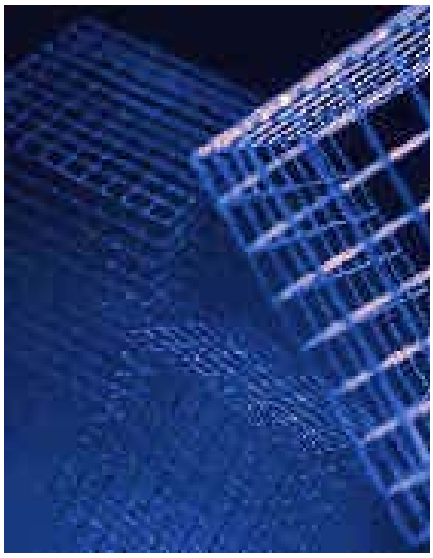
Development Environment: SAP DB



Copyright

© Copyright 2003 SAP AG.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation.

For more information on the GNU Free Documentaton License see http://www.gnu.org/copyleft/fdl.html#SEC4.

Icons

Icon	Meaning
	Caution
	Example
	Note
	Recommendation
	Syntax

Typographic Conventions

Type Style	Description
Example text	Words or characters that appear on the screen. These include field names, screen titles, pushbuttons as well as menu names, paths and options. Cross-references to other documentation.
Example text	Emphasized words or phrases in body text, titles of graphics and tables.
EXAMPLE TEXT	Names of elements in the system. These include report names, program names, transaction codes, table names, and individual key words of a programming language, when surrounded by body text, for example, SELECT and INCLUDE.
`Example text`	Screen output. This includes file and directory names and their paths, messages, source code, names of variables and parameters as well as names of installation, upgrade and database tools.
`EXAMPLE TEXT`	Keys on the keyboard, for example, function keys (such as `F2`) or the `ENTER` key.
`Example text`	Exact user entry. These are words or characters that you enter in the system exactly as they appear in the documentation.
`<Example text>`	Variable user entry. Pointed brackets indicate that you replace these words and characters with appropriate entries.

Development Environment: SAP DB 9
General Directory Structure of the SAP DB Development Environment 9
Development Hierarchy 10
Working with the SAP DB Development Hierarchy 11
Environment Variables 11
$VMAKE_VERSION 12
$VMAKE_DEFAULT 12
$VMAKE_OPTION 13
Parameters for $VMAKE_OPTION 13
$VMAKE_OPTION Parameters for Searching for Source Code 15
$VMAKE_OPTION Parameters for Searching for Object Files 15
$INSTROOT 15
$VMAKE_PATH 16
$SHELL 16
$TOOLSHELL 16
$TOOLOPT 17
$TOOL 17
$TOOLEXT 17
$DLL_TYPE 17
$ARC_TYPE 17
$OWN 18
$WRK 18
$PATH 18
%INCLUDE% 18
$RELVER 19
$CORRECTION_LEVEL 19
$BUILD_PRAEFIX 19
$PYTHONPATH 19
$PERL5LIB 19
$EDITOR 20
$DBROOT 20
$NOQUIET 20
$TOOLVARS 20
Tools in the SAP DB Development Environment 21
Tools for Operating the Development Environment 21
ims.pl; imq.pl; imf.pl 21
idiff.pl 21
ils.pl 22
ips.pl, ipq.pl and ipf.pl 22

iview.pl 22
icp.pl 23
VMAKE 23
Terms 23
Files 23
Module File 24
Module File Directory 24
Include Files 24
Include Directory 24
Description Files 24
Link Descriptions 25
Compilation Descriptions 25
Processing Lists 26
Object Files 26
Object Directory 26
Initialization Files 26
Langextinfo 27
Langinfo 28
Extra 29
DirectoryMapping 29
Files That do not Require Translation 30
Date Files 30
Layer 31
Target 31
Debug Mode 31
File Storage 32
Naming Conventions for Files 32
Naming Conventions for Module Files 32
Naming Conventions for Description Files 33
Using VMAKE 34
MAKE Operation 34
Process Flow of the MAKE Operation 35
Defining the MAKE Operation 36
VMAKE Versions 36
fast VMAKE Version 36
quick VMAKE Version 37
slow VMAKE Version 37
Grammar for the VMAKE Call 37
Options for Calling VMAKE 38
VMAKE Tools 40

Storage Location of the VMAKE Tools ... 40
Scripts for the VMAKE Tools ... 40
Selecting the Translation Tools ... 42
Options for VMAKE Tools ... 43
Creating Description Files ... 43
General Grammar for Description Files ... 44
Grammar for Link Descriptions ... 45
Grammar for Compilation Descriptions ... 46
Grammar for Processing Lists ... 46
Options for Description Files ... 47
Options for Link Descriptions ... 47
Options for Compilation Descriptions ... 48
Options for Processing Lists ... 48
! <command> ... 49
!! <command> ... 49
!? <command> ... 49
? defaultlayer : <layer> ... 50
? defaultlayer : ... 50
? defaultversion : <vmake_version> ... 50
? defaultversion : ... 50
? distribute : <list> ... 51
? link with : <list> ... 51
? linkoption : <list> ... 51
? output : <list> ... 51
? propagate : <variable>[=<value>] ... 52
? require : <target> ... 52
? tooloption : <list> ... 52
-><output> ... 53
ascii ... 53
binary ... 53
debug|d ... 54
definition ... 54
demand ... 54
demand{<relative path>} ... 54
demand=<list> ... 54
demand{<relative path>}=<list> ... 55
dep=<list> ... 55
distrib ... 55
exec ... 55
extdep=<list> ... 55

inc=<list> ... 56
interface ... 56
nobind ... 56
nodistrib ... 56
noobjcopy ... 56
noshrglob ... 57
noversion ... 57
obj=<list> ... 57
profile|p ... 57
remake ... 57
shrglob ... 58
uncond ... 58
VMAKE Logs ... 58
Translation Process ... 58
Unpacking the Module Files ... 58
Removing Frame Parts that are not Relevant for Translation ... 59
Conditional Compilation ... 59
Translating the Module Files ... 59
Link Operation ... 59
Working with the SAP DB Development Environment: Examples ... 60
Operating the Development Environment ... 60
Objective ... 60
Process Flow ... 61
Explanation of the Process Flow ... 61
Logs ... 62
Further Options ... 64
Forced Translation ... 64
Working with Debug Information ... 65
Further Information on Dependencies ... 65
Displaying the New Targets to be Generated ... 65
Displaying the Module and Description Files Used ... 66
Displaying the Module and Description Files Used, and Their Dependencies ... 66
Comparison Between SAP DB VMAKE and a Conventional Make Program ... 67
Creating DBMCLI with the SAP DB Development Environment ... 67
Structure of the Database Manager CLI ... 68
Creating the Database Manager CLI ... 68
Link Description dbmcli.lnk ... 69
Explanation of the Link Description dbmcli.lnk ... 70
Components of DBMCLI ... 70
dbmcli.rc ... 71

vcn12.cpp, vcn13.cpp, vcn14.c 71
Compilation Description 71
Include Files 72
cservlib 72
splib 73
eoxlib 73
sqlusr, enalib, enblib 73
Other Dependencies 73
Function Check 74

Development Environment: SAP DB

General Directory Structure of the SAP DB Development Environment [Page 9]

Development Hierarchy [Page 10]

Environment Variables [Page 11]

Tools in the SAP DB Development Hierarchy [Page 21]

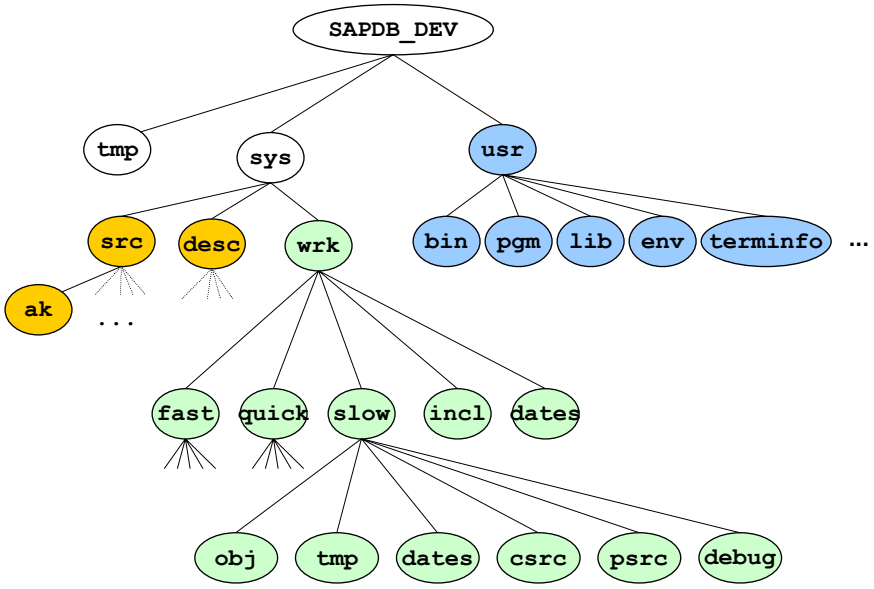
Working with the SAP DB Development Environment: Examples [Page 60]

General Directory Structure of the SAP DB Development Environment

The SAP DB development environment uses a fixed directory structure.

The top node (`SAPDB_DEV`) corresponds to the environment variable $OWN [Page 18].

The system creates the following directory structure when you call VMAKE for the first time.



The top `SAPDB_DEV` node is divided into 3 areas:

- Development area: `sys`
- Temporary area: `tmp`
- Directory for created software components: `usr`

Development Area

The development area is divided into the subdirectories `desc`, `src`, and `wrk`.

The development area is not always necessary. Using the development hierarchy [Page 10], the system accesses the sources from the next hierarchy level when the software is created.

- All types of description files [Page 24] are stored in the `desc` directory, which can also have subdirectories.
- The `src` directory contains all the module files [Page 24] in separate directories or layers [Page 31].

- The actual software creation process takes place in the `wrk` directory. All of the intermediate products (object files [Page 26], include files [Page 24]) and logs [Page 58] are also stored here.
- A separate directory (`fast, quick, slow`) with a standard structure exists for each of the three VMAKE versions [Page 36]. Each of these directories has a `tmp` subdirectory. This is the temporary work area that is used during the MAKE operation [Page 34].

  - All of the object files and static libraries created are stored in the `obj` subdirectory.
  - The date files are stored in the `dates` subdirectory.
  - The `psrc` and `csrc` directories can be used as buffers for module files if the VMAKE_OPTION parameters [Page 13] have been set accordingly.

  - The module files that were translated in debug mode [Page 31] are stored in the `debug` directory.

- Since include files are independent of the VMAKE version, they are stored directly below the `wrk` directory in the `incl` directory.
- The associated date files are stored directly below the `wrk` directory in the `dates` directory.

Temporary Area

This is a temporary area that allows the iview.pl tool to use the development environment. When the tool is called, the specified module and/or description files are copied to the tmp file. These files, however, cannot be modified in this directory.

Directory for Created Software Components

The `usr` directory for the created software components is the future DBROOT directory of the SAPDB software. All of the components that are required to operate the database are copied here.

Development Hierarchy

VMAKE provides access to module [Page 24] and description files [Page 24] across several levels. Several development states can be logically superimposed on one another in different directories (also from other computers). This provides an overview of a complete software version.

Each hierarchy level does not have to contain all the module and description files. For the translation process [Page 58], the module and description files are usually taken from the top

level in the hierarchy. This hierarchy sequence is defined through the environment variable VMAKE_PATH [Page 16].

A two-level hierarchy is implemented with the standard installation of the SAPDB development environment. This ensures that the original module and description files are always retained.

Example of a two-level hierarchy

Directory 1 (`C:\SAPDB\SAPDB_ORG`)	Contains the original SAP DB sources; these are located in the subdirectories `sys/desc` and `sys/src` (General Directory Structure [Page 9])
Directory 2 (`C:\SAPDB\SAPDB_DEV`)	Directory for creating and developing SAPDB software

Working with the SAP DB Development Hierarchy [Page 11]

Working with the SAP DB Development Hierarchy

If you want to modify files, use the icp.pl [Page 23] tool to copy them from the `SAPDB_ORG` directory to the `SAPDB_DEV` directory in the development hierarchy [Page 10].

The `SAPDB_ORG` directory is located und the following path:
`C:\SAPDB\SAPDB_ORG`

The `SAPDB_DEV` directory is located under the following path:
`C:\SAPDB\SAPDB_DEV`

In this case, the environment variable $VMAKE_PATH [Page 16] must contain the following entry:

```
VMAKE_PATH= C:\SAPDB\SAPDB_DEV,C:\SAPDB\SAPDB_ORG
```

In a MAKE operation, SAPDB_DEV would now be searched for a module or description file. VMAKE only searches the next level if it cannot find the file here.

If a path in the VMAKE_PATH [Page 16] does not end with two slashes, it is also searched for current object files [Page 26] in the `wrk` directory. The files are copied if they match the current development status. The relevant module files do not need to be translated.

You can use the idiff.pl [Page 21] tool to compare the modified and original module or description files.

The iview.pl [Page 22] tool enables you to display a module or description file.

Environment Variables

The VMAKE initialization process is based on environment variables.

In Windows NT, environment variables must be encapsulated in percentage symbols.

On UNIX systems, environment variables are prefixed with a $. This is also the selected notation in this documentation.

All of the environment variables listed here, with the exception of $NOQUIET, can be set with the `initDev_SAPDB.bat` (Windows NT) or `initDev_SAPDB` (UNIX) initialization script. This script is stored in the `SAPDB_DEV` directory when the development environment is installed.

