DBPL Memo 121-91

VAX Modula-2 User's Guide
VAX DBPL User's Guide

F. Matthes
J. W. Schmidt
G. Schröder

This document describes how to compile, link and execute Modula-2 programs on a VAX under the VMS operating system. The VAX Modula-2 Compiler is based on an implementation of Modula-2 for the PDP-11 developed by a group under the direction of Prof. N. Wirth at the ETH Zürich.

This document also describes how to compile, link and execute DBPL programs on a VAX under the VMS operating system. DBPL is a relational database programming language that has Modula-2 as its algorithmic kernel. The DBPL Compiler is a fully upward compatible extension of the VAX Modula-2 Compiler.

Software Version 2.0, December 1991

Arbeitsbereich DBIS
Fachbereich Informatik
Universität Hamburg
Schlüterstraße 70
D-2000 Hamburg 13
Federal Republic of Germany

Contents

PREFACE

Manual Objectives

The VAX Modula-2 User's Guide has been designed to support the development and execution of Modula-2 programs in the VAX/VMS environment. The Modula-2 language constructs are described in the Modula-2 Report, 3rd Edition, by N. Wirth [?]. The compiler also incorporates some extensions proposed in the emerging ISO Modula-2 standard [?]. $\top$

Note for DBPL users: The VAX DBPL User's Guide describes the extensions to the VAX Modula-2 compiler that support the development and execution of DBPL database applications in the VAX/VMS environment. The DBPL language constructs are described in the "Report on the Database Programming Language DBPL" [?]. Further documentation and publications on DBPL are available through Hamburg University (see also references in the bibliography). $\perp$

Intended Audience

This manual is intended for programmers who have a general knowledge of the VAX/VMS Command Language and the DBPL programming language. In some sections, a more detailed understanding of the operating system is required. The reader should consult the corresponding manuals for the information needed.

Related Documents

VAX-11 Symbolic Debugger Reference Manual.

VAX-11 Linker Reference Manual.

VAX-11 Architecture Handbook.

VAX-11 PASCAL User's Guide.

VAX-11 Run-Time Library Reference Manual.

VAX/VMS System Services Reference Manual.

Notational Conventions

Within command language descriptions

$\Big[\ \Big]$ large square brackets indicate that the enclosed sequence of symbols is optional,

{ } braces indicate that the enclosed sequence of symbols may be repeated zero or more times.

Parts of the text which apply to extensions of the compiler that are needed to use it as a DBPL rather than a Modula-2 compiler are marked by $\top$ and $\perp$.

DBPL and Modula-2

DBPL is an extended version (true superset) of Modula-2. The VAX DBPL compiler accepts both DBPL and Modula-2 modules. So you can replace the term "DBPL" by "Modula-2" in the following text. Parts that apply only to DBPL modules are marked by $\top$ and $\perp$.

Chapter 1

Preliminaries

1.1 Files

In general, a application is composed of a number of separate *modules*.

The first step in creating an application program is to edit the different source files containing the compilation units (main modules, definition modules and implementation modules).

NOTE

The DBPL compiler distinguishes upper and lower case characters. All reserved words and standard names must be capitalized.

Source files are compiled separately. The order in which this has to be done depends on the import/export relationships between modules. In particular, definition modules must be compiled prior to the corresponding implementation modules.

Six kinds of output files may be produced:

- Compilation of a definition module yields in a **symbol file**, containing a module description readable by the compiler. This file is needed when the implementation module or a compilation unit importing the module is compiled.
- The result of processing an implementation or main module is an **object file** readable by the linker.
- Compilation of a database definition, implementation or main module additionally generates a **database dictionary file** and (several) **data files** and **index files** for persistent database variables.
- Optionally, a **listing file** can be produced. It contains a listing of the module with error messages. A machine code listing and a cross reference table can be included.
- Finally, the separate object files have to be linked together to form an executable **image file**, which can be executed by the `RUN` command.

1.2 File Specifications and Defaults

A full VAX/VMS file specification has the following form:

```
node::device:[directory]filename.filetype;version
```

Node, device, directory, and version are optional items; their default values are:

`node`:	applicable only to systems supporting DECnet-VAX,
`device`:	user's current device,
`directory`:	user's current default directory,
`version`:	highest version on input, highest version+1 on output.

The filetype specification may be omitted in certain cases. The default type depends on usage:

`DEF,MOD,DBP`	source files, input to DBPL compiler
`SYM`	symbol file, input to and output from DBPL compiler
`DIC,DAT,IOO`	database files, output from DBPL compiler (when creating a database)
`OBJ`	object code file, output from DBPL compiler
`LIS`	source file listing, output from DBPL compiler
`OBJ`	object code file, input to linker
`EXE`	executable file, output from linker

Chapter 2

Compiling

Each DBPL compilation unit has to be compiled separately. In the case of a definition module, a symbol file is created, an object file otherwise. Optionally, a source file listing may be produced. The compiler also searches for and reads the symbol files belonging to imported modules. If not specified otherwise, a standard search strategy is applied (see ??). The DBPL compiler is invoked ⊤
by the `DBPLC` or `MODULA` command which has the following form:

```
$ DBPLC {/qualifier} file-spec {/qualifier}
```

respectively

```
$ MODULA {/qualifier} file-spec {/qualifier}
```

file-spec
determines the source file containing the compilation unit. Only one file is accepted. The default file type is `MOD` for Modula-2 files (command `MODULA` or qualifier `/NODBPL`) and `DBP` for DBPL files (command `DBPLC` or qualifier `/DBPL`). Type `DEF` is recommended for definition ⊥
modules.

In interactive mode, you can also enter the file specification on a separate line by typing a carriage return after you type `DBPLC` or `MODULA`. The system responds with a prompt for the file specification:

```
$ DBPLC RETURN
$_File:
```

On termination the DBPL compiler returns a status code in the reserved global symbols $STATUS and $SEVERITY which indicates either a success or a severe error during compilation. This information can be used for controlling the execution of VMS command procedures (see *VAX/VMS Guide to Using Command Procedures* for details).

