" and therefore cannot be listed within an "import" statement, yet are available to the "use" statement. See IMPORTSTR.PRX, EXPORTSTR.PRX

details: A comparison of the field name of a constructor and the list of field names needed within the CST_FIELD procedure of WALKCON.PRX was coded as a compare of the pointers. This was changed to a compare of the names pointed at by the pointers. For some unclear reason the pointers for constructors that are imported do not match the pointers within the list of fields yet they do for items that are completely contained within one module.

Note that this problem was always thought to be based upon the incomplete export or import of initialized types, but instead appears to have been caused by a rather insignificant coding bug.

2.7.2 Encode External Names

In previous versions of the compiler, the "encode external names" switch for PDP-11 code occasionally encoded names improperly.

The problem was that implicitly exported enumeration types caused the encoded names to be skewed. That is, an enumeration which was not explicitly exported but which was incorperated in another type which was exported. A bug caused code generation for the the exporting module to be based upon the error that it had put one more item in the synopsis than it actually had. Depending on the order in which things were declared in the module this bug may cause an importing module to reference the wrong object, or it may cause undefined symbols at link time.

Note: while encode_external_names on the vax is not necessary it can be used and does encode the names. The above problem existed on both versions and has now been fixed on both version.

Details: Modified synout.prx. Added additional logic to procedure export_object to handle case of implicitly exported enumerations correctly.

2.8 PROCEDURE/FUNCTION INTERFACE IMPROVEMENTS

2.8.1 Clarification Of Size Check On Ref Parameters

INOUT and OUT REF parameters must match the size of the formal statement, but IN REF parameters which do not match the size of the Formal parameter are converted to the proper size by pushing them onto the stack then passing the address of the copy on the stack. This is another example of the compiler not doing what the user expects, but it is too ingrained in our present code to change it now. When debugging this type of argument, one must remember that the parameter is just a copy.

2.8.2 Procedure Parameter Combinations

Formal parameters have many possible variations. Many of the possible variations were not supported completely (or correctly) and this led to confusion by the users in earlier versions of the compiler. This was particularly true for non-optional parameters with initially clauses, and OUT parameters.

A through study of all combinations of the following fields of the parameter specification was performed, and diagnostics were added on the combinations that were not fully implemented or did not make sense. The fields for the study were:

Possible values of each field:

Field 1	Field2	Field3	Field4
OPTIONAL (required)	IN INOUT OUT	REF VAL	INITIALLY (no initially)

The number of combinations is 2 * 3 * 2 * 2 = 24.

2.8.2.1 General Rules For Parameters -

1. The defaults are IN VAL

2. OUT and INOUT called with a constant is illegal

3. Non-optional IN and INOUT cannot have initially

4. VAL GENERAL and VAL GENERAL_WITH_SIZE can only be IN

5. For multilevel calls, of which there are (22 x 22) possible combinations, it is assumed that a procedure with a INOUT or OUT formal parameter will modify the corresponding actual parameter. Therefore it is illegal for a procedure which recieves an argument as IN, to call a secondary procedure which has a corresponding INOUT or OUT formal parameter.

6. It is EXTREMELY poor practice to write a function with anything but IN parameters. Everything out of a function should be in the return argument. This is not enforced however by the compiler.

7. Anything can be returned as the RETURN of a function, including an array or a structure.

2.8.2.2 Unworkable Parameter Combinations -

The previous version of the compiler did not implement nor did it give a diagnostic warning on the use of several combinations of parameter qualifiers. Specifically the INITIALLY clause of the following parameter statement qualifiers had no effect upon the operation of the program:

	IN	REF	INITIALLY
	INOUT	REF	INITIALLY
	OUT	REF	INITIALLY
	IN	VAL	INITIALLY
	INOUT	VAL	INITIALLY
OPTIONAL	OUT	REF	INITIALLY

For compatibility with existing control system code which has used these combinations even though the "initially" clause had no effect, implementation of these combinations cannot be completed within version 7.2 of the compiler. However, a diagnostic has been added to version 7.2 (INITIALLY works only on OPTIONAL or OUT VAL) to signal the use of these combinations. With this diagnostic, code will be clearer as to its actual function since unimplemented combinations will be discouraged in new code. For compatibility with old code, a compiler switch has been added to disable these diagnostics. (%set V72_parameter_conventions=false)

2.8.2.3 All Possible Combinations -

At some time in the future when these combinations are no longer in general use, a new version of the compiler could be released which added the implementation of those of these combinations which make sense.

During the study of parameter combinations, the table of notes on each combination shown on the following page was developed and included herein to assist in the explaination of their operation.

Of particular interest is the unusual connotation of the INITIALLY clause on the OUT VAL INITALLY and OPTIONAL OUT VAL INITIALLY combinations. These combinations use the INITIALLY to assign to the output parameter. This seems like an error in design, and it is suggested that this combination not be used, however, no diagnostic has been added to discourage its use. All other combinations use the INITIALLY as the "default" value for the OPTIONAL case when a parameter is not supplied by the calling procedure.

TABLE OF NOTES ON VARIOUS COMBINATIONS OF PROCEDURE PARAMETERS

	Parameter Combinations				1	2	3	4z	5	6	7
	(see Note descriptions):										
1.	Normal	IN	REF	Nonint			3				
2.	Normal	INOUT	REF	Nonint				4a			
3.	Normal	OUT	REF	Nonint				4a			
4.	Normal	IN	VAL	NoInit						6	
5.	Normal	INOUT	VAL	NoInit				4b			
6.	Normal	OUT	VAL	NoInit				4b			7
7.	Normal	IN	REF	Init	1						
8.	Normal	INOUT	REF	Init	1			4a			
9.	Normal	OUT	REF	Init	1			4a			
10.	Normal	IN	VAL	Init	1						
11.	Normal	INOUT	VAL	Init	1						
12.	Normal	OUT	VAL	Init				4b	5		
13.	Op	IN	REF	NoInit			3				7
14.	Op	INOUT	REF	NoInit				4a			7
15.	Op	OUT	REF	NoInit				4a			7
16.	Op	IN	VAL	NoInit						6	7
17.	Op	INOUT	VAL	NoInit				4b			7
18.	Op	OUT	VAL	NoInit				4b			7
19.	Op	IN	REF	Init		2					
20.	Op	INOUT	REF	Init		2		4a			
21.	Op	OUT	REF	Init	1			4a			
22.	Op	IN	VAL	Init		2				6	
23.	Op	INOUT	VAL	Init		2		4b			
24.	Op	OUT	VAL	Init				4b	5		

Notes Descriptions:

1. Initial does nothing, therefore diagnostic added in version 7.2
2. Initial is Default for optional
3. Read only inside procedure
4. Actual parameter modified:
   - 4.a Upon assignment in procedure
   - 4.b Upon exit from procedure
5. If no assignment is made within the procedure, initial cond assigned to argument on exit
6. Procedure uses local copy which can be modified but is not copied out
7. Danger of reading garbage when parameter is not supplied

2.8.3 Functions Returns For Flexible Parameters

A function can now be used to return the value which is to be supplied to another function or a procedure which expects a flexible array. This was not implemented in the previous version of the compiler. An example of this is shown in FUNCT2FLX.PRX in the chapter of examples.

Details: The formal_array cases of the Pre_Argument and Place_Argument functions in WALKEXP were modified to provide for passing non_addressable objects (ie obtained from function calls) to procedures with flexible array arguments.

2.9 BETTER INTERFACE TO VMS AND RMS

2.9.1 Fortran Procedure Parameter Changes

In order to interface in a more general manner with the VMS and RSX operating system interfaces, the details of the parameters of Fortran procedures were changed as follows:

1. VAL parameters are now allowed

2. OPTIONAL GENERAL no longer requires an INITIALLY.

3. OPTIONAL REF parameters when no formal initial value is specified and caller does not supply the matching parameter (ie no actual parameter) now will cause 0 to be pushed on argument stack. Previous versions of the compiler caused the address of 0 to be pushed onto the argument stack (which was an incorrect interface to Fortran.

Details:

```
optional ref parameter_type initially inital_value
  if initial_value = nil
      -- beginning of modification --
      push 0 instead of address of 0 on stack
      for caller unspecified argument
      -- end of modification ----------
  otherwise
      push address of initial value on stack
      for caller unspecified argument
  endif
```

2.9.2 Variadic Parameters

2.9.2.1 Variadic REF GENERAL WITH SIZE -

Previous versions of the compiler produced bad stack arguments for Variadic REF GENERAL WITH SIZE. This has been fixed.

2.9.2.2 Variadic Flexible Arrays -

Previous versions of the compiler did not allow variadic flexible arrays. This was added. Variadic parameters are conceptually like a flexible array of parameters. High of the variadic parameter returns the number of variadics. See XVARIADIC.PRX in the chapter of examples. The following examples show the 'with size' characteristic and other uses.

Example 1:

```
procedure p (g: variadic in ref general with size)
    for i := 1 to high(g) do     // high(g) now works
        s := sizeof(g[i])        // Now gives correct size
    endfor
endprocedure

p("STRING", 123, "ddd") // Now works
```

Example 2:

```
procedure p (s: variadic in ref packed array[1..?n] of char)

    for i := 1 to high(s) do     // High works
        n := high(s[i])          // Gives the number of chars in s[i]
        Out_string (tty, s[i])   // Outputs the string correctly
        Out_record (tty)
    endfor

    //  high(s[i]) gives in turn:
    //  3 ("ALL"), 5 ("THESE"), 7 ("STRINGS"), 3 ("GET"),
    //  6 ("PASSED"), 9 ("CORRECTLY")
endprocedure

p ("ALL", "THESE", "STRINGS", "GET", "PASSED", "CORRECTLY")
```

output:

```
ALL
THESE
STRINGS
GET
PASSED
CORRECTLY
```

2.10 OPTIMIZATION

2.10.1 Constant Expresions Evaluated At Compile Time

All expressions - no matter how complicated - that can be evaluated as a short constant (integer, cardinal, logical) are now evaluated at compile time. This results in remarkable shorter and faster code when elements of constant arrays or constant structures are utilized to define a constant expression that is then used in an assignment or as an index for an array. In the (un-typical) example program OPTIMUM1.PRX (see chapter of examples), the amount of program data space was reduced by a factor of 10 while the number of executable instructions was reduced by a facter of 3.

Previous versions of the compiler would do structure or array accessing to constant structures or arrays at run time. Not only did this require many more instructions, but the entire constant array or structure had to be stored in memory even if only a part of it was used.

