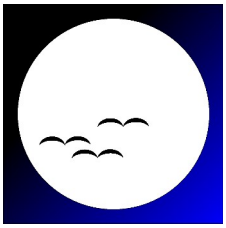

QSpice C-Block Code Generator (QCodeGen)

Developer Notes

Document Revisions:

2024.01.27 – Initial Version.

Caveats & Cautions

First and foremost: Do not assume that anything I say is true until you test it yourself. In particular, I have no insider information about QSpice. I've not seen the source code, don't pretend to understand all of the intricacies of the simulator, or, well, anything. Trust me at your own risk.

If I'm wrong, please let me know. If you think this is valuable information, let me know. (As Dave Plummer, ex-Microsoft developer and YouTube celebrity (Dave's Garage) says, "I'm just in this for the likes and subs.")

Note: This is the third document in a clearly unplanned series:

- ***The first C-Block Basics document covered how the QSpice DLL interface works.***
- ***The second C-Block Basics document demonstrated techniques to manage multiple schematic component instances, multi-step simulations, and shared resources.***

This document builds on the prior information so consider them "required reading." See my GitHub repository here for the prior documents and code samples for this one.

Overview

This document provides notes about parsing the QSpice schematic file format. Only bits relevant for parsing C-Block data are included. That is, not all record formats are described and not all elements within a particular record type are known and described.

Since this is an exercise in reverse-engineering, the information may be incomplete or inaccurate. Rely on it at your own peril.

Sample File Data

Here is an excerpt from a schematic:

```
ÿØÿÛ«schematic
 «component (-600,1000) 0 0
   «symbol
      «type: Ø(.DLL)»
      «description: Test CBlock»
      «shorted pins: false»
      «rect (-300,400) (2100,-2900) 0 0 0 0x4000000 0x4000000 -1 1 -1»
      «text (900,300) 1 12 0 0x1000000 -1 -1 "X1"»
      «text (900,200) 0.681 13 0 0x1000000 -1 -1 "Test_X1"»
      «text (900,0) 0.681 13 0 0x1000000 -1 -1 "char* Attr1="abc""»
...
      «pin (-300,300) (0,0) 1 7 145 0x0 -1 "In_d"»
      «pin (-300,100) (0,0) 1 7 17 0x0 -1 "In_b"»
      «pin (-300,-100) (0,0) 1 7 97 0x0 -1 "In_i"»
      «pin (-300,-300) (0,0) 1 7 161 0x0 -1 "In_i64"»
...
      «pin (2100,-500) (0,0) 1 11 34 0x0 -1 "Out_c"»
      «pin (2100,-700) (0,0) 1 11 50 0x0 -1 "Out_uc"»
      «pin (-300,-1100) (0,0) 1 7 81 0x0 -1 "In_us"»
...
    »
  »
  «component (-2800,-1500) 0 0
    «symbol V
      «type: V»
      «description: Independent Voltage Source»
      «shorted pins: false»
      «line (0,-130) (0,-200) 0 0 0x1000000 -1 -1»
      «line (0,200) (0,130) 0 0 0x1000000 -1 -1»
      «rect (-25,77) (25,73) 0 0 0 0x1000000 0x3000000 -1 0 -1»
      «rect (-2,50) (2,100) 0 0 0 0x1000000 0x3000000 -1 0 -1»
      «rect (-25,-73) (25,-77) 0 0 0 0x1000000 0x3000000 -1 0 -1»
      «ellipse (-130,130) (130,-130) 0 0 0 0x1000000 0x1000000 -1 -1»
      «text (100,150) 1 7 0 0x1000000 -1 -1 "V1"»
      «text (100,-150) 1 7 0 0x1000000 -1 -1 "10"»
      «pin (0,200) (0,0) 1 0 0 0x0 -1 "+"»
      «pin (0,-200) (0,0) 1 0 0 0x0 -1 "-"»
    »
  »
  «net (300,-2300) 1 13 0 "GND"»
  «net (-1100,1300) 1 11 0 "Vin_d"»
  «net (-1100,1100) 1 11 0 "Vin_b"»
...
  «wire (300,-2300) (300,-1900) "GND"»
  «wire (-1100,1300) (-900,1300) "Vin_d"»
  «wire (-1100,1100) (-900,1100) "Vin_b"»
...
»
```

File Structure & Binary Values

The first four bytes are a file prefix:

```
const unsigned char *idBytes = "\xff\xd8\xff\xdb";
```

This is followed by hierarchical data in human-readable text format. Each hierarchy element begins with “«” (0xab) and ends with “»” (0xbb).

The hierachy opens with “«schematic” and ends with the final “»”. The example shows two “component blocks” which contain “symbol blocks.”

The symbol block contains optional symbol text, “V” in the second symbol block. This is omitted for DLL component symbol blocks.

The symbol blocks contain detail items, one on each line:

«type: ...» – Type identifier = “Ø(.DLL)” for DLL components. (The phi character is 0xd8.)
«description: ...» – Description (***optional***, set in Symbol Properties)
«shorted pins: ...» – Shorted element indicator.

These are followed by various detail records – line, rect, ellipse, text, pin, and so on. The types appear to be in a fixed order grouped by type.

For DLL components, we can skip anything other than text and pin records. These are detailed below.

The remaining example records are nets and wires which we don’t need for this project.

Text Records

For DLLs, there will be at least two text records. The first is the symbol label (“X1”) followed by the DLL/component name (“Test_X1”).

