MIA 99-35: bidl - IDL to BETA Translation

Table of Contents

Copyright Notice

Mjłlner Informatics Report
MIA 99-35
August 1999

bidl - IDL to BETA Translation

Introduction

A tool - `bidl` - for translating IDL specifications into BETA programs is being constructed. It is based on an IDL grammar in structured BNF format. The tool is contructed using the Yggdrasil, the Metaprogramming System of the Mjłlner System.

Requirements

The tool translates IDL files, which are assumed to be legal IDL files in the Microsoft sence - i.e. that the IDL files are accepted by [MIDL]. Attemps to translate files that are not accepted by MIDL may yield unpredictable results.

To run `bidl`, the following requirements must be met:

1. On both UNIX and Windows, a `grep` utility must exist. If not already present on the system, a version may be obtained from GNU software.
2. On both UNIX and Windows, a C preprocessor must exist. On UNIX it is assumed that `gcc` exists, and on Windows it is assumed that `cl` exists (alternative C preprocessor may be specified using the BIDL_CPP environment variable).
3. On Windows the tool (which runs on console) has only been tested in the `4NT` shell. It is not known if it will work in other shells (such as `DOS prompt`).

`bidl` Synopsis

The tool is invoked as follows:

```
Usage: bidl [Options] <filename>

Options:
  -c
  --compact               Compact output - no comments, less newlines.
  -w
  --nowarnings            Don't display warnings.
  -e
  --errorwarnings         Make warnings become errors (displayed despite -q).
  -f
  --force                 Force generation of BETA file for argument IDL file.
  -F
  --forceall              Force generation of BETA files for all IDL files
                          (including imported ones)
  -v
  --verbose               Print verbose info on progress
  -q
  --quiet                 Be quiet - print minimal output, like files opened,
                          error messages (but not warnings)
  -m
  --mute                  Be absolutely quiet - print nothing
  -p
  --preserve              Preserve temporary files
  -D name
  --define name           Define preprocessor name
  -I path
  --include path          Add path to list of paths known by preprocessor

Debug options:
  -d
```

```
--debug             Enable debug of bidl
-t
--trace             Enable tracing of bidl
-T
--mps_trace         Enable tracing of mps
```

Environment Variables

The following environment variables are used by `bidl`:

`BETALIB`

Must point to the root of the BETA installation

`BIDL_CPP`

The command used to C-preprocess IDL files. Defaults are:
UNIX: `/usr/local/bin/gcc -E -P`
Windows: `cl -nologo -EP -P`

A typical bidl session

The following illustrates a typical session using `bidl`.

Compiling the IDL file(s)

In this example, we will compile the `Deep_Thought.idl` file used in phase 7 of Pilot application for COM Taxonomy.

