# Good practices for editing VHDL or C codes in a word processor

This is an example of a paragraph (Calibri (Body)) in a word processor text. And below you have a box representing C code. In this way you have coloured text and not an image which occupies a lot more of memory and is difficult to edit.

|  |
| --- |
| 206 /\* =============================================================================207 Scan inputs and process the bits so that we can set our convenient variables208 many of the variables are "char" because we can easily monitor them using209 the watch window for debugging purposes.210 ================================================================================ \*/211 void read\_inputs (void){212 // Let's prepare our E\_L bit as a byte of memory (Var\_E),213 // so that we'll be able to "watch" it when running the simulation214215 // 1) Read all the bits (like LD a parallel 8-bit register)216 Buffer = PORTC;217 // 2) Mask the bit of interest ("&" is a bitwise AND)218 Buffer = Buffer & 0b00001000;219220 // 3) Change the active level and save221 if (Buffer == 0)222 Var\_E = 1;223 else224 Var\_E = 0; // Now it's an Active high variable225226227 // Let's read/sample/poll all the PORTC to get Ch3(1..0):228 Buffer = PORTA;229 Buffer = Buffer & 0b00011000; // Now mask the bits of interest230 Var\_Ch3 = Buffer >> 3; // Now shift and save231232233 // Let's poll all the PORTD to get Ch2(1..0):234 Buffer = PORTD;235 Var\_Ch2 = (Buffer & 0b00001100)>> 2; // In a single instruction236237238 // Let's poll all the PORTC to get Ch1(1..0):239 Buffer = PORTC;240 Var\_Ch1 = (Buffer & 0b11000000)>> 6;241242 // Let's poll all the PORTB to get Ch0(0) and the PORTA to get Ch0(1)243 Buffer = (PORTB & 0b00001000)>> 3; // Get the Ch0(0)244245 Buffer2= (PORTA & 0b00100000)>> 4; // Get the Ch0(1))246 Var\_Ch0 = Buffer | Buffer2; // "|" is a bitwise OR247248249 // Let's read the channel selection variable250 Var\_Sel = PORTB & 0b00000011;251 }252 |

And below you have another box representing VHDL code which has been marked as bold.

|  |
| --- |
| **1 --------------------------------------------------------------------------------****2 -- UPC - EETAC - CSD****3 --------------------------------------------------------------------------------****4 -- P7. An example of a synchronous BCD counter****5 --****6 -- FSM design style. There are better or shorter ways to write****7 -- this counter, but this examples can be used as a seed file****8 -- to adapt many others FSM.****9 --****10 -- http://digsys.upc.edu****11 --------------------------------------------------------------------------------****12****13 LIBRARY ieee;****14 USE IEEE.STD\_LOGIC\_1164.all;****15 USE IEEE.STD\_LOGIC\_UNSIGNED.all;****16****17****18 ENTITY counter\_BCD\_1digit IS****19 Port ( CLK : IN STD\_LOGIC;****20 CD : IN STD\_LOGIC;****21 CE : IN STD\_LOGIC;****22 Q : OUT STD\_LOGIC\_VECTOR(3 DOWNTO 0);****23 TC10 : OUT STD\_LOGIC****24 );****25****26 END counter\_BCD\_1digit;** |

Annex A: Full program listing example

1 /\* =============================================================================

2 UPC - EETAC - CSD - http://digsys.upc.edu

3 Project P9 tutorial on the introduction of microcontrollers.

4 Dual\_MUX4 combinational circuit.

5

6 Example of using C language and PIC microcontrollers to solve logic equations or

7 design combinational blocks. A simple way to introduce the basics of low level

8 programming (bitwise operations) and code organisation.

9

10 ============================================================================= \*/

11

12 // This general header file includes the device specific headers like

13 // "pic18f4520.h". This file contains all the microcontroller declarations

14 // and the MPLAB X IDE handles it all.

15 #include <xc.h>

16

17 /\* ------------------------------------

18 Definitions

19 -------------------------------------------------------------------------- \*/

20

21 /\* ------------------------------------

22 Function prototypes

23 -------------------------------------------------------------------------- \*/

24 void init\_system (void);

25 void read\_inputs (void);

26 void truth\_table (void);

27 void write\_outputs (void);

28

29 /\* -----------------------------------

30 Global variables

31 -------------------------------------------------------------------------- \*/

32 /\* Let's use convenient byte-type (char) variables. Each variable is an 8-bit

33 \* RAM memory position to make simple watching their values. \*/

34

35 static char Var\_E; // Enable pin (E\_L) that will be changed to active high.