Processing of a source file comprises several passes. Temporary work files named `INTERFILE.DVC` serve for interpass communication. The normal succession of passes is for definition modules

1. syntax analysis
2. declaration analysis
3. symbol file generation

4. listing generation (optional)
5. database creation (for database definition modules)

and for implementation and main modules

1. syntax analysis
2. declaration analysis
3. body analysis
4. code generation
5. listing generation (optional)
6. database creation (for database modules).

If the compiler option `/LOG` is specified, the succesion of passes mentioned above is displayed. A more detailed information about compiler options is given in section ??.

VAX/VMS Modula-2 symbol files are compatible with DBPL symbol files, hence modules written in Modula-2 may also be imported to DBPL programs.

2.1 Compiler Command Qualifiers

`DBPLC` resp. `MODULA` command qualifiers control optional compiler settings. For every qualifier there is a negative form to suppress the respective effect. The negative form is indicated by a leading "`NO`", e.g., "`NOLIST`" is the negative form of "`LIST`". If you want to change the default values of the qualifiers, edit the file `DBPLC.CLD` resp. `MODULA.CLD`. A default value is marked by the keyword "`DEFAULT`". All other values are set by default to "`NO...`". The meaning of the qualifiers and their default values is listed below.

CHECK
Code is generated to perform run-time checks for out-of-range array bounds and case labels.

By default, `CHECK` is enabled.

CREATE
A new database is created when compiling a database definition module.

By default, `CREATE` is disabled.

CROSS_REFERENCE
A cross reference table is included in the source listing. This qualifier is effective only when used in conjunction with `LIST`.

By default, `CROSS_REFERENCE` is disabled.

DBPL
A DBPL module will be compiled. All DBPL enhancements of Modula-2 are available.

By default, `DBPL` is enabled when using command `DBPLC` and disabled when using command `MODULA`.

DEBUG
The `DEBUG` qualifier produces information to be used by the VAX Symbolic Debugger and the run-time error reporting mechanism.

By default, `DEBUG` is disabled and only traceback information is generated.

LIST

A listing file is written. The qualifier has the general form:

```
/LIST[=file-spec-list]
```

You may include a file specification for the listing file. If no specification is given a file of type `LIS` with the name of the source file is created. The file contains at least the source text and the error messages, but further listings are added by applying `CROSS_REFERENCE` and/or `MACHINE_CODE`.

By default, `LIST` is disabled for interactive mode and enabled for batch mode (but note that even in batch mode the listing file is not automatically printed!).

LOG

If the `LOG` option is enabled, the following information is displayed:

- compiler version,
- qualifiers applied,
- succession of the compiler passes, and
- the names of the symbol files read.

By default, `LOG` is disabled for interactive mode and enabled for batch mode.

MACHINE_CODE

A readable representation of the generated code is included in the source listing. This qualifier is effective only in conjunction with the `LIST` qualifier.

By default, `MACHINE_CODE` is disabled.

OBJECT

An object file for the linker is produced. The qualifier has the form:

```
/OBJECT[=file-spec]
```

You may specify a file for the object module. The default is a file of type `OBJ` which takes the name of the source file. If the source file contains a definition module this qualifier has no effect.

By default, `OBJECT` is enabled for main and implementation modules, disabled otherwise.

QUERY

The `QUERY` qualifier forces the compiler to ask explicitly for the symbol files to be read.

By default, `QUERY` is disabled and the compiler uses a standard strategy to search for the symbol files (see section ??).

The standard search strategy is also effective when `QUERY` is enabled but no file specification is given.

SYMFILE

A symbol file is created if the source contains a definition module. It has the form:

```
/SYMFILE[=file-spec]
```

You may supply an optional file specification for the symbol file. The default is a file of type `SYM` with the name of the source file.

By default, `SYMFILE` is enabled for definition modules, disabled otherwise.

Summary:
DBPL Command Qualifiers

Qualifier	*Negative Form*	*Default*
`/CHECK`	`/NOCHECK`	`/CHECK`
`/CREATE`	`/NOCREATE`	`/NOCREATE`
`/CROSS_REFERENCE`	`/NOCROSS_REFERENCE`	`/NOCROSS_REFERENCE`
`/DBPL`	`/NODBPL`	`/DBPL`
`/DEBUG`	`/NODEBUG`	`/NODEBUG`
`/LIST[=file-spec]`	`/NOLIST`	`/NOLIST (interact.)` `/LIST (batch)`
`/LOG`	`/NOLOG`	`/NOLOG (interactive)` `/LOG (batch)`
`/MACHINE_CODE`	`/NOMACHINE_CODE`	`/NOMACHINE_CODE`
`/OBJECT[=file-spec]`	`/NOOBJECT`	`/OBJECT`
`/QUERY`	`/NOQUERY`	`/NOQUERY`
`/SYMFILE[=file-spec]`	`/NOSYMFILE`	`/SYMFILE`

2.2 Local Control of Run-Time Checks

The insertion of run-time checks into the generated code can additionally be controlled during compilation by using the S- and T-options. An option must be entered into the source text at the beginning of a comment clause; it has the general form:

```
Option = "$" Name Switch.
Name = "S" | "T".
Switch = "+" | "-" | "=".
```

where the associated switch

+	enables run-time checks,
-	disables run-time checks,
=	resets the option to the previous value.

The S-option controls subrange checks, e.g. when assigning a value to a variable of a subrange type. The T-options controls index checks when accessing arrays.

Examples:

```
(* $T+ to generate test code when accessing arrays *)

(* $T- to turn off test code generation *)

(*$S-*) i:=j; (*$S=*) (* no subrange checks *)
```

2.3 Search Strategies

In the default case (i.e., when no file name is specified in query mode) the compiler applies a standard strategy to search for the symbol files needed: The default file name is constructed out of the module's name and the file extension `SYM`.

The file is first searched in the user's default directory. If the search fails, directories with the logical names `MOD$LIBRARY`, `MOD$LIBRARY_1` up to `MOD$LIBRARY_9` are searched (in this order). The entire process is performed first in the process logical name table, second in the job logical name table, then in the group logical name table, and finally in the system logical name table. Each search terminates when the first undefined logical name is encountered. If the file has not yet been found, the directory with the logical name `MOD$SYSTEM` is searched.

