AD 748226

APPLIED DATA RESEARCH, INC.

SEE AD748215

222

APPLIED DATA RESEARCH, INC.
LAKESIDE OFFICE PARK • WAKEFIELD, MASSACHUSETTS 01880 • (617) 245-9540

FIFTH SEMI-ANNUAL TECHNICAL REPORT
(14 January 1972 - 13 July 1971)
FOR THE PROJECT
COMPILER DESIGN FOR THE ILLIAC IV

VOLUME II

Principal Investigator and Project Leader:
Robert E. Millstein

	Phone (617) 245-9540
ARPA Order Number	ARPA 1554
Program Code Number	0D30
Contractor:	Applied Data Research, Inc.
Contract No.:	DAHC04 70 C 0023
Effective Date:	13 January 1970
Amount:	$916,712.50

Sponsored by
Advanced Research Projects Agency
ARPA Order No. 1554

APPLIED DATA RESEARCH, INC.
LAKESIDE OFFICE PARK • WAKEFIELD, MASSACHUSETTS 01880 • (617) 245-9540

FIFTH SEMI-ANNUAL TECHNICAL REPORT
(14 January 1972 - 13 July 1972)
FOR THE PROJECT
COMPILER DESIGN FOR THE ILLIAC IV

VOLUME II

CADD-7208-1411

Principal Investigator and Project Leader:
Robert E. Millstein Phone (617) 245-9540

TABLE OF CONTENTS

VOLUME II

1. IVTRAN TRANSCRIBER

General Information

The transcriber is a package of FORTRAN routines which looks at the IVTRAN internal representation, called intermediate language or intermediate language tables, of an IVTRAN program and outputs the corresponding IVTRAN source program. The package consists of two main subroutines, TRNSCR and TSPECP which output procedure and specification statements respectively, and 17 auxiliary subroutines. However, one call to TRNSCR will output an entire program. The total size of the package is about 6.2K words.

The Transcriber is particularly useful in conjunction with the Paralyzer. After the Paralyzer has made its transformations on the original IVTRAN program, the Transcriber can output the newly created program in standard IVTRAN source format. The output file can then be edited and fed back to the IVTRAN compiler.

The transcriber code is very closely linked to the structure of the Intermediate Language. Knowledge of the Intermediate Language tables is essential when the transcriber code is examined.

Subroutine Descriptions

In the descriptions that follow, there is often a list of files under the heading of external references. These are files which contain global information. They are inserted into the source language by INSERT prior to compilation of the routines.

PROGRAM NAME: TRNSCR (SUBROUTINE)

PURPOSE: To output, in ASCII FORTRAN source form, a complete IVTRAN program.

FUNCTION: It may call TDELIM to output delimiter lines. Then it calls TSYMFX to fix generated symbols and TSPECP to output specification statements. Finally it outputs all procedure statements.

EXTERNAL REFERENCES:

FILES - TCOM.DEF, CTAB.DEF, CTAB.NIT, LTAB.DEF, LTAB.NIT, STAB.DEF, CGLOB.DEF

FUNCTIONS - L

SUBROUTINES - TDELIM, TSYMFX, TSPECP, TINTPR, TCHOUT, TDOUT, TINT, TKONPR, TCLPRT, TSYMPR, TLABFX

COMMON - TRNSC

SIZE: 1841 words

CALLING SEQUENCE: CALL TRNSCR (NAME)

NAME (Hollerith) the name which is used if the delimiter comment lines are printed. These lines are printed only if the global variable, LTRNF, is set to 1.

TECHNIQUES:

TRNSCR has a short loop which follows the STATOP chain and transfers control to a separate piece of code for each type of STATOP.

The IO list processor can combine LABEL, DO, IODATA, and IOFIN STATOPs into one IO list. It uses a stack to keep track of nested implied DO loops.

The LOGICAL IF processor is able to handle multiple statements after the condition. Even though this isn't legal IVTRAN input, it can be generated by the PARALYZER. So, if there is only one statement after the condition, the standard LOGICAL IF is printed. If there are more statements, the negative of the condition is printed and a GO TO statement, that transfers control past the multiple statements, is printed as the last part of the LOGICAL IF. Even though this modified form is printed, the internal structure of the LOGICAL IF is not changed.

The debug processor must combine several different types of STATOPs into one concatenated debug statement. However, since each of these STATOPs has a simple linear structure, the procedure is not difficult.

To output expressions TRNSCR uses recursive code* to walk the tree and output operands and operators as it encounters them. It uses stacks to keep track of its position in the tree and other pertinent data. IX is the index into these stacks; it indicates the depth of the current position in the tree. The stacks are:

IRET - return address stack
ICPT - pointer CTAB stack
IPRES - operator precedence stack
USPRN - IF expression parenthesized stack
IOPR - operator stack

---

* Recursive calls are implemented via the ASSIGN and assigned GO TO statements.

PROGRAM NAME: TSPECP (SUBROUTINE)

PURPOSE: To output all specification statements.

FUNCTION: It outputs PROGRAM UNIT, IMPLICIT, TYPE, DIMENSION, COMMON, OVERLAP, EQUIVALENCE, DEFINE, DATA, FORMAT, FREQUENCY, EXTERNAL, and END statements.

EXTERNAL REFERENCES:

FILES - TCOM.DEF, CGLOB.DEF, STAB.DEF, STAB.NIT, XCTAB.DEF, XCTAB.NIT, OTAB.DEF, OTAB.NIT, QTAB.DEF, QTAB.NIT, DTAB.DEF, DTAB.NIT, FTAB.DEF, FTAB.NIT, RTAB.DEF, RTAB.NIT, ETAB.DEF, ETAB.NIT, LTAB.DEF, LTAB.NIT

FUNCTIONS - L, TFIELD

SUBROUTINES - TCLOUT, TINT, TDOUT, TCHOUT, TDIMEN, TCLPRT, TSYMPR, TARRAY, TDIM, TINTPR, TKONPR

COMMON - TRNSC

SIZE: 1210 words

CALLING SEQUENCE: CALL TSPECP

TECHNIQUES:

The code is very straightforward, but descriptions of the various tables are essential to understanding it.

PROGRAM NAME: TKONPR (SUBROUTINE)

PURPOSE: To output constants (from the constant table).

FUNCTION: It looks at the type and values of the constants to generate the output characters. It uses TDPREC to output all double precision values.

EXTERNAL REFERENCES:

FILES - TCOM.DEF, KTAB.DEF, KTAB.NIT, ETAB.DEF, ETAB.NIT
FUNCTIONS - L, TFIELD
SUBROUTINES - TINTPR, TCHOUT, TINT, TDPREC, TDIM, TDIMEN
COMMON - TRNSC
FORTRAN SUPPORT - ENCODE for integer and real, FLOTA, EXP

SIZE: 631 words

CALLING SEQUENCE: CALL TKONPR (KTBPNT, NEGPRN)

KTBPNT (KTAB pointer, integer) pointer to the KTAB entry which holds the specified constant

NEGPRN (integer) needed for integer, double integer, real, and double precision constants. If it is 1, as opposed to 0, a negative value will be enclosed in parentheses.

TECHNIQUES:

To output double integer constants TKONPR creates a double integer, power of ten table. Each entry in this table consists of two 24 bit quantities (the same way the constant is stored). This table is then used to generate the output digits one at a time.

PROGRAM NAME: TINT (SUBROUTINE)

PURPOSE: To output a group of characters, ignoring spaces or asterisks, which must not be split between lines.

FUNCTION: The block of characters to be sent is in a common array. TINT makes two passes through this array. The first pass counts the characters. Then, if there isn't enough room on the current line, TINT forces a continuation line to be created. The second pass sends the characters. TINT also clears the input array.

EXTERNAL REFERENCES:
FUNCTIONS - TFIELD
SUBROUTINES - TCHOUT
COMMON - TRNSC

SIZE: 129 words

CALLING SEQUENCE: CALL TINT (IFSPAC)

IFSPAC (integer). If this variable is zero, spaces and null characters will not be sent. If it is 1, asterisks and nulls will not be sent.

TECHNIQUE: None

PROGRAM NAME: TCHOUT (SUBROUTINE)

PURPOSE: To append a character to the line buffer.

FUNCTION: It uses the character pointer to place the character in the packed array (line buffer). If a continuation line is needed, it sends the current line and starts the continuation line. TCHOUT also updates the character pointer.

EXTERNAL REFERENCES:
SUBROUTINES - IVOUT, TSBYT
COMMON - TRNSC

SIZE: 68 words

CALLING SEQUENCE: CALL TCHOUT (CHAR)

CHAR (integer) the 7-bit ASCII character, right justified, which is to be appended.

TECHNIQUE: None

PROGRAM NAME: TCLOUT (SUBROUTINE)

PURPOSE: To clear the line buffer.

FUNCTION: It resets the character pointer and clears the line buffer.

EXTERNAL REFERENCES:
COMMON - TRNSC

SIZE: 20 words

CALLING SEQUENCE: CALL TCLOUT

TECHNIQUE: None

PROGRAM NAME: TDOUT (SUBROUTINE)

PURPOSE: To output one line of characters which have been stored in the line buffer.

FUNCTION: It sends the line to a compiler utility routine for output, resets the character pointer, and clears the line buffer.

EXTERNAL REFERENCES:
SUBROUTINES - IVOUT
COMMON - TRNSC

SIZE: 30 words

CALLING SEQUENCE: CALL TDOUT

TECHNIQUE: None

PROGRAM NAME: TLABFX (SUBROUTINE)

PURPOSE: To convert a generated label into a legal FORTRAN label.

FUNCTION: It takes generated labels, whose values start at 131072, and converts them into generated labels, whose values start at 131072 + 90000. These labels are later printed by the TRANSCRIBER with values starting at 90000. TLABFX makes sure it doesn't generate duplicate labels.

EXTERNAL REFERENCES:
FILES - TCOM.DEF, LTAB.DEF, LTAB.NIT
FUNCTIONS - L
SUBROUTINES - S

SIZE: 67 words

CALLING SEQUENCE: CALL TLABFX (LTBPNT,NLAB)

LTBPNT (LTAB pointer, integer) points to label table entry which contains the generated label

NLAB (integer) next available label value

TECHNIQUE: None

PROGRAM NAME: TDPREC (SUBROUTINE)

PURPOSE: To output a double precision number.

FUNCTION: It outputs the double precision number.

EXTERNAL REFERENCES:
FUNCTIONS - TFIELD
SUBROUTINES - TCHOUT, TSBYT, TINT, TINTPR
FORTRAN SUPPORT - ENCODE for DOUBLE PRECISION, DOUBLE PRECISION arithmetic

SIZE: 276 words

CALLING SEQUENCE: CALL TDPREC (EXP, HIGH, LOW, CHARCT, ICHAR)

EXP (integer) binary exponent of number

HIGH (integer) sign and 35 high-order bits of fraction in two's complement form

LOW (integer) sign and 35 low-order bits of fraction in two's complement form

CHARCT (integer) number of character positions left on line

ICHAR (integer array (10)) place to store output characters

TECHNIQUE:

Because ILLIAC-IV DOUBLE PRECISION numbers have a much larger legal range than PDP-10 DP numbers, TDPREC converts the DP number into a PDP-10 number between .1 and .999 ... or -.1 and -.999 ... and an integer base ten exponent. These numbers are then encoded and sent to output routines.

PROGRAM NAME: TDELIM (SUBROUTINE)

PURPOSE: To output a delimiter between different PARALYZER rewritings of the same program.

FUNCTION: It outputs a three line delimiter which has the following form:

```
C
C ! ! NAME
C
```

EXTERNAL REFERENCES:

FUNCTIONS - TFIELD

SUBROUTINES - TCLOUT, TCHOUT, TDOUT

SIZE: 60 words

CALLING SEQUENCE: CALL TDELIM (NAME)

NAME (program name, Hollerith) this name becomes part of the delimiter.

TECHNIQUES: None

PROGRAM NAME: TSYMFX (SUBROUTINE)

PURPOSE: To convert any compiler generated symbols into legal FORTRAN symbols.

FUNCTION: It transforms all symbol names of the form %CCCCC into the form TDDDDT, where the C's are any character or blank and the D's are decimal digits. It also sets the explicit type bit if T is not implicitly of the same type of the symbol.

EXTERNAL REFERENCES:
FILES - TCOM.DEF, STAB.DEF, STAB.NIT, XCTAB.DEF, XCTAB.NIT
FUNCTIONS - L, TFIELD
SUBROUTINES - S, TSBYT

SIZE: 227 words

CALLING SEQUENCE: CALL TSYMFX

TECHNIQUES: None

PROGRAM NAME: TCLPRT (SUBROUTINE)

PURPOSE: To output an entry name, a subroutine name, or a function name with its associated dummy arguments.

FUNCTION: It outputs the name and follows pointers to the ETAB and back to the STAB to get the dummy arguments.

EXTERNAL REFERENCES:
FILES - TCOM.DEF, STAB.DEF, STAB.NIT, ETAB.DEF, ETAB.NIT
FUNCTIONS - L
SUBROUTINES - TSYMPR, TCHOUT, TINT
COMMON - TRNSC

SIZE: 75 words

CALLING SEQUENCE: CALL TCLPRT (STBPNT)
STBPNT (STAB pointer, integer) pointer to the STAB entry which contains the name of the entry point.

TECHNIQUES: None

PROGRAM NAME: TSYMPR (SUBROUTINE)

PURPOSE: To output a symbol name.

FUNCTION: It uses TFIELD to get the packed SIXBIT ASCII of the symbol name and passes it on to TINT in 7-bit ASCII form.

EXTERNAL REFERENCES:
FILES - TCOM.DEF, STAB.DEF, STAB.NIT
FUNCTIONS - L, TFIELD
SUBROUTINES - TINT
COMMON - TRNSC

SIZE: 45 words

CALLING SEQUENCE: CALL TSYMPR (STBPNT)
STBPNT (STAB pointer, integer) pointer to the STAB entry which contains the name.

TECHNIQUES: None

PROGRAM NAME: TINTPR (SUBROUTINE)

PURPOSE: To output an integer.

FUNCTION: It uses an ENCODE statement to get the ASCII characters for the integer and calls TINT to send the characters.

EXTERNAL REFERENCES:
SUBROUTINES - TINT
COMMON - TRNSC
FORTRAN SUPPORT - ENCODE for INTEGER

SIZE: 26 words

CALLING SEQUENCE: CALL TINTPR (NUMBER)
NUMBER (integer) the integer to be output

TECHNIQUES: None

PROGRAM NAME: TDIMEN (SUBROUTINE)

PURPOSE: To output the extent and allocation of an array.

FUNCTION: It outputs the extent and allocation of an array.

EXTERNAL REFERENCES:

FILES - TCOM.DEF, ETAB.DEF, ETAB.NIT

FUNCTIONS - L

SUBROUTINES - TDIM, TCHOUT, TINTPR

SIZE: 104 words

CALLING SEQUENCE: CALL TDIMEN (ETBPNT)

ETBPNT (ETAB pointer, integer) points to the ETAB entry which contains the extent and allocation data.

TECHNIQUES: None

PROGRAM NAME: TDIM (SUBROUTINE)

PURPOSE: To output the extent of an array.

FUNCTION: It outputs the constants or symbolic names which define the extent of an array.

EXTERNAL REFERENCES:
FILES - TCOM.DEF, ETAB.DEF, ETAB.NIT
FUNCTIONS - L
SUBROUTINES - TCHOUT, TINTPR, TSYMPR

SIZE: 65 words

CALLING SEQUENCE: CALL TDIM (ETBPNT)

ETBPNT (ETAB pointer, integer) points to the ETAB entry which contains the extent data.

TECHNIQUES: None

PROGRAM NAME: TARRAY (SUBROUTINE)

PURPOSE: To output some simple array macros for the DEFINE and EQUIVALENCE specification statements.

FUNCTION: It calls various routines to output the array name and subscripts. The only types of subscripts it recognizes are integer constant, "*", $ constant, $ constant + constant, or $ constant-constant.

EXTERNAL REFERENCES:
FILES - TCOM.DEF, CTAB.DEF, CTAB.NIT
FUNCTIONS - L
SUBROUTINES - TSYMPR, TCHOUT, TKONPR, TINTPR

SIZE: 139 words

CALLING SEQUENCE: CALL TARRAY (CTBPNT)

CTBPNT (CTAB pointer, integer) points to the CTAB entry which contains the ARRAY EXOP.

TECHNIQUES: None

PROGRAM NAME: TFIELD (INTEGER FUNCTION)

PURPOSE: To extract a specified bit field from a word.

FUNCTION: It extracts a specified bit field from a word and returns it right justified.

EXTERNAL REFERENCES: None

SIZE: 19 words

CALLING SEQUENCE: I = TFIELD (SWORD, SBIT, NBITS)

SWORD (source word, integer) word from which field is to be extracted

SBIT (start bit number, integer) bit number of the leftmost bit of the field

NBITS (number of bits, integer) number of bits in the field

TECHNIQUES: Written in MACRO-10

PROGRAM NAME: TSBYT (SUBROUTINE)

PURPOSE: To insert a specified bit field into a word.

FUNCTION: It inserts a value into a specified bit field of a word.

EXTERNAL REFERENCES: None

SIZE: 20 words

CALLING SEQUENCE: CALL TSBYT (DWORD, SBIT, NBITS, VALUE)

DWORD (destination word, integer) word into which field will be inserted.

SBIT (start bit, integer) bit number of leftmost bit of field.

NBITS (number of bits, integer) number of bits in field.

VALUE (new field value, integer) right justified value of field to be inserted.

TECHNIQUES: Written in MACRO-10

2. STORAGE ALLOCATOR

2.1 Introduction

On most machines array allocation is trivial. Successive arrays are merely mapped into contiguous one dimensional segments in core. On the ILLIAC-IV, however, arrays are mapped into rectangles which must then be allocated within ILLIAC-IV memory. Since storage is two dimensional, allocation is not nearly as simple as on conventional machines. Indeed, because of physical skewing the problem actually becomes one of packing n-dimensional solids. The restrictions on how skewed objects can be packed together makes it simplest to think of arrays as solids which can be packed into solids and then mapped as a unit into ILLIAC memory. Mapping each array into a two dimensional object and then trying to fit those objects together would be a very unmanageable task. Consequently, ILLIAC data allocation is essentially an n-dimensional packing problem (where n is the highest dimensionality of any array to be allocated). The object, of course, is to pack all arrays into the smallest object possible. The only restriction on the object is that one of the extents must be an exact multiple of the number of PE's in a row of ILLIAC-IV memory. While there are several devious ways of doing this, it is doubtful that an analytic solution or even an optimum solution exists.

2.2 Overview

The description above implies that arrays are all packed into one large array. Actually arrays are packed into several blocks of different sizes, each block being an exact multiple of PE row width in one dimension. Arrays are currently allocated in order of decreasing size (volume). As each new array is encountered a list of available holes is checked to determine if there is any place where the array will fit. If there is a place for the array then the array is placed in the smallest available spot. If there is no hole which will contain the array then the cost of expanding a hole to be large enough for the array is examined. If the minimum cost is less than that of putting the array in a new block then the hole is expanded and the array inserted. Otherwise, the smallest possible block that will contain the array is created and the array is placed in the new block.