$VMAKE_VERSION [Page 12]	$WRK [Page 18]
$VMAKE_DEFAULT [Page 12]	$PATH [Page 18]
$VMAKE_OPTION [Page 13]	%INCLUDE% [Page 18]
$INSTROOT [Page 15]	$RELVER [Page 19]
$VMAKE_PATH [Page 16]	$CORRECTION_LEVEL [Page 19]
$SHELL [Page 16]	$BUILD_PRAEFIX [Page 19]
$TOOLSHELL [Page 16]	$PATHONPATH [Page 19]
$TOOLOPT [Page 17]	$PERL5LIB [Page 19]
$TOOL [Page 17]	$EDITOR [Page 20]
$TOOLEXT [Page 17]	$DBROOT [Page 20]
$DLL_TYPE [Page 17]	$NOQUIET [Page 20]
$ARC_TYPE [Page 17]	$TOOLVARS [Page 20]
$OWN [Page 18]	

$VMAKE_VERSION

This variable defines the system default for the VMAKE version [Page 36].

VMAKE only evaluates the first letter of the set variable.

```
set VMAKE_VERSION=fast
```

$VMAKE_DEFAULT

This variable is optional. If this variable is not set, VMAKE uses the entry for the $VMAKE_VERSION [Page 12] variable. VMAKE only evaluates the first letter of the set variable.

It defines the system default for translating module files that are listed in description files.

```
set VMAKE_DEFAULT=fast
```

If translation is carried out with the `slow` VMAKE version and the $VMAKE_DEFAULT set to `fast`, only the module files in the description files are translated with the `fast` method; all other files are translated using the `slow` method.

$VMAKE_OPTION

You use the parameters [Page 13] for the $VMAKE_OPTION environment variable to define the VMAKE mode of operation.

You can override some of the $VMAKE_OPTION parameters by entering options for the VMAKE call.

If you want to specify several parameters, do not enter a blank space between the individual parameters.

```
set VMAKE_OPTION=DISOhvWMLRg
```

Parameters for $VMAKE_OPTION

Different parameters can be set for the $VMAKE_OPTION [Page 13] variable.

If you want to specify several parameters, do not enter a blank space between the individual parameters.

Parameters for Searching for Source Code [Page 15]

Parameters for Searching for Object Files [Page 15]

General Parameters for $VMAKE_OPTION

A	Object files [Page 26] are also copied to the first hierarchy level [Page 10] of $VMAKE_PATH if they can be found in an integrated file system.
a	A warning is output instead of an error message for differences regarding upper and lower-case notation of a target [Page 31] in the hierarchy. Discrepancies regarding upper and lower-case letters in the targets within the hierarchy are usually regarded as an error.
b	Normally, C module files are not allowed to use PASCAL include files [Page 24]. If this option is set, the C header files generated from the PASCAL include file are used.
C | c	C Level: The same rules apply as with the Pascal level, with the exception that no PASCAL module files are allowed to exist here. If PASCAL and C Level are used, non-PASCAL files are copied from the PASCAL Level to the C Level during the translation process [Page 58]. C | c, therefore, implies that a C Port is to be carried out. PASCAL files are translated into C files. C requires that source code be managed at this level. If you use c, this is optional. In other words, the source text from this level is used if it exists.
D	Date files are used to flag the date of a module file that was used to create a program.
e	All object files are assigned debug information.
g	global make: If objects are found in the path hierarchy that are younger than the those in the first hierarchy, they are copied to the first hierarchy (standard behavior).

h	If you specify a description file without an extension in the command line, VMAKE attempts to find a description file with a file name that matches the one entered in the command line. The system follows a specific search sequence. With the h option, you can assign description files the following order in accordance with their file extension: the system first searches for a file with this name and the extension shm, then with the extension lnk, then with the extension shr, and finally with the file extension rel.
H	PASCAL include files are not automatically converted to C header files.
I | i	Include Level: Include files are assigned a frame. This must be removed, and the include file must be transferred to the include directory [Page 24]. If neither the i nor the I option is set, include files are not interdependent. If i is specified, include files are subject to the standard dependency rules. If I is specified, include files are generated with size information for PASCAL. Include files are only considered if one of the S | s options is set.
l	local make: Object files whose module files are located in the first path are regarded as being up to date. This means that object files from the path hierarchy are never used.
L	Libraries are like relocatables: The contents of libraries are not analyzed. The date of the library is checked against the timestamp of the object files. The library is rebuilt by linking all of the relevant object files.
M	Modules are independent: Object files are stored in the file system, as they cannot be accessed otherwise. Alternatively, object files from programs that have already been linked can be used.
O | o	Object Level: This level contains object files that can be linked to generate programs. If O is set, these files are used accordingly. If o is specified, program generation ends once all of the object files have been created.
P | p	PASCAL Level: The frame is missing from all module files. PASCAL Level, since most module files are written in PASCAL and, therefore, PASCAL module files exist at this level. Files in different languages are stored here. If P is specified, PASCAL Level files remain in the PASCAL Level directory. P requires that source code be managed at this level; with p, this is optional, in other words, the source code from this level is used if it exists.
R	The shrglob and noshrglob options for description files [Page 47] are ignored.
S | s	Source Level: Many module files have a frame that has to be removed before they are translated. If S is specified, all of the module files have to be present. If s is set, these files are only used if they exist. Dependent files are only processed if the source code is present.
t	timestamp the start of bigger targets If description files are translated, a timestamp is output.
T	A timestamp is always output before a module file is translated.
X	Specify this option if write-protected source code (for example, from CD) is to be accessed.
Z	The system searches for module files, whose name does not contain a coded layer [Page 31] and which ends in .rc, .ico, .def, .mc, .dlg, .idl, .ycc, .lex or .rgs, in the Resource layer.

$VMAKE_OPTION Parameters for Searching for Source Code

Different parameters [Page 13] can be set for the $VMAKE_OPTION [Page 13] variable.

If you want to specify several parameters, do not enter a blank space between the individual parameters.

Parameters for Searching for Source Code

(No parameter set)	The first item of source code found is used.
`v`	The first item of source code found is used. The system outputs a warning if a more recent file is found in the remaining path hierarchy.
`V`	The most recent source code in the hierarchy is used and, if necessary, copied to the local source area.
`vV`	The most recent source code in the hierarchy is used and a warning is output.

$VMAKE_OPTION Parameters for Searching for Object Files

Different parameters [Page 13] can be set for the $VMAKE_OPTION [Page 13] variable.

If you want to specify several parameters, do not enter a blank space between the individual parameters.

Parameters for Searching for Object Files

(No parameter set)	The first object file found is used.
`w`	The first object file found is used. The system outputs a warning if a more recent object file is found in the remaining path hierarchy.
`W`	The most recent object file in the hierarchy is used.
`wW`	The most recent object file in the hierarchy is used and a warning is output.

$INSTROOT

This variable contains an absolute path specification within the current file system. This describes where VMAKE is to store linked software components.

VMAKE decides, on the basis of the software component, under which relative path under $INSTROOT the component will be stored.

```
set INSTROOT=D:\V72
```

$VMAKE_PATH

This variable lists paths under which VMAKE can search for files [Page 23].

The individual path specifications are separated by a comma. The paths can be absolute paths in the file system of the local computer, or of other computers that are integrated in the file system.

If a file system that is not integrated is to be used, the VMAKE Server program is required on the computer with this file system.

The path must then be specified as follows:

<server_name>:<absolute_path>

<server_name>	Name of the other computer
<absolute_path>	Absolute path on the file system of the other computer

If you enter two slashes at the end of a path specification, the system will ignore this path when searching for object files [Page 26].

```
set
VMAKE_PATH=D:\V72,P26326:F:\Develop//,\\P26208\share\tools
```

$SHELL

Description files [Page 24] can contain operating system commands.

You use this variable to specify that a certain command shell is to be used to process these OS commands.

```
set SHELL=C:\MyTools\4dos.exe
```

$TOOLSHELL

This variable defines the shell that executes the scripts used by VMAKE (VMAKE Tools [Page 40]).

```
set TOOLSHELL=D:\Perl\5.00502\bin\MSWin32-x86-
object\perl5.00502.exe
```

$TOOLOPT

You can use this variable to transfer additional options to the shell specified in the $TOOLSHELL [Page 16] variable.

```
set TOOLOPT=-d
```

$TOOL

The relative path containing the scripts and programs used by VMAKE is specified in this variable.

```
set TOOL=D:\DevTool
```

$TOOLEXT

The file extension of the scripts (VMAKE Tools [Page 40]) is defined in this variable.

VMAKE only recognizes the first part of the script file names. VMAKE uses this variable to determine the file extension.

```
set TOOLEXT=.pl
```

$DLL_TYPE

The file extension for dynamic libraries is defined in this variable.

This variable is optional.

VMAKE has the following default settings:

Win32	`.dll`
UNIX	`without file extension`

$ARC_TYPE

The file extension for static libraries is defined in this variable.

This variable is optional.

Default settings:

Win32	`.lib`
UNIX	`.a`

$OWN

The environment variable OWN corresponds to the absolute path for the SAPDB_DEV directory (General Directory Structure of the SAP DB Development Environment [Page 9]).

This environment variable is set by the initialization script initDev_SAPDB(.bat) (Environment Variables [Page 11]).

$WRK

The environment variable $WRK is used by various description files. It refers to the sys/wrk directory below the SAPDB_DEV directory (General Directory Structure of the SAP DB Development Environment [Page 9]).

This environment variable is set by the initialization script initDev_SAPDB(.bat) (Environment Variables [Page 11]).

$PATH

The $PATH variable must be adapted to include the following directories:

$DBROOT/bin

$DBROOT/pgm (NT)

$TOOL/Posix

$TOOL/bin

$TOOL/pgm (NT)

This environment variable is set by the initialization script initDev_SAPDB(.bat) (Environment Variables [Page 11]).

%INCLUDE%

This environment variable only applies to Windows NT.

The compiler uses this environment variable to locate all of the SAPDB-internal include files.

The following entries must be listed here:

%DBROOT%\incl

%TOOL%\incl

%OWN%\sys\wrk\incl

%OWN%\sys\wrk\incl\SAPDB

This environment variable is set by the initialization script `initDev_SAPDB(.bat)` (Environment Variables [Page 11]).

$RELVER

This environment variable specifies the current software version. It generates the build information in the program components, and is used by the link tools.

This environment variable is set by the initialization script `initDev_SAPDB(.bat)` (Environment Variables [Page 11]).

$CORRECTION_LEVEL

This environment variable specifies the number of the correction level for the build information. This information is also used by the link tools.

This environment variable is set by the initialization script `initDev_SAPDB(.bat)` (Environment Variables [Page 11]).

$BUILD_PRAEFIX

This environment variable specifies the patch level number of the current software version. This information is also used by the link tools.

This environment variable is set by the initialization script `initDev_SAPDB(.bat)` (Environment Variables [Page 11]).

$PYTHONPATH

This environment variable must be set so that the Python scripts in the tools can be executed.

It should contain the following entries:

$TOOL\Lib\Python

$INSTROOT/lib

This environment variable is set by the initialization script `initDev_SAPDB(.bat)` (Environment Variables [Page 11]).

$PERL5LIB

This environment variable must be set so that the Perl scripts in the tools can be executed.

It should contain the following entries:

$INSTROOT/misc

$TOOL/bin

$TOOL/Lib/Perl

This environment variable is set by the initialization script `initDev_SAPDB(.bat)` (Environment Variables [Page 11]).

$EDITOR

This environment variable specifies the program that is used to display and edit files using the development environment operating tool iview.pl [Page 22].

This environment variable is set by the initialization script `initDev_SAPDB(.bat)` (Environment Variables [Page 11]).

$DBROOT

This environment variable is an obsolete name for the $INSTROOT [Page 15] variable. As a result, it should be set in the same way as $INSTROOT.

This environment variable is set by the initialization script `initDev_SAPDB(.bat)` (Environment Variables [Page 11]).

$NOQUIET

If this environment variable is set, every call of the compiler is output during the creation process.

$TOOLVARS

The path specification in this environment variable refers to the central initialization file `toolvars.pl` for most Perl scripts. The default storage location for this file is the `$TOOL/bin` directory.

A large part of the platform-specific adjustments is made in this Perl file. This includes the definition of the various platform-specific compiler, link, and other programs, as well as the associated options. Certain additional environment variables that are required for the creation process are also initialized in this file.

This environment variable is set by the initialization script `initDev_SAPDB(.bat)` (Environment Variables [Page 11]).

Tools in the SAP DB Development Environment

The development tools consist of the following components:

- Tools for Operating the Development Environment [Page 21]
- VMAKE [Page 23], as a software creation tool
- Various auxiliary programs

The various tools in the SAP DB development environment are essentially Perl scripts. These can be called with various options. You can use the `-h` option to display a list of the available options.

Tools for Operating the Development Environment

ims.pl; imq.pl; imf.pl [Page 21]

idiff.pl [Page 21]

ils.pl [Page 22]

ips.pl, ipq.pl und ipf.pl [Page 22]

iview.pl [Page 22]

icp.pl [Page 23]

ims.pl; imq.pl; imf.pl

These commands are tools for operating the development environment [Page 21] and are used to create SAP DB software components. A distinction is made here between the three VMAKE versions [Page 36]. At least the name of a processing list [Page 26], a link description [Page 25], or a module file [Page 24] is expected as a parameter. You can also specify several names as parameters.

Once a defined environment has been initialized, these scripts call the VMAKE program.

All of the available options for calling VMAKE [Page 38] are also available for these scripts and are passed on directly to VMAKE.

Specify the name of the file (and not the complete path) in accordance with the naming conventions for files [Page 32] for all of the file names transferred as parameters.

Example [Page 60]

idiff.pl

This command is a tool for operating the development environment [Page 21] and compares the file specified as a parameter in the local development area (General Directory Structure [Page 9]) with an older version of the same file within the development hierarchy.