The result of the searching process is displayed to the user when the `LOG` option is enabled.

2.4 Compiler Listing

The contents of the listing is controlled by the command qualifiers `LIST`, `CROSS_REFERENCE`, and `MACHINE_CODE`. The application of `LIST` is necessary to obtain a listing at all.

Each listing page has a two line heading. The first line contains (from left to right):

- the name of the compilation unit,
- the date of compilation,
- the compiler version, and
- a running page number.

In the second line you will find

- the listing subject,
- the creation date, and
- the name of the source file.

The listing terminates with a compilation summary, informing about the active compiler options and giving some performance data.

2.4.1 Source Listing

Initiated by `LIST`.

The source list contains a copy of the source file preceded by the compiler-generated line numbers. Error messages are inserted after the erroneous lines. The position where an error has been detected is marked by an arrow.

2.4.2 Cross Reference Table

Initiated by `CROSS_REFERENCE`, presupposes `LIST`.

The cross reference table presents all identifiers defined or accessed in the compilation unit in alphabetical order. Standard names are not tabulated.

Each table entry consists of at least two items: the identifier and its declaration class. The latter distinguishes constants, types, variables, parameters (value and variable), fields, procedures, functions, and modules.

Further specifications which may follow are:

- A short type characterization enclosed in angle brackets. For subranges the base type is denoted; if the type has been denoted by a synonym, the original type name is given. This item is omitted for hidden types, predefined types, procedures, and modules.
- The name of the scope in which the identifier is defined, preceded by "in" or "from". This item is omitted when the name is defined at global level. The prefix "from" indicates that the defining scope is external. Alternatively, the word "imported" may appear if the name designates an external module; in this case the full symbol file name is also given.
- The word "exported", which means that the identifier is exported from the enclosing module.
- A list of numbers of the lines where the name is referenced. A definition line is marked by an asterisk.

Included in the table are also identifiers imported indirectly (as part of another imported identifier), although they are not referenced in the source text. There is, of course, no list of line numbers in this case.

2.4.3 Machine Code Listing

Initiated by `MACHINE_CODE`, presupposes `LIST`.

The generated code is represented as a VAX MACRO program source listing. Correspondence with the source text is established via the line numbers inserted as comments on the right hand side.

Chapter 3

Linking

After compilation, the separate object modules of the application program are linked together to produce an executable image file.

3.1 The Link Command

The modules of an application program are linked with the normal `LINK` command:

```
$ LINK/command-qualifiers file-spec-list
```

command-qualifiers
Specify output file options.

file-spec-list
Specifies the input files to be linked. File specifications can be separated by commas or plus signs. To create executable code you must specify all your object files to be linked, except the ones stored in the default link libraries. The first file named should contain the main program (module). By default, object files are assumed to be of type `OBJ`.

If some or all of the object files reside in a library, you can specify this library with the qualifier `/LIB`, e.g. `MYLIB/LIB`. The linker will automatically search these libraries for the needed files. The default file type for object libraries is `OLB`.

If no other specification is given, the executable image file takes the name of the first object file and is of type `EXE`.

In interactive mode you may enter the file specifications on a separate line by typing a carriage return after the command name. The system responds with a prompt for the file specification.

3.2 Linker Command Qualifiers

Below, a few qualifiers are listed which may be of interest to the DBPL programmer. For some qualifiers there are also negative forms. Refer to the *VAX-11 Linker Reference Manual* for detailed information.

3.2.1 Map File Qualifiers

The `MAP` qualifier directs the linker to generate a map file containing a summary of the image's characteristics, a list of contributing modules, and a list of global symbols and values. The qualifier has the form:

```
/MAP[=file-spec]
```

where the optional file specification designates the map file. The default is a file of type MAP with the name of the first object file.

By default, `MAP` is disabled for interactive mode and enabled for batch mode.

In conjunction with `/MAP` the modifying qualifiers `BRIEF` or `FULL` and `CROSS_REFERENCE` may be used. `/BRIEF` and `/FULL` define the amount of information included in the map file, as follows:

BRIEF produces a summary of the image's characteristics and a list of contributing modules.

FULL adds a summary of characteristics of image sections in the linked image to the normal map file.

CROSS_REFERENCE can be used with `/MAP` or `/MAP/FULL` to include cross-reference information for global symbols in the map file.

By default, `CROSS_REFERENCE` is disabled

3.2.2 Debugging and Traceback Qualifiers

If the `DEBUG` qualifier is used, the program will always be executed under the control of the debugger. (You may suppress this by issuing `/NODEBUG` with the `RUN` command.)

By default, `DEBUG` is disabled.

The `TRACEBACK` qualifier is used to have error messages accompanied by symbolic traceback information.

By default, `TRACEBACK` is enabled. `/DEBUG` implies `/TRACEBACK`.

Summary:
Link Command Qualifiers

Qualifier	*Negative Form*	*Default*
`/BRIEF`	none	not applicable
`/CROSS_REFERENCE`	`/NOCROSS_REFERENCE`	`/NOCROSS_REFERENCE`
`/DEBUG`	`/NODEBUG`	`/NODEBUG`
`/FULL`	none	not applicable
`/MAP[=file-spec]`	`/NOMAP`	`/NOMAP (interactive)` `/MAP (batch)`
`/TRACEBACK`	`/NOTRACEBACK`	`/TRACEBACK`

3.3 Identification Checking

The VAX DBPL compiler provides identification information for describing the relationship between symbol files and object files belonging to the same module. This identification is in fact the compilation date and time of the corresponding definition module and it is passed to the linker in the form of an "Entity Ident Consistency Check" subrecord.

The linker checks the "Entity Ident Consistency Check" subrecord of each object module before it links them together. If object files of two modules importing different versions of the same definition module are encountered the linker issues a warning message. In such a case, the object files giving rise to the message should be replaced by newly compiled ones.

For more information about the "Entity Ident Consistency Check" subrecords, see the *VAX-11 Linker Reference Manual.*

Chapter 4

Executing

The application program can be started by the `RUN` command:

```
$ RUN[/[NO]DEBUG] file-spec
```

The `DEBUG` qualifier allows to run the program under control of the debugger, even if you have issued the `DBPLC` and `LINK` commands without this qualifier. `/NODEBUG` will override a `/DEBUG` given at link time.