2.11 DEBUGGING IMPROVEMENTS

2.11.1 Better Diagnostic Messages

In general, all diagnostic messages should supply some information about the particular variable, routine name, type, etc., that they apply to. Debugging is very discouraging when looking for the reason for a diagnostic within a large module with no such information. Many of the diagnostic messages have been changed to include more such information. To leave more room for diagnostic information the line number part of the diagnostic message was shortened to "Line <= {N}".

The following diagnositics now include information about the label or type in the area listed as {label} symbols.

a. "too many components in table constructor {label}"

b. "too few components in table constructor {label}"

c. "range overlap on case {label}",

d. "multiple defaults in SELECT or CATCH {label}"

e. "type of {label} incompatible with type of {label}"

f. "Attempt to modify read-only object {label}"

e. "Size of {label} not equal to size of {label} on call to {label}"

f. "missing argument in call to {label}"

e. "duplicate or extra argument in call to {label}"

f. "optional parameter {label} requires INITIALLY"

g. "Range check on assign of {label} to {label}"

The following diagnostic messages were added:

a. "{label} is out val parameter with holes"

b. "{label} non-initialized dynamic declare with holes"

c. "{label} constructor with holes"

d. "<">S<"> OUT VAL parameter must have initially"

e. "Statement skiped (halt) due to previously reported error"

f. "Too many user defined compiler directives"

g. "Need initial value on allocate of flexible {label}"

e. "general with ID not implemented"

The following diagnostics were clarified:

a. "{label} non-optional IN parameter cannot have initially"

b. "Code or dynamic decl outside proc, funct, or main module"

Note that this was the old "Illegal construct in non-main module" diagnostic, which was useless because it didn't supply enough information. This error is generated whenever dynamic variables or code is being generated outside of the scope of any procedure, function, or the main module.

2.11.2 Unencoded And Encoded Names In PDP11 M11 File

The unencoded name of a variable is now placed in the listing of PDP11 Praxis modules along with the encoded name. This should be useful for those cases where encode_external_names is employed.

Details: The code to do this was found as conditionally compiled code for the pdp10.

2.11.3 Wider Symbols In Listings

This version of the compilers allow 31 character symbols in the symbol table of the listing. (previously it was 15)

2.12 BATCH AND INTERACTIVE MODE INCONSISTENCIES

It has been reported (by J. Wilkerson) that some modules do not compile correctly in the Batch mode but do at the terminal. We have been unable to repeat this error, but if it did exist it is exactly the kind of thing that could have happened to the compiler due to the holy structure problems solved and described elsewhere in this document. Therefore, unless this is re-reported (hopefully with an example), this bug is considered fixed.

2.13 INTERNAL COMPILER DIAGNOSTICS

2.13.1 Trace Messages

Extensive trace messages have been added to the compiler in order to study its internal operations. These are principally enabled by the debug_print_ccode compiler directive.

2.13.2 Printing The Forest

A Print_Forest procedure was added to print the entire linked chain of compiler trees (nodes) for debugging difficult problems with the internal compiler data structure. The list generated shows the complete contents of every tree from the tree pointed to downward. Indenting is used to show tree recursion.

CHAPTER 3

EXAMPLE PROGRAMS

3.1 SIZE AND CONVERTERS OF ENUMERATIONS

```
//-----------------------------------------------------------------------
// Example of setting the size of an enumeration, and of using the
// enumeration
//-----------------------------------------------------------------------

main module bitenum

    use TEXTIO

    declare
        colors is [red, green, blue, orange, yellow, magenta, black, white,
cyan, tan, pink, violet]

        large_colors is 5 bit colors    // SET SIZE OF ENUMERATION

        color_array:    static packed array[1..10] of large_colors
        color:          static colors
        large_color:    static large_colors
    enddeclare

    for i := 1 to 10 do
        color_array[i] := colors(i)
        out_integer (tty, integer(color_array[i]), " ")
    endfor
    out_record (tty)

    large_color := pink
    out_integer (tty, integer(large_color), " ")

    color := large_colors(magenta)              // CONVERTER
    out_integer (tty, integer(color), " ")

    color := color_array[4]
    out_integer (tty, integer(color), " ")

    color := large_color
    out_integer (tty, integer(color), " ")

    color := pred(large_color)
    out_integer (tty, integer(color), " ")

    large_color := color
    out_integer (tty, integer(large_color), " ")

    large_color := succ(color)
    out_integer (tty, integer(large_color), " ")

    large_color := pred(color)
    out_integer (tty, integer(large_color), " ")
    out_record (tty)
endmodule
```

3.2 ARRAYS OF 1 CHARACTER STRINGS

```
//_Header
//*********************************************************************
//     NOVA CONTROL SYSTEM  ---  Lawrence Livermore Laboratory
//
//     Copyright 1984 by The Regents of the University of California
//*********************************************************************
//
//_Module_Name:  ONECHAR
//
//_Description:
//       This module shows a bug in versions of Praxis previous to 7.2.
//       The Test_Char should
//       be set to an "A" but it was set to a "P" instead.
//       The indexing into the table appeared to be off by a factor
//       of 2, in that the 1st, 3rd, 5th, index values actually
//       pick up the 1st, 2nd, 3rd elements, while the other
//       index values picked up either nothing or at least no
//       recognizable ascii character.
//
//       This program runs correctly under version 7.2
//_Call:
//_Identifier: { none }
//*********************************************************************
//_Author:  F. Holloway          _Creation_Date:  25-JAN-1984
//_Revisions:
//  1.000  25-JAN-1984  FWH     Initial Key-in.
//*********************************************************************
//_End


main module ONECHAR
    use TEXTIO
    declare
        Status_Symbol   is packed array [1..1] of char
        Enums is
                [IS_C,          IS_P,           IS_A,           IS_S]

        Enums_Status is packed array [Enums] of Status_Symbol
        Test_Status = table Enums_Status (
                "C",    "P",    "A",    "S")

         Text_Char : packed array [1..1] of char
    enddeclare

        Text_Char := Test_Status [IS_A]

        out_string      (TTY, "Text_Char =")
        out_string      (TTY,   Text_Char)
        out_record      (TTY)
endmodule
```

3.3 ARRAY CONSTRUCTORS

```
//-----------------------------------------------------------------
// Example of array constructors
//-----------------------------------------------------------------
//
// The following pattern it generated by various array constructor techniques.
// For the purpose of this test, so as to thoroughly test the indexing in the
// contructors, only the items in the middle (not the edges) of the matrix
// is set up by most of the constructor statements.
//
// The 7's are do to the initially of the original Integer_item declaration.
// The "1, 6, 6, 6, 1" rows are due to the initially of the X_array
//
//                              columes 1 to 5
//                              --------------
//                      (       1, 6, 6, 6, 1   ),      // row 1
//                      (       7, 1, 2, 2, 7   ),      // row 2
//                      (       7, 2, 2, 3, 7   ),      // row 3
//                      (       7, 3, 4, 5, 7   ),      // row 4
//                      (       7, 4, 5, 5, 7   ),      // row 5
//                      (       1, 6, 6, 6, 1   ) )     // row 6
//
// The size of the Integer_item, and packing of the array can be editted
// for more interesting tests.
//_Author: J. Duffy / F. Holloway
//-----------------------------------------------------------------

main module TARCONST
    use TEXTIO
declare

        //              v--- edit this for interesting test (4 to 32 is legal)
        Integer_item is 6 bit integer initially 7

        XRANGE is 1..5
        YRANGE is 1..6

        //                 v--- use packed for very demanding test
        X_array         is packed array [XRANGE] of Integer_item


        X               is X_array  initially
                                X_array ( [1]:1, [2..4]:6, [5]:1)
        YX              is array [YRANGE] of X

        CGRID  =
        //              row     column2         column 3        column 4
        //              ---     -----------     -----------     -----------
                YX (    [2] : X ( [2]   : 1,    [3..4]  : 2                 ),
                        [3] : X ( [2..3]: 2,                    [4]     : 3),
                        [4] : X ( [2]   : 3,    [3]     : 4,    [4]     : 5),
                        [5] : X ( [2]   : 4,    [3..4]  : 5                 ))

        GRID    : YX
        GRID_s  : static YX
        GRID_c  : YX
        GRID_cs : static YX
```

```
        GRID_t  : YX
        GRID_ts : static YX

    enddeclare


    procedure out_grid ( G: in ref YX )

        if G <> GRID_ts do      // assume the static table construct is correct
            out_line (TTY, "ERROR in grid comparison")

            for YIndex in YRANGE do
                for XIndex in XRANGE do
                    out_integer (TTY, G [ YIndex, XIndex], "  ")
                endfor
                out_record (TTY)
            endfor
        otherwise
            out_line (TTY, "PASSED")
        endif
    endprocedure


    declare

        //-----------------------------------------------------------------
        // Dynamic grid initialized from inline constructor

        GRID_i  : YX initially
        //              row     column2          column 3         column 4
        //              ---     ----------       ----------       ----------
                YX (    [2] : X ( [2]    : 1,    [3..4]  : 2                ),
                        [3] : X ( [2..3]: 2,                     [4]     : 3),
                        [4] : X ( [2]    : 3,    [3]     : 4,     [4]     : 5),
                        [5] : X ( [2]    : 4,    [3..4]  : 5                ))

        //-----------------------------------------------------------------
        // Static  grid initialized from inline constructor

        GRID_is : static YX initially
        //              row     column2          column 3         column 4
        //              ---     ----------       ----------       ----------
                YX (    [2] : X ( [2]    : 1,    [3..4]  : 2                ),
                        [3] : X ( [2..3]: 2,                     [4]     : 3),
                        [4] : X ( [2]    : 3,    [3]     : 4,     [4]     : 5),
                        [5] : X ( [2]    : 4,    [3..4]  : 5                ))

        //-----------------------------------------------------------------
        // Dynamic grid initialized from constant constructor

        GRID_ic : YX initially CGRID

        //-----------------------------------------------------------------
        // Static grid initialized from constant constructor

        GRID_ics: static YX initially CGRID

enddeclare
```

```
//-----------------------------------------------------------
// Dynamic grid assigned from inline constructor

GRID :=
        //                row     column2          column 3         column 4
        //                ---     -----------      ----------       -----------
                YX (     [2] : X ( [2]    : 1,    [3..4]   : 2                ),
                         [3] : X ( [2..3]: 2,                      [4]      : 3),
                         [4] : X ( [2]    : 3,    [3]      : 4,    [4]      : 5),
                         [5] : X ( [2]    : 4,    [3..4]   : 5                ))

//-----------------------------------------------------------
// Static  grid assigned from inline constructor


GRID_s :=
        //                row     column2          column 3         column 4
        //                ---     -----------      ----------       -----------
                YX (     [2] : X ( [2]    : 1,    [3..4]   : 2                ),
                         [3] : X ( [2..3]: 2,                      [4]      : 3),
                         [4] : X ( [2]    : 3,    [3]      : 4,    [4]      : 5),
                         [5] : X ( [2]    : 4,    [3..4]   : 5                ))

//-----------------------------------------------------------
// Dynamic grid assigned from constant constructor

GRID_c := CGRID

//-----------------------------------------------------------
// Static grid assigned from constant constructor


GRID_cs := CGRID

//-----------------------------------------------------------
// Dynamic grid assigned from inline table constructor

GRID_t := table
//                                 columes 1 to 5
//                                 --------------
        YX (table       X (      1, 6, 6, 6, 1   ),      // row 1
            table       X (      7, 1, 2, 2, 7   ),      // row 2
            table       X (      7, 2, 2, 3, 7   ),      // row 3
            table       X (      7, 3, 4, 5, 7   ),      // row 4
            table       X (      7, 4, 5, 5, 7   ),      // row 5
            table       X (      1, 6, 6, 6, 1   ) )     // row 6

//-----------------------------------------------------------
// Static  grid assigned from inline table constructor

GRID_ts := table
//                                 columes 1 to 5
//                                 --------------
        YX (table       X (      1, 6, 6, 6, 1   ),      // row 1
            table       X (      7, 1, 2, 2, 7   ),      // row 2
            table       X (      7, 2, 2, 3, 7   ),      // row 3
            table       X (      7, 3, 4, 5, 7   ),      // row 4
```

```
            table       X (     7, 4, 5, 5, 7   ),      // row 5
            table       X (     1, 6, 6, 6, 1   ) )     // row 6

//-----------------------------------------------------

tty_line ("test pattern is:")
            for YIndex in YRANGE do
                for XIndex in XRANGE do
                    out_integer (TTY, GRID_ts [ YIndex, XIndex], "  ")
                endfor
                out_record (TTY)
            endfor

tty_line ("dynamic grid assigned from inline constructor")
out_grid( GRID )

tty_line ("static  grid assigned from inline constructor")
out_grid( GRID_s )

tty_line ("dynamic grid assigned from constant constructor")
out_grid( GRID_c )

tty_line ("static grid assigned from constant constructor")
out_grid( GRID_cs )

//-----------------------------------------------------

tty_line ("dynamic grid initialized from inline constructor")
out_grid( GRID_i )

tty_line ("static  grid initialized from inline constructor")
out_grid( GRID_is )

tty_line ("dynamic grid initialized from constant constructor")
out_grid( GRID_ic )

tty_line ("static grid initialized from constant constructor")
out_grid( GRID_ics )

//-----------------------------------------------------

tty_line ("dynamic grid assigned from inline table constructor")
out_grid( GRID_t )

tty_line ("static  grid assigned from inline table constructor")
out_grid( GRID_ts )

endmodule
```