You can also use the –I option [Page 38] to compare the same files at different levels in the development hierarchy.

Specify the name of the file (and not the complete path) in accordance with the naming conventions for files [Page 32] for all of the file names transferred as parameters.

ils.pl

This command is a tool for operating the development environment [Page 21], and is similar to the commands `dir` or `ls` in other operating systems. The module [Page 24] or description files [Page 24] specified as parameters within the development hierarchy are displayed.

Specify the name of the file (and not the complete path) in accordance with the naming conventions for files [Page 32] for all of the file names transferred as parameters.

ips.pl, ipq.pl and ipf.pl

These commands are tools for operating the development environment [Page 21], and are used to display both of the VMAKE logs [Page 58] of the translation process [Page 58].

There are three different commands for the three VMAKE versions [Page 36]. The logs from a creation process started with imf.pl [Page 21] can be displayed with the `ipf.pl` tool. The file name used with the `imf.pl` command must be specified as a parameter here:

```
imf.pl   dbmcli
ipf.pl    dbmcli
```

iview.pl

This command is a tool for operating the development environment [Page 21], and is used to display and edit module [Page 24] and decription files [Page 24].

If the files are located in the `src` or `desc` subdirectory below the `SAPDB_DEV` directory (General Directory Structure [Page 9]), they are opened directly and can be modified accordingly.

If the are not located in these subdirectories, the system searches for their first occurrence in the development hierarchy and copies them to the `tmp` subdirectory below the `SAPDB_DEV` directory so that they can be displayed.

The files are displayed with the program that is referred to by the environment variable $EDITOR [Page 20].

```
iview vcn12.cpp
iview dbmcli.lnk
```

You can also use the -l option [Page 38] to access all of the versions of a file within the development hierarchy, even though they are also located in the `SAPDB_DEV` directory.

```
iview -l1 co.com
```

Specify the name of the file (and not the complete path) in accordance with the naming conventions for files [Page 32] for all of the file names transferred as parameters.

Changes to the files that are relevant for translation can only be carried out if the files are directly located in the development area. You can copy these files with icp.pl [Page 23].

icp.pl

This command is a tool for operating the development environment [Page 21], and is used to copy a module [Page 24]- or description file [Page 24] from the development hierarchy to the development area in the `SAPDB_DEV directory`. As a result, this file is used instead of the file in the development hierarchy the next time a MAKE operation [Page 34] is carried out.

You can use the iview.pl [Page 22] command to display the copied file, an then edit it.

This command can only be used to copy one file. If you specify a second file name as a parameter in addition to the file name, the name of the copied file is changed to the one in the second parameter.

Specify the name of the file (and not the complete path) in accordance with the naming conventions for files [Page 32] for all of the file names transferred as parameters.

VMAKE

This document is designed as a reference guide for creating or maintaining programs, libraries, and so on with the VMAKE tool.

The VMAKE tool supports all versions of the SAP DB software.

To develop the SAPDB software, or parts thereof, enter the command for the required VMAKE version [Page 36].

Various options [Page 38] are provided for modifying the call for the VMAKE tool.

Terms

Files [Page 23]

Layer [Page 31]

Target [Page 31]

Debug Mode [Page 31]

Files

When software is created, module files [Page 24] are translated using description files [Page 24].

The resulting files are called object files [Page 26].

The <file_name> of a file is preserved across the various phases of the MAKE operation. The file extension <ext> changes.

Module File

Source code comprises individual files. These files are referred to as module files.

Module files can be combined in module groups [Page 31].

Module files are stored separately from description files [Page 24] in the file system of the OS (Storage System [Page 32]).

Module files can be written in any programming language. VMAKE features a complete set of tools [Page 40] for unpacking [Page 58] and translating [Page 59] module files for all of the programming languages supported by SAP DB.

Module File Directory

The module file directory is used to store the module files [Page 24] .

The structure of this directory is identical to that of the object directory (General Directory Structure of the SAP DB Development Environment [Page 9]).

Include Files

An include file is a module file [Page 24] that is included in a different module file.

When the source code of an include file changes, both the file and the file, in which this include file is integrated, must be retranslated (Translation Process [Page 58]) as part of the software creation process.

Include files are independent of VMAKE versions [Page 36].

Include Directory

Unlike module files, include files [Page 24] are independent of the called VMAKE version (VMAKE Call Versions [Page 36]) and, therefore, are always stored in the same directory.

This directory is called `incl` and is created by the VMAKE tool (General Directory Structure [Page 9]).

Description Files

Description files contain information on how the software is created, and which module files [Page 24] are used to do so. This information includes:

- Link descriptions [Page 25]
- Compilation descriptions [Page 25]
- Processing lists [Page 26]

Link Descriptions

Link descriptions specify how the module files [Page 24] are to be linked to form a program.

Essentially, link files consist of a list of references to module files and other description files [Page 24] that are processed sequentially by the system. You can also specify the options that are to be used during the link operation [Page 59].

The file extension of the link description is used to distinguish between the resulting software components (Using VMAKE [Page 34]).

File Extension of Link Description	Type of Software Component Created
`lnk`	Program
`shm`	Program that facilitates shared memory access
`lib`	Static library
`dld`	Dynamic library
`rel`	Program that can be linked to other programs (relocatable)
`shr`	Program that can be linked to other programs (relocatable) and facilitates shared memory access
`mac`	Other types of object files

Compilation Descriptions

Use

Compilation description specify how module files [Page 24] are to be translated. Essentially, a compilation description contains a list of module files and their interdependencies, as well as translation-specific options [Page 48]. These interdependencies include:

- Include dependencies (include file [Page 24])
- Dependencies, such as certain auxiliary files that must be located in the same directory as the module file to be translated (demand=<list> [Page 54]) during the translation process [Page 58]
- Logical dependencies – if, for example, the module file `X` links to a file `Z` that is created as a by-product of the translation of the module file `Y` (extdep=<list> [Page 55])

Compilation descriptions are created for groups of module files, and not for individual module files. The compilation description has the same name as the layer [Page 31]. Compilation descriptions have the file extension `.com`

If the layer `ak` exists in the directory containing the module files (`src`), the directory with the description files (`desc`) must contain the compilation description `ak.com` .

If a layer (subdirectory) under the directory with the module files [Page 24] (`src`) contains further sublayers, the hierarchy must be recreated up to the second last level in the directory

containing the description files (`desc`). The name of the compilation description is then formed from the last sublayer and the file extension `.com`

The compilation description for module files in the `SAPDB/DBM/Cli` layer has the name `Cli.com` and is located in the directory containing the description files (`desc`) under `SAPDB/DBM`.

Compilation descriptions are not processed sequentially. The processing points of a description of this kind are addressed selectively.

Processing Lists

Use

A processing list is a list of description files [Page 24] and module files [Page 24] to be processed, as well as OS commands to be executed. A processing list can also contain other processing lists. In addition, options [Page 48] can be specified.

The entries in a processing list are processed by VMAKE from top to bottom. All of the listed files, as well as any other files listed as links, are first checked down to the lowest level to determine whether they have been changed with respect to a target created previously. Only if this is the case does VMAKE recreate (update) the targets.

Object Files

Module files [Page 24] are unpacked and translated. The files created as a result are called object files.

These files are usually stored in the directories that correspond to the VMAKE version (VMAKE Call Versions [Page 36]), that is, under `fast`, `quick` or `slow` (see General Directory Structure of the SAP DB Development Environment [Page 9]).

If the noversion [Page 57] option is specified in the compiler description for a module file, the object file is stored directly in the **`wrk/obj`** directory.

Object files have the file extension `.o`

Object Directory

The object files [Page 26] are stored in the object directory.

The object directory has the same structure as the source code directory (General Directory Structure of the SAP DB Development Environment [Page 9]).

Initialization Files

VMAKE uses the following initialization files:

Langextinfo [Page 27]	Contains the assignments between the module files [Page 24] and the translation tools, based on the file extension or the last character

	in the file name.
Langinfo [Page 28]	Contains assignments between the programming languages and certain tools for unpacking [Page 58] and translating [Page 59] module files [Page 24].
EXTRA [Page 29]	Contains assignments between files that do not require translation [Page 30] and their storage location.
DirectoryMapping [Page 29]	This file is used to assign module files to a layer [Page 31], on the basis of the first part of their file name (directory mapping function).

These files can be modified.

The files are stored in the `desc` directory (General Directory Structure of the SAP DB Development Environment [Page 9]).

Langextinfo

Use

`Langextinfo` is an initialization file [Page 26] of the VMAKE tool. This file is used to assign file extensions to a programming language. Each programming language is identified by a letter.

VMAKE uses certain default settings for each language.

If you want to change the existing assignments, create a file with the name `Langextinfo` in the `desc` directory. You can also assign new letters, programs, and file extensions. In this case, however, you must create an appropriate Langinfo [Page 28] file. `All further changes are then made in this file.`

`If VMAKE finds a Langextinfo file, the information contained in the file overrides the system defaults.`

Syntax

```
<langextinfo_line> = <comment_line> |
(<lang_id><comment><token_sep><token>{<token_sep>,<token_sep><token>}
<new_line>)
```

General Rules

`<comment>` must not contain `<token_sep>`

Lines can contain the comment character `#`. The subsequent text in this line is treated as a comment.

System Default in VMAKE

Assigned letter (program)	File extension(s)
x(C++)	`cpp, hpp`
c(C)	`c, h`
t(PASCAL)	`t`
a(Assembler)	`s`
r(Resources)	`rc, ico, def, mc, dlg, idl, ycc, lex, rgs`

Langinfo

Use

`Langinfo` is an initialization file [Page 26] of the VMAKE tool. The file is used to assign certain tools for unpacking [Page 58] and translating [Page 59] module files [Page 24] to programming languages. The relevant programming language is specified by a letter (Langextinfo [Page 27]).

VMAKE uses certain default settings for each language.

If you want to change the existing assignments, create a file with the name `Langinfo` in the `desc` directory. `All further changes are then made in this file.`

`If VMAKE finds a Langinfo file, the information contained in the file overrides the system defaults.`

Syntax

```
<langinfo_line> = <comment_line> |
(<lang_id><token_sep><unpack_inc_tool><token_sep><unpack exp
tool><tokensep><unpack mod
tool><tokensep><extension><tokensep><option prefix><tokensep>
<compiler tool>)

<unpack inc tool> = <token>

<unpack exp tool> = <token>

<unpack mod tool> = <token>

<extension> = „."<token>

<option prefix> = <ext character><letter>

<compiler tool>  = <token>
```

General Rules

Lines can contain the comment character `#`. The subsequent text in this line is treated as a comment.

The scripts for unpacking a module file use the table entry under *extension* as the file extension.

System Default in VMAKE

#	last char	unpack include	unpack exports	unpack module	extension	option prefix	compiler	
	t	mfpinc	mfpexp	mfp	. p	%T	comppc	# PASCAL
	p	mfpinc	mfpexp	mfp	.p	%P	compp	# PASCAL
	c	mfcinc	mfcexp	mfc	.c	%C	compc	# C
	x	mfcinc	mfcexp	mfc	.cpp	%X	compc	# C++
	a	mfainc	mfaexp	mfa	.s	%A	compa	# Assembler
	r	mfcinc	mfcexp	mfrc	.rc	%R	comprc	# Resources

Extra

`Extra` is an initialization file [Page 26] of the VMAKE tool.

This file contains assignments between files that do not require translation [Page 30] and their storage location.

Syntax

```
<extra_line> = <comment_line> |
(<extension><token_sep><layer><token_sep><dest_dir><token_sep><option
><new_line>)
```

#	*Extension*	*Layer*	*Destination directory*	*Options*
	`.ino`	`xx`	`usr/env`	`unpack=no`
	`.mp`	`px`	`usr/bin`	`unpack=no nodot`
	`.dm`	`px`	`tmp`	`exec`

Explanation:

Module files [Page 24] with the file extension `.ino` belong to the `xx` layer [Page 31]. They must be copied to the `usr/env` directory in $INSTROOT [Page 15]. The module file does not contain any textual frame (option `unpack=no`).

Module files with the file extension `.mp` belong to the `px` layer. These must be copied to the `usr/bin` directory (relative to `$INSTROOT`). The module file does not contain any textual frame. Once the files have been copied, the extension `.mp` must be removed (option `unpack=no dot`).

Module files with the file extension `.dm` belong to the `px` layer. These must be copied to the `tmp` directory (relative to `$INSTROOT`). The module file contains a textual frame that must be removed. Once the frame has been removed, the file is identified as executable (exec [Page 55] option, only relevant for UNIX).

General Rule

Lines can contain the comment character `#`. The subsequent text in this line is treated as a comment.

DirectoryMapping

Use

`DirectoryMapping` is an initialization file [Page 26] of the VMAKE tool. This file is stored in the <path>/sys/desc [Page 9] directory.

VMAKE uses the second and third letters of the file name to assign module files [Page 24] to a layer [Page 31]. An alternative method of assigning module files to a layer is to use the DirectoryMapping function. This method also allows you to use longer and more meaningful file names.

File names specified with the DirectoryMapping function consist of two parts, which are linked by an underscore. The first part specifies the layer in short. The second part specifies the name of the module file.

```
<file_name> = <short_layer>_<module_file>
```

The short form is mapped to the corresponding layer in the `DirectoryMapping` file.

Syntax

```
<extra_line> = <comment_line> | (<short_layer>,
<full_layer><new_line>)
```

General Rules

1. The `<full_layer>` expression stands for the layers below `<path>/sys/src/SAPDB`.
2. Lines can contain the comment character `#`. The subsequent text in this line is treated as a comment.
3. The layer requires a compilation description, in which the structure of the layers, including the second last layer, is specified.