In interactive mode, the file specification may be entered on a separate line by typing a carriage return after the command name. The system responds with a prompt for the file specification.

4.1 Debugging the Program

Currently, the VAX Symbolic Debugger runs DBPL programs with the language mode set to PASCAL. See the corresponding chapter of the *PASCAL User's Guide* (or the *VAX-11 Symbolic Debugger Reference Manual*) for a description. Because of the differences between the two languages, some features work differently, as there are:

- No distinction is made between lower and upper case letters in identifiers.
- Nested modules are not recognized; all variables and procedures declared inside a nested module are made global to the compilation unit.
- Module bodies are treated as parameterless procedures with the names of the modules.

NOTE

The differences mentioned above may lead to name conflicts inside the debugger, which are not always handled properly.

- Type `CARDINAL` is treated as `INTEGER`.
- All variables and procedures can only be accessed in the context of the module they are defined in (see the debugger commands `SET MODULE, SET SCOPE`).
- Sets are displayed and must be entered in PASCAL notation, using square brackets ("[", "]").
- Open array parameters are known to the debugger as fixed size arrays with a lower bound of 0 and an upper bound of $2**16-1$.

- Constant and type identifiers are not known to the debugger.
- All tagfields and fields in record variants are known as ordinary record fields to the debugger, i.e., the tagfield and variant information is not present.

DBPL transactions are provided with a special condition handler which enforces an abort of the current transaction if a severe runtime condition is signalled. As a result, the debugger usually does not catch transaction failures. You may direct the debugger to treat transaction failures also by issuing the `SET BREAK/EXCEPTION` command.

NOTE

Debugging a transaction may cause the **delay of concurrent transactions** started by other users.

Chapter 5

Procedure Calling Conventions

In the context of the VAX/VMS operating system, a procedure is a routine entered by a CALL instruction. In a DBPL program, such a routine can be a function or a procedure written in DBPL, a VAX/VMS system service, or a VAX Run-Time Library procedure.

This chapter provides information on the calling conventions used by the VAX DBPL compiler and on calling VAX/VMS system services and VAX Run-Time Library procedures. A basic knowledge of the VAX procedure calling and argument passing mechanisms is assumed. For more information see the *VAX-11 Run-Time Library Reference Manual* and the *VAX-11 Architecture Handbook*.

VAX DBPL uses the VAX CALLS instruction to call procedures. Each time a procedure is called, the DBPL compiler constructs an argument list on the stack. The arguments in the list are based on the parameter type and the parameter kind (value or reference) specified in the formal parameter list and the values in the actual parameter list.

5.1 Parameter Passing Mechanisms

The VAX procedure calling standard defines three mechanisms by which arguments are passed to procedures:

1. by-reference
2. by-immediate-value
3. by-descriptor

Table ?? describes the mechanisms that are used by the VAX DBPL compiler, depending on the type and the kind of the formal parameter:

An open array parameter is passed by two arguments. The parameter list created by the compiler for an open array parameter definition

```
(...; a: ARRAY OF type; ...)
```

is equivalent to the following foreign parameter definition (see below)

```
(...; %IMMED higha: CARDINAL; %IMMED adra : ADDRESS; ...)
```

with `adra = ADR(a)` and `higha = HIGH(a)`.

type	value parameter	reference parameter
BOOLEAN		
CHAR		
INTEGER		
CARDINAL		
POINTER		
SET		
Enumeration		
Subrange		
BYTE	by-reference	by-reference
SHORTWORD		
WORD		
ADDRESS		
REAL		
LONGREAL		
F_FLOATING		
D_FLOATING		
G_FLOATING		
H_FLOATING		
QUADWORD		
OCTAWORD		
RECORD		
ARRAY		
RELATION	by-reference	by-reference
SELECTOR		
CONSTRUCTOR		
PROCEDURE	by-reference	
TRANSACTION		

Table 5.1: Parameter passing mechanisms

5.2 Function Return Values

A function returns a value to the calling program. The method by which a value is returned depends on its type as listed in table ??.

5.3 Foreign Procedures

Access to procedures and variables written in other VAX programming languages, especially to the VAX/VMS system services and Run-Time Library procedures, is provided through special definition modules, so-called foreign definition modules. They serve only to export the descriptions of the respective procedures and have no corresponding DBPL implementation modules.

A foreign definition module is syntactically indicated by the prefix "`%FOREIGN`":

```
$ ForeignDefinitionModule =
$     "%FOREIGN" DEFINITION MODULE ident ";"
$     { import } { foreigndefinition } END ident ".".
```

This section contains the formal definitions of the procedures and variables to be linked. Pro-

type	return method
BOOLEAN CHAR INTEGER CARDINAL POINTER Enumeration Subrange BYTE SHORTWORD WORD ADDRESS REAL F_FLOATING	Register R0
LONGREAL D_FLOATING G_FLOATING QUADWORD	Registers R0,R1
H_FLOATING OCTAWORD	by reference as the first parameter in the function's parameter list

Table 5.2: Function value return methods

cedure and variable names must be known to the linker and the calling convention must match at the linker level.

To allow full utilization of the VAX/VMS procedure calling standard, additional parameter qualifiers have been introduced, applicable only within foreign definition modules:

```
$ ForeignFormalParameters =
$     "(" [ ForeignFPSection {";" ForeignFPSection }] ")"
$     [":" qualident ].
$ ForeignFPSection =
$     [ VAR ][ "%REF" | "%IMMED" | "%STDESCR"]
$     IdentList ":" FormalType.
```

where

`%IMMED`	denotes passing by immediate value,
`%REF`	denotes passing by reference, and
`%STDESCR`	denotes passing by string descriptor.

As an example, we give a definition module which allows access to some screen handling procedures:

```
%FOREIGN DEFINITION MODULE CommandLanguageInterface;
  ...
  PROCEDURE CLI$PRESENT
      (    %STDESCR entityDesc:ARRAY OF CHAR):CARDINAL;

  PROCEDURE CLI$GET_VALUE
```

```
      (     %STDESCR entityDesc: ARRAY OF CHAR;
       VAR  %STDESCR retdesc:    ARRAY OF CHAR;
       VAR           retlength: CARDINAL):CARDINAL;
    ...

  END CommandLanguageInterface.
```

Foreign definition modules should not be written without knowledge about the parameter passing mechanisms involved.