36 static char Var\_Ch0;

37 static char Var\_Ch1;

38 static char Var\_Ch2;

39 static char Var\_Ch3;

40 static char Var\_Sel; // Selection of channels

41 static char Var\_Y; // Output

42

43 static char Buffer; // An 8-bit memory position where to store partial results

44 // while processing the data internally in the functions

45 static char Buffer2;

46

47 // ---> So, as you see, this application will require at least 9 bytes of RAM

48 /\* =============================================================================

49 Main function

50 ========================================================================== \*/

51 void main(void){ // Let's make main() the simplest possible.

52 init\_system();

53 while(1) { /\* loop forever reading inputs and calculating outputs\*/

54

55 read\_inputs();

56 truth\_table();

57 write\_outputs();

58 }

59 }

60

61 /\* ------------------------------------

62 Function definitions

63 -------------------------------------------------------------------------- \*/

64

65 //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

66 // Initialise the microcontroller system

67 //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

68 void init\_system (void){

69 // Configuration bits

70

71 // These are the configuration bits translated to the XC8 Compiler

72 // Microcontroller general configuration bits. You don't have to change them.

73 // PIC18F4520 Configuration Bit Settings. They can be generated automatically

74 // in the MPLAB X IDE

75

76 // CONFIG1H

77 #pragma config OSC = XT // Oscillator Section bits (XT oscillator)

78 #pragma config FCMEN = OFF

79 // Fail-Safe Clock Monitor E bit (Fail-Safe Clock Monitor disabled)

80 #pragma config IESO = OFF

81 // Internal/External Osc Switch-over bit (Oscillator Switch-over mode disabled)

82

83 // CONFIG2L

84 #pragma config PWRT = ON // Power-up Timer E bit (PWRT enabled)

85 #pragma config BOREN = SBORDIS

86 // Brown-out Reset E bits (Brown-out Reset enabled in hardware

87 // only (SBOREN is disabled))

88 #pragma config BORV = 3 // Brown Out Reset Voltage bits (Minimum setting)

89

90 // CONFIG2H

91 #pragma config WDT = OFF

92 // Watchdog Timer E bit (WDT disabled (control is on the SWDTEN bit))

93 #pragma config WDTPS = 32768

94 // Watchdog Timer Postscale Sect bits (1:32768)

95

96 // CONFIG3H

97 #pragma config CCP2MX = PORTC

98 // CCP2 MUX bit (CCP2 input/output is multiplexed with RC1)

99 #pragma config PBADEN = OFF

100 // PORTB A/D E bit (PORTB<4:0> pins are configured

101 // as digital I/O on Reset)

102 #pragma config LPT1OSC = OFF

103 // Low-Power Timer1 Oscillator E bit (Timer1 configured for

104 // higher power operation)

105 #pragma config MCLRE = ON

106 // MCLR Pin E bit (MCLR pin enabled; RE3 input pin disabled)

107

108 // CONFIG4L

109 #pragma config STVREN = ON

110 // Stack Full/Underflow Reset E bit (Stack full/underflow will cause Reset)

111 #pragma config LVP = OFF

112 // Single-Supply ICSP E bit (Single-Supply ICSP disabled)

113 #pragma config XINST = OFF

114 // Extended Instruction Set E bit (Instruction set extension and Indexed

115 // Addressing mode disabled (Legacy mode))

116

117 // CONFIG5L

118 #pragma config CP0 = OFF

119 // Code Protection bit (Block 0 (000800-001FFFh) not code-protected)

120 #pragma config CP1 = OFF

121 // Code Protection bit (Block 1 (002000-003FFFh) not code-protected)

122 #pragma config CP2 = OFF

123 // Code Protection bit (Block 2 (004000-005FFFh) not code-protected)

124 #pragma config CP3 = OFF

125 // Code Protection bit (Block 3 (006000-007FFFh) not code-protected)

126

127 // CONFIG5H

128 #pragma config CPB = OFF

129 // Boot Block Code Protection bit

130 // (Boot block (000000-0007FFh) not code-protected)

131 #pragma config CPD = OFF

132 // Data EEPROM Code Protection bit (Data EEPROM not code-protected)