The file is easily compiled using this command line:

```
command> bidl -f -q Deep_Thought.idl
Processing 'Deep_Thought.idl'...
      Deep_Thought.c
Processing '~beta/comlib/oaidl.idl'...
      oaidl.c
Processing '~beta/comlib/ObjIdl.Idl'...
      ObjIdl.c
Processing '~beta/comlib/Unknwn.Idl'...
      Unknwn.c
Processing '~beta/comlib/wTypes.Idl'...
      wTypes.c
Back from 's:\r4.2\comlib\wTypes.Idl'
Continuing processing of '~beta/comlib/Unknwn.Idl'...
```

```
Back from ’s:\r4.2\comlib\Unknwn.Idl’
Continuing processing of ’~beta/comlib/ObjIdl.Idl’...
Back from ’s:\r4.2\comlib\wTypes.Idl’
Continuing processing of ’~beta/comlib/ObjIdl.Idl’...
Back from ’s:\r4.2\comlib\ObjIdl.Idl’
Continuing processing of ’~beta/comlib/oaidl.idl’...
Back from ’s:\r4.2\comlib\oaidl.idl’
Continuing processing of ’Deep_Thought.idl’...
Processing ’~beta/comlib/ocidl.idl’...
      ocidl.c
Processing ’~beta/comlib/OleIdl.Idl’...
      OleIdl.c
Back from ’s:\r4.2\comlib\ObjIdl.Idl’
Continuing processing of ’~beta/comlib/OleIdl.Idl’...
Back from ’s:\r4.2\comlib\OleIdl.Idl’
Continuing processing of ’~beta/comlib/ocidl.idl’...
Processing ’~beta/comlib/OAIdl.Idl’...
      OAIdl.c
Back from ’s:\r4.2\comlib\ObjIdl.Idl’
Continuing processing of ’~beta/comlib/OAIdl.Idl’...
Back from ’s:\r4.2\comlib\OAIdl.Idl’
Continuing processing of ’~beta/comlib/ocidl.idl’...
Processing ’~beta/comlib/ServProv.Idl’...
      ServProv.c
Back from ’s:\r4.2\comlib\ObjIdl.Idl’
Continuing processing of ’~beta/comlib/ServProv.Idl’...
Back from ’s:\r4.2\comlib\ServProv.Idl’
Continuing processing of ’~beta/comlib/ocidl.idl’...
Processing ’~beta/comlib/UrlMon.Idl’...
      UrlMon.c
Back from ’s:\r4.2\comlib\ObjIdl.Idl’
Continuing processing of ’~beta/comlib/UrlMon.Idl’...
Back from ’s:\r4.2\comlib\OleIdl.Idl’
Continuing processing of ’~beta/comlib/UrlMon.Idl’...
Back from ’s:\r4.2\comlib\ServProv.Idl’
Continuing processing of ’~beta/comlib/UrlMon.Idl’...
Processing ’~beta/comlib/MsXml.Idl’...
      MsXml.c
Back from ’s:\r4.2\comlib\Unknwn.Idl’
Continuing processing of ’~beta/comlib/MsXml.Idl’...
Back from ’s:\r4.2\comlib\wTypes.Idl’
Continuing processing of ’~beta/comlib/MsXml.Idl’...
Back from ’s:\r4.2\comlib\ObjIdl.Idl’
Continuing processing of ’~beta/comlib/MsXml.Idl’...
Back from ’s:\r4.2\comlib\OAIdl.Idl’
Continuing processing of ’~beta/comlib/MsXml.Idl’...
Back from ’s:\r4.2\comlib\MsXml.Idl’
Continuing processing of ’~beta/comlib/UrlMon.Idl’...
Back from ’s:\r4.2\comlib\UrlMon.Idl’
Continuing processing of ’~beta/comlib/ocidl.idl’...
Back from ’s:\r4.2\comlib\ocidl.idl’
Continuing processing of ’Deep_Thought.idl’...
  Translated BETA file in: ’Deep_Thought.bet’
```

The output shown comes from the PC variant of bidl, which use `cl` to preprocess the IDL files. This is the reason for the lines like `MsXml.c`.

You will notice that bidl processes all the files imported by the `Deep_Thought.idl` file. The `-f` option given to `bidl` indicates that if `Deep_Thought.bet` already exist and is newer than `Deep_Thought.idl`, the BETA file should be generated anyway (it will not be default). The `-q` option tells `bidl` to be somewhat quiet (there’s also a `-m` aka. `--mute` switch to make `bidl` be completely silent, and a `-v` aka. `--verbose` option to make very verbose output (each type definition, interface name and method name is displayed).

