UNISYS

ALLY® Software Development Environment

ADL User's Guide

Priced Item

April 1987
Printed in U.S.A.
UP–12507

NO WARRANTIES OF ANY NATURE ARE EXTENDED BY THE DOCUMENT. Any product and related material disclosed herein are only furnished pursuant and subject to the terms and conditions of a duly executed Program Product License or Agreement to purchase or lease equipment. The only warranties made by Unisys, if any, with respect to the products described in this document are set forth in such License or Agreement. Unisys cannot accept any financial or other responsibility that may be the result of your use of the information in this document or software material, including direct, indirect, special or consequential damages.

You should be very careful to ensure that the use of this information and/or software material complies with the laws, rules, and regulations of the jurisdictions with respect to which it is used.

The information contained herein is subject to change without notice. Revisions may be issued to advise of such changes and/or additions.

Preface

The *ADL User's Guide* describes the ALLY Development Language (ADL) and how you use it in building applications with the ALLY Application Developer's Dialog.

This manual is for people who are developing ALLY applications. We assume that you have read *Introduction to ALLY, Concepts and Facilities*, and *Introduction to the Dialog*.

The manual has five chapters and three appendixes.

Chapter/ Appendix	
1	Introduces the ALLY Development Language (ADL) and provides examples of ADL procedures
2	Describes the four sections that can make up an ADL procedure
3	Describes ADL's constructs and operators
4	Describes ADL's built-in instructions and functions
5	Describes ADL's Generic Data Manipulation Language (DML)
A	Contains lists of ADL's reserved words and the syntax of ADL reserved words
B	Contains the tables of ADL operators and of precedence of operators
C	Contains tables of data type conversion and arithmetic, DATE format pictures, and DATE picture precisions for rounding and truncating dates

Conventions

You should read carefully the description of documentation conventions before reading this manual.

We use the following conventions in this manual:

Single quotes (' ')	Identify command names.
Boldface type (**bold**)	Highlights text you are to enter. Boldface is also used within command syntax statements.
Double quotes (" ")	Identify text strings within text sections. These strings are typically located in examples or as part of the prompts that ALLY sends to your display. Sometimes the exact content of a text string is affected by the traditional rules of punctuation. In these cases, we place the closing quotation mark at the end of the text string. For example, instead of: You see the prompt "Macro number:." We say: You see the prompt "Macro number:".

End of Preface

Contents

Chapter 1. Introduction to ADL

Chapter 2. Sections of an ADL Procedure

Chapter 3. ADL Constructs

Chapter 4. ADL Functions and Instructions

Chapter 5. Generic DML

Appendix A. ADL Reserved Words

Appendix B. ADL Operators

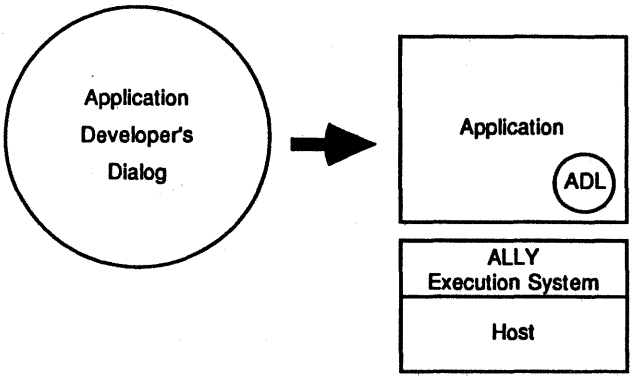
Appendix C. Data Types and DATE Pictures

Figures

Tables

Chapter 1
Introduction to ADL

The ALLY Development Language (ADL) is a simple yet powerful programming language. ADL provides special built-in functions and instructions that you can combine with the other elements of ALLY to accomplish special goals in an application.



F002-0572-00

Figure 1-1. ADL's Relationship to ALLY

You include ADL procedures in your applications by writing and compiling them through the "Procedural Languages" branch of the Dialog. The ADL compiler runs quickly and provides descriptive error messages.

The syntax of ADL is similar to PASCAL. However, a complete ADL procedure can consist of only one procedure statement. Many procedures that you write will consist of only a few lines. For example, the following procedure performs a query and prints

a report for the appropriate record. The text within the braces ({ }) is a comment.

```
BEGIN
   IF (FORM_PURGER.REC_TYPE := 'Z')
   THEN
      BEGIN
            CALL_CMD (QUERY);          {perform the query}
            CALL_CMD (PALL);           {print the report }
      END;
END;
```

You can invoke an ADL procedure from any event in an application. You can use ADL to build special field validation, arithmetic computations, task action and flow control, and database management interactions. For example, ADL procedures can:

- compute, compare, and validate form/report data
- access databases and files
- invoke any ALLY task, action, or command
- access global variables
- call other ADL procedures
- be used as tasks, entry points, or menu choices
- perform database operations such as:
  - posting
  - generating fixed sequential files that ALLY can use through Data Source Definitions

ADL provides built-in functions that manipulate and convert data in form/report fields and in ADL variables in other currently-executing ALLY actions. ADL allows you to:

- manage flow of control for tasks
- use ALLY commands
- display help and error messages
- report to a calling event the success or failure of a procedure

ADL also contains a group of special instructions called the Generic Data Manipulation Language (DML). These instructions allow you to perform database operations on all databases supported by ALLY.

The procedures you write will often consist of statements that verify a value entered into a field, or do some checking or calculations. These procedures can be invoked before or after:

- a query, commit, or rollback to a form/report
- an insert, update, or delete to a form/report group
- a field value is changed
- control passes to a:
  - form/report
  - form/report field
  - menu
  - menu choice

The syntax of a procedural statement in ADL is very similar to the corresponding statement in the PASCAL language in terms of execution control (IF ELSE), iteration (WHILE), and construction of compound statements (BEGIN END). PASCAL programmers should be aware that ADL does not support all of the features of PASCAL and that ADL uses semicolons differently.

Examples of ADL Procedures

Here are some examples of simple ADL procedures. This example is called before control passes to a form/report. This procedure sets a state that may be changed by another ADL procedure during the execution of the form/report.

```
VAR
    mode : NUMBER GLOBAL;

BEGIN
    mode := 1;
END;
```

The next example executes the ALLY query command if the value of a field is null ('').

```
    IF (report.name = ('')) THEN        {compare to null}
       CALL_CMD (QUERY);                {execute query  }
```

The last example changes the flow of control in a form/report. When the user invokes the 'next field' command, the first record is created for a new group.

```
VAR
    C     : NUMBER;

BEGIN
    C := GET_CMD ();
    IF (C = FNEXT) THEN          {next field command}
       CALL_CMD (FINSNEXT);      {do an insert in next group}
END;
```

Summary

ADL allows you to build special flow control, computations, validations, and database operations into your ALLY applications. The remainder of this manual provides a description and example of each section that can make up a procedure and of each built-in function, instruction, and operator.

End of Chapter 1

Chapter 2
Sections of an ADL Procedure

Table 2-1 shows the four sections that can make up an ADL procedure.

Table 2-1. The Sections of an ADL Procedure

Procedure declaration	Not required unless arguments are passed in
Constant declarations	Not required unless the procedure uses at least one constant
Variable declarations	Not required unless the procedure uses at least one variable
Procedure statements	Required in every ADL procedure

Although a procedure can have four sections, it can also consist of only one procedure statement. Figure 2-1 shows a valid procedure with one procedure statement.

```
employee.hiredate := personnel.date_employed;
```

Figure 2-1. ADL Procedure with One Statement

Figure 2-2 shows a procedure that contains all four sections.

```
PROCEDURE PERSONNEL_CALCS  (VAR count_females    : NUMBER;
                            VAR count_employees : NUMBER;);
CONST
    rate1 = 0.15;
    rate2 = 0.27;

VAR
    admin : CHAR;
    tech  : CHAR;

BEGIN

   IF (emp_form.salary < 29,000)
      THEN  payroll.withhold := emp_form.salary * rate1;
      ELSE  payroll.withhold := emp_form.salary * rate2;

   count_employees := count_employees + 1;
   IF (emp_form.female = 'X')
      THEN  count_females := count_females + 1;

   IF (emp_form.clerical = 'X')
      THEN admin := admin + 1;
      ELSE tech := tech + 1;
END;
```

Figure 2-2. ADL Procedure with Four Sections

Procedure Declarations

A procedure declaration allows you to name a procedure. It can also allow you to pass data between procedures. The syntax of each type of procedure declaration is:

PROCEDURE *name*

PROCEDURE *name* **(VAR** *parameter_name* **:** *data_type*;**);**

A procedure declaration begins with the reserved word PROCEDURE followed by the name of the procedure. Any parameters are declared within a set of parentheses followed by a semicolon. A parameter list requires at least one parameter, and it can contain multiple parameters. A parameter list is easier to read when you align multiple parameters on succeeding lines.

A procedure that can be called by another procedure must be named in a procedure statement, even though it passes no parameters. If the procedure statement simply names a procedure and there are no parameters, you omit the list, the parentheses, and the semicolon.

Procedure Parameters

An ADL procedure parameter can receive a value from a calling procedure argument and return a value to a calling procedure argument. A parameter name must be preceded by the reserved word VAR. Each argument of the procedure invocation must be a container for a value (a variable or a field). An argument cannot be a constant, because a constant cannot be manipulated.

The information passed from an argument to its corresponding parameter is not the value but the address of the argument. The procedure's manipulations of this parameter are made directly to the corresponding argument.

Figure 2-3 illustrates the communication between the fields "admin" and "employees" of the "summary" form and the parameters "count_admin" and "count_employees".

```
{procedure 1}
PROCEDURE PERSONNEL_CALCS (VAR count_admin       : NUMBER;)
                          VAR count_employees    : NUMBER;);
        (Remainder of procedure 1)

-------------------------------------------------------
{procedure 2}

        (Beginning of procedure 2)

CALL PERSONNEL_CALCS (summary.admin, summary.employees);

        (Remainder of procedure 2)
```

Figure 2-3. Procedure Declaration and its Invocation

Procedure Invocation Arguments

The invocation from the calling procedure must include an argument list with a field or variable for each parameter in the called procedure. There must be a direct correspondence in number and in data type between the arguments of the procedure invocation and the parameters of the called procedure. Values are passed between these parameters in a one-to-one correspondence; that is, from first to first, second to second, and so forth.

Constant Declarations

An ADL constant can be any ALLY data type. The ADL reserved word CONST must label all constant declarations. Single quotes must surround the value of a constant of CHAR data type. Constant declarations are easier to read when each name is indented on a line beneath the heading. Figure 2-4 shows some typical constant declarations.

```
CONST
    tax_rate1      = .25;        {number constant}
    self_employed  =  'SE';      {character constant}
    year_end       =  12/31/86;  {date constant}
```

Figure 2-4. Constant Declarations

Variable Declarations

The ADL reserved word VAR must label a variable declaration section. A variable declaration specifies a variable name, its data type, and its scope. Each variable must have a name distinct from all others in a procedure.

For readability, you can declare the name and data type of each variable on a line following the VAR label.

Figure 2-5 shows the syntax of variable declarations. In it, the variable "country_name" can contain 120 characters. "National_debt" can contain up to eighteen digits to the left of the decimal and two digits to the right. "Nano-second" can contain 10^{-9}. "Last_date_paid" can contain date values. We have made this list easy to read by indenting lines and aligning entries.

```
VAR
   country_name     : CHAR   (120) ;
   national_debt    : NUMBER (20,2); {fixed point output}
   nano_second      : NUMBER (12);   {floating point output}
   last_date_paid   : DATE;
```

Figure 2-5. Variable Declarations

Variable Data Types

Each variable must be declared as the ALLY data type: CHAR (character), NUMBER (number), or DATE (date). Table 2-2 lists each data type's characteristics, default value, and default and maximum attributes. The *Concepts and Facilities* manual contains a detailed discussion of data types.

Table 2-2. ADL Data Type Default Values and Attributes

Data Type	ADL Reserved Word	Field/Variable Can Contain	Default Value*	Default Attribute*	Maximum Attribute
Character	CHAR	Any ASCII characters		80	512
Number	NUMBER	0 through 9	0 (zero)	30	$500^{10^{\pm 32764}}$
Date	DATE	1/1/4712 BC through 12/31/4999	01/01/00	n/a	n/a

* The defaults are the data type defaults specified for the application that the ADL procedure belongs to. The values shown in these columns are the defaults provided when the Dialog builds a new AFILE. If these values are changed for the application, then the new values will be the ADL defaults.

Scope of Variables

Each variable has a scope. The scope determines which procedures may reference a variable or whether other sections of an application may reference it. A variable's value may be local to the ADL procedure or global to the entire application. The value of a local variable may also be exported to or imported from other ALLY actions that are active.

A variable that is:

LOCAL cannot be referenced outside the procedure

GLOBAL can be referenced everywhere in an application

EXPORT can be referenced by selected ADL procedures

The label for a variable's scope is placed between the data type and the terminator (;), as shown below.

```
mode : NUMBER (15) EXPORT;
```

LOCAL Scope

A local variable can be referenced only within the ADL procedure in which it is declared. At the start of each new execution of a procedure, a local variable has the default value given in Table 2-2. By default, ADL variables are local. Therefore, you are not required to label a local variable with the reserved word LOCAL.

You can use the name of a local variable in several procedures. That is, each of your ADL procedures can have a variable named "FRED" as long as each FRED variable is local.

Since a variable is local by default, you simply declare the name and data type, and no scope.

```
VAR
   FRED : NUMBER (30);
```

GLOBAL Scope

A value stored in a global variable can be referenced by any ADL procedure. You designate a variable as global by using the *Create a Global Variable* form of the Dialog.

To use a global variable in a procedure, you may declare it as a variable with GLOBAL scope. However, you are not required to declare it in the procedure. If you simply use the name in a procedure statement, ALLY will check its status. An error message will be displayed if the name is not defined as a global variable. The following example shows the two ways of referencing a global variable called "mode" in a procedure.

```
ADL Procedure 1                          ADL Procedure 2

VAR                                        mode := 1;
    mode   : NUMBER GLOBAL;

BEGIN
    mode := 1;
END;
```

EXPORT and IMPORT Scope

The label EXPORT allows the value of a local variable to be referenced by other ADL procedures or ALLY tasks or actions while the defining procedure is active.

To use the value of an EXPORT variable in another procedure, you declare a variable in the importing procedure and define it as IMPORT. You then reference the exporting procedure and its EXPORT variable before terminating the declaration. Figure 2-6 shows the communication of the values of EXPORT and IMPORT variables between two procedures.

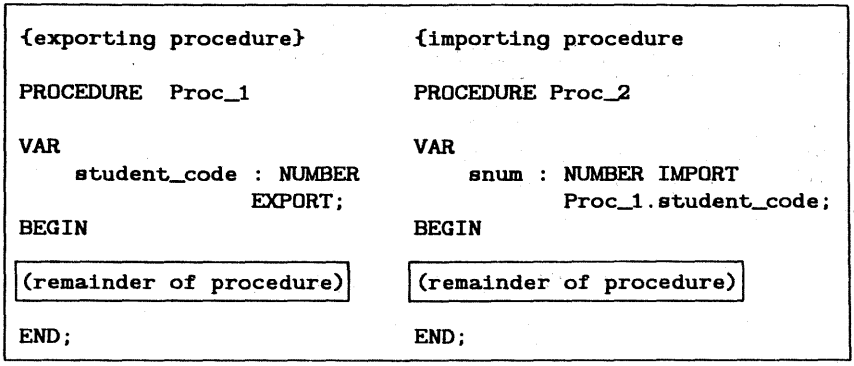


Figure 2-6. EXPORT and IMPORT Variables

The scope of ALLY form/report fields is EXPORT by default, so their values are available to all ADL procedures that execute while the form/report is active.

