XDS Family of Products

H2D User's Guide

http://www.excelsior-usa.com

Contents

Chapter 1

Introduction

Sooner or later, every Modula-2 programmer encounters four problems. These are: absence, incompleteness, unportability, and low quality of libraries. At the same time, C/C++ programmers usually have problems *choosing* from a huge set of free, public domain, shareware, and commercial libraries of various purpose, size, and quality which are in many cases portable or are available for a number of platforms. Moreover, the *Application Programming Interfaces (APIs)* of the most widely used software products (operating systems, database engines, etc.), are defined in terms of the C programming language.

In order to use this resources galore from Modula-2, a programmer needs, first, a Modula-2 compiler which supports C calling/naming conventions and a set of types corresponding to C types, and, second, definition modules corresponding to the C headers of the library/API. Finding a suitable compiler is not a very big deal, but manual conversion of C headers turns to a real nightmare when it comes to, say, the X Window API. That is why we created H2D.

H2D does the job automatically, i.e. translates C header files into Modula-2 definition modules. H2D is intended to be used with XDS (see Appendix A) version 2.10 or later and is included in the XDS distribution package. However, the generated definition modules may be used with any ISO-compliant Modula-2 compiler. The required modifications are minor and may be done using text editor macros or a simple **REXX**, **sed**, etc script.

The source language is a subset of ANSI C, which includes declarations and preprocessor directives, with some extensions (See 4.7 and Chapter 6). Destination language is **ISO Modula-2** with some XDS language extensions. XDS allows to use the resulting definition modules with *both* Modula-2 and Oberon-2.

H2D generates definition modules suitable for either XDS-C, Native XDS, or

both. In case of Native XDS, module template for function-like C macros may be generated (See 5.2.1). In case of XDS-C, an extra header file containing C declarations of types introduced by H2D is generated (See 5.2.2).

1.1 New in version 1.30

Major improvements in v1.30:

- Generalized `#variant` directive (see 4.9.2)
- Custom mapping of C base types to Modula-2 types (see 5.3.1)
- Non-standard directives extraction (see 4.9)
- Options renamed to follow XDS compilers style (see Chapter 7)
- Control file syntax now closely matches used by XDS compilers (see Chapters 2 and 6)

1.2 Typographic conventions

1.2.1 Language descriptions

Where formal descriptions for language syntax constructions appear, an *extended Backus-Naur Formalism (EBNF)* is used.

These descriptions are set in a monospaced font.

```
Text = Text [{Text}] | Text.
```

In EBNF, brackets `[` and `]` denote optionality of the enclosed expression, braces `{` and `}` denote repetition (possibly 0 times), and the line `|` denotes other possible valid descriptions.

Non-terminal symbols start with an upper case letter (e.g. `Statement`). Terminal symbols either start with a lower case letter (e.g. `ident`), or are written in all upper case letters (e.g. `BEGIN`), or are enclosed within quotation marks (e.g. `":="`).

1.2.2 Source code fragments

When fragments of a source code are used for examples or appear within a text they are set in a monospaced font.

```
/* example.h */

typedef unsigned long int UINT;
```

Chapter 2

Configuring H2D

2.1 Setting up system search path

If you installed H2D as part of an XDS package, no additional setup is required. Otherwise you must tell your operating system where to find the executable before using H2D. Refer to the `h2d.txt` file from the on-line documentation.

2.2 Working configuration

The H2D working configuration includes an executable file and a set of system files:

`h2d.red`	Search path redirection file (see 2.3)
`h2d.cfg`	Configuration file (see 2.4)
`h2d.msg`	Message file (see 2.5)

Upon invocation, H2D tries to locate these files in the current directory and then in the directory where H2D executable resides. If a *redirection file*, `h2d.red` is found, all other files are searched for/created using paths defined in it, otherwise the current directory is used for all input and output, except files specified with directories.

The *configuration file* contains various H2D settings. If the configuration file is not found, default settings are used.

The *message file* contains texts of error messages.

2.3 Redirection file

Upon activation, H2D looks for a file called `h2d.red` — the *redirection file*. This file defines directories in which all other files are searched for or created. A redirection file has to be placed in the current directory, otherwise the *master redirection file* from the directory where H2D executable resides is used.

A redirection file consists of several *redirections*:

```
Redirection = Pattern "=" directory {";" directory}
```

`Pattern` is a regular expression which all file names used by H2D will be compared with. A regular expression is a string containing certain special characters:

Sequence	Denotes
`*`	an arbitrary sequence of any characters, possibly empty (equivalent to `{\000-\377}` expression)
`?`	any single character (equivalent to `[\000-\377]` expression)
`[...]`	one of the listed characters
`{...}`	an arbitrary sequence of the listed characters, possibly empty
`\nnn`	the ASCII character with octal code `nnn`, where n is `[0-7]`
`&`	the logical operation AND
`|`	the logical operation OR
`^`	the logical operation NOT
`(...)`	the priority of operations

A sequence of the form `a-b` used within either `[]` or `{}` brackets denotes all characters from `a` to `b`.

When H2D looks for or intends to create a file, its name is sequentially compared with all patterns from the top of the redirection file. A file is created in the first directory of the list corresponding to the matched pattern. A file is searched for in all directories in the list (from first to last) until it is found or the directory list is exhausted. If a match is not found, the file is created or searched for in the current directory. **Note:** If a match is found, the current direcory is *not* searched unless it is explicitly specified in the directory list.

It is possible to put comment lines into the redirection file. A comment line should be started with the `%` character.

Example

```
*.h       = h; .; c:\bc\include
mac_*.def = macro;
*.def     = def;
mac_*.mod = macro;
*.h2d     = h2d;
```

2.4 Configuration file

The configuration file is used to set options which control various aspects of H2D behaviour: names of generated files, source/target language extensions, mapping of C base types to Modula-2 types etc. It should reside in the current directory or in the directory with H2D executable (the *master* configuration file). However, it is recommended to use a *project file* (see Chapter 6) instead of a local configuration file to specify options for a particular set of header files.