Any remaining text records are component attributes. The relevant elements seem to be:

```
«text (900,0) 0.681 13 0 0x1000000 -1 -1 "char* Attr1="abc""»

(900,0) – coordinates?, signed integer (may contain “-”)
0.681 13 – unknown, decimal values (no decimal if integral)
0 – Comment flag:  0 = normal attribute; -1 = comment text?
0x1000000 – unknown, hex value
-1 -1 – unknown
"char* Attr1="abc"" – Attribute text (see below)
```

As mentioned, the first two text records contain the symbol name and DLL name in the attribute text.

For actual attribute parameters, the text contains the attribute data type, the attribute name, and the attribute value. This, of course, varies for different data types. The attribute text field is quoted and, as shown, may contain quoted text.

The attribute text must be parsed carefully to extract the data type, name, and value. For example, above we have “char* Attr1=”abc””. “char *Attr1=”abc”” would be equivalent and both forms are considered valid by QSpice.

Note this well: QSpice parses and validates attributes somewhat inflexibly. If it encounters an attribute that it “doesn’t like,” it ***silently skips*** *that record* ***and*** *any remaining attribute records when generating the C-Block template.*

Examples of invalid attributes include:

- “unsigned int Attr=1” – doesn’t like “unsigned”.
- “short Attr=1” – doesn’t like “short”.
- “int Attr = 1” – doesn’t like spaces around “=”.
- “int Attr” – must have “=” and value.
- “long int Attr=1” – doesn’t like “long”.
- “char*Attr=”string” – requires space before and/or after “*”.

And… “float Attr” is promoted to type double when the code is generated.

In summary, here are the patterns that I think are allowed:

- “bool Attr=[bval]”
- “char Attr=[cval]”
- “int Attr=[ival]”
- “float Attr=[fval]” (promoted to “double Attr=[fval]”)
- “double Attr=[fval]”
- “char* Attr=[sval]”
- “char *Attr=[sval]”
- “char * Attr=[sval]”

Where

- “[bval]” is 0 or 1 (possibly any numeric value) or a .param statement name. Note that “true” or “false” will likely be interpreted as undefined .param statement names (unless, of course, “true” and false are defined with .param).
- “[cval]” is a (properly) single-quoted character, an integer, or a .param statement name.
- “[ival]” is an integer or a .param statement name.
- “[fval]” is float or double value or a .param statement name.
- “[sval]” is a (properly) double-quoted string value or an unquoted .param statement name.

I’ve not tested all of the possibilities. Regardless, it’s advisable to parse very carefully and warn the user of any possible problems.

Another oddity: Component attributes can be marked as comments. However, the flag is ignored – an attribute marked as a comment ***is*** processed for code generation. And, if it fails to parse properly, subsequent text records are skipped. Note that the attribute ***is*** treated as a comment when generating the netlist, i.e., it’s not included in the netlist. We should warn about this if a commented attribute is present.

Finally, all attribute parameters are component inputs (you can’t output through an attribute) and appear in the C-Block template in the order they appear in the component/symbol block.

Pin Records

Pin items are less difficult to decode. There’s a single byte for data type and port direction and the pin name.

```
«pin (2100,-500) (0,0) 1 11 34 0x0 -1 "Out_c"»
```

In the above, “34” is the type/direction value and “Out_c” is the pin name. Note that the type/direction value may be out of range for an integer byte for unknown reasons. Just mask off any extra high bits with 0xff.

Port Direction (Lsb 4 bits):

Decimal Value	Binary Value	Port Direction
1	0001	Input
2	0010	Output
3	0011	Component GND

Port Data Type (Msb 4 bits):

Decimal Value	Binary Value	Port Data Type	UDATA Definition
1	0001	bool	data[].b
2	0010	char	data[].c
3	0011	unsigned char	data[].uc
4	0100	short	data[].s
5	0101	unsigned short	data[].us
6	0110	int	data[].i
7	0111	unsigned int	data[].ui
8	1000	float	data[].f
9	1001	double	data[].d
10	1010	long long int	data[].i64
11	1011	unsigned long long int	data[].ui64
12	1100	bit vector*	(depends on bit width)

*Bit vector will not be supported initially.

The text portion of the record is the pin name. If the pin label has an overbar, each character of the name text is preceded with "¬" (0xac) and the generated code replaces it with an underscore in the generated variable name. For example, a pin named “Pin1” will appear in the schematic file as “¬P¬i¬n¬1” and the QSpice-generated code variable name would be “_P_i_n_1”. (Not a fan of this so may add an option to not add the underscores.)

For code generation, the order of pins is retained but inputs and outputs are generated separately.

Code Generation

When generating code, the order data passed in the union uData *data parameter is:

1. First, all input pins.
   Input pins values are copied to the stack. Changing these values does not change the actual input variable memory values.
2. Next, all attribute inputs.
   Non-string attributes are copied to the stack. String attributes are directly accessible via reference but const-qualified to prevent modification.
3. Finally, all output pins.
   Output pins are defined as reference variables.

Within each of the above groups, the order is determined by the order of item records in the schematic file.

Final Notes

More?

The union uData declaration includes both “char* str” and “unsigned char* bytes.” Not sure if the latter is used by the Qspice template generator or if it’s just there for convenience for those who know about it. Might be for bit vectors of some arbitrary width. Open question.

Conclusion

If you made it this far, well, congratulations and thanks for sharing the ride. I hope that you find the information useful.

Please let me know if you find problems or suggestions for improving this documentation.

--robert