3.4 THREE DIMENSIONAL ARRAY CONSTRUCTORS

```
//-----------------------------------------------------------------------
// Example of 3 dimensional array constructors
//-----------------------------------------------------------------------
main module ARCONST3
use TEXTIO

declare
        ZRANGE is 1..4
        XRANGE is 1..5
        YRANGE is 1..6
        Y       is array [YRANGE] of integer
        XY      is array [XRANGE] of Y
        zxy     is array [zrange] of xy

        Ccage  =table ZXY (
         XY ([1] : Y ( [1..1] : 1, [2..6] : 1),
             [2] : Y ( [1..2] : 2, [3..6] : 1),
             [3] : Y ( [1..3] : 3, [4..6] : 1),
             [4] : Y ( [1..4] : 4, [5..6] : 1),
             [5] : Y ( [1..5] : 5, [6..6] : 1) ),
         XY ([1] : Y ( [1..1] : 1, [2..6] : 2),
             [2] : Y ( [1..2] : 2, [3..6] : 2),
             [3] : Y ( [1..3] : 3, [4..6] : 2),
             [4] : Y ( [1..4] : 4, [5..6] : 2),
             [5] : Y ( [1..5] : 5, [6..6] : 2) ),
         XY ([1] : Y ( [1..1] : 1, [2..6] : 3),
             [2] : Y ( [1..2] : 2, [3..6] : 3),
             [3] : Y ( [1..3] : 3, [4..6] : 3),
             [4] : Y ( [1..4] : 4, [5..6] : 3),
             [5] : Y ( [1..5] : 5, [6..6] : 3) ),
         XY ([1] : Y ( [1..1] : 1, [2..6] : 4),
             [2] : Y ( [1..2] : 2, [3..6] : 4),
             [3] : Y ( [1..3] : 3, [4..6] : 4),
             [4] : Y ( [1..4] : 4, [5..6] : 4),
             [5] : Y ( [1..5] : 5, [6..6] : 4) )        )


        cage    : ZXY
        cage_c  : static ZXY

enddeclare

procedure out_grid ( i: integer, G: in ref XY )
        declare
                tab is array[zrange] of packed array [1..4] of char
                tabs = table tab (
                                "    ",
                                "      ",
                                "            ",
                                "                        ")

        enddeclare
        for XIndex in XRANGE do
            out_string( tty, tabs[i] )
            for YIndex in YRANGE do
                out_integer     (TTY, G [XIndex, YIndex], "  ")
```

```
            endfor
            out_record (TTY)
        endfor
endprocedure

procedure out_cage ( C: in ref ZXY )
        for zindex in zrange do
                out_grid ( zindex, c[zindex] )
                tty_line (" ")
        endfor
endprocedure

cage := table ZXY (
         XY ([1] : Y ( [1..1] : 1, [2..6] : 1),
             [2] : Y ( [1..2] : 2, [3..6] : 1),
             [3] : Y ( [1..3] : 3, [4..6] : 1),
             [4] : Y ( [1..4] : 4, [5..6] : 1),
             [5] : Y ( [1..5] : 5, [6..6] : 1) ),
         XY ([1] : Y ( [1..1] : 1, [2..6] : 2),
             [2] : Y ( [1..2] : 2, [3..6] : 2),
             [3] : Y ( [1..3] : 3, [4..6] : 2),
             [4] : Y ( [1..4] : 4, [5..6] : 2),
             [5] : Y ( [1..5] : 5, [6..6] : 2) ),
         XY ([1] : Y ( [1..1] : 1, [2..6] : 3),
             [2] : Y ( [1..2] : 2, [3..6] : 3),
             [3] : Y ( [1..3] : 3, [4..6] : 3),
             [4] : Y ( [1..4] : 4, [5..6] : 3),
             [5] : Y ( [1..5] : 5, [6..6] : 3) ),
         XY ([1] : Y ( [1..1] : 1, [2..6] : 4),
             [2] : Y ( [1..2] : 2, [3..6] : 4),
             [3] : Y ( [1..3] : 3, [4..6] : 4),
             [4] : Y ( [1..4] : 4, [5..6] : 4),
             [5] : Y ( [1..5] : 5, [6..6] : 4) )        )
cage_c := ccage

//-----------------------------------------------------
tty_line ("-------------------correct-----------------------")

tty_line ("static  cage assigned from constant table constructor")
out_cage( cage_c )

tty_line ("-------------------ERROR-----------------------")
tty_line ("dynamic cage assigned from inline table constructor")
out_cage( cage )


endmodule
```

3.5 ARM AND USE OF RUN-TIME RANGECHECKING

```
//-----------------------------------------------------------------
//                RANGE CHECKING EXAMPLE
//-----------------------------------------------------------------
// Set the compiler directives "subrange_check" and "subscript_check"
// at the top of the module to specify either or both
// types of run time range checking code be generated when
// the "X_Range_Error" is armed.
//
// The compiler directives "subrange_check" and "subscript_check"
// are defaulted to false when compiler is envoked.
//
// The "subscript_check" may be the most useful for programs
// where space is limited because array subscript violations
// are probably the most usual cause of crashing codes.
// "Subrange_check" generates a LOT of code.
//



%set subrange_check = false
%set subscript_check = true

main module testarm

    use TEXTIO
    declare
        I : static 16 bit integer range 1..4 initially 1
        A : static array [1..4] of 16 bit integer
    enddeclare

guard

    I *= +2
    A [i] := 2
    procedure JOIE ()
        ARM X_RANGE_ERROR                // Range checking code only generated
                                         // in scopes where it is armed.
                                         // And, it applies over the entire
                                         // scope so it doesnt matter where
                                         // in the scope the statement is.
        I := I + 1
        A [i] := 2                       // It is illegal to both arm and
                                   // disarm within the same scope.
    endprocedure

    procedure SAM ()
        DISARM X_RANGE_ERROR
        I := I + 2
        procedure PETE ()
            ARM X_RANGE_ERROR
            A [i] := 2             // THIS STATEMENT CAUSE X_RANGE ERROR
            I := 2
        endprocedure

        A [i] := 1
        PETE ()
```

```
    endprocedure

    JOIE ()
    SAM ()

  catch
    case X_RANGE_ERROR:
        out_line (TTY, "Caught you you bugger")
endguard
endmodule
```

3.6 RANGECHECKING OF SUBRANGES

```
//-----------------------------------------------------------------
// Example of subrange checking.
//-----------------------------------------------------------------

main module subrange
        declare
            THis_range is 1..10
            Sub_range is this_range range 1..5
            Item : Sub_range
        enddeclare

        Item := 6       // This is a compile time range error
endmodule
```

3.7 EXPORT OF LARGE STRUCTURE WITH INITIAL CONDITIONS

```
//-----------------------------------------------------------------
// Example of exporting a complicated structure with initial conditions
//-----------------------------------------------------------------

Module EXPORTSTR
        Export A_type, B_type
        Declare
                A_type is
                        Structure
                                A_1: 7 bit integer initially 4
                                A_2: integer
                        EndStructure
                blank_char is char initially $.
                stg_type is packed array [1..3] of blank_char

                B_type is
                        Structure
                            stg : stg_type initially  ("AB")
                            B_1 : 4 bit integer initially 7
                            B_2 : A_type initially A_type ( A_1: 7, A_2: 9 )
                        EndStructure
        EndDeclare
EndModule
```