Anatomy of an ADL Procedure

The following table summarizes the anatomy of an ADL procedure. It shows the sections that are required and those that are optional. It also shows the syntax for each type of ADL statement.

Table 2-3. Anatomy of an ADL Procedure

Element of Procedure	Required or Optional
PROCEDURE *name*;	Optional
PROCEDURE *name* (VAR *parameter_name* : *data_type;*);	Optional
CONST *name* : *value*;	Optional
VAR	Optional
name : *data_type*;	
name : GLOBAL;	Optional
name : *data_type* EXPORT;	Optional
name : *data_type* IMPORT;	Optional
BEGIN	Optional
procedure statement(s)	Required
END;	Required with BEGIN

End of Chapter 2

Chapter 3
ADL Constructs

This section discusses the ADL constructs, grouped into:

- procedure construction
- punctuation and operator constructs
- comment text
- ADL terminology
- control statements
- functions with no arguments
- BEGIN and END
- referencing form/report and DSD fields
- character string literals
- the logical operators

Procedure Construction

ADL allows you to position the words and lines within the parts of a procedure in any way that you choose. You can include blank spaces within the procedure statements and blank lines between the statements.

In Figure 3-1, the assignment operator (:=) ends a line, the plus sign (+) and terminator (;) are on lines by themselves, and lines are indented.

```
portfolio_details.cost :=
   (portfolio.buy_price * portfolio.position_buy )
   +
   portfolio.buy_commission
   ;
```

Figure 3-1. Free-Form Procedure Construction

Punctuation and Operator Constructs

Punctuation and operator constructs affect the:

- terminator
- assignment operator
- relational and arithmetic operators

Terminator

A semi-colon (;) must terminate each procedure statement, each declaration, and each END. This example shows several instances of the terminator in an ADL procedure.

```
PROCEDURE count_schools (VAR school_count : NUMBER;);

VAR
    num_students : NUMBER;

BEGIN
    num_students := school_form.population;
    IF  (population > 0)
        THEN  school_count := school_count + 1;
END;
```

Assignment Operator

A colon followed by an equal sign (:=) designates the assignment of value to a variable or form/report field. The following example illustrates a variable "num_students," which was declared as NUMBER data type, and assigned a value of the "population" field of the "school_form."

```
num_students := school_form.population;
```

Relational and Arithmetic Operators

The ADL relational and arithmetic operators are shown in Table 3-1. The precedence of operators is shown in Table 3-2, followed by the rules for evaluating expressions.

Table 3-1. Relational and Arithmetic Operators

Relational		Arithmetic	
Symbol	**Meaning**	**Symbol**	**Meaning**
=	equality	+	addition
<>	inequality	−	subtraction
<	less than	*	multiplication
>	greater than	/	division
<=	less than or equal to		
>=	greater than or equal to		
NOT	logical negation		
OR	logical or		
AND	logical and		

The order of precedence of operators is shown below.

Table 3-2. Precedence of Operators

NOT	Done first
*, /, AND	↓
+, −, OR	↓
<, <=, =, <>, >=, >	Done last

The rules for evaluating expressions are:

- When all operators have the same precedence, the expression is evaluated from left to right.
- When operators do not have the same precedence, the highest precedence operators are evaluated first from left to right, then the next highest, etc.
- The preceding two rules can be overridden if you include parentheses in an expression. Then, the part of the expression in parentheses is evaluated first, with the above rules applied within the parentheses.

Comment Text

ADL comments must be enclosed within braces ({ }). Each comment line must begin with a left brace ({) and end with a right brace (}). Comments may be placed on lines with procedure declarations and statements. However, the comment portion of the line must begin and end with the appropriate brace. Figure 3-2 shows the use of each style of comment.

```
{Example A}
      {count only the schools that have students}
VAR   count : NUMBER GLOBAL;

IF  (num_students > 0)
    THEN  count := count + 1;

{Example B}
VAR   count : NUMBER GLOBAL;      {number data type}

IF  (num_students > 0)            {count only the schools}
    THEN  count := count + 1;     {that have students    }
```

Figure 3-2. Comment Text in a Procedure

ADL Terminology

In writing an ADL procedure, you can use:

- reserved words
- arithmetic and relational operators
- the names of forms/reports, fields, constants, and variables

Reserved Words

ADL recognizes some strings as reserved words. These reserved words are ADL functions, instructions, and ALLY commands used as arguments for ADL instructions. Each reserved word has a special meaning to ADL and cannot be used in any other context in an ADL procedure. Reserved words are listed in Appendix A.

Case Sensitivity

ADL is case-sensitive. All reserved words must be typed in uppercase. Names of form/reports, fields, and variables must be typed exactly as you defined them elsewhere in the Dialog. The following procedure shows the ADL reserved words in uppercase and the names of the form and the fields in lowercase, as they were defined.

```
VAR
    null_date : DATE;

BEGIN
   null_date := TO_DATE ('01/01/01');

{is sell date valid?}
 IF (portfolio.sell_date) = null_date
      THEN
          portfolio.net := 0;
      ELSE
{calculate net return only if sell date is valid}
          portfolio.net :=
            portfolio.proceeds - portfolio.cost
            + portfolio.dividend;
END;
```

Naming Conventions

ADL requires that all names:

- consist only of the letters a-z or A-Z, the numbers 0-9, and the underscore symbol
- start with a letter
- contain a maximum of eighty characters

Control Statements

ADL provides two constructs that execute statements conditionally.

- IF-THEN-ELSE
- WHILE-DO

IF () THEN () ELSE ()

An IF statement can take two forms: IF THEN, and IF THEN ELSE. In the first, you describe a condition, followed by statements to execute when the condition is true. In the second, you write statements to execute when the condition is false.

In Figure 3-3, a "price" calculation is performed based on the value of "cost." One example has multiple statements following THEN and requires a BEGIN and END. The other demonstrates an ELSE statement used to display an error message.

```
{Example A - IF THEN}

IF (invoice.cost > 0) THEN
   BEGIN
         inventory.item_num := invoice.item_num;
         inventory.price    := 1.5 * invoice.cost;
   END;


{Example B - IF THEN ELSE}

IF (invoice.cost > 0)
   THEN
         inventory.price := 1.5 * invoice.cost;
   ELSE
         ERROR (1234);
```

Figure 3-3. IF Statement

WHILE () DO

A WHILE statement specifies that WHILE a condition is true DO one or more statements. When the condition is false, the statements following DO will not be executed. BEGIN and END must surround multiple statements that follow DO.

If the WHILE condition is not true for the first iteration, the DO statements will be skipped and not executed.

Figure 3-4 deletes all records up to the one with the value "[EOB]."

```
CALL_CMD (RGLAST);
WHILE (DEL_FORM.DESC <> '[EOB]') DO
      CALL_CMD (DELREC);
```

Figure 3-4. WHILE Statement

Functions With No Arguments

ADL requires that a set of parentheses be included in procedure statements using functions that require no arguments. Figure 3-5 lists the ADL functions that take no arguments.

```
DB_END_GROUPS ();
DB_OPEN ();
DB_RELATED_GROUPS ();
GET_CMD ();
SET_FAILURE ();
SET_SUCCESS ();
```

Figure 3-5. Functions that Take No Arguments

BEGIN and END

BEGIN and END must surround multiple statements in IF and WHILE control constructs.

BEGIN and END must surround the body of a procedure that begins with the reserved word VAR or CONST. Otherwise they may optionally surround the procedure body.

Referencing Form/Report and DSD Fields

ADL requires that references to form/report and DSD fields be "form_or_DSD_name.fieldname," exactly as they have been defined in the application. For example, an ADL reference to a form defined in the application as "portfolio" with a field named "buy_price" would be "portfolio.buy_price."

Character String Literals

ADL character string constants are delimited by single quotes. To get a single quote in a character literal, you must enter two successive single quotes. This example puts the value "literal with quote ' here" in the string field.

```
string := 'literal with quote '' here';
```

The Logical Operators

The logical operators are:

- AND
- OR
- NOT

(*expression*) AND (*expression*);

AND is a relational operator that you can use in IF and WHILE condition statements. Its two arguments must be relational expressions enclosed within parentheses. The value of the AND operation is true only when both expressions are true.

In this example, the value of the "count" variable will be incremented if both expressions joined by AND are true.

```
VAR
  count : NUMBER;

BEGIN
   IF (school.num_students > school.num_athletes)  AND
      (school.num_students > school.num_musicians)
      THEN  count := count + 1;
END;
```

(*expression*) OR (*expression*)

OR is a relational operator used in IF and WHILE condition statements. It requires two arguments that must be relational expressions and must be enclosed in parentheses. The value of the OR operation is true if either one or both arguments are true. The value is false only when both arguments are false.

In this example, the value of "count" is incremented by one when "a" is greater than either "b" or "c".

```
VAR   a      : NUMBER;
      b      : NUMBER;
      c      : NUMBER;
      count  : NUMBER GLOBAL;
BEGIN
  IF (a > b) OR (a > c) THEN
    count := count + 1;
END;
```

NOT (*expression*);

NOT is a relational operator used in IF and WHILE condition statements. It requires one argument that must be a relational expression and must be enclosed in a set of parentheses. The value of the NOT operation is the negative of its argument. That is, the value is true when the argument is false, and the value is false when the argument is true.

In this example, the value in "count" will be incremented by one whenever "a" is not greater than "b".

```
VAR   a      : NUMBER;
      b      : NUMBER;
      count  : NUMBER GLOBAL;
BEGIN
   IF NOT (a > b) THEN
      count := count + 1;
END;
```

End of Chapter 3

Chapter 4
ADL Functions and Instructions

This chapter describes the ADL functions and instructions that allow you to:

- manipulate variable and field values
- perform calculations with dates
- invoke tasks and actions
- use ALLY commands
- invoke help and error messages
- monitor the success or failure of an ADL statement

Table 4-2, at the end of this chapter, summarizes these functions and instructions.

Manipulating Variable and Field Values

The MAKE_NULL instruction and following built-in functions nullify the value of a variable or convert it to a different data type.

- TO_NUMBER
- TO_DATE
- TO_CHAR
- ROUND
- TRUNC

Since the instruction MAKE_NULL affects variables regardless of data type, it is described below. The functions are grouped by data type.

MAKE_NULL (*variable_or_field_name*);

MAKE_NULL is an instruction that causes a variable or field of any data type to have no value (i.e., to become null). It allows you to set a value to null and use it for comparisons.

Null is different from zero. Zero is a value. Null means the contents of the field or variable has no value. Null values are ignored in the evaluation of all arithmetic expressions and the computations of all functions. Null values are treated as unknowns in the evaluation of logical expressions. But, values can be tested to see if they are null.

In this example, the variable "null" sets the comparison for an error condition.

```
VAR
    null : NUMBER;

BEGIN
     MAKE_NULL (null);
     CALL_CMD (QUERY);
     IF (REP_PRINT_LONG.STR_NO = null) THEN
        BEGIN
              ERROR (16);
              CALL_CMD (QBE);
        END;
END;
```

CHAR Formatting Functions

ADL provides two formatting functions that enable you to convert the value of a variable or field from CHAR to NUMBER or from CHAR to DATE data type.

TO_NUMBER (*CHAR_argument*);

This function converts the internal storage of a value from CHAR (ASCII) to NUMBER (binary). The result is a value of NUMBER data type. The character variable or field must contain the character representation of a number.

In this example, the value of the character field ZIP_STRING of the form/report EMPLOYEE is converted to a number, to be used subsequently in a zip code.

```
VAR
   ZIP_CODE_NUMBER : NUMBER;

BEGIN
    ZIP_CODE_NUMBER := TO_NUMBER (EMPLOYEE.ZIP_STRING);
END;
```

TO_DATE (*CHAR_argument, optional_date_picture*);

This function converts the internal storage of a CHAR argument from ASCII to date data. You specify the format of the CHAR argument with the date picture, which has the ALLY data type CHAR. If the format of the CHAR argument is the DATE default (MM/DD/YY), you are not required to specify the date picture. The result has the data type DATE. The character variable or field must contain the character representation of a date (e.g., '02-29-88') in the same format that you specify in the date picture (e.g., 'MM-DD-YY'). TO_DATE converts a date to the format specified for that field in your form/report.

If you do not specify a date format picture, TO_DATE converts the date string to the default format of MM/DD/YY if that date string will fit into the default format. If the date string will not fit into the default format, you will receive a syntax error. TO_DATE ignores both leading and trailing blanks.

In this example, the character string 02/29/88 will be converted to date data type using the date picture MM/DD/YY.

```
VAR
    LEAP_YEAR   : DATE;

BEGIN
    LEAP_YEAR := TO_DATE ('02/29/88', 'MM/DD/YY');
END;
```

In the second example, the value of PAY_DATE will be Tuesday 03-08-84.

```
VAR
    PAY_DATE : DATE;
BEGIN
    PAY_DATE := TO_DATE ('TUES 03-08-84', 'DAY MM-DD-YY');
```

NUMBER Formatting Functions

The ADL functions that reformat numbers allow you to:

- convert the value of a variable or field internally from binary to ASCII
- round a number value
- truncate a number value

TO_CHAR (*NUMBER_argument*);

This function converts the value of a number field internally from binary to ASCII representation. TO_CHAR ignores both leading and trailing blanks.

This example converts the value of the field ZIP_NUM from NUMBER to CHAR data type.

```
VAR
   ZIP_STR : CHAR;

BEGIN
    ZIP_STR := TO_CHAR (MAIL_FORM.ZIP_NUM);
END;
```

ROUND (*NUMBER_argument, optional_NUMBER_precision*);

This function rounds a number to the precision you specify. ROUND can take two arguments, both NUMBER values. The first argument is required and the second (the precision) is optional. The result is a NUMBER value when ROUND is used with a NUMBER argument. If no precision is specified, ROUND rounds the fractional part of the number up to the next integer value.

In this example, ROUND produces two different results when a precision is specified and when it is not.

```
VAR    x : NUMBER;
       y : NUMBER;
       z : NUMBER;
BEGIN
  x := 1.5367;
  y := ROUND (x,2);   {value will be 1.54}
  z := ROUND (x);     {value will be 2}
END;
```

TRUNC (*NUMBER_argument, optional_NUMBER_precision*);

This function truncates a number to the precision you specify. The first argument is required and the second (the precision) is optional. The result is a NUMBER value when TRUNC is used with a NUMBER argument. If no number precision is specified, the fractional part of the number is truncated.

In this example, TRUNC produces different results, depending on whether a precision is specified.

```
VAR    x : NUMBER;
       y : NUMBER;
       z : NUMBER;
BEGIN
  x := 1.5367;
  y := TRUNC (x,2);   {value will be 1.53}
  z := TRUNC (x);     {value will be 1}
END;
```

DATE Formatting Functions

The ADL functions that reformat dates allow you to:

- convert the value of a variable or field internally from DATE to ASCII
- round a date value
- truncate a date value

TO_CHAR (*DATE_argument, optional_date_picture*);

TO_CHAR converts the value of a date field internally from date to ASCII representation. The first argument must have the data type DATE. The second argument, which is the optional date picture, must be the data type CHAR. Date pictures are described in Appendix C.