The generated BETA file

The generated BETA file `Deep_Thought.bet` looks like this:

```
(* BETA interface generated from "Deep_Thought.idl" Wed May 13 22:58:34 1998 *)

ORIGIN '~beta/comlib/comtypes';
INCLUDE '~beta/comlib/OAIdl';
INCLUDE '~beta/comlib/OCIdl';
--LIB: attributes--

  ICalc_IID: (# exit '6FB6D32F-9E06-11D1-AE78-0020AF72F3D6' #);

  ICalc: IDispatch
    (* ICalc Interface *)
    (# <<SLOT ICalcLib: attributes>>;
       Add:<
         (* method Add *)
         (# result: @int32 (* HRESULT *);
            v1: @int32 (* long *);
            v2: @int32 (* long *);
            res: ^int32Holder (* *long *);
         enter (v1, v2, res[])
         do INNER;
         exit result
         #);
       Subtract:<
         (* method Subtract *)
         (# result: @int32 (* HRESULT *);
            v1: @int32 (* long *);
            v2: @int32 (* long *);
            res: ^int32Holder (* *long *);
         enter (v1, v2, res[])
         do INNER;
         exit result
         #);
       Multiply:<
         (* method Multiply *)
         (# result: @int32 (* HRESULT *);
            v1: @int32 (* long *);
            v2: @int32 (* long *);
            res: ^int32Holder (* *long *);
         enter (v1, v2, res[])
         do INNER;
         exit result
         #);
       Divide:<
         (* method Divide *)
         (# result: @int32 (* HRESULT *);
            v1: @int32 (* long *);
            v2: @int32 (* long *);
            res: ^int32Holder (* *long *);
         enter (v1, v2, res[])
         do INNER;
         exit result
         #);
       Modulus:<
         (* method Modulus *)
         (# result: @int32 (* HRESULT *);
            v1: @int32 (* long *);
            v2: @int32 (* long *);
```

```
           res: ^int32Holder (* *long *);
        enter (v1, v2, res[])
        do INNER;
        exit result
        #);
  #);

ICalc2_IID: (# exit '08407980-9e15-11d1-ae78-0020af72f3d6' #);

ICalc2: IDispatch
  (* ICalc2 Interface *)
  (# <<SLOT ICalc2Lib: attributes>>;
     Power:<
       (* method Power *)
       (# result: @int32 (* HRESULT *);
          v: @int32 (* long *);
          res: ^int32Holder (* *long *);
       enter (v, res[])
       do INNER;
       exit result
       #);
  #);

IHello_IID: (# exit 'a3985ef0-b4e6-11d1-aabb-006097636471' #);

IHello: IDispatch
  (* IHello Interface *)
  (# <<SLOT IHelloLib: attributes>>;
     World:<
       (* method World *)
       (# result: @int32 (* HRESULT *);
          res: ^BSTRHolder (* *BSTR *);
       enter res[]
       do INNER;
       exit result
       #);
  #);

IConv_IID: (# exit 'ed297100-9e0b-11d1-ae78-0020af72f3d6' #);

IConv: IDispatch
  (* IConv Interface *)
  (# <<SLOT IConvLib: attributes>>;
     Hex2Dec:<
       (* method Hex2Dec *)
       (# result: @int32 (* HRESULT *);
          v: @int32 (* BSTR *);
          res: ^int32Holder (* *long *);
       enter (v, res[])
       do INNER;
       exit result
       #);
     Dec2Hex:<
       (* method Dec2Hex *)
       (# result: @int32 (* HRESULT *);
          v: @int32 (* long *);
          res: ^BSTRHolder (* *BSTR *);
       enter (v, res[])
       do INNER;
       exit result
       #);
  #);
(* Calc Class *)
Calc_CLSID: (# exit '88bc6440-ab04-11d1-b55a-00600889e712' #);
(* Conv Class *)
Conv_CLSID: (# exit 'bd4067e0-ab04-11d1-b55a-00600889e712' #);
```

```
  (* Deep_Thought 1.0 Type Library *)
  DEEP_THOUGHTLib_LIBID: (# exit '6FB6D322-9E06-11D1-AE78-0020AF72F3D6' #);

  (* Reference Holders *)

  ICalcHolder: refHolder
    (# type:: ICalc #);
  ICalc2Holder: refHolder
    (# type:: ICalc2 #);
  IHelloHolder: refHolder
    (# type:: IHello #);
  IConvHolder: refHolder
    (# type:: IConv #);
```

For each interface specified in the IDL file, a BETA pattern is generated with *virtuals* corresponding to the member functions. Also the *interface-identifiers* (IID) of each interface, and *library-identifier* (LIBID) and *class-identifiers* (CLSID) from the type library are mapped by patterns.

You will notice that the parameter types in IDL, which are modelled by this BETA code, are mentioned in comments for clarity (there is a `-c` aka. `--compact` option for `bidl` which will cause such comments to be left out, as well as generating less line breaks in the BETA code).

Notice also, that the two IDL files which are `imported` by `Deep_Thought.idl` yields corresponding `INCLUDE`s in the BETA file. Notice that the two files are INCLUDEd from `~beta/comlib`, sice they could not be found in the local directory.