133

134 // CONFIG6L

135 #pragma config WRT0 = OFF

136 // Write Protection bit (Block 0 (000800-001FFFh) not write-protected)

137 #pragma config WRT1 = OFF

138 // Write Protection bit (Block 1 (002000-003FFFh) not write-protected)

139 #pragma config WRT2 = OFF

140 // Write Protection bit (Block 2 (004000-005FFFh) not write-protected)

141

142 #pragma config WRT3 = OFF

143 // Write Protection bit (Block 3 (006000-007FFFh) not write-protected)

144

145 // CONFIG6H

146 #pragma config WRTC = OFF

147 // Configuration Register Write Protection bit (Configuration registers

148 // (300000-3000FFh) not write-protected)

149 #pragma config WRTB = OFF

150 // Boot Block Write Protection bit (Boot block (000000-0007FFh)

151 // not write-protected)

152 #pragma config WRTD = OFF

153 // Data EEPROM Write Protection bit (Data EEPROM not write-protected)

154

155 // CONFIG7L

156 #pragma config EBTR0 = OFF

157 // Table Read Protection bit (Block 0 (000800-001FFFh) not protected from

158 // table reads executed in other blocks)

159 #pragma config EBTR1 = OFF

160 // Table Read Protection bit (Block 1 (002000-003FFFh) not protected from

161 // table reads executed in other blocks)

162 #pragma config EBTR2 = OFF

163 // Table Read Protection bit (Block 2 (004000-005FFFh) not protected from table

164 // reads executed in other blocks)

165 #pragma config EBTR3 = OFF

166 // Table Read Protection bit (Block 3 (006000-007FFFh) not protected from table

167 // reads executed in other blocks)

168

169 // CONFIG7H

170 #pragma config EBTRB = OFF

171 // Boot Block Table Read Protection bit (Boot block (000000-0007FFh)

172 // not protected from table reads executed in other blocks)

173

174 /\* -+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+\*/

175 // Initialise starts here:

176 PORTA = 0x00; // Reset all Flip-Flops at PORTA

177 LATA = 0x00; //Another way to initialise the PORTA

178

179 /\* All the PortA is set to be digital I/O. Remember that the PORTA

180 \* can be also defined as analogue inputs, and this is why we must

181 \* configure this special register ADCON1 \*/

182 ADCON1 = 0x0F;

183 //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

184 TRISA = 0b00111000; // This is for setting the pin direction

185 // '1' means an input; '0' means an output

186 // RA4, RA3 and RA5 are inputs

187

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

188 PORTB = 0x00; // Reset all Flip-Flops at PORTB

189 LATB = 0x00;

190 TRISB = 0x00001011; // RB3, RB1 and RB0 are inputs

191

192 PORTC = 0x00; // Reset all Flip-Flops at PORTC

193 LATC = 0x00;

194 TRISC = 0x11001000; // RC7, RC6 and RC3 are inputs

195

196 PORTD = 0x00; // Reset all Flip-Flops at PORTD

197 LATD = 0x00;

198 TRISD = 0b00001100; // RD3 and RD2 are inputs

199

200

201 GIE = 0; // No interrupts used in this very simple application

202

203 Var\_Y = 0x00; //Output variable cleared

204 }

205

206 /\* =============================================================================

207 Scan inputs and process the bits so that we can set our convenient variables

208 many of the variables are "char" because we can easily monitor them using

209 the watch window for debugging purposes.

210 =========================================================================== \*/

211 void read\_inputs (void){

212 // Let's prepare our E\_L bit as a byte of memory (Var\_E),

213 // so that we'll be able to "watch" it when running the simulation

214

215 // 1) Read all the bits (like LD a parallel 8-bit register)

216 Buffer = PORTC;

217 // 2) Mask the bit of interest ("&" is a bitwise AND)

218 Buffer = Buffer & 0b00001000;

219

220 // 3) Change the active level and save

221 if (Buffer == 0)

222 Var\_E = 1;

223 else

224 Var\_E = 0; // Now it's an Active high variable

225

226

227 // Let's read/sample/poll all the PORTC to get Ch3(1..0):

228 Buffer = PORTA;

229 Buffer = Buffer & 0b00011000; // Now mask the bits of interest

230 Var\_Ch3 = Buffer >> 3; // Now shift and save

231

232

233 // Let's poll all the PORTD to get Ch2(1..0):