An option is a pair (*name*, *value*). Every line in the configuration file may contain only one option setup directive. Arbitrary spaces are permitted. The % character starts a one-line comment. Option setup directives have the following syntax:

```
Option = "-" name ("-" | "+" | "=" (string | integer))
```

The same syntax is used for command line options and in a *project file* (see Chapter 6). Command-line options have the highest priority. Options specified in a project file override the configuration file settings.

Options, their meanings and valid values are described in Chapter 7.

Figure 2.1 contains a configuration file example.

2.5 Customizing H2D messages

The file `h2d.msg` contains error messages in the form

```
number text
```

The following is an excerpt from `h2d.msg`:

```
001 Can't open file %s
   .  .  .
010 Invalid use of modifier
```

. . .

Some messages contain format specifiers for additional arguments. In the example above, the message number 001 contains a `%s` specifier which is substituted with a file name when the message is printed.

In order to use a language other than English for messages it is necessary to translate message texts, preserving error numbers and the number and *order* of format specifiers.

```
-DEFEXT  = def    % file extensions
-HEADEXT = h
-MODEXT  = d
-PRJEXT  = prj
-TREEEXT = inc
-DIREXT  = dir

-DEFPFX = h2d_    % prefix for output definition modules
-MACPFX = m_      % prefix for macro prototype modules

-BACKEND = COMMON % M2 compiler compatibility mode: C, NATIVE, COMMON
-GENMACRO-        % do not generate macro prototype modules
-GENWIDTH = 70    % maximum string length in output files
-COMMENTPOS = 0   % comment position
-CHANGEDEF+       % allow to overwrite existing definition modules
-PROGRESS+        % enable progress indicator
-CSTDLIB-         % do not set C standard library option
-CPPCOMMENTS+     % recognize C++ comments
-MERGEALL+        % merge all #included headers
-GENSEP-          % separate merged headers with comments
-GENLONGNAMES-    % prepen module name with directory names
-GENENUM = CONST  % enum transtaltion mode: CONST, ENUM, AUTO
-GENTREE+         % create file with include/merge tree
-GENDIRS+         % extract non-standrard directives
-GENROVARS+       % translate constants to read-only variables

% C BASE TYPES SYNONYMS:

-ctype = signed char        =  1, CHAR
-ctype = signed int         =  4, SYSTEM.int
  .  .  .
-ctype = long float         =  8, UNDEF
-ctype = long double        =  8, UNDEF

% MODULA-2 TYPES:

-m2type = INTEGER         = 4, SIGNED
-m2type = SHORTINT        = 1, SIGNED
  .  .  .
-m2type = SYSTEM.SET32    = 4, SET
-m2type = SYSTEM.int      = 4, SIGNED
-m2type = SYSTEM.unsigned = 4, UNSIGNED
```

Figure 2.1: Configuration file example

Chapter 3

Getting Started

In this chapter we assume that H2D is properly installed and configured (See Chapter 2).

3.1 Creating a working directory

Redirection files (see 2.3) give you complete freedom over where you keep your header files and any files which H2D itself creates for further use. It is recommended to work in a project oriented fashion — i.e. to have a separate directory hierarchy for each set of header files you wish to translate.

In this case, each project shall have a main working directory. The script called `h2dwork` may be used to create the required subdirectories and a redirection file. For example, to create a directory structure for a project called `myproj` in the current directory, issue the following commands:

```
mkdir myproj
cd myproj
h2dwork
```

Note: Since H2D preserves directory hierarchies of original header files. you may also need to create additional subdirectories. See 4.8.2 for more information.

3.2 Invoking H2D

H2D is implemented as a command line utility called `h2d`. To translate a header file (or a set of header files), type

```
h2d { HeaderFile } { Option } [ -prj=ProjectFile ]
```

at the command prompt, where `HeaderFile` is a header file name.

The syntax for `Option` is described in 2.4.

If you specify the `-prj` option, each header will be translated as if it was specified in a `!module` directive (see 6.2.2) in `ProjectFile`.

To process a *project file* (see Chapter 6), type

```
h2d =p ProjectFile { Option }
```

To view the default option values, type

```
h2d =o
```

If invoked without parameters, the utility prints a brief help information.

3.3 H2D usage example

Copy the H2D sample included in your XDS or H2D distribution to a working directory and type

```
h2d =p example.h2d
```

at the command prompt. The H2D banner line will appear:

```
H2D v1.30 (c) XDS 1996-1997
File example.h
```

After translation the following lines will be displayed:

```
no errors, lines 23.
-------------------------------------------
Files 1, lines 23, no errors, time 0:3.
```

showing the number of errors, the number of source lines in the file, and some statistics. The following files will be generated:

`h2d_example.def`	basic definition module
`h2d_example.h`	definitions of types generated by H2D (see 5.2.2)
`mac_example.def`	macro definition module (see 5.2.1)
`mac_example.mod`	prototype macro implementation module (see 5.2.1)

3.4 Error reporting

When H2D detects an error in the input file, it displays an error report. It contains the file name and position (line and column numbers) where the error occurred:

```
Error [ example.h 16:44 ] ** Duplicate identifier 'insert'
```

The error which is often encountered is

```
Error [...] ** Expected  , or ;
```

In most cases it means that an identifier is undefined for some reason. Try to put "," or ";" at the specified position to find out what is the problem source.

Chapter 4

Translation Rules

4.1 Comments

All comments from the original C text are copied to generated definition modules. Their placement, however, is not preserved in some cases. The **COMMENTPOS** option may be used to align comments which are placed next to declarations.

C++ compilers are usually able to recognize C++-style comments (beginning with '`//`') even while operating in C mode. The **CPPCOMMENTS** option controls whether H2D recognizes such comments as well.

4.2 Identifiers

In most cases, H2D preserves original C identifiers. Exceptions are structure, union, and enumeration tags, which constitute a separate name space in C. If there is a constant, type, variable, or function identifier which coincides with a tag, H2D appends "`_struct`", "`_union`" or "`_enum`" to that tag.

In some situations, H2D itself generates additional identifiers, e.g. for unnamed function arguments (see 4.6), derived types (see 4.3.1), and formal types (see 4.6).

H2D may append digits to generated identifiers to avoid conflicts with existent ones.