   The `<path>/sys/src/DirectoryMapping` file contains the following entries:

   ```
   # <short_layer>, <directory>
   # Component SQLStudio
   StudioTD, SQLStudio/TableDefinition    # Table Definition
   ```

   If the `StudioTD_TableDef.cpp` and `StudioTD_Wrapper.cpp` files are to be created, the compilation description `TableDefinition.com` must be stored under `<path>/sys/desc/SAPDB/SQLStudio/`.

   VMAKE searches for the `StudioTD_TableDef.cpp` and `StudioTD_Wrapper.cpp` files in the `<path>/sys/src/SAPDB/SQLStudio/TableDefinition` layer.

Files That do not Require Translation

Certain files contain source code that is already in the required text or binary format. These files do not need to be translated. They should simply be copied to the object directory [Page 26], but can be manipulated, if necessary, using the `mfextra` tool (VMAKE Tools [Page 40]).

Use the ascii [Page 53] or binary [Page 53] options to prevent these files from being translated by VMAKE.

Alternatively, you can add an entry in the EXTRA [Page 29] initialization file.

Date Files

A date file is created for every target [Page 31] that is to be generated.

The name of the date file comprises the name of the processed file [Page 23] and the file extension `.dat`: `<file_name>.dat`

The timestamp is identical to the timestamp of the corresponding module file [Page 24] or link description [Page 25].

Use

The date file is used during the MAKE operation to determine whether the local targets correspond to the module files/link descriptions that are currently being used. If the two files do not have the same timestamp, the target is regarded as not up to date and is regenerated.

Storing the Date Files

Date files for include files [Page 24] are stored in the `$WRK/dates` directory within their layer [Page 31] irrespective of the VMAKE version [Page 36] used in the MAKE operation, since include files are not version specific.

Date files for all other module files are stored in the version-specific `$WRK/<version>/dates` directory, under the relevant subdirectories for the layer in question.

Date files for link descriptions are stored directly in the version-specific `$WRK/<version>/dates` directory.

Layer

Layers are logical units that represent subdirectories in the module file directory [Page 24].

Module files are stored in various layers and subordinate layers. In addition to providing a clearer overview, this enables a logical distinction to be drawn between the individual database areas.

Target

Targets are all types of output files that are created during the entire software development process.

These can be individual object files [Page 26], link descriptions [Page 25], or output files of files that do not require translation [Page 30].

Debug Mode

If debug mode is activated, all of the module files [Page 24] with a debug option are compiled so that the resulting object files [Page 26] contain additional debugging information. The relevant module files are copied to the version-specific debug directory `$WRK/<version>/debug`, from where they can be used for subsequent troubleshooting.

You can use the following methods to set a debug option for module files:

- -e Option for Calling VMAKE [Page 38]
- VMAKE_OPTION Parameter e [Page 13]

- with the option for description files debugId [Page 54]

File Storage

The module files [Page 24] and description files [Page 24] are stored in the file system of the OS.

The files required to create a program can be stored at different locations in a file system, in different file systems, and on different computers.

The potential paths are specified in the `$VMAKE_PATH` environment variable. This variable must be adapted accordingly to be able to access older file versions.

There is a separate directory for description files, and a separate directory for module files.

Each of these directories has at least one subdirectory that reflects the logical hierarchy of the files. Files are only stored at the lowest directory level.

The files can also be stored in a database. As a rule, however, databases do not provide adequate support for search operations, with the result that the files cannot be accessed efficiently.

Naming Conventions for Files

The names of module files [Page 24] and description files [Page 24] are subject to certain conventions.

Basic Elements

```
<digit> = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

```
<letter> = A | B | C | D | E | F | G | H | I | J | K | L | M | N | O
| P | Q | R | S | T | U | V | W | X | Y | Z | a | b | c | d | e | f |
g | h | i | j | k | l | m | n | o | p | q | r | s | | u | v | w | x |
y | z
```

`<extended letter>` = all other printable characters, apart from hyphen and period

```
<character> = <digit> | <letter> | <extended letter>
```

```
<token> = <character>{<character>}
```

Naming Conventions for Module Files [Page 32]

Naming Conventions for Description Files [Page 33]

Naming Conventions for Module Files

```
<source_spec> = <qual_source> | <std_source> | <ext_std_source> |
<map_source>
```

```
<qual_source> = :<ext_layer><source>
```

```
<std_source> =
<source_id><layer><digit><digit>{<digit>}[{<letter>}<lang_id>][<varia
nt>]
```

```
<ext_std_source> = <std_source>[<primary_lang_id>]
<map_source> = <shortform>_<specification>
<source> = <token>[<variant>]<primary_lang_id>
<source_id> = g | h | i | v
<ext_layer> = [<ext_layer>]<character><character>{< character >}/
<shortform> = <letter>{<letter>}
<specification> = <letter>{<letter>}[<variant>][<primary_lang_id>]
<lang_id> = <letter>
<layer> = <letter><letter>{<letter>}
<primary_lang_id> = .<token>
<variant> = -<token>
```

General Rules

1. `<layer>` and `<ext_layer>` can contain a maximum of 256 characters
2. `<source_spec>` can contain a maximum of 512 characters

```
<qual_source> ⇒ :Demo/Test/demo.cpp
<std_source> ⇒ hak33c | vbd23x | vkb00
<ext_std_source> ⇒ hak33.h | vbd23x.cpp | vkb00.hxx
<map_source> ⇒  StudioTD_TableDef.hpp |
StudioTD_Wrapper.cpp
```

Naming Conventions for Description Files

```
<desc_spec> = <qual_desc> | <std_desc>
<qual_desc> = ::[<ext_layer>]<std_desc>
<std_desc> = <character>{<character>}<desc_ext>
<desc_ext> = .(mac | prj | shm | lnk | dld | shr | rel | lib | com)
```

General Rule

`<desc_spec>` can contain a maximum of 512 characters

```
<qual_desc> ⇒ ::x_demo.lnk | ::Kern/diagnose.lnk
<std_desc> ⇒ ak1.lib | kernel.shm | ak.com
```

Using VMAKE

Use

The procedure for creating SAP DB software with VMAKE is divided into two phases:

- Translation Process [Page 58]
- Link Operation [Page 59]

The following software components can be generated:

- Programs
- Programs that facilitate shared memory access
- Static libraries
- Dynamic libraries
- Program that can be linked to other programs (relocatable)
- Programs that can be linked to other programs (relocatable) and facilitate shared memory access

MAKE Operation

The MAKE operation is divided into a collection phase and a make phase.

The individual steps carried out by VMAKE during the MAKE operation are described in the section entitled Process Flow of the MAKE Operation [Page 35].

Collection Phase

In the collection phase, the description files [Page 24] of a target [Page 31] are collected and analyzed. The interdependencies between all of the files are collected at the same time. The system does not yet search for the module files [Page 24] in the hierarchy.

If errors occur during this phase, VMAKE terminates without starting the MAKE phase.

MAKE Phase

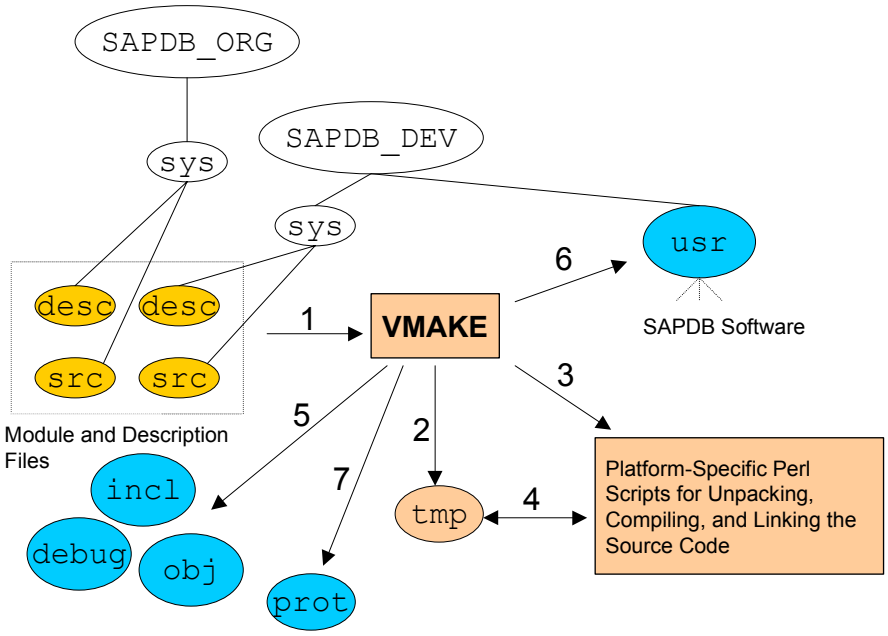
The MAKE phase marks the actual beginning of the creation process for the SAP DB software. During this phase, the module files are created in the order of their interdependency.

If a module file `vak10` is dependent on `vak00` (`vak10 inc=vak00`), vak00 is handled first. `vak10` cannot be created until this file has been successfully generated.

If an error occurs while a module file is being created, all the targets that are dependent on this module file will not be created either. This applies across all dependency levels.

If an error occurs while `vak00` is being created, `vak10` will not be processed, as well as other dependent files:
since `vak10` is contained in `ak1lib.lib`, `ak1lib.lib` is not linked. Since `ak1lib.lib` is contained in the `kernel.shm` kernel, the kernel is also not linked in the result.

Process Flow of the MAKE Operation



At the beginning of a MAKE operation [Page 34], VMAKE uses the date files [Page 30] to determine the targets that have to be recreated. The system then searches for all of the relevant module files in the hierarchy **(1)** and copies them to the temporary working directory `tmp` (General Directory Structure [Page 9]) **(2)**.

The files are then further processed in this directory with VMAKE tools [Page 40]. These tools are essentially Perl scripts **(3)**. VMAKE selects the appropriate Perl scripts on the basis of the programming language in the module files (Selecting the Translation Tools [Page 42]). The Perl scripts are platform specific, and call the respective compilers, translators, and unpackers.

The output files produced as a result (object files [Page 26] and include files [Page 24]) are initially also stored in the temporary working directory **(4)**. VMAKE then copies the object files to the respective `obj` directory and the include files to the `incl`-directory for the relevant layer **(5)**.

If VMAKE is operating in debug mode [Page 31], the module files are also copied to the corresponding debug directory.

A date file [Page 30] is stored for every module file that is used to create a target.

The relevant tools also perform the link operations in the temporary directory, based on the applicable link descriptions. The output files are also stored here. Following this, VMAKE copies the targets to the relevant directories **(6)**.

The files located in the temporary working directory are deleted once the targets have been successfully created. If an error occurs while the targets are being created, the files are not deleted in the temporary directory.

VMAKE also logs the entire software creation process.

Defining the MAKE Operation [Page 36]

VMAKE Log [Page 58]s

Translation Process [Page 58]

Link Operation [Page 59]

Defining the MAKE Operation

VMAKE Versions [Page 36]

Grammar for the VMAKE Call [Page 37]

VMAKE Tools [Page 40]

Creating Description Files [Page 43]

VMAKE Versions

There are three VMAKE versions:

- fast [Page 36]
- quick [Page 37]
- slow [Page 37]

The versions relate to the various trace and check levels in the software created.

You specify the required version when you call VMAKE:

If the VMAKE version in the description files [Page 24] for the module groups is different to the one you specified when you called VMAKE, the call version is overridden by the version specified in the validity range of the description file.

fast VMAKE Version

Syntax

```
imf.pl
```

Use

When you call VMAKE as a `fast` version (VMAKE versions [Page 36]), the software created in the result does not carry out any additional internal error checks, and does not log any information other than the standard trace output of the database.

quick VMAKE Version

Syntax

```
imq.pl
```

Use

When you call VMAKE as a `quick` version, the software created as a result does carries out additional internal error checks.

slow VMAKE Version

Syntax

```
ims.pl
```

Use

When you call VMAKE as a `slow` version, the software created as a result carries out additional internal error checks, and logs information in addition to the standard trace output of the database.

Grammar for the VMAKE Call

The VMAKE call is based on the general grammar for description files [Page 44], the naming conventions for module files [Page 32], and the naming conventions for description files [Page 33].

