DATAPOINT

DASL™

User's Guide

50807

This manual has been prepared according to the standards of the Information Mapping Method. Information Mapping is a Registered Trademark of Information Mapping, Inc., of Lexington, MA, USA.

Document No. 50807. 4/84.

Preface

The DASL USER's GUIDE is the first document to be produced in the new DATAPOINT documentation standard format. It is written using the Information Mapping writing method. It was produced in draft form on the 9660 Laser Printer and in final form via the ATD (Automated Technical Documentation) system.

Please forward your comments on this document to:

DATEC Publications
DATAPOINT Corporation
9725 Datapoint Dr. MS T-72
San Antonio, Texas 78284

CONTENTS

CHAPTER 1.
INTRODUCTION TO THE DASL REFERENCE SECTION

Contents

OVERVIEW

Introduction

This section is a reference guide on the use of Datapoint's Advanced Systems Programming Language (DASL). It is designed to be used by programmers as they design, code, and debug DASL programs. It can also be used in conjunction with training to reinforce and review information.

Description of DASL

DASL is a high level language that provides the powerful general constructs required for successful structured programming. DASL also has many of the 'low level' capabilities of an assembler language.

Continued on next page

Features of DASL

The features of DASL include the following:

- simplicity
- ease of program
  - analysis
  - maintenance
  - modification
- variety of data types and operators
- a simple macro facility
- fast compilation
- efficient execution
- independence from a specific machine or operating system.

ORGANIZATION OF THIS SECTION

Rationale for organization of this section

This section is organized to provide programmers with reference information that

- is comprehensive enough to address all the components of DASL,
- is in categories that are most useful to a programmer,
- anticipates the types of questions that might prompt the programmer to turn to the guide,
- includes many specific examples illustrating concepts, and
- is easy to retrieve.

Information is presented so that there is a logical progression from the beginning to the end of this section. However, each chapter can be read and understood independently of other chapters.

Information is organized by major components of DASL. A programmer can quickly find the answer to a specific question about using one component of the language.

Continued on next page

Coming up

The following table describes the chapters of this section of the guide.

THIS chapter...	DESCRIBES...
INTRODUCTION	the features of DASL and the organization of the DASL section of the guide.
PROGRAM STRUCTURE	the structure of a DASL program, and introduces the major components of DASL.
STRUCTURED PROGRAMMING	guidelines for using the structured programming approach with DASL.
LEXICAL RULES	DASL vocabulary rules.
DATA TYPES AND DECLARATIONS	the declaration and uses of data types.
FUNCTIONS	the declaration and uses of functions.
EXPRESSIONS	the meaning and use of expressions.
OPERATORS	the syntax, semantic rules, and use of DASL operators.

Continued on next page

Coming Up, (continued)

THIS chapter...	DESCRIBES...
STATEMENTS	the syntax, semantic rules, and use of statements, including flow of control constructs.
MACROS	the rules for, and uses of, macros, including predefined and recursive macros.

I/O independent

Because DASL is designed to be machine and operating system independent, the language itself does *not* contain any input/output (I/O) facilities. Therefore, this section does not refer to any specific I/O package or operating system. Rather, it describes the syntax and structure of the language.

CHAPTER 2.
STRUCTURE OF A DASL PROGRAM

Contents

OVERVIEW OF DASL PROGRAM STRUCTURE

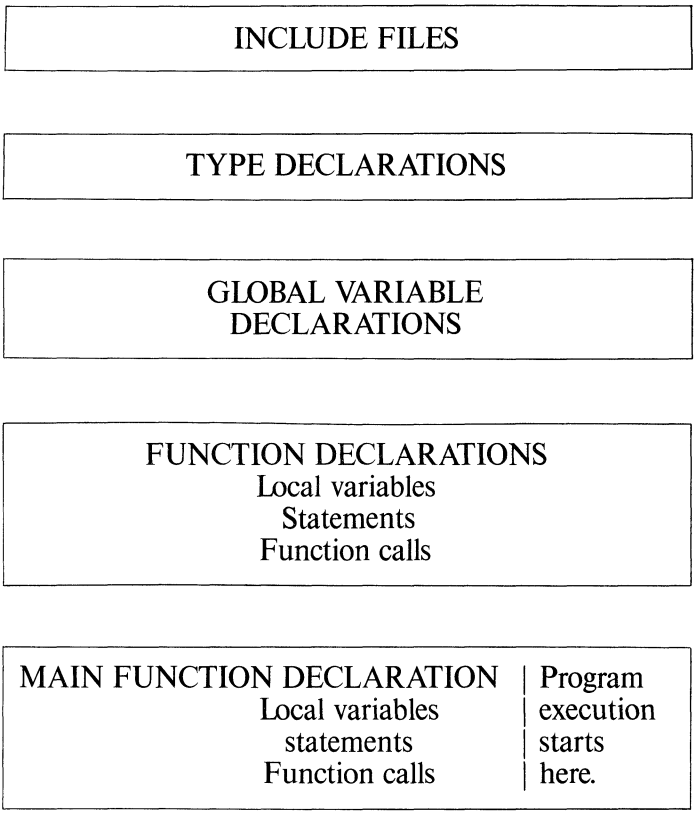
Introduction

A DASL program is made up of declarations of functions and data variables. The diagram below gives you a 'picture' of a DASL program.

Continued on next page

Diagram

The following diagram outlines the main parts of a DASL program. The parts do not necessarily occur in this order.



Continued on next page

Description of INCLUDE files

INCLUDE is a compile time directive to temporarily switch the current input file to the one indicated.

Description of type declaration

Every data variable in DASL must be declared to be a specific type. The data type determines the amount of storage to allocate and how to interpret the data item.

The data types in DASL include scalar, pointer, array, structure, and function. DASL includes a facility called TYPDEF which allows the programmer to create new names for data types.

Description of local and global variables

Data variables may be either local or global.

A local variable is declared within a function, and its scope is limited to that particular function.

A global variable is declared outside of any function. Its scope lasts from the point at which it is declared to the end of the module.

Continued on next page

Definition of a DASL function

A *function* in DASL is an independent unit which consists of local variables and statements.

A function performs one or more actions, usually related to one specific task.

You can reference a function with a statement from within any other function which is declared subsequently.

Functions allow for modular programming because they can be coded, debugged, or re-implemented independently of each other.

Statements, expressions, and operators

Statements specify the actions which a function must take. Statements are composed of expressions and flow of control constructs, such as LOOP WHILE and IF...THEN...ELSE.

One or more expressions are often combined with an operator, which indicates what is to be done with the expression(s). DASL provides a large variety of operators.

Continued on next page

Description of MAIN function

Every DASL program has one function named MAIN, which is where program execution starts.

The MAIN function usually serves to define the overall structure of the program and consists of calls to other functions and I/O routines.

A SAMPLE PROGRAM ILLUSTRATING PROGRAM STRUCTURE

Introduction

On the following page, a short DASL program is presented to illustrate the basic structure of the language.

Following the sample program are:

- a brief description of the major components of the program,
- a description of some of the statements, operators and expressions used in the program, and
- an explanation of what the program does.

Purpose of sample program

The purpose of the sample program is to provide a general introduction to DASL. You may want to refer back to the sample after you have studied specific details of the language.

Continued on next page

A SAMPLE PROGRAM ILLUSTRATING PROGRAM STRUCTURE, Continued

Example

```
/* This program computes the average value
in an array of numbers, determines the
largest value in the array, and prints the
results. */                                          /* 1 */

INCLUDE (D$INC)
INCLUDE (D$RMS)                                     /* 2 */

DEFINE (sizeArray, 10)                              /* 3 */

list [sizeArray] BYTE := {                          /* 4 */
   10, 9, 7, 1, 2, 6, 5, 4, 2, 9
   };

calculateAverage (ptrList ^ BYTE,                   /* 5 */
                  sizeList BYTE) UNSIGNED :=
VAR    sum UNSIGNED;
       n   BYTE;
{

   IF sizeList = 0 THEN RESULT := 0
    ELSE {
      sum := n := 0;
      LOOP {
         sum += ptrList++^;
       WHILE ++n < sizeList;
         };
      RESULT := sum / sizeList;
      };
   };
```

Continued on next page

Example, (continued)

```
largestValue ( ptrList ^ BYTE,                          /* 6 */
               sizeList  BYTE) BYTE :=
VAR   item, n  BYTE;
{
   RESULT := ptrList^;                                  /* 7 */
   n := 0;
   LOOP {
      item := ptrList++ ^;
      IF item > RESULT THEN RESULT := item;
    WHILE ++n < sizeList;
      };
   };

ENTRY MAIN ( ) :=
VAR   largestItemInList BYTE;                           /* 8 */
      averageOfList UNSIGNED;
{
   averageOfList :=
     calculateAverage (&list[0], SIZEOF list);
   largestItemInList :=
     largestValue (&list[0], SIZEOF list);              /* 9 */

   write ('Average is ', averageOfList);
   write ('Largest value in list is ',
           largestItemInList);
   };
```

Continued on next page

Part of program

The following chart describes the important elements of the example program.

PART	DESCRIPTION
Comment 1	Documents part of program; ignored by compiler
INCLUDE 2	Compile time directive to temporarily switch current input file to one indicated
DEFINE 3	A macro which can be used to define a name for a constant
global variable declaration 4	Declares global variables, which may be initialized
function 5	Independent unit which performs a specific task
function type 6	Describes the formal parameters and return value of a function
statement body 7	Includes one or more statements which indicate actions to be performed

Continued on next page

Part of program (continued)

PART	DESCRIPTION
local variable declaration 8	Declares one or more variables which are local to the function in which they are declared
function call 9	A statement which passes control to another function

Description of statements

Statements are made up of expressions and flow of control constructs, such as LOOP – WHILE.

Statements in DASL are normally executed in sequence, but several kinds of statements alter the flow of control.

Each statement ends with a semi-colon.

Compound statements begin and end with braces.

Continued on next page

Description of operators

DASL provides a variety of operators, only a few of which are illustrated in the sample program.

An understanding of the operators which involve pointers is essential to understanding the sample program.

These operators are described in the following chart.

OPERATOR	USE OF OPERATOR WITH POINTERS
^	Returns the value of the object to which a pointer points
&	Returns the address of an object in memory
++	Increments a pointer by the size of the object to which a pointer points The position of the ++ operator before or after the operand determines whether the operand is incremented before or after the value of the operand is used in the expression.

Continued on next page

Summary of program

The chart below summarizes the actions performed by the sample program.

STAGE	THIS function...	PERFORMS this action...
1	MAIN	calls the function calculateAverage with 2 parameters.
2	calculateAverage	computes the average value of the components of list, and returns the value to MAIN.
3	MAIN	assigns the result of the function calculateAverage to the variable averageOfList.
4	MAIN	calls the function largestValue with two parameters.
5	largestValue	determines the largest value among the components of list, and returns the value to MAIN.

Continued on next page

Summary of program, (continued)

STAGE THIS function...		PERFORMS this action...
6	MAIN	assigns the result of the function largestValue to the variable largestItemList.
7	MAIN	calls on standard I/O routines to print the values of averageOfList and largestItemInList on the screen.

CHAPTER 3.
STRUCTURED PROGRAMMING

Contents

OVERVIEW

Introduction

This chapter highlights the major concepts of structured programming as they relate to DASL.

The remaining chapters of this section provide the information a programmer needs to code a DASL program based on structured programming concepts.

Most programmers have had some exposure to, and experience with, structured programming. This chapter is included to provide

- a summary of the structured programming approach,
- a common framework for DASL programmers,
- guidelines for coding a DASL program that reflect structured programming, and
- criteria for a quality DASL program.

Continued on next page

Definition

Structured programming is an approach to programming that systematically integrates the process of program design, coding, and testing.

Structured programming involves the use of

- top down design
- independent program functions
- structured coding principles.

Benefits

Structured programming offers the following benefits to the DASL programmer.

BENEFIT	EXPLANATION
Easier to develop large programs	• Programming is based on an explicit general design. • Hierarchy of blocks of code allows programmer to solve small problems in order. • Programmer can work from general aspects of problem to specific details.

Continued on next page

Benefits, (continued)

BENEFIT	EXPLANATION
Fewer initial errors	• Time spent designing program minimizes logic errors. • Use of small functions narrows the focus, reducing the factors the programmer must keep track of at one time. • Improved readability allows programmer and others to spot errors before compiling.
Easier to test and debug	• A function with a single entry and single exit generally limits the source of errors to that one function. • Use of stub (dummy functions) allows the program to be tested before it is complete.
Easier to maintain	• Improved readability allows programmer (or others) to go back to a program and follow its logic. • Use of independent functions generally allows one function to be changed with no impact on other functions.

Continued on next page

Coming up

The following pages describe

- principles of structured programming,
- top down design,
- independent functions,
- three basic types of statements,
- guidelines for coding, and
- criteria for a quality DASL program.

PRINCIPLES OF STRUCTURED PROGRAMMING

Introduction

The programmer's objective is to write correct, efficient, and easily modifiable programs. The principles of structured progamming provide a basis for obtaining this objective.

Principles

The following are four basic principles of structured programming.

- The design approach is top down.
- A program is composed of basically independent functions.
- Only three types of statements are necessary for constructing functions.
- Programs should be readable by programmer and anyone involved in the maintenance of the programs.

Each of these principles is discussed on the following pages.

TOP DOWN DESIGN GUIDELINES

Introduction

The design approach in structured programming is top down.

Top down design involves the following steps:

- definition of the problem
- general design of the whole program
- development of a main module which controls flow of processing for the whole program
- continuous refinement from general task to specific subtasks.

Continued on next page

Designing a program

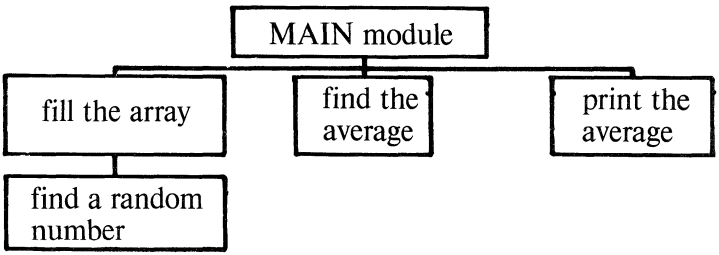
STAGE	DESCRIPTION	GUIDELINES
1	Define the problem	• Describe what you want to do. • Describe input and output.
2	Design general structure of the program	• Use English language statements. • Indicate what you want to accomplish, not how.
3	Refine program in steps	• Break each general task into specific subtasks. • Develop data structure details needed by each subtask. • Check that subtasks, together, achieve task. • Validate each subtask before further refinement.

Continued on next page

Hierarchy diagram

A hierarchy diagram helps in designing a structured program. A diagram graphically indicates the relationships between the tasks and subtasks.

Example: The following is an example of a hierarchy diagram developed for a program that generates several random numbers, finds the average of those numbers, and prints that average.



Use of stubs

Use stubs to test the coding of general tasks or subtasks.

A stub is a dummy function included in a program to allow compilation and execution to take place. Generally, stubs only write a message that you have reached this function, indicating that the link to the function is correct.

Continued on next page

Additional design hints

The following hints help insure a well designed program:

- Have someone else read your programs; a fresh set of eyes can often spot flaws in logic or omissions that you may not notice.

- Don't be afraid to start over; sometimes it is more efficient to start over than it is to keep refining a problematic unit of code.

- Refer to documentation to determine available options.

INDEPENDENT FUNCTIONS

Description

A structured program is made up of a series of small functions. Each function addresses a single task.

Each program function should be self-contained and independent of other functions in the program (to the greatest degree possible). Each function should have a single entry and a single exit.

Advantages of independent functions

A program consisting of independent functions allows a programmer to modify or remove one function without affecting the rest of the program (assuming the input/output specifications for the function remain the same).

Function length guideline

Although structured programming does *not* have rules for the length of a function, most functions should not include more than 50 to 60 lines. This should be enough code to perform one task, which is all a function should do.

Continued on next page

Making functions independent

The following hints can help you make functions independent.

- Avoid unnecessary modifictions of global variables (keep function free of undesirable side effects).
- Declare all temporary variables local to the function in which they occur.
- Avoid changes to input parameters that are passed indirectly whenever possible (if there are not memory space limitations); input parameters should be passed by value.

THREE BASIC TYPES OF STATEMENTS

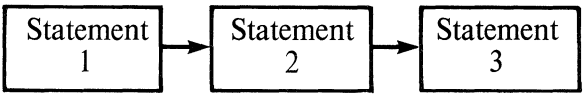
Introduction

Coding a structured program can be done using only three basic types of statements:

- sequential
- conditional
- iterative.

Description of sequential statement

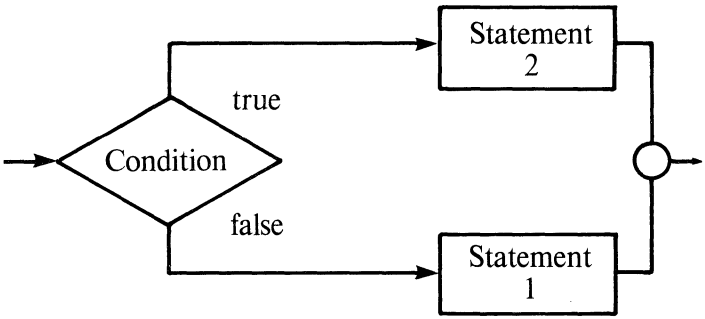
A sequential statement performs an operation and then continues on to the next statement in the program, as the diagram below indicates:



Continued on next page

Description of conditional statement

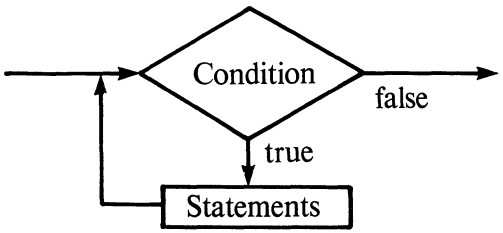
A conditional statement performs a test to decide which statements to execute next. The IF/THEN/ELSE and CASE statements are examples of DASL conditional statements. The diagram on the next page illustrates a conditional statement.



Continued on next page

Description of iterative statement

An iterative statement repetitively executes a statement or number of statements as long as the condition is true. The LOOP WHILE statement is the only iterative statement in DASL. The following diagram illustrates an iterative statement.



Comment: The GOTO statement is not necessary in DASL, although it can be used. (See pages called GOTO Statements.)

GUIDELINES FOR CODING

Principle

The program code should clearly reflect the logic of a structured program. The programmer should never sacrifice program clarity for

- cleverness
- minor reductions in machine execution time.

Coming up

The following pages provide guidelines that promote readable code that clearly reflects the structured program logic.