Using the generated BETA interface

The generated BETA interface can now be used in a client, simply by including it in the usual BETA manner. The following `client7.bet` is a very simple client, which obtains and uses the `IConv` interface from the component. Notice the use of BSTR:

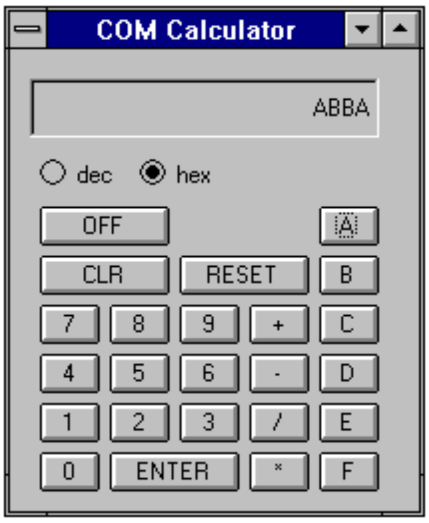
```
ORIGIN '~beta/comlib/comlib';
INCLUDE 'Deep_Thought';

--PROGRAM: descriptor--
(# CLSID_Conv: @CLSID;
   IID_IConv: @IID;
   pIConv: ^IConv;
   b: @BSTR;
   bh: @BSTRHolder;
   i: @int32Holder;
   hr: @HRESULT;

do 0->CoInitialize;

   IConv_IID -> IID_IConv;
   Conv_CLSID -> CLSID_Conv;

   'Create component and get interface IConv'->putLine;

   (CLSID_Conv[], NONE, CLSCTX_INPROC_SERVER, IID_IConv[])
     -> CoCreateInstance->pIConv[];

   'CoCreateInstance done'->putLine;

   (if pIConv[]<>NONE then
       'Succeeded creating component.'->putLine;
       '(17,bh[])->pIConv.dec2hex '->putText;
       (17,bh[])->pIConv.dec2hex->hr;
       (if hr.succeeded then
           '= '->putText; bh.getText->putLine;
        else
           'FAILED' -> putLine;
       if);
       '(\'ABBA\',i[])->pIConv.hex2dec '->putText;
       'ABBA' -> b.setText;
       (b, i[])->pIConv.hex2dec->hr;
       (if hr.succeeded then
           '= '->putText; i->putInt; newLine;
        else
           'FAILED' -> putLine;
       if);
       b.free;

       'Trying to release interface IConv'->putline;
       pIConv.Release;
       'Succeeded releasing interface IConv'->putline;
    else
       'Could not create component'->putline;
   if);
   'CoUninitialize'->putline;
   CoUninitialize;
   'CoUninitialize done'->putline;
#)
```

Assuming that the component has been registered, this BETA program can use one of the components made in Java or C++ (not Visual Basic, since we are not at `IDispatch` level).

The above program has been tested with success using both the Java server and the VC++/ATL server.

Furthermore a graphical client, which uses most of Deep_Thought has been constructed using the Mjłlner Interface Builder .

This has also been tested with success against the Java, and VC++/ATL servers.



Grammar

The grammar used by bidl to parse IDL files is constructed using the CORBA IDL specification (see [CORBASPEC], section 3.4), and adapting it for the extensions mentioned in [IDLEXT]. Finally we had to make several adaptions from the [MIDL] specification from Microsoft.

The current grammar can be seen in IDL Grammar (will open in a separate window).

It should be noticed that this is not a strict IDL grammar - it will probably accept much more IDL files (gramatically) than are accepted by [MIDL]. This is one of the reasons for requiring that the IDL files are accepted by MIDL before applying `bidl` on them.

Also it should be noted that the MIDL grammar has not been available in a complete version. So most of the grammar has been constructed by appliying it to concrete IDL files from Microsoft and changing the grammar until the files could be parsed. Currently all IDL files from the Microsoft Platform SDK are accepted by `bidl` (with a few exceptions, see limitations), but it is uncertain whether all other IDL files currently accepted by MIDL will be parsed by `bidl`.