Chapter 6

System Dependent Facilities

6.1 Low Level Facilities

There are applications (e.g., system programming) for which the language rules may turn out to be too restrictive. For this reason, DBPL provides some means for "low level programming"; most of them are implementation dependent.

The programmer is urged to use these low level facilities very carefully and only if it seems unavoidable. He will be much less protected against errors because low level facilities are *not* checked for consistency with language rules.

6.1.1 Type Transfer Function

The type transfer function `SYSTEM.CAST` serves to breach DBPL's type system. It does not involve any actual computation but modifies the compiler's type checking. For instance, if `c` is an expression of type `CARDINAL` then

```
SYSTEM.CAST(BITSET,c)
```

is interpreted as the corresponding value of type `BITSET`. This is a correspondence determined by the underlying system and not by the programming language itself. Therefore, using the type transfer function requires familiarity with the internal representation and storage allocation scheme of the corresponding data types.

Note: The original type transfer of Modula-2 is provided by the type name as function, e.g. `BITSET(c)`. This will be changed by the Modula-2 Standard. The DBPL compiler supports both forms, but it is recommended to use `SYSTEM.CAST`.

6.1.2 Absolute Addresses

The assignment of absolute addresses is still allowed in DBPL but this feature should be used with utmost care because all addresses are usually relocated by the linker! The Modula-2 Standard will change this feature.

6.1.3 The Module SYSTEM

The module `SYSTEM` offers some further facilities of the DBPL language. Most of them are implementation dependent and/or refer to the given processor. Facilities of that kind are sometimes necessary for the so called "low level programming". `SYSTEM` contains also types and procedures which allow a very basic coroutine handling.

The module `SYSTEM` is known to the compiler because its exported objects obey special rules, that must be checked by the compiler. If a compilation unit imports objects from module `SYSTEM`, no symbol file has to be supplied for this module.

Objects exported from module `SYSTEM`:

Types

`BYTE, SHORTWORD, WORD, QUADWORD, OCTAWORD`

Objects of these types represent individually accessible storage units (`BYTE` = one byte, `SHORTWORD` = two bytes, `WORD` = four bytes, `QUADWORD` = eight bytes, `OCTAWORD` = sixteen bytes). Only assignment is allowed for variables of these types. A parameter of type `BYTE, SHORTWORD, WORD, QUADWORD` or `OCTAWORD` may be substituted by an actual parameter of any type that uses the same number of bytes in storage. If the parameter is a value parameter of type `BYTE, SHORTWORD` or `WORD`, then values of types using up to four bytes are also allowed for substitution. The same holds for the substitution of an open array parameter with elements of type `SHORTWORD, WORD, QUADWORD` or `OCTAWORD` by a value of a type which size is not an integral multiple of the open array element size.

For compatibility with the VAX/VMS data type names the identifier `LONGWORD` is also exported from the module `SYSTEM` as a synonym for the identifier `WORD`.

The Modula-2 Standard will support only `BYTE` and `WORD`.

`ADDRESS`

Objects of type `ADDRESS` represent the byte address of a storage location. The type `ADDRESS` is compatible with all pointer types and is itself defined as `POINTER TO WORD`.

All integer arithmetic operators apply to this type. This will be changed by the Modula-2 Standard. It is recommended to use no arithmetic with pointers.

`PROCESS`

Objects of type `PROCESS` are used for process handling. Upon termination of a process other than the main process an exception condition is signaled.

This type and the corresponding procedures will be separated in another system module called `COROUTINES` by the Modula-2 Standard.

`F_FLOATING, D_FLOATING, G_FLOATING, H_FLOATING`

The DBPL standard type `REAL` is implemented using the VAX `F_FLOATING` data type; the type `LONGREAL` is implemented by `D_FLOATING`. The other VAX floating point data types are made available in the VAX DBPL compiler through export from module `SYSTEM`. All operators defined in DBPL for the types `REAL` and `LONGREAL` are also defined for the types `F_FLOATING, D_FLOATING, G_FLOATING` and `H_FLOATING`. Different floating point types cannot be mixed in expressions (i.e., there is no implicit conversion). However the module `Conversions` exports procedures for the conversion of the different floating point data types into each other.

The standard functions `TRUNC` and `INT` are applicable to all floating point data types.

NOTE

If your VAX processor is not equipped with the extended instruction set option and you are using variables of type `G_FLOATING, H_FLOATING` or `OCTAWORD`, the

VAX extended instruction set emulator, LIB$EMULATE, should be established as a condition handler. See the *VAX-11 Run-Time Library Reference Manual* for more information.

Procedures

`NEWPROCESS(p:PROC; a: ADDRESS; n: CARDINAL; VAR p1: PROCESS)`
Instantiates a new process. At least 160 bytes are needed for the workspace of a process. This procedure will be put to a new module `COROUTINES` when the Modula-2 Standard is published.

`TRANSFER(VAR p1, p2: PROCESS)`
Transfers control between two processes. This procedure will be put to a new module `COROUTINES` when the Modula-2 Standard is published.

Functions

`CAST(type, expression): type`
The bit representation of `expression` is interpreted as a value of `typename`.

`ADR(variable): ADDRESS`
Returns the storage address of the variable.

`TSIZE(type): CARDINAL`
See below.

`TSIZE(type, tag1const, tag2const, ... ): CARDINAL`
Returns the number of bytes allocated for a variable of the type. If the type is a record with variants, then tag constants of the last FieldList (see DBPL syntax) may be substituted in their nesting order. If no or not all tag constants are specified, then the remaining variant with maximal size is assumed. Be careful when using this option, because the compiler is not checking if you are using the correct variants!

`REGISTER(num: CARDINAL): CARDINAL`
Returns the content of the specified register. This function applies only to the VAX hardware.