Note that once an array is allocated its position is not substantially changed. The greatest change that can occur is merely a translation in one or more directions for the purpose of expanding holes. Each array is allocated in such a way as to minimize the cost of adding the array, without consideration of any further arrays to be allocated. This, of course, does not necessarily produce an optimum allocation and statistics on distribution of array sizes could be very useful in designing schemes for dealing with interactions between storage requirements. These statistics would be very easy to gather while compiling user programs and some provision should be made for obtaining them for later work. Backtracking could also be useful in many situations and depending on how expensive allocation is for typical users some backtracking to explore particularly hopeful allocations might be of value.

2.3 Conventions

1) <u>Subroutine Naming</u>

All routines which are part of allocation itself begin with the letter A. Auxillary debugging or data gathering routines do not begin with A. No other subroutine naming conventions exist.

2) <u>Tables</u>

Allocation makes use of five tables. Four of these are used to keep information required by allocation and the fifth (TBLPNT) is used to keep track of the location and sizes of the other four tables -- TBLPNT is used when the other tables must be expanded. The four main tables are:

ARRAYTBL -- The array table, which contains the original array sizes and orientations.

BLOCKTBL -- The block table, which contains descriptions of each block resulting from allocation. The information contained includes block extents and orientation as well as a pointer to a list of arrays contained in the block.

ALLTBL -- The allocation table. This table contains a description of each array which has been allocated. Array orientation and a pointer to the block containing the array are kept as well as a duplicate of the array extent information. Each element in this table is in a chain which has its root in BLOCKTBL.

HOLETBL -- The hole table. This table maintains information on all available places ("holes") for putting arrays.

These four tables are all threaded lists with fixed size entries. Each table actually has three parts associated with it:

I. The table header - which keeps information like the number of entries in the table, the number of fields in an entry, the size of an entry, etc.

II. Table byte pointers - these pointers define table fields by standard IV-TRAN compiler conventions.

III. Working storage - where table entries are actually kept.

Each table has two three letter prefixes, one for referring to elements in the table header and one for referring to byte pointers. These prefixes, along with a two or three letter suffix are used for naming all table parameters and fields. The following suffixes are used:

I. Common to all table headers

CAP -- CAPacity of the table, in maximum number of entries the table can hold. While these variables are defined they are not used by allocation.

NE -- Number of Entries contained in the table. These variables are kept updated but are not used by allocation. They are kept only because they are interesting statistics. Conventions on updating this count are somewhat confused. In some tables the count is updated immediately after an entry is removed from the free chain while in others it is updated only when an entry is added to the active chain.

ELS -- ELement Size - the size of an entry in the table.

FRE -- FREe - a pointer to the head of the table free entry list.

BYT -- Number of BYTe pointers - the number of byte pointers associated with the table (i.e., the number of fields in the table).

INC -- INCrement - the amount by which to expand the table when more space is needed.

II. Uncommon header suffixes

HD -- HeaD - this is an anachronism to the original design. It was intended to point to the start of the active chain in each table. However, entry zero is now assumed to be a dummy entry at the head of the active chain in each table except

ALLTBL (which is pointed to by BLOCKTBL entries) and thus keeping a pointer to the head of the chain is unnecessary. These variables will probably eventually be deleted.

MXN -- MaXimum Number - this is used only in the hole table and records the maximum number of holes occurring during the allocation.

III. Byte pointer suffixes for scalar fields

FST (in HOLETBL) -- FirST - a byte pointer to the first word of an entry. This field is redundant and appears only in the hole table (ABLFST has a different meaning). Normally NXT is used when a pointer to the first word is desired (the NXT field always comes in the first word of the entry).

USE -- This field indicates whether a table entry is in use or not. This field is not used by allocation but is kept updated for convenience in debugging. This flag is set to one when an entry is removed from the free chain and is set to 0 when the entry is returned.

NXT -- NeXT - this is the chain field and points to the next entry in the chain (each table has two chains, the active chain and the free chain, with entry zero containing a pointer to the first element in the active chain). A value of zero indicates the end of the chain.

SZE -- SiZE - this field exists in all but ALLTBL and is used for recording the size of the array, block, or hole as the case may be.

DIM -- DIMension - this also occurs in all but ALLTBL and records the dimension of the structure in question.

ORG -- ORiGin - this occurs only in ARRAYTBL and points back to the table (symbol table or overlap table) which contained the original definition of the array.

OVL -- OVerLap - this also occurs only in ARRAYTBL and indicates whether the array occurred in an overlap statement or not.

BLK -- BLocK - this occurs only in HOLETBL and records which block a particular hole belongs to.

ID -- IDentification - this occurs only in ALLTBL and points to the array entry in ARRAYTBL so that more information about the array can be extracted if necessary.

RMN -- ReMaiNing - used only in BLOCKTBL, this field records the amount of space left in a block.

FST (in BLOCKTBL) -- FirST - this field appears in this usage only in BLOCKTBL, where it points to the chain of arrays (in ALLTBL) which are allocated to that particular block.

IV. Byte pointer suffixes for vector fields -- these fields contain one subfield for each dimension. Thus L(PNT, ALACRD (5)) would be the fifth coordinate of an array in the allocation table.

EXT -- EXTents - these contain the extents of the appropriate structure (array, hole, or block). Extents are stored in descending order by size and do not imply any specific orientation.

PRM -- PeRMutation - this describes the orientation of a structure. For example L(PNT, ARAPRM (1)) would give the index of the extent associated with the first dimension of array PNT. In order to get the first extent of array PNT one would execute:

```
IT = L(PNT, ARAPRM (1) )
EXTENT = L(PNT, ARAEXT (IT) )
```

CRD -- CooRDinates - this is used only in ALLTBL and HOLETBL and gives the coordinates of point (0, 0, ..., 0) of an array or hole. Coordinates are relative to the beginning of the block in which the structure is defined. Coordinates are not permuted so that the field CRD(I) contains the I'th coordinate.

The prefixes for the different tables are:

ART	-- Array table header entries
ARA	-- Array table field pointers
AHT	-- Hole table header entries
AHL	-- Hole table field pointers
ALT	-- Allocation table header entries
ALA	-- Allocation table field pointers
ABT	-- Block table header entries
ABL	-- Block table field pointers

3) Instrumentation Conventions

All routines in the allocation package begin with the statement

```
CALL IENTRD ('SUB NAME')
```

and exit with the statements

```
CALL IEXIT ('SUB NAME')
RETURN
```

where sub.name is padded with spaces to at least six and not more than 10 characters. This provides a convenient handle for writing trace and timing routines. Currently there is a trace and a statistics package included with the allocator which allows optional tracings of subroutine entries as well as frequency counts and execution timing histograms.

4) Table Organization Conventions

The four major tables, as mentioned above, are composed of chained entries. All tables except the allocation table contain two chains, an active chain and a free chain. Both chains use the same entry field for chaining. The location of the beginning of the free chain is stored in the table header whereas the location of the first entry in the active chain is stored in the chain field of entry 0. Entry zero is otherwise unused (indeed, a pointer to entry zero indicates the end of the chain). ALLTBL is somewhat different. The free chain is still pointed to by a header entry but ALLTBL can contain several active chains, one for each block in the block table. Each block table entry points to the beginning of a unique active chain in ALLTBL and there are no active chain pointers stored in ALLTBL itself. Entry 0 is completely unused in ALLTBL.

2.4 Table Expansion

These routines are concerned with expanding the four main tables when more room is needed. There is one short routine for each of the four tables (ARTEXP, ALTEXP, ABTEXP, and AHTEXP for ARRAYTBL, ALLTBL, BLOCKTBL, and HOLETBL respectively) which does nothing but call ATBEXP with the address of the appropriate table fields. ATBEXP calls AEXPND to get more space immediately following the specified table. It then calls ACHAIN to chain together the new entries and it finally updates the table's free chain pointer and the table capacity.

PROGRAM NAME:	ARTEXP
PURPOSE:	To expand the array table.
FUNCTION:	Calls ATBEXP with appropriate parameters to expand ARRAYTBL.
EXTERNAL REFERENCES:	ATBEXP
SIZE:	
CALLING SEQ.:	Call ARTEXP
TECHNIQUES:	None

PROGRAM NAME:	ALTEXP
PURPOSE:	To expand the allocation table.
FUNCTION:	Calls ATBEXP, with appropriate parameters to expand ALLTBL.
EXTERNAL REFERENCES:	ATBEXP
SIZE:	
CALLING SEQ.:	Call ALTEXP
TECHNIQUES:	None

PROGRAM NAME:	ABTEXP
PURPOSE:	To expand the block table.
FUNCTION:	Calls ATBEXP with appropriate parameters to expand BLOCKTBL.
EXTERNAL REFERENCES:	ATBEXP
SIZE:	
CALLING SEQ.:	Call ABTEXP
TECHNIQUES:	None

PROGRAM NAME:	AHTEXP
PURPOSE:	To expand the hole table.
FUNCTION:	Calls ATBEXP with appropriate parameters to expand HOLETBL.
EXTERNAL REFERENCES:	ATBEXP
SIZE:	
CALLING SEQ.:	AHTEXP
TECHNIQUES:	None

PROGRAM NAME: ATBEXP
Allocation TaBle EXPansion

PURPOSE: To expand a specified table.

FUNCTION: ATBEXP is passed the table name as an ASCII constant, a field pointer to the entry chain field, and pointers to the table capacity, table entry size, free chain pointer, and table increment in the table header. ATBEXP calls AEXPND to expand the table by the table increment. It then calls ACHAIN to chain together the new free entries. After returning from ACHAIN it checks to see if entry 0 is in the new free chain and if it is the free chain pointer is made to point to entry 1 rather than entry 0, otherwise the free pointer is left pointing to the first entry in the newly allocated space. Finally, ATBEXP increments the table capacity appropriately and returns.

EXTERNAL REFERENCES: AEXPND, ACHAIN, S

SIZE:

CALLING SEQ.: Call ATBEXT (TBLNAM, TBLCP, TBLENS, TBLFRE, TBLINC, TBLCHN)

Args:

TBLNAM	- 2 word ASCII constant containing the table name
TBLCP	- table capacity in the table header
TBLENS	- table entry size entry in the table header
TBLFRE	- free chain pointer in the table header
TBLINC	- table header entry giving the proper amount by which to expand the table

TBLCHN	- table chain field pointer - used for establishing the new free chain.

TECHNIQUES:

It is assumed that if after expanding the table the first free entry is entry zero then all table entries are in the free chain. Consequently, TBLFRE is made to point to entry one and the chain field of entry 0 is set to 0 to indicate that the active chain is empty.

PROGRAM NAME: AEXPND
Allocation EXPaND

PURPOSE: Expand working tables for the allocator.

FUNCTION: AEXPND finds the table name in TBLPNT. If it can't find the table it pauses and prints out 'BAD CALL TO AEXPND', if this happens there is a definite bug in the allocator. Upon finding the specified name AEXPND expands core by the necessary amount, indicates the increase in size in TBLPNT, moves any following tables down in core to make the available space adjacent to the appropriate table, updates the byte pointers for any table moved (using TBLPNT to find the number and location of the byte pointers for each table) and finally zeros the newly appended table space.

EXTERNAL REFERENCES: XCORE, [illegible]DX, M UP, CSETZ

COMMON: TBLPNT (an array), XCESS, COREND

SIZE:

CALLING SEQ.: Call AEXPND (NAME, AMNT)

NAME - 2 word ASCII constant containing the table name

AMNT - amount by which to expand the table.

PROGRAM NAME:	ACHAIN Allocation Table CHAINing
PURPOSE:	To chain together newly acquired table free space.
FUNCTION:	ACHAIN is given the first and last address (relative to the beginning of the table) of a new table area as well as the table entry size and a field pointer to the table chain field. ACHAIN merely chains together all the entries in the new area.
EXTERNAL REFERENCES:	S
SIZE:	
CALLING SEQ.:	Call ACHAIN (CHNFLD, FSTFRE, LSTFRE, ENTSZE)

CHNFLD - field pointer to the table chain field
FSTFRE - first location (relative to the beginning of the table) of the area to be chained.
LSTFRE - last location of the area to be chained
ENTSZE - size of a table entry.

TECHNIQUES:

A variable (NXT) is set equal to

FSTFRE + I * ENTSZE

and deposited in the chain field of entry

FSTFRE + (I - 1) * ENTSZE

for all I such that

FSTFRE + I * ENTSZE < = LSTFRE.

PROGRAM NAME:	MVUP
PURPOSE:	To copy information from one area to another, possibly destroying the first area but producing a true copy even if the first area overlaps the second.
FUNCTION:	MVUP calculates the end of the first area and the end of the second and then copies the table from the end. This allows the first area to overlap the second area without producing garbage (it is assumed that the first area starts below the second area).
EXTERNAL REFERENCES:	None
SIZE:	
CALLING SEQ.:	CALL MVUP (FROM, TO, STOP) FROM - first word of first area TO - first word of second area STOP - last word of second area.
TECHNIQUES:	None

2.5 Table Manipulation Routines

These routines provide a standard means of getting free table entries, adding or deleting new entries for tables, etc. The routines for the different tables are not completely analogous since some tables need more manipulation than others. The tables do have the following analagous routines:

AGTARY, AGTALL, AGTBL, AGTHOL -- gets the next free block from the free chain of the appropriate table, expanding the table if necessary and in some tables setting the entry use bit and/or incrementing the table entry count.

ARAADD, ADDARY, ADDBLK, ADHOLE -- adds a new entry to the active chain of the appropriate table. In some tables this routine sets the use flag or increments the entry count, rather than the routine which first got the free block.

ADLBLK, ALLRLS, AHLRLS -- release a table entry which is not currently in the table active chain. That is, put the entry back on the free list, performing any necessary bookkeeping in the process.

Other routines covered in this section are:

ADLTHL	- deletes a hole entry from the hole table
ADLHL2	- a subroutine of ADLTHL
ARMHL	- removes a hole from the active chain but does not return it to the free chain
ANWHL	- creates a new hole to go with a new block
ACLRBL	- deletes all holes belonging to a specified block (clears the block of holes).
ABLCPY	- copies an entire block, along with all the arrays allocated to it and one (and only one) of the holes associated with it.

PROGRAM NAME: AGTARY

Allocation - Get ARraY

PURPOSE: To get a block from the ARRAYTBL free chain.

FUNCTION: AGTARY makes sure the free chain is not empty (it calls ARTEXP if the chain is empty). It then removes the first block from the free chain, and initializes it by setting all extent fields (ARAEXT) to one, all permutation fields I to I (I .TO. ARAPRM (I) for all 1 <= I <= N where N is the maximum number of dimensions). AGTARY also sets the ARAUSE flag to one, ARASZE (tne array size) to 1, and sets the array dimension (ARADIM) to the maximum allowable dimension.

EXTERNAL REFERENCES: ARTEXP, S, L

COMMON: ARTFRE, ARANXT, ARAEXT, ARAPRM, ARASZE, ARAUSE, ARADIM, AMXDIM

SIZE:

CALLING SEQ.: CALL AGTARY (NWARY)

On returning NWARY points to the location of the newly available block.

TECHNIQUES: None

PROGRAM NAME:	AGTALL Allocation - GeTALLocation block
PURPOSE:	To get the next block from the ALLTBL free chain.
FUNCTION:	AGTALL insures that the free chain is not empty by calling ALTEXP if necessary. It then updates the free chain pointer, sets the use bit (ALAUSE) in the newly extracted block, and returns.
EXTERNAL REFERENCES:	ALTEXP, S, L
COMMON:	ALTFRE, ALAUSE
SIZE:	
CALLING SEQ.:	CALL AGTALL (NWBLK) Upon returning NWBLK points to the newly extracted block.
TECHNIQUES:	None

PROGRAM NAME: AGTBL

Allocation - GeTBLock

PURPOSE: To get the next entry from the BLOCKTBL free chain.

FUNCTION: AGTBL insures that the block free chain is not empty, calling ABTEXP if necessary. It then extracts the first entry from the free chain, sets ABLUSE, increments ABTNE (the table entry count), and returns.

EXTERNAL REFERENCES: ABTEXP, S, L

COMMON: ABTFRE, ABTNE

SIZE:

CALLING SEQ.: CALL ABTBL (NWBLK)

On returning NWBLK points to the new entry in the block table.

TECHNIQUES: None

PROGRAM NAME:	AGTHOL Allocation - GeT HOLe
PURPOSE:	To get the next entry from the HOLETBL free chain.
FUNCTION:	AGTHOL insures that the free chain is non-empty -- calling AHTEXP if necessary. It then extracts the first entry from the free chain, increments the table entry count, updates AHTMXN, if appropriate (AHTMXN is the maximum number of holes -- a variable kept only for information on necessary HOLETBL size, the variable is unused in the ALLOCATOR). AGTHOL also zeros out the AHLEXT, AHLPRM, and AHLCRD fields.
EXTERNAL REFERENCES:	AHTEXP, S, L, MAX0
COMMON:	AHTFRE, AHLNXT, AHTNE, AHTMXN, AMXDIM, AHLEXT, AHLPRM, AHLCRD
SIZE:	
CALLING SEQ.:	CALL AGTHOL (NWHOLE) On returning NWHOLE points to the newly extracted and initialized entry block.
TECHNIQUES:	None

PROGRAM NAME:	ARAADD ArrAY ADDition
PURPOSE:	Add an array entry to the active chain of the ARRAYTBL.
FUNCTION:	Link the new block into the active chain between the entries zero and one and increment the ARRAYTBL entry count.
EXTERNAL REFERENCES:	S, L
COMMON:	ARANXT, ARTNE
SIZE:	
CALLING SEQ.:	CALL ARAADD (NWARY) Where NWARY points to the array entry to be added to the chain.
TECHNIQUES:	None

PROGRAM NAME:	ADDARY ADD ARray
PURPOSE:	To add an array to the ALLTBL active chain associated with a specified block.
FUNCTION:	The block pointer pointing to the chain in ALLTBL is copied to the chain field of the new entry and the BLOCKTBL pointer is then updated to point to the new entry. ALTNE (number of entries in ALLTBL) is incremented, the ABLNAR (number of arrays allocated to the block) field in the BLOCKTBL entry is incremented, and the ABLRMN field (the amount of space remaining in the block) is decremented by the size of the new entry (currently the array size is calculated but this is unnecessary as the entry contains a pointer to ARRAYTBL, if the efficiency of this routine becomes critical the array size could be extracted from ARRAYTBL).
EXTERNAL REFERENCES:	S, L
COMMON:	ABLFST, ALANXT, ALTNE, ABLNAR, ABLDIM, ALAEXT, ABLRMN
SIZE:	
CALLING SEQ.:	CALL ADDARY (NWARY, BLOCK) Where NWARY is a pointer to the new entry in the ALLTBL and BLOCK is a pointer to the block in BLOCKTBL to which the array is being allocated.
TECHNIQUES:	None

PROGRAM NAME:	ADDBLK ADD BLocK
PURPOSE:	To add a block to the BLOCKTBL active chain.
FUNCTION:	The new entry is merely put on the beginning of the chain with the chain fields of the new block and of entry 0 updated appropriately.
EXTERNAL REFERENCES:	S, L
COMMON:	ABLNXT
SIZE:	
CALLING SEQ.:	CALL ADDBLK (BLKPNT) BLKPNT should point to the entry being added.
TECHNIQUES:	None