In this example, TO_CHAR extracts the year from the date stored in the form/report field "hiredate."

```
VAR   year        : CHAR;
BEGIN
    year := TO_CHAR (employee_form.hiredate,'YYYY');
END;
```

ROUND (*DATE_argument , optional_date_picture*);

ROUND rounds a DATE value to the precision you specify. The optional second argument is the date precision and must be a CHAR data type. The result is a DATE value when ROUND is used with a DATE argument. If you do not specify a precision, ALLY rounds the new date to MM/DD/YY. The valid date picture precisions for rounding of dates are listed in Appendix C.

In this example, ROUND rounds the employment date to the nearest year. The date picture YEAR rounds a date to the current year for dates in January through June and to the next year for dates in July through December. Both form/report fields referenced are DATE fields.

```
employee_form.pension_year
        := ROUND (employee_form.hiredate, 'YEAR');
```

TRUNC (*DATE_argument , optional_date_picture*);

TRUNC truncates a DATE value to the precision you specify. The optional date precision is a CHAR data type. The result is a DATE value when TRUNC is used with a DATE argument. If you do not specify a precision, ALLY truncates the new date at MM/DD/YY. The valid date picture precisions for truncating dates are listed in Appendix C.

The following example shows the use of TRUNC in a form/report calculation.

```
personnel.hireyear := TRUNC (employee.hiredate, 'YY');
                              {last 2 digits of the year}
```

Manipulating CHAR Strings

ADL provides two functions for string manipulation:

- SUBSTR
- Concatenate (||)

SUBSTR (*CHAR_container, offset, length*);

SUBSTR allows you to assign to a CHAR container a subset of the value of another CHAR container. The arguments are the number of the offset in the string and number of characters of the subset. The offset and length can be numbers or NUMBER containers.

To select the substring, ALLY identifies a starting character in the original string using an offset you supply. Then ALLY selects the number of characters indicated by the length argument. Offsets into a character string start with zero rather than one, and the length includes the starting character. To illustrate, SUBSTR ('ABCDEFG', 2, 3) will select "CDE", by using the offset of 2 to identify "C" as the starting point and using the length of 3 to define the number of characters.

In the following example, the value of hiredate is 01/02/86. SUBSTR will assign the year to "x". Note that multiple ADL functions can be used together, in this case SUBSTR and TO_CHAR.

```
VAR
    x          : CHAR;

BEGIN
    x := SUBSTR (TO_CHAR (emp_form.hiredate), 6, 2);
END;
```

Concatenate (||)

The string concatenation function is represented by two vertical bars (||). In this example, the value assigned to the "list" field is the value of the "form_name" field concatenated with a period (.) to the value of the "field_name" field.

```
forms.list := dept.form_name || '.' || dept.field_name;
```

Calculating with DATE Values

This subsection describes the arithmetic you can perform with DATE values and the ADL functions that allow you to perform calculations with DATE values.

- ADD_MONTHS
- LAST_DAY
- MONTHS_BETWEEN
- NEXT_DAY

Arithmetic with DATE Values

ADL allows you to add days to dates and subtract days from dates. To perform either of these arithmetic operations, you specify the date or the name of the variable or form/report field that contains the date, the appropriate arithmetic sign (+ or −), and the number of days to be added or subtracted.

In the following example, the value of the field INITIAL_REVIEW is calculated by adding ninety days to the employee's hire date. Then the value of the NEXT_REVIEW field of the MANAGER_TICKLER form is calculated by subtracting ten days from the review date.

```
MANAGER_REPORT.INITIAL_REVIEW := EMP_FORM.HIRE_DATE + 90;

MANAGER_TICKLER.NEXT_REVIEW
       := MANAGER_REPORT.INITIAL_REVIEW - 10;
```

You can also perform date arithmetic using the TO_DATE function as shown in the following example. The arithmetic in this example would assign the value −2 to the variable days_diff.

```
VAR
    days_diff : NUMBER;

days_diff := TO_DATE ('01/07/86') - TO_DATE ('01/09/86');
```

ADD_MONTHS (*DATE_argument, NUMBER_argument*);

This function adds the number of months you specify to a date. The actual calendar months, with their different numbers of days, are added. The result is a DATE.

The time component that may be a part of the DATE argument is not modified by the ADD_MONTHS function.

In this example, the variable "first_date" will be assigned the value 6/11/84, "second_date" will be assigned the value 10/31/84, and "third_date" will be assigned the value 10/31/84.

```
VAR  first_date  : DATE;
     second_date : DATE;
     third_date  : DATE;
BEGIN
           {add 2 months}
  first_date  := ADD_MONTHS (4/11/84,  2);

           {subtract 3 months}
  second_date := ADD_MONTHS (1/31/84, -3);

           {preserve last-day-of-month in result}
  third_date  := ADD_MONTHS (9/30/84,  1);
END;
```

LAST_DAY (*DATE_argument*);

This function calculates the last day of the month of the date argument. The result is a DATE value. It takes leap year into account in its calculations. So, if the result is the last day of February 1984, it would return February 29, since 1984 was a leap year.

In this example, LAST_DAY will return the last day of the month that employees were hired. This might be used to calculate the number of people employed by the company on the last day of each month.

```
VAR   month_end : DATE;
BEGIN
  month_end := LAST_DAY (employee_form.hiredate);
END;
```

MONTHS_BETWEEN (*DATE_argument*, *DATE_argument*);

This function calculates the number of months between the first date and the second date specified. The result has the data type NUMBER. The time component is not considered when determining the months between two dates.

If the date in the first argument is later than the date in the second argument, the returned number is positive. If the date in the second argument is the later date, the returned number is negative.

Only whole months are considered, and the concept of "last day" is preserved. From the last day of one month to the last day of the next month returns a count of one month. Thus, from January 31 to February 28 counts as one month in a non-leap year but in a leap year counts as zero. From January 15 to March 31 counts as two months.

In this example, MONTHS_BETWEEN will calculate the number of months between the dates in the "end" and "start" fields of the

subscription form "subs." The subscription end date is in the first argument, since it should be later than the subscription start date.

```
subscription.num_months
    := MONTHS_BETWEEN (subs.end, subs.start);
```

NEXT_DAY (*DATE_argument, day_of_week*);

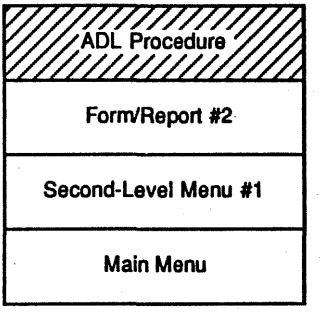
This function requires a date and spelled-out day of the week. It calculates the date of the next occurrence of that day of the week. The first argument must be DATE and the second CHAR, the spelled-out day of the week. The result is a DATE data type.

In this example, NEXT_DAY will calculate the date of the first payday for each employee, since this company's employees are paid every Friday. Both form/report fields referenced here are DATE fields.

```
        {day of week may be uppercase or lowercase}
employee_form.first_paydate
   := NEXT_DAY (employee_form.hiredate, Friday);
```

Invoking Tasks and Actions

ADL uses the concept of a stack of tasks and actions, as does PASCAL. Figure 4-1 shows an example of an execution stack. ADL flow of control mechanisms provide access to the actions on the stack. Invocation mechanisms allow you to determine which task or action is active. They let you move from one task to another, and let you set the sequence in which a series of actions within a task is executed.



F002-0573-00

Figure 4-1. An Execution Stack

Where PASCAL and other languages provide only one standard invocation mechanism, ADL provides seven options, three for tasks and four for actions.

Invoke a Task

A task consists of one or more actions. There are three ADL instructions for invoking tasks.

- FORK
- START
- RESUME

FORK *task_name*;

FORK pauses the ADL procedure, then starts executing the first action in the named task. All activity takes place within the named task until that task is aborted or exited, or another invoca-

tion mechanism is encountered. The argument can be the name of any task.

This example invokes the ALLY Text Editor (ALLYedit) when the value of "a" is greater than the value of "b."

```
IF (a > b) THEN
   FORK ALLYEDIT_TASK;
```

START *ALLY_task_name*;

START pauses the ADL procedure, then starts executing the first action in the named task until that task requires user input. The named task is then paused, and ALLY continues executing the original action in the original task. It appears to the user that the application activity has never left the original task, unless the started task displayed something on the terminal. The argument must be the name of an ALLY task.

The procedure in the example will start the editing task.

```
START ALLYEDIT_TASK;
```

RESUME *ALLY_task_name*;

RESUME pauses the ADL procedure, then resumes executing the current action in the named, previously-paused task. The argument must be the name of an ALLY task.

In this example, the procedure resumes the text editor task when the value of "a" is greater than the value of "b".

```
IF (a > b) THEN
 RESUME ALLYEDIT_TASK;
```

Invoke an Action

An action is a form/report packet, a menu, another ADL procedure, a parameter packet, an external program link, an action list, or a text editor. There are four ADL instructions for invoking actions.

- CALL
- EXECUTE
- RETURN_TO
- RETURN

CALL *action_name*;

CALL leaves the ADL procedure on a task's execution stack, executes the called action, and returns to the calling action when the called action ends. CALL marks the original action to show where the action should be resumed. Its argument must be the name of an action.

After the called action executes, the ADL procedure becomes the active action. As long as an action is present on the task's stack, the task continues.

This example shows a call to an ADL verification procedure. The call passes the values of two arguments to the procedure.

```
CALL ADL_VERIFY (x, y);
```

EXECUTE *action_name*;

EXECUTE removes the ADL procedure from a task's execution stack, then executes the called action. This means that any statement in an ADL procedure that follows an EXECUTE statement will not be executed. After the called action executes, the task terminates unless there is an action pushed beneath the calling action. Its argument can be any ALLY action.

The procedure in the example executes the final menu when "a" is not greater than zero.

```
IF (a <= 0) THEN
EXECUTE FINAL_MENU;
```

RETURN_TO *action_name*;

RETURN_TO removes from a task's execution stack all actions back to the named action, then resumes execution of the called action. This means that any statement in your ADL procedure that follows a RETURN_TO statement will not be executed. Its argument can be any ALLY action.

This example invokes the application's main menu when it encounters the end of the file.

```
IF (status := DB_EOF) THEN
    RETURN_TO MAIN_MENU;
```

RETURN;

RETURN transfers control from the ADL procedure back to the calling action. It takes no arguments. It is useful as part of error checking; if an error condition is found, control can leave the procedure and return to the calling action.

```
IF (status <> 0) THEN
   RETURN;
```

Using ALLY Commands

The following ADL instructions allow you to use an ALLY command.

- CALL_CMD
- EXECUTE_CMD
- GET_CMD

CALL_CMD (*ALLY_command_name*);

CALL_CMD executes an ALLY command. The argument may be the name of any ALLY command (listed in Appendix A) or a constant or variable whose value is the name of an ALLY command. If the argument is a variable, the variable must have the data type NUMBER, since ALLY internally stores each ALLY command as a number value.

In this example, all records will be retrieved for which the value in the field "last_name" is "SMITH."

```
CALL_CMD (QBE);
form.last_name := 'SMITH';
CALL_CMD (QUERY);
```

EXECUTE_CMD (*ALLY_command_name*);

EXECUTE_CMD stops the ADL procedure and then executes the command you specify. The ADL procedure is not restarted, and any statements in an ADL procedure that follow an EXECUTE_CMD will not be executed. The argument may be the name of any ALLY command (listed in Appendix A) or a constant or variable whose value is the name of an ALLY command. If the argument is a variable, the variable must have the data type NUMBER, since ALLY stores ALLY commands as number values.

In the following example, when the value of the "price_field" is less than zero, the text of error message 1234 will be displayed. The form/report and ADL procedure will stop.

```
IF (form.price_field < 0) THEN
   BEGIN
         ERROR (1234);
         EXECUTE_CMD (EXITACTION);
   END;
```

GET_CMD ();

GET_CMD requires no argument. It returns an internal number that references the form/report command that preceded the invocation of the ADL procedure. ALLY does not execute a form/report command that immediately precedes a GET_CMD.

You can use GET_CMD only in ADL when a form/report is active. It is most frequently used in after-field, before- and after-value change, and validation procedures to compare a returned command. If it is used outside a form/report, it produces no result.

You can use GET_CMD in an "after" field procedure to intercept a form/report command and change it. Or, your procedure can perform some manipulations and not change the command. If you want the intercepted command to perform its original purpose, you must re-invoke that command (with CALL_CMD).

This allows you to change the command to another command or to perform some manipulations and then reuse that command.

In this example, the values of each record will be committed when the user issues the 'next record' command. When the user enters RNEXT, ALLY internally references RNEXT as a number. Note that the 'next record' command is invoked after it has been intercepted and checked since we want to use that command but not change it.

```
VAR  LAST_COMMAND : NUMBER;
BEGIN
   LAST_COMMAND := GET_CMD ();
   IF (LAST_COMMAND = RNEXT) THEN
      BEGIN
            CALL_CMD (COMMIT);
            CALL_CMD (RNEXT);
      END;
   ELSE
            CALL_CMD (LAST_COMMAND);
END;
```

Invoking Help and Error Messages

There are two ADL instructions that enable you to display the text of a help or error message.

- HELP
- ERROR

HELP (*NUMBER_argument*);

HELP displays the text of a help message. The help message can be part of an application's AFILE or a library AFILE that contains help and error messages. The application developer creates the text of a message and assigns it a number through the Dialog. The argument must be a number or a NUMBER variable.

```
VAR
   LAST_COMMAND : NUMBER;
BEGIN
   LAST_COMMAND := GET_CMD ();
   IF (LAST_COMMAND = (QUERY)) THEN
      HELP (342);
END;
```

ERROR (*NUMBER_argument*);

ERROR displays the text for an error message. The argument must be a number or NUMBER variable. The application developer creates this error number and its text through the Dialog while designing an application.

In the following example, when the value of the "price_field" is less than zero, the text of error message 1234 will appear on the user's terminal.

```
IF (SALES_FORM.PRICE_FIELD < 0)
   THEN
        ERROR (1234);
```

Reporting Procedure Success or Failure

These instructions allow you to notify the calling action that a procedure has succeeded or failed.

- SET_SUCCESS
- SET_FAILURE

SET_SUCCESS ();

This instruction, which takes no argument, sets a flag that tells the calling event that the ADL procedure has executed successfully.

If the following procedure is called before a field update then the update will take place if the field "num_students" does not have a value of zero.

```
IF (SCHOOL.NUM_STUDENTS > 0) THEN
   SET_SUCCESS ();
```

SET_FAILURE ();

This instruction takes no argument. It sets a flag that tells the calling event that the ADL procedure has failed.

You can influence the operation of a form/report by your placement of the procedure in which the failure flag is set. Table 4-1 lists the effect of setting a failure flag in forms/reports.