6.2 Storage Management

6.2.1 The Module Storage

The storage management procedures `ALLOCATE` and `DEALLOCATE` which serve to dynamically obtain and return storage space can be imported from the standard module `Storage`:

```
DEFINITION MODULE Storage;

  FROM SYSTEM IMPORT ADDRESS;

  PROCEDURE ALLOCATE (VAR p: ADDRESS; size: CARDINAL);
  PROCEDURE DEALLOCATE (VAR p: ADDRESS; size: CARDINAL);

END Storage.
```

`ALLOCATE` signals an exception whenever there is not enough free heap space. This will be changed by the Modula-2 Standard: `ALLOCATE` will return `NIL` if there is not enough heap space.

When writing your own storage management procedures you should care for an appropriate declaration. `DEALLOCATE` should set `p` to `NIL`. Since `NEW` and `DISPOSE` are translated into calls to `ALLOCATE` and `DEALLOCATE`, these must be compatible with the type

```
PROCEDURE (VAR ADDRESS, CARDINAL)
```

6.2.2 Storage Allocation

Table ?? summarizes the storage allocation and alignment for variables of unstructured and set type (note that "word", "longword", "quadword", and "octaword" are DEC-terms!):

type	storage allocation	alignment boundary

`BOOLEAN`
`CHAR`
`BYTE`
	8 bits (1 byte)	byte

`SHORTWORD`	16 bits (1 word)	word

`BITSET`
`INTEGER`
`CARDINAL`
`F_FLOATING`
`REAL`
`POINTER`
`WORD`
`ADDRESS`
`RELATION`
`SELECTOR`
`CONSTRUCTOR`
	32 bits (1 longword)	longword

`LONGREAL`
`D_FLOATING`
`G_FLOATING`
`QUADWORD`
	64 bits (1 quadword)	longword

`H_FLOATING` `OCTAWORD`	128 bits (1 octaword)	longword
Enumeration	8 bits (1 byte) if type contains 256 or less elements; 16 bits (1 word) if type contains 256 or more elements	byte word
`SET`	8 bits (1 byte), 16 bits (1 word), or 32 bits (1 longword), according to cardinality of base type	byte word longword

Table 6.1: Storage allocation and alignment

Variables of subrange type are allocated and aligned in the same way as variables of their base type.

An array is stored and aligned according to the type of its elements. Every element is also aligned according to its type.

Records are stored field by field. Each field is allocated according to its type and aligned on a byte boundary. ⊤

Variables of Type **`RELATION`**, **`SELECTOR`** and **`CONSTRUCTOR`** are allocated to one longword and aligned on a longword boundary. ⊥

Appendix A

Restrictions and Extensions

The implementation of DBPL on the VAX differs in some respect from the language definition [?]; some restrictions have to be considered. Most parts of the language follow Wirth's 3rd edition of the language report on Modula-2 [?]. Some are taken from the 4th draft proposal of the Modula-2 Standard [?].

Identifiers

Identifiers may contain the additional characters "$" and "_":

$ ident = letter { letter | digit | "$" | "_" }.

Note: The Modula-2 Standard will also allow the underline character "_" in identifiers. Due to the naming conventions of the linker some additional restrictions must be obeyed for module names and exported procedure identifiers. An external name is — at the linker level — composed of the module name and the exported name, both converted to upper case and separated by a dot. If the composed name is longer than 31 characters (including the dot) the individual names are shortened to 15 characters. A conflict will occur if the resulting name is not unique anymore.

The module name of a compilation unit should not contain the additional characters "$" and "_".

Qualified export

Is is possible to restrict exports of a definition module with the statement `EXPORT QUALIFIED`. This language element is obsolete and may be missing in future versions of the DBPL compiler.

Opaque types

According to the Modula-2 standard opaque types should be implemented as pointers. The DBPL compiler allows all types of size `WORD` as implementations of opque types. Suspect this feature to be changed.

String constants

The length of string constants is restricted to a maximum of 10000 characters. String constants can be concatenated with the operator +. This will also be in the Modula-2 Standard.

Aggregates

Even in the Modula-2 subset it is possible to use aggregates as value constructors for arrays and records. This has been adopted by the Modula-2 standard.

Example:

```
TYPE Rec = RECORD x,y: CARDINAL; s: ARRAY [0..10] OF CHAR END;
VAR r: Rec;
...
r := Rec{0,100,'Hello'};
```

In the case of nested records or structured array elements only the outmost type has to be mentioned, e.g.

```
TYPE Arr: ARRAY [0..1] OF Rec;
VAR a: Arr;
...
a := Arr{{0,1,'a'},{0,2,'b'}};
```

⊤

Aggregates can also be used to construct relation values, e.g.

```
TYPE Rel = RELATION x,y OF Rec;
VAR r: Rel;
...
r := Rel{{0,1,'a'},{0,2,'b'}}
```

⊥

Alternative character set

The Modula-2 Standard states that an alternative character set has to be supported, because the american character set is not always available. The following table shows the alternative constructs.

ASCII	alternative
[	(!
]	!)
{	(:
}	:)
|	!
^	@

Real number conversion

The real number conversion routines are implemented according to VAX instructions, i.e. they work with type `INTEGER` instead of type `CARDINAL`. The results of `FLOAT(x)` and `LFLOAT(x)` will be wrong if `x` is a cardinal number outside the range of `INTEGER`. The result of `TRUNC(x)` will be wrong if `x` is a negative number or a positive number outside the `INTEGER` range.

Open arrays

The maximal length of open arrays is 65536 bytes. Only one dimension is allowed.

FOR statement

The step must not be greater than 7FFFFFFFH (`MAX(INTEGER)`).

CASE statement

The labels of a case statement must not be greater than 7FFFFFFFH.

Priority specifications

Priority specifications are not implemented.

Standard procedures

In contrast to Wirth's 3rd edition of the language report, but in accordance with the Modula-2 standard this compiler implements the standard procedures `NEW` and `DISPOSE`. They are translated into calls to `ALLOCATE` and `DEALLOCATE`, so these procedures have to defined in the context, e.g. by import from the module `Storage`.

Elementary data types

As numerical types `INTEGER`, `CARDINAL`, `REAL` and `LONGREAL` have been implemented. There are no types `LONGINT` or `LONGCARD`, because `INTEGER` and `CARDINAL` are already 32 bit long. It is recommended to use subrange types of `INTEGER` and `CARDINAL` to support portability.

