FILE NAME: BDDL\_guidelines.txt

AUTHOR: Martin O'Brien

COPYRIGHT (c) 2013 GLASSWALL SOLUTIONS LIMITED

------------------------------------------------------------------------

BDDL GUIDELINES

Table of Contents

1. Introduction

2. Combining Fields

3. Include Statements

4. Naming Bitfields

5. AcceptRules

5.1 Placement

5.2 Enumerations

5.3 Arrays of Structures

5.4 File Validation

6. Pointer Metadata

7. Version Control

8. optional restrictions

9. char and wchar

10. Allowempty

11. Structure Names

11.1 BDDL File Names

11.2 MIME Types

12. Precedence and Associativity

13. Glasswall Additions

13.1 Calling legacy or external cameras

13.2 Validating Text fields as integers

13.3 Guiding code generator

13.3.1 Linking Target Objects

13.3.2 Marking fields as Tokens

13.3.3 Marking fields as Whitespace

13.3.4 Inserting Handwritten Code

13.3.5 Changing Filename

14. Recommendations

14.1 Extra Annotations

1. Introduction

These guidelines are designed to facilitate a standardised way of

writing BDDL files. The guidelines are designed to provide code

that is clear, robust, efficient, maintainable and concise.

These guidelines may be referred to in code reviews.

2. Combining Fields

When combining fields, name the field using all the names, for

example, [MS-DOC] 2.9.207, PnFkpPapx, do this:

uint32 pn\_unused

3. Include Statements

For the include statement, use chevrons (<>) for Miraplacid library

files, and double quotes ("")for Glasswall Solution files:

include <strings.bdd>

include "office\_bddl/common/sttb.bdd"

include "office\_bddl/MS-DOC/WordDocument\_Stream.bdd"

4. Naming Bitfields

When combining bitfields, prefix the name with 'f' (for 'flag'),

e.g., the flags stored in the first byte of the Rfs structure

([MS-DOS] Section 2.9.227, page 441) would be named fABCDEFG

The uppercase sequence of letters of the alphabet distuinguishes

them from true variable names in the spec, such as fExtend.

If you're combining other fields too, name them in sequence, e.g.,

the first 8 bits in DopBase can be named

fABCD\_fpc\_fE

So each set of one of more flags is preceded by 'f'.

5. AcceptRules

5.1 Placement

Place AcceptRules as early as possible in the structure definitions

in order to speed up code generation. That is, they do not have to

be the last items in the structure definition.

5.2 Enumerations

AcceptRules reject the current structure if the condition is

false, so use an AcceptRule only when you want processing of the

structure to stop.

Use an AcceptRule for the types you specify in an enumeration,

for example:

struct Type1 {

...

AcceptRule ( condition1 );

}

struct Type2 {

...

AcceptRule ( condition2 );

}

struct Type3 {

...

AcceptRule ( condition3 );

}

(Type1 | Type2 | Type3 ) types;

Make sure condition1, condition2, and condition3 are mutually

exclusive. Type3 does not need an AcceptRule, if you want it

to be the default if the other conditions fail.

5.3 Arrays of Structures

An AcceptRule can be used in a structure that is the type of an

array. As soon as the structure is not accepted, the array is

finished and the parser/validator will proceed with the next

item in the data definition.

5.3 File Validation

Use an AcceptRule if you want processing of the file to stop,

for example, if a "magic" number identifying the file type is

not present.

6. Pointer Metadata

Suffix pointer metadata directory names with '\_Ptr', e.g.,

directory SttbfAssoc\_Ptr { ... }

7. Version Control

Consider checking in your BDDL files at appropriate intervals in order

to preserve your work, especially before a weekend. Checking in

ensures the changes are stored on the server. Files edited but not

checked in are stored only in your workspace, which is mapped to your

local drive. If the drive fails, your edits are lost.

8. optional restrictions

The 'optional' keyword specifies that if the current data item is

not present, it is not an error.

The 'restrictions' keyword allows an element to be validated and an

error reported if it is false.

It is important to know that When used with an array, the restrictions

rule is applied to each element, and when it fails, the array is ended.

This allows for variable-length arrays.

If 'optional' and 'restrictions' are used together, then an error is

not reported if the 'restrictions' rule is false.

9. char and wchar

'Char' is 8-bit and 'wchar' is 16-bit Unicode. The BDDL

documentation refers to char as 'ANSI' but it is a largely

meaningless description (ANSI is the acronym of the American

National Standards Institute). It usually refers to Windows Code

Pages.

Where the Microsoft specification refers to characters or wide

characters, then use the 'char' and 'wchar' types as required.

10. Allowempty

When an array size can be zero, then 'allowempty' must be used.