```
<usage> =
vmake<space>{<vmake_opt_list>}<target>[//<ext_option>]{<space><target
><ext_option>}

<target> = <source_spec> | <desc_spec>

<vmake_optlist> = [-<vmake_opt>{<vmake_opt>}] [+e]

<vmake_opt> = b | c | d | D<digit> | F | g | i | I | k | K | l | L |
m | M | n | p | r | S | u | U | w | x
```

General Rules

1. In `<ext_option>`, `<token_sep> = /`
2. `<desc_ext>` can be omitted in `<desc_spec>`

   ```
   vmake -D4i vzz34x//debug/profile

   vmake -D4 -l -p demo.mac

   vmake demo
   ```

   Explanation:

You can specify additional Options for Description Files [Page 47] in the VMAKE call for the translation of module files using VMAKE. Where applicable, these options are combined with the options that are specified in the description files used at the same time. The options specified for description files in the VMAKE call must be separated from `<target>` with **two** slashes.

The options for description files are separated from each other with **one** slash.

Options for Calling VMAKE

Option	Description
`D<digit>`	Information on internal data and program runs is output. The larger `<digit>` is, the more information is output.
`i`	interactive VMAKE is executed interactively (default setting for Windows NT). In UNIX systems, VMAKE creates a new process that assumes all of the tasks. The parent process shows all of the messages from the child process, and waits for termination signals. If you use the `i` option, a new process is not started, and the standard signal handlers are used.
`w`	This option corresponds to the default setting for UNIX systems, with the exception that the parent process is terminated by a termination signal without a query. Normally, this signal leads to a query asking you to confirm whether you want to terminate or continue.
`b`	background This option terminates the parent process once the child process has been successfully generated. Note: the child process is generated once all of the descriptions have been collected. Normally, the first child process message is `<date> start of <target>`. VMAKE is executed interactively before this message, that is, the standard signal handlers are used.
`c`	print collection only Once all of the description files have been collected, all of the description information is displayed, and VMAKE is terminated. This can be used to check the description files (or the VMAKE tool).
`m`	print module list only Once all of the description information has been collected, all of the required module files are output.
`x`	print commands before execution All of the file system commands executed by VMAKE are first displayed.
`n`	no execution VMAKE functions as normal, but no file system or OS commands are executed. VMAKE assumes that all of these operations have been successfully carried out.
`r`	unpack module frames only ('retrieve') Normally, module files are 'freed' from their frame and then translated. This option suppresses translation and linking. The unpacked module files remain in the working directory. No further operations are executed.
`L`	lint This option analyzes C module files using `Lint`. No further operations are executed.
`S`	sizeof Size information for types and variables is extracted from PASCAL module files. No further operations are executed.

d	do not make dependencies Only the targets listed in the command line are created (if they are not up to date). Dependent module files are assumed to be up to date. For test purposes, therefore, you can avoid having to update all of the dependent module files.
u	unconditionally make named targets All named module files are explicitly retranslated and linked. All dependent module files are translated and linked as standard, depending on whether they are up to date. All named description files are explicitly relinked.
U	unconditionally make ALL targets (except includes) All named module files, and files that are dependent on them, are translated and linked (exception: include files).
I	unconditionally make includes too (with -U) When this option is used with the -U option, include files are regenerated.
k	keep temporary files Temporary files created by VMAKE are not deleted. These are unpacked module files, object files, and others.
K	keep temporary files Temporary files created by VMAKE are not deleted. In addition, all scripts are assigned the -k option, with the result that they also do not delete their temporary files.
p	include profiling calls into all targets You use this option to specify profiling for all module files.
M	more information (print manual entry) You use this option to display the vmakman file.
l	local make This option prevents object files from smaller path hierarchies from being used. VMAKE only translates module files that originate from the lowest path hierarchy. This option overrides the $VMAKE_OPTION parameter g.
g	global make Use of the complete path hierarchy $VMAKE_PATH for finding object files is initiated. This option overrides the $VMAKE_OPTION parameter I.
R	require To speed up the creation process, the ? require : <source> option in the description files is not evaluated (default setting in VMAKE). If you use the R option, the ? require : <source> option is taken into account.
F	file list for release You use this option to create a distribution list; any obsolete targets, however, are not updated. All files from the called target that have been assigned the distrib or distribute option for description files are entered in a list. This list is saved in a file in the $INSTROOT/etc directory. The name of the file is the same as the name of the called target. The file is given the extension .lst. The all.lst file is created when you enter **imf -F all** .
+e	If you have specified software creation in debug mode using the $VMAKE_OPTION parameter, you can deactivate debug mode with the +e option.

Integration

Grammar for the VMAKE Call [Page 37]

VMAKE Tools

The tools used by VMAKE are essentially scripts [Page 40]. These can be written in any script or prototyping language. The shell used to execute the scripts is defined in the environment variable $TOOLSHELL [Page 16].

The VMAKE tools accept both absolute and relative path specifications for module files [Page 24] and description files [Page 24]. The targets [Page 31] are created in the current run directory specified by VMAKE and are then stored in the specified target directory.

Storage Location of the VMAKE Tools

VMAKE initially assumes that the VMAKE tools are stored in the `$TOOL/bin` directory. If this is not the case, VMAKE searches every local `<path>` along the `<path>/sys/tool/bin` path.

Scripts for the VMAKE Tools

Script Name	Description
`mfainc [-<token>=<value>] source`	Extracts the include code from an Assembler include file [Page 24] and writes it to the file `<file_name [Page 23]>.h`. Further options define conditional compilation [Page 59]
`mfcinc [-<token>=<value>] source`	Extracts the include code from a C/C++ include file and writes it to the file `<file_name>.h`. Further options define conditional compilation [Page 59]
`mfpinc [-<token>=<value>] source`	Extracts the include file from a C/C++ include file. Extracted CONST definitions are written to the file `<file_name>.con`; extracted TYPE definitions are written to the file `<file_name>.typ`. Further options define conditional compilation [Page 59]
`mfcexp [-<token>=<value>] source`	Writes the exported interface (*define section* of the text frame) to the file `<file_name>.h`. Further options define conditional compilation [Page 59]

`mfa [-trace=<procedure>] [-check=no] [-<token>=<value>] source [includes]`	Extracts the code section of an Assembler module file and writes it to the file `<file_name>.s`. If include files are required, they must be specified in the correct order. The `-trace=...` and `-check=...` options are inserted to ensure compatibility with `mfp` scripts, and are ignored by the `mfa` script. Further options define conditional compilation [Page 59]
`mfc [-trace=<procedure>] [-check=no] [-<token>=<value>] source [includes]`	Extracts the code section of a C/C++ module file [Page 24] and writes it to `<file_name>(.c |.cpp)`. If include files are required, they must be specified in the correct order. If $VMAKE_VERSION [Page 12] is defined as slow [Page 37], `#define DEBUG` is used as a prefix for the file name. The `-trace=...` and `-check=...` options are inserted to ensure compatibility with `mfp` scripts, and are ignored by the `mfc` script. Further options define conditional compilation [Page 59]
`mfp [-trace=<procedure>] [-check=no] [-<token>=<value>] source [includes]`	Extracts the defined section, use section, and code section of a PASCAL module file and writes them to the file `<file_name>.p`. If include files are required, they must be specified in the correct order. The `-trace=<procedure>` option specifies the name of the trace procedure, whose call is inserted at the start of every function and procedure. The `-check=no` option defines whether the name prefixes of exported procedures are to be checked. Further options define conditional compilation [Page 59]
`mfsize [include-list] module`	Creates a file `<file_name>.siz` that contains size information on the types of module file used. If include files are required, they must be specified in the correct order.
`mfextra [-<token>=<value>] source destination [includes]`	Creates files that do not require translation [Page 30]. Further options define conditional compilation [Page 59] and attributes of the files that do not require translation.
`compa [as-flags] source`	Translates an Assembler module file. The file `<file_name>.o` is created.
`compc [cc-flags] source`	Translates a C/C++ module file. The file `<file_name>.o` is created.
`compp [pc-flags] source`	Translates a PASCAL module file. The file `<file_name>.o` is created.
`comppc [pc-flags] source`	Translates a PASCAL module file into a C module file.

`complint [cc-flags] source`	Uses LINT to check a C module file.
`archive [ar-flags] library objects`	Writes object files [Page 26] to the *library*.
`linkrel [ld-flags] relocatable objects`	Writes object files to a relocatable object [Page 34]. The name of the created relocatable object is `relocatable`.
`linkshr [ld-flags] relocatable objects`	Writes object files to a relocatable object, the global variables of which must be stored in the shared memory. The name of the created relocatable object is `relocatable`.
`linkdll [ld-flags] dll archive objects`	Links object files to a dynamic library `dll` and generates a static library `archive`.
`linklnk [ld-flags] [-o program] objects`	Links a program with the name `program`.
`linkshm [ld-flags] [-o program] objects`	Links a program with the name `program`, the global variables of which must be stored in the shared memory.

VMAKE completes the names of the scripts using the $TOOLEXT [Page 17] variable (if available), which defines the file extension of the scripts. If additional parameters are to be sent to $TOOLSHELL [Page 16], they must be defined in the environment variable $TOOLOPT [Page 17].

Selecting the Translation Tools

Module files [Page 24] can be written in different programming languages.

Special VMAKE tools [Page 40] are provided for translating PASCAL, C/C++, and Assembler, as well as a tool for translating a group of various description languages, such as IDL, Resource, Yak, Lex, and so on.

Names of the Available Tools

		Assembler	PASCAL	C / C++	Description Language Group
Phase 1	Unpack imported interface	mfainc	mfpinc	mfcinc	mfcinc
	Unpack exported interface	mfaexp	mfpexp	mfcexp	mfcexp
	Unpack module	mfa	mfp	mfc	mfrc
Phase 2	Translate	compa	comppc	compc	comprc

Each tool consists of a set of scripts that can be modified in the `Langinfo` initialization file [Page 26].

Assignment Between File Extension or Last Character and Translation Tool

VMAKE assigns module files [Page 24] to a specific tool on the basis of the file extension. If the file does not have an extension, it is assigned on the basis of the last character in the file name.

	Assembler	PASCAL	C / C++	Description Language Class
File extension	`s`	`p, t`	`c, h / cpp, hpp`	`rc, ico, def, mc, dlg, idl, ycc, lex, rgs`
Last character	`a`	`p, t` or a number	`c / x`	`r`

The assignment of the translation tools on the basis of the file extension or last character in the file name can be modified in the `Langextinfo` initialization file [Page 26].

Options for VMAKE Tools

The following options are transferred by VMAKE

`-language=<lang id>`	Abbreviation for the programming language of the source code.

The following options can be transferred by VMAKE

`-k`	Temporary files are not deleted
`-p`	Profiling is activated (for at least one module file)
`-g`	Debug mode [Page 31] is activated (for at least one module file)
`-Y`	Shared global data, PTOC pointer conversion
`-unpack=no`	The source code does not contain a frame (mf* scripts only)

Depending on the tool in question, these options are either used or ignored.

The specified options for calling VMAKE [Page 38] and options for description files [Page 47] also apply here.

Creating Description Files

General Grammar for Description Files [Page 44]

Options for Description Files [Page 47]

General Grammar for Description Files

<debug_opt> = debug | d

<digit> = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<letter> = A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | r | u | v | w | x | y | z

<new_line> = explicit line break

<profile_opt> = profile | p

<sign> = + | -

<tabulator> = tab

<vmake_default> = f | q | s

<vmake_version> = f | q | s

<extended_letter> = all other printable characters

<character> = <digit> | <letter> | <extended_ letter>

<identifier> = <token>

<space> = [<space>] | [<space>]<tabulator>

<token_sep> = {<token_sep>}<space>

<token> = <character>{<character>}

<list> = [<list>,],{<character>}<token_sep>

<ext_list> = [<ext_list>,],[<sign>]{<character>}<token_sep>

<comment> = [<comment><token_sep>]<token>

<compile_option> = <extended_letter><token>

<demand_spec> = demand[{<layer>{<layer>}}]

<dep_list> = (obj | inc | dep | <demand spec>)=(<list> | <ext_list> | EMPTY)

<expr> = <character>{<character>} | (<expr>) | not<space><expr> | ![<space>]<expr> | <expr><space>[(not | !)<space>]in<space>[[<space>]<expr>{[<space>] ,[<space>]<expr>}[<space>]] | <expr>=[=]<expr> | <expr>!=<expr> | <expr>&&<expr> | <expr>||<expr>

<layer> = <letter><letter>{<letter>}

<mf_option> = <token>[=<value>]

<option> = [<option><token_sep>](<dep_list> | <mf_option> | <compile_option> | <target_option>)

<option_block> = ([f | [q | [s) <option> {<option>} (f] | q] | s])

<ext_option> = <option_block> | <option> | EMPTY

<target_option> = uncond | remake | <debug opt> | binary | ascii | exec | definition | interface | <profile opt> | shrglob | noshrglob | noobjcopy | (<vmake_version>[<vmake_default>]) | -><token> | <ext_target_option> | <ext_target_option_list>

```
<ext_target_option> = ?<token_sep>(default_layer | default_version |
require | -><token> | propagate)<token_sep>(: |
([<character>]))<token_sep><token>

<ext_target_option_list> = ?<token_sep>(output | link_with |
link_option | tool_option)<token_sep>(: |
([<character>]))<token_sep><token>{<token_sep>,<token_sep><token>}

<ext_target_option_line> = (<ext_target_option> |
<ext_target_option_list>) <new_line>

<value> = <character>{<character>}

<def line> = &define | &undef <identifier> [<value>]<new_line>

<comment_line> = #<comment><new_line>

<block> = <instance_block> | <sub_block>

<instance_block> =
(<instance_block_start>{<sub_block>}<instance_block_end>) | EMPTY

<instance_block_start> = (&fast | &quick | &slow) <comment><new_line>

<instance_block_end> = (&endfast | &endquick | &endslow)
<comment><new_line>

<sub_block> = {<line>} | <sub_block_start>{<line>}<sub_block_end>

<sub_block_start> = &if <expr>{<space>}<new_line> | &ifdef | &ifndef
| &ifvar | &ifnvar) <identifier>{<space>}<new_line>

<sub_block_end> = &endif <comment><new_line> | &else
<comment><new_line> | &elif <comment><new_line>
```

`<line>` Line

For more details, see Grammar for Link Descriptions [Page 45], Grammar for Compilation Descriptions [Page 46], or Grammar for Processing Lists [Page 46].

General Rules

1. `<token>` can contain a maximum of 512 characters
2. `<token>` must not contain a `<space>`
3. If `<instance_block_start>` contains the string `&xxx`, `<instance_block_stop>` must contain the character string `&endxxx`.
4. If the `<option_block>` expression starts with `[x` , it must end with `x]`.
5. In `<mf_option>`, `<token>` must start with `<letter>` or `<character>`.

Grammar for Link Descriptions

Link descriptions are based on the general grammar for description files [Page 44], the naming conventions for module files [Page 32], and the naming conventions for description files [Page 33].

```
<link_description> = {<block>} | EMPTY

<line> = <link_desc_line>

<link_desc_line> = <comment_line> | <link_line> | <def_line> |
<ext_target_option_line> | <tool_option_line> | <file> | EMPTY
```

```
<tool_option_line> = <extended_letter><token><new_line>

<file> = (/ | $)<token>

<link_line> = [<source_spec>|<desc_spec>]<new_line>
```

General Rule

In <tool_option_line>, <ext_character> must not begin with / | $ | !

Grammar for Compilation Descriptions

Compilation descriptions are based on the general grammar for description files [Page 44] and the naming conventions for module files [Page 32].