Identifiers matching Modula-2 *keywords* are not allowed in source files. However, Modula-2 *pervasive identifiers* (e.g. `INTEGER` or `HALT`) are permitted.

Example

The following C declarations were taken from the `sys/stat.h` file:

```
struct stat { ... };
int stat( const char *, struct stat * );

TYPE stat_struct = RECORD ... END;
     PtrChar = POINTER TO CHAR;
PROCEDURE stat(arg0: PtrChar; arg1: stat_struct): SYSTEM.int;
```

4.3 Types

C types are translated to Modula-2 types according to the following table:

C type	Modula-2 type
base (`int`, `char`, etc.)	(see 5.3.1)
`void*`	`SYSTEM.ADDRESS`
pointer	pointer
array	array
enumeration	(see 4.3.2)
structure	record
union	variant record
pointer to function	procedure type

Notes:

- Structure, union, and enumeration *tags* are not preserved in cases of collision. It may cause problems with Modula-2 to C convertors (e.g. XDS-C). See 4.2.

Examples

```
struct STRUCTURE{
  int    field1;
  char   field2;
  double field3;
};
```

```
TYPE STRUCTURE = RECORD
    field1: SYSTEM.int;
    field2: CHAR;
    field3: LONGREAL;
  END;
```

```
union  UNION {
  int    field1;
  char   field2;
  double field3;
};
```

```
TYPE UNION = RECORD
    CASE : INTEGER OF
     0: field1: SYSTEM.int;
    |1: field2: CHAR;
    |2: field3: LONGREAL;
    END;
  END;
```

4.3.1 Derived types

For objects declared as having derived types (pointer or array) either a new Modula-2 type is introduced or a previously declared type synonym is used to improve readability. For pointers to base types, a new type is always declared.

In particular, a C structure may contain fields which type is defined as pointer to that structure. In this case H2D also automatically inserts a necessary forward pointer type declaration.

This may cause type compatibility problems. Fortunately, in XDS (see Appendix A), compatibility rules for foreign objects are relaxed, e.g. two "C" pointer types are compatible if their *base types* are the same. Additional setup or postprocessing may be required when H2D is used with third-party Modula-2 compilers.

```
char *str1;
char *str1;
```

```
TYPE
  H2D_PtrSChar = POINTER TO CHAR;

VAR
  str1: H2D_PtrSChar;
  str2: H2D_PtrSChar;
```

```
struct s {
  int i;
};

struct s *p;
```

```
TYPE
  s = RECORD
    i: SYSTEM.int;
  END;
  H2D_Ptrs = POINTER TO s;

VAR
  p: H2D_Ptrs;
```

```
struct s {
  int i;
};

typedef struct s *ps;

struct s *p;
```

```
TYPE
  s = RECORD
    i: SYSTEM.int;
  END;

  ps = POINTER TO s;

VAR
  p: ps;
```

```
struct Node {
   int i;
   struct Node *next;
};
```

```
TYPE
  PtrNode = POINTER TO Node;
  Node = RECORD
    i: SYSTEM.int;
    next: PtrNode;
  END;
```

4.3.2 Enumeration

An enumeration (`enum`) is not actually a distinct type in C — it is just a convenient way to declare integer constants (but in C++ enumeration *is* a distinct type). Moreover, since it is possible in C to explicitly specify enumeration constant value, translation to Modula-2 enumeration type may be incorrect. H2D may translate C enumerations into either Modula-2 enumeration types or Modula-2 constant declarations, depending on the **GENENUM** option setting. For instance, if **GENENUM** is set to `"Const"` or `"Mixed"`, the following C type synonym declaration:

```
typedef enum{ one=1, two } Number;
```

will be translated to

```
(* H2D: enumerated type: Number *)
CONST
  one = 1;
  two = 2;
TYPE
  Number = SYSTEM.int;
(* H2D: End of enumerated type: Number *)
```

If **GENENUM** is set to `"Enum"`, the same declaration will be translated unsafely (a warning comment will be added):

```
TYPE
  Number = (
    one, (* H2D: integer value was 1 *)
    two
  );
```

4.4 Type synonyms

C declarations of type synonyms (`typedef`) are translated to Modula-2 type declarations. If there are multiple synonyms declared for a type, their equivalence is preserved:

```
typedef char String[256];  TYPE String = ARRAY [0..255] OF CHAR;
typedef String *PString;         PString = POINTER TO String;
typedef String *Buffer;          Buffer = PString;
```

Note: In C, function type synonyms may be used in function declarations. These synonyms are processed in a way they are processed by a C compiler and do not appear in output files (see 4.6).

4.5 Variables

Variables are translated to variables. Variables declared with the `const` qualifier are translated to read-only variables (XDS extension). The `volatile` qualifier is currently ignored.

```
extern int i;              VAR i  : SYSTEM.int;
extern const int j;        VAR j- : SYSTEM.int;
```

4.6 Function prototypes

C function prototypes declared as `void` are translated to proper procedure declarations; other are translated to function procedure declarations. If there is no name specified for a function parameter, `arg`*x* is substituted, where *x* is a number unique for each unnamed parameter.

In C, a derived type (more precisely, pointer or array) may be specified for a function parameter. In Modula-2, the *formal type* of a procedure parameter have to be either type name or open array type. H2D translates parameters of array type to open array value parameters.

The translation procedure for pointers is more complicated. By default, H2D searches for a type synonym, previously declared via `typedef`. The synonym, if found, is used as formal type; otherwise H2D automatically declares one. If automatic declaration is undesirable, required synonyms may be declared in the *project file* (see Chapter 6). Other variants of translation may be explicitly specified by means of the `#variant` directive (see 4.9.2).

See also 4.7.

Examples

```
void p(int,int);           PROCEDURE p ( arg0 : SYSTEM.int;
                                         arg1 : SYSTEM.int );

int f(char c);             PROCEDURE f ( c : CHAR ): SYSTEM.int;

void P(T *t);              TYPE PtrT = POINTER TO T;
                           PROCEDURE P ( t : PtrT );

void Q(T t[])              PROCEDURE Q ( t : ARRAY OF T );

int strlen(char *);        PROCEDURE strlen ( arg0 : ARRAY OF CHAR )
#variant strlen(0) : ARRAY                  : SYSTEM.int;
```

4.7 Non-standard qualifiers

In practice, header files are not "pure" ANSI C. The most common extension is a set of additional keywords (qualifiers) which may be used to specify calling/naming conventions used in a particular library or API.