3.8 IMPORT OF LARGE STRUCTURE WITH INITIAL CONDITIONS

```
//-----------------------------------------------------------------
// example of importing a complicated structure with initial conditions
//-----------------------------------------------------------------
Main Module IMPORTSTR

        use TEXTIO
        Use EXPORTSTR   // Must use a USE statement
//      import B_type, A_type  from EXPORTSTR

        Declare
                C_dyn           : B_type
                C_static        : static B_Type
        EndDeclare

    procedure T ( A : in ref B_type)
        out_string (TTY, A . stg)
        out_string (TTY, " . B_2 . A_1 = ")
        out_integer (TTY, A . B_2 . A_1)
        out_string (TTY, " . B_2 . A_2 = ")
        out_integer (TTY, A . B_2 . A_2)
        out_record (TTY)
    endprocedure

    T (C_static)
    T (C_dyn)
EndModule
```

3.9 ALLOCATION SYNTAXS

```
//-----------------------------------------------------------------
// Example of various forms of allocate syntax
//-----------------------------------------------------------------

main module ALLOCATEX
    use TEXTIO

    // Allocation of an array with an initial condition

    declare
        X_type          is packed array [1..9] of char
        X_pointer       is pointer X_Type
        P_X             : X_Pointer
        P_X_Blank       : X_Pointer
    enddeclare

    P_X := allocate X_pointer ("ALLOCATEX" )
    P_X_Blank    := allocate X_Pointer ()
    P_X_Blank@   := ".PRX      "
    out_string (TTY, P_X@)
    out_string (TTY, P_X_Blank@)
    out_record (TTY)

    // test allocation with initial conditions through a pointer

    declare
        Init_Type is integer initially 77
        Init_pointer is pointer Init_Type
        Init     : Init_Pointer
        Final    : Init_Pointer
        Init_val : Init_Type
    enddeclare

        Init := allocate Init_Pointer (Init_Val)
        Final := allocate Init_Pointer (Init@)

        out_integer (TTY, Final@, "Correct Initial condition = 77 :")
        out_record (TTY)

    // Allocate initialized flexible arrays

    declare
        Flex_array      is packed array [1..?MM] of char
        P_Flex          is pointer Flex_array
        Top1            : static P_Flex
        Top2            :  P_Flex
        Top3            :  P_Flex
        I_Count         : static integer initially 4
    enddeclare
    Top1 := allocate P_Flex ("ABCDE")

    // allocate non-initialized flexible array of fixed size

    Top2 := allocate P_Flex (8)

    // Assign string to array
```

```
    Top2@ := "12345678"

    // allocate non_initialized flex array of variable size

    Top3 := allocate P_Flex (MM:I_Count)

    Top3@ := " WHY"
    out_string (TTY, top1@)
    out_string (TTY, top2@)
    out_string (TTY, top3@)
    out_record (TTY)

    // Note that an "allocate P_Flex (I_Count)" syntax is accepted by the
    // compiler and some older software is written with it in the compiler.
    // It is shown here, but it is to be discouraged in the future.

    Free (Top3)
    Top3 := allocate P_Flex (I_Count)
    Top3@ := " WHY"
    out_string (TTY, top1@)
    out_string (TTY, top2@)
    out_string (TTY, top3@)
    out_record (TTY)


    // high and size of flexible array

    out_integer (TTY, High (Top1@), "High of string = ")
    out_record  (TTY)
    out_integer (TTY, integer(sizeof (Top1@)), "Sizeof string = ")
    out_record (TTY)

    Free (Top1)
    Free (Top2)

    // Test allocation of a structure with initial conditions

    declare
        A is
            structure
                B : integer
            endstructure
        Pointer_To_A    is pointer A
        P       : pointer_to_A
    enddeclare

    P := allocate Pointer_to_A (B : 40)

    out_integer (TTY, P@.B, "Initial value of structure = ")
    out_record (TTY)
endmodule
```

3.10 PROTECTION FOR USE OF NIL POINTERS

```
//-----------------------------------------------------------------
// example of exception that checks for nil pointers
//-----------------------------------------------------------------
main module XNILPOINT
    ARM X_NIL_POINTER

    declare
        A is integer
        P is pointer A
        My_P : static P
    enddeclare

    My_P@ := 4
endmodule
```

3.11 FUNCTIONS RETURNS FOR FLEXIBLE PARAMETERS

```
//-----------------------------------------------------------------
// Example of the use of a function to return an array that is to be
// passed to a procedure which expects a flexible array
//
main module FUNCT2FLX

    use TEXTIO

    // Functions which supply arrays

    function GET_STRING_2 ()
        returns String : packed array [1..2] of char initially "AB"
    endfunction
    function GET_STRING_3 ()
        returns String : packed array [1..3] of char initially "CDE"
    endfunction
    function GET_STRING_12 ()
        returns String : packed array [1..12] of char initially "FGHIJKLMNOPQ"
    endfunction

    // Procedure which expects flexible arrays

    procedure TYPE_IT
        param
            String1: in ref packed array [1..?m] of char
            String2: in ref packed array [1..?n] of char
            String3: optional in ref packed array [1..?p] of char
                        initially "............"
        endparam
        Out_string (TTY, String1 )
        Out_string (TTY, String2 )
        Out_string (TTY, String3 )
        out_record (TTY)
    endprocedure

    TYPE_IT (GET_STRING_2(), GET_STRING_3(), GET_STRING_12() )
    TYPE_IT (GET_STRING_2(), GET_STRING_3()                  )
endmodule
```

3.12 HIGH OF VARIADIC FLEXIBLE ARRAY

```
//-----------------------------------------------------------------
// Example of use of variadic, high of variadic, variadic flexible array
//
main module XVARIADIC
    use TEXTIO

    procedure VAR (a: variadic in ref packed array[1..?n] of char)

        out_String      (tty, "high=")
        out_integer     (tty, high (a))
        out_record      (tty)

        declare (s: static integer)

        for i := 1 to high (a) do
            s := high (a [i])
            out_string (tty, "high=")
            out_integer(tty, s)
            out_string (tty, " ")
            out_string (tty, a[i], s)
            out_Record (tty)
        endfor
    endprocedure

    VAR ("HI", "THERE", "YOU", "ALL")

endmodule
```

3.13 OPTIMIZATION OF CONSTANT EXPRESSIONS

```
//-----------------------------------------------------------------
// Example of Constant Evalutation Optimization added in Version 7.2
//-----------------------------------------------------------------
main module OPTIMUM1
        declare
            // --- multi-level structure
                    C is structure
                            Joe : integer
                    endstructure

                B is structure
                    This_C : C
                endstructure

            A is structure
                This_B : B
            endstructure

            // --- Multi-level structure with array inside

                        a4_type is array[1..5] of integer

                    a3_type is structure
                        a3:     integer
                        a4:     a4_type
                    endstructure

                a2_type is structure
                    a2: a3_type
                endstructure

            a1_type is structure
                a1:     a2_type
            endstructure

            //--- constant multi-level structure with array inside

            a0 = a1_type(a1: a2_type(a2: a3_type(a3: 12345, a4:
                a4_type([1]:123,[2]:234,[3]:7,[4]:5))))

            //--- constant array

            One_D_type is array[1..10] of integer
            Con_1_D = table One_D_type (1,1,2,3,4,5,6,7,8,9)

            //--- constant 2-dimensional array

            Two_D_1_type is array[0..9] of integer
            Two_D_type is array[0..9] of Two_D_1_type

            Two_D = table Two_D_type
                (table Two_D_1_type (00,01,02,03,04,05,06,07,08,09)
                ,table Two_D_1_type (10,11,12,13,14,15,16,17,18,19)
                ,table Two_D_1_type (20,21,22,23,24,25,26,27,28,29)
                ,table Two_D_1_type (30,31,32,33,34,35,36,37,38,39)
                ,table Two_D_1_type (40,41,42,43,44,45,46,47,48,49)
```

```
            ,table Two_D_1_type (50,51,52,53,54,55,56,57,58,59)
            ,table Two_D_1_type (60,61,62,63,64,65,66,67,68,69)
            ,table Two_D_1_type (70,71,72,73,74,75,76,77,78,79)
            ,table Two_D_1_type (80,81,82,83,84,85,86,87,88,89)
            ,table Two_D_1_type (90,91,92,93,94,95,96,97,98,99)
            )

        //-- static places to assign things to

        This_A      : static A              // multi-lev structure
        I           : static integer
        This_One_D : static One_D_type      // 1 dim array
    enddeclare

    //-- assignment of fields of the structure that can be
    //   evaluated as constants at compile time to various fields
    //   of the structure

    This_A.This_B.This_C.Joe := a0.a1.a2.a4[5]
    This_A.This_B.This_C.Joe := a0.a1.a2.a4[Con_1_D[5]]
    This_A.This_B.This_C.Joe := a0.a1.a2.a4[Con_1_D[Con_1_D[5]]]
    This_A.This_B.This_C.Joe :=
            a0.a1.a2.a4[Con_1_D[Con_1_D[Con_1_D[5]]]]
    This_A.This_B.This_C.Joe :=
            a0.a1.a2.a4[Con_1_D[Con_1_D[Con_1_D[Con_1_D[5]]]]]


    //-- assignment of expresions of the fields of the one and
    //   two-dimensional arrays which can be evalutated as
    //   constants at compile time.

    I := Two_D[1,2]
    I := Two_D[4,5]
    I := Two_D[7,3]
    I := Two_D[Con_1_D[Con_1_D[Con_1_D[7]]],3]
    I := (Two_D[4,2]/Con_1_D[Con_1_D[8]]) *
                                Two_D[1,Con_1_D[Con_1_D[2]]]

    //-- assignments of arrays with constant indexes

    This_One_D [Con_1_D[Con_1_D[1]]]:= Con_1_D[Con_1_D[Con_1_D[1]]]
endmodule
```

3.14 SEGMENT CONTROL ON THE LSI-11

```
//-----------------------------------------------------------------
// Example of the use of 'segment'
//
%segment root = rel,con,lcl
Module segtest segment ( root )

        Declare
                C       : segment ( root ) integer
        EndDeclare

        Procedure A () segment ( root )
        EndProcedure

        Function B () Returns I : integer segment ( root )
        EndFunction

EndModule
```

CHAPTER 4

SUGGESTED FUTURE IMPROVEMENTS

4.1 SETS

The entire concept of the use of sets is not well understood within the present users of the language. It is suspected that many features envolving sets do not function correctly. Good useful example programs need to be prepared and tested.

4.1.1 Initially Of Sets

Initially of sets Sets doesn't work on the PDP11 (Vax OK). See SETTEST.PRX.