```
<compile_description> = {<block>} | EMPTY

<line> = <comp_desc_line>

<comp_desc_line> = <comment_line> | <comp_line> | <def_line> |
<option_line> | EMPTY

<comp_line> = (<source_spec> |
<reg_source>)<token_sep>{<option>}<new_line>

<reg_source> = regex(<regex>) |
([<source_id>]*[<lang_id>][<variant>])

<source_id> = g | h | i | v

<regex> = regular expression (GNU)
```

Grammar for Processing Lists

Processing lists are based on the general grammar for description files [Page 44], the naming conventions for module files [Page 32], and the naming conventions for description files [Page 33].

```
<macro_description> = {<block>} | EMPTY

<line> = <macro_desc_line>

<macro_desc_line> = <comment_line> | <macro_line> | <def_line> |
<ext_target_option_line>

| <command_line> | EMPTY

<command_line> = ![! | ?]<command>

<command> = {<character>}

<macro_line> = [<source_spec>|<desc_spec>]<new_line>
```

General Rule

In <tool_option_line>, <ext_character> must not begin with / | $ | !

Options for Description Files

By specifying options in description files, you can influence the translation process [Page 58] and link operation [Page 59], as well as modify the system default for certain processes.

Some of these options differ depending on the type of description file.

Options for Link Descriptions [Page 47]

Options for Compilation Descriptions [Page 48]

Options for Processing Lists [Page 48]

Options for Link Descriptions

The following options are available for link descriptions [Page 25]:

? defaultlayer : <layer> [Page 50]

? defaultlayer : [Page 50]

? defaultversion : <vmake_version> [Page 50]

? defaultversion : [Page 50]

? distribute : <list> [Page 51]

? link with : <list> [Page 51]

? linkoption : <list> [Page 51]

? output : <list> [Page 51]

? propagate : <variable>[=<value>] [Page 52]

? require : <target> [Page 52]

? tooloption : <list> [Page 52]

-><output> [Page 53]

ascii [Page 53]

binary [Page 53]

debug|d [Page 54]

definition [Page 54]

demand [Page 54]

demand{<relative_path>} [Page 54]

demand=<list> [Page 54]

demand{<relative_path>}=<list> [Page 55]

dep=<list> [Page 55]

distrib [Page 55]

extdep=<list> [Page 55]

inc=<list> [Page 56]

interface [Page 56]

nobind [Page 56]

nodistrib [Page 56]

noobjcopy [Page 56]
noshrglob [Page 57]
obj=<list> [Page 57]
profile|p [Page 57]
remake [Page 57]
shrglob [Page 58]
uncond [Page 58]

Options for Compilation Descriptions

The following options are available for compilation descriptions [Page 25]:

? defaultlayer : <layer> [Page 50]
? defaultlayer : [Page 50]
ascii [Page 53]
binary [Page 53]
debug|d [Page 54]
definition [Page 54]
demand=<list> [Page 54]
demand{<relative_path>}=<list> [Page 55]
dep=<list> [Page 55]
exec [Page 55]
extdep=<list> [Page 55]
inc=<list> [Page 56]
interface [Page 56]
noobjcopy [Page 56]
noshrglob [Page 57]
noversion [Page 57]
obj=<list> [Page 57]
profile|p [Page 57]
remake
shrglob [Page 58]
uncond [Page 58]

Options for Processing Lists

Options

The following options are available for processing lists [Page 26]:

!! <command> [Page 49]
!? <command> [Page 49]
! <command> [Page 49]
? defaultlayer : <layer> [Page 50]
? defaultlayer : [Page 50]
? defaultversion : <vmake_version> [Page 50]
? defaultversion : [Page 50]
? distribute : <list> [Page 51]
? propagate : <variable>[=<value>] [Page 52]
? require : <target> [Page 52]
ascii [Page 53]
binary [Page 53]
debug|d [Page 54]
distrib [Page 55]
nodistrib [Page 56]
remake [Page 57]
shrglob [Page 58]
uncond [Page 58]

! <command>

You can only use this option in processing lists [Page 26].

VMAKE processes a processing list from top to bottom. When the tool reaches this option, and if VMAKE has updated targets [Page 31] up to this point, the OS command **<command>** is executed.

If no targets have been updated, this command is ignored.

!! <command>

You can only use this option in processing lists [Page 26].

VMAKE processes a processing list from top to bottom. When the tool reaches this option, the operating system command **<command>** is executed.

Unlike the ! <command> [Page 49] option, the command is executed irrespective of whether targets have been updated up to this point.

!? <command>

You can only use this option in processing lists [Page 26].

VMAKE processes a processing list from top to bottom. When the tool reaches this option, the operating system command **<command>** is only executed if all of the preceding targets [Page 31] have been successfully updated.

? defaultlayer : <layer>

You can use this option in processing lists [Page 26], link descriptions [Page 25], and compilation descriptions [Page 25].

If the name of a module file [Page 24] does not follow the naming conventions [Page 32], and if the system cannot determine the corresponding layer [Page 31] as a result, the layer specified in **<layer>** is accessed.

This option can be repeated several times in a description file.

? defaultlayer :

You can use this option in processing lists [Page 26], link descriptions [Page 25], and compilation descriptions [Page 25].

If the name of a module file [Page 24] does not follow the naming conventions [Page 32], and if the system cannot determine the corresponding layer [Page 31] as a result, the layer specified with the **? defaultlayer : <layer>** option is accessed.

Use the **? defaultlayer :** option if you want to delete the default value for a layer.

? defaultversion : <vmake_version>

You can use this option in processing lists [Page 26] and link descriptions [Page 25].

This option defines the VMAKE version [Page 36] to be used to process the following listed files.

To specify a different version subsequently, use the `? defaultversion : <vmake_version>` option. This option, therefore, can be used repeatedly in a description file.

To reset VMAKE to the VMAKE version specified in the call, use the `? defaultversion :` option.

? defaultversion :

You can use this option in processing lists [Page 26] and link descriptions [Page 25].

You use this option to delete a VMAKE version [Page 36] that was set with ? defaultversion : <vmake_version> [Page 50].

The system then uses the VMAKE version specified when VMAKE is called.

? distribute : <list>

You can use this option in processing lists [Page 26] and link descriptions [Page 25].

This option can be used to create distribution lists (-F option for calling VMAKE [Page 38]). Enter the list of files to be included in the distribution list in **<list>**.

Specify the file paths in **<list>**, relative to the $INSTROOT [Page 15] directory (for example `pgm/kernel`) .

We recommend that you use the distrib [Page 55] option, if possible, for this purpose.

? link with : <list>

You can only use this option in link descriptions [Page 25].

If a target [Page 31] requires additional libraries, these can be included in the link description with the `? link with : <list>` option.

? linkoption : <list>

You can only use this option in link descriptions [Page 25].

You use this option to transfer the list of options specified in **<list>** to the tool used for the link operation [Page 59].

? output : <list>

You can only use this option in link descriptions [Page 25].

Targets [Page 31] of link descriptions are given the name of the link description and are stored in a path specified by VMAKE.

Changing Names and Paths

If you use the `? output : <list>` option, you can change the name and/or path of **several** output files, or create copies of individual files. To do so, you create a list **<list>**, in which you assign one or more paths to one or more targets. VMAKE then changes the name and path definition specified for this target by the system.

If you specify a directory in the list or end an entry with a slash, the file retains the name assigned by the system and is copied to the specified directory.

If you want to change the name and/or path for **one individual** output file, use the -><output> [Page 53] option.

Output Files for Different Platforms

An output file can have different file extensions on different platforms. In this case, specify the file extension `*` for the target. By doing so, you can avoid having to define individual output files for each platform.

Dynamic libraries have the file extension `dll` on Windows NT and `so` or `sl` (HP-UX) on UNIX. In this case, the option should read:

```
? output : $DBROOT/misc/libtest.*
```

? propagate : <variable>[=<value>]

You use this option in processing lists [Page 26] and link descriptions [Page 25] to set an environment variable.

The system deletes this variable once the relevant list or link description has been processed.

? require : <target>

You can use this option in processing lists [Page 26] and link descriptions [Page 25]. It is only effective with the -R option for calling VMAKE [Page 38].

You use this option to specify whether a link description requires certain targets [Page 31] to be created.

A tool for forwarding a file to the pre-compiler is to be called in a processing list. In this case, you can use this option to ensure that the pre-compiler is generated first.

? tooloption : <list>

You can only use this option in link descriptions [Page 25].

The options in the **<list>** are forwarded to the tool that is responsible for the link operation [Page 59].

-><output>

You can use this option in link descriptions [Page 25] and compilation descriptions [Page 25].

Targets [Page 31] of link descriptions are given the name of the link description and are stored in a path specified by VMAKE.

Use

Changing Names and Paths

If you want to change the name and/or path for **one individual** output file, use the **`-><output>`** option.

If you use the ? output: <list> [Page 51] option, you can change the name and/or path of **several** output files, or create copies of individual files. To do so, you create a list **`<list>`**, in which you assign one or more paths to one or more targets. VMAKE then changes the name and path definition specified for this target by the system.

If you specify a directory in the list or end an entry with a slash, the file retains the name assigned by the system and is copied to the specified directory.

Output Files for Different Platforms

An output file can have different file extensions on different platforms. In this case, specify the file extension `*` for the target. By doing so, you can avoid having to define individual output files for each platform.

Dynamic libraries have the file extension `dll` on Windows NT and `so` or `sl` (HP-UX) on UNIX. In this case, the option should read:
`->$DBROOT/misc/libtest.*`

ascii

You can use this option in processing lists [Page 26], link descriptions [Page 25], and compilation descriptions [Page 25].

The option prevents module files [Page 24] from being translated. VMAKE creates the corresponding target [Page 31] by copying the module file only to the object directory [Page 26]. In contrast to the binary [Page 53] option, the file is treated as a text file.

If you want to store the file in a different location, use the -><output> [Page 53] option.

binary

You can use this option in processing lists [Page 26], link descriptions [Page 25], and compilation descriptions [Page 25].

The option prevents module files [Page 24] from being unpacked and translated. VMAKE creates the corresponding target [Page 31] by copying the module file only to the object directory [Page 26]. In contrast to the ascii [Page 53] option, the file is treated as a binary file.

If you want to store the file in a different location, use the ->output [Page 53] option.

debug|d

You can use this option in processing lists [Page 26], link descriptions [Page 25], and compilation descriptions [Page 25].

Module files [Page 24] are translated in debug mode [Page 31].

definition

You can use this option in link descriptions [Page 25] and compilation descriptions [Page 25].

The definition option identifies a module file [Page 24] as a definition file for dynamic libraries on Windows NT. Semantically speaking, this option has the same effect as a combination of the binary [Page 53] and demand [Page 54] options.

When this option is assigned to a module file, the file is copied as a binary file to the current working directory tmp [Page 9] before the link operation [Page 59] takes place.

demand

You can only use this option in link descriptions [Page 25].

An object file [Page 26] is copied to the tmp [Page 9] directory before the link operation [Page 59] takes place.

If you assign this option to a module file [Page 24], the resulting object file is requested for the link operation; in other words, it is copied to the working directory specified by VMAKE. Expression of physical dependency.

demand{<relative path>}

You can only use this option in link descriptions [Page 25].

The object file [Page 26] of the identified module file [Page 24] is requested in a specific path for the link operation [Page 59]; in other words, it is copied to the specified path relative to the working directory. Expression of physical dependency.

demand=<list>

You can use this option in link descriptions [Page 25] and compilation descriptions [Page 25].

The object files (see also "binary") of the listed files are requested when translation is carried out; in other words, they are copied to the relevant working directory. Expression of physical dependency.

demand{<relative path>}=<list>

You can use this option in link descriptions [Page 25] and compilation descriptions [Page 25].

The object files (see also "binary") of the listed files are requested to a specific path when translation is carried out; in other words, they are copied to the path relative to the working directory. Expression of physical dependency.

dep=<list>

You can use this option in link descriptions [Page 25] and compilation descriptions [Page 25].

Lists the module files [Page 24] on which the specified module file is dependent, that is,

```
vco01.idl    dep=vco03.idl
```

distrib

You can use this option in processing lists [Page 26] and link descriptions [Page 25].

This option is required to create distribution lists (-F option for Calling VMAKE [Page 38]). The files resulting from the target [Page 31] listed directly above this option are included in the distribution list, provided that they are located in the $INSTROOT [Page 15] directory.

exec

You can only use this option in compilation descriptions [Page 25]. It is only evaluated on UNIX and in conjunction with the ascii [Page 53] or binary [Page 53] options.

When you specify this option, the corresponding target [Page 31] is assigned the *execute* access authorization.

extdep=<list>

You can use this option in link descriptions [Page 25] and compilation descriptions [Page 25].

You use this option to define a logical dependency between targets and/or files.

Translating module file X results in the by-product Y. You can use this option to specify that X should be retranslated as soon as Y no longer exists.

inc=<list>

You can use this option in link descriptions [Page 25] and compilation descriptions [Page 25]. You list include files that are used by the module file.

The files in `<list>` must be identifiable as include files.

Files not identifiable as an include file by the extension `.h` or `00`, or by the letter `h` as the first letter in the file name, can be identified as such by appending an `(i)`.