Guidelines are presented for the following aspects of coding:

- identifiers and constants
- indentation
- multiple statements and multiple lines
- parentheses
- comments.

GUIDELINES FOR CODING IDENTIFIERS AND CONSTANTS

Hints

Good mnemonic identifiers are very important in making a program easy to read and understand.

The following table provides some hints for clear identifiers.

HINT	EXAMPLE
Use description identifier	average := sum/count;
Don't abbreviate unnecessarily; DASL allows 30 characters	Instead of tx1, use a more readable identifier, such as textLine.
Make identifiers made up of more than one word easier to read by using uppercase and lowercase letters. Capitalize the first letter of each word after the first word.	The identifier negativesum should be written as negativeSum.

Continued on next page

Hints, (continued)

HINT	EXAMPLE
Name associated items with the same prefix (or suffix)	The prefix 'scroll' could be used with identifiers associated with scrolling: • scrollUp • scrollDown • scrollStop The suffix 'line' could be used with identifiers associated with lines: • firstLine • maxLine • currentLine
Do *not* choose identifiers that might be confused.	yz and y2 are easily confused, as are positionMax and positionMax2.

Continued on next page

Hints, (continued)

HINT	EXAMPLE

Use relevant function identifiers.	The following excerpt of a MAIN function illustrates that relevant function names permit the reader to understand what the program does:
	
	`readData(list);`
	`validDate(list,ok);`
	`IF ok THEN {`
	`  computeMean(List,mean);`
	`  computeVariance (list,`
	`  variance);`
	`  printResults (mean,`
	`  variance);`
	`  }`
	`ELSE errorHandler;`

Continued on next page

Constants

Name constants instead of using numbers in the code. This provides the following advantages:

- easier reading (the reader doesn't have to figure out the significance of a number in a statement).

- easier maintenance (only the value associated with the constant name needs to be changed; no other code needs to be changed when a constant value).

NOTE: For information on assigning names to constants, see pages called The Define Macro.

GUIDELINES FOR INDENTATION

Introduction

Since DASL is a free-format language, indentation is simply used to promote readablility.

DASL programs should be written as a series of properly nested control constructs. The depth of this nesting is usually called the level of nesting.

Standard

A recommended standard for indentation is to indent 3 spaces for each level. All statements on the same level should be aligned.

Formats for different statements

The recommended indentation formats for the following types of statements are discussed in the section called Statements:

- LOOP WHILE,
- IF..THEN and IF..THEN...ELSE, and
- CASE.

GUIDELINES FOR CODING MULTIPLE STATEMENTS AND MULTIPLE LINES

Guidelines

The following guidelines relate to multiple statements per line and multiple lines per statement.

- Generally, put only one control statement (i.e., LOOP WHILE, IF...THEN, CASE) on one line.

- Two non-control statements may be put on the same line only if they are logically related, or if you want to describe them both with a single comment.

- If a statement is too long to fit on a single line, position the continuation on the next line so that structurally related parts of the statement are aligned, and/or indent at least 2 spaces.

Example:

```
IF (a < 0 | a > 99) &
   (b < 0 | b > 99) &
   (c < 0 | c > 99) THEN getArray ();
```

GUIDELINES FOR CODING PARENTHESES

Purpose

Use parentheses to make clear to the reader the operands associated with each operator. The reader should not have to memorize precedence and syntax rules to interpret code.

Example:

The following statement is ambiguous to the reader unfamiliar with DASL precedence rules:

```
status := lines > page & sum + 1 < totalLines;
```

Parentheses make the statement easier to read:

```
status := (lines > page) & ((sum + 1) <totalLines);
```

NOTE: The expression in the inner parentheses is performed first.

Caution

Although parentheses make the program more readable, using too many of them can make the code cluttered and confusing.

Also, be careful that the use of parentheses is consistent with the processing you want to occur.

GUIDELINES FOR CODING COMMENTS

Purpose

Comments should be included as part of the program code. These comments should help make explicit to the reader the logic of the program.

Comments do not affect the processing of the program.

Rules

Comments start with a /* and end with a */. You can also use a period, plus sign, or asterisk at the beginning of a line to indicate that line is a comment.

Comments may be placed on a separate line, at the end of a statement or expression, or even nested in the middle of a line.

Comment as you code

You should include comments in a program as you write it, while the code is fresh in your mind.

Continued on next page

Guidelines

The following are some guidelines for using comments to improve the readability of a program.

- Don't restate the obvious.

  Example: The following comment is unnecessary:
  read(date); /* input the date */

- Use simple, clear language.

  Example: For the following statement:
  IF text ^ = page^ THEN

  instead of this comment:
  /* check the 2 link fields in the doubly linked list to see if they are pointing to the same node element */

  it is clearer to say:
  /* determine if we have come to the end of the list */

- Comment on the purpose of a section of code, not on how it works.

- Use comments to indicate the beginning and ending of LOOP WHILE statements, if there are many layers of nesting.

- Comment on each item in a declaration.

Continued on next page

Guidelines, (continued)

Examples:

```
DEFINE (high, 999)  /* highest valid character
                       count */
counter INT;        /* total number of times
                       through the loop */
```

- Comment at the beginning of the program, indicating

  - what the program does
  - the method(s) it uses
  - programmer's name and the date the program was written.
  - documents used (if any)
  - date of, and reason for, modification

- Separate function with comments explaining

  - what the function does
  - entry conditions (initial values passed into function)
  - exit conditions (values returned and any resulting changes in the state of the program).

CRITERIA FOR A QUALITY DASL PROGRAM

Introduction

In addition to consistency with the structured programming principles and guidelines for coding already discussed in this chapter, there are several criteria for a quality DASL program.

These additional criteria include

- the program produces meaningful results from any data set
- the program does *not* terminate during execution because of run-time errors
- functions are used properly
- the program is portable
- the program is generalizable.

Continued on next page

Meaningful results from any data set

A quality program should produce meaningful results from any data. The program should *not* terminate because of invalid data.

The program should contain validity tests for the data and error messages. Error messages should be generated when the program encounters invalid data.

These messages should indicate:

- what data is invalid,
- why it is invalid, and
- how the problem can be corrected.

Continued on next page

Avoid run-time errors

The following run-time errors may cause a program to terminate abnormally during execution:

- an improper argument for a function
- an attempt to read past end-of-file
- referencing a data object before it is initialized
- a condition which the CASE statement does not address (use of the DEFAULT statement avoids this possibility).

The following run-time conditions lead to meaningless results:

- an array subscript which is greater than the number of elements in the array
- division by zero
- a loop without a condition to terminate it.

Continued on next page

Functions are used properly

A program uses functions properly if it:

- takes advantage of commonly used functions that are already in a program file
- protects the values of input parameters (if possible, pass values directly rather than indirectly, i.e. using pointers)
- keep temporary variables local to the function in which they are used
- uses global variables minimally
- uses, where appropriate, signal flags to return to the calling function error or abnormal condition indications for a computation.

Program is portable

A portable program is independent from the hardware, operating system, or compiler on which it is run.

To the degree possible, avoid machine dependent constructs. Localize and identify any such constructs.

Continued on next page

Program is generalizable

A generalizable program is *not* dependent on any specific data set. This means that the program requires minimal changes as user needs change.

To promote generality

- use variables instead of constants
- use flexible data formats (e.g., use a named constant for array bounds).

CHAPTER 4.
LEXICAL RULES

Contents

OVERVIEW

Introduction

In this chapter, we discuss the vocabulary of the DASL language and the specific rules which apply to each DASL token.

Classes of tokens

There are six classes of tokens in DASL:

- identifiers,
- keywords,
- numbers,
- strings,
- operators, and
- separators.

Continued on next page

White space and comments between tokens

Blanks, tabs, newlines, and comments may be used freely between tokens. They are ignored by the compiler except as they serve to separate tokens.

Either a blank or newline is required to separate otherwise adjacent identifiers, keywords, and numbers.

Coming up

The following pages describe the lexical rules for

- identifiers and keywords
- numbers and strings
- operators and separators
- comments.

LEXICAL RULES: IDENTIFIERS AND KEYWORDS

Kinds of identifiers

An identifier in DASL can represent a

- label,
- macro,
- variable,
- type, or
- function.

Continued on next page

Lexical rules for identifiers

The following rules apply to all identifiers regardless of what kind of object they represent.

Composition: An identifier can contain any combination of upper or lower-case letters, digits, and the characters $ and _ (underscore). The first character may *not* be a digit.

The compiler distinguishes between upper and lower-case letters. For example, the identifiers number, NUMBER and Number are all different.

Length: An identifier can be any length, as long as it fits on one line. However, the compiler uses only the first 29 characters and the last character to distinguish one identifier from another.

Only the first seven characters and the last character are significant in ENTRY or EXTERN identifiers.

Examples of identifiers

The following are examples of valid DASL identifiers:

```
b
file_Name
$CLOSE
```

Continued on next page

Predefined identifiers

The identifiers on the next page are reserved use as keywords. They may *not* be redefined. These keywords are grouped here according to their use in a program.

USED IN... STATEMENT	DECLARATIONS	OTHER
CASE	ENTRY	FAST
DEFAULT	EXTERN	SIZEOF
ELSE	RECURSIVE	SYSTEM
GOTO	STATIC	
IF	STRUCT	
LOOP	TYPDEF	
THEN	UNION	
WHILE	VAR	

The following identifiers are predefined by the compiler.

TYPES	MACROS
BOOLEAN	DEFINE
BYTE	IFELSE
CHAR	INCLUDE
INT	INCR
LONG	SUBSTR
UNSIGNED	

The identifier RESULT is predefined inside functions.

LEXICAL RULES: NUMBERS AND STRINGS

Lexical rules for numbers

The following chart shows the rules for representing decimal, octal, or hexadecimal numbers.

THIS type of number...	IS a sequence of digits...
decimal	*not* beginning with the digit 0. *Examples*: 56 700
octal	• beginning with the digit 0, and • *not* including the digits 8 or 9. *Examples*: 05 0706
hexadecimal	• combined with the characters A through F (upper or lower-case), and • beginning with the characters 0x or 0X. *Examples*: 0XA5 0xfb6

Continued on next page

Lexical rules for strings

A string is a sequence of characters surrounded by apostrophes. A newline in the middle of a string is ignored.

Example:

'cat'
'This is a string.'
'hello
good-bye'

An apostrophe within a string is represented by two consecutive apostrophes.

Example:

'This is Tom''s book.'

LEXICAL RULES: OPERATORS AND SEPARATORS

Operator symbols

Each of the following characters can be used independently, or in combinations, to represent operators.

! % | & * ( ) − + = [ ] ∿ ^ < > .: ? /

Separator symbols

The following characters are used as separators.

{ } :;,

LEXICAL RULES: COMMENTS

Lexical rules for comments

A comment is a line or other portion of text which is completely ignored by the compiler.

A comment line is any line starting with a period, asterisk, or plus character.

A comment is any text which begins with the characters /* and ends with the characters */.

Comments may be nested.

CHAPTER 5.
DATA TYPES AND DECLARATIONS

Contents

OVERVIEW

Introduction

DASL is a typed language. This means that every variable must be declared as a type before it can be used in a DASL program.

Definition of data type

A *data type* is a construct which is used to define the set of values a variable may assume.

The data type determines

- the amount of storage to allocate for a particular data item, and
- how to interpret the data item when it is used in an expression.

Description of variable declaration

A variable declaration establishes the type of a variable and determines its scope. The variable declaration may also cause storage to be allocated.

All variables must be declared before they can be used in a program.

Continued on next page

Coming up

The chapter describes

- data types
- TYPDEF declaration
- variable declaration
  - local and global variables
  - storage classes
  - initialization
- scalar types
- pointer type
- array type
- structure types.

OVERVIEW OF DATA TYPES

Introduction

The following pages provide a brief description of the data types in DASL.

Each type is discussed in more detail later in the chapter.

Kinds of data types

DASL provides several predefined types, and various methods for creating more complex data structures.

The data types in DASL include

- scalar,
- pointer,
- array,
- structure, and
- function.

Continued on next page

Definition of scalar type

A *scalar type* describes an integer value.

The six predefined scalar types in DASL are:

- BOOLEAN
- CHAR
- BYTE
- UNSIGNED
- INT
- LONG.

NOTE: There is currently no floating point type.

Definition of pointer type

A *pointer type* describes a value which is the address of some object in memory.

Continued on next page

Definition of array type

An *array type* is an aggregate type containing a fixed number of components which are all the same type. Each component of an array can be accessed by an index.

Definition of structure type

A *structure type* is an aggregate type containing a fixed number of components which may be of different types. Each component of a structure can be accessed by name.

There are two kinds of structure types, STRUCT and UNION. The difference between the STRUCT and UNION types is in their memory allocation.

Continued on next page

Definition of function type

A *function type* describes the parameters and results of function call.

A function is considered a type in DASL because

- a pointer can contain the address of a function, and
- a function type can be defined in a TYPEDEF declaration.

Description of TYPDEF

TYPDEF is a facility for creating new data type identifiers. See the pages called TYPDEF Declarations.

Type compatibility

Several operators require their operands to be of compatible type. The rules for type compatibility are presented in the discussions of each specific data type.

Continued on next page

Other derived types

It is possible to simulate other data types by using the DASL macro facility.

For example, the DASL programmer can use macros to declare enumerated types and sets. See the chapter called Macros.

Changing an expression's type

You can cause the type of an expression to be reinterpreted locally by using the cast operator. See the pages called Cast Operator.

TYPDEF DECLARATION

Introduction

A data type in DASL may be either directly described in the variable declaration or referenced by a type identifier.

DASL provides several standard type identifiers for scalar types. In addition, DASL provides a facility called TYPDEF for creating new data type identifiers.

Description of TYPDEF

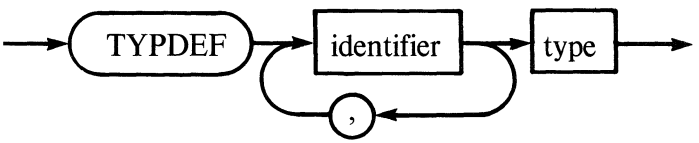
TYPDEF is a facility for creating new data type identifiers.

TYPDEF declarations do *not* reserve storage. Instead, they define identifiers which can be used later as if they were type keywords.

TYPDEF does *not* introduce brand new types, only synonyms for types which could be specified in another way.

Syntax

The syntax for a TYPDEF declaration is:



Continued on next page

Example

In the following example, the TYPDEF facility is used to create the type identifier Matrix.

```
TYPDEF Matrix [50] [50] BOOLEAN;
```

Important: In the declaration above, Matrix has not been declared as a variable. Rather, Matrix has been declared as a type identifier for a particular two-dimensional array.

Naming convention for type identifiers

The naming convention for type identifiers is to capitalize the first letter of the identifier.

Continued on next page

Using TYPDEF in a program

By using TYPDEF declarations a programmer can avoid repetition of lengthy definitions.

Example: In the following example, Date is first declared as a type. Later in the program, variables are declared which are of type Date, or which have elements of type Date.

```
TYPDEF Date STRUCT {
   month, date, year INT;
   reminder [25] CHAR;
   };
   .
   .
   .

appointment, meeting Date; /* declares variables
                              of Date  */

dateArray [50] Date;       /* declares an array
                              of Date */

person STRUCT {            /* declares a
                              structure which
                              includes another
                              structure */
   name [25] CHAR;         /* as one */
   birthdate Date;         /* of the fields */
   };

dateptr ^ Date;            /* declares a pointer
                              to a Date type */
```

Continued on next page

Advantages

One of the advantages of using TYPDEF declarations is that they eliminate repetition of definitions, as illustrated in the previous example.

TYPDEFs also make program modification easier. If a programmer decides to change the definition of a variable, the change only has to be made in one place.

Another advantage in using TYPDEFs is that they add clarity to program documentation. It is easier to read and refer to type definitions which are grouped together rather than scattered and repeated throughout a program.

VARIABLE DECLARATION–OVERVIEW

Description

A variable declaration establishes the type of a variable, and determines its scope. The variable declaration may also cause storage to be allocated. Storage is discussed on the pages called Storage Classes.

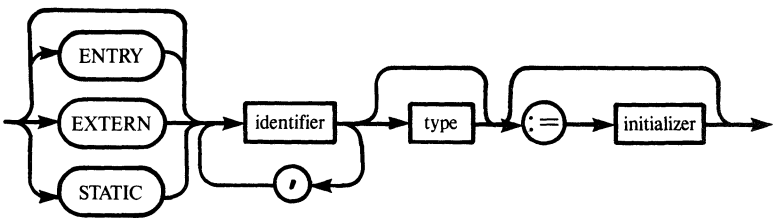
The type of a variable may be any type except a function type.

Rule

You must declare all variables before you can use them in a program.

Syntax

This diagram shows the syntax for declaring a variable.



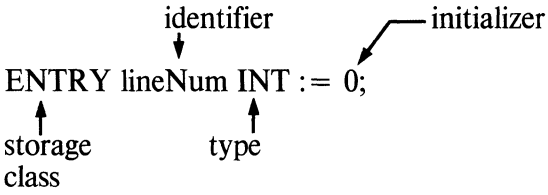
The type is omitted *only* for ENTRY declarations which follow EXTERN declarations in the same block.

Continued on next page

VARIABLE DECLARATION–OVERVIEW, Continued

Example

This is an example of a variable declaration which includes an initializer and the storage class ENTRY.



Coming up

Next, the following aspects of variable declaration are discussed:

- local and global variables
- storage classes
- initialization.

VARIABLE DECLARATION: LOCAL AND GLOBAL VARIABLES

Introduction

In this section, we discuss local and global variables and how they differ in

- declaration,
- scope, and
- storage.

Scope of variables

The scope of a variable refers to the part of the program for which the variable is defined.

Continued on next page

Comparison

The following table compares various aspects of local and global variables.

ASPECT OF COMPARISON:	LOCAL VARIABLES	GLOBAL VARIABLES
Declaration	Local variables are declared at the beginning of a function. The keyword VAR precedes local variable declarations. Function parameters and the RESULT identifier are also local varables, but they are not explicitly declared.	Global variables are declared outside of any function.

Continued on next page

Comparison, (continued)

ASPECT OF COMPARISON:	LOCAL VARIABLES	GLOBAL VARIABLES
Scope	The scope of a local variable is limited to the function in which it is declared. Variables of the same name outside of the function are unrelated.	The scope of a global variable lasts from the point at which it is declared in a module to the end of that module. A global variable can have no effect inside a function which has a local variable of the same name.
Optional storage classes	Local variables can be declared STATIC or EXTERN.	Global variables can be declared EXTERN or ENTRY. Use of the keyword STATIC is not necessary for global variables since they are always static.