Standard functions

The following standard functions are provided by the current implementation:

Conversion functions

Conversions are only allowed between elementary data types. All possible conversions can be done with the function `VAL`; see table ??:

```
VAL(typename,expression)
```

Example:

```
intvar := VAL(INTEGER,realvar)
```

	typename (or subrange)						
expr.	CARDINAL	INTEGER	REAL	LONGREAL	CHAR	BOOLEAN	enum.
CARDINAL	√	√	√	√	√	√	√
INTEGER	√	√	√	√	√	√	√
ZZ[a]	√	√	√	√	√	√	√
REAL	√	√	√	√			
LONGREAL	√	√	√	√			
RR[b]	√	√	√	√			
CHAR/SS1[c]	√	√			√		
BOOLEAN	√	√				√	
enum.	√	√					√

[a] ZZ is the type for literal `INTEGER/CARDINAL` constants.
[b] RR is the type for literal `REAL/LONGREAL` constants.
[c] SS is the type for literal `CHAR/ARRAY OF CHAR` constants.

Table A.1: Possible conversions with `VAL(typename,expr)`

Some common conversions are available in a shorter form, as explained below:

`CHR(whole):CHAR`
: whole number → character.

`FLOAT(numeric):REAL`
: whole or floating point number → real number (short floating point)

`LFLOAT(numeric):LONGREAL`
: whole or floating point number → longreal number (long floating point)

`INT(all):INTEGER`
: whole or floating point number, character, enumeration or boolean value → signed number

`ORD(ordinal):CARDINAL`
: whole number, character, enumeration or boolean value → unsigned number

`TRUNC(real):CARDINAL`
: floating point number → unsigned number

Other standard functions

`CAP(CHAR):CHAR`
: letter to capital

`ABS(numeric):numeric`
: absolute value of a whole or floating point number

`ODD(whole):BOOLEAN`
: is the whole number odd?

`SIZE(var/typename):whole`
: Returns the number of bytes allocated for the variable or a variable of the specified type. If the variable is of a record type with variants, then the variant with maximal size is assumed. Neither dynamic variables of type ARRAY OF ... nor dereferenced pointers or elements of records or fields are allowed as arguments. Use **`HIGH`** or **`TSIZE`** instead.

`HIGH(dynvar):whole`
: highest index of a dynamic variable (type ARRAY OF...); only one dimension is possible.

`LENGTH(string):whole`
: length of a string, which is terminated by the character 0C.

`MAX(typename):type`
: maximum value of this type.

`MIN(typename):type`
: minimum value of this type.

Appendix B

Modula-2 / DBPL Compiler Installation

The DBPL distribution tape contains the DBPL system (a compiler and a runtime system) as well as a Modula-2 compiler, both running under VAX/VMS and developed at the University of Hamburg. Both compilers are based on the Modula-2 compiler M2RT11 developed at the "Institut für Informatik, Eidgenössische Technische Hochschule Zürich".

The DBPL compiler is able to utilize the symbolfiles produced by the Modula-2 compiler. Using Foreign-Definition modules (see the Modula-2 User's Guide on tape), it is furthermore possible to develop arbitrary mixed language systems with the DBPL compiler.

Before proceeding with the installation process, make sure that you have enough disk space available. Approximately 9000 disc blocks are needed to install both compilers and their libraries.

1. Create a directory where the compiler should reside:

   ```
   $ create/dir [.modula]
   ```

2. Step into this directory:

   ```
   $ set default [.modula]
   ```

3. Mount the tape:

   ```
   $ mount/foreign mua0:
   ```

4. Copy the files to your directory:

   ```
   $ backup mua0:dbpl.bck [...]/log
   ```

5. Dismount the tape:

   ```
   $ dismount mua0:
   ```

6. Now you find two sub-directories in the directory `[.modula]`:

   [.system] the compiler and all files needed for runtime (including symbol and object files of library modules);

   [.lib] sources of the library definition modules;

[.demo] sources of a tiny DBPL application.

7. Step to the directory [.system]:

   ```
   $ set default [.system]
   ```

8. Installation of path names. You have two possibilites to install the compiler:

   (a) *INSTALL:* The compiler resides in this directory and each user has to set some path names to use it:

   ```
   $ @install
   ```

   A command procedure `STARTUP.COM` is created which every user has to execute before using the compiler.

   (b) *SYSTEM_INSTALL:* The compiler and all stuff from `[.system]` is copied to a system directory and some system path variables are set so that all users can use the compiler without setting own path variables:

   ```
   $ @system_install
   ```

   All files are copied so that you can then delete all files from `[.system]`.

Appendix C

A Sample DBPL Program

The distribution kit of the Modula-2 compiler includes a small DBPL demonstration program. This chapter describes how to compile and execute this program. Further information on the language DBPL and its use is available on request from Hamburg University.

These are the general steps required for the development of database applications:

1. Create the DBPL source files.
2. Compile them with `dbplc/`*command_qualifier file_name*. Look at the file `MOD$SYSTEM:dbpl.cld` for allowed command qualifiers. *DATABASE MODULE' s* have to be compiled with the qualifier `/create` to create an (uninitialized) database. Details of the command qualifiers can be found in the VAX DBPL User's Guide. The first application program accessing that database will execute the user-definable database initialization code for that database.
3. Link your objectfiles with `link` *file*$_1$*.obj, ..., file*$_n$*.obj*. File$_1$ should be the main program module. It is not necessary to name the object libraries of the Modula-2 and DBPL system explicitely.
4. Start your application with `run` *file*$_1$*.exe*.

.

For Modula-2 programs you can use the Modula-2 compiler in the same way by compiling this files with `mod/`*command_qualifier file_name*. Look at the file `MOD$SYSTEM:modula.cld` for allowed command qualifiers. They are also explained in the VAX DBPL User's Guide.

C.1 An Example Application

Go to the directory `[.demo]`

```
$ set def [.demo]
```

The example application consists of the following files

1. `gtypes.def` and `gtypes.dbp`
2. `gdb.def`, a database module and `gdb.dbp`
3. `gdoc.def` and `gdoc.dbp`

4. `gdisplay.def` and `gdisplay.dbp`
5. `gdemo.dbp` and `gdemo2.dbp` the main program modules importing the other modules.