11. Structure Names

11.1 BDDL File Names

The name of the BDDL file should have the same case as the

structure, as specified in the specification, with file type

'.bdd'. For example:

STSH.bdd

SttbfRMark.bdd

11.2 MIME Type

The MIME type specified as the value of the 'mime\_type'

annotation key should retain the case as specified in the

specfication, prefixed by 'ms/', for example:

mime\_type="ms/STSH"

mime\_type="ms/SttbfRMark"

12. Precedence and Associativity

The table below summarises the rules for precedence and

associativity of all operators. Operators on the same line have the

same precedence; rows are in order of decreasing precedence, so, for

example, \*, /, and % all have the same precedence, which is higher

than that of binary + and -. The "operator" () refers to function

call. The operators -> and . refer to the access of structures.

----------------------------------+---------------

Operators | Associativity

----------------------------------+---------------

() [] -> . | left to right

----------------------------------+---------------

! ~ ++ -- + - \* (type) sizeof | right to left

----------------------------------+---------------

\* / % | left to right

----------------------------------+---------------

+ - | left to right

----------------------------------+---------------

<< >> | left to right

----------------------------------+---------------

< <= > >= | left to right

----------------------------------+---------------

== != | left to right

----------------------------------+---------------

& | left to right

----------------------------------+---------------

^ | left to right

----------------------------------+---------------

| | left to right

----------------------------------+---------------

&& | left to right

----------------------------------+---------------

|| | left to right

----------------------------------+---------------

?: | right to left

----------------------------------+---------------

= += -= \*= /= %= &= ^= |= <<= >>= | right to left

----------------------------------+---------------

, | left to right

----------------------------------+---------------

13. Glasswall Additions

In order to add some functions to BDDL that will be used by BDD to

C++ or DVL to C++ conversion programs, the annotation mechanism in

BDDL is used to add extra notes for those programs that will be

ignored by Miraplacid.

13.1 Calling legacy or external cameras

[camera=<camera name>]

uint8 [\*] data;

We prefix the data item with the 'camera annotation' where we have

loaded a block of bytes that are known to be the raw data of a file

type for which we have a camera, e.g., JPEG.

So in this particular case here is the structure from DggInfo that

wraps round a JPEG image with the annotation added.

struct OfficeArtBlipJPEG{

OfficeArtRecordHeader rh;

AcceptRule rh.recType == 0xF01D || rh.recType == 0xF02A;

uint8[16] rgbUid1;

uint8[16] rgbUid2 optional restrictions{((parent.rh.frecVer\_recInstance) & (0xFFF0) >> (4)) == 0x46B

|| ((parent.rh.frecVer\_recInstance) & (0xFFF0) >> (4)) == 0x6E3};

uint8 tag;

[camera="jpegCamera"]

uint8[rh.recLen - sizeof(rgbUid1) - sizeof(rgbUid2) - sizeof(tag)] BLIPFileData allowempty;

}

13.2 Validating Text fields as integers

13.2.1 BDD:

[fieldtype=<type of field>]

char [\*] my\_value;

13.2.2 DVL:

my\_value := RANGE(current, <x>, <y>); /\*or similar rule\*/

Where <type of field> is:

"integer" for interpreting text like "2344" or "-9766" as an

integer number in DVL validation phase.

"real" for interpreting text like "33.45" or "-322.83" as a real

(floating point) number in DVL validation phase.

This tells DVL\_Checker to build code that reads the text of the

my\_value field and turn it into a real value.

If you don't put the fieldtype annotation, then because char[\*] is

turned into an object we would be comparing against the address in

memory of the object and not its interpreted value.

13.3 Guiding code generator

13.3.1 Linking Target Objects

[targetObject=<fully specified path to object>]

Where <fully specified path to object> could be things like:

root.tableStream

root.dataStream

A way of helping the BDDL to C++ converter with preparing to connect

directories to objects.

When using a 'directory' in BDDL, Miraplacid works out which object

to connect it to, by looking at the address range of the location

where the target object ends up. To remove this job from the

runtime stage of the generated code, we can use an annotation to the

BDDL that tells the generated code where the connect the completed

object.

For example:

BDD:

[targetObject=”root.tablestream”]

directory D\_NAME {

type \*\*\*\*

PointerRule \*\*\*\*\*\*

AcceptCondition \*\*\*\*\*

13.3.2 Marking fields as Tokens

[token="<acceptRule/restriction>"]

Where <acceptRule/restriction> is used to read characters from

the current location until the expression fails.

This tells the BDDL to C++ convertor that the following string is

a Token (like a element name in XML or control word in RTF).

The field will be replaced by an enumerated lookup for the camera

"e<CameraName>\_Token" and the string will be looked up in a

dictionary from all the ...

field == "string"

... style AcceptRules and Restrictions found in the BDDL for this

camera. These string compares will be replaced by a Enum compare

in the code.