Continued on next page

VARIABLE DECLARATION: LOCAL AND GLOBAL VARIABLES, Continued

Comparison, (continued)

ASPECT OF COMPARISON:	LOCAL VARIABLES	GLOBAL VARIABLES
Storage allocation	Storage is allocated for a local variable only when the function is called. The variable disappears when the function is exited. *Exception*: Storage is allocated permanently for the class STATIC.	Storage is allocated for a global variable when it is defined *except* in the case of EXTERN declarations. A global variable remains in existence permanently and retains its value between different function calls.

Continued on next page

Example

The following program fragment shows the declaration of both local and global variables.

```
pWork ^ [256]  BYTE;        /* global variable       */
flags          BOOLEAN;     /* declarations          */
size           BYTE;

func (b INT) INT :=         /* b is a local variable*/
VAR   length, count INT;    /* local variable
                               declarations          */
{
   .
   .
   .
   RESULT := length;        /* RESULT is a local
                               variable              */
   };

ENTRY MAIN ( ) :=
VAR   ch CHAR;              /* local variable        */
      intArray [20] INT;    /* declaration           */
{
   ch := 'A';               /* assignment to local
                               variable              */
   size := 5;               /* assignment to global
                               variable              */
   .
   .
   .
   };
```

VARIABLE DECLARATIONS: STORAGE CLASSES

Introduction

DASL provides three optional storage classes for variables. These classes are:

- STATIC
- ENTRY
- EXTERN.

Description of STATIC storage class

The storage class STATIC causes a local variable to be permanently allocated instead of being allocated dynamically each time a function is called.

A STATIC variable can be initialized. The value of a STATIC variable remains in memory between one function call and the next.

Continued on next page

Declaration of a STATIC local variable

To declare a STATIC variable, you preface the variable declaration with the keyword STATIC.

Example:

```
random ( ) UNSIGNED: =
VAR STATIC seed UNSIGNED :=14723;
{
   RESULT : = seed :=(seed * k1) + k2) + k3;
   };
```

NOTE: Each time the function random is called, the previous result becomes the seed for the next calculation.

Use of ENTRY and EXTERN storage classes

The storage classes ENTRY and EXTERN are commonly used in large programs in order to allow communication between different modules.

Continued on next page

Description of ENTRY storage class

A global variable declared as ENTRY becomes an entry point which may be referenced by other program modules when the modules are linked together.

The ENTRY definition causes new memory space to be allocated for the variable.

An ENTRY definition for a particular variable can occur in only one module in a complete program.

Description of EXTERN storage class

The keyword EXTERN in a variable declaration indicates that storage for the identifiers being declared is allocated somewhere else in the program.

If a variable is declared to be EXTERN, then somewhere among the modules of the complete program there must be a definition for the variable which allocates storage. Often this definition occurs with the ENTRY class in another DASL program module or in an assembly language routine.

Local variables may be declared as EXTERN, but this is not very common.

Continued on next page

ENTRY and EXTERN in the same block

If a variable has been declared EXTERN, such as by an INCLUDE file, it may be redeclared as ENTRY in the same block.

In this case, the ENTRY declaration omits the type specification. In all other cases the type is required.

If is a good idea to put EXTERN declarations in an INCLUDE file so that the type specification appears in only one place.

Example of ENTRY and EXTERN declarations

The ENTRY and EXTERN storage classes are more commonly used with function declarations than with other variable declarations.

Therefore, a complete example of how the ENTRY and EXTERN classes are used in a program is presented in the next chapter on the pages called Function Storage Classes.

VARIABLE DECLARATION: INITIALIZATION

Description

Certain kinds of variables may be given an initial value in their declaration.

Initialization takes place when the program is loaded rather than at run time.

No default initialization

If you do not explicitly initialize variables, or assign values to them in program statements, then the variables have undefined values.

Continued on next page

General rules for variable initialization

There are four general rules which apply to the initialization of variables of any type.

1. You may initialize a variable in its declaration if the variable is

   - STATIC, or
   - global but not EXTERN.

2. If you specify an initializer, then you may give only one identifier in the identifier list.

3. An initializer for a scalar or pointer variable is an expression of compatible type. You may also initialize an array of characters with a string constant of identical size.

4. Initialized expressions must be values which are constants when the program is linked. These expressions may involve

   - numeric and string constants,
   - addresses of STATIC variables, and
   - addresses of arrays subscripted by constants.

Continued on next page

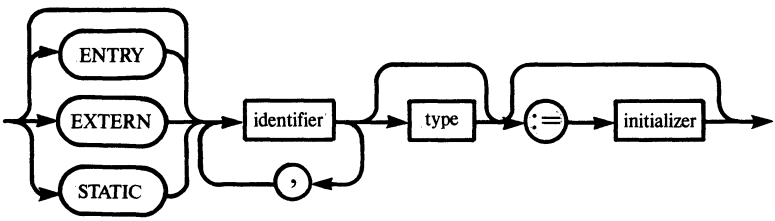
Specific rules for initialization

Specific rules for initialization are given on the pages called

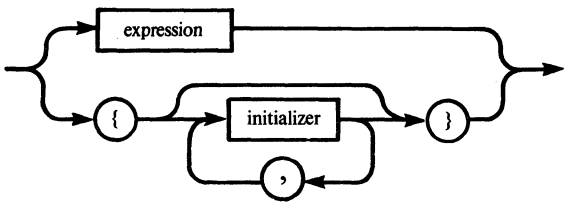
- Scalar Types,
- Pointer Type,
- Array Type, and
- Structure Types.

Syntax for initializer

This first diagram shows where the optional initializer comes at the end of a variable declaration.



The following diagram shows the syntax for initializers which may be nested.



SCALAR TYPES

Description of scalar type

A scalar type describes an integer value. Each scalar has associated with it

- its length in bytes, and
- whether it is signed or unsigned.

Predefined scalar types

The following scalar types are predefined by the compiler.

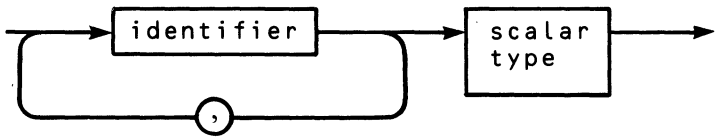
TYPE IDENTIFIER	NUMBER OF BYTES	SIGNED	UNSIGNED
BOOLEAN	1		X
CHAR	1		X
BYTE	1		X
UNSIGNED	2		X
INT	2	X	
LONG	4	X	

NOTE: The only scalar types which may be defined by a TYPDEF declaration are types equivalent to the predefined types.

Continued on next page

Syntax for scalar declaration

To declare a variable as a scalar type, use the following syntax.



Example of scalar variable declaration

The following variable declarations declare two variables of type CHAR and two variables of type UNSIGNED.

```
c1, c2    CHAR;
length    UNSIGNED;
currPos   UNSIGNED; /* line number of current
                        record  */
```

Comment: You can declare variables of the same type together, separating them with a comma.

Sometimes it is more convenient to declare variables of the same type on separate lines. This allows you to add a comment for a specific declaration.

Continued on next page

Variable initialization

You may initialize a global or STATIC scalar variable in the declaration.

See the pages entitled Variable Declaration: Initialization for the general rules for initializing variables.

Example:

```
lineOnBL  UNSIGNED := 0;
currPos   UNSIGNED := 0;
eof       BOOLEAN  := FALSE;
```

Type compatibility for scalars

Any two scalar types are compatible with each other.

Continued on next page

Operations permitted on scalars

The following chart lists the operations which are permitted on scalars.

See the chapter called Operators for a complete description of each operator.

Continued on next page

Operations permitted on scalars, (continued)

OPERATOR(S)	SEE THESE PAGES
+ −	Additive Operators
* | %	Multiplicative Operators
++ −−	Increment/Decrement Operators
= ∿ = <> <= >=	Relational Operators
& |	Logical Operators
? :	Conditional Operators
<< >>	Shift Operators
− ∿∿	Unary Arithmetic Operators
&& || !!	Bit Operators
&	Pointer Operators
SIZEOF	SIZEOF Operator
<type>	Cast Operator
:= op=	Assignment Operators
∿	Unary Not Operator

POINTER TYPE

Description of pointer type

A pointer type describes a value which is either

- the address of some object in memory, or
- the value zero, representing a null pointer.

Definition of null pointer

A *null pointer* is a pointer whose value is zero.

DASL guarantees that no pointer that points to data ever contains zero. Therefore, a null pointer can be used to signal a special event such as the end of a linked list.

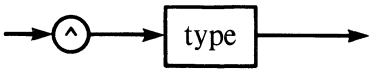
Pointer size

For the 5500 instruction set, all pointers are two bytes long.

Continued on next page

Syntax for pointer type

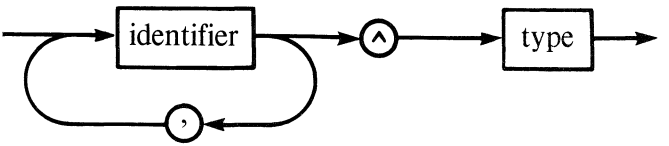
The syntax for a pointer type is:



A pointer may contain the address of an object of any data type. This type must be declared in the pointer declaration following the ^ symbol.

Syntax for pointer declaration

To declare a variable as a pointer, use the following syntax.



Continued on next page

Example of pointer declaration

The following variable declarations declare pointers to several different types.

```
intPtr ^ INT;
charPtr ^ CHAR;
pWork ^ [256] BYTE;
pNode ^ STRUCT {
   name [12] CHAR;
   flags     BYTE;
   };
```

Pointers to arrays or structures contain the address of the first component of the object.

Continued on next page

Forward reference

The identifier for the type which a pointer points to may be a forward reference which is defined later by a TYPDEF declaration.

Example:

```
TYPDEF Node STRUCT {
   name [12] CHAR;
   link ^ Arc;
   };

TYPDEF Arc STRUCT {
   start, end Node;
   };
```

You only have to define a type identifier if the definition is needed by the compiler to determine the meaning of an expression involving the pointer type.

Continued on next page

Variable initialization

You may initialize a global or STATIC pointer variable in the declaration.

See the pages called Variable Declaration: Initialization for the general rules for initializing variables.

Usually the values for pointer initialization are zero (null pointer) or an expression involving addresses of previously defined data of appropriate types.

Example of pointer initialization

This example uses the address operator &. The address operator yields the address of the operand which follows it.

```
intPtr ^ INT := 0;
fileNode [255] BYTE;
pFileNode ^ BYTE := &fileNode[0];
```

The last statement in this example assigns the address of the first component of fileNode to the pointer pFileNode.

Continued on next page

Type compatibility

Pointer types are compatible only if the types pointed to are compatible types of the same size.

Two pointers must be of compatible type if they are operands for the assignment operator, conditional operator, or one of the relational operators.

Continued on next pag

Operations permitted on pointers

The following chart lists the operations which are permitted on pointers or which return a pointer value.

See the chapter entitled Operators for a complete description of each operator.

OPERATOR(S)	USE OF OPERATOR(S) WITH POINTERS	SEE THESE PAGES
+	add a pointer and a scalar	
−	subtract a scalar from a pointer, *or* subtract a pointer from a pointer	Pointer Arithmetic
++ −−	increment (or decrement) a pointer by the size of the object to which the pointer points	
= ∿= < > <= >=	establish an ordered relationship	Relational Operators
& |	establish a logical relationship	Logical Operators
? :	establish a conditional relationship	Conditional Operators

Continued on next page

Operations permitted on pointers, (continued)

OPERATOR(S)	USE OF OPERATOR(S) WITH POINTERS	SEE THESE PAGES
&	return the address of an object	Pointer
^	return the value of the object to which a pointer points	Operators
SIZEOF	return the size of, in bytes	SIZEOF Operators
~	return the logical "not"	Unary Negation Operator
<type>	cast (reinterpret locally) as the Specified type	Cast Operator
:=	assign a value	Simple Assignment Operator
+= -=	add (subtract) a scalar to a pointer, and then assign the result to pointer.	Assignment Operators

ARRAY TYPE

Definition of array

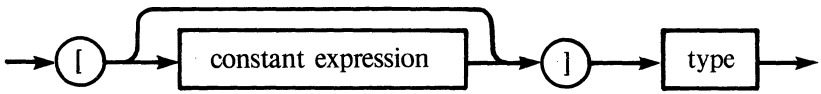
An *array* is an aggregate type containing a number of components which are all the same type. The array components may be any type except function type.

The number of components in an array is given by a compile-time constant.

The compiler allocates sequential memory locations for the array components.

Syntax for array type

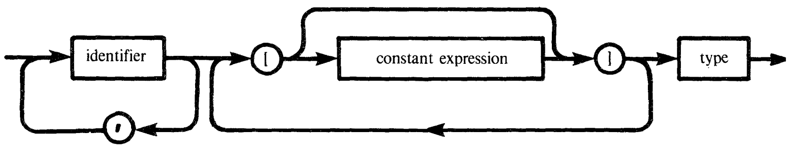
The syntax for the array type is:



Continued on next page

Syntax for array declaration

To declare a variable as an array, use the following syntax.



Example: The following variable declaration declares typefaces to be an array of six characters.

```
                typefaces [6] CHAR ;

identifier          upper bound         type
                    (number of
                    elements)
```

General syntax for initializing arrays

You may initialize an array by listing initializers for components of the array in ascending order inside braces.

Example:

```
date [4] INT:={1, 2, 3, 4};
```

See the pages called Variable Declaration: Initialization for the general rules for initializing variables.

Continued on next page

Formats for initializing arrays

The following 5 formats for initializing arrays represent variations of the general syntax.

FORMAT	EXAMPLE
Bound may be omitted: The first bound of the array type specification may be omitted if the array is initialized. The compiler will determine the bound from the number of initializers specified.	`date [ ] INT := {1,2,3,4};`
String constants as initializers: String constants may be used to initialize part or all of a character array.	`line [ ] CHAR := {'abc', 015};`
Braces may be omitted with string constants: You may omit the braces around a string constant initializer if the length of the string matches the array bound.	`options [3] CHAR := 'abc';`
Braces must be nested: Braces must be nested if a component of an array is an array or a named structure.	`dimension [3][2] INT := {{1,2},{2,4}, {3,6}};`

Continued on next page

Formats for initializing arrays, (continued)

FORMAT	EXAMPLE
Partial initialization: You do not have to initialize all the components when you initialize an array. However, any initializers listed will be assigned to the array starting with the *first* components of the array.	`date [4] INT := {1,2};`
NOTE: The bound may *not* be omitted if you are not initializing all the components of an array.	

No default initialization

If you do *not* assign values to array components, the compiler does *not* automatically assign zeros or blanks.

The array components assume the value of whatever values exist in the memory space reserved for the array.

Continued on next page

Accessing a component of an array

The components of an array are numbered with indices from zero to one less than the number of components specified as the upper bound.

You can access an individual component of an array using the index. Set the index to one less than the position in the array of the component you wish to access.

Example:

```
VAR lineNumber [8] INT:
{
    .
    .
    .
    lineNumber[0] := 5;
    };
```

In the statement, lineNumber[0] := 5; the index 0 refers to the first component of the array, lineNumber. Thus, this statement assigns the value 5 to the first component of the array, lineNumber.

Continued on next page

Index out of bounds

There is no run-time checking for array indices which are out of bounds.

For example, if you declare the variable charArray [15] CHAR, then the expression charArray[20] does *not* necessarily result in an error.

Continued on next pag

Definition of multi-dimensional arrays

A *multi-dimensional array* is an array whose components are arrays.

An array such as page [24] [15] is considered a two-dimensional array. It can be thought of as a matrix or grid. You can have arrays with as many dimensions as you need.

Example:

```
VAR   textLine   [15] CHAR;
      page [24] [15] CHAR;
{
  textLine[0] :='A';
  page[1] [5] :=textLine[0];
  .
  .
  };
```

The expression, page [1] [5], accesses the sixth component of page's second array:

page	[1]	[5]
↑	↑	↑
identifier	index of second element (an array) of the page (array)	index of the sixth element (of the second array)

Thus, the statement, page[1] [5] := textLine[0];, assigns the character 'A' (the value of the first component of the first component of the array, textLine), to the sixth component of page's second array.

Continued on next page

Arrays of other aggregate types

The components of an array can be *any* type except the function type. The following two examples illustrate arrays of the structure types, STRUCT and UNION.

Example 1: This variable declaration declares an array date of structures.

```
date [10] STRUCT {
    month, day, year INT;
    message [20] CHAR;
    };
```

Example 2: This example shows a TYPDEF declaration for a UNION, uval and then a variable declaration for an array of type Uval.

```
TYPDEF Uval UNION {
    ival INT;
    cval CHAR;
    };

list [50] Uval;
```

Continued on next pag

Type compatibility for arrays

Two array types are compatible if they

- have the same upper upper bound, and
- are made up of components which are of compatible types of the same size.

Two arrays must be of compatible type if they are operands for the assignment operator or for one of the relational operators.

You can assign values of components of one array to the components of another array if both arrays are compatible.

Example:

```
VAR a [3] CHAR;
    b [3] BYTE;
{
  a := b;
  .
  .
  .
  };
```

The statement, a := b; assigns the values of components of array b to the components of array a.

Continued on next page

Operations permitted on arrays

This chart lists the operations which are permitted on arrays.

See the chapter entitled Operators for a complete description of each operator.

OPERATOR(S)	DESCRIPTION	SEE THESE PAGES
:=	assign a value	Simple Assignment Operator
= ~= <> <= >=	establish an ordered relationship	Relational Operators
&	Take the address of	Pointer Operators
SIZEOF	return the size of, in bytes	SIZEOF Operators
<type>	Cast (reinterpret locally) as the specified type	Cast Operator
[ ]	select an array component	Subscripting Operator

STRUCTURE TYPES

Definition of structure

A *structure* is a collection of one or more variable components, possibly of different types, grouped together under a single name.

A structure can include components of any type except the function type.

Every component (also known as a field) of a structure can be named and referenced.

Names of fields are unique to each structure.

Continued on next page

Kinds of structure types

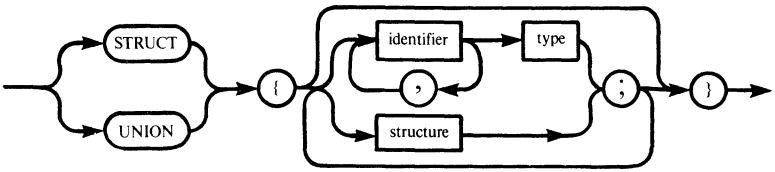
DASL provides two different kinds of structure types, STRUCT and UNION. The following table shows how these types differ in their use and memory allocation.