Table 4-1. Effect of SET_FAILURE Flag in a Form/Report

Failure Flag Set	
Event	**Effect**
Before a commit	The commit will not take place
Before an update	The update will not take place
Before a delete	The delete will not take place
After a form/report packet	The cursor cannot leave the form/report
Field validation	The cursor cannot leave the field

If the following procedure is called to validate the form/report field "num_students," then the cursor will not leave that field when its value is zero. You can use this technique to force users to enter valid data into a field.

```
IF (SCHOOL.NUM_STUDENTS = 0) THEN
   SET_FAILURE ();
```

Summary

The ADL functions and instructions that allow you to manipulate dates, convert data, and manage flow of control for tasks and actions are listed in Table 4-2.

Table 4-2. ADL Functions and Instructions

Operation	ADL Function Name
Manipulate variable and field values	MAKE_NULL
CHAR values	TO_NUMBER TO_DATE
NUMBER values	TO_CHAR ROUND TRUNC
DATE values	TO_CHAR ROUND TRUNC
Manipulate CHAR strings	SUBSTR Concatenate (||)
Perform calculations with dates	ADD_MONTHS LAST_DAY MONTHS_BETWEEN NEXT_DAY
Invoke tasks and actions	FORK START RESUME CALL EXECUTE RETURN_TO RETURN

Operation	ADL Function Name
Use ALLY commands	CALL_CMD EXECUTE_CMD GET_CMD
Invoke help and error messages	HELP ERROR
Report the success or failure of a validation	SET_SUCCESS SET_FAILURE

End of Chapter 4

Chapter 5
Generic DML

The ADL Generic Data Manipulation Language (DML) instructions provide an interface to any access methods that are supported by ALLY. The general purpose of these instructions is to allow input or output, either in addition to or as an alternative to forms/reports.

Specifically, Generic DML instructions use any Data Source Definition (DSD) and permit you to:

- open and close a DSD or a hierarchy of DSDs
- specify and modify selection criteria
- retrieve individual records
- insert, update, or delete individual records
- use access-method-specific commands

Table 5-1 lists the Generic DML instructions, grouped by operation. DML instructions are distinguished from the rest of ADL by their names, which all start with "DB_."

Table 5-1. Generic DML Instructions by Operation

Instruction Name	Operation
DB_OPEN DB_CLOSE DB_RELATED_GROUPS DB_END_GROUPS	Open and close a DSD or a hierarchy of DSDs
DB_RESET DB_QUERY DB_CLAUSE	Describe and modify query criteria
DB_GET_FIRST DB_GET_NEXT	Retrieve DSD records

Instruction Name	Operation
DB_UPDATE DB_DELETE DB_INSERT	Modify DSD records
DB_COMMIT DB_ROLLBACK	Perform access method transactions
DB_COMMAND	Pass access-method-specific commands

Introduction

All types of DSDs, including View Definitions and Breakup Definitions, are supported from ADL. A DSD must have a name and be associated with a particular dataset, file, or table in a file system or database. Optional selection criteria may be included if they are supported by the access method.

Fields are referenced from ADL much as they are from a form/report. The same DSD can, in fact, be opened in a form/report concurrently with the opening from an ADL action. This should be done only where there is no possibility of a deadlock in which two or more transactions are in a simultaneous wait state, each waiting for the others to release a lock before it can proceed. You should refer to documentation that accompanies your access method for specific information on deadlocks.

From ADL, you directly reference a DSD with the syntax "DSD_name.field_name," just as you reference a form/report field with "form_report_name.field_name." You do not explicitly declare DSDs from ADL; they are available once they are opened with the DB_OPEN instruction. Because of the ADL restriction of one logical transaction per task, a DSD can be opened only once within a task.

When a DSD is opened, it is identified by an internal "open_id" number. The DSDs remain open until explicitly closed or until the task terminates.

ADL provides no interface for managing a commit or rollback transaction to an individual DSD. Therefore, commit and rollback operate on all DSDs that are open and are not access-method specific. If the task terminates, each DSD that is open is rolled back and closed.

Mixing Form/Report and ADL Transactions

There are several ramifications of mixing form/report transactions with an ADL transaction.

The Generic DML instructions cannot suspend file updates. An ADL commit calls the corresponding access method directly and commits only the records that the ADL procedure has changed. For access methods that support multiple logical transactions in one process, ADL and form/report transactions are totally distinct. The form/report changes are stored until they are committed, and are not affected by an ADL commit.

The form/report mechanism, however, permits rollback even with access methods that do not normally support it (e.g., fixed-length sequential files). For access methods that do not support multiple active transactions for a single operating-system process, the ALLY forms/reports manager logs and batches the updates. A commit on this form/report commits all updates since the last commit, including those from ADL. This is caused by the access-method interface not allowing separate, simultaneous transactions from one ALLY application.

Related Data Source Definitions

The structure of each form/report defines hierarchical relationships among its DSDs. In ALLY, such relationships are represented with either Foreign Key Links or Breakup Definitions.

A DSD can be used by itself, or with any DSDs subordinate to it. When you use related DSDs, you must recognize the hierarchy in the commands you use to open the DSDs and perform operations on them. For example, in a hierarchy of DSDs, data queries at one level affect the subset of records retrieved for all lower levels.

Hierarchy of DSDs

Before you can open a subordinate DSD in a hierarchy, you must open all the DSDs logically above it. ADL provides two special instructions (DB_RELATED_GROUPS and DB_END_GROUPS) that must surround the openings of two or more related DSDs.

To specify the opening of a hierarchy of DSDs, you use the DB_RELATED_GROUPS instruction before a list of DB_OPENs. You use a DB_OPEN for each DSD, from the top level down. You use DB_END_GROUPS to specify the end of this group. No other operation can be performed on a DSD that belongs to a hierarchy until any related DSDs have been opened. After that, you can set query criteria for any of the DSDs.

Querying DSDs in a Hierarchy

Related DSDs must be queried as a unit. To clear query criteria and set new criteria for a subordinate DSD, DB_RESET and DB_CLAUSE must reference the top-level DSD of the hierarchy.

Record Retrieval from DSDs in a Hierarchy

Retrieval of records from the DSDs of a hierarchy must be done in order, from the top level DSD down. Before you can retrieve records (with DB_GET_NEXT or DB_GET_FIRST) of a subordinate DSD, you must first have retrieved at least one record (with DB_GET_FIRST) of each higher-level DSD. From this, ALLY determines the chain of subordinate records that is valid for each record.

Deleting and Inserting DSD Records

You must be careful when deleting and inserting records in the upper-level DSDs of a DSD hierarchy. Since the deletion and insertion operations change the record in the upper-level DSD, a new chain of subordinate records must be established. Therefore, to access records from subordinate DSDs, you must first issue a DB_GET_FIRST at each of those levels to establish the new subordinate records that are valid.

Arguments for Generic DML Instructions

The description of each Generic DML instruction includes its syntax. A few instructions require no argument. However, most of the instructions require the two arguments *open_id* and *status_code*. These arguments are described below. Any other argument required by an instruction is described with that instruction.

Both open_id and status_code must be declared as variables of NUMBER data type. You may choose any name for these variables, however, OPEN_*dsd_name*_ID and STATUS_*dsd_name* are descriptive.

Open_ID Variable

You name an open_id variable for each DSD that your ADL procedure will access. When a DSD is opened with the DB_OPEN instruction, ALLY assigns to the open_id variable an internal number identifying that opening. You must pass this number (through the open_id variable) to all successive Generic DML instructions to identify that DSD.

The open_id variable for a DSD must be exported or global if it will be referenced in another ADL procedure.

Status_Code Variable

ALLY monitors the success or failure status of DML operations through the status code it passes to each instruction. Zero (0) indicates success. An error number is set only when ALLY finds an access-method-level error that is not fatal. When the error is fatal, the ADL action terminates immediately without passing control back to the offending ADL statement.

There are two methods of using the status code to monitor your DML operations:

- You may declare one status_code variable, and monitor each DSD by checking the value of the status_code variable after each DML operation that returns a status code.
- You may declare a status_code variable for each DSD, just as you assign an open_id variable for each DSD.

You can use the status code to test for errors after each DML instruction by using the ADL ERROR instruction to display the error message text for the returned status code variable. After each DML instruction, put into the procedure the statement below. You can then handle any errors appropriately for that procedure.

```
IF (status_code_variable <> 0)
   THEN  ERROR (status_code_variable);
```

Generic DML Global Constants

Generic DML provides three global constants you can use to monitor the success or failure of a DML operation:

DB_DUPLICATE_RECORD	Signifies that this dataset, file, or table already contains a record with the same primary key value as the one you are attempting to insert
DB_EOF	Signifies that you have reached the last record in the dataset, file, or table or that you have reached the last record in the subset that you are querying
DB_OPEN_ERROR	Signifies that the DSD named in the DB_OPEN is invalid or that you have used the DB_OPEN without a preceding DB_RELATED GROUPS

To use these global constants in status checking, you compare the status code variable to the appropriate constant. The following procedure returns to the calling event if the DSD does not open.

```
IF   (STATUS_CODE = DB_OPEN_ERROR)   THEN
     RETURN;
```

Generic DML Instructions

This section is divided into subsections that describe the general operation of Generic DML instructions.

There is an example at the end of each subsection that shows the syntax and usage of the appropriate instructions. Each example uses the same procedure, which we expand as each group of instructions is added.

Figure 5-1 shows the relationship among the DSDs that are used in the example procedure. Figure 5-7 shows the completed procedure that uses each instruction. There is no example of the use of DB_COMMAND since it is access-method specific, and this manual addresses no specific access method.

Opening and Closing a DSD

The Generic DML instructions that open and close a DSD or a hierarchy of DSDs are:

- DB_OPEN
- DB_CLOSE
- DB_RELATED_GROUPS
- DB_END_GROUPS

In general, you begin an ADL procedure that operates on your access method with a DB_OPEN for each DSD and end the procedure with a DB_CLOSE for each DSD. When a procedure uses a hierarchy of DSDs, you use DB_RELATED_GROUPS before the DB_OPEN for the top-level DSD in the group and DB_END_GROUPS after the last DB_OPEN for the group. The correct positioning of these instructions is illustrated in Figure 5-2.

DSDs that are not related cannot be grouped together between DB_RELATED_GROUPS and DB_END_GROUPS. This is not allowed because it would impose unnecessary constraints on querying and on the order of record retrieval. Because a DSD

can be opened only once within an ADL procedure, an attempt to group unrelated DSDs would prevent the opening of any of the component DSDs until the group opening had ended.

DB_OPEN (*DSD_name, open_id, status_code*);

The DB_OPEN instruction readies the named DSD for access. ALLY puts into the open_id variable the internal number that identifies the DSD. Any other Generic DML instructions that operate on this DSD use this variable as an argument. The DSD opens with any selection criteria specified in the DSD definition (e.g., the initial SELECT statement).

If the DSD being opened is a top-level DSD, DB_OPEN must be the first DML instruction issued. If the DSD being opened belongs to a hierarchy of DSDs to be opened, the instruction DB_RELATED_GROUPS must precede the DB_OPEN for the top-level DSD in the hierarchy. Related DSDs must be opened in their hierarchical order, from the top down. DB_END_GROUPS must follow the DB_OPEN of the lowest DSD in the hierarchy.

If the DSD being opened is a Breakup DSD, you open the Breakup but not the underlying Base Definition, since ALLY opens it for you.

The DSD remains open until a DB_CLOSE is issued, or until the task (not the action) terminates.

DB_OPEN can generate an error message if:

- you try to open the same DSD more than once in the procedure
- the access method cannot open the underlying dataset, file, access method, or table
- you open related DSDs in the wrong order or fail to use DSD_RELATED_GROUPS
- you try to open a DSD that does not exist

DB_CLOSE (*open_id, status_code*);

DB_CLOSE closes the specified DSD so that no further DML operations can be performed on it. This frees the memory that has been allocated to that DSD in the ADL procedure.

A DSD can be closed successfully only if all record modifications have been either committed or rolled back. Thus, before you issue a DB_CLOSE on any DSD you must issue either a DB_COMMIT or a DB_ROLLBACK.

A DB_CLOSE issued on a DSD that belongs to a hierarchy causes all the other DSDs in that group to be available only for DML transactions (commit and rollback) and closing. Any other operation that affects those DSDs displays an error message that indicates that all related DSDs must be opened at the same time.

Although the order of the closings is irrelevant, we recommend closing related DSDs together in succeeding statements.

DB_RELATED_GROUPS ();

DB_RELATED_GROUPS specifies that DSD openings occurring between it and the next DB_END_GROUPS belong to a hierarchy.

ADL will issue an implicit DB_END_GROUPS if a DB_RELATED_GROUPS was already in progress for the procedure.

DB_RELATED_GROUPS generates no error messages.

DB_END_GROUPS ();

DB_END_GROUPS signals to ADL that the opening of a hierarchy of related DSDs has ended. The DB_RELATED_GROUPS instruction must precede the DB_OPEN for the top DSD in a hierarchy.

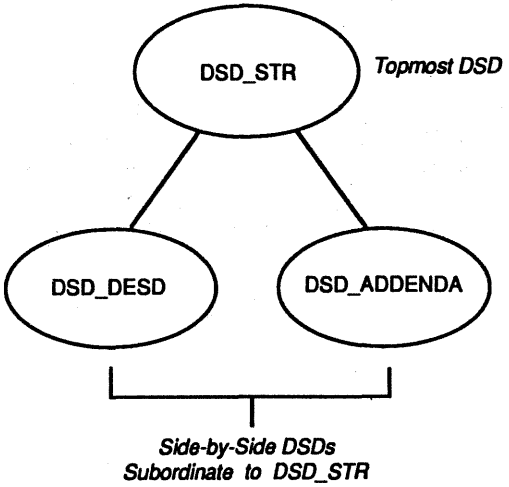
If the ADL procedure terminates prior to the DB_END_GROUPS, the DSDs in the hierarchy are closed automatically.

Until the DB_END_GROUPS instruction is issued for a DSD hierarchy, other Generic DML instructions can operate only on DSDs outside of this DSD hierarchy. However, no DML operations can be performed on any of these related DSDs. There is no restriction on calling another ADL procedure that starts a separate tree of related DSDs with another DB_RELATED_GROUPS statement.

If you have not issued a DB_RELATED_GROUPS instruction prior to opening a hierarchy of related DSDs, DSD_END_GROUPS does nothing. DB_END_GROUPS generates no error messages.

Example Illustrating DML Instructions that Open and Close DSDs

In this example, the names DSD_STR, DSD_ADDENDA, and DSD_DESC are the names of the DSDs in the application. Note that we are using the status code and global constants to check for errors. Figure 5-1 illustrates the relationship among these DSDs. This relationship exists in all the subsequent examples that use these DSDs.