Compile these modules with

```
$ dbplc/log/debug gtypes.def
$ dbplc/log/debug/create gdb.def
$ dbplc/log/debug gdoc.def
$ dbplc/log/debug gdisplay.def
$ dbplc/log/debug gtypes.mod
$ dbplc/log/debug gdb.mod
$ dbplc/log/debug gdisplay.mod
$ dbplc/log/debug gdoc.mod
$ dbplc/log/debug gdemo.mod
$ dbplc/log/debug gdemo2.mod
```

Please note that the order of these operations is significant since the compiler has to read the symbol files of imported modules during compilation. By using the qualifier `/create` the persistent database variables will be created. You will therefore find some new files with extensions `.dat, .i00, .dic` in the current default directory. These files are needed during subsequent executions of the DBPL application. Now link your applications with

```
$ link/debug gdemo.obj,gtypes.obj,gdisplay.obj,gdb.obj,gdoc.obj
$ link/debug gdemo2.obj,gtypes.obj,gdisplay.obj,gdb.obj,gdoc.obj
```

Note that the main module `gdemo` (`gdemo2`, respectively) is at the first position. If you use this link command, the VAX-Debugger will be automatically activated when you start your application. If you do not need this feature, omit the `/debug` qualifier in the link-command. Now you can start the application with

```
$ run gdemo
$ run gdemo2
```

If you have used the "debug version" you should type `go` after the debugger prompt or start your application simply with

```
$ run/nodebug gdemo
```

This avoids the activation of the debugger, too.

You can use the module-management facilites (MMS) of the VAX operating system to keep track of your module dependencies and simplify the generation of an executable image (see MMS user's guide for further informations). There is a description file `descrip.mms` for the module dependencies of the demo application and the default rules for the generation of DBPL code and DBPL symbol files. Using MMS it is sufficient to type

```
$ mms
```

Bibliography

[ERMS91] J. Eder, A. Rudloff, F. Matthes, and J.W. Schmidt. Data Construction with Recursive Set Expressions in DBPL. In *Proceedings of the Kiev East/West Workshop on Next Generation Database Technology*, volume 504 of *Lecture Notes in Computer Science*, April 1991.

[JLS85] M. Jarke, V. Linnemann, and J.W. Schmidt. Data Constructors: On the Integration of Rules and Relations. In *11th Intern. Conference on Very Large Data Bases, Stockholm*, August 1985.

[KMP82] J. Koch, J. Mall, and P. Putfarken. Modula-2 for the VAX: Description of a System Portation. In H. Langmaack, B. Schlender, and J.W. Schmidt, editors, *Tagungsband Implementierung Pascal-artiger Programmiersprachen*. Teubner Verlag, 1982. (in German).

[Mod91] ISO/IEC JTC1/SC22/WG13. *Interim Version of the 4th Working Draft Modula-2 Standard*, 1991.

[MRS84] M. Mall, M. Reimer, and J.W. Schmidt. Data Selection, Sharing and Access Control in a Relational Scenario. In M.L. Brodie, J.L. Myopoulos, and J.W. Schmidt, editors, *On Conceptual Modelling*. Springer-Verlag, 1984.

[MRS89] F. Matthes, A. Rudloff, and J.W. Schmidt. Data- and Rule-Based Database Programming in DBPL. Esprit Project 892 WP/IMP 3.b, Fachbereich Informatik, Johann Wolfgang Goethe-Universität, Frankfurt, West Germany, March 1989.

[MS89] F. Matthes and J.W. Schmidt. The Type System of DBPL. In *Proceedings of the Second International Workshop on Database Programming Languages, Salishan, Oregon*, pages 255–260, June 1989.

[SBK+88] J.W. Schmidt, M. Bittner, H. Klein, H. Eckhardt, and F. Matthes. DBPL System: The Prototype and its Architecture. Esprit Project 892 WP/IMP 3.2, Fachbereich Informatik, Johann Wolfgang Goethe-Universität, Frankfurt, West Germany, November 1988.

[SEM88a] J.W. Schmidt, H. Eckhardt, and F. Matthes. DBPL Report. DBPL-Memo 112-88, Fachbereich Informatik, Johann Wolfgang Goethe-Universität, Frankfurt, West Germany, 1988.

[SEM88b] J.W. Schmidt, H. Eckhardt, and F. Matthes. Extensions to DBPL: Towards A Type-Complete Database Programming Language. Esprit Project 892 WP/IMP 3.1, Fachbereich Informatik, Johann Wolfgang Goethe-Universität, Frankfurt, West Germany, April 1988.

[SGLJ89] J.W Schmidt, L. Ge, V. Linnemann, and M. Jarke. Integrated Fact and Rule Management Based on Database Technology. In J.W. Schmidt and C. Thanos, editors, *Foundations of Knowledge Base Management*, Topics in Information Systems. Springer-Verlag, 1989.

[SL85] J.W. Schmidt and V. Linnemann. Higher Level Relational Objects. In *Proc. 4th British National Conference on Databases (BNCOD 4)*. Cambridge University Press, July 1985.

[SM90] J.W. Schmidt and F. Matthes. Language Technology for Post-Relational Data Systems. In A. Blaser, editor, *Database Systems of the 90s*, volume 466 of *Lecture Notes in Computer Science*, pages 81–114, November 1990.

[SM91a] J.W. Schmidt and F. Matthes. Naming Schemes and Name Space Management in the DBPL Persistent Storage System. In *Proceedings of the Fourth International Workshop on Persistent Object Systems, Martha's Vineyard, Massachusetts.* Morgan Kaufmann Publishers, January 1991.

[SM91b] J.W. Schmidt and F. Matthes. The Rationale behind DBPL. In *3rd Symposium on Mathematical Fundamentals of Database and Knowledge Base Systems*, volume 495 of *Lecture Notes in Computer Science.* Springer-Verlag, May 1991.

[Wir85] N. Wirth. Report on the Programming Language Modula-2. Springer-Verlag, 3rd edition, 1985.