ASPECT OF COMPARISON	STRUCT	UNION
use	The STRUCT type permits a group of variables to be treated as a unit instead of as separate entities. STRUCTs can be used to build linked lists, trees and other recursive data structures.	UNIONs are usually used within STRUCTs to represent an element whose type is different under different run-time conditions.
memory allocation	The compiler allocates sequential memory locations for each separate field of a STRUCT The first field starts at offset zero.	The compiler allocates only the amount of storage needed for the longest field of the UNION. Each field starts at the beginning of the UNION at offset zero.

Continued on next page

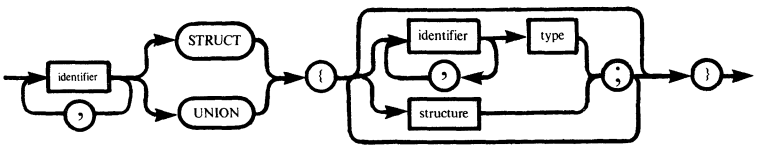
Syntax for structure types

The syntax is the same for STRUCT and UNION types.



Syntax for structure declaration

To declare a variable as a STRUCT or UNION, use the following syntax.



The identifier is optional if the structure is nested within another structure.

Continued on next page

Example of STRUCT declaration

A description of a student consists of name, age, sex and grade.

These four variables can all be placed into a single STRUCT like this:

```
student STRUCT {
    name [30] CHAR;
    age, grade INT;
    sex        CHAR;
    };
```

Example of UNION declaration

The following declaration declares a UNION variable, uval, which may contain a value of type BYTE or CHAR.

```
uval UNION {
    b BYTE;
    ch CHAR;
    };
```

Either a BYTE or a CHAR may be assigned to uval and then used in expressions. The type retrieved from a UNION must be the type most recently stored.

Continued on next page

Declaring structures with TYPDEF

Structures are commonly declared in a TYPDEF declaration. Subsequent variable declarations can then refer to the name of the structure.

Example:

```
TYPDEF NodeInfo STRUCT {
    name [12] CHAR;
    flags     BYTE;
    };

node NodeInfo;     /* declares a variable of type */
                   /* NodeInfo */
nodeArray [256] NodeInfo;  /* declares an array of
                              NodeInfo        */
```

Continued on next page

Syntax for initializing a STRUCT

You may initialize a STRUCT by listing in order the fields of the STRUCT inside braces.

See pages called Variable Declaration: Initialization for general rules for initializing variables.

Example:

```
TYPDEF String STRUCT {
   firstPtr ^ CHAR;
   len, maxLen UNSIGNED;
   firstFlag, prevFlag  BOOLEAN;
   };

line [201] CHAR;

str String := {
   &line[0],
   0, SIZEOF line,
   TRUE, TRUE
   };
```

Continued on next pag

Example of accessing structure fields

The following example shows how you can access different fields of STRUCT.

```
person STRUCT {
   firstName [10] CHAR;
   lastName [10] CHAR;
   age INT;
   UNION {
      unemployed STRUCT {
         sex  CHAR;
         numMonth INT;
         };
      employed STRUCT {
         company [15] CHAR;
         numYears INT;
         };
      };
   };

init () :=
{
   person.firstName := 'Joe       ';
   person.lastName  := 'Brown     ';
   person.age                    := 30;
   person.unemployed.sex         := 'M';
   person.unemployed.numMonths := 7;
   };
```

Note that the field unemployed of the unnamed UNION is referred to as part of the outer STRUCT.

Continued on next page

Syntax for initializing a nested STRUCT

You must nest braces if you are initializing a STRUCT which contains an array field or a named STRUCT or UNION field.

Example:

```
str STRUCT {
   array [5] INT;
   flag BOOLEAN;
   s STRUCT {
      ch CHAR;
      num INT;
      };
   } := {{1,2,3,4,5}, TRUE, {'A',4}};
```

If the nested STRUCTs were unnamed, then you would write 'A', 4 instead of {'A',4}.

Accessing fields of a structure

To access a field of a structure, use both the name of the structure and the name of the field, separated by a period.

To access a field of a nested structure, name each of the outer structures in order, and the field.

If a nested structure is unnamed, then you refer to its fields as part of the outer structure.

Continued on next pag

Partial initialization of a STRUCT

You do not have to initialize all of the fields when you intialize a STRUCT. However, any initializers listed will be assigned to the first fields of the STRUCT.

Syntax for initializing a UNION

A UNION is treated like a STRUCT in its initialization. However, only the first alternative field of a UNION may be initialized.

Example:

```
uval UNION {
    i INT;
    b BYTE;
    un UNSIGNED;
    } := {5};
```

In this example i is assigned the value 5. You cannot initialize the variables b or un in the declaration of uval.

Continued on next page

Nested structures

A UNION or a STRUCT may be nested within another UNION or STRUCT. Any nested structure can be unnamed.

The following example shows a STRUCT nested within a UNION, which is in turn nested within a STRUCT. In this example, the UNION is unnamed.

```
TYPDEF Node STRUCT {
   name [12] CHAR;
   left, right ^ Node;
   typ INT;
   UNION {
      str [8] CHAR;
      s STRUCT {
         val1, val2 INT;
         };
      };
   };
```

This example declares a STRUCT type, Node, which contains an array of 12 characters, two pointers to the same STRUCT type, and a UNION. The UNION contains an array or a STRUCT of two integers.

Continued on next pag

Type compatibility for structures

Two structures are of compatible type if they are equivalent types.

Two types are equivalent if

- they are declared in the same type declaration, or
- one is a TYPDEF name for a type which is equivalent to the other.

Two structures must be of compatible type if they are operands for the assignment operator or for one of the relational operators.

Continued on next page

Operations permitted on structures

This chart lists the operations which are permitted on structures.

See the chapter called Operators for a complete description of each operator.

OPERATOR(S)	DESCRIPTION	SEE THESE PAGES
:=	assign a value	Simple Assignment Operator
= ~= <> <= >=	establish an ordered relationship	Relational Operators
&	take the address of	Pointer Operators
SIZEOF	return the size of, in bytes	SIZEOF Operator
<type>	cast (reinterpret locally) as the specified type	Cast Operator
.	select a field	Field Operator

CHAPTER 6.
FUNCTIONS

Contents

Definition of function

A *function* in DASL is an independent unit which consists of local variables and statements.

A function performs one or more actions which are usually related to one specific task.

You can reference a function from within any other function, as long as the first function has already been declared.

A function may or may not return a value to its caller. A function returns a value either directly, by an assignment to RESULT, or indirectly through a pointer.

Functions and modular programming

Functions allow you to break longer computing tasks into small modules.

DASL programs usually consist of a number of small functions, each dedicated to performing a specific task.

You can write, debug, test, and change functions independently.

Continued on next page

Coming up

The following aspects of functions are discussed in this chapter:

- function declaration
- function call
- pointers as function arguments
- function storage classes.

FUNCTION DECLARATION

Description of function declaration

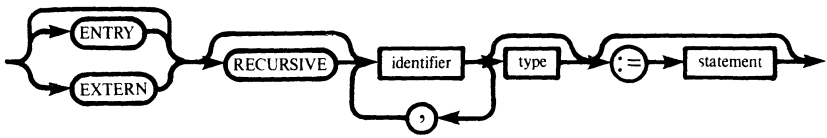
A function declaration declares one or more identifiers to be names of a function.

The function declaration may also include

- storage class(es)
- function type,
- statement body.

Syntax diagram for function declaration

This diagram shows the syntax for a function declaration.



The type is omitted *only* when functions are redeclared as ENTRY or RECURSIVE.

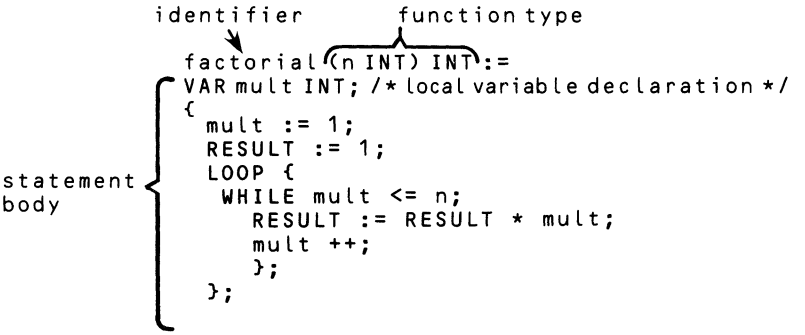
Function declarations may *not* be nested, except EXTERN declarations.

The class specifications, ENTRY, EXTERN, and RECURSIVE, are discussed on the pages called Function Storage Classes.

Continued on next page

Example of a function declaration

The following example shows the different parts of a function declaration.



Continued on next page

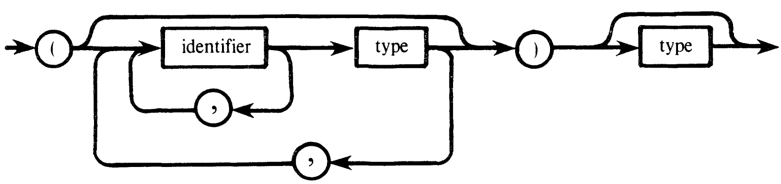
Description of function type

The function type describes the formal parameters and the return value of a function.

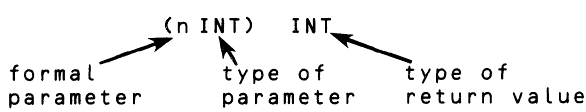
The parameters and return value must be scalar or pointer types.

Syntax for function type

This diagram shows the syntax for a function type.



Example: The type of the function factorial in the previous example is



A function may have more than one formal parameter or none at all.

A function may return either one value or none.

Continued on next page

Definition of formal parameters

The *formal parameters* in a function type are value parameters.

Formal parameters declare the type and the number of arguments which can be passed to the function in a function call.

Parentheses required

If a function has no formal parameters, you still need to include parentheses in the function type declaration.

Example:

```
response ( ) CHAR :=
{

    .
    .
    .
    };
```

Continued on next page

Return value

To have a function return a value, you must

- declare the value in the function declaration, *and*
- assign a value to the predefined variable, RESULT, in one of the statements in the function body.

Example:

```
RESULT := time * mult;
```

Description of statement body

The statement body of a function specifies the code to be executed when the function is called.

The statement may include local variable declarations.

If the statement body is present, only one identifier may be specified.

Continued on next page

Omission of statement body

The statement body is omitted if

- the storage class is EXTERN, or
- the function type needs to be declared before the function body is given, as in the case of mutual recursion.

See the pages called Storage Classes for more information.

Type compatibility for functions

Two functions are of compatible type if their types are equivalent.

Two types are equivalent if

- they are declared in the same type declaration, or
- one is a TYPDEF name for a type which is equivalent to the other.

Continued on next pag

Operations permitted on functions

The only three operators that may be used with functions are the

- function call operator
- address operator
- cast operator.

FUNCTION CALL

Definition

A *function call* is an expression which shifts the flow of control to the beginning of the called function.

A function may be called from within any other function, including itself or the function MAIN.

A function may be called with parameters and it may return one value to the function which called it.

Description of function arguments

The parameters in the function call are referred to as arguments in order to distinguish them from the formal parameters in the function declaration.

Continued on next pag

Example of a function call

This example shows a function call in the MAIN function to the function max.

```
max (a, b INT) INT :=
{
   IF a > b THEN RESULT := a
    ELSE RESULT := b;
   };

ENTRY MAIN ( ) :=
VAR n1, n2, n3 INT;
{
    .                    actual
    .                    arguments
    .
    n3 := max (n1, n2);

                  function
                  call

    };
```

In this example, max returns a value to the calling function MAIN. The value of the function call max (n1, n2) is the larger of n1 or n2.

Continued on next page

Semantic information for function calls

The following rules apply to function calls.

- The expression preceding the parentheses in a function call must be an lvalue referring to the function. See the pages called Lvalues and Variables.

- The expressions inside the parentheses (if any) are the arguments for the function. The arguments must agree in number, and be of compatible type, with the formal parameter specified in the function type.

- The type of the result (if specified) is the same as the result type specified in the function type declaration.

Order of evaluation of arguments

The order of evaluation of the function arguments is undefined by the language and depends on the particular code generator used.

Usually the order of evaluation is not significant. However, it can be cases where argument expressions involve side effects as in the function call, func(p,p++).

Continued on next pag

Description of call by value

All function calls in DASL are 'call by value.' This means that the called function receives a private, temporary copy of each argument, *not* its address.

Within a function, each argument is a local variable initialized to the value with which the function was called. When the function is exited, the values of the local variables are indeterminant.

The values of the formal parameters may be changed within a function. However, these changes do *not* affect the values of the actual arguments in the calling function.

Function calls with pointers

You can *indirectly* modify variables in a calling function by passing pointers as arguments.

The called function may change the value of the object to which a pointer points.

For more information, see the following pages called Pointers As Function Arguments.

POINTERS AS FUNCTION ARGUMENTS

Introduction

In DASL, pointers are frequently used as actual arguments in functions.

Using a pointer as a function argument is the only way in DASL to

- modify a variable in the calling function,
- pass variables of aggregate types to functions, or
- pass function variables to other functions.

Modifying variables in the calling function

All argument passing in DASL is strictly by value.

Therefore, there is no *direct* way for the called function to modify a variable in the calling function.

However, the called function can *indirectly* modify variables in a calling function if pointers are passed as arguments. The called function may change the value of the object to which a pointer points.

Continued on next page

Example of using pointers to modify variables

In this example, the function swap is used to exchange two integers.

```
swap (a, b ^ INT) :=
VAR temp INT;
{
    temp := a^;
    a^ := b^;
    b ^ :=temp;
    };

ENTRY MAIN ( ) :=
VAR x, y INT;
{
    .
    .
    .
    IF x > y THEN swap (&x, &y);
    };
```

Comment: The arguments in the function call swap (&x, &y) are pointers to x and y. In the function swap, the formal parameters a and b also become pointers to x and y. The function swap can change the values x and y (in the MAIN function) by changing the values of the objects to which the pointers point.

Continued on next page

Passing aggregate variables

Function parameters and return values must be of type scalar or pointer. However, you can achieve the effect of passing a variable of *any* type to a function by passing a pointer to that type.

Continued on next page

Example of passing an array

The function maxAge has two parameters which are both pointers to the structure type Person. The function returns the value of the greatest age in an array of Person.

```
TYPDEF Person STRUCT {
   name [30] CHAR;
   age       INT;
   };

maxAge (p, maxP ^ Person) INT :=
{
   RESULT := p^.age;
   LOOP { /* cycle through array using       */
    WHILE ++p <= maxP; /* pointer arithmetic */
      IF p^.age > RESULT THEN RESULT := p^.age;
      };
   };


ENTRY MAIN ( ) :=
VAR personList [100] person;
    max INT;
{
   .
   .
   .
   max:=maxAge(&personList[0],&personList[99]);
   };
```

Continued on next page

Example of passing an array, (continued)

Comment: The arguments in the function call

```
&personList[0], &personList[99]
```

are pointers to the first and last components of the array personList.

The function maxAge, using pointer arithmetic, accesses the actual address of each component of the array. In this example, the values in the array are *not* changed. However, any change to the array within the function maxAge would also change the array in the calling function.

Passing function variables

Sometimes it is useful to develop a general function which can call on alternative functions at each step of a process.

You *cannot* pass a function as a parameter to another function. However, you can pass a pointer to a function, and then call that function indirectly.

Continued on next page

Passing function variables, (continued)

In order to do this, you must first define a function using a TYPDEF declaration. You can then declare

- actual functions of this type, and
- pointers to this type.

Example of passing function variables

In the following example, the function treeWalk traverses a tree and calls on one of two functions, printTree or scanTree, to perform a process at each node.

Continued on next page

Example of passing function variables, (continued)

```
TYPDEF FType (n ^ Node);

treeWalk (tree ^ Node, process ^ Ftype);
{
   visit node of tree

   process ^ (n); /* indirect function call to the */
   .              /*function pointed to by process*/
   .
   };

printTree FType :=
{
   .
   .
   };

scanTree FType :=
{
   .
   .
   };

ENTRY MAIN ( ) :=
VAR tr ^ Node;
{
   .
   .
   treeWalk(tr, &printTree);
   .
   treeWalk(tr, &scanTree);
   .
   .
   };
```

FUNCTION STORAGE CLASSES

Introduction

DASL provides three optional storage classes for function variables:

- ENTRY,
- EXTERN, and
- RECURSIVE

Description of ENTRY and EXTERN classes

The ENTRY and EXTERN class specifications have the same meaning in function declarations as they do in variable declarations.

The class ENTRY in a function declaration causes the function name to be an entry point which may be referenced by other program modules when the modules are linked together.

The class EXTERN causes no storage to be allocated but declares that the function name is defined as an entry point in some program module.

If the storage class of a function is EXTERN, then the statement body is omitted in the declaration but is located in another DASL program module.

Continued on next page

Description of ENTRY and EXTERN classes, (continued)

A function name which has been declared as EXTERN may also be redeclared as ENTRY. In this case, the type is omitted in the ENTRY declaration.

EXTERN declarations in INCLUDE files

EXTERN declarations are often grouped together in an INCLUDE file.

In this way, declarations which are accessed by multiple module only need to be written once. Also, the types of variables and functions do not have to be redeclared in the ENTRY declarations.

Example of ENTRY and EXTERN declarations

In the following example, the INCLUDE file COMMON declares one variable and one function as EXTERN.

The module FILE1 redeclares the variable as ENTRY while the module FILE2 redeclares the function as ENTRY.

The two modules must be linked together using the LINK utility.

Continued on next page

Example of ENTRY and EXTERN declarations, (continued)

```
INCLUDE file: COMMON

EXTERN var INT;
EXTERN func (x INT);
```

```
Module: FILE1

  INCLUDE(COMMON)

  ENTRY var := 1;

  f() :=
  {
       .
       .
       .
       func(y);
       .
       .
       .
       };
```

```
Module: FILE2

  INCLUDE(COMMON)

  ENTRY func :=
  {
       .
       .
       .
       var := x;
       .
       .
       .
       };
```

Continued on next page

Description of RECURSIVE storage class

The RECURSIVE storage class declares that the function may call itself. The keyword RECURSIVE precedes the function name.

Example: This function computes the factorial of an integer recursively.

```
RECURSIVE factorial (n INT) INT :=
{
    IF n = 0 THEN RESULT := 1 /* non-recursive */
     ELSE RESULT := n * factorial(n-1); /* conditions*/

    };
```

Uses of recursion

Recursion may be used in a number of programming applications, such as

- parsing,
- tree and graph processing, and
- sorting.

Recursive routines usually cost more in terms of time but sometimes work better or are easier to read than iterative routines.

Continued on next page

Non-recursive condition

Every recursive function must have a non-recursive condition, in which the function stops calling itself and allows the stack to pop off all of its accumulated return values.

Each call of the function must in some manner bring you closer to the non-recursive condition.