Processing IDL Grammar

Currently there is a bug in the Mjłlner grammar tools, which means that << is *not* allowed in languages handled by the grammar tools. Since << is used for specifying constant expressions in IDL, a work-around has been needed. Thus when processing the IDL grammer, it should be done as follows (requires `perl`):

```
cd BETALIB/grammar/idl
perl do-idl-gram.perl
```

Reserved Words

A number of words are reserved to bidl, and are treated specially.

Grammatically reserved BETA words

In the generated BETA files, occurences of the following words (in any case) are substituted by the word with underscore ('_') prepended. The reason for this is that the words are reserved in the BETA grammar.

1. enter
2. exit
3. do
4. for
5. repeat
6. if
7. restart
8. inner
9. suspend
10. code
11. then
12. else
13. tos
14. this
15. or
16. xor
17. div
18. mod
19. and
20. not
21. NONE

Reserved BETA patterns

The following are names of predefined pattern from `betaenv`, which it is allowd, but very confusing to redefine. `bidl` renames identifiers with these names by prepending an underscore.

1. integer
2. shortInt
3. char
4. boolean
5. false
6. true
7. real
8. int8
9. int8u
10. int16
11. int16u
12. int32
13. int32u
14. int64
15. int64u

16. wchar
17. COM
18. Holder

Grammatically reserved IDL words

The following words (in this case) are reserved in the IDL grammar used by bidl, and thus cannot appear as e.g. a function name in an IDL file. These words are also reserved for e.g. MIDL.

1. __stdcall
2. _stdcall
3. any
4. attribute
5. boolean
6. case
7. char
8. coclass
9. const
10. context
11. cpp_quote
12. dispinterface
13. double
14. enum
15. exception
16. float
17. hyper
18. import
19. importlib
20. int
21. interface
22. library
23. long
24. methods
25. module
26. octet
27. properties
28. raises
29. sequence
30. short
31. signed
32. small
33. struct
34. switch
35. typedef
36. union
37. unsigned
38. void

The following words are currently also reserved in the IDL grammar used by bidl. They ought not to be...

1. default
2. readonly

3. uuid

Internal bidl words

Some words are used internally by bidl. E.g. the HRESULT exit parameter of a function is always called `result`. If an enter parameter is also called result, this will give an error. Not yet handled.

---

bidl - IDL to BETA Translation

Limitations

`bidl` currently have a few limitations of both grammatical and type-mapping kinds. In the cases where the problem is IDL constructs, that cannot be parsed or otherwise accepted by `bidl`, the construct

```
#ifndef bidl
...
#endif /* bidl */
```

can be used in the IDL source files, since `bidl` defines the symbol `bidl` in the preprocessing stage. These `#ifdefs` can be constructed so that they do not interfere with e.g. midl.

The current limitations are:

1. Cannot handle `static` declarations. E.g.

   ```
   static unsigned short* MMC_CALLBACK = ((unsigned short *)(-1));
   ```
   which is found in `mmc.idl`. Global data in IDL??? Can be fixed by #ifdef'ing out the declaration. Any possible references to the symbol will be treated as `int32` by `bidl`.
2. Cannot handle type casts. E.g.

   ```
   [switch_type(DWORD), switch_is((DWORD) tymed)]
   ```
   which is found in `ObjIdl.Idl`. Since there is the `switch_type` attribute, the type cast seems redundant anyway.

   Can be fixed by commenting out the type cast `(DWORD)`. It is not used by `bidl` anyway.
3. Cannot handle unicode string constants The form

   ```
   L"Hello, World!"
   ```
   currently cannot be parsed. The meta programming system does not support this kind of strings constants, and adding it to the grammar will make 'L' and 'l' reserved words.

   The fix is to remove the 'L' in the constants - the BETA compiler will handle it instead.

4. Cannot handle constants like `1L`
   Numerical constants with size indicator, like e.g. `1L` cannot be parsed. Fix by removing the `L` from the IDL.
5. AST overflows may occur
   There is a limit to the size of the IDL files that can be handled, see Handling AST overflows.
6. Inlined arrays in structs are translated incorrectly
   The following typedef

```
typedef struct tagfoo
{
  int  x;
  char y[4];
} foo;
```

   is incorrectly translated to

```
foo: struct_tagfoo(# #);
struct_tagfoo: DATA
  (# x: @int32 (* int *);
     y: ^charHolder (* char *);
  #);
```

   i.e. the y member is treated as a pointer instead of 4 inlined `chars`.