Since XDS provides the similar mechanism called *direct language specification (DLS)*, H2D recognizes a number of such keywords, which are translated to the following DLS strings:

C keyword	DLS String
`cdecl`	none
`fortran`	none
`interrupt`	none
`pascal`	`"Pascal"` for types and variables `"StdCall"` for functions
`syscall`	`"Syscall"`

`near`, `far`, and `huge` qualifiers are recognized but ignored.

4.8 Preprocessor directives

4.8.1 Macro definitions

A `#define` C preprocessor directive may contain an *object-like* definition or a *function-like* definition which are translated differently.

```
#define identifier Text
```

If `Text` is a constant expression, the directive is translated to a constant declaration or a read-only variable declararion (see 5.3.3). If `Text` is a type identifier, the directive is translated to a type declaration. If `Text` is an identifier of a function, a macro definition, or a constant, the directive is translated to a constant declaration. In all other cases, it is interpreted in a C preprocessor manner.

```
#define identifier"(" idenifier, { identifier } ")" Text
```

Translated to a proper procedure declaration with all parameters having type `ARRAY OF SYSTEM.BYTE`. These declarations are marked with a special comment and may be corrected after translation to reflect the actual semantics of a macro by changing parameter types and/or adding return types. See also 5.2.1 for information about macro prototype modules.

`#undef identifier`

Undefines `identifier` as it is done by a C preprocessor.

Example

```
#define str_constant "Hello World!\n"

#define constant 0x10
#define constant_synonym constant

#define macro_with_params(p1,p2,p3) p1+p2+p3
#define macro_with_params_synonym macro_with_params

int function(int);
#define function_synonym function

typedef int INT;
#define INTEGER INT
```

```
CONST
  str_constant = 'Hello World!' + 12C;
  constant = 10H;
  constant_synonym = constant;

<* IF __GEN_C__ THEN *>

(* H2D: this procedure was generated from Macro. *)
PROCEDURE macro_with_params ( p1, p2, p3: ARRAY OF SYSTEM.BYTE );

<* ELSE *>

PROCEDURE  / macro_with_params ( p1, p2, p3: ARRAY OF SYSTEM.BYTE );

<* END *>

<* IF __GEN_C__ THEN *>

CONST
  macro_with_params_synonym = macro_with_params;
```

```
<* END *>

PROCEDURE function ( arg0: SYSTEM.int ): SYSTEM.int;

CONST
  function_synonym = function;

TYPE
  INT = SYSTEM.int;

  INTEGER = SYSTEM.int;
```

4.8.2 File inclusion

```
#include <file_name>
#include "file_name"
```

If the file specified by `file_name` has to be merged with the current file (see 5.1), H2D treats this directive exactly as a C preprocessor, i.e. replaces it with contents of a specified file. Otherwise, `file_name` is added to the import list and the file specified by it is translated into a separate definition module. If `file_name` contains directories, the output files are placed to the same subdirectory.

The **GENLONGNAMES** option controls conversion of included header file names which contain path:

```
#include <sys/stat.h>
```

is translated to

```
IMPORT stat;
```

if **GENLONGNAMES** is OFF and to

```
IMPORT sys_stat;
```

if **GENLONGNAMES** is ON.

See also 4.10.

4.8.3 Conditional compilation

H2D handles conditional compilation directives `#if`, `#ifdef`, `#ifndef`, `#else`, and `#endif` the same way as a C preprocessor does. A *project file*

(see Chapter 6) may be used to define constants which are used in arguments of these directives.

4.8.4 Other directives

H2D recognizes and ignores `#line`, `#error`, and `#pragma` C preprocessor directives.

4.9 Non-standard preprocessor directives

H2D recognizes two non-standard preprocessor directives: `#merge` and `#variant`. These directives are related to definiton module generation only and do not affect the C text, so they may be placed arbitrarily in a header file. Typically they are collected in project files inside a corresponding `!header` directive (see 6.2.1).

The advanced technique is to put these directives right into working copies of header files, next to the corresponding declarations. Then, after successful translation of all headers, these directives may be extracted with the help of **GENDIRS** option and moved to the project file. Now original headers may be used for translation.

4.9.1 #merge

```
#merge ( <file_name> | "file_name" )
```

This directive lists included header files which should be merged even if the **MERGEALL** option is OFF. This feature may be useful in some cases (see 5.1).

When placed in a header file, this directive has effect only in this file. When placed in a *project file* (see Chapter 6), it has effect in all headers matching the surrounding `!header` directive (see 6.2.1).

4.9.2 #variant

```
#variant Designator ":" Type
Designator = identifier { "^" | "[ ]" | "." identifier } |
             Parameter
```

```
Parameter  = identifier "(" number ")"
Type       = qualident
```

This form of the `#variant` directive allows to explicitly specify a Modula-2 `Type` for an object denoted by `Designator`. See 5.3.1 for more information.

`Designator` specifies a named object or its element which is subject to the `#variant` directive:

`"^"`	pointer dereference
`"[ ]"`	array indexing
`"." identifier`	structure or union field selection

`Parameter` specifies a function `identifier` and its parameter `number` (zero-based).

```
#variant Parameter  ":" ( "VAR" | "ARRAY" | "VAR ARRAY" )
```

This form is used to control translation of a function `Parameter`, which has a pointer type.

By default, pointer type function parameters are translated to pointer type procedure parameters (see 4.6). The `#variant` directive allows to specify one of the following alternative rules for a particular parameter:

`Modifier`	`T *p` is translated to
`VAR`	`VAR p:  T`
`ARRAY`	`p:  ARRAY OF T`
`VAR ARRAY`	`VAR p:  ARRAY OF T`

See also 5.3.2.

```
#variant f(0) : VAR ARRAY
void f(char*);

PROCEDURE f ( VAR arg0: ARRAY OF CHAR );
```

A `#variant` directive has effect only in the file where it is located, or, if specified in a project file, in all files matching the surrounding `!header` directive. Therefore, `Designator` should specify an object declared in this file or in one of the files which it includes. If an object specified by `Designator` is not present, an error message is displayed.