Syntax for mutual recursion

In the case of mutual recursion, the function type needs to be declared before the function body is given.

Subsequently, the function name must be redeclared with the body provided. The type is omitted on the second declaration, but the RECURSIVE class, if needed, must be specified.

Continued on next page

Example of mutual recursion

The following example outlines a case of mutual recursion in which the type of the function f is given before its body.

```
f (x INT);

RECURSIVE g () :=
{
    .
    .
    .
    f(y);
    .
    .
    .
    };

RECURSIVE f :=
{
    .
    .
    .
    g();
    .
    .
    .
    };
```

CHAPTER 7.
EXPRESSIONS

Contents

OVERVIEW

Definition

An *expression* is a piece of DASL code which has a value, and may or may not have an address in memory.

Expressions are used to

- compute a value to be used in statements or declarations, or
- perform actions such as function calls.

Expressions and operators

DASL provides a wide variety of operators which are used to build expressions of different kinds.

Each operator and the expressions associated with it are discussed in the following chapter.

Continued on next page

Associated type of expressions

Every expression in DASL is evaluated to result in a value of some specific type. The type of the result depends on the operator and the kind of operands in the expression.

The type helps determine the meaning of an expression and is used for error checking.

Example: The meaning of the expression a-b depends on the types of both a and b.

IF a is...	AND b is...	THEN type of a-b is...
scalar	scalar	scalar.
pointer	pointer	scalar.
pointer	scalar	pointer.

Coming up

In this chapter we discuss constant expressions and identifier expressions.

CONSTANT EXPRESSIONS

Description

A constant expression may be a

- numeric constant,
- string constant, or
- more than one scalar constant combined with arithmetic operators.

Constant expressions are evaluated at compile time, rather than run time.

Array bounds and case labels are required to be compile-time constant expressions.

Definitions

A *numeric constant* is an integral number.

A *string constant* is a sequence of characters surrounded by single quotes, such as 'abc'.

Continued on next page

Type of numeric constant

The following chart shows how the value of a numberic constant determines its type.

IF the value of a numeric constant...	THEN its type is...
fits in 16 bits	UNSIGNED.
does *not* fit in 16 bits	LONG.
is zero or the largest possible pointer value	• pointer, when used an operation with a pointer, or • UNSIGNED.

Type of string constant

A string normally has the type array of CHAR. A string consisting of a single character may also be considered to be of type UNSIGNED.

Continued on next page

String initialization

A string is initialized with the ASCII value of the characters in the string.

Examples of constant expressions

The following expressions are all constant expressions.

```
'a'
5
'a' *5
3-'b' + ('x'-320)
```

IDENTIFIER EXPRESSIONS

Definition

An *identifier expression* is an expression which refers to a previously declared (either in a variable or function declaration) variable.

Type

The type of an identifier is specified in its declaration.

CHAPTER 8.
OPERATORS

Contents

OVERVIEW

Introduction

DASL provides a wide range of operators. An operator indicates what is to be done with expressions.

In a sense, operators are like verbs in an English sentence. They perform some process with expressions or establish a relationship between expressions.

Definition of operand

An *operand* is an expression upon which an operator acts.

Continued on next page

Uses of operators

Operators have several uses, including:

- assigning values
- performing arithmetic processes
- establishing relationships between operands
- performing processes at the bit level
- performing pointer processes
- indicating the size of an operand
- reinterpreting locally an operand's type definition
- selecting an element of an aggregate type.

Operator precedence

There is an order of precedence for DASL operators reflecting the order in which they are performed.

This guide presents the DASL operators by the types of processes they perform or the relationships they establish, not in order of operator precedence. At the end of this chapter there is a discussion of operator precedence, with a chart for easy reference.

Continued on next page

Coming up

The rest of this chapter addresses the following aspects of operators:

- lvalues and variables
- type compatibility
- arithmetic conversion of scalars
- arithmetic operators used with pointers
- each DASL operator
- operator precedence.

Continued on next page

Operator information

Each operator is described in terms of

- what it does
- syntax format, and
- semantic information, including
- operand requirements
- grouping, and
- results.

Examples are provided to illustrate how each operator can be used.

LVALUES AND VARIABLES

Operators and expressions

Certain operators require their operands to be lvalues or variables, and yield results which are lvalues or variables. Other operands do not involve lvalues or variables.

It is important to distinguish between the terms lvalue and variable and to recognize expressions which are neither lvalues nor variables.

Definition of lvalue

An *lvalue* is an expression which refers to an object in memory.

Any expression which you can take the address of is an lvalue.

Definition of variable

A *variable* is an lvalue which can be modified. The only lvalue which is not a variable is a string constant, which cannot normally be modified.

An expression which can appear on the left side of an assignment statement is a variable.

Continued on next page

Distinction between lvalues and variables

The only difference between the class of lvalues and the class of variables is that lvalues include string constants.

A string constant is considered an lvalue because it has an address. However, it is not a variable because it cannot be modified.

Examples of lvalues and variables

The following expressions represent different kinds of lvalues. (struct is a structure, list is an array, and p is a pointer)

```
p
length^
struct.field
list[5]
'hello'
```

All of these expressions are also variables except for the string 'hello'.

Continued on next page

Numeric constants

In addition to lvalue and variable operands, operators may involve numeric constants.

A numeric constant is *not* an lvalue because it does not have an addresss in memory.

SIMPLE ASSIGNMENT OPERATOR (:=)

Syntax

The syntax of the simple assignment operator is

expression —> expression := expression

Example:

```
upperLimit := 10;
nextDay    := today + 1;
```

The simple assignment operator requires two operands. The assignment operator groups right to left.

Semantic information

The operands must be of compatible types (see pages called Type Compatibility).

The result of an assignment expression:

- has the value and type of the left operand
- is *not* an lvalue.

Continued on next page

SIMPLE ASSIGNMENT OPERATOR (:=), Continued

Using more than one := operator

You can use more than one assignment operator in a statement if you want to assign the same value to more than one operand.

Example:

You could combine the following two assignment statements:

```
MULT : = 1;
RESULT: = 1;
```

into the following assignment statement:

```
MULT := RESULT := 1;
```

RESULT gets the value of 1, and MULT gets the value of the expression (`RESULT := 1`).

Continued on next page

SIMPLE ASSIGNMENT OPERATOR (:=), Continued

Examples

```
ENTRY MAIN() := /* assignment operator is always */
                /* used in function declaration  */
VAR char1   CHAR;
    sum, num1, num2 INT;
    counter UNSIGNED;
{
    char1: ='A';/* assigns value of ASCII 'A' to  *
                /* char 1
    num1  := 231;/* assigns decimal 231 of num1    *
    num2  := counter := 0; /* assigns decimal 0 to
                              counter              *
                /* and to num2                     *
    sum := num1 + num2; /* assigns value of expressio
                                                    *
                        /* num1 + num2 to sum      *
    };
```

TYPE COMPATIBILITY

Type compatibility rules

Several operators require their operands to be of compatible types. The following chart lists the rules for type compatibility.

TYPE	RULE(S)
scalar	Any two scalar types are compatible.
pointer	Two pointer types are compatible if they both point to compatible types of the same size. A pointer type is also compatible with • the constant number 0, *and* • the constant number equal to the largest possible value of a pointer.
array	Two array types are compatible if they have • the same upper bound, *and* • components which are compatible types of the same size.
any type	Any two types are compatible if they are equivalent types. Two types are equivalent if • they were the same type declaration, *or* • one was a TYPDEF name for a type which is equivalent to the other.

Continued on next page

Order of evaluation of operands

DASL does not specify in what order the operands of an operator are evaluated, except in the cases of the comma, conditional, and logical operators.

In most cases, the order of evaluation does not make any difference in the result. However, it can make a difference in cases where the evaluation of one operand has side effects which affect the other operand.

Example:

In the statement `a := b + b++;`

either b or b++ could be evaluated first. In this case, the value that a assumes depends on the particular order of evaluation.

ARITHMETIC CONVERSION OF SCALARS

Introduction

Arithmetic operators, such as additive and multiplicative operators, perform type conversions on scalar operands. These type conversions insure that scalars are always compatible when used with arithmetic operators.

Operators that perform conversion

The following types of operators perform the scalar conversions described on these pages:

- additive (+)
- multiplicative (* / %)
- bit (&& || !!).

Continued on next page

Scalar operands

A scalar describes an integer value.

The following table describes the six scalar types.

TYPE IDENTIFIER	NUMBER OF BYTES	SIGNED	UNSIGNED
BOOLEAN	1		X
BYTE	1		X
CHAR	1		X
UNSIGNED	2		X
INT	2	X	
LONG	4	X	

Continued on next page

Type conversions

When you use scalar operands with arithmetic operators, the following type conversions occur.

IF one operand is type...	AND the other operand is...	THEN both operands and the result are type...
LONG	any scalar type	LONG.
INT	*not* type LONG	INT.
not LONG or INT	*not* type LONG or type INT	UNSIGNED.

Mixing INT and UNSIGNED operands

Be careful when you use INT and UNSIGNED operands together.

If and UNSIGNED operand has a value large enough to affect the bit reserved for sign indication, this will affect the result.

Continued on next page

Examples of arithmetic conversion

The following examples do not reflect a realistic DASL program, but are included only to illustrate arithmetic conversion of scalars.

```
VAR       bool        BOOLEAN;
          byt         BYTE;
          ch          CHAR;
          integer     INT;
          unsign      UNSIGNED;
          longVAR     LONG;

{
    .
    .
    .
    };
```

The following three examples illustrate the conversions involved with expressions using the above variables.

The expression longVar + bool gives a result of type LONG, and the operand bool is converted to type LONG.

The expression integer − unsign gives a result of type INT, and the operand unsign is converted to type INT.

The expression ch * byt gives a result of type UNSIGNED, and the operands are converted to type UNSIGNED.

Continued on next page

Conversion with assignment operators

In a statement which has an assignment operator, the type of the expression to the right of the assignment operator is converted to the type of the variable to the left of the assignment operator.

Example:

```
VAR page      INT;
    counter   CHAR;
    total     LONG;
{
   total := page + counter;
   .
   .
   .
   };
```

The type of the result of the expression page + counter is type INT, since one of its operands, counter, is type INT.

Then it is converted to type LONG, which is the type of total, the variable to the left of the assignment operator.

ADDITIVE OPERATORS (+, −)

Description

The + operator yields the *sum* of its operands.

The − operator yields the *difference* of its operands.

Syntax

The syntax of the additive operators is:

expression —> expression + expression

Example:
```
nextDay := today + 1;
```

expression —> expression − expression

Example:
```
text := pageSize - margin;
```

The additive operators each require two operands.
The operators group left to right.

Continued on next page

Semantic information for the + operator

IF operand types are...	THEN the result is...
both scalar	a scalar. (See pages called Arithmetic Conversions.)
a pointer and a scalar	a pointer of the same type as the pointer operand. The pointer is incremented by the value of the scalar times the size of the object to which the pointer points. (See pages called Arithmetic Operators Used With Pointers.)

Continued on next page

Semantic information for the - operator

IF operand types are...	THEN the result is...
both scalar	a scalar that is either type LONG or INT. If at least one operand is type LONG, the result it type LONG. If *neither* operand is type LONG, the result is type INT.
a pointer and a scalar	a pointer of the same type as the pointer operand. The pointer is decremented by the value of the scalar times the size of the object to which the pointer points.
two pointers of compatible types	a signed scalar. This signed scalar is given by the difference of the two pointer values, divided by the size of the object to which the pointers point. (See pages called Arithmetic Operators Used With Pointers.)

MULTIPLICATIVE OPERATORS (* / %)

Description

The following table describes multiplicative operators for DASL.

OPERATION	DASL SYMBOL	DESCRIPTION
multiplication	*	yields the product of two operands.
division	/	yields the quotient of the first operand divided by the second operand, dropping any remainder.
modulo	%	yields the remainder (only) after division of the first operand by the second.

MULTIPLICATIVE OPERATORS (* / %), Continued

Syntax

expression —> expression * expression

Example:
```
area := width * length;
```

expression —> expression / expression

Example:
```
halfTime := fullTime / 2;
```

expression —> expression % expression

Example:
```
seed := K1 % 3;
```

Continued on next page

Semantic information

The three multiplicative operators (* / %) have the following semantic requirements in common. They all

- require two operands, which must be SCALAR,
- group left to right, and
- yield a scalar result (see pages called Arithmetic Conversion).

For the division (/) and modulo (%) operators, the following table describes the result.

IF...	THEN the result is...
either operand is signed	signed.
neither operand is signed	unsigned.
the second operand is zero	undefined (however, no error occurs).
The sign of the result of a statement using the % operator is always the same as the sign of the first operand.	

Continued on next page

Examples

The following examples illustrate uses of the multiplicative operators:

```
d := a * 2; /* If a is 3, assigns the value 6      */
            /* (3*2) to d                          */

e := b / 4; /* If b is 10, assigns the value 2     */
            /* (10/4 without a remainder) to e     */

f := c % 4; /* If b is 15, assigns the value 3     */
            /* 15/4 leaves a remainder of 3) to f  */
```

INCREMENT AND DECREMENT OPERATORS (+ + − −)

Description

The + + operator *increments* its operand.

The − − operator *decrements* its operand.

Two ways to use the increment/decrement operators

The increment and decrement operators can be used

- preceding the operand, *or*
- following the operand.

The following table indicates the processing that occurs when the + + or − − operator is used with an operand which is part of an expression involving other operators.

IF the + + or − − operator...	THEN the operand is incremented or decremented...
precedes the operand	*before* the value of the operand is used in the expression.
follows the operand	*after* the value of the operand is used in the expression.

Continued on next page

INCREMENT AND DECREMENT OPERATORS (+ + − −), Continued

Syntax

expression —> expression + +

Example:
```
page := counter++;
```

expression —> expression − −

Example:
```
lineCount--;
```

expression —> + +expression

Example:
```
nextDay := ++today;
```

expression —> − −expression

Example:
```
length^ := newLength;
```

Continued on next page

INCREMENT AND DECREMENT OPERATORS (+ + – –), Continued

Semantic information

The increment and decrement operators require one operand, which must be a variable. The operand must be a scalar or pointer.

If the operand is a...	THEN it is incremented or decremented by...
scalar	one.
pointer	the length of the object to which the pointer points. (See pages called Aritmetic Operators Used with pointers.)

Caution

Do not use the increment or decrement operators with an operand that is used more than once in one expression. The order of evaluation is undefined.

Be careful when using the increment or decrement operators with the logical OR and the logical AND operators (|&). All expressions may *not* be evaluated, and the increment or decrement *not* performed.

Continued on next page

Examples using the + + and – – operators with scalars

The following four examples illustate uses of the increment and decrement operators with scalar operands.

```
nextDay  := ++today;      /* assigns the value of   */
                          /* today +1 to nextDay    */
            --today;      /* give the value today   */
                          /* -1 to today            */

counter++;                /* adds 1 to counter      */

spaceLeft := counter--;   /* assigns the value of   */
                          /* counter before         */
                          /* decrement) to          */
                          /* spaceLeft, then        */
                          /* decrements value of    */
                          /* counter by 1           */
```

ARITHMETIC OPERATORS USED WITH POINTERS

Introduction

You can use the following arithmetic operators with pointers:

- addition (+)
- subtraction (−), and
- increment and decrement (++ −−)

The following pages describe how each of these arithmetic operators is used with pointers. For additional information on the operators, see pages called Additive Operators and Increment and Decrement Operators.

Usually used with arrays

Pointer arithmetic is usually used with pointers to arrays. This allows you to point to successive components of an array. You can access any component of an array through using pointers and pointer arithmetic.

Continued on next page

Using the + operator with a scalar and pointer

Only scalars can be added to pointers.

The pointer is incremented by the value of the scalar times the size of the object which the pointer points.

Continued on next page

Example using the + operator with a scalar and pointer

The following example is a progam excerpt that illustrates how pointer arithmetic is used to access the components of an array.

```
print (p, maxp ^ INT):=
{
   LOOP {
    WHILE p <= maxp;

/* function prints the component pointed to by p  */

      p := p + 1;
      };
   };

ENTRY MAIN ( ) :=
VAR intArray [10] INT;
{
   print(&intArray[0], &intArray[9]);
   .
   .
   .
   };
```

When the loop is first entered, p points to the first component of intArray. This first component is printed.

Continued on next page

Example using the + operator with a scalar and pointer, (continued)

The statement p := p + 1; increments the pointer p by 2 (the size of intArray[0] times the scalar 1). This results in p pointing to the second component of intArray.

On each successive pass through the loop, another array component is printed, and p is incremented to point to the next component of the array.

Using the minus (–) operator with a pointer

You can subtract either a scalar or another pointer from a pointer.

Subtraction of a scalar from a pointer is performed in the same manner as addition of a scalar and a pointer, except the pointer is decremented.

Subtraction of one pointer from another pointer results in the value of the difference between the two pointer values, divided by the size of the object to which the pointers point. For two pointers to components of an array, this simply means the difference between the array indices.

Continued on next page

Example of using the minus (–) operator with two pointers

```
VAR i INT;
    p, q ^ INT;
    array [20] INT;
{
   p := &array[3];
   q := &array[9];
   i := q - p;
   };
```

The difference between p and q is 12 bytes.

This value (12) is divided by 2, since p and q are pointers to integers, which have a size of 2 bytes.

Thus, the effect of i := q − p; is to assign the value 6 to i.

Using the increment/decrement operators with pointers

The increment (++) and decrement (−−) operators can be used with a pointer to increment or decrement the address to which the pointer points. The address is incremented or decremented by the size of the object to which the pointer points.

UNARY ARITHMETIC OPERATORS (– ∿ ∿)

Description of unary negation (–) operator

The unary negation (–) operator gives the *negative* of its operand.

Description of the ∿ ∿ operator

The ∿ ∿ operator gives the *one's complement* of its operand.

The one's complement operator replaces the 0 digits, at the bit level, with 1 digits, and replaces 1 digits with 0 digits.

Example:

The binary number for 11 is 0000 0000 1011.

The one's complement of 11 is 1111 1111 0100.

Syntax

expression —> – expression

Example:

```
negNum :=  - num;
```

expression —> ∿ ∿ expression

Example:

```
complement := ∿∿b;
```

Continued on next page.

Semantic information for the unary negation (-) operator

The unary negation (-) operator requires one operand, which must be a scalar type.

IF the *operand* is ...	THEN the *result* is type ...
type LONG	LONG.
any scalar type except LONG	INT.