Handling AST overflows

Because of limitations in the current version of the Mjłlner Metaprogramming system, there is a limit on the size of the files that can be parsed. This can influence in two cases, when translating an IDL file to BETA:

1. The IDL file can be too large to be parsed by `bidl`
2. The generated BETA file can be too large to be parsed by the BETA compiler.

Let us take the second case first:

Generated BETA file too large for BETA compiler

If this happens, you should first try to use the `-c` (or `--compact`) option of bidl. This will generate code without comments, and this can in some cases make the BETA file sufficiently small.

If this does not solve the problem, currently you will have to split up the BETA file before compiling it. This is most easily done by splitting up the original IDL file and regenerating the BETA files. See the next section.

IDL file too large for bidl

If an ast-overflow error happens during parsing of an IDL file in bidl, you can try one thing first: If there are expressions, which comes from preprocessing (e.g. symbolic disp-id's, which turns out to be expressions instead of mere constants), you can try to leave out the include-files, which defines these expressions. This will make the symbolic names be used in the generated BETA files, but often the names are simply ignored in producing the BETA files. E.g. the values in `id(...)` in attributes of methods in a dispinterface is not used in the BETA files. Leaving out the expressions and only having an identifier will yield smaller ASTs.

If this is not enough/not possible, currently there is no other way of solving this problem than splitting up the IDL file.

The following scheme is the suggested one for solving the problem:

Assume that the following IDL file gives AST overflow:

foobar.idl

```
interface foo: IUnknown {
  ...
}
interface bar: IUnknown {
  ...
}
```

It is then suggested, that you split the file into the following 3 files:

foobar1.idl

```
interface foo: IUnknown {
  ...
}
```

foobar2.idl

```
import "foobar1.idl";
interface bar: IUnknown {
  ...
}
```

foobar.idl

```
import "foobar2.idl";
```

The generated `foobar.bet` will behave exactly as the one that would have resulted from converting the original `foobar.idl`, given that this was possible.

bidl - IDL to BETA Translation

Handling of Imports

If

```
import "foo.idl";
```

is met in an IDL file `bidl` opens `foo.idl` and find all the types defined in that file, for using it in the current file. Also in the BETA file generated for the importing file,

```
INCLUDE ’foo’;
```

is generated.

`bidl` has a limited search algorithm for imported files (which are nomally not specified by full path): If bidl does not find the imported file in the directory where the importing file resides, `bidl` searches for the imported file in `~beta/comlib`. If the file is not found there, the import is ignored.

NOTICE: On UNIX, the case of the imported files is significant. And unfortunately the IDL files coming from the Windows world often use another case for importing than then file name actually is (since it does not matter to the Windows file system).

bidl - IDL to BETA Translation

Holder classes

Based on the idea used in the IDL to Java mapping [IDL2JAVA], we are generating so-called *holder classes*, that the compiler treats specially. See Compiler Support for Holder objects for details.

IDL to BETA Type Mapping

The following sections describe the mappings of various IDL types to BETA.

Typedef aliases

Simple alias-typedefs are handled by adding the resolved names to the internal tables of bidl. Usages of the aliased names are then determined from this table.

If the alias does not introduce pointer levels, i.e.

```
  typedef XXX YYY;
```
as opposed to

```
  typedef XXX *YYY;
```
where XXX is a struct or union, an alias pattern

```
  YYY: XXX(# #);
```
is also generated.

Enums

enums are mapped to corresponding BETA constants:

Enum Type	BETA Type	Pointer to...	Pointer to pointer to...

`typedef enum tagfoo {`
`  foo1,`
`  foo2,`
`  ...`
`} foo;`
	`foo: (* enum tagfoo *)IntegerObject;`
	`foo1: (* foo *)(# exit 0 #);`
	`foo2: (* foo *)(# exit 1 #);`
	`...`
		`^int32Holder`	*Error*

As can be seen, I would have liked to map the enum type to an actual BETA type, but currently only maps to constants. This is because IntegerObjects cannot be used as formal parameters for COM virtual calls.