F002-0574-00

Figure 5-1. Relationship of DSDs in DML Examples

```
VAR
  STATUS_CODE      : NUMBER;
  OPEN_STR_ID      : NUMBER;
  OPEN_ADDENDA_ID  : NUMBER;
  OPEN_DESC_ID     : NUMBER;

BEGIN                                          Opening DSDs
  DB_RELATED_GROUPS ();
     DB_OPEN (DSD_STR,      OPEN_STR_ID,      STATUS_CODE);
     IF (STATUS_CODE = DB_OPEN_ERROR)  THEN
           ERROR (STATUS_CODE);
     DB_OPEN (DSD_ADDENDA, OPEN_ADDENDA_ID, STATUS_CODE);
     IF (STATUS_CODE = DB_OPEN_ERROR)  THEN
           ERROR (STATUS_CODE);
     DB_OPEN (DSD_DESC,     OPEN_DESC_ID,     STATUS_CODE);
     IF (STATUS_CODE = DB_OPEN_ERROR)  THEN
           ERROR (STATUS_CODE);
  DB_END_GROUPS ();

        (Body of ADL procedure)   Closing DSDs

  DB_CLOSE (OPEN_DESC_ID,     STATUS_CODE);
  IF (STATUS_CODE <> 0) THEN
        ERROR (STATUS_CODE);
  DB_CLOSE (OPEN_ADDENDA_ID, STATUS_CODE);
  IF (STATUS_CODE <> 0) THEN
        ERROR (STATUS_CODE);
  DB_CLOSE (OPEN_STR_ID,      STATUS_CODE);
  IF (STATUS_CODE <> 0) THEN
        ERROR (STATUS_CODE);
END;
```

F002-0584-00

Figure 5-2. Opening and Closing DSDs

Describing and Modifying Query Criteria

This section describes the Generic DML instructions that describe and modify query criteria:

- DB_RESET
- DB_QUERY
- DB_CLAUSE

At the end of this section, Figure 5-3 shows these instructions added to the example procedure. The DB_CLAUSE instruction is followed by its own example, since DB_CLAUSE cannot be used with all access methods.

DB_RESET (*open_id, status_code*);

DB_RESET prepares a DSD, or a hierarchy of DSDs, for new query criteria. DB_RESET can be issued only on a stand-alone DSD or the top-level DSD within a hierarchy. When there is a hierarchy of DSDs, this statement resets the query criteria on all the subordinate DSDs. Although DSD fields can still be accessed, no records can be retrieved from any of the subordinate DSDs until you issue a DB_GET_FIRST at each level to establish the new subset of records that is valid.

DB_RESET must be followed by DB_QUERY so that DML record operations can be performed.

DB_RESET typically is used prior to setting a search value or setting new query-clause criteria for subsequent record retrieval. After you have issued this statement, you can set new search criteria in two ways.

- You can assign a search value to a DSD field. New field value assignments are recognized as search criteria only when they are placed between a DB_RESET and a DB_QUERY. You use the usual ADL assignment statement syntax: "dsd_name.field_name := search_value;".

- You can use DB_CLAUSE to change the query criteria when the underlying access method supports a query clause. If supported, this clause must be in the syntax of the access method. Further, the fields and record types referenced in the clause must be those of the access method (the names could be different from the ones used in ALLY). The DB_CLAUSE instruction is described below.

DB_RESET can generate an error message if:

- the open_id variable is incorrect for the DSD or the DSD has not been opened
- you use a previous DB_RELATED_GROUPS but no previous DB_END_GROUPS
- you use DB_RESET on a subordinate DSD

DB_QUERY (*open_id, status_code*);

DB_QUERY executes a query that you have specified. The query is for the combination of:

- all selection criteria specified in the DSD definition
- all equality field query matches that you have set previously
- the most recent DB_CLAUSE you have set

All of these criteria are set until the next DB_RESET—even if you assign different values to some DSD fields.

DB_QUERY can be issued only for a stand-alone DSD or for the top-level DSD of a hierarchy of related DSDs. A DB_QUERY on the top DSD in a hierarchy executes the query for each subordinate DSD.

DB_QUERY can generate an error message if:

- the open_id variable is incorrect for the DSD or the DSD has not been opened
- you use a previous DB_RELATED_GROUPS but no previous DB_END_GROUPS
- you do not use a DB_RESET prior to a DB_QUERY

DB_CLAUSE (*open_id, query_clause, status_code*);

DB_CLAUSE can be used only with access methods that support a query clause syntax. For example, relational access methods support this statement, while the ALLY FX access method does not. Because the query clause is access-method specific, it may not be portable to other access methods.

DB_CLAUSE alters the initial selection criteria for the specified DSD. It appends to the initial criteria a new query clause. If no query condition exists in the DSD, this clause is set as the first one. Any appending is done with a logical *AND* instruction.

The query clause must be the query criteria itself or a variable of CHAR data type that has been assigned the value of the query clause. The query clause must follow the syntax required by the underlying access method.

The selection criteria of each DB_CLAUSE overlays the criteria of any previous one. You can add only one arbitrary query clause to the initial criteria. You can, however, build a selection string of greater length by using regular ADL procedures. After building the string in ADL, you pass it as a unit to the underlying access method.

The new criteria is not executed against the access method until the query is performed with a DB_QUERY (and, for some access methods, until the first record is retrieved with a DB_GET_FIRST).

You cannot use DB_CLAUSE until you have executed a DB_RESET on a given DSD.

Figure 5-3 shows how DB_CLAUSE is used to search for records in which the value of the "state" field is "NC". This example assumes that DSD_EMPLOYEES supports the type of query clause illustrated.

```
VAR
  STATUS_CODE              : NUMBER;
  OPEN_EMPLOYEES_ID        : NUMBER;

BEGIN

     DB_OPEN (DSD_EMPLOYEES, OPEN_EMPLOYEES_ID,
              STATUS_CODE);
     IF (STATUS_CODE = DB_OPEN_ERROR)  THEN
           ERROR (STATUS_CODE);

  DB_RESET (OPEN_EMPLOYEES_ID, STATUS_CODE);
  IF (STATUS_CODE <> 0)
     THEN ERROR (STATUS_CODE);
  DB_CLAUSE (OPEN_EMPLOYEES_ID, 'state="NC"',
             STATUS_CODE);
  DB_QUERY (OPEN_EMPLOYEES_ID, STATUS_CODE);
  IF (STATUS_CODE <> 0)
     THEN ERROR (STATUS_CODE);
  DB_GET_FIRST (OPEN_EMPLOYEES_ID, STATUS_CODE);
  IF (STATUS_CODE <> 0)
     THEN ERROR (STATUS_CODE);

        (Remainder of ADL Procedure)

  DB_CLOSE (OPEN_EMPLOYEES_ID, STATUS_CODE);
  IF (STATUS_CODE <> 0) THEN
        ERROR (STATUS_CODE);
END;
```

Figure 5-3. Procedure With DB_CLAUSE

Example Illustrating Instructions that Describe and Modify Query Criteria

Our example is expanded to include ADL instructions that set up the query so that DSD_STR is searched for records that contain a 'B' in the "REC_TYPE" field.

```
VAR
  STATUS_CODE      : NUMBER;
  OPEN_STR_ID      : NUMBER;
  OPEN_ADDENDA_ID  : NUMBER;
  OPEN_DESC_ID     : NUMBER;

BEGIN

  DB_RELATED_GROUPS ();
     DB_OPEN (DSD_STR,      OPEN_STR_ID,      STATUS_CODE);
     IF (STATUS_CODE = DB_OPEN_ERROR)  THEN
           ERROR (STATUS_CODE);
     DB_OPEN (DSD_ADDENDA, OPEN_ADDENDA_ID, STATUS_CODE);
     IF (STATUS_CODE = DB_OPEN_ERROR)  THEN
           ERROR (STATUS_CODE);
     DB_OPEN (DSD_DESC,     OPEN_DESC_ID,     STATUS_CODE);
     IF (STATUS_CODE = DB_OPEN_ERROR)  THEN
           ERROR (STATUS_CODE);
  DB_END_GROUPS ();

  DB_RESET (OPEN_STR_ID, STATUS_CODE);
  IF (STATUS_CODE <> 0) THEN
        ERROR (STATUS_CODE);
  {query DSD_STR for B   }
  {in "record type" field}
  DSD_STR.REC_TYPE := 'B';
  DB_QUERY (OPEN_STR_ID, STATUS_CODE);
  IF (STATUS_CODE <> 0) THEN
        ERROR (STATUS_CODE);

        (Remainder of ADL procedure)
  DB_CLOSE (OPEN_DESC_ID,     STATUS_CODE);
  IF (STATUS_CODE <> 0) THEN
        ERROR (STATUS_CODE);
  DB_CLOSE (OPEN_ADDENDA_ID, STATUS_CODE);
  IF (STATUS_CODE <> 0) THEN
        ERROR (STATUS_CODE);
  DB_CLOSE (OPEN_STR_ID,      STATUS_CODE);
  IF (STATUS_CODE <> 0) THEN
        ERROR (STATUS_CODE);
END;
```

F002-0575-00

Figure 5-4. Querying Records

Retrieving DSD Records

This section contains the descriptions of the Generic DML instructions that retrieve the records from an access method. Figure 5-5 shows these instructions added to the example procedure.

- DB_GET_FIRST
- DB_GET_NEXT

DB_GET_FIRST (*open_id, status_code*);

DB_GET_FIRST retrieves from the specified DSD the first record that matches any current selection criteria and group relationships. For a subordinate DSD in a hierarchy, the selection criteria used are:

- those set explicitly for that DSD, and
- those that define that DSD's relationship to the higher-level DSDs in its group (e.g., Foreign Key Links, Breakups).

You may issue DB_GET_FIRST for any DSD. However, if the DSD is a subordinate one, you must have issued a previous DB_GET_FIRST for any higher-level DSD in that group. When another record is retrieved for an upper-level DSD, a new chain of subordinate records must be established and so another DB_GET_FIRST for each subordinate DSD is required.

If you use this instruction immediately after opening the DSD, without a preceding DB_RESET and DB_QUERY or DB_CLAUSE, the search starts with the first record of the DSD using the query criteria specified in the definition of the DSD.

If no records match the given criteria, the status_code is set to an end-of-file status. You must issue another DB_GET_FIRST for each upper-level DSD in order to establish the new subordinate records that are valid.

DB_GET_FIRST can generate an error message if:

- the open_id is incorrect for the DSD or the DSD has not been opened
- you use a previous DB_RELATED_GROUPS but no previous DB_END_GROUPS
- you do not use DB_QUERY after DB_RESET and before DB_GET_FIRST
- you do not use a previous DB_GET_FIRST on any higher DSDs in the hierarchy
- no record is found because it is the end of the file

DB_GET_NEXT (*open_id, status_code*);

DB_GET_NEXT retrieves the next record from the DSD you specify. The next record is determined by the current record and by the original query criteria. The current record and query criteria are used as long as there is no change in the current record of any upper-level related DSD. If the current record is changed, a new chain of any valid subordinate records must be established. Therefore another DB_GET_FIRST is required.

DB_GET_NEXT may be issued for any DSD. However, when it is issued on a DSD in a hierarchy, you must precede it with a DB_GET_FIRST for the specified DSD and for any higher-level DSD in the hierarchy.

DB_GET_NEXT can generate an error message if:

- the open_id variable is incorrect for the DSD or the DSD has not been opened
- you use a previous DB_RELATED_GROUPS but no previous DB_END_GROUPS
- you do not use a DB_QUERY after a DB_RESET
- you do not use a previous DB_GET_FIRST
- you do not use a previous DB_GET_FIRST on any higher DSDs in the hierarchy
- you have already reached the end of the file

Example Illustrating DML Instructions that Retrieve Records

Figure 5-5 includes the DML instructions that can retrieve the records to be searched for the query.

```
VAR
  STATUS_CODE      : NUMBER;
  OPEN_STR_ID      : NUMBER;
  OPEN_ADDENDA_ID  : NUMBER;
  OPEN_DESC_ID     : NUMBER;

BEGIN
  DB_RELATED_GROUPS ();
     DB_OPEN (DSD_STR,      OPEN_STR_ID,      STATUS_CODE);
     IF (STATUS_CODE = DB_OPEN_ERROR)  THEN
            ERROR (STATUS_CODE);
     DB_OPEN (DSD_ADDENDA, OPEN_ADDENDA_ID, STATUS_CODE);
     IF (STATUS_CODE = DB_OPEN_ERROR)  THEN
            ERROR (STATUS_CODE);
     DB_OPEN (DSD_DESC,     OPEN_DESC_ID,     STATUS_CODE);
     IF (STATUS_CODE = DB_OPEN_ERROR)  THEN
            ERROR (STATUS_CODE);
  DB_END_GROUPS ();
  DB_RESET (OPEN_STR_ID, STATUS_CODE);
  IF (STATUS_CODE <> 0) THEN
        ERROR (STATUS_CODE);
  DSD_STR.REC_TYPE := 'B';
  DB_QUERY (OPEN_STR_ID, STATUS_CODE);
  IF (STATUS_CODE <> 0) THEN
        ERROR (STATUS_CODE);
  DB_GET_FIRST (OPEN_STR_ID, STATUS_CODE);
  IF (STATUS_CODE <> 0) THEN
        ERROR (STATUS_CODE);
  (Remainder of ADL procedure)
  DB_GET_NEXT (OPEN_STR_ID, STATUS_CODE);
  IF (STATUS_CODE = DB_DUPLICATE_RECORD)  THEN
        ERROR (STATUS_CODE);
  DB_CLOSE (OPEN_DESC_ID,     STATUS_CODE);
  IF (STATUS_CODE <> 0) THEN
        ERROR (STATUS_CODE);
  DB_CLOSE (OPEN_ADDENDA_ID, STATUS_CODE);
  IF (STATUS_CODE <> 0) THEN
        ERROR (STATUS_CODE);
  DB_CLOSE (OPEN_STR_ID,      STATUS_CODE);
  IF (STATUS_CODE <> 0) THEN
        ERROR (STATUS_CODE);
END;
```

F002-0576-00

Figure 5-5. Retrieving Records

Modifying Access-Method Records

This section describes the Generic DML instructions that allow you to make updates, deletions, and insertions to access-method records.

- DB_UPDATE
- DB_DELETE
- DB_INSERT

DB_UPDATE (*open_id, status_code*);

DB_UPDATE updates the last record retrieved or inserted for the specified DSD. That record is updated with the data values your ADL procedure has changed. DB_UPDATE may be issued for any DSD. However, when a record of an upper-level DSD is updated, there must be a preceding DB_GET_FIRST or DB_GET_NEXT for the specified DSD and any upper-level DSDs within a hierarchy.

An update must obey all rules for an access method. Such rules might include restrictions on key values (i.e., a primary key). If an update does not obey these rules, ALLY returns an error number to the status code for the appropriate DSD.

DSDs defined to be read-only will not support this operation. Breakup Definitions do not support insert, update, or delete operations because their definition is the result of a join-type operation.

DB_UPDATE can generate an error message if:

- the open_id variable is incorrect for the DSD or the DSD has not been opened
- you use a previous DB_RELATED_GROUPS but no previous DB_END_GROUPS
- you do not use a DB_QUERY or DB_GET_FIRST after DB_RESET

- there is no record present
- you try to update a record in a Breakup DSD

DB_DELETE (*open_id, status_code*);

DB_DELETE deletes the last record retrieved or inserted for the specified DSD. This instruction may be issued for any DSD. However, when a record of an upper-level DSD is deleted, there must be a DB_GET_FIRST preceding any instructions for a subordinate DSD to establish the subordinate records that are valid.

The delete operation must obey any rules for an access method. After you delete a record from a DSD, you must establish a new chain of valid subordinate records. Another DB_GET_FIRST is required for each subordinate DSD before any further operation can be performed on them.

Depending on the access method, it may still be valid to issue a DB_GET_NEXT after a DB_DELETE.