Continued on next page

Examples using the unary negation (–) operator

The following examples illustrate uses of the unary negation (–) operator.

```
VAR seed INT;
    val1, val2 LONG;
    num1, num2 UNSIGNED;
{
   seed := -3;              /* assigns the value      */
                            /* negative 3 to seed     */
   num1 := -(val2 + num2); /* assigns the negative   */
                            /* of the result of       */
                            /* (val2 + num1) to num1  */
   num2 := -seed;           /* assigns the value      */
                            /* positive 2 [the neg-   */
                            /* ative of -3 (from      */
                            /* first statement)] to   */
                            /* num2                   */
   };
```

Continued on next page

Semantic information for ∿∿ operator

The one's complement operator (∿∿) requires one operand, which must be a scalar.

IF the *operand* is...	THEN the *result* is type...
type LONG	LONG.
type INT	INT.
any scalar type *except*. o LONG, or o INT	UNSIGNED.

Example using the ∿∿ operator

The following example illustrates the use of the one's complement operator.

```
allColors := ∿∿black;    /* This would result in   */
                         /* all the bits except    */
                         /* black being on.        */
```

RELATIONAL OPERATORS
(= ∿= < > <= >=)

Description

The relational operators compare the value of the first operand to the value of the second operand, in terms of magnitude with sign (i.e., equal to, greater than, less than). The value of the relationship is true (1) or false (0).

Syntax

expression —> expression (relational operator) expression

Example:

```
sum = 50;
RESULT := address1 >= address2;
```

Continued on next page

DASL relational operator

The folloiwng table lists the DASL relational operators, indicates the symbol for the operator, and provides an example of the operator in use.

OPERATOR	SYMBOL	EXAMPLE
equal	=	length = width
not equal	∿=	length ∿ = width
greater than	>	IF lines > limit THEN overflow ( );
less than	<	IF year < currentYear THEN error ( );
greater than or equal to	>=	RESULT := address1 >= address2;
less than or equal to	<=	SUM <= 50

Continued on next page

Operand information

The relational operators require two operands, which must be of compatible types (see pages called Type Compatibility).

The relational operators group left to right. However, A<B<C is not very useful because it compares the result of A<B, which is true or false, with C.

Sign of comparison

The following table indicates the sign of the comparison of different types of operands, using relational operators.

IF the operands...	THEN the comparison is...
are both unsigned scalars	unsigned.
include at least one signed scalar	signed.
are pointers	unsigned.
are aggregate type, including • arrays • structures, or • unions	made one byte at a time, starting with the lowest address. The comparisons use unsigned arithmetic.

Continued on next page

Result

The relational operators yield a result of 1 or 0.

IF the relationship indicated by the operator is...	THEN the result is...
true	1.
false	0.
The type of the result is BOOLEAN.	

Example 1: Using relational operators with scalars

```
VAR status, linesPrinted, page INT;
{
    page := 50;
    status := linesPrinted > page;

/* the comparison is signed because at least one
   operand is signed. */
   };
```

If linesPrinted is greater than 50 (page), then the comparison is true, so the value 1 is assigned to status.

Continued on next page

Example 2: Using relational operators with arrays

```
VAR num [3] BYTE := {1,2,4};
    primeNum [3] BYTE := {1,2,3};
{
    IF num < primeNum THEN printNum();
    .
    .
    .
    };
```

The comparison is unsigned because aggregate operands (such as arrays) use unsigned arithmetic.

The first elements of each array are compared.

In this example, the first elements of each array are equal. Similarly, the second elements are equal.

The third elements of each array are then compared. Since 4 does *not* equal 3, a result of the comparison is returned. Because 4 is *not* less than 3, the result is 0 (false).

LOGICAL OPERATORS (& |)

Description of the & operator

The binary & operator indicates a logical *AND* condition, resulting in 1 if *both* operands are nonzero. If either operand is 0, then the result is 0.

Description of the | operator

The binary | operator indicates a logical *OR* condition resulting in 1 if *either* operand is nonzero. If both operands are zero, then the result is 0.

Continued on next page

True/false relationship

The logical operators are usually used to compare two operands involving relational expressions giving a value of true (one) or false (zero).

IF the left operand is...	AND the right operand is...	THEN the result using the binary & operator is...	AND the result using the binary | operator is...
true	true	true	true.
true	false	false	true.
false	true	false	true.
false	false	false	false.

Syntax

expression —> expression & expression

Example:
```
status := (sum > minimum) & (sum < limit);
```

expression —> expression | expression

Example:
```
IF (linesPrinted>page) | (linesPrinted = 0)
   THEN newpage();
```

Continued on next page

Semantic information

The logical operators require two operands, which must be a scalar or pointer (both operands do *not* have to be the same type).

The logical operators group left to right.

The result is always unsigned.

Evaluation of operands

The left operand is always evaluated first. The right operand is *not* evaluated unless needed to determine the result.

IF the first operand is...	AND the operator is...	THEN the second operator is...
true	&	evaluated.
	|	*not* evaluated.
false	&	*not* evaluated.
	|	evaluated.

Continued on next page

Order of precedence

The binary & operator has a higher precedence than the | operator.

Examples using logical operators

Example 1:

```
status := (sum > minimum) & (sum < limit);
```

If the expression sum > minimum is true (sum is greater than minimum) *and* the expression sum < limit is also true (sum is less than limit), then the value 1 is assigned to status.

If either expression is false, then the value 0 is assigned to status.

Continued on next page

Examples using logical operators, (continued)

Example 2:

```
VAR p ^ INT;
    b ^ INT;
    validPointer BOOLEAN;
{
   validPointer := p | b;
   .
   .
   .
   };
```

if p and b are both null pointers, then the value 0 is assigned to validPointer.

If either p or b point to an address of an object in memory then the value 1 is assigned to validPointer.

CONDITIONAL OPERATOR (?:)

Description

The conditional operator results in the value of the second or the third operand, depending on the value of the first operand.

IF the value of the first operand is...	THEN the result is the value of the...
true (nonzero)	second operand.
false (zero)	third operand.

Syntax

expression —> expression ? expression : expression

Example:

```
long side := (length > = width) ? length : width;
```

Continued on next page

Semantic information

The conditional operator requires three operands. The first operand must be a scalar or pointer. The second and third operands must both be scalars or compatible pointers.

If the second and third operands are compatible pointers, the type of the result is the type of the pointer.

If the second and third operands are scalars, the type of the result depends on the types of the operands:

IF one operand is...	AND the other operand is...	THEN both operands and the result are type...
type LONG	any scalar type	LONG.
type INT	*not* type LONG	INT.
not type LONG or type INT	*not* type LONG or type INT	UNSIGNED.
The conditional operator groups right to left.		

Continued on next page

Evaluation of operands

Only one of the second and third operands is evaluated, depending on the value of the first operand.

Example

The following example illustrates a use of the conditional operator.

```
longSide := (length >= width) ? length : width;
```

IF length is ...	THEN longSide gets the value of...
greater than or equal to width	length.
not greater than or equal to width	width.

SHIFT OPERATORS (<< >>)

Description

The shift operators (<< >>) shift the value of the first operand left or right by the number of bits indicated by the value of the second operand.

The << operator shifts to the left.

The >> operator shifts to the right.

Syntax

expression —> expression << expression

Example:

```
doubleSize := size << 1;
```

expression —> expression >> expression

Example:

```
flag := b >> 2;
```

Semantic information

The shift operators require two operands, which must be scalars

The shift operators group left to right.

Continued on next page

Shift fill

The shift caused by the shift operators is either arithmetic (sign fill) or logical (zero fill).

IF the operator is	AND the first operand is...	THEN the shift will be...
>>	signed	arithmetic (sign fill).
	unsigned	logical (zero fill).
<<	either signed or unsigned	logical (zero fill). *NOTE*: This may affect the sign bit.

Continued on next page

Result

The type of the result is determined by the first operand.

IF the first operand is type...	THEN the result is type...
LONG	LONG.
INT	INT.
any scalar *except* • LONG or • INT	UNSIGNED.
If the value of the second operand is greater than or equal to the number of bits in the result, the result is undefined.	

Continued on next page

Examples of the shift operators

Example 1:

```
e := 11;
f := 2;
c := e << f;
```

The statement e << f; shifts the value of e left two bits. Thus e (1011 as a binary number) becomes 101100.

Example 2:

```
VAR c, d BYTE;
{
    c := 14;
    d := 3;
    f := c >> d;
    };
```

The statement c >> d; shifts the value of c right three bits, using zero fill (since c is *not* signed). Thus c (1110 as a binary number) becomes 0001.

BIT OPERATORS (&& || !!)

Description

The bit operators compare two operands bit by bit. The result depends on the logical relationship established by the bit operator.

THIS operator...	ESTABLISHES this logical relationship between operands...
&&	AND.
||	inclusive OR.
!!	exclusive OR.

One/zero relationship

The following table indicates the result for each bit operator for each possible combination of operand bit patterns.

&&	0	1
0	0	0
1	0	1

||	0	1
0	0	1
1	1	1

!!	0	1
0	0	1
1	1	0

Continued on next page

Syntax

expresssion expression && expression

Example:
```
b && 2
```

expression —> expression || expression

Example:
```
flag := b || 3;
```

expression —> expression !! expression

Example:
```
flagOff := b !! 2;
```

Semantic information

The bit operators require two operands, which must be scalars. They yield a scalar result (see pages called Arithmetic Conversion).

The bit operators group left to right.

Continued on next page

Examples of the bit operators

The following examples illustrate uses of the three bit operators. This obviously is *not* a realistic program sample, but is included to demonstrate how each of the bit operators works.

```
VAR flag, error INT;
{
   flag   := 27;
   error: := 42;
   a := error && flag;
   b := error || flag;
   c := error !! flag;
   .
   .
   };
```

Example of the result using the && operator

The expression error && flag causes the following comparison and result:

0001 1011 binary number for 27 (flag)
0010 1010 binary number for 42 (error)

0000 1010 result

Note: Only those corresponding bits that have a one (1) in both operands cause a 1 to be place in the corresponding bit of the result.

Continued on next page

Example of the result using the || operator

The expression error || flag causes the following comparison and result:

0001 1011 binary number for 27 (flag)
0010 1010 binary number for 42 (error)

0011 0001 result

Note: Only those corresponding bits which have a 1 in either operand cause a 1 to be placed in the corresponding bit of the result.

Example of the !! operator

The expression error !! flag causes the following comparison and result:

0001 1011 binary number for 27 (flag)
0010 1010 binary number for 42 (error)

0011 0001 result

Note: Those corresponding bits which have a 1 in only one operand cause a 1 to be placed in the corresponding bit of the result.

Continued on next page

Example using the bit operators

The following example illustrates the use of each bit operator (&& || !!).

```
DEFINE (black, 0)
DEFINE (blue, 01)
DEFINE (yellow, 02)
DEFINE (green, 04)
DEFINE (red, 010)
DEFINE (brown, 020)
DEFINE (violet, 040)
DEFINE (orange, 0100)
DEFINE (pink, 0200)

TYPDEF Colors BYTE;

adjustColors () :=
VAR colorsInPicture Colors := blue || yellow || green;
{
   IF colorsInPicture && blue THEN highliteWater();
   .                     /* highlights blue          */
   .
   .
   colorsInPicture := colorsInPicture || brown;
   .                     /* adds brown to picture    */
   .
   .
   IF colorsInPicture && pink THEN
      colorsInPicture := colorsInPicture !! pink;
                         /* removes pink from picture*/
   };
```

POINTER OPERATORS (^ &)

Description of the address operator

The address operator (&) yields a pointer to the object referred to by its operand. In other words, the & operator takes the address of the operand following it.

Description of the indirection operator

The indirection operator (^) yields an lvalue referring to the variable to which the operand (a pointer) points.

For more information on pointers, see the pages called Pointer Type and Using Arithmetic Operators with Pointers.

Syntax

expression—> & expression

Example:

```
next := & counter;
```

expression —> expression ^

Example:

```
p2^ := temp;
```

Continued on next page

Semantic information for the & operator

The & operator (address operator) requires one operand, which must be an lvalue.

The type of the result is a pointer to the type of the operand.

Example using the & operator

The following example illustrates a use of the address operator (&).

```
VAR text CHAR;
    p ^ CHAR;
{
   p := &text;  /* assigns the address of text to  */
                /* the pointer p                   */
   };
```

Semantic information for the ^ operator

The indirection operator (^) requires one operand, which must be a pointer.

The type of the result is the type to which the operand points.

Continued on next page

Example using the ^ operator

The following example illustrates a use of the indirection operand (^). It also shows a use of the address operator (&).

```
VAR firstCh, text CHAR;
    p ^ CHAR;
{
   firstCh := 'A';/* assigns ASCII 'A' to firstCh    */
   p := &firstCh; /* p gets the address of firstCh*/
   text := p^;    /* assigns the character pointed*/
                  /* to by p (in this case, 'A'   */
   .
   .
   .
   }
```

SIZEOF OPERATOR (SIZEOF)

Description

The SIZEOF operator gives the size, in characters or bytes, of its operand.

Syntax

expression —> SIZEOF expression

Example:

```
length := SIZEOF dateArray;
```

expression —> SIZEOF < type >

Example:

```
typeSize := SIZEOF < INT >;
```

Semantic information

The SIZEOF operator requires one operand, which may be any type *except* function.

The result is an UNSIGNED constant.

Continued on next page

SIZEOF with arrays

If you don't know how large an array is, pass a pointer to the array[0], and then use SIZEOF with the array and divide by array[0].

Example

The following example illustrates a use of the SIZEOF operator.

```
DEFINE (fileSize, 30)

TYPDEF Item STRUCT {
    flag CHAR;
    filename [fileSize] CHAR;
    action CHAR;
    };

itemptr ^ Item;
{
    < ^ BYTE > itemptr := $ALLOC(SIZEOF itemptr^);
    };
```

The last line, < ^BYTE > itemptr := $ALLOC(SIZEOF itemptr^); assigns to itemptr (which has been cast as a pointer to a byte) the result of the function $ALLOC, which uses the argument (SIZEOF itemptr^).

Continued on next page

Example, (continued)

(SIZEOF itemptr^) is converted into a constant at compile time. The size of itemptr is the total size, in bytes, of the components of the structure item, to which itemptr points. Thus, (SIZEOF itemptr^) is converted to a constant of 32:

- the size of flag is 1,
- the size of filename is 30 (since fileSize is defined as 30), and
- the size of action is 1.

CAST OPERATOR (< >)

Description

An expression preceded by a type in corner brackets causes the operand to be considered to be the specified type.

The cast operator (< >) allows the type definition of the operand to be locally reinterpreted to eliminate type incompatibilities.

Syntax

expression <type> expression

Example:

```
< ^ CHAR > p := b;
```

Continued on next page

Semantic information

The cast operator requires one operand.

The result depends on the declared type of the operand and the cast type.

IF the operand is...	AND the cast type is...	THEN...
• a scalar or • pointer	• a scalar or • pointer	• the value of the operand is converted to the cast type, and • the result is not an lvalue, if the size changes.
• a scalar or • pointer	any type other than • scalar or • pointer	the result is an lvalue referring to an object of the cast type at the memory location referenced by the operand.
an lvalue which is not • a scalar or • pointer	any type	the result is an lvalue referring to an object of the cast type at the memory location referenced by the operand.

Continued on next page

Semantic information, (continued)

If the operand is a string constant of more than one character, or the cast type is other than a scalar or pointer, the operand will be padded with blanks if cast to a longer type. You must *not* cast to a shorter type is the operand is a string constant. This results in an error.

Caution

You should use the cast operator sparingly because type-checking is bypassed. Very few programs require extensive casting.

Continued on next page

Example

The following example illustrates a use of the cast operator.

```
VAR p ^ CHAR;
    i INT;
{
   i := i + <INT> p;
   .
   .
   .
   };
```

The expression <INT> p recasts p (which was defined as a pointer to a character) to an integer. Thus, the 16 bit value which resides in p will be treated as an INT (signed two-byte integer) for the statement i := i + <INT>p;.

The pointer p is unchanged by this process. If it is used in subsequent statements, it is treated as a pointer.

UNARY NOT OPERATOR (∿)

Description

The unary not operator (∿) results in the value of 1 or 0, depending on the operand's current value.

IF the value of the operand is...	THEN the result is...
nonzero	0.
zero	1.

Syntax

expression —> ∿ expression

Example:

```
If ∿flag THEN a := c;
```

Semantic information

The unary not operator requires one operand, which must be a scalar or pointer.

The type of the result is UNSIGNED.

Continued on next page

Example

The following example illustrates a use of the unary not operator.

IF ∿ ascending (p, q) THEN switch (p, q);

IF the value returned from the function ascending is...	THEN the value of ∿ ascending is...	AND the function switch...
zero	1	is performed.
nonzero	0	is *not* performed

SUBSCRIPTING OPERATOR ([ ])

Description

The subscripting operator ([ ]) references a component of an array.

For more information on arrays, see the pages called Array Type.

Syntax

expression —> expression [expression]

Example:

```
textArray[1]
```

Semantic information

The subscripting operator requires two operands:

- the left operand must be an lvalue referring to an array, and
- the right operand must be a scalar.

The result is an lvalue referring to the selected component of the array. The type of the result is the type of the component of the left operand (the array).

Continued on next page

Numbering of components

The components of an array can be referenced by sequential numbers. The first component is referenced by 0.

The value of the right operand indicates the number of the component being referenced. This value is also called a subscript, or an index, of an array.

Continued on next page

Examples

The following examples illustrate uses of the subscipting operator.

```
VAR charArray[10] CHAR;   /* declares charArray to */
                          /* be an array of 10     */
                          /* characters            */
    charCount INT;
{
   charCount := 1;
   charArray[0] := 'A';   /* assigns ASCII 'A' to  */
                          /* the first component of*/
                          /* charArray */
   charArray[8] := 'I';   /* assigns ASCII 'I' to  */
                          /* the ninth component of*/
                          /* charArray */
   charArray[charCount]   /* assigns ASCII '8' to  */
     := '8';              /* the second component  */
                          /* of charArray, since   */
                          /* charCount has a value */
                          /* of 1 */
   };
```

Continued on next page

Using the subscript operator with multi-dimensional arrays

Multi-dimensional arrays are arrays that have arrays as components. Multi-dimensional arrays require subscripting operators for each dimension.

Example:

```
VAR textLine [15] CHAR;  /* array of 15 characters*/
    page [24] [15] CHAR; /* array of 24 arrays of */
                         /* 15 characters         */
{
   page[1][6] := textLine[2];
   .
   .
   .
   };
```

The statement p[1][6] := textLine[2] assigns the second component of the array textLine to the sixth position of page's second array.

FIELD OPERATOR (.)

Description

The field operator (.) selects a member of a STRUCT or UNION. For more information on STRUCTs and UNIONs, see the pages called Structured Types.

Syntax

expression —> expression.identifier

Example: `date.year`

Semantic information

The operator requires two operands:

- the *left operand* must be an lvalue referring to a structure
- the *right operand* must be an identifier naming a component of a structure referred to by the left operand.