In case initializers are used, these values are mapped into the BETA constants, and in case *some* initializers are used, but some enums members are without initializers, the expressions are *not* calculated by `bidl`, but left the the BETA compiler to handle:

Enum Type	BETA Type	Pointer to...	Pointer to pointer to...

`typedef enum tagbar {`
`  bar1 = 20,`
`  bar2,`
`  bar3 = foo3,`
`  bar4,`
`  bar5,`
`  ...`
`} bar;`
	`bar: (* enum tagbar *)IntegerObject;`
	`bar1: (* bar *)(# exit 20 #);`
	`bar2: (* bar *)(# exit 20+1 #);`
	`bar3: (* bar *)(# exit foo3 #);`
	`bar4: (* bar *)(# exit bar3+1 #);`
	`bar5: (* bar *)(# exit bar4+1 #);`
	`...`
		`^int32Holder`	*Error*

Structs

Structs are mapped to specializations of Pattern DATA, where each member is mapped according to the general Mapping of Types:

IDL Type	BETA Type	Pointer to...	Pointer to pointer to...
`typedef struct tagfoo {` `int x;` `double y;` `char z;` `} foo;`	`foo: struct_tagfoo(# #);` `struct_tagfoo: DATA` `(# x: @int32 (* int *);` `y: @real (* double *);` `z: @char (* char *);` `#);`	^foo	^fooHolder

You will notice that *two* patterns are generated for this struct definition. This is because in IDL `struct tagfoo` and `foo` are synonomous after the above typedef, e.g. another place there may be

```
typedef struct tagfoo foo_alias;
```
which is then in BETA mapped to

```
foo_alias: struct_tagfoo(# #);
```
There are a number of special cases in struct uses.

Structs typedefs with no tag

In case a struct id defined without a tag, the type name alone will be used to identify the struct, i.e. the `struct_`XXX type/pattern will be left out:

IDL Type	BETA Type	Pointer to...	Pointer to pointer to...
`typedef struct {` `int x;` `double y;` `char z;` `} foo;`	`foo: DATA` `(# x: @int32 (* int *);` `y: @real (* double *);` `z: @char (* char *);` `#);`	^foo	^fooHolder

Forward typedefs with structs

It is allowed to do a forward declaration of a struct type without actually specifying it, as in

```
typedef struct tagfoo foo;
```
provided that there is later a definition of `struct tagfoo` (which is normally a struct typedefs with no name). This is handled by bidl by making `struct_tagfoo` a legal, but yet undefined, type. When the type is later defined, the pattern for it will be generated (and the type will be reported as a `redefinition`, when using `--verbose`.

Struct typedefs with no name

A struct typedef with no name is normally provided after a corresponding struct forward typedef.

An example could be

```
struct tagfoo {
  int x;
  double y;
  char z;
};
```

This will give rise to the generation of the `struct_tagfoo` pattern as in the normal case.

Struct typedefs with no body and no name

A struct with no tag and no name can occur when the struct is used as part of another struct or union, as in

```
typedef struct tagfoo {
  struct {
    int x;
    int y;
  } x_y;
  int z;
} foo;
```

In this case there will be generated an anonymous inlined object, prefixed with `DATA`:

```
foo: struct_tagfoo(# #);
struct_tagfoo: DATA
  x_y: @DATA
    (# x: @int32 (* int *);
       y: @int32 (* int *);
    #);
  z: @int32 (* int *);
#);
```

Unions

Unions cannot be uniquely mapped to corresponding BETA types. If a union is met, an empty pattern (prefixed with `holder`) is generated, and a warning is issued to the screen.

But the union name is introduced into the internal `bidl` type tables, so parameters qualified with the union types are treated correctly.

The user needs to fill in the union pattern herself. For a inspiration the `VARIANT` pattern in `~beta/comlib/comtypes.bet` can be consulted. This is one possible implementation of a union in BETA.

Except for the missing body, union types are treated exactly like struct types by `bidl`.