4.2 MATH ISSUES

4.2.1 Unsigned Comparisons

Unsigned comparisons are not implemented. This means that comparisons of cardinals is done with signed comparisons. Therefore, if a cardinal is large enough to use what would be the sign bit of the size of the instructions available on the respective machines (8, 16, 32), the less-than or greater-than comparison will be incorrect. The equal or not-equal comparison is still correct. This effects both mathematical user written statements and rangechecking. Also, since the compiler itself is written in Praxis, compile-time rangechecking of 32 bit cardinals is not implemented.

4.2.2 Real Exponentiation

Exponentiation is not implemented for real bases. Therefore i ** j is supported, but r ** j is not.

4.2.3 Long Real Constants

Long real constants on VAX is not supported. The following does not work:

r : long_real initially 1.0

Instead, one must type:

r : long_real initially long_real (1.0)

[This should be simple to fix] Also, suspect that other long real operations not correct.

4.2.4 Long Real On PDP Not Implemented.

4.2.5 Round, Floor, Ceiling For The PDP-11

Run time procedures that perform round(), floor(), and ceiling() need to be implemented within the runtime support package for the pdp-11. Compiler calls to the routines exist.

4.3 ADDITIONAL COMPILE-TIME RANGECHECKING

Assignment of large objects (strutures or arrays) is not rangechecked at present. This could be done. This applies to both compile time and run time rangechecking.

4.4 INTEGER AND CARDINAL CONSTANTS SHOULD BE EQUIVALENT

Integer and Cardinal Constants (ie 1, 2, 5) should be equivalent so that its not necessary to specify cardinal(2) every time one needs a 2 in an expression that requires a cardinal.

4.5 SELECT PROBLEMS

4.5.1 Tag Field Inside A Variant

Present compiler allows select on a tag field inside a variant - not good.

4.5.2 Large Range Of Cases

Select on integer with very large range of cases (1, 100000) causes the compiler to take an N*N ammount of time to process the cross-jumping optimization for the N branches in the select. The compiler is smart enough to convert

```
from:            to:

select I from    |      if i = 1 do
case 1:          |            //....
        //....   |      orif i = 100000 do
case 100000:     |            //...
        //...    |      otherwise
default:         |            //..
        //..     |      endif
endselect        |
```

The present algorithm decides based on the time/space trade offs on the actual number of "cases". This decision may not always be correct.

4.6 SILLY DECLARATIONS

```
A is integer
B is array [A] of ...
```

Is legal, but shouldnt be. (ie extremely big array)

4.7 IMPORT / EXPORT ISSUES

4.7.1 Status Of "module_name . Imported_from" Syntax?

4.7.2 WALKER

Set up logical symbols to run WALKER. Describe its use to others. for now:

```
assign DISK$LIB:[PRAXISLIB.PRODUCTS.WALKER.VMX] MYWALK
walk :== $MYWALK:WALKER
```

4.7.3 CROSS (Phebus)

Put walker program written by Phebus into Praxis and make it work within our system.

4.8 DYNAMIC ALLOCATION ISSUES

4.8.1 Allocation Of Objects With Holes

The allocation of a flexible array of items with holes in them, has the same inherent problem as holes in other objects described above. However, for Version 7.2 it was no clearing of allocated items with holes was implemented.

4.8.2 No Type Check Of Allocate Size Variable

There is no type check of the allocate size variable. See the number 0.4 in the following example - which compiles!

```
declare
    Flex_array      is packed array [1..?n] of char
    P               is pointer Flex_array
    T1:             static P
enddeclare

T1 := allocate P (0.4)
```

4.9 PROCEDURE INTERFACE ISSUES

4.9.1 Confusing N On Multiple Flexible Arrays

Parameter statement of

```
A,B : inout packed  array [1..?N] of char
```

is accepted, but which does N apply to?

4.9.2 OUT Flexible Array - What Is It?

Why is out flex array unimplemented? What are its rules?

4.10 SUGGESTIONS FOR BETTER DEBUGGING

4.10.1 Detection Of Comparisons Of Objects With Holes

An optional diagnostic to detect comparisons of objects with holes would be useful for finding code that may have had bugs with older versions of the compiler due to holes. It would also be useful to write code that runs a little faster (ie. its faster if holes dont need to be filled).

4.10.2 Source Lines Overs 132 Characters In Length

Source line over 132 characters long generates an abstract error message.

4.10.3 More Accurate Line Number Of Errors

Find line number of error more accurately. Giving the line number of the end of the module is useless.

4.10.4 Line Numbers In M11 And Listing Files

Put Line numbers in M11 file.

4.10.5 Create_Unique_Global_Symbols Compiler Directive

Add a "create_unique_global_symbols" compiler directive that causes the symbols generated in the PDP11 M11 file to be both unique within 6 characters AND have the full symbol name as follows:

```
        XXYYnn$"-----------"
where
        XX = first two characters of module name
        YY = last two characters of module name
        nn = base 36 number
        ------ = identifier name string
```

4.10.6 Diagnostice When SIZEOF Result Will Not Fit.

Need a diagnostic on sizeof if size of item will not fit into size of left variable (ie > 2**16 bits)

4.11 RUN TIME SUPPORT OF BETTER DIAGNOSTICS

4.11.1 Run Time Line Numbers

Provide a line number variable at run time so that the user program could print it out, or it could be part of the dump one gets when a process crashes? Also the name of the module that causes the crash. It might be easier to implement a "semicolon number" since the compiler knows that to a deeper level.

4.11.2 Interface To Version 4.0 Vax Debugger

4.12 OPTIMIZATION IDEAS

4.12.1 Array Referencing On LSI-11

Array referencing into small size element arrays (ie 4 bits etc.) on LSI-11 could be optimized for space.

4.12.2 Initial Conditions Of Arrays, Structures

Use P_data for storing array and structure initial conditions or constructors. As is, lists of MOV instructions are generated.

4.12.3 Register Allocation

Lots of room exists for improving register allocation. It may not even be correct in all cases. Needs study.

4.12.4 CG_Table Pattern Improvements

Some of the CG_tables are obviously less efficient than they could be.

4.12.5 Library Functions For String Operations

Library functions to do string comparisons. String descriptors - see Run Time Library users Guide, page 5-17.

4.13 VMS VERSION 4.0 ISSUES:

VMS Version 4.0 is considerable modified from earlier versions. We need to study the effect upon Praxis.

4.13.1 Module Name Limit Of 9 Characters

4.14 SUPPORT ISSUES

4.14.1 32 Bit Integer In TEXTIO

We need a 32 bit integer procedure in TEXTIO for the LSI-11. See Randy's INT32.PRX

4.14.2 Holes In The Compiler

Check all the places in the compiler that are now detected as having structures with holes. Were any other bugs solved by this fix? Can they be recoded to make the compiler run faster?

These modules have declarations with holes: OBJFIL1, OBJFIL3, PHASE3, CGOUT, CGACTION. MCODE had them, but they were removed- probable bug in reg allocation.

These modules have structure constructors with holes: ATOM, MAKCOD, LOOKUP, WALKSUB, SYNIN, MSTRC.

These modules have out val with holes: CGSPECIAL.

Note that the OPERAND structure has holes in it, and it is compared in several places in the compiler.... Also, there are some declarations of it without an initially of the null_operand.

4.14.3 Example Programs

Compile a book of example programs.

4.14.4 User Analyzer

Move the User Analyzer program into the Praxis Product and clean it up for general use. This program analyzes all use of Praxis within specific directories and generates a chart showing the quantity use of every feature in the language.

CHAPTER 5

PREVIOUSLY FIXED BUGS

The following bugs were either fixed in previous versions of the compiler, or cannot be replicated now. They are listed here for completeness.

5.1 CONVERSION FUNCTIONS

In previous versions of the compiler, the conversion functions required an object to be converted to integer before they be correctly converted to something else. This occured as run-time error on the PDP only.

For example:

Cardinal (Integer (R))

Char (Integer (L)

5.2 INTEGER OF REAL NUMBER

In previous versions of the compiler a real number must be multiplied by 1.0 before it could be converted correctly to an integer. For example: Integer (X * 1.0)

5.3 PROCEDURE/FUNCTION ISSUES

5.3.1 OUT Or INOUT By VAL On PDP-11

In previous versions of the compiler, Out or INOUT by val on PDP-11 didn't work except for the 1st argument.

5.3.2 RETURN S:Structure Or Array

In previous versions of the compiler, RETURN S:structure or array on the PDP-11 didnt work.

5.4 ASSIGNMENT OF IMPORTED STRINGS

It has been reported that an 80 character static packed array of char which is imported is not properly assigned to a local 80 character packed array of char, unless the assignment is made in a loop one character at a time. This problem cannot be replicated.

5.5 SELECT ON OUT-OF-RANGE ENUMERATION

Previous versions of the compiler would cause the compiler to crash on a SELECT on an enumeration when the enumerated item was out of range of the enumeration. This can result from doing an I/O read into a structure that contains the enumerated item.

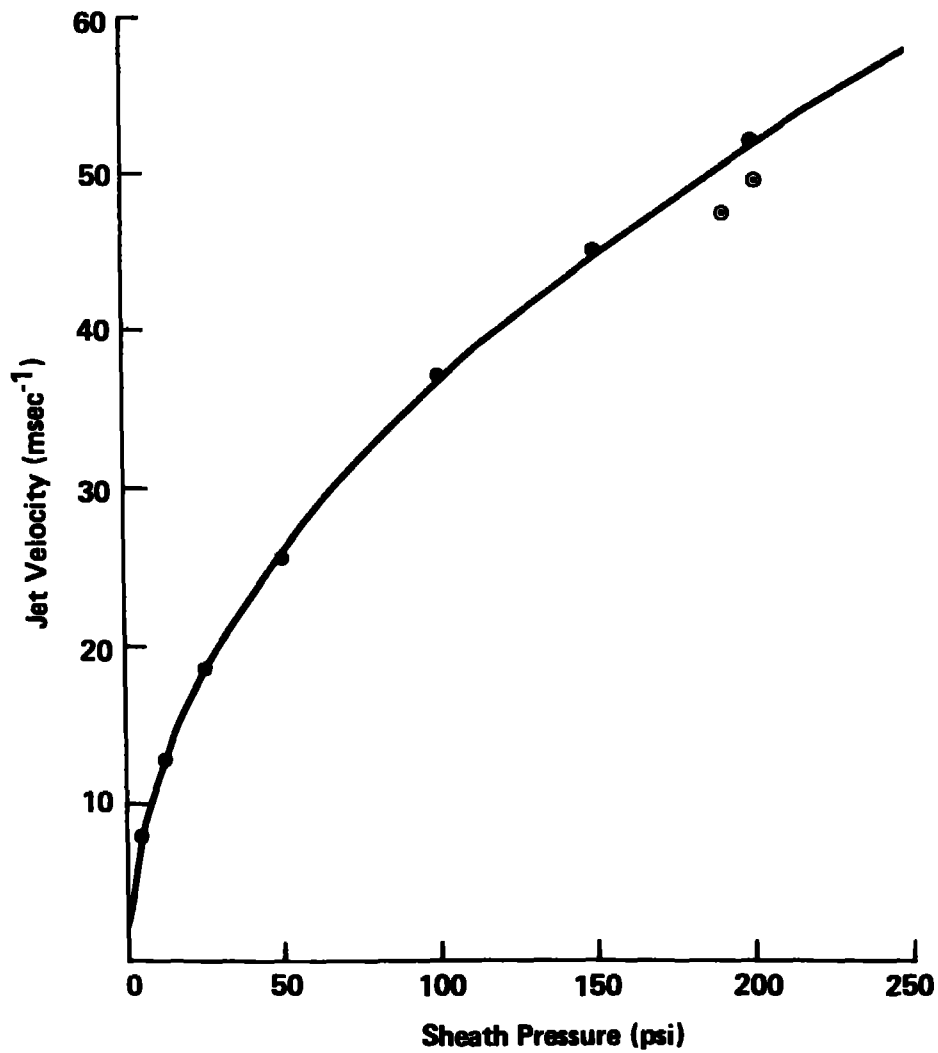
60
50
40
30
20
10
Jet Velocity (msec^{-1})
0
50
100
150
200
250
Sheath Pressure (psi)