4.10 Module names

By default, H2D uses a header file name without "`.h`" extension as a definition module name. If a file name contains characters which are not allowed in identifiers, the `!name` directive (see 6.2.3) must be used in a *project file* (see Chapter 6) to specify a proper identifier.

Chapter 5

Using H2D

5.1 Headers merging

A C header file may contain one or more `#include` directives. H2D offer the following translation variants for included headers:

- All headers are merged and translated into a single definition module (the **MERGEALL** option is set ON).
- Each included header is translated into a separate definition module which name is added to the import list (the **MERGEALL** option is set OFF).
- The headers which have to be merged are explicitly specified using the `#merge` directive (the **MERGEALL** option must be OFF).

If the **GENSEP** option is set ON, H2D separates pieces of Modula-2 text, which correspond to different merged headers, with comments containing header file names.

The `#merge` directive (see 4.9.1) provides more flexible method of merging control than the **MERGEALL** option. The example illustrates situation in which this directive is very helpful.

Example

```
/* m1.h */
typedef int INTEGER;
```

```
/* m2.h */
struct descriptor{
```

```
#include <m2.h>
#include <m3.h>
#include <m4.h>
RETVAL handler();
/* end m1.h */
```

```
  INTEGER handl;
};
typedef int far * RETVAL;
/* end m2.h */
```

In this example, the `RETVAL` declaration from `m2.h` is used in `m1.h`. On the other hand, `m2.h` uses the declaration from `m1.h` (`INTEGER`). Setting the **MERGEALL** option ON results in all headers (`m1.h`, `m2.h`, `m3.h`, and `m4.h`) being merged and translated into the single definition module `m1`. If this is not a desired behaviour, the `#merge` directive (see 4.9.1) should be used instead. If the **MERGEALL** option is OFF and the line

```
#merge <m2.h>
```

is added to either `m1.h` or the corresponding `#header` directive in a *project file* (see Chapter 6), H2D produces *three* definition modules `m1`, `m3` and `m4`, where `m1` is a result of translation of two merged headers `m1.h` and `m2.h`.

5.2 Fitting a Modula-2 compiler

Some of H2D translation rules depend on the target Modula-2 compiler; even XDS-C and Native XDS require different definition modules. The **BACKEND** option is used to reflect the major difference: whether the target Modula-2 compiler is a native code compiler (`-BACKEND = Native`) or a convertor to C (`-BACKEND = C`). It is also possible to produce definition modules suitable for both XDS-C and Native XDS (`-BACKEND = Common`). In this case H2D encloses target-dependent parts with XDS conditional compilation directives.

5.2.1 Native code

C headers often contain a number of useful function-like macros. These macros are translated into procedure declarations with parameters having type `ARRAY OF SYSTEM.BYTE`, which is assignment compatible with any other type. But C macros exist only at compile-time and are not present in object files. Therefore, a Modula-2 compiler is unable to handle them properly unless it is implemented as a convertor to C. Nevertheless, H2D provides a technique which allows to use C function-like macros even with a native code Modula-2 compiler.

If the **BACKEND** option is set to either `Native` or `Common` and the **GENMACRO** option is set ON, H2D produces an additional module containing macro *prototypes* — procedures corresponding to function-like macros, which bodies consist of a comment with C macro definition and are expected to be written by a programmer. These procedures are then declared as external in the main definition module. Thus, a macro prototype module need not to be imported, it should be just *linked* into an executable which uses the generated definition module.

A macro prototype module name is constructed from a header module name and a prefix specified by the **MACPFX** option.

Example

```
/* macro.h */
    ...
#define cube(x) (x*x*x)
    ...
```

```
(* macro.def  Sep 20  2:38:9  1996 *)
    ...
DEFINITION MODULE ["C"] macro;
    ...
PROCEDURE  / cube ( x: ARRAY OF SYSTEM.BYTE );
    ...
END macro.
```

```
(* m_macro.def  Sep 20  2:38:9  1996 *)
    ...
DEFINITION MODULE m_macro;

IMPORT SYSTEM;
    ...
PROCEDURE ["C"] cube ( x: ARRAY OF SYSTEM.BYTE );
    ...
END m_macro.
```

```
(* m_macro.mod  Sep 20  2:38:9  1996 *)
    ...
IMPLEMENTATION MODULE m_macro;

IMPORT SYSTEM;
    ...
PROCEDURE ["C"] cube ( x: ARRAY OF SYSTEM.BYTE );
(*
#define cube(x) (x*x*x)
*)
BEGIN
END cube;
    ...
END m_macro.
```

5.2.2 Convertor to C

A Modula-2 compiler implemented as a convertor to C (e.g. XDS-C) converts definition modules written by a programmer to C headers. But headers corresponding to definition modules generated by H2D already exist. To prevent them from being overridden, H2D inserts the **NOHEADER** XDS option, which disables header file generation, at the beginning of each definition module.

For all included header files, which are not merged (see 5.1), H2D also sets the **CSTDLIB** XDS option according to the parenthesis used in the `#include` directive – double quotes or angle brackets. For top-level header files, this option is set equal to the value of the **CSTDLIB** option.

H2D usually has to introduce a number of additional types in the definition module (see 4.6 and 4.3). These types are absent in the original header file, and their usage would cause C compilation to fail. To solve this problem, H2D constructs a resulting definition module name from a header file name and a prefix specified by the **DEFPFX** option. Then, it produces a *"wrapper" header file*, which name corresponds to the name of a *definition module*, containing an `#include` directive with original header name, followed by required type declarations. Type declarations from a *project file* (see Chapter 6) are copied to a wrapper file as well.