```
inc=vma18(i)
```

interface

You can use this option in link descriptions [Page 25] and compilation descriptions [Page 25].

By specifying this option, you can ensure that a module file cannot be updated once it has been translated (for example, a defined interface). For this purpose, a file with the extension `.i` is created on the computer, on which module files with this option are translated.

This module file can only be translated again when VMAKE no longer finds a file of this name with the extension `.i` in the path hierarchy (VMAKE_PATH [Page 16]).

nobind

You can only use this option in link descriptions [Page 25].

The target [Page 31] that has been assigned this option is created but not linked during the link operation [Page 59].

nodistrib

You can use this option in processing lists [Page 26] and link descriptions [Page 25].

This option is specified after the Target [Page 31] **`nodistrib`** or directly after the output file in brackets (**`nodistrib)`**. In this way, you can exclude individual components or files, that would usually be created during the make run, from the delivery list.

noobjcopy

You can use this option in link descriptions [Page 25] and compilation descriptions [Page 25].

Object files that result from module files with this option are not copied. You use this option, for example, if the object contains information that is computer specific or unique across all departments.

noshrglob

You can use this option in link descriptions [Page 25] and compilation descriptions [Page 25].

By specifying this option, you explicitly suppress the calculation of size specifications for global variables in the source code. These size specifications are required to use the shared memory.

noversion

You can use this option in compilation descriptions [Page 25].

If you want to create modules used by the header files in such a way that they are version-independent (VMAKE Versions [Page 36]), you must use `noversion` for these modules. The object files [Page 26] and the date files [Page 30] are then copied directly under the $WRK [Page 18] directory in the `obj` and `date` directories. In this way, these files are only created once for all versions (`fast`, `quick`, `slow`).

obj=<list>

You can use this option in link descriptions [Page 25] and compilation descriptions [Page 25].

Normally, the translation process [Page 58] with VMAKE generates exactly one object file [Page 26]. The name of this file comprises the name of the module file [Page 24] and the file extension `.o`.
You use this option if object files other than those that are normally created are generated. `<list>` can be empty if no object file is created.

profile|p

You can use this option in link descriptions [Page 25] and compilation descriptions [Page 25].

You can use this option to insert source code for measuring performance (profiling) in a module file.

remake

You can use this option in processing lists [Page 26], link descriptions [Page 25], and compilation descriptions [Page 25].

When object files are no longer up to date and the target is to be recreated, you can use this option to force a retranslation of the corresponding module files.

shrglob

You can use this option in processing lists [Page 26], link descriptions [Page 25], and compilation descriptions [Page 25].

With this option, you specify that a file is to be created containing the size specifications for the global variables used in the source code. These size specifications are required to use the shared memory.

uncond

You can use this option in processing lists [Page 26], link descriptions [Page 25], and compilation descriptions [Page 25].

You use this option if a module file is to be retranslated with every make operation.

VMAKE Logs

The entire MAKE operation [Page 34] is logged by VMAKE. In doing so, VMAKE generates two output logs and stores them in the `prot` directory (General Directory Structure [Page 9]).

These logs are named after the file name specified as a parameter when the required VMAKE version [Page 36] was called. The files have the extension `.p0` or `.x0`.

File extension	Log
`.p0`	Normal log; corresponds to the screen outputs during the creation process
`.x0`	Contains detailed information on the individual files as well as the options [Page 43] used to call the VMAKE tools [Page 40]

When these logs are created, older logs with the same name are renamed (the last digit is incremented by 1). A log history comprising up to 10 logs (`.p0` - `.p9`) is, therefore, possible.

You can display the current logs in the editor ($EDITOR [Page 20]) using the development environment tool ipf.pl [Page 22].

Translation Process

The translation process is divided into two phases:

Unpacking the Module Files [Page 58]

Translating the Module Files [Page 59]

Unpacking the Module Files

VMAKE uses the options in the compilation descriptions [Page 25] to determine where to unpack the module files [Page 24]. The following additional steps can be carried out:

The parts of the frame that are not relevant for translation are removed [Page 59]

Conditional compilation [Page 59]

Removing Frame Parts that are not Relevant for Translation

Module files [Page 24] that have to be unpacked [Page 58] have a text frame that contains the following information:

- Name of the module
- Module specification
- Author, version, release
- Exported interface
- Imported interface
- Source code
- Control characters

Since some of the text frame does not have to be translated, those parts of the frame that are not relevant for translation are removed when the file is unpacked.

Conditional Compilation

VMAKE hides those parts of the source code that are not required under the current conditions, such as unnecessary network connections (UNIX or NT).

The system copies the include files [Page 24] that are to be used to the start of the module file [Page 24] that is awaiting translation.

Translating the Module Files

VMAKE translates the module files [Page 24] in the temporary tmp [Page 9] directory.

The system stores the resulting object files [Page 26] in an object directory [Page 26] and deletes the files from the temporary directory.

Link Operation

Object files [Page 26] are linked to form a software component in the tmp [Page 9] directory. VMAKE then stores this component in a predefined (see below) directory, or in a target directory specified in the link description. The component is then deleted from the tmp [Page 9] directory.

For the link operation, VMAKE is equipped with various tools (VMAKE tools [Page 40]) for each type of object file that can result from the translation process [Page 58]. VMAKE uses

the file extension of the description file to decide which tool to use to link the respective object files.

The tool takes the object files and the relevant options [Page 47] to be used from the link description [Page 25].

Software Component to be Created	Link Script	Target Directory
Program	`linklnk`	$DBROOT/pgm
Program that facilitates shared memory access	`linkshm`	$DBROOT/pgm
Static library	`archive`	$WRK/fast/obj or $WRK/quick/obj or $WRK/slow/obj
Dynamic library	`linkdll`	$DBROOT/lib(32 bit) $DBROOT/lib/lib64 (64 bit)
Program that can be linked to other programs	`linkrel`	$WRK/fast/obj or $WRK/quick/obj or $WRK/slow/obj
Program that can be linked to other programs and facilitates shared memory access	`linkshr`	$WRK/fast/obj or $WRK/quick/obj or $WRK/slow/obj

Working with the SAP DB Development Environment: Examples

Operating the Development Environment [Page 60]

Creating DBMCLI with the SAP DB Development Environment [Page 67]

Operating the Development Environment

The operation of the development environment can be explained using an example (objective [Page 60]).

Process Flow [Page 61]

Explanation of the Process Flow [Page 61]

Logs [Page 62]

Further Options [Page 64]

Further Information on Dependencies [Page 65]

Comparison Between the SAP DB VMAKE Program and a Conventional Make Program [Page 67]

Objective

- The file `sys/src/SAPDB/HelloWorld.c` is to be translated.

- The include file [Page 24] `sys/src/SAPDB/HelloWorld.h` is required for this purpose.
- The result is linked to an executable program: `usr/pgm/hello` (or `usr\pgm\hello.exe` on Windows NT).

Process Flow

`imf.pl hello.lnk`

```
Build: 40283
vmake 9.8.5 28-11-2000
VMAKE_PATH=d:\V73\develop,d:\SAPSRC
17-01-01 13:33:46 start  of    collection phase
17-01-01 13:33:54 end    of    collection phase
17-01-01 13:33:54 start  of    hello.lnk   fast
HelloWorld.h
HelloWorld.c fast
hello.lnk fast
s100buildnumber_PID854.c
17-01-01 13:33:55 end    of    hello.lnk   fast
```

Explanation of the Process Flow

First, all of the required files are determined:

The description file [Page 24] `sys/desc/hello.lnk` is recognized as the description of an executable program from the file extension `.lnk`. This description contains only one file:

```
:SAPDB/HelloWorld.c
```

The colon stands for the directory with the module files [Page 24] `sys/src`.

The system recognizes from the file extension `.c` that the module file [Page 24] `sys/src/SAPDB/HelloWorld.c` is a module file that has to be translated with the C compiler. The include files [Page 24] required by this module file are listed in the description file `sys/desc/SAPDB.com`:

```
HelloWorld.c    inc=:SAPDB/HelloWorld.h
```

This line can contain further compiler options.

The include file `sys/src/SAPDB/HelloWorld.h` does not require any further files. Otherwise, these dependencies would also have to be entered in `SAPDB.com`.

Once the required files have been determined, each of these files is updated:

- Include files

HelloWorld.h

HelloWorld.c fast

hello.lnk fast

These files are copied from the development area (General Directory Structure [Page 9]) to a suitable subdirectory sys/wrk/incl, in this case, to sys/wrk/inl/SAPDB/HelloWorld.h.

File names in #include statements, therefore, must always be specified relative to sys/wrk/incl.

- C sources

HelloWorld.h

HelloWorld.c fast

hello.lnk fast

These are converted to object files [Page 26] using the C compiler.

Object files can be created in three different versions (VMAKE versions [Page 36]):

* fast: with optimization

* quick: with assertions

* slow: with assertions and trace outputs

In this case, the file created is written to sys/wrk/fast/obj/SAPDB/HelloWorld.o.

Object files are also assigned the file extension .o on Windows NT(tm).

The source file is copied to the sys/wrk/fast/tmp directory for translation [Page 58]. All of the intermediate results (this is particularly relevant for PASCAL sources) are created in this directory, and can be viewed in the event of an error.

- Executable programs

HelloWorld.h

HelloWorld.c fast

hello.lnk fast

These are linked and copied to usr/pgm.

On UNIX systems, usr/pgm/hello.f is created and a link to it is entered in the file system, under usr/pgm/hello.

On Windows NT, the file usr\pgm\hello.exe.f is created and copied to usr\pgm\hello.exe.

Logs

Using the **ipf.pl hello.lnk** command, you can display two VMAKE logs [Page 58].

hello.lnk.p0 is a copy of the information displayed on the screen.

`hello.lnk.x0` lists all of the relevant timestamps, as well as the actions that were carried out.

```
...
HelloWorld.c:
17-01-01 18:27:05 dependencies
(d:\V73\develop/sys/wrk/incl/SAPDB/HelloWorld.h)
17-01-01 12:29:20 d:\V73\develop/sys/src/SAPDB/HelloWorld.c
17-01-01 12:29:20
d:\V73\develop/sys/wrk/fast/dates/SAPDB/HelloWorld.c.dat
17-01-01 18:27:07 d:\V73\develop/sys/wrk/fast/obj/SAPDB/HelloWorld.o
unconditional     HelloWorld.c
rm d:\V73\develop/sys/wrk/fast/obj/SAPDB/HelloWorld.o
HelloWorld.c fast  debug
D:\SAPDevelop\Devtool\Perl\bin\perl.exe
D:\SAPDevelop\Devtool/bin/mfc.pl \
   -f \
   -debug=1 \
   -language=c \
   -unpack=no \
   d:\V73\develop/sys/src/SAPDB/HelloWorld.c \
   d:\V73\develop/sys/wrk/incl/SAPDB/HelloWorld.h
touch HelloWorld.c
D:\SAPDevelop\Devtool\Perl\bin\perl.exe
D:\SAPDevelop\Devtool/bin/compc.pl \
   -g \
   -language=c \
   HelloWorld.c
HelloWorld.c
touch HelloWorld.o
mv HelloWorld.o d:\V73\develop/sys/wrk/fast/obj/SAPDB/HelloWorld.o
touch d:\V73\develop/sys/wrk/fast/dates/SAPDB/HelloWorld.c.dat (from
HelloWorld.c)
17-01-01 06:30:53 PM
d:\V73\develop/sys/wrk/fast/obj/SAPDB/HelloWorld.o
...
```

If you want to find out which options were used to call the compilers or linkers, set the environment variable NOQUIET [Page 20].

UNIX: **`NOQUIET=1; export NOQUIET`**

Windows NT: **`set NOQUIET=1`**

```
imf -u :SAPDB/HelloWorld.c
HelloWorld.c fast
```

```
cl -DREL30 -DWIN32 -DI386 -DSAG -Id:\V73\develop\sys\wrk/incl/SAPDB -
DDU

MP_ENABLED -D_WIN32 -G6 -D_X86_=1 -DCRTAPI1=_cdecl -DCRTAPI2=_cdecl -
nol

ogo -Gy -W2 -MT -DFAST=FAST -Ox -ID:/SAPDevelop/Devtool/incl -
FoHelloWor

ld.o -c HelloWorld.c
```

Further Options

Forced Translation [Page 64]

Working with Debug Information [Page 65]

Forced Translation

- `-u` option

Creates the specified target [Page 31] as a new target. A new link is created to an executable program.

`imf.pl -u hello.lnk`

```
hello.lnk fast
```

A module file is retranslated.

`imf.pl -u :SAPDB/HelloWorld.c`

```
HelloWorld.c fast
```

- `-U` option

Recreates the specified target and all of the necessary module files [Page 24].

`imf.pl -U hello.lnk`

```
HelloWorld.c fast
hello.lnk fast
```

- `-U -I` option

Recreates the specified target, all of the necessary modules, and all of the required include files [Page 24].

`imf.pl -U -I hello.lnk`

```
HelloWorld.h
HelloWorld.c fast
hello.lnk fast
```

Working with Debug Information

The `-e` option instructs VMAKE to generate all targets [Page 31] with debug information (Debug Mode [Page 31]).