Fig. 1

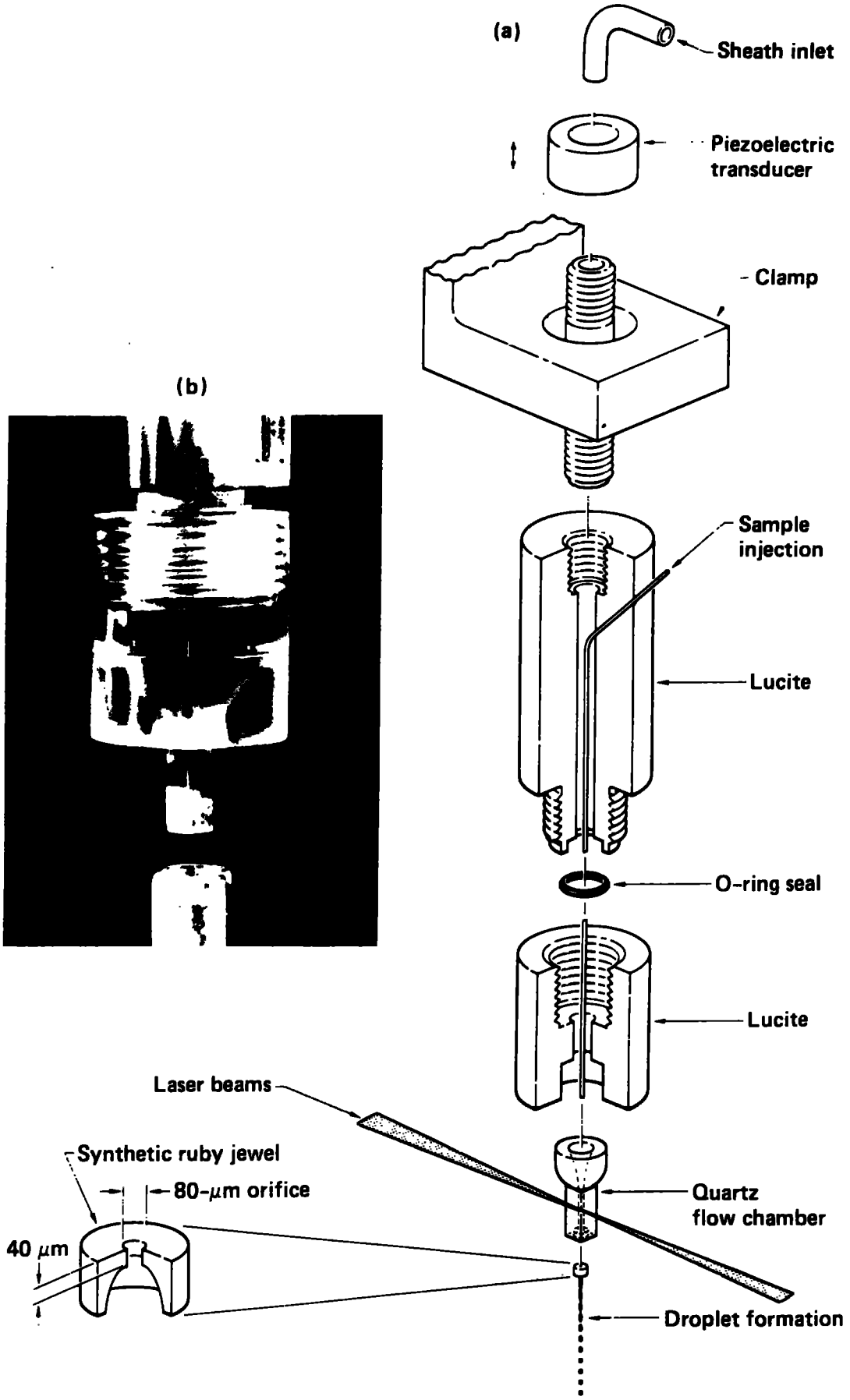
(a)
Sheath inlet
Piezoelectric transducer
Clamp
(b)
Sample injection
Lucite
O-ring seal
Lucite
Laser beams
Synthetic ruby jewel
80-μm orifice
40 μm
Quartz flow chamber
Droplet formation

Fig. 2

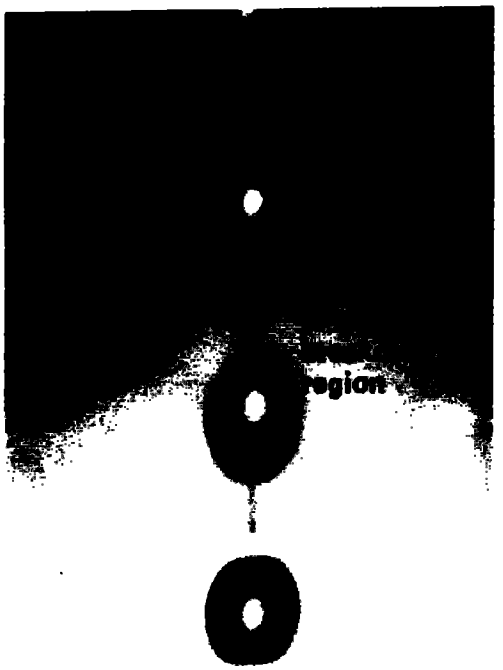

Fig. 3

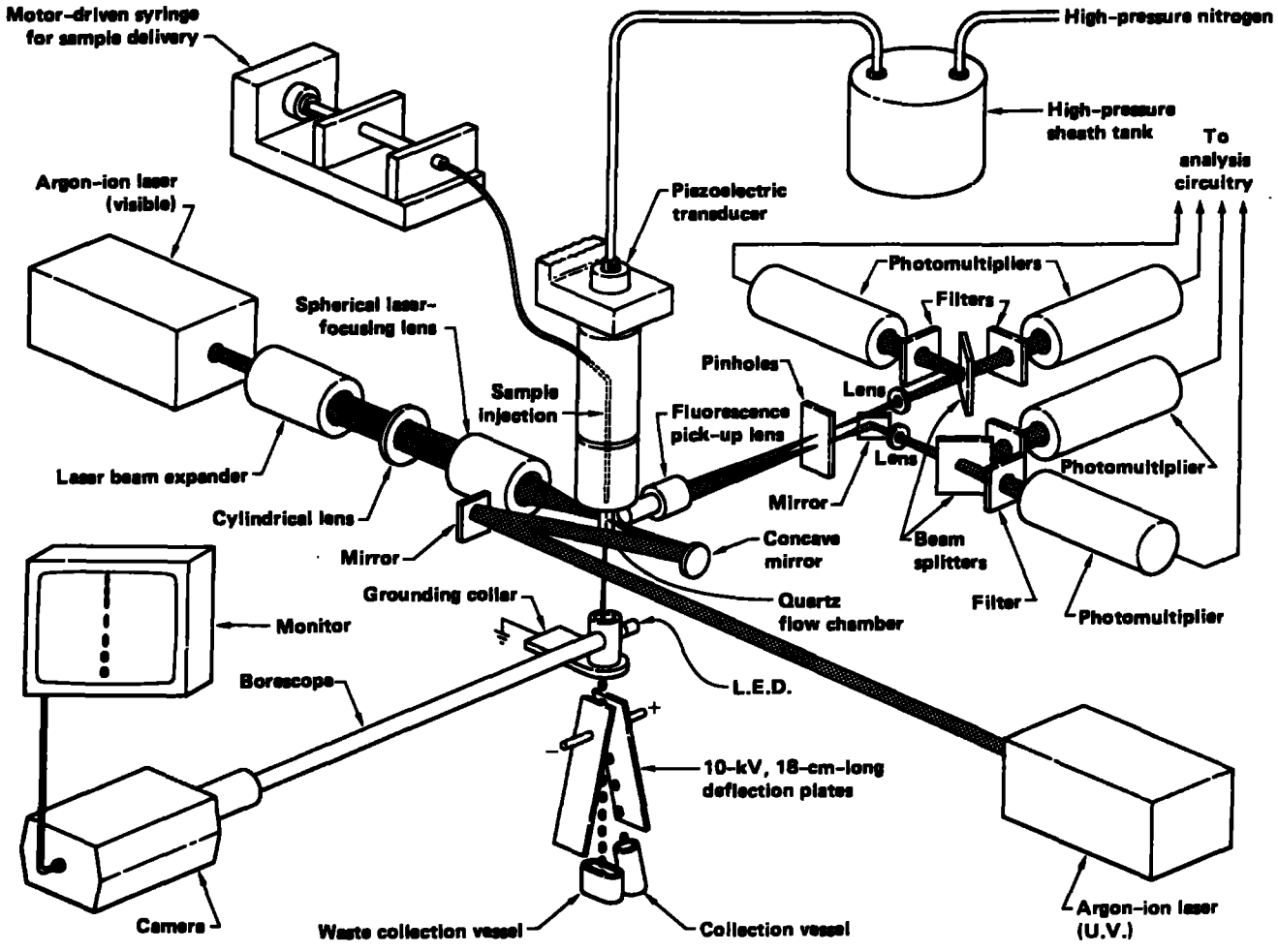
Motor-driven syringe for sample delivery
High-pressure nitrogen
High-pressure sheath tank
To analysis circuitry
Argon-ion laser (visible)
Piezoelectric transducer
Photomultipliers
Filters
Spherical laser-focusing lens
Pinholes
Lens
Sample injection
Fluorescence pick-up lens
Lens
Photomultiplier
Laser beam expander
Mirror
Cylindrical lens
Concave mirror
Beam splitters
Mirror
Grounding collar
Quartz flow chamber
Filter
Photomultiplier
Monitor
Borescope
L.E.D.
10-kV, 18-cm-long deflection plates
Camera
Waste collection vessel
Collection vessel
Argon-ion laser (U.V.)