PROGRAM NAME:	ADHOLE Add HOLE
PURPOSE:	To add a new hole to the HOLETBL active chain keeping the table sorted in ascending size and insuring that no hole is contained in any other hole.
FUNCTION:	The HOLETBL active chain is searched to find the first hole which is either larger (in volume), or has the same size but a higher dimension. When this spot is found its location is saved and the remainder of the active chain is compared against the new hole to make sure that the hole is not contained in any others. If the hole is contained ADHOLE returns immediately, otherwise the new hole is added into the list at the spot found earlier and the AHLUSE flag in the entry is set.
EXTERNAL REFERENCES:	L, S, AFLDCM, ACNTAN
COMMON:	AHLSZE, AHLDIM, AHLBLK, AHLNXT, AHLEXT, AHLCRD, AHLUSE
SIZE:	
CALLING SEQ.:	CALL ADHOLE (HOLPNT) Where HOLPNT points to the hole to be added to the table.
TECHNIQUES:	None

PROGRAM NAME:	ADLBLK Allocation - DeLete BLocK
PURPOSE:	Delete a block from the active chain in BLOCKTBL.
FUNCTION:	ADLBLK clears all holes associated with the block by calling ACLRBL, releases all arrays in ALLTBL which were allocated to the block, and returns the specified block to the BLOCKTBL free chain. It is assumed that the specified block is not in the active chain and no attempt is made to remove it from the active chain.
EXTERNAL REFERENCES:	ACLRBL, ALLRLS, L, S
COMMON:	ABLFST, ALANXT, ABLNXT
SIZE:	
CALLING SEQ.:	CALL ADLBLK (BLK) BLK points to the block to be deleted from BLOCKTBL.
TECHNIQUES:	None

PROGRAM NAME:	ALLRLS Allocation Block ReLeaSe
PURPOSE:	To return an allocation block to the ALLTBL free chain.
FUNCTIONS:	The specified block is merely linked back into the beginning of the free chain and ALTFRE and the entry chain field are updated accordingly. The entry use flag ALAUSE, is also reset.
EXTERNAL REFERENCES:	S, L
COMMON:	ALTFRE, ALANXT, ALAUSE
SIZE:	
CALLING SEQ.:	CALL ALLRLS (ARRAY) Where ARRAY points to the allocation block being released.
TECHNIQUES:	None

PROGRAM NAME:	AHLRLS Allocation HOLe ReLeaSe
PURPOSE:	To return a hole to the HOLETBL free chain.
FUNCTION:	AHLRLS places the specified hole at the beginning of the HOLETBL free chain, decrements AHTNE, and resets the entry AHLUSE bit. AHLRLS does not remove the hole from the active chain and if this has not already been done an error will result.
EXTERNAL REFERENCES:	S
COMMON:	AHLUSE, AHLNXT, AHTNE
SIZE:	
CALLING SEQ.:	CALL AHLRLS (HOLPNT) Where HOLPNT points to the hole being released.
TECHNIQUES:	None

PROGRAM NAME:	ADLTHL Allocation DeLeTe HoLe
PURPOSE:	To remove an entry from the HOLETBL active chain and place it on the free chain.
FUNCTION:	ADLTHL traces through the active chain to find the hole preceding the specified hole and then calls ADLHL2 to actually update the pointers which transfer the hole to the free chain.
EXTERNAL REFERENCES:	ADLHL2
COMMON:	AHLNXT
SIZE:	
CALLING SEQ.:	CALL ADLTHL (HOLPNT) Where HOLPNT points to the hole to be deleted.
TECHNIQUES:	None

PROGRAM NAME: ADLHL2

Allocation DeLete HoLe 2

PURPOSE: To actually transfer a hole from the HOLETBL active chain to the free chain.

FUNCTION: ADLHL2 changes the chain field of the hole proceeding the specified hole to point to the hole following the specified hole. It then puts the old hole back into the free chain by calling AHLRLS.

EXTERNAL REFERENCES: S, L, AHLRLS

COMMON: AHLNXT

SIZE:

CALLING SEQ.: CALL ADLHL2 (HOLE, PREV)

PREV	- points to the hole preceding HOLE in the active chain
HOLE	- points to the hole being deleted.

TECHNIQUES: None

PROGRAM NAME:	ARMHL Allocation ReMove HoLe
PURPOSE:	To remove a hole from the HOLETBL active chain without returning it to the free chain.
FUNCTION:	ARMHL threads through the active chain until it finds the hole preceding the specified hole. It then updates the preceding hole's chain field to remove the hole from the active chain.
EXTERNAL REFERENCES:	S, L
COMMON:	AHLNXT
SIZE:	
CALLING SEQ.:	CALL ARMHL (HOLE) HOLE points to the hole to be removed.
TECHNIQUES:	None

PROGRAM NAME:	ARMHL Allocation ReMove HoLe
PURPOSE:	To remove a hole from the HOLETBL active chain without returning it to the free chain.
FUNCTION:	ARMHL threads through the active chain until it finds the hole preceding the specified hole. It then updates the preceding hole's chain field to remove the hole from the active chain.
EXTERNAL REFERENCES:	S, L
COMMON:	AHLNXT
SIZE:	
CALLING SEQ.:	CALL ARMHL (HOLE) HOLE points to the hole to be removed.
TECHNIQUES:	None

PROGRAM NAME:	ACLRBL Allocation CLeaR BLock
PURPOSE:	To clear all the holes which belong to a specified block.
FUNCTION:	ACLRBL threads through the active hole list and calls ADLTHL on all holes whose AHLBLK field indicates that they belong to the specified block.
EXTERNAL REFERENCES:	ADLTHL, L
COMMON:	AHLNXT, AHLBLK
SIZE:	
CALLING SEQ.:	CALL ACLRBL (BLK) Where BLK points to the block in BLOCKTBL which is to be cleared.
TECHNIQUES:	None

PROGRAM NAME:	ABLCPY Allocation BLock CoPy
PURPOSE:	To copy an entire block along with all arrays which are allocated to it and one of the holes.
FUNCTION:	ABLCPY is used for estimating the cost of expanding a particular hole, and only this hole is copied. Except for the other holes, however, the entire block structure is copied. ABLCPY gets a new block entry, copies the old block entry to the new, threads through the old allocation chain and creates a new chain with duplicate entries, adjusts the new block entry to point to the new chain, and finally creates a new hole, copies the old hole, puts in a pointer to the new hole to the new block, and adds the new hole to the hole chain.
EXTERNAL REFERENCES:	AGTBL, AMVENT, AGTALL, S, L, AGTHOL, ADHOLE
COMMON:	ABLNXT, ABTELS, ABLFST, ALANXT, ALTELS, AHLNXT, AHTELS, AHLBLK
SIZE:	
CALLING SEQ.:	CALL ABLCPY (OBLK, NWBLK, OHOLE, NWHOLE) Entry Values: OBLK - the block in BLOCKTBL to be copied OHOLE - the hole to be copied with the block. Values Returned: NWBLK - a pointer to the newly created block copy NWHOLE - a pointer to the new hole copy.
TECHNIQUES:	None

PROGRAM NAME: ANWHL

Allocation - NeW HoLe

PURPOSE: Create a new hole the same size as a new block.

FUNCTION: ANWHL gets a new hole entry and copies the block extents and permutation vectors as well as the dimension, size, and block number. ANWHL then calls ADHOLE and adds the new hole to the active hole chain.

EXTERNAL REFERENCES: L, S, AGTHOL, ADDHOLE

COMMON: ABLDIM, AHLCRD, ABLEXT, AHLPRM, ABLPRM, AHLDIM, AHLSZE, AHLBLK

SIZE:

CALLING SEQ.: CALL ANWHL (BLK, HOLE)

Entry:

BLK - pointer to the block to be associated with the new hole.

Exit:

HOLE - pointer to the newly created hole.

TECHNIQUES: None

2.6 Logical Functions

There are four logical functions in the allocator and they are all concerned with testing different conditions or degrees of overlap between holes and holes, holes and arrays, or arrays and arrays. The functions, in more or less increasing strength, are: AFLDCM, AOVLP, ARAOLP, and ACNTAN. The functions have the following use:

AFLDCM -- tests to see if the extents of one structure are all less than or equal to the corresponding extents of another structure. This does not really test for any overlap condition but is a necessary condition for one structure to be entirely contained in another. Consequently, this test, which is relatively fast, is often made to decide whether or not to test for containment.

AOVLP -- this determines whether an array overlaps a hole. This is used to decide which holes must be updated when a new array is allocated.

ARAOLP -- determines whether one array overlaps another. This condition only arises while holes are being expanded and is used to determine which arrays must be moved.

ACNTAN -- tests to determine if one structure is totally contained within another structure of the same type. Currently, this is primarly used to determine if a new hole is a subset of an already existing hole in order to avoid creating duplicate holes.

PROGRAM NAME: AFLDCM

Allocation - FieLD CoMpare

PURPOSE: Compares the extent of two arrays or holes to determine if one could be totally contained in the other.

FUNCTION: Since extent fields are kept in order of decreasing extents the test succeeds iff EXTENT 1 (I) < = EXTENT 2 (I) for all I < = DIM where DIM and the extent field pointers are passed as arguments. Since the field pointers are passed as arguments holes may be compared to arrays.

EXTERNAL REFERENCES: L

COMMON: None

SIZE:

CALLING SEQ.: RESULT = AFLDCM (PNT1, PNT2, DIM, EXTP1, EXTP2)

Args:

PNT1	- pointer to first structure
PTN2	- pointer to second structure
DIM	- number of dimensions to compare
EXTP1	- extent field vector for the table corresponding to PTN1
EXTP2	- extent field vector for the table corresponding to PNT2

RESULT IS .TRUE. IFF ALL EXTENTS OF PNT1 are < = the corresponding extents of PNT2.

TECHNIQUES: None

PROGRAM NAME: AOVLP

Allocation - OVerLaP

PURPOSE: To determine if an array intersects (or overlaps) a hole.

FUNCTION: The function is false if the array does not overlap the hole in any one dimension. Effectively AOVLP computes HLSTR (I), HLEND (I), ASTR (I), AEND (I) where HLSTR (I) is the smallest coordinate in the hole in the Ith dimension and HLEND (I) is the highest and ASTR and AEND are similar for arrays. AOVLP is false iff AEND (I) < = HLSTR (I) or HLEND (I) < = ASTR (I) for some I.

EXTERNAL REFERENCES: L

COMMON: AHLDIM, AHLCRD, AHLPRM, AHLEXT, ALACRD, ALAEXT, ALAPRM

SIZE:

CALLING SEQ.: RESULT = AOVLP (ARRAY, HOLE)

Args:

ARRAY	- pointer to an array in ALLTB
HOLE	- pointer to a hole in HOLETBL.
RESULT	- .TRUE. IFF the array overlaps the hole.

TECHNIQUES: The permutation vector must be used for determining which extent is associated with the Ith dimension.

PROGRAM NAME: ACNTAN

Allocation CoNTAiN

PURPOSE: To determine if one structure is totally contained in another.

FUNCTION: This is similar to AFLDCM except that:

1) All relevant fields are passed as arguments so that two arbitrary structures of the same type (holes and holes or arrays and arrays) may be compared.

2) ACNTAN is true iff

S2 (I) < = S1 (I) < = E1 (I) < = E2 (I)

where S1 (I), E1 (I), S2 (I), E2 (I) are the first and last locations of the two structures in each dimension. In ACNTAN S1, S2, E1, E2 are not actually vectors but are computed and tested for each dimension.

EXTERNAL REFERENCES: L

COMMON: None

SIZE:

CALLING SEQ.: RESULT = ACNTAN (PNT1, PNT2, DIM, COORD, EXTNT, PERM)

Args:

PNT1	- pointer to first entry
PNT2	- pointer to second entry
DIM	- dimension of entries
COORD	- vector of coordinate field pointer

EXTNT	- vector of extent field pointers
PERM	- vector of permutation field pointers.

RESULT is true iff PTN1 is contained in PNT2.

TECHNIQUES:

The PERM vector must be used to determine the extent corresponding to dimension I.

PROGRAM NAME:	ARAOLP ARrAy OverLaP
PURPOSE:	To determine if one array overlaps another.
FUNCTION:	This routine is called after translating an array which has already been allocated to determine whether the translation has resulted in the array overlapping any other array. This is used to determine which arrays must be moved when expanding a hole. The test for overlap is performed in the same way as in AOVLP except that all coordinates, extents, and permutations are extracted from the allocation table.
EXTERNAL REFERENCES:	L
COMMON:	ALAPRM, ALACRD, ALAEXT
SIZE:	
CALLING SEQ.:	RESULT = ARAOLP (AR1, AR2, DIM)

Args:

AR1	- pointer to the first array in ALLTBL
AR2	- pointer to the second array in ALLTBL
DIM	- dimension of the arrays.
RESULT	- .TRUE. IFF the arrays overlap.

TECHNIQUES: None

2.7 Block Structure Manipulation

These routines deal with adding new blocks, taking space out of existing holes, adding arrays to an allocation for a particular block, etc. These functions involve somewhat more than mere table manipulation and are therefore treated here rather than with the table manipulation routines. The routines covered here are:

ACRBLK -- which creates an entirely new block along with a new hole to match.

ANWARY -- which puts an array in the allocation of a specified block, in the process taking the newly occupied space out of any overlapping holes.

ASPLHL -- which takes the space occupied by a newly allocated array out of a specific hole, often creating several smaller holes in the process.

ARMSPC -- removes the space occupied by new array from all holes in the block which intersect the new array.

ASREXT -- a utility routine which sorts the extents of a newly created structure (usually a hole), into descending order.

ASTPRM -- a utility routine whose primary purpose is to find the inverse of a permutation vector.

PROGRAM NAME: ACRBLK

Allocation - CReate BLocK

PURPOSE: To create a new block and hole for allocating arrays.

FUNCTION: ACRBLK is given an array which must fit in the new block. ACRBLK calculates which extent may most efficiently be expanded to an even rowsize and then creates a block with the same extents except for the one expanded dimension. It is unnecessary to sort the block extents as the extents are still in descending order even after expanding the preferred index. After it has been decided which extent (say J) is to be the preferred index, the block permutation vector is filled in with

PERM (I) = I   FOR ALL I <> 1, J

PERM (1) = J

PERM (J) = 1.

After filling in all necessary fields for the new block ANWHL is called to create a hole with the same extents as the new block (since no arrays have been taken from the new block all of its space is available) and both the hole and the block are added to the appropriate tables.

EXTERNAL REFERENCES: AGTBL, S, L, ANWHL, ADDBLK

COMMON: Block table field definitions

SIZE:

CALLING SEQ.:	CALL ACRBLK (DIM, ARAPNT, EXTFLD, HOLPNT)	
	Args:	
	DIM	- dimension of the block to be created
	ARAPNT	- table pointer to array to be fit by the block
	EXTFLD	- field pointer vector for the extent fields of the specified array.
	Results:	
	HOLPNT	- points to the location of the newly created hole.
TECHNIQUES:	None	

PROGRAM NAME: ANWARY

Allocation NEW ARray

PURPOSE: To put an array in the space occupied by a particular hole.

FUNCTION: ANWARY is given pointers to an array and a hole as arguments. ANWARY determines an orientation of the array which allows it to fit in the hole. It then puts the array in the allocation table and adds it to the allocation list for the appropriate block. Finally, ANWARY calls ASPLHL to remove the newly allocated space from the hole.

EXTERNAL REFERENCES: AGTALL, ASTPRM, S, L, ADDARY, ASPLHL

COMMON: ALLTBL, and ARRAYTBL field pointers, INVPRM, WRK

SIZE:

CALLING SEQ.: CALL ANWARY (ARAPNT, HOLPNT)

Args:

ARAPNT - points to the array (in ARRAYTBL) to be allocated

HOLPNT - points to the hole into which the array is placed.

TECHNIQUES: An attempt is made to leave the largest possible hole after allocating the new array. This is done by placing the largest array dimension in the smallest possible hole dimension, the next largest array dimension in the smallest hole dimension still available, etc. Thus, if an array which was 16 x 5 x 4 was taken out of a hole which was 32 x 16 x 5 then the array would be oriented as a 4 x 16 x 5 array before being taken out of the hole, thus leaving one hole 28 x 16 x 5 whereas if the array were allocated as a 16 x 5 x 4 array 3 holes would result (16 x 5 x 4, 32 x 16 x 1, 32 x 11 x 5) but none of the holes would be as large. It is hoped that trying to produce the largest possible holes will in general produce better allocations, although there are obviously instances in which it won't. Some sort of look ahead for deciding the proper orientation might eventually be useful.

PROGRAM NAME: ASPLHL

Allocation - SPLit HoLe

PURPOSE: To remove space from a hole, breaking the hole up into several smaller holes.

FUNCTION: ASPLHL is given a pointer to an array in ALLTBL and a hole in HOLETBL. ASPLHL first calls AOVLP to insure that the array overlaps the hole. If the array does not overlap ASPLHL merely returns. If the array does overlap ASPLHL proceeds by calling ARMHL to remove the hole from the active chain without adding it to the free chain. This is done so that subsets of the specified hole may be added to HOLETBL. The hole is not yet added to the free chain since the entry can not be reused until the hole has been completely split up. At this point ASPLHL checks the endpoints of the array against the endpoints of the hole in each dimension. ASPLHL creates one new hole for each pair of endpoints which do not correspond. Thus, ASPLHL could possibly produce 2 * N new holes (where N is the dimensionality of the array). As an example, assume that the hole starts at (5, 10, 12) and has extents (7, 20, 4) and that the array starts at (6, 10, 12) and has extents (3, 9, 4) then the following holes would be produced:

Coordinates	Extents
(5, 10, 12)	(1, 20, 4)
(9, 10, 12)	(3, 20, 4)
(5, 19, 12)	(7, 11, 4)

After all new holes are produced the original hole is released (added into the free chain) and ASPLHL returns.

EXTERNAL REFERENCES:	AOVLP, ARMHL, AGTHOL, AMVENT, S, L, ASREXT, ADHOLE, AHLRLS
COMMON:	HOLETBL and ALLTBL field pointers.
CALLING SEQ.:	CALL ASPLHL (ARRAY, HOLE)

Args:

ARRAY - pointer to array in ALLTBL

HOLE - pointer to the hole in HOLETBL.

TECHNIQUES:	None

PROGRAM NAME:	ARMSPC Allocation - ReMove SPaCe
PURPOSE:	To remove space from all holes which a newly allocated array overlaps.
FUNCTION:	ARMSPC is given an array pointer to an array in ALLTBL and a block pointer. ARMSPC merely threads through the hole chain, checking to see if any holes in the specified block overlap the specified array. If they do ASPLHL is called to break up the hole.
EXTERNAL REFERENCES:	L, AOVLP, ASPLHL
COMMON:	AHLNXT, AHLBLK
SIZE:	
CALLING SEQ.:	CALL ARMSPC (ARRAY, BLK) Args: ARRAY - points to an array in ALLTBL BLK - pointer to the block to which the array is allocated.
TECHNIQUES:	None

PROGRAM NAME:	ASREXT Allocation - SoRt EXTents
PURPOSE:	To get extent fields in descending order.
FUNCTION:	ASREXT calls ASTPRM to get the inverse of the permutation vector. It then sorts the EXTENT field using an interchange sort, updating the inverse permutation vector along with the extent fields. After the sort is completed ASREXT effectively takes the inverse of the modified inverse permutation vector in order to get the new permutation vector.
EXTERNAL REFERENCES:	ASTPRM
COMMON:	INVPRM
SIZE:	
CALLING SEQ.:	CALL ASREXT (PNT, PERM, EXT, DIM)

Args:

PNT	- points to the entry to be sorted
PERM	- the vector of permutation field pointers for the table containing the desired entry
EXT	- the vector of extent field pointers
DIM	- the dimension of the structure.