Example

```
/* type.h */
```

```
struct Node {
  struct Node *next;
  struct Node *prev;
  int          hash;
};

int Hash(char * str);
```

```
(* h2d_type.def  Sep 20  2:51:7  1996 *)
    ...
DEFINITION MODULE ["C"] h2d_type;

IMPORT SYSTEM;
    ...
<*- GENTYPEDEF *>

TYPE
  PtrNode = POINTER TO Node;

<*+ GENTYPEDEF *>

  Node = RECORD
    next: PtrNode;
    prev: PtrNode;
    hash: SYSTEM.int;
  END;

<*- GENTYPEDEF *>

  PtrSChar = POINTER TO CHAR;

PROCEDURE Hash ( str: PtrSChar ): SYSTEM.int;

END h2d_type.
```

```
/* h2d_type.h  Sep 20  2:51:7  1996 */
    ...
#include "type.h"
```

```
#ifndef h2d_type_H_
#define h2d_type_H_

typedef struct Node * PtrNode;
typedef signed char * PtrSChar;

#endif  /* h2d_type_H_ */
```

5.3 Modifying translation rules

5.3.1 Base types mapping

The **CTYPE** and **M2TYPE** options in conjunction with the `#variant` directive (see 4.9.2) provide complete control over mapping of C base types to Modula-2 types.

The **CTYPE** option specifies sizes (in bytes) of C base types, and their default mapping to Modula-2:

```
-CTYPE = float              = 4, REAL
  . . .
-CTYPE = unsigned short int = 2, SYSTEM.CARD16
```

The **M2TYPE** option specifies Modula-2 types supported by a particular compiler, with their sizes and families to which they belong:

```
-M2TYPE = CARDINAL         = 4, UNSIGNED
-M2TYPE = CHAR             = 1, CHAR
  . . .
-M2TYPE = SYSTEM.SET16     = 2, SET
```

Finally, the `#variant` directive (see 4.9.2) allows to explicitly specify a Modula-2 type for a particular object:

```
void f(unsigned short mask)      PROCEDURE f(mask : SYSTEM.SET16);
#variant f(0) : SYSTEM.SET16
```

Note: In order to keep original headers intact, `#variant` directives may be placed into the *project file* (see Chapter 6) inside the corresponding `!header` directive (see 6.2.1).

H2D checks type mappings for correctness using the following rules:

- type sizes must match
- floating point C types may only be mapped to Modula-2 types defined as REAL by **M2TYPE** option.
- signed integer C types may be mapped to any Modula-2 type except REAL and UNSIGNED.
- unsigned integer C types may be mapped to any Modula-2 type except REAL and SIGNED.

One of the most advanced features of this mechanism is the ability to use Modula-2 set types for C objects. The C programming language, as well as C++, has no built-in set type. The common practice is to treat unsigned integer types as bit scales and to use bitwise logical operators to manipulate them. Since Modula-2 provides set types (and no bitwise operators), it would be more convenient to translate individual integer constants and types to set constants and types.

Example

```
struct s{
  unsigned int field;
};
typedef unsigned long BITSCALE;
int variable;
void long function(unsigned argument);
char bitarray[10];
#define constant 0x0011

#variant s.field     : BITSET
#variant BITSCALE    : BITSET
#variant variable    : BITSET
#variant function(0) : BITSET
#variant bitarray[]  : SYSTEM.SET8
#variant constant    : SYSTEM.SET16
```

```
TYPE
  s = RECORD
    field: BITSET;
  END;
```

```
  BITSCALE = BITSET;

VAR
  variable: BITSET;

PROCEDURE function ( argument: BITSET );

VAR
  bitarray: ARRAY [0..9] OF SYSTEM.SET8;

CONST
  constant = SYSTEM.SET16{0, 4};
```

5.3.2 Pointer type function parameters

In C, the actual semantics of a pointer type function parameter depends on that function and cannot be determined automatically. A function may interpret its parameter of type `T*` as either:

- pointer to T (type defined as `POINTER TO T`)
- single value passed by reference (`VAR T`)
- array (`ARRAY OF T`)
- array passed by reference (`VAR ARRAY OF T`)

where the corresponding Modula-2 formal types are given in parenthesis.

The `#variant` directive (see 4.9.2) may be used to explicitly point out the semantics of each pointer type parameter.

Note: In order to keep original headers intact, `#variant` directives may be placed into the *project file* (see Chapter 6) inside the corresponding `!header` directive (see 6.2.1).

Example

```
#variant function(0) : VAR
#variant function(1) : ARRAY
#variant function(2) : VAR ARRAY
```

```
void function(int*, int*, int*, int*);
```

is translated to:

```
TYPE
  PtrSInt = POINTER TO SYSTEM.int;

PROCEDURE function ( VAR arg0: SYSTEM.int;
                        arg1: ARRAY OF SYSTEM.int;
                    VAR arg2: ARRAY OF SYSTEM.int;
                        arg3: PtrSInt );
```

5.3.3 Preserving constant names

By default, a `#define` directive introducing a constant is translated to a constant declaration:

```
#define ENOTEXIST 10
```

```
CONST
  ENOTEXIST = 10;
```

This is the only way in case of a native code Modula-2 compiler, since such constants are substituted by a C preprocessor and do not appear in object files. But in case of a convertor to C, the original C headers will be used after conversion and it would be useful to refer to their names in the generated C text. Setting the **GENROVARS** option ON forces constants to be translated to read-only variables (**Note:** this is an XDS language extension). This option has no effect on generation for a native-code Modula-2 compiler.

The `#variant` directive (see 4.9.2) may be used to specify a type for a variable:

```
#define  ENOTEXIST 10
#variant ENOTEXIST : CARDINAL
```

```
VAR
  ENOTEXIST-: CARDINAL;
```

Chapter 6

Project files

6.1 Overview

The most powerful and multi-purpose feature of H2D is the *project file*, which name may be specified at the command line after `=p`.