For example:

BDD:

WhiteSpace ws2 optional;

[token="isletterordigit()"]

Word name implicit;

Attribute[\*] attributes optional;

will mean 'name' will not be treated as a 'Word' (which contained

the string object) but instead will be treated as a enumerated

Token.

13.3.3 Marking fields as Whitespace

[whitespace="true"]

This tells the BDDL to C++ convertor that the following

characters should be whitespace. Therefore it only needs to

record a single three-state scalar value that indicates whether:

o No Whitespace Found

o One or more Space or Tab Found

o One or more Carriage Return or Linefeed Found

For example:

BDD:

[whitespace="true"]

WhiteSpace ws2 optional;

Word name implicit;

will mean 'ws2' is stored as a single scalar instead of a vector

of characters. A single space or CRLF or nothing will be output

in the regenerated file.

13.3.4 Inserting Handwritten Code

[insertcode\_xxxx="<your code here>"] or [replacetest\_xxxx="<your code here>"]

This tells the BDDL to C++ convertor to insert the literal code

in front of the current BDD construct in the situation defined by

XXXX.

Note: Multiline code sections between the quotes must not contain quotes, instead use '&quot;' instead

The variations are:

a) [insertcode\_asis\_onread="<your code here>"] (implemented 25-3-2015)

This tells the BDDL to C++ convertor to insert the literal

code in front of the read of the current item, or in front of

an acceptrule on a read pass.

b) [replacecode\_asis\_onread="<your code here>"] (implemented 25-3-2015)

This tells the BDDL to C++ convertor to replace the read of the current item,

or in front of an acceptrule on a read pass. With the literal

code.

c) [insertcode\_asis\_onwrite="<your code here>"]

This tells the BDDL to C++ convertor to insert the literal

code in front of the write of the current item, or at end of

writing all items if placed in front of the acceptrule in the

BDD.

d) [insertcode\_interpreted\_onread="<your code here>"] (implemented 25-3-2015)

This tells the BDDL to C++ convertor to insert the code in

front of the read of the current item, or infront of an

AcceptRule on a read pass. But it will run the BDD to C++

interpretation on the inserted code which will replace

'current' with the item name or replace 'current' with the

structure (class) name id applied to an AcceptRule.

e) [replacecode\_interpreted\_onread="<your code here>"] (implemented 25-3-2015)

This tells the BDDL to C++ convertor to replace generated code for

the read of the current item, or AcceptRule on a read pass with the

literal code but it will run the BDD to C++

interpretation on the inserted code which will replace

'current' with the item name or replace 'current' with the

structure (class) name id applied to an AcceptRule.

f) [insertcode\_interpreted\_onwrite="<your code here>"]

This tells the BDDL to C++ convertor to insert the code in

front of the write of the current item, or at end of writing

all items if placed in front of the acceptrule in the BDD.

But it will run the BDD to C++ interpretation on the inserted

code which will replace 'current' with the item name or

replace 'current' with the structure (class) name if applied

to an AcceptRule.

g) [replacetest\_asis\_onread="<your code here>"]

This tells the BDDL to C++ convertor to insert the literal

code INSTEAD OF the acceptrule code on a read pass.

h) [replacetest\_interpreted\_onread="<your code here>"]

This tells the BDDL to C++ convertor to insert the literal

code INSTEAD OF the acceptrule code on a read pass, but it

will run the BDD to C++ interpretation on the inserted code

which will replace 'current' with the structure (class) name.

For example:

BDD:

[insertcode\_asis\_onread="skipcomments();"]

Word name ;

[replacetest\_asis\_onread="name.str == &quot;STRUCT1&quot;"]

AcceptRule name == "STRUCT1";

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

NOTE: The use of '&quot;' for double quotes as the '\"' escape sequence did not work in the C# code for some reason.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

13.3.5 Changing Filename

We can use the

[fileprefix="cameraname"]

annotation in front of any structure in a file to insert the cameraname in front of the c++ and h filenames

for the files produced from the BDD file.

e.g. XML\_common.bdd would normally produce XML\_common.cpp and XML\_common.h, but this annotation will

create {cameraname}\_XML\_common.cpp and {cameraname}\_XML\_common.h.

This allows common bdd files to be used in different cameras and the code will be regenerated in each namespace with different filenames

14. Recommendations

14.1 Extra Annotations

It would be good to have

[spec=”MS-DOC x.y.z”]

in front of structures/dictionaries to help map BDDL to Specification.

Also, a way of helping the BDDL to C++ converter with preparing to connect directories to objects

[targetObject=”root.tablestream”]

directory D\_NAME {

type \*\*\*\*

PointerRule \*\*\*\*\*\*

AcceptCondition \*\*\*\*\*

}

This is because the Miraplacid SDK does it by finding the object containing the dataheap dynamically from the PointerRule offset,

which would be very messy a slow a mechanism to replicate and run in the DLL