```
imf.pl -U -e hello.lnk
HelloWorld.c fast  debug
hello.lnk fast  debug
```

`usr/pgm/hello` can then be started in the debugger. The source files are copied to `sys/wrk/fast/debug` after the translation process [Page 58]. This is particularly important for PASCAL module files [Page 24], since these can only be debugged using the generated C file, and not in the source code itself.

Further Information on Dependencies

Displaying the New Targets to be Created [Page 65]

Displaying the Module and Description Files Used [Page 66]

Displaying the Module and Description Files Used, and their Dependencies [Page 66]

Displaying the New Targets to be Generated

```
imf.pl -u -n hello.lnk
vmake 9.8.5 28-11-2000
VMAKE_PATH=d:\V73\develop,d:\SAPSRC
17-01-01 04:50:40 PM start  of    collection phase
17-01-01 04:50:48 PM end    of    collection phase
17-01-01 04:50:48 PM start  of    hello.lnk   fast
HelloWorld.c fast
hello.lnk fast
17-01-01 04:50:48 PM end    of    hello.lnk   fast

statistics:

macros:      0
programs:    1
dynlinklibs: 0
relocs:      0
libraries:   0
modules:     1
includes:    1
```

```
files:       0
commands:    0
extras:      0
```

Displaying the Module and Description Files Used

imf.pl -m hello.lnk

```
desc/Langextinfo
desc/SAPDB.com
desc/hello.lnk
src/SAPDB/HelloWorld.h
src/SAPDB/HelloWorld.c
```

Displaying the Module and Description Files Used, and Their Dependencies

imf.pl -C hello.lnk

```
program
        name            'hello.lnk'
        type            TT_PGM
        version         'f'
        language        ' '
        layer           ''
        level_i         2
        level_s         2
        level_p         0
        level_c         0
        level_o         2
        shrglob         0
        debug           0
        profile         0
        uncond          0
        forcemake       0
        binary          0
        ascii           0
        definition      0
        noobject        0
```

```
        noobjcopy       0
        deptype         0
        nodebug         0
        file
        descriptions
        options
        dependencies
                 'HelloWorld.c' 'f'
        demands
        external dependencies
        callers

...
```

Comparison Between SAP DB VMAKE and a Conventional Make Program

To those who are familiar with the traditional UNIX make or Microsoft nmake programs, some of the features of VMAKE might seem a little surprising.

- Since all of the include files are copied to `sys/wrk/incl` and all of the source files are copied to a temporary directory before translation, file names in the error messages output by the compiler never point to the actual source file.
- The program always attempts to generate as much as possible, and does not stop after the first error. This is equivalent to the `-k` option in make.
- Each object file [Page 26] has a date file [Page 30] (for example, `sys/wrk/fast/dates/SAPDB/HelloWorld.c.dat`). An new object file is created:
  - If the existing object file is older than the source file (as in make)
  - If the timestamp of the date file differs from that of the source file.

Creating DBMCLI with the SAP DB Development Environment

Database Manager CLI

The Database Manager CLI (Database Manager Command Line Interface, DBMCLI) communicates with the DBM server and can, therefore, be used to administrate an SAP DB instance. The behavior of the Database Manager CLI depends on the call parameters and user entries. For more information about the Database Manager CLI, see the *Database Manager CLI: SAP DB 7.4* documentation.

Database Manager CLI tasks:

- The Database Manager CLI establishes the connection to the DBM server, and transfers one or more commands to it.
- The Database Manager CLI accepts and outputs the DBM server response(s).
- When the Database Manager CLI has been terminated, the connection to the DBM server is closed.

See also:

Structure of the Database Manager CLI [Page 68]

Creating the Database Manager CLI

1. Follow the instructions in Creating the Database Manager CLI [Page 68].
2. Perform a function check [Page 74].

Structure of the Database Manager CLI

The Database Manager CLI consists of the following components:

- Main Components
- Communication Components
- DBM Server

Main Components

Tasks of the main components:

- Accept and evaluate the call parameters and user entries.
- Transfer the dbm commands to the communication components.
- Receive the DBM server responses from the communication components.
- Evaluate and output the DBM server responses.

Communication Components

Tasks of the communication components:

- Communication with the local DBM server.
- Communication with a DBM server on a remote host.

The communication components are part of the SAP DB runtime environment (RTE).

DBM Server

A DBM server is integrated in the Database Manager CLI.

If the user requires, this integrated DBM server can be used. In this case, the components required to connect to a separate DBM server are not used.

Creating the Database Manager CLI

The Database Manager CLI is created in the SAP DB development environment by processing information from a link description [Page 25]. This link description is a file of the `lnk` file type, and is called `dbmcli.lnk` ($OWN [Page 18]`/sys/desc/dbmcli.lnk`, see also General Directory Structure of the SAP DB Development Environment [Page 9]).

Processing this link description leads to the creation of an executable program. The file `dbmcli.lnk` also defines where the executable program is stored:

- Windows NT/Windows 2000: The executable program is stored in directory `$OWN\usr\pgm`.
- UNIX: The line `->`$INSTROOT [Page 15]`/bin/dbmcli` is inserted in the link description file `dbmcli.lnk`. As a result, the executable program is stored in the directory `$OWN/usr/bin`.

Procedure

Enter the following command:

`imf dbmcli`

For information about the command `imf`, see ims.pl; img.pl; imf.pl [Page 21].

This command instructs the development environment to process the link description dbmcli.lnk [Page 69] .

The link description file does not have to be specified with its type (`lnk`), since its name is unique. The development environment automatically determines the type of the link description file, and, therefore, the type of processing.

Result

The DBMCLI (Database Manager CLI) software component is created:

- UNIX: **`$OWN/usr/bin/dbmcli`**
- Windows NT / Windows 2000: **`$OWN\usr\pgm\dbmcli.exe`**

Perform a function check [Page 74].

Integration

Creating DBMCLI with the SAP DB Development Environment [Page 67]

Link Description dbmcli.lnk

The following extract from the link description [Page 25] **`dbmcli.lnk`** contains the main commands required for Creating the Database Manager CLI [Page 68]. Special commands and switches, which may be needed for the link operation on the individual platforms, have been omitted from this example.

To create the Database Manager CLI, you must always use the complete link description file `dbmcli.lnk` in the source directory (see General Directory Structure of the SAP DB Development Environment [Page 9]).

```
&if $OS in [ WIN32, OS2 ]
    :Resource/dbmcli.rc [Page 71] remake
&endif
```

vcn12.cpp [Page 71]
vcn13.cpp [Page 71]
vcn14.c [Page 71]

cservlib [Page 72]

splib [Page 73]
eoxlib [Page 73]

```
&if $OS in [ WIN32, OS2 ]
```
    sqlusr [Page 73]
```
&else
```
    enalib [Page 73]
    enblib [Page 73]
```
&endif
```

Further Information

Explanation of the Link Description dbmcli.lnk [Page 70]

Components of DBMCLI [Page 70]

Explanation of the Link Description dbmcli.lnk

The link description dbmcli.lnk [Page 69] contains references to module files [Page 24] and link descriptions [Page 25].

- References to module files cause these module files to be translated. The translation tools are assigned on the basis of the file extension of the module files.
  Module files in this example link description:
  `dbmcli.rc`, `vcn12.cpp`, `vcn13.cpp`, `vcn14.c`
- References to link descriptions cause these descriptions to be processed in accordance with their description type.
  Link descriptions in this example:
  `cservlib`, `splib`, `eoxlib`, `sqlusr`, `enalib`, `enblib`

You can find explanations of the module files and link descriptions in the section Components of DBMCLI [Page 70].

Components of DBMCLI

The Database Manager CLI consists of the following components:

- Module file [Page 24] dbmcli.rc [Page 71]
- Module file vcn12.cpp [Page 71]
- Module file vcn13.cpp [Page 71]
- Module file vcn14.c [Page 71]
- Link description [Page 25] cservlib [Page 72]
- Link description splib [Page 73]
- Link description eoxlib [Page 73]
- Link description sqlusr [Page 73]
- Link description enalib [Page 73]
- Link description enblib [Page 73]

Other Dependencies [Page 73]

Integration

Link description dbmcli.lnk [Page 69]

dbmcli.rc

The module file [Page 24] **dbmcli.rc** in the link description dbmcli.lnk [Page 69] is only relevant under Windows NT/Windows 2000. For this reason, the module is encapsulated in a condition in the link description file `dbmcli.lnk`.

The module file `dbmcli.rc` is a resource file and is stored in the subdirectory $OWN [Page 18]`\sys\src\Resource` (General Directory Structure of the SAP DB Development Environment [Page 9]). If the module file `dbmcli.rc` is specified in the link description file `dbmcli.lnk`, the `Resource` directory must also be specified explicitly:

```
:Resource/dbmcli.rc
```

The module file `dbmcli.rc` contains source code that is specific to Windows NT/Windows 2000. When this file is specified in the link description file, the relevant resources are integrated in the Database Manager CLI.

Integration

Components of DBMCLI [Page 70]

vcn12.cpp, vcn13.cpp, vcn14.c

The module files [Page 24] **vcn12.cpp**, **vcn13.cpp**, and **vcn14.c** from the link description dbmcli.lnk [Page 69] are stored in the directory $OWN [Page 18]`/sys/src/cn`. This directory name is determined implicitly from the module file name.

These module files are the main components (see Structure of the Database Manager CLI [Page 68])of the Database Manager CLI, and contain C or C++ source codes.

- The `main` function is stored in module file `vcn12.cpp`.
- The module file `vcn13.cpp` contains certain functions required for file access and character set conversion.
- The module file `vcn14.c` is used to connect main components with communication components.

In the module files **vcn12.cpp**, **vcn13.cpp** and **vcn14.c**, `#include` instructions are used to refer to include files [Page 72] .

The dependency of a module file on an include file is described in the compilation description [Page 71].

Integration

Components of DBMCLI [Page 70]

Compilation Description

In the module files [Page 24] vcn12.cpp, vcn13.cpp, and vcn14.c [Page 71] of the link description dbmcli.lnk [Page 69], `#include` instructions are used to refer to include files.

The dependency of a module file on an include file [Page 24] is described in the compilation description [Page 25]. The compilation description appropriate to the module file is determined and evaluated implicitly using the directory in which the module file to be translated is stored.

In addition to the dependency of a module file on include files, the compilation description specifies further properties of the module files that control how they are translated by the development environment.

Further Information

Include Files [Page 72] of the module files `vcn12.cpp`, `vcn13.cpp`, and `vcn14.c`

Include Files

The dependencies of a module file [Page 24] on include files [Page 24] is explained in the following example of a module file of the link description dbmcli.lnk [Page 69].

The dependencies for the module file vcn12.cpp [Page 71] in the directory $OWN [Page 18]`/sys/src/cn` are contained in compilation description [Page 71] `$OWN/sys/desc/cn.com`. These are described by the following lines in the compilation description:

```
vcn12.cpp inc=gsp09.h,heo02.h,hcn13.h,hcn14.h,hcn90.h,...
```

This example only contains the keyword **`inc`**. The keyword **`inc`** refers to the required include files.

The include files listed after the keyword **`inc`** are created by the development environment in include directory `$OWN/wrk/incl` before the module file is translated. In doing so, the development environment processes the include files according to their file type.

Only type **`h`** include files are shown in the example above. Type **`h`** include files are copied from the relevant subdirectory under `$OWN/sys/src` to a subdirectory of the include directory. The name of this subdirectory is determined implicitly by the development environment from the name of the include file.

cservlib

The link description [Page 25] **`cservlib`** ($OWN [Page 18]`/sys/desc/cservlib.lib`) is of the type **`lib`**. The link description type **`lib`** generates a static library.

A **`lib`** type link description consists of several module files [Page 24] that are grouped after translation to form the appropriate library. The dependencies and properties of the individual module files are stored in the directory-specific compilation descriptions [Page 25].

If **`cservlib`** is entered in link description dbmcli.lnk [Page 69], the **`cservlib`** library is integrated in the Database Manager CLI.

The `cservlib` library contains the functionality of the DBM server integrated in the Database Manager CLI (see Structure of the Database Manager CLI [Page 68]).

Integration

Components of DBMCLI [Page 70]

splib

The link description [Page 25] **splib** ($OWN [Page 18]/sys/desc/splib.lib) in the link description dbmcli.lnk [Page 69] is of the type **lib**.

The splib library contains modules with functions that can be used generally.

Integration

Components of DBMCLI [Page 70]

eoxlib

The link description [Page 25] **eoxlib** ($OWN [Page 18]/sys/desc/eoxlib.lib) in the link description dbmcli.lnk [Page 69] is of the type **lib**.

The library eoxlib contains communication components of the SAP DB runtime environment (see Structure of the Database Manager CLI [Page 68]).

Integration

Components of DBMCLI [Page 70]

sqlusr, enalib, enblib

The link descriptions [Page 25] **sqlusr**, **enalib**, **enblib** ($OWN [Page 18]/sys/desc/sqlusr.lib, $OWN/sys/desc/enalib.lib, $OWN/sys/desc/enblib.lib) are of the type **lib**.

The libraries sqlusr, enalib, enblib contain communication components of the SAP DB runtime environment (see Structure of the Database Manager CLI [Page 68]). Since the SAP DB runtime environment is platform specific, different libraries are used on Windows NT/Windows 2000 and UNIX. For this reason, the link description file is encapsulated in a condition in the link description dbmcli.lnk [Page 69].

Integration

Components of DBMCLI [Page 70]

Other Dependencies

If you are using Windows NT or Windows 2000, note that the Database Manager CLI requires a dynamic library. This library must be created explicitly.

To do so, enter the following command:

```
imf sqltcp
```

For information about the command **imf**, see ims.pl; imq.pl; imf.pl [Page 21].

Integration

Components of DBMCLI [Page 70]

Function Check

You can check the function of the Database Manager CLI by executing a dbm command.

Prerequisites

Creating the Database Manager CLI [Page 68]

Procedure

1. Check whether you are actually accessing the software component you have just created, the Database Manager CLI (`$OWN` [Page 18]`/usr/bin/dbmcli`, Windows NT/Windows 2000: `$OWN\usr\pgm\dbmcli.exe`).
2. Enter the dbm command for displaying the Database Manager version:

   `dbmcli [-n <server_node>] -d <database_name> -u <dbm_userid>,<dbm_password> dbm_version`

   `dbmcli -d TST -u dbm,dbm dbm_version`

   You can find more information on the various dbm commands in the documentation *Database Manager CLI: SAP DB 7.4*.

Result

The output should roughly be as follows:

```
OK
VERSION     = 7.2.4
BUILD       = DBMServer 7.2.4    Build 009-000-220-192
OS          = UNIX
INSTROOT    = /usr/sapdb-srv
LOGON       = False
CODE        = ASCII
SWAP        = full
```