The project file may be used:

- To translate a set of header files at once, using the same option settings:

```
-BACKEND=C
-GENROVARS+
!module <stdio.h>
!module <stdlib.h>
```

- To map calling/naming convention qualifiers used by a particular C compiler to those supported by H2D (see 4.7), or to make them ignorable:

```
!header <*.h>            /* define for all headers */
#define _System syscall
#define far
!end
```

- To define macros which are predefined by a particular C compiler or are used in conditional compilation preprocessor directives (see 4.8.3):

```
!header <*.h>
#define __WATCOM_C__
#define INCL_DOS
!end
```

- To declare type synonyms in order to prevent H2D from automatic type declarations (see 4.6 and 4.3):

```
!header <*.h>
typedef char *String     /* no more PtrChar */
!end
```

- To collect non-standard preprocessor directives in order to keep original header files intact (see 6.2):

```
!header <string.h>
#variant strlen(0) : ARRAY
!end
```

Example

The header files `a.h` and `m1.h`:

```
/* a.h */
  #include "m1.h"
  #define constant1 0x11u
  __PASCAL function3( float * arg0, unsigned long arg1 );
/* end a.h */

/* m1.h */
  #define constant2 0x111u
  __PASCAL function1( float * arg0, float * arg1 );
  function2( unsigned long arg0, unsigned long arg1 );
/* end m1.h */
```

with project file `p.h2d`:

```
!header "*.h"
  #define __PASCAL pascal
!end
!header "a.h"
  #merge "m1.h"
!end
!header "m1.h"
  #variant function1 (1) : VAR
  #variant constant2 : BITSET
  #variant function2 (1) : BITSET
!end
!module "a.h"
```

are translated to

```
(* ********************** *)
(*           m1.h         *)
(* ********************** *)
CONST
  constant2 = {0, 4, 8};
<*- GENTYPEDEF *>
TYPE
  PtrFloat = POINTER TO REAL;
PROCEDURE ["StdCall"] function1 ( arg0: PtrFloat;
                            VAR arg1: REAL ): SYSTEM.int;
PROCEDURE function2 ( arg0: LONGCARD; arg1: BITSET ): SYSTEM.int;
(* ********************** *)
(*           a.h          *)
(* ********************** *)
CONST
  constant1 = 111H;
PROCEDURE ["StdCall"] function3 ( arg0: PtrFloat;
                             arg1: LONGCARD ): SYSTEM.int;
```

6.2 Project file contents

A project file may contain options settings and directives. Option settings in a project file override settings in the configuration file.

H2D recognizes the following directives in project files: `!header`, `!module`, and `!name`, which are described is the following sections.

6.2.1 !header

```
!header ( '<' Pattern '>' | '"' Pattern '"' )
  Prologue
[
!footer
  Epilogue
]
!end
```

`Pattern` is a regular expression (see 2.3) representing a set of file names.

`Prologue` and `Epilogue` are arbitrary sequences of C language tokens. `Prologue` is inserted at the beginning of any header file which name matches `Pattern`; `Epilogue` is appended to its end. `!footer` and `Epilogue` may be omitted.

If a header file name matches `Pattern` in more than one `!header` directive, their `Prologue` and `Epilogue` sections are merged.

`Prologue` usually contains:

- `#merge` and `#variant` directives
- Predefined macros of a particular C compiler
- Type synonym declarations to prevent automatic type names generation

Note: If there are `#include` directives in either `Prolodue` or `Epilogue` ensure that names of the included files do not match `Pattern`, to avoid recursive inclusion:

```
!header <*.h&^mytypes.h>
#include <mytypes.h>
!end
```

6.2.2 !module

```
!module ( <file_name> | "file_name" )
```

The `!module` directive is used to specify header files which are to be translated when H2D processes the project file.

Translating more than one header at once has one more advantage. A header file name may occur multiple times in `#include` directives. H2D keeps information about each translated header in memory, and if an already transtaled header file is encountered, it is not processed again. **Note:** In this case H2D requires more memory.

Example

```
!module <ctype.h>
!module <math.h>
```

```
!module <stdio.h>
!module <stdlib.h>
!module <string.h>
```

6.2.3 !name

```
!name ( '<' file_name '>' | '"' file_name '"' ) identifier
```

H2D replaces `file_name` with `identifier` when generating module names. This may be useful when `file_name` contains special characters (e.g. `my-header.h`), or when there are headers with equal names in different directories. See also the description of the **GENLONGNAMES** option.

Example

```
!name <errno.h>     errno
!name <sys\errno.h> syserrno
```

Chapter 7

Options Reference

This chapter contains brief descriptions of all options that may be specified in either configuration file (See 2.4). or *project file* (see Chapter 6).

The general option syntax is:

```
Option = "-" name ("+" | "-" | "=" (string | integer))
```

Option names are case insensitive. Case is preserved when string option values are stored or emitted, but is ignored when they are compared.

Examples

```
-mergeall+
-GENWIDTH=78
-BackEnd=Common
```

7.1 File extensions and prefixes

This group of options sets file name extensions and prefixes for H2D input and output files.

Option	Sets extension for...	Default
DEFEXT	definition modules	`def`
DIREXT	directive files (see 4.9)	`dir`
HEADEXT	header files	`h`
MODEXT	implementation modules	`mod`
PRJEXT	project files (see Chapter 6)	`h2d`
TREEEXT	include tree files (see the **GENTREE** option)	`tre`

Option	Sets prefix for...	Default
DEFPFX	definition modules (see 5.2.2)	none
MACPFX	macro prototype modules (see 5.2.1)	`_`

7.2 Translation options

Option	Meaning	Default
BACKEND	target compiler back-end	`Common`
CHANGEDEF	enable retranslation	`OFF`
COMMENTPOS	preserved comments position	undefined
CPPCOMMENTS	recognize C++ comments	`ON`
CSTDLIB	set CSTDLIB value	`OFF`
GENDIRS	extract non-stsndard directives	`OFF`
GENENUM	enum translation mode	`Const`
GENLONGNAMES	keep directory names	`OFF`
GENMACRO	produce macro prototype modules	`OFF`
GENROVARS	translate constants to r/o vars	`OFF`
GENSEP	separate merged headers	`OFF`
GENTREE	generate inclusion tree	`OFF`
GENWIDTH	limit output line length	unlimited
MERGEALL	merge all headers	`OFF`
PROGRESS	enable progress indicator	`OFF`

For each option, a set of possible values is given in parenthesis:

string	an arbitrary sequence of characters
numeric	an unsigned integer number
boolean	ON or OFF

BACKEND (Native/C/Common)

Enables generation of definition modules suitable for either native-code

compiler, translator to C, or both. It also affects generation of additional modules (See 5.2).