TECHNIQUES:	Since all required field pointers are given as arguments ASREXT may be used to sort the extents of any structure (hole, array, or block).

PROGRAM NAME: ASTPRM

PURPOSE: To take the inverse of the permutation vector and to set the WRK array to .FALSE.

FUNCTION: ASTPRM sets all entries in WRK to .FALSE. (This array is used in ANWARY to keep track of when an extent has been allocated) and establishes the inverse of the permutation vector in INVPRM. This is done by setting INVPRM (PERM (I)) = I for all I.

EXTERNAL REFERENCES: L

COMMON: WRK, INVPRM

SIZE:

CALLING SEQ.: CALL ASTPRM (ENTPNT, PERM, DIM)

Args:

ENTPNT - pointer to the entry containing the permutation vector

PERM - field pointer for the permutation vector

DIM - dimension of the vector.

TECHNIQUES: None

2.8 Hole Expansion

Two routines, AINFL8 and AINFL1, provide the mechanism for expanding holes. AINFL1 is a recursive routine which expands the hole in one direction. AINFL8 calls AINFL1 for each direction in which the hole must be expanded and then updates HOLETBL to reflect the new structure of holes (AINFL1 does not alter the hole table).

PROGRAM NAME:	AINFL8 Allocation - INFLate
PURPOSE:	To expand or inflate a hole.
FUNCTION:	AINFL8 determines how much the specified hole must be expanded in each direction to fit the desired array. It then calls AINFL1 for each direction to move arrays out of the way and expand the block. After moving all necessary arrays it deletes all holes associated with the block and creates one new hole the size of the entire new block. It then calls ARMSPC for each array in the block in order to construct the correct list of holes.
EXTERNAL REFERENCES:	L, S, ASTPRM, AINFL1, ACLRBL, ARMSPC
COMMON:	All table field definitions, INVPRM
SIZE:	
CALLING SEQ.:	CALL AINFL8 (ARY, HOLE) Args: ARY - the array to be fit HOLE - the hole to be expanded to fit the array.
TECHNIQUES:	None

PROGRAM NAME:	AINFL1 Allocation - INFLate in 1 dimension
PURPOSE:	To expand a hole in one dimension.
FUNCTION:	AINFL1 creates a dummy allocated array expanded appropriately and then checks to see if the array overlaps any other arrays. If so, the offending arrays are translated and AINFL1REC (the recursive part of AINFL1) is called to move any arrays which overlap. When no more arrays overlap then AINFL1REC either pops back up a level or returns to the caller if it is already at level 0. When AINFL1 returns, all necessary arrays have been translated and the highest coordinate encountered is recorded for use by AINFL8.
EXTERNAL REFERENCES:	AINFL1REC, L, S, AGTALL, ALLRLS
COMMON:	ALLTBL and HOLETBL field definitions.
SIZE:	
CALLING SEQ.:	CALL AINFL1 (HOLE, DIR, AMNT, MAXCRD)

Args:

HOLE	- the hole to be expanded
DIR	- the direction for expansion
AMNT	- the amount to expand.

Results:

MAXCRD	- the highest coordinate occupied by any array moved.

TECHNIQUES: None

PROGRAM NAME: ACOST

Allocation - COST

PURPOSE: To estimate the cost of putting an array into an existing hole.

FUNCTION: Expanding a hole will, in general, expand the block which contains the hole. It is the increase in the volume of the block which determines the cost of expanding a given hole. If the hole is expanded along a direction in which the hole is not "internal" to the block, then the block must expand by the same amount as the hole (a hole is considered internal in a given direction if the right boundary of the hole occurs before the right boundary of the block in that direction, if the two boundaries coincide then the hole is not considered to be internal). If a hole is internal then the block may be expanded less than the hole. If a block is expanded in a preferred direction then the block must be expanded by a multiple of ROWSZE (the number of PE's in a row).

ACOST compares each array extent with the corresponding row extent (largest array extent compared to largest hole extent, etc.). If the hole must be expanded in some direction then the volume of the block must be multiplied by (new extent) / (old extent) (note that volume/(old extent) is always an integer), if the hole is internal then the volume is additionally multiplied by IFACT/100 (IFACT currently is 95) -- this merely means that of two holes which are nearly the same size, preference is given to putting a new array into an internal hole. If the preferred direction must be expanded (which implies a quantum jump in block size) then ARLCST is called (if the hole is internal), to determine the exact cost, since it may turn out to be possible to expand the hole for free. If no expansion in the preferred direction is necessary, then the estimated cost is merely the increase in volume resulting from expanding each array the necessary amount multiplied once by IFACT/100 for each expansion along an internal direction (multiplication by IFACT is done before the increase in volume is measured).

EXTERNAL REFERENCES: ASTPRM, L, ARLCST

COMMON: INVPRM, BLOCKTBL field definitions, HOLETBL field definitions.

SIZE:

CALLING SEQ.: CALL ACOST (ARAPNT, HLEPNT, COST)

Args:

ARAPNT - points to an array in ARRAYTBL

HLEPNT - points to a hole in HOLETBL.

Results:

COST - estimated cost of putting the array in the hole.

TECHNIQUES: None

PROGRAM NAME: ARLCST

Allocation - ReaL CoST

PURPOSE: To determine the exact cost of putting a specified array into a specific hole.

FUNCTION: ARLCST makes a copy of the block containing the specified hole, all of the arrays allocated to the block, and the specified hole itself. It then calls AINFL8 to put the array into the hole in the copy and records the increase in size (if the increase is 0 ARLCST sets the cost to 1 since a cost of 0 is assumed to mean the array fits into the hole without hole expansion). Before returning ARLCST deletes the duplicate blocks, arrays, and holes.

There are several improvements which could be made to ARLCST. First, AINFL8 automatically recreates all the holes in a block after expanding a particular hole. This could be suppressed for ARLCST since the newly created holes are immediately deleted. Secondly, and of more interest to the allocation, ARLCST could determine the maximum amount the hole could be expanded in the preferred direction without increasing the block size. It could then determine if any of the extents of the array would fit in the hole as so expanded. If so, the largest possible extent would be placed in the preferred direction. The remaining extents would still have to be placed. This could be done by determining MAXEX (I) (the maximum that a hole could be expanded to in direction I without expanding the block, once the hole is expanded this much increasing the hole size by one would increase the block size by one also. MAXEX could be determined by expanding the hole by a very large amount and then determining how much the block expanded). After determining the MAXEX the array extents should then be oriented so as to minimize the product, over all I for which the hole was expanded, of the factors (EXTENT (I) - MAXEX (I)) / BLKEX (I) where only I's for which EXTENT (I) - MAXEX (I) > 0 are considered and where EXTENT (I) is the extent of the array to be put in the Ith direction and BLKEX (I) is the block extent in the Ith direction. If this change were made the optimum orientation would have to be communicated to the calling program.

EXTERNAL REFERENCES: ABLCPY, AINFL8, L, ADLBLK

COMMON: BLOCKTBL and HOLETBL field definitions

SIZE:

CALLING SEQ.: CALL ARLCST (ARAPNT, HLEPNT, COST)

Arg:

ARAPNT - pointer to array in ARRAYTBL

HLEPNT - pointer to hole.

Results:

COST - cost of putting array in hole.

2.10 Allocation

AL2 is the routine which actually does the allocation. SINCE THE BASIC ALGORITHM HAS BEEN DESCRIBED ELSEWHERE, LITTLE WILL BE SAID ABOUT IT HERE.

PROGRAM NAME:	AL2 Allocation Number 2
PURPOSE:	To perform an allocation.
FUNCTION:	For each array AL2 calls ACOST to determine the cost of putting an array in each of the existing holes. If the estimated cost for a hole is zero it is assumed that the array fits and AL2 calls AFLDCM to verify this. If the array does fit ANWARY is called to put the array in the hole. If the array does not fit in any hole, then AL2 determines whether or not it is cheaper to create a new block (and hole) or to expand an existing one. If AL2 expands a hole it then goes back and searches for a hole which will fit the array again. Since there now exists a sufficiently large hole the array will be placed and AL2 will go on to consider the next array. If AL2 creates a new hole it immediately takes the array out of the new hole. Note that AL2 could be made more efficient when expanding holes if it knew where the newly expanded hole was in the table. This information is difficult to obtain, nevertheless, AL2 could still be improved by avoiding the call to ACOST (which occassionally makes very expensive calls to ARLCST) and just searching for the hole which contains the array.
EXTERNAL REFERENCES:	ACOST, L, AFLDCM, ANWARY, AINFL8, ACRBLK
COMMON:	HOLETBL and ARRAYTBL field definitions.
SIZE:	
CALLING SEQ.:	CALL AL2
TECHNIQUES:	None

2.11 Auxillary Routines

These routines (ASORT, ASIZE, ADDR, AMVENT) all are obvious routines to do straightforward but necessary functions:

ASORT -- sorts ARRAYTBL into descending size so that the largest arrays will be allocated first.

ASIZE -- computes the size of a structure by multiplying the extent fields.

ADDR -- returns the address of a word so that word pointers may be created in FORTRAN.

AMVENT -- copies an entry from one spot in a table to another spot.

2.11 Initialization

ALLINT sets up tables necessary for allocation.

PROGRAM NAME:	ALLINT ALLocation INiTialization
PURPOSE:	Initialize the allocation.
FUNCTION:	Zeros all tables, obtains some initial core for tables, initializes byte pointers and table headers. Initializes TBLPNT.
EXTERNAL REFERENCES:	SETFF, XCORE, CSETZ, PFIX, ADDR
COMMON:	Table definitions, TBLPNT, ALLCTB, COREND, XCES size.
CALLING SEQ.:	CALL ALLINT
TECHNIQUES:	None

3. MACRO EXPANSION

A. Array Reference

This section discusses the array access formulations required by the macro expansion and optimization phases of code select.

The skewed storage technique described in the first semi-annual report provides the basis for this discussion. For reference, the results presented were:

An element, $A(i_1, i_2, \ldots, i_n)$ of an array. $A(\hat{i}_1, \hat{i}_2, \ldots \hat{i}_n)$ is located by the two parameters ℓ_1 (PE number) and ℓ_2 (PEM displacement).

Let: P = number of PE's

$$Q = \sum_{j=1}^{n} (i_j - 1)$$

Define: $R_{m_1}^{m_2} = \prod_{j=m_1}^{m_2} \hat{i}_j$; $R_{m_1}^{m_1-1} = 1$; $R_{m_1}^{m_1-2} = Q$

$$M = \left\lceil \frac{\hat{i}_n}{P} \right\rceil R_1^{n-1} \quad \text{(block modulus)}$$

Then: $\ell_1 = Q \bmod P$

$$\ell_2 = \Big(\underbrace{\sum_{k=1}^{n-1} (i_k - 1)\, R_{k+1}^{n-1}}_{I} + \underbrace{\left\lfloor Q/P \right\rfloor R_1^{n-1}}_{II} \Big) \bmod M$$

This allocation formula implies that the first array element resides in PE O at displacement O, and that the n^{th} index is the preferred index. We must extend it to include arbitrary index preferences and initial locations. But first, examining term I observe that

$$\sum_{k=1}^{n-1} (i_k - 1) R_{k+1}^{n-1} \leq \sum_{k=1}^{n-1} (\hat{i}_k - 1) R_{k+1}^{n-1} = \sum_{k=1}^{n-1} R_k^{n-1} - \sum_{k=1}^{n-1} R_{k+1}^{n-1}$$

$$= R_1^{n-1} - R_n^{n-1} = R_1^{n-1} - 1$$

now $\ell_2 = I + II - \left\lfloor {}^{I+II}/_{M} \right\rfloor M$

but $\left\lfloor {}^{I+II}/_{M} \right\rfloor = \left\lfloor {}^{I}/_{M} + {}^{II}/_{M} \right\rfloor = \left\lfloor {}^{I}/_{M} + {}^{\lfloor Q/P \rfloor}/_{\lceil \hat{i}_n/P \rceil} \right\rfloor$

$$= \left\lfloor \left(\left({}^{I}/_{R_1^{n-1}} \right) + \lfloor Q/P \rfloor \right) / \left\lceil {}^{\hat{i}_n}/_{P} \right\rceil \right\rfloor = \left\lfloor {}^{\lfloor Q/P \rfloor}/_{\lceil \hat{i}_n/P \rceil} \right\rfloor$$

since ${}^{I}/_{R_1^{n-1}} < 1$

$$\text{therefore} \quad \ell_2 = I + II - \overbrace{\left\lfloor \lfloor Q/P \rfloor / \left\lceil {}^{\hat{i}_n}/_{P} \right\rceil \right\rfloor}^{III} M$$

$$= I + \left(\lfloor Q/P \rfloor - \left\lfloor \lfloor Q/P \rfloor / \left\lceil {}^{\hat{i}_n}/_{P} \right\rceil \right\rfloor \right) M$$

thus the formula ℓ_2 reduces to

$$\ell_2 = \sum_{k=1}^{n-1} (i_k - 1) R_{k+1}^{n-1} + \left(\lfloor Q/P \rfloor \bmod \left\lceil {}^{\hat{i}_n}/_{P} \right\rceil \right) * R_1^{n-1}$$

the modulus in this expression being much easier to calculate in certain situations.

Let us assume that the allocation program produces a permutation of the indices which orders the preferred index last. This permutation, T, will have n members as follows:

T_1 = least preferred index (in usual terms, the index which varies least rapidly)

.
.
.

T_n = preferred index

It is necessary to assume that multi-indices have been transformed into the appropriate single index. We can now write a more general allocation formula:

$A(i_1, i_2, \ldots, i_n)$ of the array $A(\hat{i}_1, \hat{i}_2, \ldots \hat{i}_n)$ is located by the parameters ℓ_1 and ℓ_2.

Let: Q_o = PE number of first element of A

L_o = PEM displacement of first element of A

P = number of PE's

T be the index ordering

Define: $R_{m_1}^{m_2} = \prod_{j=m1}^{m_2} \hat{i}_j$; $R_{m1}^{m1-1} = 1$; $R_{m1}^{m1-2} = O$

$$Q = Q_o - n + \sum_{j=1}^{n} i_j$$

$$M = \left\lceil \frac{\hat{i}_{T_n}}{P} \right\rceil R_{T_1}^{T_{n-1}}$$

Then: $\ell_1 = Q \bmod P$

$$\ell_2 = L_o + \sum_{j=1}^{n-1} (i_{T_j} - 1)\, R_{T_{j+1}}^{T_{n-1}} + R_{T_1}^{T_{n-1}} \left(\left\lfloor \frac{Q}{P} \right\rfloor \bmod \left\lceil \frac{\uparrow T_n}{P} \right\rceil \right)$$

This formula can also describe FORTRAN standard allocation and aligned indices by dropping appropriate forms. The macro expander and optimizer are designed around the existence of an allocation program which assigns values to L_o, Q_o, M and the R_n skew factors for each array. The macro expander uses these constants to replace all references to array elements with the arithmetic expressions required in order to calculate ILLIAC addresses, and the optimizer then optimizes the resultant expressions. We expect that the optimization of these calculations will be among the most fruitful for all our optimizations.

The macro expander deals with two cases -- array element references occurring in sequential code, and references within DO FOR ALL scopes -- the latter being the crucial case:

1. References in sequential code:

The macro expander turns each reference of the form $A(i_1 \ldots i_n)$ into $A(Q, \ell_2)$, where Q and ℓ_2 are the expanded forms of the given formulas. These are transformed by the optimizer by moving invariant calculations, reducing operator strength, etc. The technique is straightforward and comparable to that used by typical FORTRAN compilers.

2. References in DO FOR ALL scopes:

The problem which immediately confronts us is that we must use a technique which makes the simultaneous calculations made by all enabled PE's explicit in a single formula. A PE-dependent function and certain transformations of IVTRAN programs (in intermediate language) gives the necessary formulations.

Macro Expander creates a DO statement immediately before each DO FOR ALL statement, using the same index variable, deriving the range of the DO statement from the control set of the DO FOR ALL, and using an increment of 64. By convention, the DO statement expresses the use of the index variable for address calculation and also is the required DO statement if the range of the DO FOR ALL > 64. The DO FOR ALL statement provides PE control information and defines the scope of parallel operation.

With this done, we may pass on to the array references within the scope of a DO FOR ALL. Each reference defines a slice of the array made up to P members, one in each PE. We wish to express the calculation in each PE of the PEM displacement of its member of the slice. Note that each slice has what may be called a base member -- the array element for which the value of Q is minimum. This value, Q', is calculated from the DO index variables.

If the DO FOR ALL scope is controlled by a DO statement of the form:

$$\text{DO label } i_m = e_1, e_2, 64$$

where e_1 and e_2 are the minimum and maximum values of the DO FOR ALL index i_m derived from the DO FOR ALL statement itself: then

$$Q' = Q_o - n + \sum_{j=1}^{n} i_j$$

where i_m has the current value of the DO index variable and the other i_j are defined outside the loop.

Calculation of the constants appearing in the access formula is handled by the Allocator, and placed in an IL table (YTAB).

The contents of each entry (one per array) in YTAB will include:

Constant	Definition
m	preferred index number (Tn)
R_k, k = o...n	$R_k = R^{T_n - 1}_{T_{k + 1}}$
$\left\lceil \frac{\uparrow m}{P} \right\rceil$	number of rows to hold preferred index
Q_f	PE fudge factor, usually $Q_f = Q_o - n$ Allocator techniques may introduce further constant factors
L_f	PEM fudge factor: usually $L_f = L_o - \sum_{j=1}^{j=n} R_j$ but some allocation technicques may introduce further constant factors. (Note: the L_o above will probably be the relative displacement in a block of the array, and the loader will supply the block address.)

The expansion of array references in the intermediate language takes the following form:

1. Replace all multi-indices with the correct single index

2. For each array reference in sequential code:

   a. generate the following IVTRAN statement preceding the statement containing the reference

$$Q = Q_f + \sum^{j} i_j$$

2. b. Replace the reference $A\,(i_1 \ldots i_n)$ with $A(\ell_1, \ell_2)$ where ℓ_1 and ℓ_2 are the above expressions.
Note: ℓ_1 and ℓ_2 may be combined into a single CU address. This is as yet unresolved.