DSDs defined as read-only will not support this operation. Breakup Definitions do not support insert, update, or delete operations since their definition is the result of a join-type operation.

DB_DELETE can generate an error message if:

- the open_id variable is incorrect for the DSD or the DSD has not been opened
- you use a previous DB_RELATED_GROUPS but no previous DB_END_GROUPS
- you do not use a DB_QUERY or DB_GET_FIRST after a DB_RESET
- you do not use a previous DB_GET_FIRST on any higher DSDs in the hierarchy
- you try to delete a record from a Breakup DSD

DB_INSERT (*open_id, status_code*);

DB_INSERT creates a new record in the access method underlying the DSD specified by the open_id. The fields of that record initially contain the default values defined for the DSD. You may assign values to the DSD fields prior to the insert and override the default values.

Some access methods place importance on the position of each record relative to another. For a DSD used with these access methods, the new record will be positioned *after the last record for the DSD that was retrieved or updated.*

DB_INSERT requires only that any upper-level related DSDs contain a record. This requirement is necessary so that ALLY can determine the valid chain of subordinate records.

Since this instruction changes the current record, you must issue a DB_GET_FIRST at each subordinate level to establish the new subset of records that is valid. The type of access method determines whether you can successfully do a DB_GET_NEXT after a record insertion. (In any case, you must have done a DB_GET_FIRST before a DB_GET_NEXT.)

DSDs defined as read-only will not support this operation. Breakup Definitions do not support insert, update, or delete operations, because their definition is the result of a join-type operation.

DB_INSERT can generate an error message if:

- the open_id variable is incorrect for the DSD or the DSD has not been opened
- you use a previous DB_RELATED_GROUPS but no previous DB_END_GROUPS
- you do not use a previous DB_GET_FIRST on any higher DSDs in the hierarchy
- you do not use a DB_QUERY or DB_GET_FIRST after a DB_RESET
- you try to insert a record into a Breakup DSD

Example Illustrating DML Instructions that Modify Access-Method Records

Figure 5-6 illustrates the Generic DML instructions that allow you to modify the records of the access method.

```
VAR
  STATUS_CODE      : NUMBER;
  OPEN_STR_ID      : NUMBER;
  OPEN_ADDENDA_ID  : NUMBER;
  OPEN_DESC_ID     : NUMBER;

BEGIN

  DB_RELATED_GROUPS ();
     DB_OPEN (DSD_STR,      OPEN_STR_ID,      STATUS_CODE);
     IF (STATUS_CODE = DB_OPEN_ERROR)  THEN
        BEGIN
            ERROR (STATUS_CODE);
            RETURN;
        END;
     DB_OPEN (DSD_ADDENDA, OPEN_ADDENDA_ID, STATUS_CODE);
     IF (STATUS_CODE = DB_OPEN_ERROR)  THEN
        BEGIN
            ERROR (STATUS_CODE);
            RETURN;
        END;
     DB_OPEN (DSD_DESC,     OPEN_DESC_ID,     STATUS_CODE);
     IF (STATUS_CODE = DB_OPEN_ERROR)  THEN
        BEGIN
            ERROR (STATUS_CODE);
            RETURN;
        END;
  DB_END_GROUPS ();
  DB_RESET (OPEN_STR_ID, STATUS_CODE);
  IF (STATUS_CODE <> 0) THEN
         ERROR (STATUS_CODE);
  {query all DSDs for B}
  {in record type field}
  DSD_STR.REC_TYPE := 'B';
  DB_QUERY (OPEN_STR_ID, STATUS_CODE);
  IF (STATUS_CODE <> 0) THEN
         ERROR (STATUS_CODE);
  DB_GET_FIRST (OPEN_STR_ID, STATUS_CODE);
  IF (STATUS_CODE <> 0) THEN
         ERROR (STATUS_CODE);

  WHILE (STATUS_CODE <> DB_EOF) DO
     BEGIN
         DB_GET_FIRST (OPEN_ADDENDA_ID, STATUS_CODE);
         IF (STATUS_CODE <> 0) THEN                    continued
```

```
                ERROR (STATUS_CODE);
  {assign field values }
        DSD_DESC.FLD1 := DSD_ADDENDA.FLD1;
     {from ADDENDA to DESC}
        DSD_DESC.FLD2 := DSD_ADDENDA.FLD2;

  {insert new record}
        DB_INSERT (OPEN_DESC_ID, STATUS_CODE);
        IF (STATUS_CODE = DB_DUPLICATE_RECORD)  THEN
              ERROR (STATUS_CODE);
  {delete record with copied fields from ADDENDA}
        DB_DELETE (OPEN_ADDENDA_ID, STATUS_CODE);
        IF (STATUS_CODE <> 0) THEN
              ERROR (STATUS_CODE);
           {assign C to record type}
        DSD_STR.REC_TYPE := 'C';
  {update STR}
        DB_UPDATE (OPEN_STR_ID, STATUS_CODE);
        IF (STATUS_CODE <> 0) THEN
              ERROR (STATUS_CODE);

        DB_GET_NEXT (OPEN_STR_ID, STATUS_CODE);

        (Remainder of ADL Procedure)

     END;
  DB_CLOSE (OPEN_DESC_ID,     STATUS_CODE);
  IF (STATUS_CODE <> 0) THEN
        ERROR (STATUS_CODE);
  DB_CLOSE (OPEN_ADDENDA_ID, STATUS_CODE);
  IF (STATUS_CODE <> 0) THEN
        ERROR (STATUS_CODE);
  DB_CLOSE (OPEN_STR_ID,      STATUS_CODE);
  IF (STATUS_CODE <> 0) THEN
        ERROR (STATUS_CODE);
END;
```

F002-0577-00

Figure 5-6. Modifying Records

Performing Access-Method Transactions

Access-method transactions are performed from the last commit or rollback to the next one. The open of the procedure is the implied first commit. The instructions that perform access method transactions are:

- DB_COMMIT
- DB_ROLLBACK

DB_COMMIT (*status_code*);

DB_COMMIT causes all record changes since the last commit or rollback to be saved. This instruction affects each DSD open in any ADL procedure in the current task.

DB_COMMIT performs a commit to the underlying access method for each open DSD that has had at least one modification since the last commit or rollback. DSDs that have not been modified are not committed (a commit is unnecessary in such cases).

DB_COMMIT can generate an error message if no DSD is open for the commit.

DB_ROLLBACK (*status_code*);

DB_ROLLBACK removes all record changes made since the last commit or rollback to open DSDs in ADL procedures in the current task. These changes are rolled back only to the extent supported by the underlying access method.

DB_ROLLBACK can generate an error message if no DSD is open for the rollback.

Example Illustrating DML Instructions that Perform Access-Method Transactions

Figure 5-7 shows the DML transaction instructions added to the example procedure. We have added status variables to monitor the rollback and the commit operations.

```
VAR
  STATUS_ROLLBACK : NUMBER;        {monitor DB_ROLLBACK}
  STATUS_COMMIT   : NUMBER;        {monitor DB_COMMIT}
  STATUS_CODE     : NUMBER;
  OPEN_STR_ID     : NUMBER;
  OPEN_ADDENDA_ID : NUMBER;
  OPEN_DESC_ID    : NUMBER;

BEGIN

  DB_RELATED_GROUPS ();
     DB_OPEN (DSD_STR,      OPEN_STR_ID,      STATUS_CODE);
     IF (STATUS_CODE = DB_OPEN_ERROR)  THEN
        BEGIN
            ERROR (STATUS_CODE);
            RETURN;
        END;
     DB_OPEN (DSD_ADDENDA, OPEN_ADDENDA_ID, STATUS_CODE);
     IF (STATUS_CODE = DB_OPEN_ERROR)  THEN
        BEGIN
            ERROR (STATUS_CODE);
            RETURN;
        END;
     DB_OPEN (DSD_DESC,     OPEN_DESC_ID,     STATUS_CODE);
     IF (STATUS_CODE = DB_OPEN_ERROR)  THEN
        BEGIN
            ERROR (STATUS_CODE);
            RETURN;
        END;
  DB_END_GROUPS ();
  DB_RESET (OPEN_STR_ID, STATUS_CODE);
  IF (STATUS_CODE <> 0) THEN
        ERROR (STATUS_CODE);
  DSD_STR.REC_TYPE := 'B';
  DB_QUERY (OPEN_STR_ID, STATUS_CODE);
  IF (STATUS_CODE <> 0) THEN
        ERROR (STATUS_CODE);
  DB_GET_FIRST (OPEN_STR_ID, STATUS_CODE);
  IF (STATUS_CODE <> 0) THEN
        ERROR (STATUS_CODE);

  WHILE (STATUS_CODE <> DB_EOF) DO
     BEGIN
        DB_GET_FIRST (OPEN_ADDENDA_ID, STATUS_CODE);
        IF (STATUS_CODE <> 0) THEN
```

continued

```
                ERROR (STATUS_CODE);
          DSD_DESC.FLD1 := DSD_ADDENDA.FLD1;
          DSD_DESC.FLD2 := DSD_ADDENDA.FLD2;
          DB_INSERT (OPEN_DESC_ID, STATUS_CODE);
          IF (STATUS_CODE = DB_DUPLICATE_RECORD)  THEN
                ERROR (STATUS_CODE);
          DB_DELETE (OPEN_ADDENDA_ID, STATUS_CODE);
          IF (STATUS_CODE <> 0) THEN
                ERROR (STATUS_CODE);
          DSD_STR.REC_TYPE := 'C';
          DB_UPDATE (OPEN_STR_ID, STATUS_CODE);

          IF (STATUS_CODE <> 0) THEN
                BEGIN
                    ERROR (STATUS_CODE);
                    DB_ROLLBACK (STATUS_ROLLBACK);
          {on error, rollback record changes}
                    IF (STATUS_ROLLBACK > 0) THEN
                       ERROR (STATUS_ROLLBACK);
                END;
             ELSE
                BEGIN
                {commit changes}
                   DB_COMMIT (STATUS_COMMIT);
                   IF (STATUS_ROLLBACK > 0) THEN
                       ERROR (STATUS_COMMIT);
                END;

          DB_GET_NEXT (OPEN_STR_ID, STATUS_CODE);
       END;

    DB_CLOSE (OPEN_DESC_ID,     STATUS_CODE);
    IF (STATUS_CODE <> 0) THEN
          ERROR (STATUS_CODE);
    DB_CLOSE (OPEN_ADDENDA_ID, STATUS_CODE);
    IF (STATUS_CODE <> 0) THEN
          ERROR (STATUS_CODE);
    DB_CLOSE (OPEN_STR_ID,      STATUS_CODE);
    IF (STATUS_CODE <> 0) THEN
          ERROR (STATUS_CODE);
END;
```

F002-0578-00

Figure 5-7. Performing Transactions

Passing Access-Method-Specific Commands

ADL provides a Generic DML instruction (DB_COMMAND) that allows you to send a Data Definition Language (DDL) command directly to an access method. Many access methods (e.g., the ALLY FX access method and most ISAMs) do not support DDL commands.

DB_COMMAND (*db_system_name, ddl_command_string, status_code*);

DB_COMMAND allows you to send DDL statements directly from ADL to an underlying access method. Any legal DDL command may be sent to an access method. Such commands must follow the syntax of the underlying access method and must be DDL commands. DML instructions are not allowed.

As part of its normal operation, ALLY will log you on to all access methods used in an application at the time the application is started. Therefore, you need not open any DSD.

The *db_system_name* argument must be one of the pre-defined constants that identify an underlying access method that supports pass-through DDL commands.

The *ddl_command_string* argument is access-method specific. It must be the actual DDL command or the name of a variable of CHAR data type that is assigned the value of the DDL command string. This string must follow the syntax required by the underlying access method. This must be a legal DDL command. Illegal DDL commands will raise an error.

This command will not corrupt any ALLY DSD status or other state. Therefore, at runtime, ALLY verifies that the command string is a legal DDL command.

When ALLY has determined that the command string is a DDL command, that command is passed directly to the underlying access method via that access method's standard call-level interface.

DB_COMMAND allows you to make runtime decisions about the structure of the underlying access method or file. While tables may be created dynamically, DSDs may not be. Therefore, it is possible to create a physical environment that is not immediately usable from ALLY since a required DSD has not yet been created.

End of Chapter 5

Appendix A
ADL Reserved Words

Table A-1, Table A-2, and Table A-3 list the ADL reserved words, the ADL functions and instructions, and the ALLY commands. Each ADL function and instruction is described in the alphabetical syntax listing that follows the tables. Command arguments show the data type or item required. Optional arguments are labeled. ALLY commands are described individually in the *ALLY Command Reference Manual.*

Table A-1. ADL Reserved Words

ABORTACTION
ABORTAPPL
ABORTTASK
ADD_MONTHS
ADDNL
AND
ARRAY*

BDELETE
BEGIN
BOL
BOTTOM
BOX
BUDMODE

CALL
CALL_CMD
CASE*
CHAR
CLRCASESENS
CLRDRAWMODE
CLROVERTYPE
CLRPWRTYPE
COMMIT
COMPRESSWDW
CONST
CPTOBUF
CTRLCHAR

DATE
DB_CLAUSE
DB_CLOSE
DB_COMMAND
DB_COMMIT
DB_DELETE
DB_DUPLICATE_RECORD
DB_END_GROUPS
DB_EOF
DB_GET_FIRST
DB_GET_NEXT
DB_INSERT
DB_OPEN
DB_OPEN_ERROR
DB_QUERY
DB_RELATED_GROUPS
DB_RESET
DB_ROLLBACK
DB_UPDATE

DEFMACRO
DELBOL
DELEOL
DELLINE
DELREC
DELTOMARK

DO
DOWN
DOWNPAGE
DUPDATE

ELSE
EOL
ERROR
EXECUTE
EXECUTE_CMD
EXEMACn #
EXEMACF
EXITACTION

EXITTASK
EXPANDWDW
EXPLODEWDW
EXPORT

FALSE*
FDELETE
FHOME
FIND
FINDANDDEL
FINSNEXT
FLAST
FLISTVAL
FNEXT
FOR*
FORK
FPICKVAL
FPREV
FRFUNCTION
DEFINEWDW

GET_CMD
GLBLREPLACE
GLOBAL
GOTO*

DELWORD
HIGHTOMARK
HIGHTYPESET
HOME
HOMEMCH

IF
IGNORE
IMPORT
INSAFTER
INSBEFORE
INSERTLINE
ISNOTNULL*
ISNULL*
EXITAPPL
JUMPTOMARK

KHELP
KMPPRINT

LAST_DAY
LDTOMARK
LOADMACROS
LOCAL

MACFMFILE
MACTOFILE

MAKE_NULL	QBE	SHELL
MARK	QUERY	START
MENU	QWHERE	SUBSTR
MOD*		
MONTHS_BETWEEN	READFILE	TASKn @
MOVEWDW	REDRAW	TERMINATOR
		REFRESH
NEXT_DAY	REMOVEBLK	TO_CHAR
NEXTLINE	REPEAT*	TO_DATE
NEXTMCH	REPLACE	TOGCASESENS
NEXTWORD	RESIZEWDW	TOGDRAWMODE
NIL*	RESUME	TOGGLETASK
NOT	RETURN	TOGOVERTYPE
NUMBER	RETURN_TO	TOGPWRTYPE
		RGHOME
OR	RGLAST	TOP
OVERLAYBLK	RGNEXT	TOPMENU
		RGPREV
PALL	RHOME	TRUNC
PHOME	RIGHT	TURTLECLEAR
PICKFIELD	RLAST	TURTLEHL
PICKTASK	RNEXT	TURTLELD
PLAST	ROAMFIRST	
PNEXT	ROAMLAST	ULDTOMARK
PPAGE	ROLLBACK	ULDTURTLE
PPREV	RPREV	UNBOX
PREST		UNDELLINE
PREVMCH	SAVE	UNDELWORD
PREVMENU	SAVEMACROS	UNTIL*
PREVWORD	SCROLLWDW	UP
PRHOME	SELECT	UPPAGE
PRLAST	SETCASESENS	
PRNEXT	SETDELAYCNT	VAR
PRNTSCRN	SETDRAWMODE	
PRNTVNUM	SET_FAILURE	WHILE
PROCEDURE	SETOVERTYPE	WINDONE
PROMPT	SETPWRTYPE	WINDOWN
PRPREV	SETRPTCNT	WINLEFT
PUTFIELD	SET_SUCCESS	WINRIGHT
		WINUP