Default: `Common` - both native and translator.

CHANGEDEF (boolean)

If this option is set OFF, H2D does not translate a header file if a definition module corresponding to it already exists. Otherwise, H2D produces a new definition module which may overwrite the old one.

See also 2.3.

Default: OFF - do not process already translated headers

COMMENTPOS (numeric)

Sets starting position of comments preserved by H2D in output files. Has effect only on comments which are placed next to declarations.

Default: delimit comments with a single blank.

CPPCOMMENTS (boolean)

If this option is set ON, H2D recognizes C++-style comments (started with `'///'`) in header files.

Default: ON - recognize C++ comments.

CSTDLIB (boolean)

Sets value of the **CSTDLIB** XDS option in output definition modules corresponding to top-level header files (listed on the command line or in a *project file* (see Chapter 6)). See 5.2.2 for more information.

Default: OFF - set **CSTDLIB** off in definition modules.

GENDIRS (boolean)

If this option is set ON, a file containing non-standard preprocessor directives (see 4.9) is produced for each header file.

Deafult: OFF - do not extract non-standard directives

GENENUM (Const/Enum/Mixed)

Defines whether C enumerations should be unconditionally translated to integer constants (`Const`) or Modula-2 enumeration types (`Enum`). If set to `Mixed`, only enumerations with default (or explicitly specified, but matching default) constant values are translated to enumeration types; all other are translated to constants. See also 4.3.2.

Default: `Const` - always translate to constants

GENLONGNAMES (boolean)

This option controls translation of the `#include` directive in cases when a specified file name contains directories. If this option is set OFF, H2D strips directory names. Otherwise directory names are kept and separators are replaced with underscore characters:

```
#include <sys\errno.h>          IMPORT sys_errno;
```

See also 6.2.3.

Default: OFF - strip directory names

GENMACRO (boolean)

Setting this option ON forces H2D to produce prototype modules for function-like C macros encountered in header files (*prototype* means that implementation modules contain procedures with empty bodies; the actual code has to be written by hand). These modules have not to be imported, but to be added to the link list. See also 5.2.1.

This option is ignored if the target is XDS-C (the **BACKEND** option is set to `C`).

Default: OFF - do not generate macro modules.

GENROVARS (boolean)

Being set ON, enables translation of `#defined` constants to read-only variables. Have no effect in native back-end sections. See 5.3.3 for more information.

Default: OFF - do not translate constants to variables.

GENSEP (boolean)

Setting this option ON forces H2D to insert a comment containing name of a merged header before declarations from that header.

Default: OFF - do not insert separators

GENTREE (boolean)

If this option is set ON, H2D produces a text file containing a tree of `#include` directives for each header file specified on the command line or in a project file using the `!module` directive (see 6.2.2). A name of a tree file is a name of a corresponding header file with extension defined by the **TREEEXT** option.

Default: OFF - do not produce tree files

GENWIDTH (numeric)

Sets maximum length of a string in the output file.

Default: do not limit string length.

MERGEALL (boolean)

If this option is set ON, H2D merges all header files included into the translated header by means of the `#include` directive. If this option is set OFF, H2D generates separate definition module for each header which is not specified in the `#merge` directive.

See also 4.8.2 and 5.1.

Default: OFF - do not merge headers which are not specified in the `#merge` directive.

PROGRESS (boolean)

Setting this option ON enables progress indicator.

Default: OFF - show no progress indicator

7.3 Base types definition

The base types definition and mapping options, **CTYPE** and **M2TYPE**, have a special syntax. Either of them may be specified more than once in a project or configuration file, provided that each time a new type is defined. See 5.3.1 for more information on usage of these directives.

By default, H2D is configured to support XDS compilers.

CTYPE (special)

This option defines size and default mapping of a C base type.

```
CTypeOption = '-' 'CTYPE' '=' Type '=' size ',' qualident
Type = 'signed char'     | 'unsigned char'      |
       'short int'       | 'unsigned short int' |
       'signed int'      | 'unsigned int'       |
       'signed long int' | 'unsigned long int'  |
       'float'           | 'double'             |
       'long float'      | 'long double'
```

`size` is the size of `Type` in bytes, and `qualident` is a Modula-2 type to which `Type` should be translated by default:

```
-CTYPE = signed char = 1, CHAR
```

See 5.3.1 for more information.

M2TYPE (special)

This option defines a Modula-2 type.

```
M2TypeOption = '-' 'M2TYPE' '=' qualident '=' size ',' Attr
Attr = 'BOOL'   | 'CHAR' |
       'REAL'   | 'SET'  |
       'SIGNED' | 'UNSIGNED'
```

`qualident` is a Modula-2 type being defined, `size` is its size in bytes, and `Attr` is a family to which it belongs:

```
-M2TYPE = SYSTEM.INT16 = 2, SIGNED
```

See 5.3.1 for more information.

Appendix A

XDS

XDS is a family name for development systems featuring Modula-2 and Oberon-2 programming languages, available for Windows and Linux on the IBM PC and compatibles. XDS provides an uniform programming environment for all mentioned platforms and allows to design and implement truly portable software.

The XDS Modula-2 compiler implements ISO standard of Modula-2. The ISO standard library set is accessible for both Modula-2 and Oberon-2.

All XDS implementations share the same platform-independent front-end for both source languages. The output code can be either native code for the target platform (**Native XDS**) or text in the ANSI C language (**XDS-C**). ANSI C code generation enables you to cross-complile your programs for any platform.

XDS includes standard ISO and PIM libraries along with a set of utility libraries and interfaces to the host operating system API (Win32 or POSIX/X11) and the ANSI C library (XDS-C only).

Native XDS-x86 produces highly optimized Intel x86 code. It is available for Windows and Linux. Windows version comes with an IDE, debugger, prfiles, and other tools.

For more information about XDS, please visit our Web page at:

http://www.excelsior-usa.com/xds.html

Index

This page had been intentionally left blank.

Excelsior, LLC
6 Lavrenteva Ave.
Novosibirsk 630090 Russia
Tel: +7 (383) 330-5508
Fax: +1 (509) 271-5205
Email: info@excelsior-usa.com
Web: http://www.excelsior-usa.com