3. For each array reference in parallel code:

a. Generate two IVTRAN statements preceding the statement containing the reference, as follows:

$$Q = Q_f + \sum^{j} i_j$$

$$DQ = NPMOD\ (Q)$$

b. Replace the reference $A(i_1 \ldots i_n)$ with $A(\ell_2)$, where

$$\ell_2 = L_f + \sum^{j} i_j R_j + DQ * R_{DFA} + R_o \left(\left\lfloor \frac{Q+DQ}{P} \right\rfloor \bmod \left\lceil \frac{\uparrow m}{P} \right\rceil \right)$$

note: NPMOD may be defined as:
NPMOD = Q mod P - PENO
IF (NPMOD.LT.O) NPMOD = NPMOD + P

B. Flow Analyzer

Introduction

The Flow Analyzer is a package of FORTRAN routines which performs flow analysis of IVTRAN programs. The Flow Analyzer separates the IVTRAN program in flow blocks and regions, and builds up the flow block and region connectivity matrix. The entire package is designed so that it can be called at any point of the compilation process.

Flow blocks are contiguous sets of statements with a single entry point and a single exit point. Regions are contiguous sets of flow blocks with single entry and exit points.

Flow blocks and regions are recorded in a table (VTAB) one block per entry. The connectivity matrix is stored in the IL table MATAB, it is a bit matrix: C(I, J) = 1 if control can transfer from block I to block J, C(I, J) = 0 otherwise. Support routines exist to make inquiries in the connectivity matrix.

PROGRAM NAME:	MXPHP
PURPOSE:	To perform flow analysis of an IVTRAN program.
FUNCTION:	Drives program to separate the IVTRAN program in flow blocks, program to separate IVTRAN program in regions, and program to build the connectivity matrices.
EXTERNAL REFERENCES:	MXFLOW, MXREGN, MXCOMA
COMMON REQUIREMENTS:	TCOM.DEF, TABL.EQU, CGLOB.DEF
SIZE:	
CALLING SEQUENCE:	CALL MXPHP
TECHNIQUES:	None

PROGRAM NAME:	MXFLOW
PURPOSE:	To separate IVTRAN program in flow blocks.
FUNCTION:	Builds flow block entries in VTAB, one entry per flow block. Records flow block number in statement level CTAB entries.
EXTERNAL REFERENCES:	CSETZ, L, S, MXVOPN, MXVCLS, UCLNST
COMMON REQUIREMENTS:	TCOM.DEF, CTAB.DEF, VTAB.DEF, LTAB.DEF
SIZE:	
CALLING SEQUENCE:	CALL MXFLOW
TECHNIQUES:	MXFLOW traces through CTAB at the statement level. If statement starts a new flow block, the current entry in VTAB is closed and a new one is initialized. If a statement ends a flow block, the current VTAB entry is closed and FLAG is set, indicating that the successive statement begins a new flow block.

PROGRAM NAME:	MXCOMA
PURPOSE:	To build flow block connectivity matrix.
FUNCTION:	Builds a bit matrix in MATAB such that C(I, J) = 1 if control transfers from flow block I to flow block J, C(I, J) = 0 otherwise.
EXTERNAL REFERENCES:	CSETZ, EXTEND, L, MXMSET
COMMON REQUIREMENTS:	TCOM.DEF, TABL.EQU, VTAB.DEF, MATAB.DEF
SIZE:	
CALLING SEQUENCE:	CALL MXCOMA
TECHNIQUES:	MXCOMA looks at each flow block entry in VTAB. For each flow block, the last statement of the flow block is used to determine which flow blocks can be reached from the current one.

PROGRAM NAME:	MXVOPN
PURPOSE:	To open a flow block VTAB entry.
FUNCTION:	Stores flow block number and pointer to first statement in flow block in a new VTAB entry.
EXTERNAL REFERENCES:	S, EXTENT
COMMON REQUIREMENTS:	TCOM.DEF, VTAB.DEF, TABL.EQU
SIZE:	
CALLING SEQUENCE:	CALL MXVOPN (CNDX) CNDX - CTAB pointer to first statement in flow block.
TECHNIQUES:	None

PROGRAM NAME:	MXVCLS
PURPOSE:	To close a flow block VTAB entry.
FUNCTION:	Stores CTAB pointer to last statement in flow block, and last statement's class in current VTAB entry.
EXTERNAL REFERENCES:	S
COMMON REQUIREMENTS:	TCOM.DEF, VTAB.DEF
SIZE:	
CALLING SEQUENCE:	CALL MXVCLS (CNDX, CLASS)

CNDX - CTAB pointer to last statement in flow block

CLASS - statement class of last statement in flow block.

CLASS = 1 : GOTO, AGOTO, CGOTO
AIF, LIF2, LIF

CLASS = 2 : RETURN, END, STOP

CLASS = 3 : CALL

CLASS = 4 : terminal statement of DO (DFA) range

TECHNIQUES: None

PROGRAM NAME:	MXMSET
PURPOSE:	To set to one C(I, J), where C is the connectivity matrix.
FUNCTION:	Sets a bit in MATAB.
EXTERNAL REFERENCES:	MXMTST, L, S
COMMON REQUIREMENTS:	TCOM.DEF, MATAB.DEF, VTAB.DEF
SIZE:	
CALLING SEQUENCE:	CALL MXMSET (BF, BT) BF - block number control may transfer from BT - block number control may transfer to
TECHNIQUES:	None

PROGRAM NAME: MXMTST

PURPOSE: To test whether control may flow from one flow block to another. The value of the function is .TRUE. if the test succeeds, .FALSE. if the test fails.

FUNCTION: Tests a list in the connectivity matrix.

EXTERNAL REFERENCES: L

COMMON REQUIREMENTS: TCOM.DEF, MATAB.DEF, VTAB.DEF

SIZE:

CALLING SEQUENCE: MXMTST (BF, BT)

BF - flow block number control is to transfer from

BT - flow block number control is to flow to.

TECHNIQUES: None

PROGRAM NAME: MXMTO

PURPOSE: To extract from the flow block connectivity matrix all the numbers of the flow blocks which may transfer control to a specified flow block (i.e. - all I's such that C(I, J) = 1 for a given J).

FUNCTION: Stores flow block numbers in an array passed in the calling sequence.

EXTERNAL REFERENCES: L

COMMON REQUIREMENTS: TCOM.DEF, MATAB.DEF, VTAB.DEF

SIZE:

CALLING SEQUENCE: CALL MXMTO (BT, ARRAY)

BT - flow block number transferred to

ARRAY - array used to store all flow block numbers which can transfer control to BT. A -1 is stored in ARRAY following the last block number.

TECHNIQUES: None

PROGRAM NAME:	MXMFRM
PURPOSE:	To extract from the flow block connectivity matrix all the numbers of the flow blocks which can be reached from a specified flow block (i.e. - all J's such that C(I, J) = 1 for a given I).
FUNCTIONS:	Stores flow block numbers in an array passed in the calling sequence.
EXTERNAL REFERENCES:	L
COMMON REQUIREMENTS:	TCOM.DEF, MATAB.DEF, VTAB.DEF
SIZE:	
CALLING SEQUENCE:	CALL MXMFRM (BF, ARRAY)

BF	- flow block number to transfer from
ARRAY	- array used to store all flow block numbers which can be transferred to from flow block BF. A -1 is stored in ARRAY following the last flow block number.

TECHNIQUES:	None

C. Macro Expansion - Phase B

Macro Expansion - Phase B (MXPHB) has been completed and debugged. MXPHB has served as a test-bed for the debugging of the Macro Expansion support package documented elsewhere.

MXPHB transforms all implied loops in the IVTRAN program into explicit loops. An implied loop exists whenever an array reference appears (with an asterisk appearing in one or more index positions) or when an array name appears without an index expression, implying an asterisk in every index position.

MXPHB locates all array references in the program by using the LINK chain associated with each array name. When an array reference is found in which one or more asterisks appear, MXPHB locates all other matching array references in that statement, then creates a DO FOR ALL statement with the original IVTRAN statement as its scope, and replaces one asterisk in each reference with the DO FOR ALL index. If asterisks still exist, the DO FOR ALL - IVTRAN statement pair is nested in a DO loop with the DO index replacing another asterisk. This process continues until all asterisks are replaced.

MXPHB copies all LINK chains in a work area (WTABLE). It then examines each array reference in CTABLE, and if the entry contains no asterisks, the LINK copy in WTAB is deleted. When an asterisk is found, explicit loops are created as above, and then the appropriate LINK copies are deleted. When all entries in WTAB become zero, MXPHB is done.

When an array reference has more than one asterisk, the first one encountered is replaced by the DO FOR ALL index. This naive approach will be replaced in MXPHB version 2. Creation of multi-indices will be considered at the same time.

The root program for Macro Expansion (MX) and Macro Expansion Phase B (MXPHB), with attendant sub-programs, are documented below. The naming convention for the programs is:

1st letter	-	M
2nd letter	-	phase letter, or U if used by more than one phase
3rd letter	-	action taken (e.g., L - locale)
4th - 6th letter	-	rough mnemonic for operand(s)

PROGRAM NAME:	MBISEX (LOGICAL FUNCTION)
PURPOSE:	Is this an explicit array reference?
FUNCTION:	CNDX must be an OPRAND pointing to an ARRAY element in STAB. If CNDX is the first OPRAND of an ARRAY EXOP, and if none of the other OPRAND's in the EXOP are asterisks (CTABL.EQ.ASTER) then MBISEX ←.TRUE., else MBISEX ←.FALSE. Error trap on improper argument.
EXTERNAL REFERENCES:	INSERTS: TCOM.DEF, CTAB.DEF, STAB.DEF, TABL.EQU, NOYES.EQU ROUTINES: L, UEHALT, UCLCNH
COMMON:	TCOM, CTABLE, STABLE
CALLING SEQUENCE:	MBISEX (CNDX)
TECHNIQUES:	

PROGRAM NAME:	MBISRF (LOGICAL FUNCTION)
PURPOSE:	Is this an array reference?
FUNCTION:	If CTABLE element CNDX occurs in an ARRAY EXOP, MBISRF← .TRUE., else MBISRF ←.FALSE.
EXTERNAL REFERENCES:	INSERTS: TCOM.DEF, CTAB.DEF
	ROUTINES: L, UCLCNH
COMMON:	TCOM, CTABLE
CALLING SEQUENCE:	MBISRK (CNDX)
TECHNIQUES:	

PROGRAM NAME: MBLNAS (INTEGER FUNCTION)

PURPOSE: Locate next asterisk.

FUNCTION: CNDX must be an ARRAY EXOP with at least one asterisk in its index list. MBLNAS is set to the index value of the first OPRAND in the list for which CTABL.EQ.ASTER. Error trap on improper argument or no asterisk.

EXTERNAL REFERENCES: INSERTS: TCOM.DEF, CTAB.DEF, TABL.EQU, NOYES.EQU

ROUTINES: L,UEHALT

COMMON: TCOM, CTABLE

CALLING SEQUENCE: MBLNAS (CNDX)

TECHNIQUES:

PROGRAM NAME: MBMARF (SUBROUTINE)

PURPOSE: Make array reference.

FUNCTION: CNDX is an OPRAND pointing to an ARRAY element in STAB, but CNDX does not appear in an ARRAY EXOP. Replace this OPRAND with an ARRAY EXOP whose first OPRAND points to the same STAB elements, and whose entire index list is asterisks.

EXTERNAL REFERENCES: INSERTS: TCOM.DEF, CTAB.DEF, STAB.DEF, TABL.EQU, NOYES.EQU

ROUTINES: L, S, UCMSKE, UCMCPY, USDLNK, UCBOPR, USUWCY

COMMON: TCOM, CTABLE, STABLE

CALLING SEQUENCE: CALL MBMARF (CNDX)

TECHNIQUES:

PROGRAM NAME:	MBRPAS (SUBROUTINE)
PURPOSE:	Replace asterisk with scalar.
FUNCTION:	WNDX is the index in WTAB of a list of array references in CTAB. Replace the first asterisk in each of these with a reference to the INT SCALAR at SNDX in STAB.
EXTERNAL REFERENCES:	INSERTS: TCOM.DEF, WTAB.DEF, TABL.EQU
	ROUTINES: L, MBLNAS, UCBOPR
COMMON:	
CALLING SEQUENCE:	CALL MBRPAS (WNDX,SNDX)
TECHNIQUES:	

PROGRAM NAME:	MBRXTN (INTEGER FUNCTION
PURPOSE:	Return extent of asterisk index.
FUNCTION:	CNDX is an array EXOP with at least one asterisk in its index list. Locate the first asterisk and find the extent of the index - available in the ETAB element associated with the STAB entry for this array. Errop trap on improper argument, no asterisk or adjustable extent.
EXTERNAL REFERENCES:	INSERTS: TCOM.DEF, CTAB.DEF, STAB.DEF, ETAB.DEF, TABL.EQU ROUTINES: L, UEHALT
COMMON:	TCOM, CTABLE, STABLE, ETABLE
CALLING SEQUENCE:	MBRXTN (CNDX)
TECHNIQUES:	

PROGRAM NAME:	MULRFS (SUBROUTINE)
PURPOSE:	Locate array references in statement.
FUNCTION:	Locate all the ARRAY EXOP's containing asterisks in the IVTRAN' statement containing CNDX. Make a list of these EXOP's in WTAB, and set WNDX to the index of the list. Error trap on bad program structure or stack overflow.
EXTERNAL REFERENCES:	INSERTS: TCOM.DEF, CTAB.DEF, WTAB.DEF, NOYES.EQU ROUTINES: L, S, UEHALT, UCLTST, UCLSTO, UGZSTK, UGPUSH, UGPOP, MBISEX, UWDLNK, UGTELM
COMMON:	TCOM, CTABLE, WTABLE
CALLING SEQUENCE:	CALL MULRFS (CNDX, WNDX)
TECHNIQUES:	

PROGRAM NAME:	MUMDFA (SUBROUTINE)
PURPOSE:	Make DO FOR ALL statement.
FUNCTION:	CNDX is an IVTRAN statement. Build a DO FOR ALL skeleton around it consisting of a preceding FORALL statement and a following labelled CONTIN statement. Set DNDX to the index of the FORALL statement.
EXTERNAL REFERENCES:	INSERTS: TCOM.DEF, CTAB.DEF, LTAB.DEF, TABL.EQU, NOYES.EQU, TYPE.EQU ROUTINES: L, S, UCMLBL, UCMSKS, UCCSTA, UCASTA, UCLPST, UCBOPR
COMMON:	TCOM, CTABLE, LTABLE
CALLING SEQUENCE:	CALL MUMDFA (CNDX, DNDX)
TECHNIQUES:	

PROGRAM NAME: MUMDO (SUBROUTINE)

PURPOSE: Make DO statement.

FUNCTION: DFNDX is a FORALL/DO statement. Create a DO statement preceding it with the same range. If MAKINC.EQ.TRUE. create an OPRAND for an increment value. Set DONDX to the index of the new DO statement.

EXTERNAL REFERENCES: INSERTS: TCOM.DEF, CTAB.DEF, TABL.EQU, TYPE.EQU, NOYES.EQU

ROUTINES: L, UCMSKS, UCMCPY, UCLPST, UCASTA

COMMON: TCOM, CTABLE

CALLING SEQUENCE: CALL MUMDO (DFNDX, DONDX, MAKINC)

TECHNIQUES:

PROGRAM NAME:	MX (SUBROUTINE)
PURPOSE:	Macro Expansion root program.
FUNCTION:	MX creates a STATOP back chain in CBAKOP then calls the subroutines which are the Macro Expansion phases.
EXTERNAL REFERENCES:	INSERTS: TCOM.DEF, CTAB.DCL
	ROUTINES: L, S
COMMON:	TCOM, CTABLE
CALLING SEQUENCE:	CALL MX
TECHNIQUES:	

PROGRAM NAME: MXPHB (SUBROUTINE)

PURPOSE: Macro Expansion Phase B main program

FUNCTION: MXPHB proceeds in the following steps:

0. Initialization; zero WTAB, chain all SARRAY element in STAB. Copy all LINK chains into WTAB.
1. Eliminate each LINK copy in WTAB for which no asterisks appear; if an SARRAY reference is found not in an ARRAY EXOP, create an ARRAY EXOP with all asterisks.
2. Pass through all copies in WTAB again. For each non-zero link copy:
   a. Locate all ARRAY EXOPS in that statement with asterisks, then delete the LINK copies from WTAB
   b. Write a FORALL statement around the given statement, replacing one asterisk with a generated integer scalar
   c. If asterisks remain, write DO statements around the existing group until all asterisks are replaced.

EXTERNAL REFERENCES: INSERTS: TCOM.DEF, CTAB.DEF, STAB.DEF, WTAB.DEF, TABL,EQU, TYPE.EQU, NOYES.EQU

ROUTINES: L, S, CSETZ, USCHAN, USCLNK, MBISEX, MBISRF, MBMARF, MULRFS, UCLTST, MBRXTN, USMSCA, UKMISC, MBRPAS, MUMDFA, UCLNCN, UCBOPR, MUMDO, UKMINT

COMMON: TCOM, CTABLE, STABLE, WTABLE

CALLING SEQUENCE: CALL MXPHB

TECHNIQUES:

D. Macro Expander and Optimizer Support Package

The Macro Expander and Optimizer phases of the IVTRAN compiler share a support package of approximately 90 routines. This package is 85 per cent complete and debugged as of this writing.

The purpose of the support package is to allow dealing with the Intermediate Language representation of an IVTRAN program at a higher level than its basic block and table structure; resulting in more rapid coding and debugging of the Macro Expander and Optimizer, and making these programs more insensitive to changes in the Intermediate Language. The routines in the support package fall into the following categories.

1. Adding or deleting table elements and constructs.

2. Maintaining or modifying program structure.

3. Locating specific program units or structures.

4. Performing symbolic arithmetic on IVTRAN expressions.

Naming Conventions

All routines in the support package are named according to the following conventions:

1st letter	-	U
2nd letter	-	table prefix letter (C, K, S, L, W), or G for general routines, or A for algebraic routines
3rd letter	-	action (e.g., D - delete or decrement, A - add or append, L - locate
4th-6th letters	-	roughly mnemonic for the routine's operand(s).

The routines are grouped by function below.

CN chain routines:

The following routines perform various manipulations of the CN chains attached to EXOP's and STATOP's in the CTABLE.

PROGRAM NAME: UCAOCN (SUBROUTINE)

PURPOSE: Add OPRAND to CN chain

FUNCTION: Add the OPRAND at CNDX2 to a CN chain immediately following the element at CNDX1. CNDX1 may be STATOP/EXOP/OPRAND. Error trap on improper argument.

EXTERNAL REFERENCES: INSERTS: TCOM.DEF, CTAB.DCL, NOYES.EQU

ROUTINES: UEHALT, L, S

COMMON: TCOM, CTABL

CALLING SEQUENCE: CALL UCAOCN (CNDX1, CNDX2)

TECHNIQUES:

PROGRAM NAME:	UCAOCT (SUBROUTINE)
PURPOSE:	Add OPRAND to end of CN chain.
FUNCTION:	Add the OPRAND at CNDX2 to the end of the CN chain containing CNDX1. CNDX1 may be STATOP/EXOP/OPRAND. Error trap on improper argument.
EXTERNAL REFERENCES:	INSERTS: TCOM.DEF, CTAB.DCL, NOYES.EQU
	ROUTINES: UEHALT, L, S, UCLTCN
COMMON:	TCOM, CTABLE
CALLING SEQUENCE:	CALL UCAOCT (CNDX1, CNDX2)
TECHNIQUES:	

PROGRAM NAME: UCLCNH (INTEGER FUNCTION)

PURPOSE: Locate head of CN chain

FUNCTION: Locate the EXOP/STATOP which begins the CN chain containing the OPRAND at CNDX. Error call on improper argument.