Fig. 4

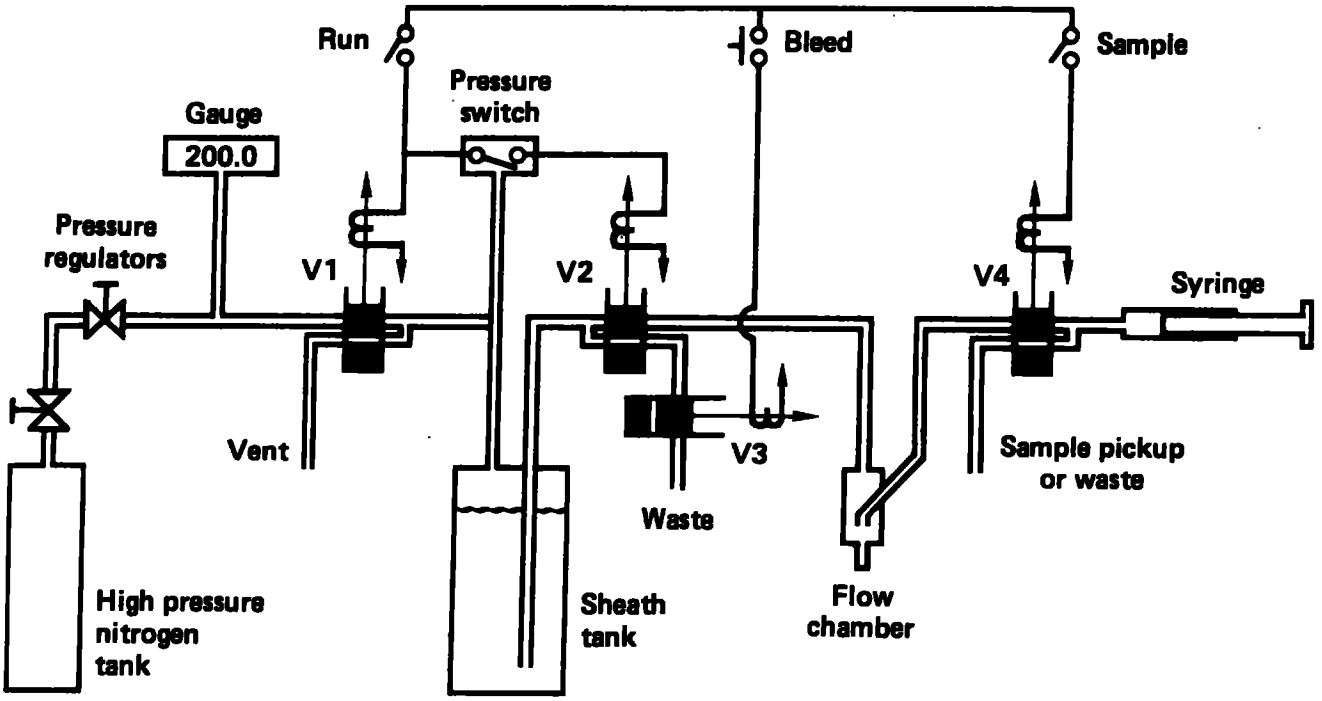
Run
Bleed
Sample
Pressure switch
Gauge
200.0
Pressure regulators
V1
V2
V4
Syringe
Vent
V3
Waste
Sample pickup or waste
High pressure nitrogen tank
Sheath tank
Flow chamber

Fig. 5

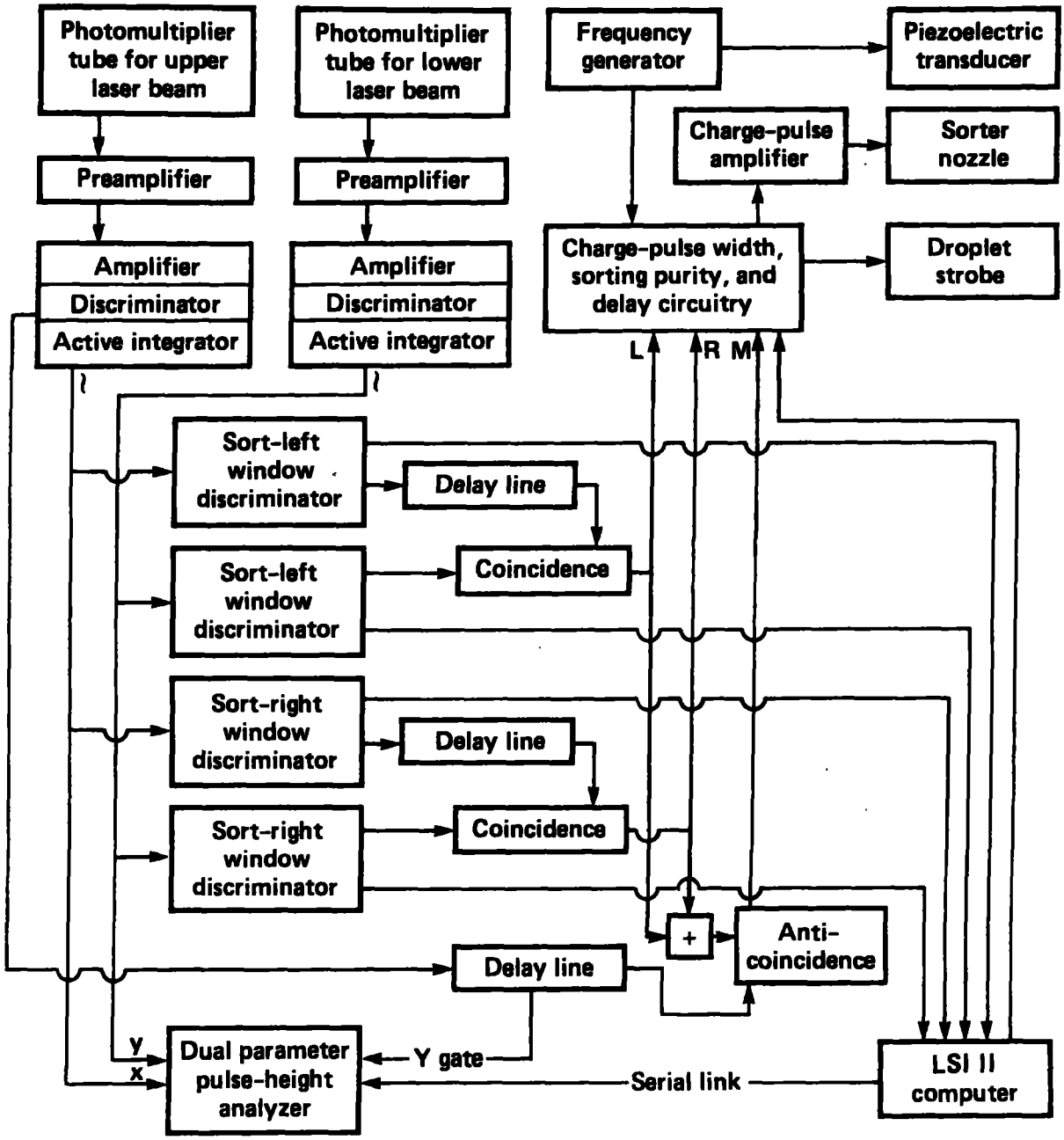
Photomultiplier tube for upper laser beam
Photomultiplier tube for lower laser beam
Frequency generator
Piezoelectric transducer
Charge-pulse amplifier
Sorter nozzle
Preamplifier
Preamplifier
Amplifier
Discriminator
Active integrator
Amplifier
Discriminator
Active integrator
Charge-pulse width, sorting purity, and delay circuitry
Droplet strobe
L
R
M
Sort-left window discriminator
Delay line
Sort-left window discriminator
Coincidence
Sort-right window discriminator
Delay line
Sort-right window discriminator
Coincidence
+
Anti-coincidence
Delay line
Y
X
Dual parameter pulse-height analyzer
Y gate
Serial link
LSI 11 computer

Fig. 6

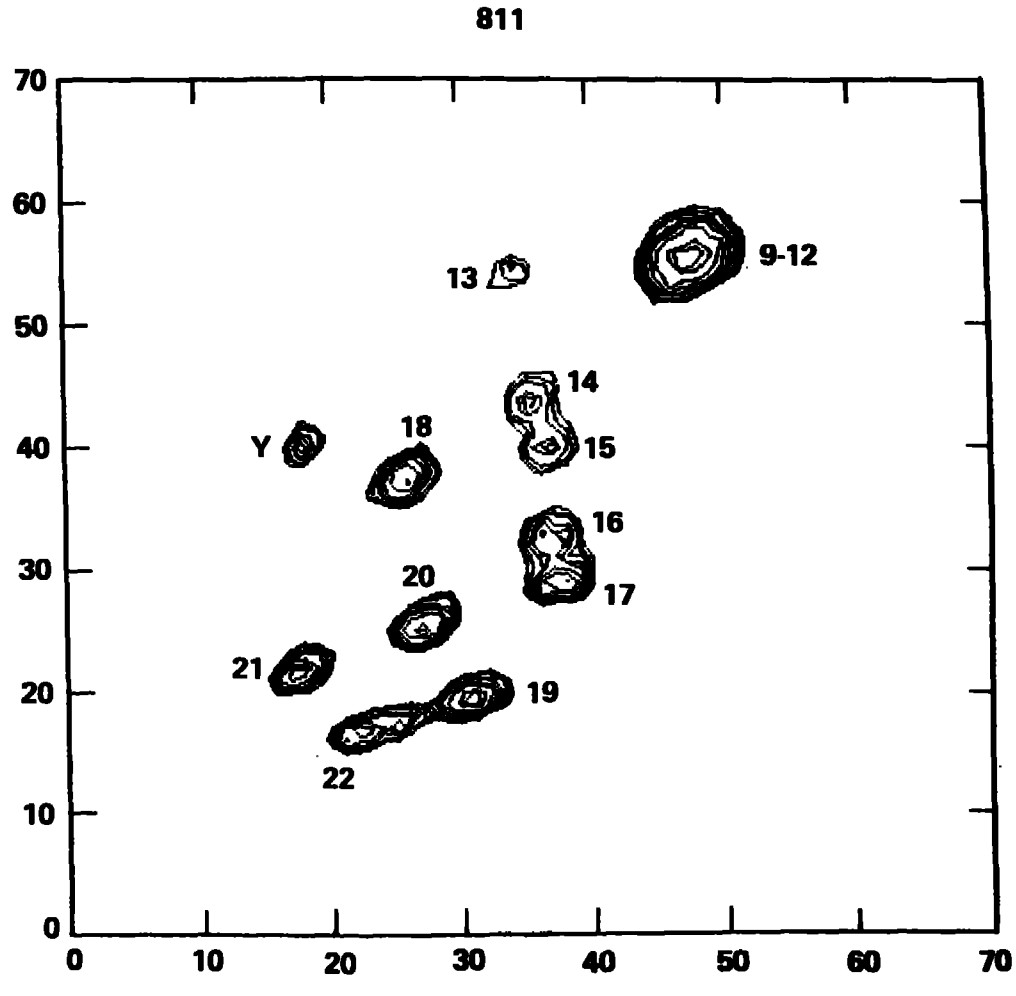
811
70
60
50
40
30
20
10
0
0
10
20
30
40
50
60
70
9-12
13
14
15
Y
18
16
17
20
21
19
22