The result is an lvalue referring to the named component of the STRUCT or UNION.

Continued on next page

Example using the field operator

The following example illustrates a use of the field operator.

```
VAR date STRUCT { month, day, year INT; };
{
    IF date. month > 12 THEN error();
    .
    .
    .
    };
```

The expression date.month selects the component month from the STRUCT date. If month is greater than 12, then the error function is performed.

FUNCTION CALL OPERATOR (( ))

Description

The function call operator (( )) calls a function.

Calling a function involves an lvalue referring to the function and parameters to the function. These parameters correspond to the formal parameters in the function type declaration.

For more information on functions, see the pages called

- Function Declaration and Type
- Function Call.

Syntax

expression —> expression (expression−list)

The expression list enclosed by the function call operator (( )) is optional. However, the function call operator is required, even if there is no expression list of parameters.

Example:

```
getArray(p^, size)
```

Continued on next page

Semantic information

The first operand must be an lvalue referring to the function.

The remaining expression, if any, are the arguments to the function. These expressions must

- agree in number with the formal parameters specified in the function type declaration, and
- be of a compatible type with the corresponding formal parameters.

The type of the result (if a result is specified) is that specified in the function type declaration.

Order of evaluation

The order of evaluation of the function agruments is undefined. This order of evaluation depends on the particular code generator used.

Continued on next page

Example

The following program excerpt demonstrates the relationship between the use of the function call operator and the function type declaration.

```
getArray (p ^ CHAR, size INT) INT :=
{
   .
   .
   .
   };

{
   .
   .
   .
   b := getArray(&array[0], length);
   };
```

The function arguments &array[0] and length correspond to the formal parameter p and size. The type of the result is INT, as declared in the function type.

ASSIGNMENT OPERATORS (operator =)

Description

There are ten assignment operators in addition to the simple assignment operator, described earlier in this chapter (see pages called Simple Assignment Operators).

These other assignment operators can be thought of as the combination of other DASL operators which require two operands and the simple assignment operator. These assignment operators assign to the left operand the value of the processing indicated by the operator, using the left and right operands.

The left operand is evaluated only one time.

The assignment operators

The following table lists the ten assignment operators and explains what they assign to the left operand.

THIS operator...	ASSIGNS to the left operand the result of...
*=	multiplying the operands.
/=	dividing the left operand by the right operand.
%=	computing the remainder after dividing the left operand by the right operand.

Continued on the next page

The assignment operators, (continued)

THIS operator...	ASSIGNS to the left operand the result of...
+=	adding the operands.
-=	subtracting the right operand from the left operand.
<<=	shifting left the bits of the left operand by the value of the right operand.
>>=	shifting right the bits of the left operand by the value of the right operand.
&&=	making a bitwise AND comparison between the left and right operands.
||=	making a bitwise inclusive OR comparison between the left and right operands.
!!=	making a bitwise exclusive OR comparison between the left and right operands.

Semantic information

The assignment operators require two operands, which both must be scalars, with the following exceptions.

Exceptions: For the += and the −= assignment operators, the left operand may be a pointer (see pages called Using Arithmetic Operators With Pointers).

The result has the value and type of the left operand after assignment. The result is *not* an lvalue.

The assignment operators group right to left.

Examples

```
RESULT *= factor;/* assigns the product of RESULT */
                 /* * factor to RESULT           */

flag ||= b;      /* assigns to flag the result of */
                 /* performing a bitwise inclusive*/
                 /* OR comparison of flag and b to*/
                 /* flag                          */
```

THE PARENTHESES AND COMMA OPERATORS (( ) ,)

Description of the parentheses operator

The parentheses operator (( )), may be used to modify precedence or grouping. Operators within parentheses are always performed first. If an expression within parentheses involves more than one operator, the operators are performed in order of precedence.

Otherwise, parentheses do not change the meaning of an expression. See pages called Operator Precedence.

Syntax for the parentheses operator

expression —> (expression—list)

The expression-list is made up of one or more expressions.

Example:

```
num := (seed + k1) * k2;
```

The addition operator (+) has a lower precedence than the multiplication operator (*), and would normally (if there were no parentheses) be performed after the multiplication of k1 by k2. However, the parentheses operator causes seed + k1 to be performed first, then its result is multiplied by k2.

Continued on next page

Description of the comma operator

The comma operator is used when the expression−list within parentheses contains more than one expression.

Processing of comma operator

The comma operator causes the following processing to occur:

- the expression preceding each comma is evaluated in left-to-right order,
- the values are discarded, and
- the value of the last expression is used, if necessary, in any subsequent expression outside the parentheses parentheses

Continued on next page

Syntax for the comma operator

expression —> (expression—list)

The expression—list can be made up of one or more expression. Each expression must be separated by a comma.

Example:

```
counter := (total := 1,f());
```

Semantic information for the comma operator

The last operator of each operand preceding a comma has a side effect. The operators that have a side effect are:

- function call operator (( )),
- increment and decrement operators (++ −−),
- assignment operators (:= operator=), or
- comma (,).

The result of a comma expression is the same as the last operand.

Continued on next page

Comma operator useful with macros

The comma operator is useful in macros which return expressions. See the chapter called Macros for more information and examples.

OPERATOR PRECEDENCE

Introduction

There is a pre-established order in which DASL operators are performed.

The order in which operators are performed can affect the result obtained from a statement.

Example of the importance of precedence

The following example demonstrates the importance of operator precedence.

```
d := a + b * c;
```

The multiplication operator (*) has a higher precedence than the addition operator (+), which in turn has a higher precedence than the assignment operator (:=). Thus, the product of b * c is added to a, and the result is assigned to d.

Because of the pre-established order of precedence, this statement would *not* be equivalent to the sum of a + b multiplied by c.

Continued on next page

Operator precedence chart

The following chart indicates the operator precedence when more than one operator is used in a statement.

The operators are listed in order of precedence. Those listed first have the highest precedence and are performed first.

Operators listed on the same line have the same precedence. The column labeled Grouping indicates the order in which operators of the same precedence are performed.

PRECEDENCE	OPERATORS	GROUPING
1	^ [ ] . ( ) *and* ++ −− (when used *after* the operand)	left to right
2	& − ∿∿ ∿ SIZEOF <type> *and* ++ −− (when used *in front of* the operand)	right to left

Continued on next page

Operator precedence chart, (continued)

PRECEDENCE	OPERATORS	GROUPING
3	* / %	left
4	+ −	
5	<< >>	to
6	&&	
7	|| !!	right
8	= ∿= < > <= >=	
9	&	
10	|	
11	?:	right to
12	:= operator=	left
13	,	left to right

Continued on next page

Parentheses

Parentheses may be used to modify precedence or grouping. Operators within parentheses are performed first, regardless of the pre-established order of precedence.

Example illustrating order of precedence

```
a := b > (c - e) / d++;
```

The processing in this statement occurs in the following order:

- c − e (parentheses have highest precedence) and d++ (the ++ operator has the highest precedence of operators used in this statement) are performed first;
- the value of c − e is divided by (/) the value of d++ (the / operator has the next highest precedence);
- b is compared to the value of the above division to determine if b is greater than (>) that value (the > operator has the next highest precedence);
- the value of the above comparison is assigned (:=) to a (the := operator has the lowest precedence of the operators in this statement).

Continued on next page

Order of evaluation of operands

DASL does not specify in what order the operands of an operator are evaluated, except in the cases of the comma, conditional, and logical operators.

In most cases, the order of evaluation does not make any difference in the result. However, it can make a difference in cases where the evaluation of one operand has side effects which affect the other operand.

Example:

```
In the statement a := b + b++;
```

either b or b++ could be evaluated first. In this case, the value that a assumes depends on the particular order of evaluation.

CHAPTER 9.
STATEMENTS

Contents

OVERVIEW

Description

Statements specify the actions which a function must take.

Statements have effects only; they do not have values.

Definitions of effects and values

An *effect* is a change that occurs as the result of processing part of a DASL program.

Examples of effects include changing what one sees on the screen or assigning a value to an lvalue.

A *value*, on the other hand, is a numeric quantity.

Components of a statement

A statement is composed of expressions and flow of control constructs (i.e., IF...THEN...ELSE, LOOP...WHILE, CASE, and function call).

Continued on next page

Expressions as components of a statement

Expressions are used to compute values to be used in a statement.

Each expression making up a statement has a value, but the statement as a whole has no value, only an effect. The values associated with the component expressions are not stored in memory.

Example of statement/expression relationship

The statemment numVar := (x+2); is made up of the following expressions:

- numVar
- x
- 2
- (x+2), and
- numVar := (x+2)

Because numVar := (x+2); is a statement, the values of the expressions that make up the statement 'disappear' after they have been evaluated. The effect of the statement remains; that is, the value of the expression (x+2) is assigned to the expression numVar.

Continued on next page

Coming up

The rest of this chapter discusses

- expression statements,
- compound statements,
- IF...THEN statements,
- CASE statements,
- LOOP WHILE statements,
- labeled and GOTO statements, and
- null statements.

EXPRESSION STATEMENTS

Description

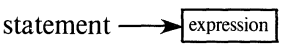
An expression used with a semicolon (;) is a statement. The expression must be one whose last operator performed has a side effect.

The operators that have a side effect are

- function call,
- increment and decrement,
- assignment, and
- comma.

Syntax notation

The following is the syntax notation for an expression statement.



Example:

```
sum := 0;
b++;
random();
```

Continued on next page

Function call

You can use an expression whose last operator performed is the function call operator as a statement (by adding a semicolon at the end).

If the function called returns a value, that value is ignored unless it is assigned to a variable. Only the effect of the called function remains.

Continued on next page

Example of a function call expression

The following example illustrates the use of a function call expression as a statement.

```
ascending (p1, p2 ^ INT) BOOLEAN :=
{
    RESULT := true;
    p1 += 1;
    };

ENTRY MAIN () :=
VAR dateArray [10] INT;
    counter INT;

{
   ascending(&dateArray [0], & dateArray [9]);
   .
   .
   };
```

The function call expression ascending(&dateArray[0], &dateArray[9], is used as a statement, since it has

- an operator (the function call operator) which has a side effect and which is the last operator in this expression to be performed, and

- a semicolon at the end.

Although the function called (ascending) returns a value (RESULT), the function call statement results only in the side effects of the function, not the value of RESULT.

COMPOUND STATEMENTS AND BLOCKS

Description of compound statement

A compound statement is a series of statements treated as if they were one statement.

Description of local block

A local block is like a compound statement, but it starts with declarations.

Only EXTERN function declarations are permitted in a local block.

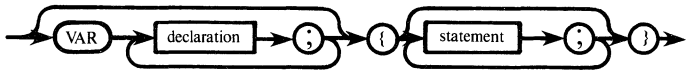
Local variables are initialized when the program is loaded, *not* when the block is entered.

Continued on next page

Syntax notation

The following is the syntax notation for a compound statement and local block.

Compound Statement



Example 1:

The following is an example of a local block.

```
VAR m INT;
    p1^, p2^ INT;
{
   m := p1^;
   p1^ := p2^;
   p2^ := m;
   };
```

Example 2:

```
LOOP { WHILE dCount < nDates;
   getDates (&dateArray[dCount++]);
   };
```

NOTES:

- The braces indicate the beginning of the compound statement. Note that each statement making up the compound statement ends with a semicolon.

- Note the use of the braces at the end of the compound statement.

Continued on next page

Compound statements with flow of control constructs

Frequently, the flow of control constructs IF...THEN, IF...THEN...ELSE, and LOOP WHILE use compound statements. This permits a number of actions to be performed depending on one condition.

Example: The following example shows the structure of a compound statement used with the IF...THEN...ELSE construct.

```
IF expression THEN{
   statement1;
   statement2;
   }
 ELSE statement3;
```

This whole example represents one compound statement, made up of

- a compound statement, consisting of statement1 and statement2 (notice the use of braces to indicate these statements are part of a compound statement), and

- statement3.

IF...THEN AND IF...THEN...ELSE STATEMENTS

Description of IF...THEN statement

IF...THEN statements establish a true/false condition and the statement(s) to be performed if the condition is true (non-zero). IF...THEN statements create a two-way branch.

Description of IF...THEN...ELSE statement

The use of the ELSE statement after the IF...THEN... statement contains statements to be performed if the condition is false (zero).

Syntax

The following is the syntax notation for the IF...THEN...ELSE statement.

Example:

```
IF side 1 > maxLength THEN
   errorMessage() /* note there is no semicolon    */
 ELSE                    /* preceding an ELSE statement*/
   area := side1 * side2;
```

Continued on next page

Semantic information

The expression following the IF must be a scalar or pointer expression.

The statement following the THEN and ELSE may be any of the types of statements discussed in this chapter.

Recommended formats

Indentation is important to indicate to the reader the structure of IF...THEN...ELSE statements. The examples below are recommend structured coding guidelines; indentation does *not* affect the execution of these statements.

- Format if whole IF...THEN statement fits on one line:

```
IF expression THEN statement1;
```

- Format if whole IF...THEN ELSE statement fits on one line:

```
IF expression THEN statement1 ELSE statement2;
```

Continued on next page

Recommended formats, (continued)

- Format if the statement following THEN wouldn't fit on the same line as THEN:

```
IF expression THEN
   statement1
  ELSE statement2;
```

- Format for more than one statement following THEN:

```
IF expression THEN {
   statement1;
   statement2;
   }
  ELSE statement3;
```

- Format for more than one statement after ELSE:

```
IF expression THEN statement1;
 ELSE {
   statement2;
   statement3;
   };
```

Continued on next page

Braces

To make clear with which THEN an ELSE statement is associated, use braces.

Example:

The following statement:

```
IF sum >0 THEN
   IF seed >0 THEN root := seed
     ELSE root := sum;
```

could be made more clear by using braces:

```
IF sum >0 THEN {                    /* Use brace to */
   IF seed >0 THEN root := seed/* mark the be- */
    ELSE root := sum;           /* ginning of   */
   };                           /* main THEN    */
                                /* statement.   */
                                /* Use brace to */
                                /* mark the end */
                                /* of main      */
                                /* statement.   */
```

Continued on next page

Flow of control

The IF...THEN...ELSE statements determine a specific flow of control, depending on whether the condition is true or false.

Example 1:

The following example illustrates the flow of control in a simple IF...THEN statement.

```
IF date <= 30 THEN  date++;
printDate(date);  /* this is the next statement   */
                  /* after the IF...THEN statement*/
```

IF date is...	THEN the statement(s)...
less than or equal to 30	date++; and printDate(date); are performed.
greater then 30	printDate(date); is performed.

Continued on next page

Flow of control, (continued)

Example 2:

The following example illustrates the flow of control for an IF...THEN...ELSE statement.

```
IF date <= 30 THEN  date++
 ELSE
   errorMessage();
printDate(date);     /* this is the next statement */
                     /* after the IF...THEN...ELSE */
                     /* statement                  */
```

IF date is...	THEN only these statements are performed...
less than or equal to 30	date++ and printDate(date)
greater than 30	errorMessage( ) and printDate(date).

Continued on next page

Use of multiple conditions

IF...THEN statements can include other IF...THEN statements.

Each THEN and ELSE is associated with the nearest preceding IF.

Recommendation: Combine multiple conditions, such as

```
IF a = 2 THEN
 IF status = 1 THEN
   printChar();
```

by using the binary operator &, as below

```
IF (a = 2) & (status = 1) THEN
   printChar();
```

Continued on next page

Example using IF...THEN statements

This function compares two dates to determine whether the first date is smaller than the second date.

Example:

```
ascending (p1, p2 ^ date) BOOLEAN := /* date is a   */
{                                    /* structure */
  RESULT := 1;                       /* made up of */
  IF p1^.year > p2^.year THEN        /* year, and  */
     RESULT := 0                     /* day        */
   ELSE {
     IF p1^.year = p2^.year THEN {
        IF p1^.month > p2^.month THEN
           RESULT := 0
         ELSE {
           IF p1^.month = p2.month THEN {
              IF p1^.day > p2^.day THEN
                 RESULT := 0;
              };
           };
        };
     };
  };
```

Continued on next page

Example using IF...THEN statements, (continued)

Explanation:

The RESULT is initialized as 1 (true); if *none* of the following conditions are true, the RESULT will remain true.

If the year of the first date is greater than the year of the second, the RESULT is assigned a value of 0 (false) and no further processing occurs in this function, since there are no statements after the whole compound IF...THEN statement.

If the year of the first date is not greater, then the next condition is tested (IF p1^.year = p2^.year). If the years are equal, then the months are compared, using the same approach as used to compare the year part of the dates. Similarly, if the months turn out to be equal, then the days are compared.

If the year of the first date is smaller than the year of the second, then no further testing is necessary, and the RESULT remains 1. Similar logic is used for comparing months and comparing days.

CASE STATEMENT

Description

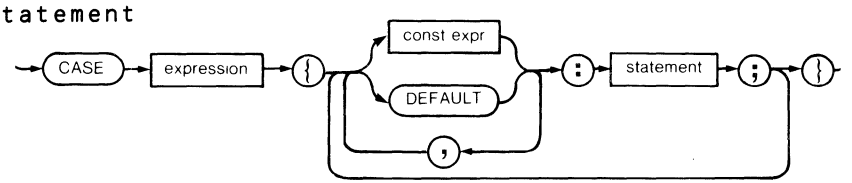
The CASE statement causes control to be transferred to one of several statements, depending on the value of an expression.

Thus, the CASE statement is a multi-branch flow of control statement that allows different actions to be taken, depending on the run-time value of an expression.

Continued on next page

Syntax notation

The following is the syntax notation for the CASE statement.



The DEFAULT statement is optional. Each CASE statement may only have one DEFAULT statement.

Example:

```
CASE in {
    '1':      x++;
    '2','3': x--;
    DEFAULT: x:=0;
    };
```

Continued on next page

Semantic information

The expression that is compared to the constant expression must be a scalar. The constant expressions must be compile-time constants.

The statement following each constant expression may be any of the types of statements discussed in this chapter.

Recommended format

Indentation is important to indicate to the reader the structure of a CASE statement. The example below is a recommended structured coding guideline; indentation does *not* affect the execution of CASE statements.

```
CASE expression {
    constant expression : statement1;
    constant expression : statement2;
    DEFAULT             : statement3;
    };
```

Continued on next page

Flow of control

When the CASE statement is executed, the following flow of control occurs:

IF the run-time value of the expression...	THEN control is transferred to...
matches the value of of the constant expressions	the statement which follows the matching constant expression.
does not match any constant expression	statement which follows DEFAULT or, if there is *no* DEFAULT, control is transferred to the next statement after the CASE statement.

After CASE statement execution

After execution of any one statement in the CASE statement, control is tranferred to the first statement *after* the CASE statement.