EXTERNAL REFERENCES: INSERTS: TCOM.DEF, CTAB.DCL, NOYES.EQU

ROUTINES: L, UEHALT

COMMON:

CALLING SEQUENCE: UCLCNH (CNDX)

TECHNIQUES:

PROGRAM NAME:	UCLNCN (INTEGER FUNCTION)
PURPOSE:	Locate n'th OPRAND
FUNCTION:	Set UCLNCN to the index of the N'th OPRAND on the CN chain containing CNDX, with CNDX counting as 1.
EXTERNAL REFERENCES:	INSERTS: TCOM.DEF, CTAB.DCL
	ROUTINES: L
COMMON:	TCOM, CTABLE
CALLING SEQUENCE:	UCLNCN (CNDX, N)
TECHNIQUES:	

PROGRAM NAME:	UCLPOP (INTEGER FUNCTION)
PURPOSE:	Locate previous element in CN chain.
FUNCTION:	Locate the element in the CN chain which precedes the OPRAND at CNDX by following the chain around the ring. Error trap on improper argument.
EXTERNAL REFERENCES:	INSERTS: TCOM.DEF, CTAB.DCL
	ROUTINES: L, UEHALT
COMMON:	TCOM, CTABLE
CALLING SEQUENCE:	UCLPOP (CNDX)
TECHNIQUES:	

PROGRAM NAME: UCLTCN (INTEGER FUNCTION)

PURPOSE: Locate tail of CN chain.

FUNCTION: Locate the OPRAND in the CN chain containing the element at CNDX for which CLARG.EQ.YES.

EXTERNAL REFERENCES: INSERTS: TCOM.DEF, CTAB.DCL, NOYES.EQU

ROUTINES: L

COMMON: TCOM, CTABLE

CALLING SEQUENCE: UCLTCN (CNDX)

TECHNIQUES:

PROGRAM NAME: UCNOCN (INTEGER FUNCTION)

PURPOSE: Count OPRANDS in CN chain.

FUNCTION: Set UCNOCN to the number of OPRANDS in the CN chain which begins at the element at CNDX.

EXTERNAL REFERENCES: INSERTS: TCOM.DEF, CTAB.DCL, NOYES.EQU

ROUTINES: L

COMMON: TCOM, CTABLE

CALLING SEQUENCE: UCNOCN (CNDX)

TECHNIQUES:

PROGRAM NAME:	UCUOCN (SUBROUTINE)
PURPOSE:	Unlink OPRAND from CN chain
FUNCTION:	Remove the OPRAND at CNDX from its CN chain, and update the chain. Error trap on improper argument.
EXTERNAL REFERENCES:	INSERTS: TCOM.DEF, CTAB.DCL, NOYES.EQU
	ROUTINES: L, S, UEHALT, UCLPOP
COMMON:	TCOM, CTABLE
CALLING SEQUENCE:	CALL UCUOCN (CNDX)
TECHNIQUES:	

LINK chain routines

The following routines maintain the LINK chains which begin at elements in CTABLE, STABLE, KTABLE and LTABLE and use the CLINK field in CTABLE to locate all references to a given element. Also included are routines which create and maintain copies of these chains in the work area table.

PROGRAM NAME:	UCALNK (SUBROUTINE)
PURPOSE:	Add element to CLINK chain.
FUNCTION:	Add the CTAB element at CNDX to the CLINK chain which begins at CLINK (CNDX).
EXTERNAL REFERENCES:	INSERTS: TCOM.DEF, CTAB.DCL
	ROUTINES: L, S, UGALNK
COMMON:	TCOM, CTABLE
CALLING SEQUENCE:	CALL UCALNK (NDX, CNDX)
TECHNIQUES:	

PROGRAM NAME:	UCDLNK (SUBROUTINE)
PURPOSE:	Delete element from CLINK chain
FUNCTION:	Remove the element NDX from the CLINK chain which begins at CLINK (CNDX). Error trap if CLINK (CNDX) = 0.
EXTERNAL REFERENCES:	INSERTS: TCOM.DEF, CTAB.DCL
	ROUTINES: L, S, UEHALT, UGDLNK
COMMON:	TCOM, CTABLE
CALLING SEQUENCE:	CALL UCDLNK (NDX, CNDX)
TECHNIQUES:	

PROGRAM NAME:	UGALNK (SUBROUTINE)
PURPOSE:	Add link to chain with non-zero head.
FUNCTION:	Add VAL to the chain whose (non-zero) head is HD and whose field definition is FLD. Error trap if HD is zero.
EXTERNAL REFERENCES:	INSERTS: none
	ROUTINES: UEHALT, UGLEND, S
COMMON:	None
CALLING SEQUENCE:	CALL UGALNK (HD, FLD, VAL)
TECHNIQUES:	

PROGRAM NAME:	UGAPND (SUBROUTINE)
PURPOSE:	Add element to chain and set head.
FUNCTION:	Add VAL to the chain whose head is HD and whose field definition is FLD. If HD.EQ.0, HD ← VAL.
EXTERNAL REFERENCES:	INSERTS: None
	ROUTINES: UGLEND, S
COMMON:	None
CALLING SEQUENCE:	CALL UGAPND (HD, FLD, VAL)
TECHNIQUES:	

PROGRAM NAME:	UGDLNK (SUBROUTINE)
PURPOSE:	Delete link from chain.
FUNCTION:	Remove VAL from the chain whose head is HD and whose field definition is FLD. Error trap if HD.EQ.VAL.
EXTERNAL REFERENCES:	INSERTS: None
	ROUTINES: L, S, UEHALT
COMMON:	None
CALLING SEQUENCE:	CALL UGDLNK (HD, FLD, VAL)
TECHNIQUES:	

PROGRAM NAME: UGLEND (INTEGER FUNCTION)

PURPOSE: Locate end of chain.

FUNCTION: Set UGLEND to the index of the last element of the chain whose head is HD and whose field definition is FLD.

EXTERNAL REFERENCES: INSERTS: None

ROUTINES: L

COMMON: None

CALLING SEQUENCE: UGLEND (HD, FLD)

TECHNIQUES:

PROGRAM NAME: UGNLNK (INTEGER FUNCTION)

PURPOSE: Count links in chain.

FUNCTION: Set UGNLNK to the number of links in the chain whose head is HD and whose field definition is FLD.

EXTERNAL REFERENCES: INSERTS: None

ROUTINES: L

COMMON: None

CALLING SEQUENCE: UGNLNK (HD, FLD)

TECHNIQUES:

PROGRAM NAME:	UKALNK (SUBROUTINE)
PURPOSE:	Add element to KLINK chain
FUNCTION:	Add the CTAB element at CNDX to the KLINK chain beginning at KLINK0 (KNDX)
EXTERNAL REFERENCES:	INSERTS: TCOM.DEF, CTAB.DCL, KTAB.DEF
	ROUTINES: L, S, UGALNK
COMMON:	TCOM, CTABLE, KTABLE
CALLING SEQUENCE:	CALL UKALNK (KNDX, CNDX)
TECHNIQUES:	

PROGRAM NAME:	UKDLNK (SUBROUTINE)
PURPOSE:	Delete element from KLINK chain.
FUNCTION:	Delete the CTAB element at CNDX from the KLINK chain beginning at KLINK0 (KNDX). Error trap if chain empty.
EXTERNAL REFERENCES:	INSERTS: TCOM.DEF, CTAB.DCL, KTAB.DEF
	ROUTINES: L, S, UEHALT, UGDLNK
COMMON:	TCOM, CTABLE, KTABLE
CALLING SEQUENCE:	CALL UKDLNK (KNDX, CNDX)
TECHNIQUES:	

PROGRAM NAME:	ULALNK (SUBROUTINE)
PURPOSE:	Add element to LLINK chain.
FUNCTION:	Add the CTAB element at CNDX to the LLINK chain which begins at LLINK0 (LNDX).
EXTERNAL REFERENCES:	INSERTS: TCOM.DEF, CTAB.DCL, LTAB.DEF
	ROUTINES: L, S, UGALNK
COMMON:	TCOM, CTABLE, LTABLE
CALLING SEQUENCE:	CALL ULALNK (LNDX, CNDX)
TECHNIQUES:	

PROGRAM NAME:	ULDLNK (SUBROUTINE)
PURPOSE:	Delete element from LLINK chain.
FUNCTION:	Remove the CTAB element at CNDX from the LLINK chain beginning at LLINK0 (LNDX). Error trap if chain empty.
EXTERNAL REFERENCES:	INSERTS: TCOM.DEF, CTAB.DCL, LTAB.DEF
	ROUTINES: L, S, UEHALT, UCDLNK
COMMON:	TCOM, CTABLE, LTABLE
CALLING SEQUENCE:	CALL ULDLNK (LNDX, CNDX)
TECHNIQUES:	

PROGRAM NAME:	USALNK (SUBROUTINE)
PURPOSE:	Add element to SLINK chain
FUNCTION:	Add the CTAB element at CNDX to the SLINK chain which begins at SLINK0 (SNDX).
EXTERNAL REFERENCES:	INSERTS: TCOM.DEF, STAB.DCL, CTAB.DCL
	ROUTINES: L, S, UGALNK
COMMON:	TCOM, CTABLE, STABLE
CALLING SEQUENCE:	CALL USALNK (SNDX, CNDX)
TECHNIQUES:	

PROGRAM NAME:	USCHAN (SUBROUTINE)
PURPOSE:	Chain all STAB elements of like SKIND
FUNCTION:	Chain all STAB elements for which SKIND.EQ. KIND using HD as the chain head and field definition FLD for the link field.
EXTERNAL REFERENCES:	INSERTS: TCOM.DEF, STAB.DCL
	ROUTINES: L, S
COMMON:	TCOM, STABLE
CALLING SEQUENCE:	CALL USCHAN (KIND, HD, FLD)
TECHNIQUES:	

PROGRAM NAME:	USCLNK (SUBROUTINE)
PURPOSE:	Copy SLINK chain to work area
FUNCTION:	Copy the SLINK chain for the STAB element at SNDX into WTAB. Set WNDX to the index of the copy element in WTAB.
EXTERNAL REFERENCES:	INSERTS: TCOM.DEF, STAB.DCL, CTAB.DCL, WTAB.DEF
	ROUTINES: L, S, UGNLNK, UGAPND
COMMON:	TCOM, STABLE, CTABLE, WTABLE
CALLING SEQUENCE:	CALL USCLNK (SNDX, WNDX)
TECHNIQUES:	

PROGRAM NAME:	USDLNK (SUBROUTINE)
PURPOSE:	Delete element from SLINK chain.
FUNCTION:	Remove the CTAB element at CNDX from the SLINK chain which begins at SLINK0 (SNDX). Error trap if chain empty.
EXTERNAL REFERENCES:	INSERTS: TCOM.DEF, CTAB.DCL, STAB.DCL
	ROUTINES: L, S, UEHALT, UGDLNK
COMMON:	TCOM, CTABLE, STABLE
CALLING SEQUENCE:	CALL USDLNK (SNDX, CNDX)
TECHNIQUES:	

PROGRAM NAME:	UWDLNK (SUBROUTINE)
PURPOSE:	Delete link copy from work area.
FUNCTION:	CNDX is an ARRAY EXOP whose CLINK field appears in a SLINK chain copy in WTAB. Locate the copy element via the cross-reference pointer in STAB (SLINK), then locate the link copy and delete. Error trap on improper argument or copy not found.
EXTERNAL REFERENCES:	INSERTS: TCOM.DEF, CTAB.DEF, STAB.DCL, WTAB.DEF
	ROUTINES: L, S, UEHALT
COMMON	TCOM, CTABLE, STABLE, STABLE
CALLING SEQUENCE:	CALL UWDLNK (CNDX)
TECHNIQUES:	

PROGRAM NAME:	UWUCPY (SUBROUTINE)
PURPOSE:	Update link copy in work area.
FUNCTION:	WNDX is the index of a link chain copy element in WTAB in which CNDX1 should appear. Replace CNDX1 with CNDX2. Error trap if improper argument or copy not found.
EXTERNAL REFERENCES:	INSERTS: TCOM.DEF, WTAB.DEF
	ROUTINES: L, S, UEHALT
COMMON:	TCOM, WTABLE
CALLING SEQUENCE:	CALL UWUCPY (WNDX, CNDX1, CNDX2)
TECHNIQUES:	

REF counter routines

The following routines manipulate the CREF, KREF and LREF fields in CTABLE, KTABLE and LTABLE.

PROGRAM NAME:	UCDREF (SUBROUTINE)
PURPOSE:	Decrement CREF and flag if zero.
FUNCTION:	Decrement the CREF field of the CTAB element at CNDX. ZRO ←.TRUE. if CREF becomes zero, else ZRO←.FALSE.
EXTERNAL REFERENCES:	INSERTS: TCOM.DEF, CTAB.DCL
	ROUTINES: L, S
COMMON:	TCOM, CTABLE
CALLING SEQUENCE:	CALL UCDREF (CNDX, ZRO)
TECHNIQUES:	

PROGRAM NAME:	UCIREF (SUBROUTINE)
PURPOSE:	Increment CREF
FUNCTION:	Add one to the CREF field of the CTAB element at CNDX.
EXTERNAL REFERENCES:	INSERTS: TCOM.DEF, CTAB.DCL
	ROUTINES: L, S
COMMON:	TCOM, CTABLE
CALLING SEQUENCE:	CALL UCIREF(CNDX)
TECHNIQUES:	

PROGRAM NAME:	UKDREF (SUBROUTINE)
PURPOSE:	Decrement KREF
FUNCTION:	Decrement the KREF field of the KTAB element at KNDX. ZRO ← .TRUE. if KREF goes to zero, else ZRO ← .FALSE.
EXTERNAL REFERENCES:	INSERTS: TCOM.DEF, KTAB.DEF
	ROUTINES: L, S
COMMON:	TCOM, KTABLE
CALLING SEQUENCE:	CALL UKDREF (KNDX, ZRO)
TECHNIQUES:	

PROGRAM NAME:	UKIREF (SUBROUTINE)
PURPOSE:	Increment KREF
FUNCTION:	Add one to the KREF field of the KTAB element at KNDX.
EXTERNAL REFERENCES:	INSERTS: TCOM.DEF, KTAB.DEF
	ROUTINES: L, S
COMMON:	TCOM, KTABLE
CALLING SEQUENCE:	CALL UKIREF (KNDX)
TECHNIQUES:	

PROGRAM NAME:	ULDREF (SUBROUTINE)
PURPOSE:	Decrement LREF
FUNCTION:	Decrement the LREF field of the LTAB element at LNDX. ZRO ←.TRUE. if LREF goes to zero, else ZRO ←.FALSE.
EXTERNAL REFERENCES:	INSERTS: TCOM.DEF, LTAB.DEF
	ROUTINES: L, S
COMMON:	TCOM, LTABLE
CALLING SEQUENCE:	CALL ULDREF (LNDX, ZRO)
TECHNIQUES:	

PROGRAM NAME:	ULIREF (SUBROUTINE)
PURPOSE:	Increment LREF
FUNCTION:	Add one to the LREF field of the LTAB element at LNDX.
EXTERNAL REFERENCES:	INSERTS: TCOM.DEF, LTAB.DEF
	ROUTINES: L, S
COMMON.	TCOM, LTABLE
CALLING SEQUENCE:	CALL ULIREF (LNDX)
TECHNIQUES:	

IVTRAN statement routines

The following statements allow the programmer to manipulate the program in units of IVTRAN statements. An IVTRAN statement is defined as a sequence of STATOPS which may contain a SEQNO STATOP, a LABEL STATOP, and exactly one executable STATOP (COPR.GT.LABEL), unless COPR.EQ.LIF, in which case there are two executable STATOPS. When MX is initialized, it creates a back chain of STATOPS in the CBAKOP loop field which makes program manipulation far more efficient.

PROGRAM NAME: UCAPST (SUBROUTINE)

PURPOSE: Add IVTRAN statement preceding given statement.

FUNCTION: Add the IVTRAN statement at CNDX1 to the program immediately preceding the statement at CNDX2, by modifying the CNOPR and CBAKOP chains. Error trap on improper arguments.

EXTERNAL REFERENCES: INSERTS: TCOM.DEF, CTAB.DCL

ROUTINES: L, UCLSTT, UCCSTA, UEHALT

COMMON: TCOM, CTABLE

CALLING SEQUENCE: CALL UCAPST (CNDX1, CNDX2)

TECHNIQUES:

PROGRAM NAME:	UCASTA (SUBROUTINE)
PURPOSE:	Add IVTRAN statement following given statement.
FUNCTION:	Add the IVTRAN statement at CNDX1 to the program immediately following the statement at CNDX2 by modifying the CNOPR and CBAKOP chains. Error trap on improper arguments.
EXTERNAL REFERENCES:	INSERTS: TCOM.DEF, CTAB.DCL
	ROUTINES: L, UEHALT, UCLSTT, UCLNST, UCCSTA
COMMON:	TCOM, CTABLE
CALLING SEQUENCE:	CALL UCASTA (CNDX1, CNDX2)
TECHNIQUES:	

PROGRAM NAME: UCCSTA (SUBROUTINE)

PURPOSE: Chain STATOP's

FUNCTION: CNOPR (CNDX1) ← CNDX2, CBAKOP(CNDX2) ← CNDX1

EXTERNAL REFERENCES: INSERTS: TCOM.DEF, CTAB.DCL

ROUTINES: S

COMMON: TCOM, CTABLE

CALLING SEQUENCE: CALL UCCSTA (CNDX1, CNDX2)

TECHNIQUES:

PROGRAM NAME:	UCLNST (INTEGER FUNCTION)
PURPOSE:	Locate next IVTRAN statement
FUNCTION:	Locate the first STATOP in the IVTRAN statement following the STATOP at CNDX. Error trap if improper argument.
EXTERNAL REFERENCES:	INSERTS: TCOM.DEF, CTAB.DEF
	ROUTINES: L, UEHALT
COMMON:	TCOM, CTABLE
CALLING SEQUENCE:	UCLNST (CNDX)
TECHNIQUES:	

PROGRAM NAME:	UCLPST (INTEGER FUNCTION)
PURPOSE:	Locate previous IVTRAN statement
FUNCTION:	Locate the STATOP which begins the IVTRAN statement preceding the statement containing the STATOP at CNDX. Error trap on improper argument.
EXTERNAL REFERENCES:	INSERTS: TCOM.DEF, CTAB.DCL
	ROUTINES: L, UEHALT, UCLTST
COMMON:	
CALLING SEQUENCE:	UCLPST (CNDX)
TECHNIQUES:	

PROGRAM NAME:	UCLSTO (INTEGER FUNCTION)
PURPOSE:	Locate executable STATOP of IVTRAN statement.
FUNCTION:	Locate the first executable (COPR.GT.LABEL) STATOP in the IVTRAN statement containing the element at CNDX.
EXTERNAL REFERENCES:	INSERTS: TCOM.DEF, CTAB.DEF
	ROUTINES: L, UCLTST
COMMON:	TCOM, CTABLE
CALLING SEQUENCE:	UCLSTO (CNDX)
TECHNIQUES:	

PROGRAM NAME:	UCLSTT (INTEGER FUNCTION)
PURPOSE:	Locate tail STATOP of IVTRAN statement.
FUNCTION:	Locate the last STATOP in the IVTRAN statement containing the element at CNDX.
EXTERNAL REFERENCES:	INSERTS: TCOM.DEF, CTAB.DEF
	ROUTINES: L
COMMON:	TCOM, CTABLE
CALLING SEQUENCE:	UCLSTT (CNDX)
TECHNIQUES:	

PROGRAM NAME:	UCLTST (INTEGER FUNCTION)
PURPOSE:	Locate head of IVTRAN statement
FUNCTION:	Locate the STATOP which begins the IVTRAN statement containing the element at CNDX. Error trap on bad program structure.
EXTERNAL REFERENCES:	INSERTS: TCOM.DEF, CTAB.DEF, NOYES.EQU
	ROUTINES: L, UEHALT, UCLCNH
COMMON:	TCOM, CTABLE
CALLING SEQUENCE:	UCLTST
TECHNIQUES:	

PROGRAM NAME:	UCRSTO (INTEGER FUNCTION)
PURPOSE:	Return value of executable STATOP.
FUNCTION:	UCRSTO is set to the value of the COPR field of the first executable STATOP of the IVTRAN statement containing CNDX.
EXTERNAL REFERENCES:	INSERTS: TCOM.DEF, CTAB.DCL
	ROUTINES: L, UCLSTO
COMMON:	TCOM, CTABLE
CALLING SEQUENCE:	UCRSTO (CNDX)
TECHNIQUES:	

PROGRAM NAME:	UCUSTE (SUBROUTINE)
PURPOSE:	Unlink STATOP from chains.
FUNCTION:	Remove the STATOP at CNDX from the CNOPR and CBAKOP chains, and update the chains. Error trap on improper argument.
EXTERNAL REFERENCES:	INSERTS: TCOM.DEF, CTAB.DCL
	ROUTINES: L, S, UEHALT
COMM	TCOM, CTABLE
CALLING SEQUENCE:	CALL UCUSTE (CNDX)
TECHNIQUES:	

Routines for creating table elements and structure, including routines for locating KTAB elements.