Fig. 7

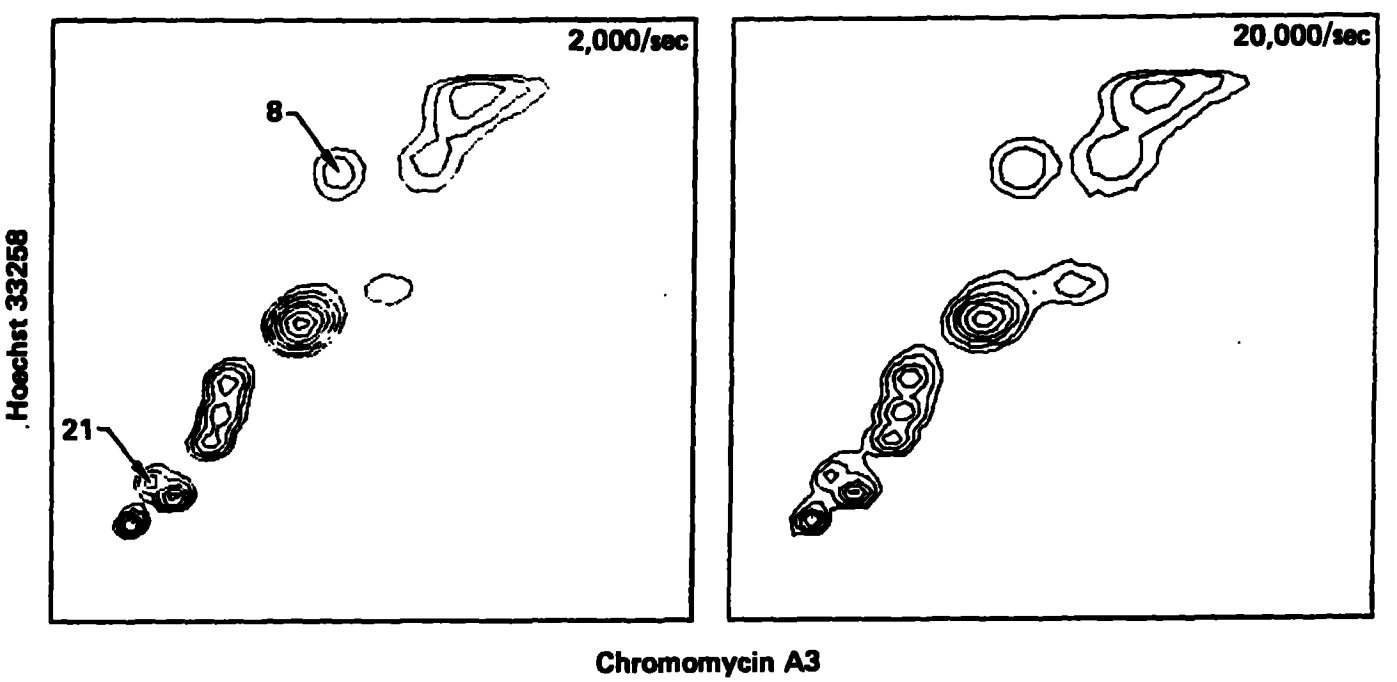

Fig. 8

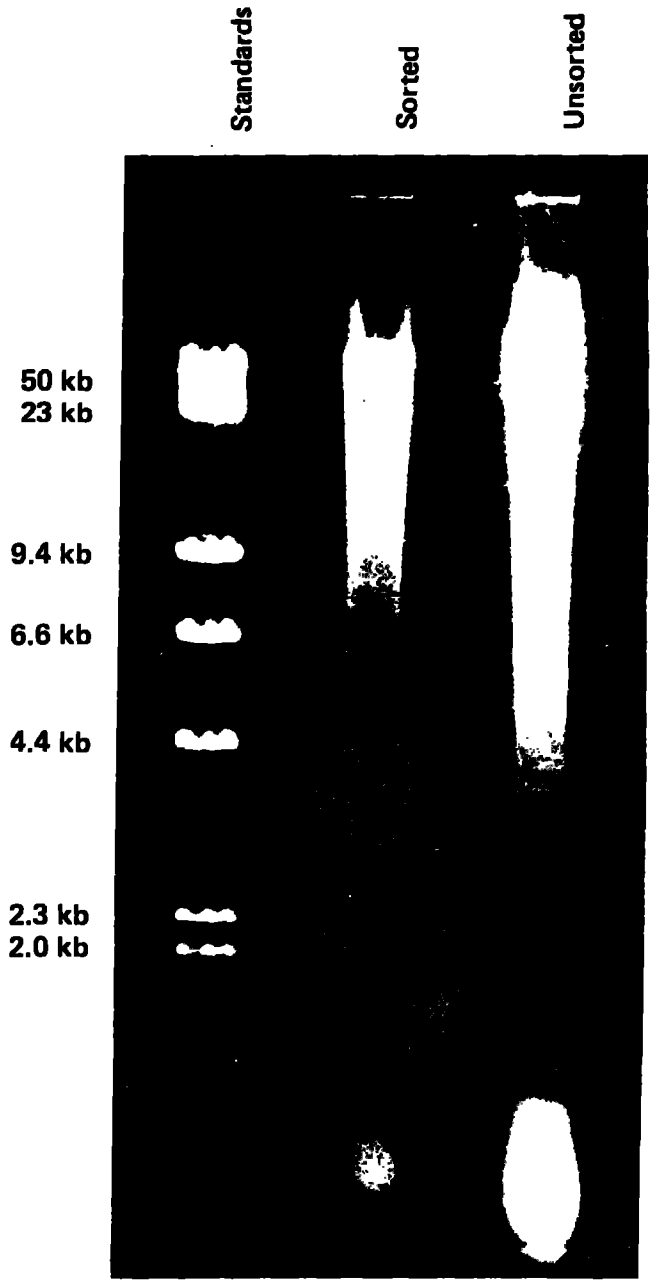
Standards
Sorted
Unsorted
50 kb
23 kb
9.4 kb
6.6 kb
4.4 kb
2.3 kb
2.0 kb

Fig. 9

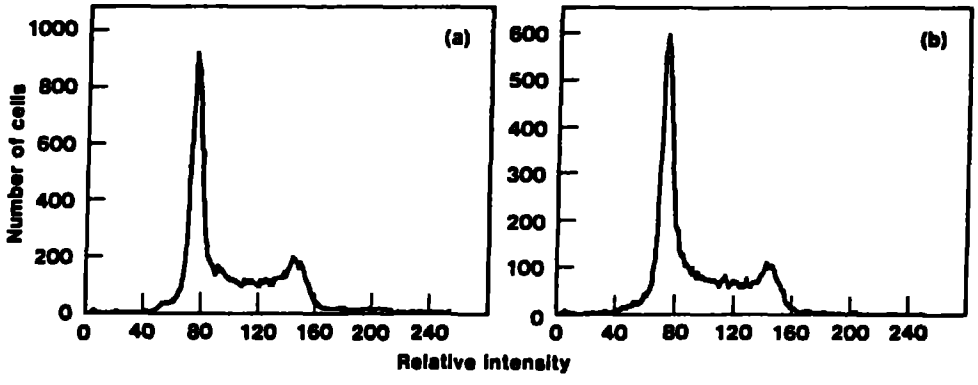
(a)
(b)
Number of cells
Relative intensity
0
200
400
600
800
1000
100
300
500
40
80
120
160
240

Fig. 10

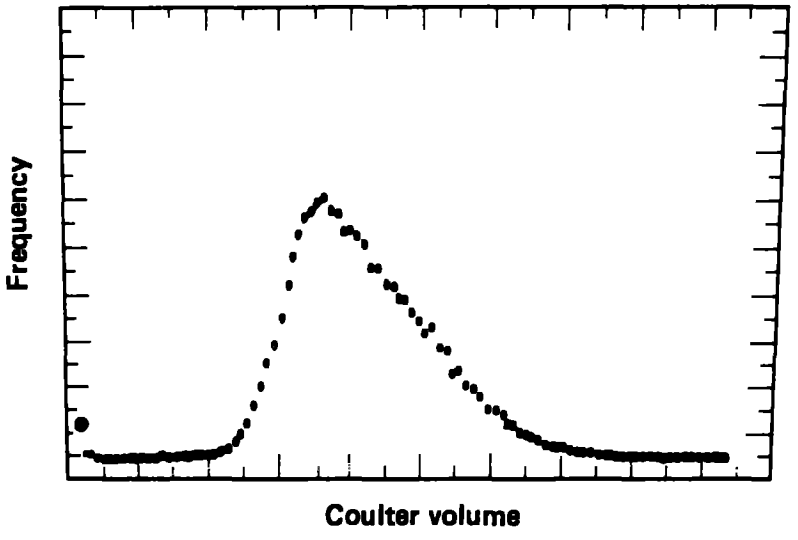

Fig. 11