Continued on next page

Caution

If possible, avoid using large ranges with the case statement, as the ranges require considerable memory.

If you do *not* use a DEFAULT, be sure that the constant expressions account for all possible values of the CASE statement expression.

Continued on next page

Example using the CASE statement

The following example shows a CASE statement that causes different functions to be performed, depending on the character keyed in.

```
{
    in := character keyed in at terminal
  CASE in {
      '1':          x++;
      '2','3':      x--;
      DEFAULT:      x:=0;
      };
   p :=0;
   };
```

This CASE statement takes the value keyed in and compares it to the constant expressions:

- if 1 is keyed in, then x is incremented, or
- if 2 or 3 is keyed in, then x is decremented, or
- if anything other than 1, 2, or 3 is keyed in, x is assigned the value 0.

The next statement performed is p :=0;.

LOOP WHILE STATEMENTS

Description

The LOOP statement causes the statement which follows it to be repeated.

The WHILE statement contains the condition that stops the performance of the LOOP statement. If the value of the expression following WHILE is zero, then the LOOP processing stops.

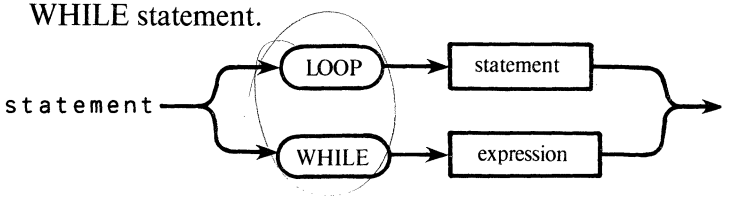
Use of LOOP statement alone

You can use the LOOP statement without the WHILE statement. However, this causes the loop to be repeated infinitely. This would only be used for an application you want to run indefinitely.

Continued on next page

Syntax notation

The following is the syntax notation for the LOOP WHILE statement.



NOTES:

- The LOOP statement can stand alone (with no WHILE expression)

- The WHILE expression must follow LOOP directly, or be part of a compound statement immediately following LOOP.

Example:

```
LOOP {
   read (x,y,z);
 WHILE ~eof;
   };
```

Continued on next page

Semantic information

The statement following LOOP may be any of the types of statements discussed in this chapter.

Recommended formats

Indentation is important to indicate to the reader the structure of a LOOP WHILE statement. The examples below are recommended structured coding guidelines; indentation does *not* affect the execution of LOOP WHILE statements.

```
LOOP {              /* format if there is not state-*/
 WHILE expression; /* ment between LOOP and WHILE  */
   statement1;
   statement2;
   };

LOOP {              /* format if there are statement*/
   statement1;      /* between Loop and WHILE       */
   statement2;
 WHILE expression;
   };
```

Continued on next page

Recommended format for a LOOP WHILE/IF...THEN statement

The following example shows the indentation format for a LOOP WHILE statement that includes an IF...THEN statement and another LOOP WHILE statement.

```
LOOP {
 WHILE expression;
   IF expression2 THEN {
      statement1;
      LOOP {
         statement2;
       WHILE expression3;
         statement3;
         };
      };
   statement4;
   };
```

Caution

Be careful to avoid the following examples of possible problems when using LOOP WHILE statements.

Continued on next page

Caution, (continued)

Example:

```
a := 1;              /* This LOOP WHILE statement tests  */
i := 0;              /* the condition (WHILE i++< <10;)-  */
LOOP{                /* if it is true, then it performs  */
 WHILE i++<10;       /* the next statement, then tests   */
   a := a*i;         /* the condition again, etc.        */
   };
a := 1;              /* The LOOP WHILE statement per-    */
i := 1;              /* forms the statement a := a*i;    */
LOOP {               /* then tests the condition         */
   a := a*i;         /* i++<10; - if it is true, the     */
 WHILE i++<10;       /* loop is performed again, and     */
   };                /* so on.                           */
```

- This is an example of placing the WHILE statement incorrectly; the boolean expression following the WHILE is evaluated at the time it is executed.

Example:

```
LOOP                 /* There are no statements to be  */
 WHILE j < 10;       /* performed other than the test  */
   j++;              /* of the condition. The state-    */
                     /* ment j++ is not considered part*/
                     /* of LOOP WHILE; braces after    */
                     /* LOOP and after j++ would avoid */
                     /* the null loop.                 */
```

- This is an example of omitting braces, which can cause a null loop, which as no statements to be performed.

Continued on next page

Example using LOOP WHILE statements

The following example illustrates the use of the LOOP WHILE statement that includes another LOOP WHILE statement as one of its substatements.

```
r := 0;
LOOP {
 WHILE r <= row;
   c := 0;
   LOOP {
    WHILE c <= col;
      displayChar ();
      c++;
      };
   r++;
   };
```

If r is less than or equal to row, then the condition c <= col is tested.

If c is less than or equal to col, then the next two statements are performed (the function displayChar is called, and c is incremented by 1).

The last statement, r++; is associated with the main LOOP WHILE statement. Thus, r++ is performed as long as r <= row, regardless of whether c <= col; is true or false.

The main loop is then repeated with the statement r <= row; being evaluated with the new value of r (which is one greater than the first time through the loop).

LABELED AND GOTO STATEMENTS

Description of labeled statements

Any statement may be identified by a label.

GOTO statements branch to another statement in the same function by referring to the label of that statement.

Syntax notation

The following is the syntax notation for a labeled statement.

The colon (:) declares the identifier as a label.

The label *cannot* begin with an integer.

Example:

```
statement1 : next := &counter;
```

Semantic information for labeled statements

The scope of the label is the function in which it appears.

Continued on next page

Description of the GOTO statement

The GOTO statement is an unconditional branch, transferring control to a labeled statement.

Caution

Generally, DASL programs should *not* include GOTO statements. Structured programming is composed of properly nested segments that are entered only at the top and exited from the bottom. GOTO statements do *not* follow that pattern.

IF...THEN...ELSE and LOOP WHILE statements can be used instead of GOTO statements.

Use GOTO statements if there is no clearer method for exiting with an error or forward branching out of a complex nested structure. Be careful that you do not bypass statements you want to be executed.

Continued on next page

Syntax notation

The following is the syntax notation for the GOTO statement.

The identifier must be a label on a statement in the same function.

Example:

```
IF index >maxNum THEN
   GOTO statement3; /* statement3 must be in the   */
   .                /* same function as this       */
   .                /* IF...THEN statement         */
   .
statement3 : error();
```

Continued on next page

Example using the GOTO statement

The following example illustrates the use of the GOTO statement to exit from a loop with an error.

```
b:=0;
LOOP {
 WHILE b < size;
   b++;
   IF key + list[b] THEN
       GOTO errorStatement;
   };
errorStatement : printError();

In the statement

    IF key + list[b] THEN GOTO errorStatement;
```

The GOTO statement exits the loop with an error message. The GOTO statement branches to the statement (in the same function) labeled errorStatement.

The statement labeled errorStatement is printError( );, which is a function call to the function which displays an error message.

NULL STATEMENT

Description

A null statement causes no special action, but it may be used wherever any other statement may appear.

Syntax notation

The following is the syntax notation for the null statement.

statement ⟶

The null statement includes *no* expression or operators.

Example:

```
b := 0;
;          /* a null statement*/
```

Continued on next page

Possible uses of the null statement

Two examples of uses of the null statement are using it with

- the GOTO statement, to exit a function without performing any action, and
- the CASE statement, if *no* action is to be performed for a specific condition.

Example 1:

```
IF counter=50 then GOTO endStatement;
.
.
.
endStatement: ; /* This last statement in  */
              /* the function is a null  */
              /* statement. Thus         */
              /* GOTO passes control to  */
              /* the end of the function */
              /* without performing      */
              /* any action.             */
```

Example 2:

```
CASE in {
    '1'     : x++;
    '2'     : ;  /* If in is '2' then no   */
                 /* action is taken.       */
    DEFAULT: x--;
    };
```

CHAPTER 10.
DASL MACROS

Contents

OVERVIEW

Description

The DASL macros provide text substitution capabilities. In the simplest case, an identifier may be replaced by any desired string of characters.

Macros are independent of the DASL syntax rules.

Advantages of macros

Some of the advantages of macros are:

- constants can be named, increasing program clarity;
- macros can have parameters which provide the equivalent of a function call, except that the text substitution is done at compile time...not at run time;
- macros allow extension of the language.

Continued on next page

DASL macros

There are five predefined DASL macros:

- DEFINE — defines a new macro
- IFELSE — compares two text strings
- INCR — increments a numeric string
- SUBSTR — selects a substring
- INCLUDE — includes additional files in a source program.

In addition to the predefined macros, you can create your own macros or use the macros already defined in an INCLUDE file.

Continued on next page

Coming up

This chapter discusses the following topics:

- macro call formats
- the five predefined macros
- recursive macros
- evaluation suppression
- commas and parentheses.

MACRO CALL FORMATS

Macro syntax

A macro can be called by one of the following sequences:

```
identifier

or

identifier (parameter, parameter . . .)
```

NOTE: You can have up to nine parameters.

Definition of identifier

An *identifier* either names a predefined macro IFELSE, SUBSTR, INCLUDE, INCR, DEFINE or names macros defined by the DEFINE macro.

Continued on next page

Parameter rules

The following rules apply to macros called with parameters.

- Blanks are significant inside parameters.
- Line ends are treated as characters.
- If the parameter contains commas as part of a character sequence, you must put evaluation supression brackets #[#] around each parameter. Otherwise, the commas will be treated as parameter separators.
- Unspecified parameters are considered to be null strings.
- Extra parameters are ignored.

Continued on next page

Macro call process

The macro call and text substitution process includes the following stages:

STAGE	DESCRIPTION
1	DASL compiler sends macro call to the macro processor.
2	If the macro contains parameters, each parameter is scanned for any macros inside the parameter. If there is a macro, that call is performed.
3	The parameters are substituted into the macro definition.
4	The adjusted macro definition replaces the macro call. The result is rescanned for more macro calls. If there are no macro calls, the result is sent to the compiler scanner and parser.

Continued on next page

how does #E affect this procedure
or #macro name ,

Macro expansion

Macro expansion is depth first as opposed to breadth first.

THE INCLUDE MACRO

Description

During compile time, the INCLUDE macro temporarily changes the compiler's current input file. This macro call is useful for packaging logically related sets of macros, declarations, etc.

Format

The format for the INCLUDE macro is

INCLUDE(filename)

Example:

```
INCLUDE (D$INC)
INCLUDE (D$RMS)
```

NOTE: INCLUDE macros may be nested.

THE DEFINE MACRO

Introduction

The DEFINE macro creates new macros. A new macro is created by the macro call

DEFINE(identifier, definition)

NOTE: You can have up to nine parameters in the definition.

Description

The DEFINE macro has the effect of defining the identifier as the name by which the macro is called. The definition is the string to be substituted for the macro.

Example:

```
DEFINE(x,z)    /* replace x with z */
```

Text substitution

Text substitution occurs inside comments, quoted string constants, and the #[#] symbols.

Use of the evaluation suppression symbols around the definition will prevent the substitution until the macro being defined is called.

Continued on next page

Scope of DEFINE macro

The scope of the definition of an identifier is normally local to the current block. If the identifier is already defined as a macro, that definition (even if outside the current block) will be replaced by a new definition.

Use the #[#] brackets to surround the identifier to prevent the definition substitution.

Example: If you wanted months to be defined as two different numbers in your program, you would have to do the following:

DEFINE(month,3) in the first part of your program.

When you wanted to redefine month later, you would use the #[#] brackets.

DEFINE(#[month#],5).

If you didn't use the brackets, every time month was called, it would be replaced with 3.

DEFINE(3,5)

THE IFELSE MACRO

Description

The IFELSE macro call compares two character strings and performs either the third or fourth depending on the comparison.

IF the first two strings are...	THEN...
equal	result is the third string.
unequal	result is the fourth string.

Format

The format for the IFELSE macro is as follows:

IFELSE(string1, string2, equal, unequal)

Continued on next page

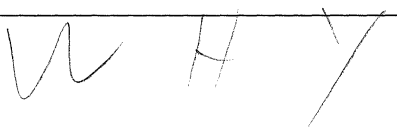
Examples

The following table illustrates the possible results of string comparisons using the IFELSE macro.

STRING COMPARISON	RESULT	REASON
IFELSE(1,2,3,4)	4 (unequal)	The first 2 strings are not equal.
IFELSE(1,1,3,4)	3 (equal)	The first 2 strings are equal.
IFELSE(1,,3,4)	4 (unequal)	The first string and a null string are not equal.
IFELSE(,,3,4)	3 (equal)	The first 2 strings are null. They are therefore, assumed to be equal.

Uses of the #[#] symbols

If the third or fourth strings contain macro calls, it may be necessary to use the #[#] symbols to ensure that the calls are made in the proper order.



THE INCR MACRO

Description

The INCR macro is used for numeric incrementation.

Format

The format for an INCR call is

INCR(number)

NOTE: Leading blanks are allowed.

Parameter values

The parameter may be:

- an unsigned decimal
- an octal number
- a hexadecimal number
- a string constant.

If the value is a string constant, the value is considered to be zero.

Continued on next page

Result of an INCR

The result of an INCR macro call is a string which is the decimal number following the value of the parameter.

Example:

```
DEFINE(a,4)

DEFINE(b,INCR(a))

DEFINE(c,INCR(b))
```

THE SUBSTR MACRO

Description

The SUBSTR macro selects the portion of the string parameter specified by the start and length parameters. The result is the number of characters specified by the length parameters, starting as the position specified from the start position.

Example:

```
SUBSTR(ABC,1,2) /* starting at the first  */
                /* go over 2              */
```

Format

The SUBSTR macro takes three parameters:

SUBSTR(string, start, length)

Example: `SUBSTR(ABCD,1,2)`

Continued on next page

SUBSTR rules

The following rules apply to the SUBSTR macro call:

- The start and length numbers must be an unsigned decimal, octal, or hexadecimal.

- The start position begins at position zero.

Example: SUBSTR(ABC,0,2) is replaced by AB

- If there is no length parameter, the result is the rest of the string from the start position.

Example: SUBSTR(ABC,0,2) is replaced by BC

- If the string is shorter than specified by the start and length parameters, then the result is the string from tl start position until the end of the string.

Example: SUBSTR(ABC,2,2) is replaced by C

Use of SUBSTR

A SUBSTR can be used to examine a parameter character by character. This is helpful when special processing is required for different characters such as blanks.

RECURSIVE MACROS

Five types

There are five recursive macros that are not predefined to the compiler but are defined in the INCLUDE file, D$INC.

These macros are

- ENUM and ENUMV
- SET, SETV, and SETW.

Continued on next page

Description and format

The description and format for each macro follows:

- ENUMV—defines up to eight arguments as ascending values from the first argument.

```
DEFINE(ENUMV,
       #[IFELSE(#2,,,#[DEFINE(#2,#1)
                       ENUMV(INCR(#1),#3,#4,
                       #5,#6,#7,#8,#9)#])#])
```

- ENUM—defines up to nine arguments as ascending values zero to eight, and declares a preceeding variable name as type BYTE.

```
DEFINE(ENUM,
       #[DEFINE(#1,Ø)
         ENUMV(1,#2,#3,#4,#5,#6,#7,#8,#9)#]BYTE)
```

- SETV—defines up to eight arguments in ascending values from the first argument in successive powers of 2.


```
DEFINE(SETV,
       #[IFELSE(#2,,,#[DEFINE)#2,(#1))
                       SETV(#1<<1,#2,#3,#4,
                       #5,#6,#7,#8,#9)#])#])
```

- SET—defines up to eight arguments in ascending values in successive powers of 2.

```
DEFINE(SET,#[SETV(1,#1,#2,#3,#4,#5,#6,#7,
       #8)#]BYTE)
```

- SETW—Same as SET except that the type is unsigned and it takes up to nine arguments.

```
DEFINE(SETW,#[DEFINE(#1,1)SETV(2,#2,#3,#4,#5
                        #6,#7,#8,#9)#]
                        UNSIGNED)
```

Continued on next page

Possible uses for ENUM/ENUMV

The ENUM and ENUMV macros are useful for defining a type and instances of that type.

Example: You want to define Color as a type and RED, GREEN and BLUE as instances of that type.

You could express this as:

```
TYPDEF Color BYTE;

DEFINE(red,0)
DEFINE(green,1)
DEFINE(blue,2)
```

or, you could use the ENUM or ENUMV macros and express it as

```
TYPEDEF Color ENUM(red,green,blue);
```

or,

```
TYPEDEF Color BYTE;
ENUMV(0,red,green,blue);
```

Continued on next page

Example of SET and SETV use

One example of SET and SETV use is from the DASL compiler. These macros are how it keeps track of expressions.

```
TYPDEF Flags SET(variable,lvalue,constant)
```

If none of these bits are on, it is the result of an expression.

EVALUATION SUPRESSION SYMBOLS #[#]

Evaluation supression description

The evaluation supression symbols #[and #] are an important part of the macro call process. These symbols indicate to the compiler that the characters between these symbols are to be treated as ordinary characters and to supress the macro call at that time.

Example of evaluation supression

If you wanted to count how many times a macro is called, you would use the brackets as follows:

```
DEFINE (#[count#],INCR(count))
```

COMMAS AND PARENTHESES IN MACROS

Use of comma operator

The comma operator is used to separate a list of expressions in macros and as a delineator in parameters.

How the comma operator works in an expression list

The comma operator causes the preceding expression to be evaluated. However, only the result or side effect of the last expression is retained. The result of a comma expression is the same as a right operand.

Last expression operators

The last expression must contain one of the following operators:

- function call ()
- increment ++
- decrement --
- assignment := or
- comma ,

Continued on next page

Comma operator in parameters

If the definition contains parameters, the parameters will be completed in the same order as they appear in the macro call.

Each comma causes the preceding parameter to be evaluated.

Example:

```
DEFINE(INIT,#1:=#3:=#2)

      .
      .
      .
      INIT(A,B,C)
```

Use of parentheses with expression lists

Parentheses are useful in a list of expressions separated by commas because they force proper evaluation order.

Example:

```
DEFINE(getC,(lp^+$LEOR ? (getline(),lp++^) : lp++^))

    .
    .
    .
    c := getC();
```

Continued on next page

Parentheses and macro definitions

Parentheses also help 'insulate' macro definitions. If you don't use them, you may get incorrect results.

Example:

BASIS . WITH PARENTHESES . WITHOUT PARENTHESES		
format	DEFINE(mult,((#1)*(#2))	DEFINE(mult,#1*#2)
Statement	. . . answer INT := mult(2+2,2) . . .	
result	answer gets 8	answer gets 6

102706928