The following routines create individual elements in CTABLE, KTABLE, LTABLE, STABLE and ETABLE, and also create structures in CTABLE. All the basic routines use a single general routine to acquire table elements.

Definitions:

A skeleton in the CTABLE is a STATOP or EXOP with a CN chain of OPRAND's. The COPR value of the STATOP/EXOP is set. None of the OPRAND's are bound to table elements.

Binding an OPRAND to an element in CTABLE, STABLE, KTABLE, or LTABLE involves setting the CLINE, CTABLE and CTYPE fields of the OPRAND, adding to the table elements LINK chain, and incrementing the element's REF counter if defined. Copying and unbinding OPRAND's involve operations on the same fields.

PROGRAM NAME: UCBOPR (SUBROUTINE)

PURPOSE: Bind OPRAND to table element.

FUNCTION: Binds the OPRAND at CNDX to the element in table TABNO at index value NDX. CLINE←NDX, CTABL←TABNO, and the REF counters and LINK chains for each table entry are updated. CTYPE←UCRTYP(NDX). Tables recognized are CTAB, STAB, KTAB and LTAB. Error trap on improper arguments.

EXTERNAL REFERENCES: INSERTS: TCOM.DEF, CTAB.DCL, STAB.DCL, KTAB.DEF, LTAB.DEF, TABL.EQU

ROUTINES: L, S, UEHALT, UCIREF, UCALNK, USALNK, UKIREF, UKALNK, ULIREF, ULALNK

COMMON: TCOM, CTABLE, STABLE, KTABLE, LTABLE

CALLING SEQUENCE: CALL UCBOPR (CNDX, TABNO, NDX)

TECHNIQUES:

PROGRAM NAME:	UCMCPY (SUBROUTINE)
PURPOSE:	Make copy of OPRAND
FUNCTION:	Copy all the fields from the OPRAND at OPNDX1 into the OPRAND at OPNDX2 except CN, CLINK and CLARG. Update the CLINK field and other table LINK fields appropriately. Error trap on improper arguments.
EXTERNAL REFERENCES:	INSERTS: TCOM.DEF, CTAB.DCL, TABL.EQU
	ROUTINES: L, S, UEHALT, UCALNK, UCIREF, USALNK, UKALNK, UKIREF, ULALNK, ULIREF
COMMON:	TCOM, CTABLE
CALLING SEQUENCE:	CALL UCMCPY (OPNDX1, OPNDX2)
TECHNIQUES:	

PROGRAM NAME:	UCMEXE (SUBROUTINE)
PURPOSE:	Make EXOP
FUNCTION:	Create a CTAB element with CKIND ← EXOP and COPR ← OP. Set CNDX to the index of the new element.
EXTERNAL REFERENCES:	INSERTS: TCOM.DEF, CTAB.DCL
	ROUTINES: UGTELM, S
COMMON:	TCOM, CTABLE
CALLING SEQUENCE:	CALL UCMEXE (OP, CNDX)
TECHNIQUES:	

PROGRAM NAME:	UCMLBL (SUBROUTINE)
PURPOSE:	Create LABEL STATOP and associated LTAB element.
FUNCTION:	Create a LABEL STATOP with one OPRAND. Generate a new element in LTAB using LGEN, and bind the OPRAND to it. Set CNDX to the index of the LABEL STATOP, and set LNDX to the index of the new LTAB element.
EXTERNAL REFERENCES:	INSERTS: TCOM.DEF, CTAB.DEF, LTAB.DEF, TABL.EQU
	ROUTINES: L, S, ULMELM, UCMSKS, UCBOPR
COMMON:	TCOM, CTABLE, LTABLE
CALLING SEQUENCE:	CALL UCMLBL (CNDX, LNDX)
TECHNIQUES:	

PROGRAM NAME:	UCMOPR (SUBROUTINE)
PURPOSE:	Make OPRAND element.
FUNCTION:	Create a new CTAB element with CKIND ← OPRAND. Set CNDX to the index of the new element.
EXTERNAL REFERENCES:	INSERTS: TCOM.DEF, CTAB.DCL
	ROUTINES: S, UGTELM
COMMON:	TCOM, CTABLE
CALLING SEQUENCE:	CALL UCMOPR (CNDX)
TECHNIQUES:	

PROGRAM NAME:	UCMSKE (SUBROUTINE)
PURPOSE:	Make EXOP skeleton.
FUNCTION:	Make an EXOP element with COPR ← OP, then make up a CN chain with NARG OPRANDS. Set CNDX to the index of the EXOP element.
EXTERNAL REFERENCES:	INSERTS: TCOM.DEF, CTAB.DEF
	ROUTINES: UEHALT, UCMEXE, UCMSKO
COMMON:	TCOM, CTABLE
CALLING SEQUENCE:	CALL UCMSKE (OP, NARG, CNDX)
TECHNIQUES:	

PROGRAM NAME: UCMSKO (SUBROUTINE)

PURPOSE: Make skeleton CN chain.

FUNCTION: Make NARG OPRANDS and link them into a CN chain. CN (CNDX) ←index of first OPRAND in chain CN (Last OPRAND) ←CNDX, and CLARG (last OPRAND) ←YES.

EXTERNAL REFERENCES: INSERTS: TCOM.DEF, CTAB.DCL

ROUTINES: UCMOPR, S

COMMON: TCOM, CTABLE

CALLING SEQUENCE: CALL UCMSKO (CNDX, NARG)

TECHNIQUES:

PROGRAM NAME: UCMSKS (SUBROUTINE)

PURPOSE: Make STATOP skeleton.

FUNCTION: Make a STATOP element with COPR ← OP, then make a CN chain with NARG OPRANDS and link it to the STATOP. Set CNDX to the index of the STATOP element.

EXTERNAL REFERENCES: INSERTS: TCOM.DEF, CTAB.DEF

ROUTINES: S, UEHALT, UCMSTE, UCMSKO

COMMON: TCOM, CTABLE

CALLING SEQUENCE: CALL UCMSKS (OP, NARG, CNDX)

TECHNIQUES:

PROGRAM NAME:	UCMSTE (SUBROUTINE)
PURPOSE:	Make a STATOP element.
FUNCTION:	Create a CTAB element with CKIND ←STATOP and COPR ←OP. Set CNDX to the index of the new element.
EXTERNAL REFERENCES:	INSERTS: TCOM.DEF, CTAB.DCL
	ROUTINES: S, UGTELM
COMMON:	TCOM, CTABLE
CALLING SEQUENCE:	CALL UCMSTE (OP, CNDX)
TECHNIQUES:	

PROGRAM NAME:	UCRTYP (INTEGER FUNCTION)
PURPOSE:	Return type of structure.
FUNCTION:	UCRTYP is set to the value of the first valid CTYPE field encountered in the program structure beginning at CNDX.
EXTERNAL REFERENCES:	INSERTS: TCOM.DEF, CTAB.DCL
	ROUTINES: L
COMMON:	TCOM, CTABLE
CALLING SEQUENCE:	UCRTYP (CNDX)
TECHNIQUES:	

PROGRAM NAME: UGTELM (SUBROUTINE)

PURPOSE: Get table element.

FUNCTION: Acquire a new element in table TABNO of size SIZE and set NDX to its index. Take the new element from a free list if possible. Extend table if necessary. Element is set to all zeroes.

EXTERNAL REFERENCES: INSERTS: TCOM.DEF, CTAB.DCL, STAB.DCL, KTAB.DEF, LTAB.DEF, ETAB.DEF, VTAB.DEF, WTAB.DEF, PTAB.DEF, TABL.EQU

ROUTINES: L, UGZELM, EXTEND

COMMON: TCOM, CTABLE, STABLE, KTABLE, LTABLE, ETABLE, VTABLE, WTABLE, PTABLE

CALLING SEQUENCE: CALL UGTELM (TABNO, SIZE, NDX)

TECHNIQUES:

PROGRAM NAME:	UKLINT (INTEGER FUNCTION)
PURPOSE:	Locate INT constant.
FUNCTION:	If a KTAB element exists with KTYPE.EQ.INT and KVAL.EQ.VAL, set UKLINT to its index, else UKLINT ← 0.
EXTERNAL REFERENCES:	INSERTS: TCOM.DEF, KTAB.DEF, TYPE.EQU
	ROUTINES: L
COMMON:	TCOM, KTAB
CALLING SEQUENCE:	UKLINT (VAL)
TECHNIQUES:	

PROGRAM NAME:	UKLISC (INTEGER FUNCTION)
PURPOSE:	Locate iterated set constant.
FUNCTION:	If a KTAB element exist with KTYPE.EQ.SET, whose initial value and increment are one, and whose extent is XTNT, then UKLISC is set to its index value, else UKLISC ← 0.
EXTERNAL REFERENCES:	INSERTS: TCOM.DEF, KTAB.DEF, ETAB.DEF, TYPE.EQU, NOYES.EQU
	ROUTINES: L
COMMON:	TCOM, KTABLE, ETABLE
CALLING SEQUENCE:	UKLISC (XTNT)
TECHNIQUES:	

PROGRAM NAME:	UKMINT (SUBROUTINE)
PURPOSE:	Make INT constant.
FUNCTION:	If a KTAB element exists with KTYPE.EQ.INT and KVAL.EQ.VAL, set KNDX to its index, else create a KTAB element with KTYPE ←INT and KVAL ←VAL and set KNDX to its index.
EXTERNAL REFERENCES:	INSERTS: TCOM.DEF, KTAB.DEF, TYPE.EQU
	ROUTINES: S, UKLINT, UGTELM, UGAPND
COMMON:	TCOM, KTABLE
CALLING SEQUENCE:	CALL UKMINT (VAL,KNDX)
TECHNIQUES:	

PROGRAM NAME:	UKMISC (SUBROUTINE)
PURPOSE:	Make iterated set constant.
FUNCTION:	If UKLISC fails, create a KTAB element with KTYPE ← SET, initial value and increment one, final value ←XTNT, and an associated ETAB element with extent ←XTNT. Set KNDX to the new KTAB element index.
EXTERNAL REFERENCES:	INSERTS: TCOM.DEF, KTAB.DEF, ETAB.DEF, TABL.EQU, TYPE.EQU, NOYES.EQU
	ROUTINES: S, UKLISC, UGTELM, UGAPND
COMMON:	TCOM, KTABLE, ETABLE
CALLING SEQUENCE:	CALL USMISC (XTNT, KNDX)
TECHNIQUES:	

PROGRAM NAME:	ULMELM (SUBOUTINE)
PURPOSE:	Make LTAB element.
FUNCTION:	Create a new LTAB element: LVAL $\leftarrow$ next value of LGEN, LISDF $\leftarrow$ YES.
EXTERNAL REFERENCES:	INSERTS: TCOM.DEF, LTAB.DEF, NOYES.EQU
	ROUTINES: S, UGTELM
COMMON:	TCOM, LTABLE
CALLING SEQUENCE:	CALL ULMELM (LNDX)
TECHNIQUES:	

PROGRAM NAME:	USGENM (SUBROUTINE)
PURPOSE:	Generate variable name.
FUNCTION:	Generates the next available 6-character SIXBIT name from the value stored in SGEN.
EXTERNAL REFERENCES:	INSERTS: TCOM.DEF, STAB.DEF, CGLOB.DEF, XCNAM.EQU
	ROUTINES: None
COMMON:	TCOM, STABLE, CGLOB
CALLING SEQUENCE:	CALL USGENM (SN)
TECHNIQUES:	

PROGRAM NAME:	USMSCA (SUBROUTINE)
PURPOSE:	Make scalar element.
FUNCTION:	Create an STAB element with SNAME set to the next available generated name, SKIND ←SCALAR, STYPE ←TYP. Set SNDX to the index of the element.
EXTERNAL REFERENCES:	INSERTS: TCOM.DEF, STAB.DEF
	ROUTINES: S, UGTELM, USGENM, UGAPND
COMMON:	TCOM, STABLE
CALLING SEQUENCE:	CALL USMSCA (TYP, SNDX)
TECHNIQUES:	

Routines for deleting CTABLE elements and structures

The following routines are used to delete elements and structure in CTABLE. The basic routine is UCDOPR, which may involve extensive deletion of structure through following CLINE pointers.

PROGRAM NAME: UCDEXE (SUBROUTINE)

PURPOSE: Decrement and delete EXOP.

FUNCTION: Decrement the CREF counter of the EXOP at CNDX, and if the count goes to zero, delete the EXOP and its OPRAND chain. Error trap on improper argument.

EXTERNAL REFERENCES: INSERTS: TCOM.DEF, CTAB.DCL

ROUTINES: L, UEHALT, UCDREF, UCDOCN, UGFELM

COMMON: TCOM, CTABLE

CALLING SEQUENCE: CALL UCDEXE (CNDX)

TECHNIQUES:

PROGRAM NAME:	UCDOCN (SUBROUTINE)
PURPOSE:	Delete OPRAND chain.
FUNCTION:	Delete all the OPRANDS in the CN chain beginning at CNDX with successive calls to UCDOPR. Error trap on improper argument.
EXTERNAL REFERENCES:	INSERTS: TCOM.DEF, CTAB.DCL, NOYES.EQU
	ROUTINES: L, UEHALT, UCDOPR
COMMON:	TCOM, CTABLE
CALLING SEQUENCE:	CALL UCDOCN (CNDX)
TECHNIQUES:	

PROGRAM NAME:	UCDOCN (SUBROUTINE)
PURPOSE:	Delete OPRAND chain.
FUNCTION:	Delete all the OPRANDS in the CN chain beginning at CNDX with successive calls to UCDOPR. Error trap on improper argument.
EXTERNAL REFERENCES:	INSERTS: TCOM.DEF, CTAB.DCL, NOYES.EQU
	ROUTINES: L, UEHALT, UCDOPR
COMMON:	TCOM, CTABLE
CALLING SEQUENCE:	CALL UCDOCN (CNDX)
TECHNIQUES:	

PROGRAM NAME:	UCDOPR (SUBROUTINE)
PURPOSE:	Delete OPRAND and its substructure.
FUNCTION:	Delete the OPRAND at CNDX. Update the REF counter and LINK chain for the table element CNDX points to. If CNDX points down to more structure in the CTAB, decrement all REF counts and continue deletion if zero. Error trap on improper argument or program structure error.
EXTERNAL REFERENCES:	INSERTS: TCOM.DEF, CTAB.DCL, STAB.DCL, KTAB.DEF, LTAB.DEF, TABL.EQU, NOYES.EQU
	ROUTINES: L, UEHALT, UGZSTK, UCDLNK, UCDREF, UGPUSH, UGPOP, UGELFM, USDLNK, UKDREF, UKDLNK, ULDREF, ULDLNK
COMMON:	TCOM, CTABLE, STABLE, KTABLE, LTABLE
CALLING SEQUENCE:	CALL UCDOPR (CNDX)
TECHNIQUES:	A stack is used to trace the program structure.

PROGRAM NAME: UCDSTA (SUBROUTINE)

PURPOSE: Delete IVTRAN statement.

FUNCTION: Delete the IVTRAN statement containing CNDX from the program, by successive calls to UCDSTE on all the STATOP's in the statement.

EXTERNAL REFERENCES: INSERTS: TCOM.DEF, CTAB.DCL

ROUTINES: UCLTST, UCLNST, L, UCUSTE

COMMON: TCOM, CTABLE

CALLING SEQUENCE: CALL UCDSTA (CNDX)

TECHNIQUES:

PROGRAM NAME:	UCDSTE (SUBROUTINE)
PURPOSE:	Delete STATOP
FUNCTION:	Delete the STATOP at CNDX and its CN chain from the program. Update the CNOPR and CBAKOP chains. Error call if improper argument.
EXTERNAL REFERENCES:	INSERTS: TCOM.DEF, CTAB.DCL ROUTINES: L, UEHALT, UCUSTE, UCDOCN, UGFELM
COMMON:	TCOM, CTABLE
CALLING SEQUENCE:	CALL UCDSTE (CNDX)
TECHNIQUES:	

PROGRAM NAME:	UGFELM (SUBROUTINE)
PURPOSE:	Place element on free list
FUNCTION:	Zero out the element in table TABNO at index NDX, and place it on the table's free list. Error trap if table has no free list.
EXTERNAL REFERENCES:	INSERTS: TCOM.DEF, CTAB.DEF, STAB.DEF, TABL.EQU
	ROUTINES: S, UEHALT, UGZELM
COMMON:	TCOM, CTABLE, STABLE
CALLING SEQUENCE:	CALL UGFELM (TABNO, NDX)
TECHNIQUES:	

STACK routines

The following routines implement stacks and the usual PUSH and POP operations on stacks. Stacks are used for tracing program structure.