* Not yet implemented
# Where n = 0-29
@ Where n = 1-255

Table A-2. ADL Functions and Instructions

ADD_MONTHS+	ELSE	NEXT_DAY+
AND	END	NIL*
ARRAY*	ERROR	NOT
	EXECUTE	NUMBER
BEGIN	EXECUTE_CMD	
	EXPORT	OR
CALL		
CALL_CMD	FALSE*	PROCEDURE
CASE*	FOR*	
CHAR	FORK	REPEAT*
CONST	FROM*	RESUME
		RETURN
DATE	GET_CMD+	RETURN_TO
DB_CLAUSE	GLOBAL	ROUND+
DB_CLOSE	GOTO*	
DB_COMMAND		SET_FAILURE
DB_COMMIT	HELP	SET_SUCCESS
DB_DELETE		START
DB_END_GROUPS	IF	SUBSTR
DB_DUPLICATE_RECORD	IMPORT	
DB_EOF	ISNOTNULL*	THEN
DB_GET_FIRST	ISNULL*	TO_CHAR+
DB_GET_NEXT		TO_DATE+
DB_INSERT	LAST_DAY+	TO_NUMBER+
DB_OPEN	LOCAL	TRUE*
DB_OPEN_ERROR		TRUNC+
DB_QUERY	MAKE_NULL	
DB_RELATED_GROUPS	MOD*	UNTIL*
DB_RESET	MONTHS_BETWEEN+	
DB_ROLLBACK		VAR
DB_UPDATE		
DO		WHILE

+ ADL function.
* Not yet implemented

Table A-3. ALLY Commands

Command Name	Command Number@	Purpose
* These commands are used in ADL only as arguments for the ADL instructions CALL_CMD and EXECUTE_CMD.		
@ You can use this number to reference a command returned by GET_CMD.		
ABORTACTION	1100	Abort action
ABORTAPPL	1104	Abort application
ABORTTASK	1102	Abort task
ADDNL	1506	Add new line
BDELETE	1534	Back delete
BOL	1511	Beginning of line
BOTTOM	1510	Bottom
BOX	1526	Box
BUDMODE	6004	Browse, update, delete mode
CLRCASESENS	1549	Clear case sensitive
CLRDRAWMODE	1552	Clear draw mode
CLROVERTYPE	1544	Clear overtype
CLRPWRTYPE	1546	Clear powertype
COMMIT	6013	Commit
COMPRESSWDW	1121	Compress window
CPFROMBUF	1524	Copy from buffer
CPTOBUF	1542	Copy to buffer
CTRLCHAR	1540	Enter control character
DEFINEWDW	1123	Define window
DEFMACRO	1140	Define macro
DELBOL	1512	Delete to beginning of line
DELEOL	1514	Delete to end of line
DELLINE	1531	Delete line
DELREC	6008	Delete current record
DELTOMARK	1523	Delete to mark
DELWORD	1521	Delete word
DOWN	1501	Down
DOWNPAGE	1507	Down page
DUPDATE	6015	Deferred update
EOL	1513	End of line
EXEMAC0	1150	Execute macro 0
EXEMAC1	1151	Execute macro 1
EXEMAC2	1152	Execute macro 2

Command Name	Command Number@	Purpose
* These commands are used in ADL only as arguments for the ADL instructions CALL_CMD and EXECUTE_CMD.		
@ You can use this number to reference a command returned by GET_CMD.		
EXEMAC3	1153	Execute macro 3
EXEMAC4	1154	Execute macro 4
EXEMAC5	1155	Execute macro 5
EXEMAC6	1156	Execute macro 6
EXEMAC7	1157	Execute macro 7
EXEMAC8	1158	Execute macro 8
EXEMAC	1159	Execute macro 9
EXEMAC10	1161	Execute macro 10
EXEMAC11	1162	Execute macro 11
EXEMAC12	1163	Execute macro 12
EXEMAC13	1164	Execute macro 13
EXEMAC14	1165	Execute macro 14
EXEMAC15	1166	Execute macro 15
EXEMAC16	1167	Execute macro 16
EXEMAC17	1168	Execute macro 17
EXEMAC18	1169	Execute macro 18
EXEMAC19	1170	Execute macro 19
EXEMAC20	1171	Execute macro 20
EXEMAC21	1172	Execute macro 21
EXEMAC22	1173	Execute macro 22
EXEMAC23	1174	Execute macro 23
EXEMAC24	1175	Execute macro 24
EXEMAC25	1176	Execute macro 25
EXEMAC26	1177	Execute macro 26
EXEMAC27	1178	Execute macro 27
EXEMAC28	1179	Execute macro 28
EXEMAC29	1180	Execute macro 29
EXEMACF	1160	Execute macro from file
EXITACTION	1101	Exit action
EXITAPPL	1105	Exit application
EXITTASK	1103	Exit task
EXPANDWDW	1120	Expand window
EXPLODEWDW	1125	Explode window
FDELETE	1530	Forward delete
FHOME	6140	First field
FIND	1529	Find
FINDANDDEL	1539	Find and delete
FINSNEXT	6022	Insert first record in next group

Command Name	Command Number@	Purpose
* These commands are used in ADL only as arguments for the ADL instructions CALL_CMD and EXECUTE_CMD.		
@ You can use this number to reference a command returned by GET_CMD.		
FLAST	6141	Last field
FLISTVAL	6144	Move to list of values
FNEXT	6142	Next field
FPICKVAL	6021	Pick from list of values
FPREV	6143	Previous field
FRFUNCTION	6023	Invoke local function
GLBLREPLACE	1541	Global replace
HIGHTOMARK	1553	Highlight to mark
HIGHTYPESET	1555	Set highlight type
HOME	1504	Home
HOMEMCH	2004	Home area
IGNORE	1538	Ignore
INSAFTER	6002	Insert record after
INSBEFORE	6003	Insert record before
INSERTLINE	1532	Insert line
JUMPTOMARK	1536	Jump to mark
KHELP	1111	Help
KMPPRINT	3005	Print menu
LDTOMARK	1554	Line draw to mark
LEFT	1502	Left
LOADMACROS	1144	Load macros
MACFMFILE	1142	Macro from file
MACTOFILE	1141	Macro to file
MARK	1522	Set mark
MENU	2500	Function key choice
MOVEWDW	1122	Move window
NEXTLINE	1505	Next line
NEXTMCH	2001	Next area
NEXTWORD	1520	Next word
OVERLAYBLK	1562	Overlay block

Command Name	Command Number@	Purpose
* These commands are used in ADL only as arguments for the ADL instructions CALL_CMD and EXECUTE_CMD.		
@ You can use this number to reference a command returned by GET_CMD.		
PALL	6016	Print all
PHOME	6102	First page
PICKFIELD	6019	Copy to field-buffer
PICKTASK	1202	Pick task
PLAST	6103	Last page
PNEXT	6100	Next page
PPAGE	6017	Print page
PPREV	6101	Previous page
PREST	6018	Print rest
PREVMCH	2000	Previous area
PREVMENU	3002	Previous menu
PREVWORD	1518	Previous word
PRHOME	6130	First display area
PRLAST	6131	Last display area
PRNEXT	6132	Next display area
PRNTSCRN	1114	Print screen
PRNTVNUM	1115	Print version number
PROMPT	2005	Prompt line
PRPREV	6133	Previous display area
PUTFIELD	6020	Copy from field-buffer
QBE	6005	Query by example
QUERY	6007	Execute query
QWHERE	6006	Query by where clause
READFILE	1516	Read from file
REDRAW	1535	Redraw
REFRESH	1110	Refresh
REMOVEBLK	1561	Remove block
REPLACE	1528	Replace
RESIZEWDW	1126	Resize window
RGHOME	6110	First logical group
RGLAST	6111	Last logical group
RGNEXT	6112	Next logical group
RGPREV	6113	Previous logical group
RHOME	6120	First record
RIGHT	1500	Right
RLAST	6121	Last record

Command Name	Command Number@	Purpose
* These commands are used in ADL only as arguments for the ADL instructions CALL_CMD and EXECUTE_CMD.		
@ You can use this number to reference a command returned by GET_CMD.		
RNEXT	6122	Next record
ROAMFIRST	3000	First area
ROAMLAST	3001	Last area
ROLLBACK	6014	Rollback
RPREV	6123	Previous record
SAVE	1515	Save
SAVEMACROS	1143	Save macros
SCROLLWDW	1124	Scroll window
SELECT	2006	Choose from roam area
SETCASESENS	1548	Set case sensitive
SETDELAYCNT	1116	Pause
SETDRAWMODE	1551	Set draw mode
SETOVERTYPE	1543	Set overtype
SETPWRTYPE	1545	Set powertype
SETRPTCNT	1113	Set repeat count
SHELL	1112	Go to OS command line processor
TASK	1210	Start task
TERMINATOR	3004	Choose from prompt line
TOGCASESENS	1547	Toggle case sensitive
TOGDRAWMODE	1550	Toggle draw mode
TOGGLETASK	1200	Toggle task
TOGOVERTYPE	1527	Toggle overtype
TOGPWRTYPE	1537	Toggle powertype
TOP	1509	Top
TOPMENU	3003	First menu
TURTLECLEAR	1558	Clear turtle
TURTLEHL	1556	Highlight with turtle
TURTLELD	1557	Line draw with turtle
ULDTOMARK	1559	Erase line draw
ULDTURTLE	1560	Erase line draw with turtle
UNBOX	1525	Unbox
UNDELLINE	1533	Undelete line
UNDELWORD	1519	Undelete word
UP	1503	Up
UPPAGE	1508	Up page

Command Name	Command Number@	Purpose
* These commands are used in ADL only as arguments for the ADL instructions CALL_CMD and EXECUTE_CMD.		
@ You can use this number to reference a command returned by GET_CMD.		
WINDONE	7504	Window-action
WINDOWN	7501	Window down
WINLEFT	7502	Window left
WINRIGHT	7503	Window right
WINUP	7500	Window up
WRITEFILE	1517	Write to file

Syntax of ADL Reserved Words

ADD_MONTHS (*DATE_argument, NUMBER_argument*);

Adds a NUMBER argument to a DATE argument. The result is a DATE.

(*expression*) **AND** (*expression*);

AND is a relational operator used in IF and WHILE condition statements. Its two arguments must be relational expressions enclosed within parentheses.

ARRAY

This is an ADL reserved word to which no action has been assigned.

BEGIN *procedure_statement_sequence* **END;**

BEGIN must precede procedure statements when the program starts with CONST or VAR. BEGIN is also required to execute multiple statements in IF or WHILE conditions. "END;" is required after procedure statements introduced by BEGIN.

CALL *ALLY_action_name*;

Invokes a specified ALLY action—a form/report packet, a menu, another ADL procedure, a parameter packet, an external link, an action list, or a text editor.

CALL_CMD (*ALLY_command_name*);

Invokes a specified ALLY command. The argument must be the name of an ALLY command or of a variable whose value is the name of an ALLY command. If the argument is a variable, the variable must have the data type NUMBER.

CASE

This is an ADL reserved word to which no action has been assigned.

variable_name : **CHAR;**

CHAR labels a variable that can contain character values.

CONST *constant_declaration_statement*;

CONST labels the constant declaration part of an ADL procedure. It is required only when a procedure uses as least one constant.

variable_name : **DATE;**

DATE labels a variable that can contain date values.

DB_CLAUSE (*open_id , query_clause, status_code*);

DML command used in record retrieval.

DB_CLOSE (*open_id, status_code*);

DML command to close DSDs.

DB_COMMAND (*db_system_string, command_string, status_code*);

DML command to pass DDL command through directly to underlying access method.

DB_COMMIT (*status_code*);

DML command to save changes to records.

DB_DELETE (*open_id, status_code*);

DML command to delete current record.

DB_DUPLICATE_RECORD

DML global constant signifying that this dataset, file, or table already contains a record with the same primary key value as the one you are attempting to insert.

DB_END_GROUPS ();

DML command that ends a related group definition.

DB_EOF

DML global constant signifying that you have reached the last record in the dataset, file, or table or that you have reached the last record in the subset that you are querying.

DB_GET_FIRST (*open_id, status_code*);

DML command to retrieve first record.

DB_GET_NEXT (*open_id, status_code*);

DML command to retrieve next record.

DB_INSERT (*open_id, status_code*);

DML command to insert a record.

DB_OPEN (*DSD_name, open_id, status_code*);

DML command to open DSDs.

DB_OPEN_ERROR

DML global constant signifying that the DSD named in the DB_OPEN is invalid or that you have used the DB_OPEN without a preceding DB_RELATED GROUPS.

DB_QUERY (*open_id, status_code*);

DML command to execute a query.

DB_RELATED_GROUPS ();

DML command to introduce related DSDs.

DB_RESET (*open_id, status_code*);

DML command to reset query status.

DB_ROLLBACK (*status_code*);

DML command to ignore record changes.

DB_UPDATE (*open_id, status_code*);

DML command to write current record.

DO *instruction_statement*;

Required for a WHILE statement. If more than one statement follows DO, they must be surrounded by BEGIN and END.

ELSE *instruction_statement*;

ELSE can be used in an IF condition to specify an alternative to the statement following IF. BEGIN must precede compound ELSE statements and END must follow them.

END;

Required after procedure statement(s) that are preceded by BEGIN.

ERROR (*NUMBER_argument*);

Displays the text of the error number argument.

EXECUTE *ALLY_action_name*;

Invokes a specified ALLY action—a form/report packet, a menu, another ADL procedure, a parameter packet, an external link, an action list, or a text editor.

EXECUTE_CMD (*ALLY_command_argument*);

Invokes a specified ALLY command. The argument must be the name of an ALLY command or of a variable whose value is the name of an ALLY command. If the argument is a variable, the variable must have the data type NUMBER.

variable_name : *data_type* **EXPORT;**

EXPORT labels a local variable whose value can be used in other ADL procedures.

FALSE

This is an ADL reserved word to which no action has been assigned.

FOR

This is an ADL reserved word to which no action has been assigned.

FORK *ALLY_task_name*;

Invokes a specified ALLY task.

FROM

This is an ADL reserved word to which no action has been assigned.