Basic Types

IDL Type	BETA Type	Pointer to... [6]	Pointer to pointer to...

int	int32	^int32Holder / @int32	*Error*
long	int32	^int32Holder / @int32	*Error*
short	int16	^int16Holder / @int32	*Error*
small	int8	^int8Holder / @int32	*Error*
hyper	int32 [1]	^int32Holder / @int32	*Error*
signed	int32	^int32Holder / @int32	*Error*
unsigned	int32u	^int32uHolder / @int32	*Error*
signed int	int32	^int32Holder / @int32	*Error*
unsigned int	int32u	^int32uHolder / @int32	*Error*
signed long	int32	^int32Holder / @int32	*Error*
unsigned long	int32u	^int32uHolder / @int32	*Error*
signed long int	int32	^int32Holder / @int32	*Error*
unsigned long int	int32u	^int32uHolder / @int32	*Error*
signed short	int16	^int16Holder / @int32	*Error*
unsigned short	int16u	^int16uHolder / @int32	*Error*
signed short int	int16	^int16Holder / @int32	*Error*
unsigned short int	int16u	^int16uHolder / @int32	*Error*
char	char	^charHolder / @int32	*Error*
wchar	wchar	^wcharHolder / @int32	*Error*
float	real [2]	^realHolder / @int32	*Error*
double	real	^realHolder / @int32	*Error*
boolean	boolean	^booleanHolder / @int32	*Error*
sequence	int32 [3]	^int32Holder / @int32	*Error*
octet	int8u [4]	^int8uHolder / @int32	*Error*
any	int32 [5]	^int32Holder / @int32	*Error*

[1] `hyper` is a 56 bit IDL type, which is mapped to 32 bit in BETA, i.e. a potential lossy mapping.
[2] `float` is a 32 bit IDL floating point type, which is mapped to 64 bit floating real in BETA, i.e. a potential lossy mapping.
[3] `sequence` is an OMG CORBA IDL type defining a bounded one-dimensional array. It is not used in Microsoft IDL. See [CORBA_SPEC] section 15.2.7 (page 386) for a mapping between CORBA sequences and MIDL.
[4] The `octet` type is an 8-bit OMG CORBA IDL quantity that is guaranteed not to undergo any conversion when transmitted by the communication system. It does not seem to be used by MIDL.

[5] The `any` type permits the specification of values that can express any OMG CORBA IDL type. It does not seem to be used by MIDL.
[6] If a pointer to a basic type is an `[out]` parameter, it will be translated to a Holder to the appropriate type, otherwise to an `int32`.

String Types

In the following, the notation `[string] char*` means a `char*` type, which is `[string]` attributed in the IDL attributes.

IDL Type	BETA Type	Pointer to...	Pointer to pointer to...
[string] char*	[0]@char	[0]@char [7]	^textHolder
[string] wchar*	wtext	^wtext	^wtextHolder
BSTR	BSTR	^BSTR	^BSTRHolder

[7] It would be more natural to use `^text` here, but the compiler currently does not support this as parameter type to COM virtuals.

COM Interfaces

These are handled gracefully by the compiler - notice that pointer-to-pointer-to-interface is using a `refHolder` instead of a simple `holder`.

IDL Type	BETA Type	Pointer to...	Pointer to pointer to...
Ifoo	Ifoo	^Ifoo	^IfooRefHolder
void	IUnknown	^IUnknown	^IUnknownHolder

Notice the usage of `^IUnknown` for `void*` and `^IUnknownHolder` for `void**` types - this is currently done because e.g. `QueryInterface` has an out parameter qualified with `void**` in IDL, but which is surely at least an IUnknown pointer. To allow for typecheck in BETA, this mapping of void pointers is thus introduced.

bidl - IDL to BETA Translation

References

[CORBASPEC]

CORBA Specification from Object Management Group.

[IDLEXT]

COM IDL Extensions as specified in the CORBA Specification.

[IDL2JAVA]

IDLto JAVA mapping from the OMG IDL-Java mapping submission (orbos/97-02-01).

[MIDL]

MIDL Language Reference from the Microsoft Developer Network Online Library.

[MIDLTYPES]

MIDL Base Types from the Microsoft Developer Network Online Library.

[OLESTR]

Strings The OLE Way from the Microsoft Developer Network Online Library.

[DCEIDL]

DCE Interface definition Language from the OSF DCE Application Development Guide - Core Components.