PROGRAM NAME:	UGPOP (SUBROUTINE)
PURPOSE:	Pop value from stack.
FUNCTION:	Set VAL to the next value in STACK and reset the stack pointer. NONE $\leftarrow$ .TRUE. if no values remain, else NONE $\leftarrow$ .FALSE. Error trap if stack underflow.
EXTERNAL REFERENCES:	INSERTS: None
	ROUTINES: UEHALT
COMMON:	None
CALLING SEQUENCE:	CALL UGPOP (STACK, VAL, NONE)
TECHNIQUES:	

PROGRAM NAME:	UGPUSH (SUBROUTINE)
PURPOSE:	Push value onto stack.
FUNCTION:	Reset the stack pointer in STACK and push VAL onto it. Error trap if stack overflow.
EXTERNAL REFERENCES:	INSERTS: None
	ROUTINES: UEHALT
COMMON:	None
CALLING SEQUENCE:	CALL UGPUSH (STACK, VAL)
TECHNIQUES:	

PROGRAM NAME: UGZSTK (SUBROUTINE)

PURPOSE: Zero stack.

FUNCTION: Zero out STACK and its stack pointer.

EXTERNAL REFERENCES: INSERTS: None

ROUTINES: None

COMMON: None

CALLING SEQUENCE: CALL UGZSTK (STACK)

TECHNIQUES:

Algebraic routines

The following routines allow limited symbolic algebraic manipulation of arithmetic expressions in the CTABLE. They form the necessary basis for some phases of Macro Expansion and most of Optimization. This package is approximately 50 per cent complete as of this writing.

Definitions:

A standard PLUS is a single PLUS EXOP with one or more OPRANDS, each pointing to one term of the plus. Negative terms are indicated by unary minus (MINUS EXOP with one OPRAND). This form is more convenient for sub-expression analysis than the free form containing both PLUS and MINUS as binary EXOP's.

An analytic structure is an algebraic construct which is recognized as a candidate for combination with other analytic structures. Currently, analytic structures are defined to be INT constants, INT scalars, TIMES of an INT constant and an INT scalar, or unary minus of any of the above.

PROGRAM NAME: UAAASS (SUBROUTINE)

PURPOSE: Add two occurrences of scalar and/or analytic TIMES.

FUNCTION: CNDX1 and CNDX2 must point to OPRANDS with SCALAR contents, or to analytic TIMES, or unary minus of these. The SCALAR's must be identical. Perform symbolic addition of the two elements, and set RES to the result. Error trap on improper argument.

EXTERNAL REFERENCES: INSERTS: TCOM.DEF, CTAB.DEF, TABL.EQU

ROUTINES: UALSAS, UAISOE, UEHALT, UAISUM, UAKRVL, UAAISS, UALIAT, UCMSKI, S, L, UKMINT, UCBOPR, UCMCPY, UAMUMN, UCMOPR

COMMON: TCOM, CTABLE

CALLING SEQUENCE: CALL UAAASS (CNDX1, CNDX2, RES)

TECHNIQUES:

PROGRAM NAME:	UAAII (SUBROUTINE)
PURPOSE:	Add two integer constants
FUNCTION:	INDX1 and INDX2 must be OPRANDS pointing to INT elements in KTAB, or unary minus of same. Add the values of the two constants, create a KTAB element with that value, bind an OPRAND to it, and set RES to the OPRAND. Error trap on improper arguments.
EXTERNAL REFERENCES:	INSERTS: TCOM.DEF, TABL.EQU
	ROUTINES: UAISIK, UEHALT, UALOUM, UAKRVL, UKMINT, UCMOPR, UCBOPR
COMMON:	TCOM
CALLING SEQUENCE:	CALL UAAII (INDX1, INDX2, RES)
TECHNIQUES:	Uses PDP-10 arithmetic.

PROGRAM NAME:	UAAISS (SUBROUTINE)
PURPOSE:	Add two occurrences of a scalar.
FUNCTION:	SNDX1 and SNDX2 must be OPRANDS pointing to the same INT SCALAR in STAB, or unary minus of same. Add the two symbolically, and point RES to the result. Error trap on improper argument.
EXTERNAL REFERENCES:	INSERTS: TCOM.DEF, CTAB.DEF, TABL.EQU, TYPE.EQU ROUTINES: UAISIS, UEHALT, UALOUM, UAISOE, UCMSKE, L, S, UKMINT, UCBOPR, UCMCPY, UAMUMN
COMMON:	TCOM, CTABLE
CALLING SEQUENCE:	CALL UAAISS (SNDX1, SNDX2, RES)
TECHNIQUES:	

PROGRAM NAME: UAATPS (SUBROUTINE)

PURPOSE: Append term to standard PLUS.

FUNCTION: PNDX must point to a standard PLUS, or be zero. TNDX may be any expression except non-standard PLUS or MINUS. If PNDX is zero, a new standard PLUS macro is formed, and PNDX is set to it. TNDX is then added to the PLUS. If TNDX is an analytic structure, UAATPS attempts to combine it with like occurrences already present in PNDX.

EXTERNAL REFERENCES: INSERTS: TCOM.DEF, CTAB.DEF, TABL.EQU

ROUTINES:

COMMON: TCOM, CTABLE

CALLING SEQUENCE: CALL UAATPS (PNDX, TNDX)

TECHNIQUES:

PROGRAM NAME:	UAISAS (LOGICAL FUNCTION)
PURPOSE:	Is this an analytic structure?
FUNCTION:	If CNDX is an analytic strucutre -- i.e., an OPRAND pointing to an integer scalar or constant, or TIMES of an integer scalar and a constant, or unary minus of any these -- then UAISAS ←.TRUE., else UAISAS ←.FALSE.
EXTERNAL REFERENCES:	INSERTS: None ROUTINES: UAISPR, UAISIK, UAISIS, UAISAT
COMMON:	None
CALLING SEQUENCE:	UAISAS (CNDX)
TECHNIQUES:	

PROGRAM NAME:	UAISAT (LOGICAL FUNCTION)
PURPOSE:	Is this an analytic TIMES?
FUNCTION:	If CNDX points to an analytic TIMES -- TIMES of an integer scalar and an integer constant -- or unary minus of this -- then UAISAT ←.TRUE., else UAISAT ←.FALSE. Error trap on bad program structure.
EXTERNAL REFERENCES:	INSERTS: TCOM.DEF, CTAB.DEF, TABL.EQU ROUTINES: UAISPR, UALOUM, L, UEHALT, UCNOCN, UAISIK, UAISIS
COMMON:	TCOM, CTABLE
CALLING SEQUENCE:	UAISAT (CNDX)
TECHNIQUES:	

PROGRAM NAME:	UAISIK (LOGICAL FUNCTION)
PURPOSE:	Is this an integer constant?
FUNCTION:	If CNDX is an OPRAND pointing to an INT element in KTAB, or unary minus of one, then UAISIK ←.TRUE., else UAISIK ←.FALSE. Error trap if CNDX not OPRAND.
EXTERNAL REFERENCES:	INSERTS: TCOM.DEF, CTAB.DCL, KTAB.DEF, TABL.EQU, TYPE.EQU
	ROUTINES: L, UEHALT, UALOUM
COMMON:	CTABLE, KTABLE, TCOM
CALLING SEQUENCE:	UAISIK (CNDX)
TECHNIQUES:	

PROGRAM NAME: UAISIS (LOGICAL FUNCTION)

PURPOSE: Is this an integer scalar?

FUNCTION: If CNDX is an OPRAND pointing to an INT SCALAR element in STAB, or unary minus of one, then UAISIS ← .TRUE., else UAISIS ← .FALSE. Error trap if CNDX is not OPRAND.

EXTERNAL REFERENCES: INSERTS: TCOM.DEF, CTAB.DCL, STAB.DEF, TABL.EQU, TYPE.EQU

ROUTINES: UEHALT, L, UALOUM

COMMON: TCOM, CTABLE, STABLE

CALLING SEQUENCE: UAISIS (CNDX)

TECHNIQUES:

PROGRAM NAME:	UAISOE (LOGICAL FUNCTION)
PURPOSE:	Are these OPRAND'S equal?
FUNCTION:	If CNDX1 and CNDX2 are OPRANDS whose CTABL and CLINE fields are equal, UAISOE ←.TRUE., else UAISOE ←.FALSE. Error trap if CNDX1 and CNDX2 are not OPRANDS.
EXTERNAL REFERENCES:	INSERTS: TCOM.DEF, CTAB.DCL
	ROUTINES: L, UEHALT
COMMON:	TCOM, CTABLE
CALLING SEQUENCE:	UAISCE (CNDX1, CNDX2)
TECHNIQUES:	

PROGRAM NAME:	UAISPM (LOGICAL FUNCTION)
PURPOSE:	Is this old PLUS or MINUS?
FUNCTION:	If CNDX is a PLUS EXOP with CTABL $\neq$ STPLUS, or binary MINUS, or an OPRAND pointing to either, UAISPM ← .TRUE., else UAISPM ← .FALSE.
EXTERNAL REFERENCES:	INSERTS: TCOM.DEF, CTAB.DEF, TABL.EQU
	ROUTINES: UAISPR, L, UAISUM
COMMON:	TCOM, CTABLE
CALLING SEQUENCE:	UAISPM (CNDX)
TECHNIQUES:	

PROGRAM NAME:	UAISPR (LOGICAL FUNCTION)
PURPOSE:	Is this parenthesized?
FUNCTION:	If CNDX is an EXOP with CPAR = YES, or an OPRAND pointing to one, then UAISPR = .TRUE., else UAISPR = .FALSE.
EXTERNAL REFERENCES:	INSERTS: TCOM.DEF, CTAB.DCL, NOYES.EQU, TABL.EQU
	ROUTINES: L
COMMON:	TCOM, CTABLE
CALLING SEQUENCE:	UAISPR (CNDX)
TECHNIQUES:	

PROGRAM NAME:	UAISSP (LOGICAL FUNCTION)
PURPOSE:	Is this standard PLUS?
FUNCTION:	If CNDX is a PLUS EXOP with CTABL = STPLUS, or an OPRAND pointing to one, then UAISSP ←.TRUE., else UAISSP ←.FALSE.
EXTERNAL REFERENCES:	INSERTS: TCOM.DEF, CTAB.DEF, TABL.EQU
	ROUTINES: L
COMMON:	TCOM, CTABLE
CALLING SEQUENCE:	UAISSP (CNDX)
TECHNIQUES:	

PROGRAM NAME:	UAISUM (LOGICAL FUNCTION)
PURPOSE:	Is this a unary minus?
FUNCTION:	If CNDX is a MINUS EXOP with one OPRAND, or an OPRAND pointing down to one, UAISUM ←.TRUE., else UAISUM ←.FALSE.
EXTERNAL REFERENCES:	INSERTS: TCOM.DEF, CTAB.DEF, NOYES.EQU, TABL.EQU
	ROUTINES: L
COMMON:	TCOM, CTABLE
CALLING SEQUENCE:	UAISUM (CNDX)
TECHNIQUES:	

PROGRAM NAME:	UAKRVL (INTEGER VALUE)
PURPOSE:	Return value of integer constant.
FUNCTION:	CNDX must be an OPRAND pointing to an element in KTAB. UAKRVL ←contents of KVAL field. Error trap on improper argument.
EXTERNAL REFERENCES:	INSERTS: TCOM.DEF, CTAB.DCL, KTAB.DEF, TABL.EQU
	ROUTINES: L, UEHALT
COMMON:	TCOM, CTABLE, KTABLE
CALLING SEQUENCE:	UAKRVL (CNDX)
TECHNIQUES:	

PROGRAM NAME:	UALIAT (INTEGER FUNCTION)
PURPOSE:	Locate integer term of TIMES.
FUNCTION:	CNDX must point to an analytic TIMES. UALIAT is set to the index value of the OPRAND pointing to the integer element of the TIMES.
EXTERNAL REFERENCES:	INSERTS: TCOM.DEF, CTAB.DEF, TABL.EQU
	ROUTINES: L, UCNOCN, UAISIK
COMMON:	TCOM, CTABLE
CALLING SEQUENCE:	UALIAT (CNDX)
TECHNIQUES:	

PROGRAM NAME:	UALOUM (INTEGER FUNCTION)
PURPOSE:	Locate operand of unary minus.
FUNCTION:	If CNDX is a unary MINUS or an OPRAND pointing to one, set UALOUM to the index value of its OPRAND, else set UALOUM to zero.
EXTERNAL REFERENCES:	INSERTS: TCOM.DEF, CTAB.DCL, TABL.EQU
	ROUTINES: L, UAISUM
COMMON:	TCOM, CTABLE
CALLING SEQUENCE:	UALOUM (CNDX)
TECHNIQUES:	

PROGRAM NAME:	UALSAS (INTEGER FUNCTION)
PURPOSE:	Locate scalar in analytic structure.
FUNCTION:	If CNDX is an analytic structure other than an integer constant, set UALSAS to the index value of the OPRAND pointing to tis STAB SCALAR, else set UALSAS to ZERO.
EXTERNAL REFERENCES:	INSERTS: None
	ROUTINES: UAISIK, UAISIS, UAISAT, UALSAT, UALOUM
COMMON:	TCOM, CTABLE
CALLING SEQUENCE:	UALSAS (CNDX)
TECHNIQUES:	

PROGRAM NAME:	UALSAT (INTEGER FUNCTION)
PURPOSE:	Locate scalar in analytic TIMES.
FUNCTION:	CNDX be an analytic TIMES. UALSAT is set to the index value of the OPRAND pointing to the STAB SCALAR element. Error trap if improper argument.
EXTERNAL REFERENCES:	INSERTS: TCOM.DEF, CTAB.DEF, TABL.EQU
	ROUTINES: L, UEHALT, UAISIS, UAISIK, UALOUM
COMMON:	TCOM, CTABLE
CALLING SEQUENCE:	UALSAT (CNDX)
TECHNIQUES:	

PROGRAM NAME:	UAMUMN (SUBROUTINE)
PURPOSE:	Form unary minus.
FUNCTION:	CNDX may be any expression. A unary MINUS macro is formed, its OPRAND is bound to CNDX, and RES is pointed to the MINUS EXOP. If CNDX is a unary minus, then RES is pointed to its OPRAND. Error trap on improper argument.
EXTERNAL REFERENCES:	INSERTS: TCOM.DEF, CTAB.DEF, TABL.EQU
	ROUTINES: UCRTYP, UAISUM, UCMSKE, L, S, UEHALT, UCMCPY, UCBOPR, UCMOPR, UCMCPY
COMMON:	TCOM, CTABLE
CALLING SEQUENCE:	CALL UAMUMN (CNDX, RES)
TECHNIQUES:	

PROGRAM NAME:	UATPLS (SUBROUTINE)
PURPOSE:	Transform PLUS or MINUS.
FUNCTION:	CNDX must be non-standard PLUS or binary MINUS. A standard n-ary PLUS is formed from CNDX, and PNDX is pointed to it. Error trap on improper arguments.
EXTERNAL REFERENCES:	INSERTS: TCOM.DEF, CTAB.DEF, NOYES.EQU
	ROUTINES: UAISPM, UEHALT, UGZSTK, L, UGPUSH, UCNOCN, UCLTCN, UCLPOP, UAATPS, UAMUMN, UGPOP
COMMON:	TCOM, CTABLE
CALLING SEQUENCE:	CALL UATPLS (CNDX, PNDX)
TECHNIQUES:	Tree-walk with right-most elements being stacked first.

APPENDIX I

ALLOCATION TABLES

Array Table Entry Format

```
     '      '      '      '     '     '     '     '     '    '
=======================================================================
!    A!     B!    ARADIM!            !                        ARANXT!
-----------------------------------------------------------------------
!                             ARASZE!                         ARAORG!
-----------------------------------------------------------------------
!    C1!    C2!   C3!   C4!   C5!   C6!   C7!                        !
-----------------------------------------------------------------------
!           ARAEXT(1)!       ARAEXT(2)!       ARAEXT(3)!   ARAEXT(4)!
-----------------------------------------------------------------------
!           ARAEXT(5)!       ARAEXT(6)!       ARAEXT(7)!             !
=======================================================================
```

A -- ARAUSE
B -- ARAOVL
CI -- ARAPRM(I)

Hole Table Entry Format

```
     '      '      '      '     '     '     '     '     '
=======================================================================
!    A!                       AHLBLK!                         AHLNXT!
-----------------------------------------------------------------------
!                             AHLDIM!                         AHLSZE!
-----------------------------------------------------------------------
!    B1!    B2!   B3!   B4!   B5!   B6!   B7!                        !
-----------------------------------------------------------------------
!           AHLEXT(1)!       AHLEXT(2)!       AHLEXT(3)!   AHLEXT(4)!
-----------------------------------------------------------------------
!           AHLEXT(5)!       AHLEXT(6)!       AHLEXT(7)!             !
-----------------------------------------------------------------------
!                 AHLCRD(1)!           AHLCRD(2)!          AHLCRD(3)!
-----------------------------------------------------------------------
!                 AHLCRD(4)!           AHLCRD(5)!          AHLCRD(6)!
-----------------------------------------------------------------------
!                 AHLCRD(7)!                                         !
=======================================================================
```

A -- AHLUSE
BI -- AHLPRM(I)

Allocation Table Entry Format

```
     ,     ,     ,     ,     ,     ,     ,     ,     ,     ,     ,
=====================================================================
!     A!                           !                          ALANXT!
---------------------------------------------------------------------
!     B1!   B2!   B3!   B4!   B5!   B6!   B7!                       !
---------------------------------------------------------------------
!        ALAEXT(1)!       ALAEXT(2)!      ALAEXT(3)!       ALAEXT(4)!
---------------------------------------------------------------------
!        ALAEXT(5)!       ALAEXT(6)!      ALAEXT(7)!                !
---------------------------------------------------------------------
!              ALACRD(1)!            ALACRD(2)!            ALACRD(3)!
---------------------------------------------------------------------
!              ALACRD(4)!            ALACRD(5)!            ALACRD(6)!
---------------------------------------------------------------------
!              ALACRD(7)!                                           !
=====================================================================
```

A -- ALAUSE

BI -- ALAPRM(I)

Block Table Entry Format

```
     ,       ,       ,     ,      ,      ,     ,     ,       ,
=====================================================================
!     A!     ABLDIM!              ABLNAR!                     ABLNXT!
---------------------------------------------------------------------
!                                 ABLRMN!                     ABLFST!
---------------------------------------------------------------------
!     B1!   B2!     B3!   B4!   B5!     B6!    B7!                  !
---------------------------------------------------------------------
!           ABLEXT(1)!        ABLEXT(2)!     ABLEXT(3)!    ABLEXT(4)!
---------------------------------------------------------------------
!           ABLEXT(5)!        ABLEXT(6)!     ABLEXT(7)!
=====================================================================
```

A -- ABLUSE

BI -- ABLPRM(I)

TBLPNT Entry Format

```
      '     '     '     '     '     '     '     '     '     '     '
=======================================================================
!                                                             TBLSTR !
-----------------------------------------------------------------------
!                                                              NFLDS !
-----------------------------------------------------------------------
!                                                            NMFLD1 !
!                                                            NMFLD2 !
-----------------------------------------------------------------------
!                                                             BYTSTR !
-----------------------------------------------------------------------
!                                                             TBLSZE !
=======================================================================
```

TBLSTR	--	the start of table
NFLDS	--	number of fields contained in an entry in table 1
NMFLD1	--	first five ASCII characters of table name 1
NMFLD2	--	second five ASCII characters of table name
BYTSZE	--	start of byte pointers for table 1
TBLSZE	--	size of table 1