GET_CMD ();

Returns the last keystroke that has not yet been processed in a form/report.

variable_name : GLOBAL;

GLOBAL labels a variable that has been defined in this application as a global variable.

GOTO

This is an ADL reserved word to which no action has been assigned.

HELP (*NUMBER_argument*);

Displays the text of a specified help message number.

IF (*condition(s)*)

Introduces a condition statement.

variable_name : *data_type* **IMPORT** *name_of_form_report.field or variable*;

IMPORT labels a local variable whose value will be imported from another ADL procedure that is executing concurrently or from a form/report field in the application.

ISNOTNULL

This is an ADL reserved word to which no action has been assigned.

ISNULL

This is an ADL reserved word to which no action has been assigned.

LAST_DAY (*DATE_argument*);

Calculates the last day of the month for a DATE_argument. The result is a DATE value.

variable_name : *data_type* **LOCAL;**

LOCAL labels a variable whose value is visible only to the ADL procedure in which it is declared. ADL variables are LOCAL by default.

MAKE_NULL (*variable_or_field_name*);

Causes the value of a variable or field to have no value, i.e., to become null.

MOD

This ADL reserved word has not yet been implemented.

MONTHS_BETWEEN (*DATE_argument, DATE_argument*);

Calculates the number of months between the two date arguments. The result has the data type NUMBER.

NEXT_DAY (*DATE_argument, Day_of_week_argument*);

Calculates the next occurrence of a day_of_week argument. The result is a DATE value.

NIL

This is an ADL reserved word to which no action has been assigned.

NOT (*expression*);

NOT is a relational operator used in IF and WHILE condition statements. Its argument must be a relational expression.

variable_name : **NUMBER;**

NUMBER labels a variable that can contain number values.

(*expression*) **OR** (*expression*)

OR is a relational operator used in IF and WHILE condition statements. Its two arguments must be relational expressions.

PROCEDURE *identifier*
PROCEDURE *identifier* **(VAR** *parameter_name* : *data_type*;**);**

The PROCEDURE statement is optional in an ADL program. The identifier is the name of the ADL program.

REPEAT

This is an ADL reserved word to which no action has been assigned.

RESUME *ALLY_task_name*;

Invokes an ALLY task.

RETURN;

Terminates execution of an ADL procedure and returns to the event that called the procedure.

RETURN_TO *ALLY_action_name*;

Removes from the task's execution stack all actions back to the named action, then resumes execution of the called action.

ROUND (*DATE_argument, optional_date_picture*);
ROUND (*NUMBER_argument, optional_precision_argument*);

Rounds a DATE or NUMBER value to the precision you specify. The table listing the precisions for the rounding of dates and numbers is in Appendix B.

SET_FAILURE ();

Reports to the calling event that the preceding ADL procedure has failed. This instruction takes no argument.

SET_SUCCESS ();

Reports to the calling event that the preceding ADL procedure has succeeded. This instruction takes no argument.

START *ALLY_task_name*;

START invokes an ALLY task.

SUBSTR (*CHAR_variable, offset, length*);

Assigns to a variable a subset of the value of another variable. The offset and length can be numbers or NUMBER variables.

IF (condition(s)) **THEN**

THEN must follow the IF condition statement(s) and precede the imperative statement(s).

TO_CHAR (*DATE_argument , optional_date_picture*);
TO_CHAR (*NUMBER_argument);*

Converts both DATE and NUMBER values to CHAR values.

TO_DATE (*CHAR_argument, optional_date_picture*);

Converts a CHAR value to a DATE value with a date picture you specify. The result is a DATE value.

TO_NUMBER (*CHAR_argument*);

TO_NUMBER converts a CHAR value to a NUMBER value. The result is NUMBER data type.

TRUE

This is an ADL reserved word to which no action has been assigned.

TRUNC (*DATE_argument, optional_date_picture*);
TRUNC (*NUMBER_argument, optional_precision_argument*);

Truncates a DATE or NUMBER value to the precision you specify. The table listing the precisions for the truncation of dates and numbers is in Appendix B.

UNTIL

This is an ADL reserved word to which no action has been assigned.

VAR *variable_name* : *data_type*;

Labels variable declaration statement section of the procedure. Required when there is at least one variable listed.

WHILE *condition(s) DO statement(s)*;

Executes statement(s) while condition(s) is/are true. Introduces a condition statement. If the DO statement is to execute multiple statements, then they must be surrounded by BEGIN and END.

End of Appendix A

Appendix B
ADL Operators

Table B-1. ADL Operators

Operator Type	Operator	Operation
Assignment	:=	Assign value
Arithmetic	+	Addition
	−	Subtraction
	*	Multiplication
	/	Division
Relational	=	Equality
	<>	Inequality
	<	Less than
	>	Greater than
	<=	Less than or equal
	>=	Greater than or equal
Logical	NOT	Negation
	OR	Or
	AND	And

Table B-2. Precedence of Operators

NOT	Done first
*, /, AND	↓
+, −, OR	↓
<, <=, =, <>, >=, >	Done last

End of Appendix B

Appendix C
Data Types and DATE Pictures

This appendix contains tables of:

- data type conversion and DATE arithmetic
- DATE pictures for TO_DATE function
- DATE format pictures
- DATE picture precisions for rounding and truncating a date

Manipulating ADL Data Types

Table C-1 summarizes the functions and operations that ADL provides for manipulating data types.

Table C-1. ADL Data Type Manipulation

Beginning Data Type	Function Name or Action	Operation	Resulting Data Type
CHAR	TO_NUMBER	Converts value of container* to NUMBER	NUMBER
CHAR	TO_DATE	Converts value of container* to DATE	DATE
NUMBER	TO_CHAR	Converts value of container* to CHAR	CHAR
NUMBER	ROUND	Rounds to the precision specified	NUMBER

Beginning Data Type	Function Name or Action	Operation	Resulting Data Type
NUMBER	TRUNC	Truncates to the precision specified	NUMBER
DATE	Add days	Value of container* plus n**	DATE
DATE	Subtract days	Value of container* minus n**	DATE
DATE	Subtract dates	container*_for_date minus container*_for_date	NUMBER of days
DATE	ADD_MONTHS	Adds a number of months to a date	DATE
DATE	MONTHS_BETWEEN	Determines the number of months between two dates	NUMBER of months
DATE	LAST_DAY	Calculates last day in specified month	DATE
DATE	NEXT_DAY	Calculates the date of the next occurrence of the specified day of the week	DATE
DATE	ROUND	Rounds to the precision specified	DATE

Beginning Data Type	Function Name or Action	Operation	Resulting Data Type
DATE	TO_CHAR	Converts value of date container* to character data type	CHAR
DATE	TRUNC	Truncates to the precision specified	DATE

* Container can be a variable or a form/report field

** Where "n" is the number of days

Input DATE Pictures for TO_DATE Function

Input pictures are used for inputting dates with the TO_DATE function. To avoid a problem with input picture conflicts, only one entry from each of the sections of Table C-2 is permitted.

Table C-2. Input DATE Picture Classifications

Picture Class	Date Pictures
Y	YEAR SYEAR YYYY Y,YYY SYYYY SY,YYY YYY YY Y J CC SCC
Month	MONTH MON MM J Q DDD WW
Week	DD DDD J Q W WW
Day	DAY DY D
Hour	HH HH12 HH24 SSSSS
Minute	MM SSSSS
Second	SS SSSSS
Meridian	AM A.M. PM P.M. HH24 SSSSS
Era	AD A.D. BC B.C. SYEAR SYYYY SY,YYY J SCC

Notes on Input DATE Picture Classification Table

D, DY, and DAY may be input even when their presence is redundant. If the date given is appropriate for the day entered, the day is accepted. A conflict causes an error message to be displayed. For example, if the Julian date has been entered which corresponds to a Wednesday, then the symbol "Wed" (DY) will be accepted along with the Julian designation. If, however, the day entered is "Sat" (DY), the conflict with the Julian designation causes an error.

Capitalization is ignored. Except for MONTH, MON, DAY, and DY, spelled symbols are ignored. Punctuation and literals must match. All blank literals match all entries.

On input, ALLY does not verify that the symbols completely specify a date. It is possible for you to supply only a "time" symbol without a specific date, day of week, etc. If you enter an incomplete input symbol, ALLY supplies the default value (the current year, Jan. 1, at midnight) to fill in the missing portions of the date.

White space (one or more blanks) on input is treated as one blank on output. On input, unlimited blanks are allowed.

DATE Format Number Pictures

The following table lists the date format number pictures. A section of notes that follows the table provides additional information about the number pictures.

Table C-3. DATE Format Number Pictures

Number Format Picture	Definition
CC	Unsigned century value (i.e., 1984)
SCC	Signed century value (i.e., -1984)
SYYYY	Signed year
YYYY	Unsigned year
SY,YYY	Signed year with comma (i.e., -1,984)
Y,YYY	Unsigned year with comma (i.e., 1,984)
YYY	Last 3 digits of year (0-999)
YY	Last 2 digits of year (0-99)
Y	Last digit of year (0-9)
Q	Quarter of year (1-4)
MM	Month (1-12)
WW	Week of year (1-53)
W	Week of month (1-5)
DDD	Day of year (1-366)
DD	Day of month (1-31)
D	Day of week (1-7)
HH or HH12	Hour of day produces a two-digit number based on a 12-hour clock (1-12)
HH24	Hour of day produces a two-digit number based on a 24-hour clock (0-23)
MI	Minute (0-59)
SS	Seconds (0-59)
SSSSS	Seconds past midnight (0-86399)
J	Julian day (since Jan 1, 4712 BC) produces width of 7 (0-3547272)

Notes on DATE Format Number Picture Table

Signed symbols begin with either a leading space or a minus sign, unless the fill mode (FM) operator has been turned on.

In a number with up to three digits, leading zeros are automatically inserted. For example, the four-digit symbol, YYYY, would represent the year 783 AD as 0783, and the symbol Y,YYY would output it as 0,783.

Number picture values are always right justified in a field, with leading zeros (a leading space or minus sign is also possible).

Comma and leading "S" pictures produce the same number of digits or characters (places) of output as in the date picture (e.g., Y,YYY = five places).

In the calculation of Julian dates, ALLY includes a year "zero". To compensate for the year "zero", ALLY starts its Julian date system on Jan 1, 4713 instead of 4712. Therefore, an ALLY BC date should have a -1 added to it to account for this difference from the classical Julian convention. Although actual Julian dates are incremented at noon, ALLY increments these dates (as any other dates) at midnight.

DATE Format Character Pictures

The date format character pictures are listed in the following table. A section of notes follows the table. These notes provide additional information about the character pictures.

Table C-4. DATE Format Character Pictures

Character Format Picture	Definition
SYEAR or YEAR	Year in English (i.e., NINETEEN-EIGHTY-FOUR)
MONTH	Name of month
MON	Abbreviation of month name
DAY	Name of day
DY	Abbreviation of day name
AM or PM	Meridian indicator (AM or PM)
A.M. or P.M.	Meridian indicator (A.M. or P.M.)
BC or AD	BC or AD indicator of year
B.C. or A.D.	B.C. or A.D. indicator of year

Notes on DATE Format Character Picture Table

YEAR and SYEAR are not left justified and do not preserve column alignment.

MONTH and DAY are always nine columns wide to account for the maximum length character string that could be output in those containers. MON and DY are always three columns wide, the maximum allowable length of month and day abbreviations. Blanks are added to the right of the string, if needed, to pad out the container.

The symbols AM, PM, BC, and AD always take two characters while A.M., P.M., B.C., and A.D. take four character places.

The length of a character symbol depends on the language in which the application is written. The lengths given above are for the English language. The Dialog allows the developer to specify other lengths to accommodate other languages.

DATE Format Suffix Pictures

The date format suffix pictures are listed in the following table. These suffixes are specified after the number pictures (e.g., YYYYTH or YYYYSP or YYYYSPTH). The section of notes following the table provides additional information about the suffix pictures.

Table C-5. DATE Format Suffix Pictures

Suffix Format Pictures	Description
TH	Puts ST,ND,RD,TH after the number (ex. 8th)
SP	Spells the number (e.g., thirty-five or twenty)
SPTH or THSP	Puts suffix on spelled number (ex. thirty-fifth, twentieth)

Notes on DATE Format Suffix Picture Table

TH preserves column alignment, even in languages other than English.

SP eliminates any column arrangement that had been specified. The longest character string that can be produced with spelling is eighty-two characters. With the "SP" symbols, the first letter of each word in a spelled-out date can be specified as being capitalized. And hyphens (-) can be included in the string. The hyphens are placed between the words to indicate the tens and unit values of each date word group.

The spelled-out version of the year (YYYYSP) differs from the character string YEAR. The spelled-out version for 1985 is ONE THOUSAND NINE HUNDRED EIGHTY-FIVE. The character string version for 1985 is NINETEEN-EIGHTY-FIVE.

DATE Picture Precisions for Rounding Dates

The following table lists the date pictures you can specify as the precision when you round a date.

Table C-6. DATE Picture Precisions for Rounding Dates

Date Picture	Level of Precision
CC, SCC	To the next century if the year is the 50th through the 99th.
SYEAR, YEAR SYYYY, YYYY YYY, YY, Y	To the next year if the month is the 7th through the 12th.
Q	To the next quarter, if the current quarter is more than 1 month and 15 days old. The year can change.
MONTH, MON, MM	To the next month, if the day of the month is 16 through 31. The year can change.
WW, W	To next week of year or month, respectively. If day of week is Wednesday (noon or later), it rounds to the first day of the next week of the year or month, respectively. Otherwise, it rounds to the first day of the current week (for weeks 1 through 53). This picture does not change the year.
DAY, DY, D	To the previous Sunday unless the day is Wednesday (noon or later). The year can change.

Date Picture	Level of Precision
DDD, DD, D, J	To the next day (date and time) of the year, month, and week, respectively, if the time of the current day is noon or later.
HH, HH24, HH12	To the next hour (date and time) If the minute is 30 through 59, there is a ripple carry of the hour to the day, the day to the month, and the month to the year.
MI	To next minute (date and time) If the second is 30 through 59, there is a ripple carry of the minute to the hour, the hour to the day, etc.

DATE Picture Precisions for Truncating Dates

The following table lists the DATE picture precisions you can specify for truncating a date.

Table C-7. DATE Picture Precisions for Truncating Dates

Date Picture	Level of Precision
CC, SCC	At the beginning of the century day = 1, month =1, year = 01; (time = midnight) (e.g., 1984 becomes 1900, -4712 becomes -4700)
SYEAR, YEAR SYYYY, YYYY YYY, YY, Y	At the beginning of the year day = 1, month = 1; (time = midnight)
Q	At the beginning of the quarter day = 1; month = 1 or 4 or 7 or 10; (time = midnight)
MONTH, MON, MM	At the beginning of the month day = 1; (time = midnight)
WW, W	At the beginning of the week of the year or month, respectively subtracts 0 to 6 days (time = midnight) This picture does not change the year number
DAY, DY	At previous Sunday If date is Sunday, date does not change; (time = midnight)
DDD, DD, D, J	At the beginning of the day (date and time) of the year, month or week, respectively (time = midnight)
HH, HH12, HH24	At the beginning of the hour (date and time) Minutes and seconds are removed or set to zero
MI	At the minute (date and time) Seconds are removed or set to zero

End of Appendix C

Index

End of Index