234 Buffer = PORTD;

235 Var\_Ch2 = (Buffer & 0b00001100)>> 2; // In a single instruction

236

237

238 // Let's poll all the PORTC to get Ch1(1..0):

239 Buffer = PORTC;

240 Var\_Ch1 = (Buffer & 0b11000000)>> 6;

241

242 // Let's poll all the PORTB to get Ch0(0) and the PORTA to get Ch0(1)

243 Buffer = (PORTB & 0b00001000)>> 3; // Get the Ch0(0)

244

245 Buffer2= (PORTA & 0b00100000)>> 4; // Get the Ch0(1))

246 Var\_Ch0 = Buffer | Buffer2; // "|" is a bitwise OR

247

248

249 // Let's read the channel selection variable

250 Var\_Sel = PORTB & 0b00000011;

251 }

252

253 /\* =============================================================================

254 The behavioural description of the circuit operation or the data processing

255 algorithm.

256 When this function is executed, all the variables are in the software domain,

257 meaning that they are completely independent of the hardware, thus the

258 algorithm is compatible among microcontroller families and pin connexions.

259 =========================================================================== \*/

260 void truth\_table(void){

261

262 // This is the combinational circuit truth table described in a behavioural way:

263

264 if (Var\_Sel == 0 && Var\_E == 1) {

265 Var\_Y = Var\_Ch0;

266 }

267 else if (Var\_Sel == 1 && Var\_E == 1){

268 Var\_Y = Var\_Ch1;

269 }

270 else if (Var\_Sel == 2 && Var\_E == 1){

271 Var\_Y = Var\_Ch2;

272 }

273 else if (Var\_Sel == 3 && Var\_E == 1){

274 Var\_Y = Var\_Ch3;

275 }

276 else

277 Var\_Y = 0x00;

278 }

279

280 /\* =============================================================================

281 Driving the output pins. Let's drive the output ports, converting

282 microcontroller variables into electrical signals.

283 ========================================================================== \*/

284 void write\_outputs(void){

285

286 Buffer = (Var\_Y & 0b00000010) << 6 ; // Y(1) in RB7

287 Buffer2 = (Var\_Y & 0b00000001) << 2 ; // Y(0) in RB2

288 PORTB = Buffer | Buffer2;

289 // It is better to write the PORT in a single instruction, thus, it will be

290 // required to mask and organise the port pins as when reading inputs,

291 // specially if the same port that you like to write contains already other

292 // outputs.

293 }

294\*\*\*\*

Annex B: Full program listing example

1 --------------------------------------------------------------------------------

2 -- UPC - EETAC - CSD

3 --------------------------------------------------------------------------------

4 -- P7. An example of a synchronous BCD counter

5 --

6 -- FSM design style. There are better or shorter ways to write

7 -- this counter, but this examples can be used as a seed file

8 -- to adapt many others FSM.

9 --

10 -- http://digsys.upc.edu

11 --------------------------------------------------------------------------------

12

13 LIBRARY ieee;

14 USE IEEE.STD\_LOGIC\_1164.all;

15 USE IEEE.STD\_LOGIC\_UNSIGNED.all;

16

17

18 ENTITY counter\_BCD\_1digit IS

19 Port ( CLK : IN STD\_LOGIC;

20 CD : IN STD\_LOGIC;

21 CE : IN STD\_LOGIC;

22 Q : OUT STD\_LOGIC\_VECTOR(3 DOWNTO 0);

23 TC10 : OUT STD\_LOGIC

24 );

25

26 END counter\_BCD\_1digit;

27

28 -- Internal desciption in FSM style

29

30 ARCHITECTURE FSM\_like OF counter\_BCD\_1digit IS

31 -- Internal wires

32 -- State signals declaration

33 -- A BCD counter will consist of 10 diferent states

34 TYPE State\_type IS (Num0, Num1, Num2, Num3, Num4, Num5,

35 Num6, Num7, Num8, Num9) ;

36

37 -----> This is important: specifying to the synthesiser the code for the states

38 ATTRIBUTE syn\_encoding : STRING;

39 ATTRIBUTE syn\_encoding OF State\_type : TYPE IS "sequential";

40 -- (It may be:"sequential" (binary); "gray"; "one-hot", etc.

41 SIGNAL present\_state, next\_state : State\_type ;

42

43 -- These two lines work Ok, but if the the state machine is not recognised

44 -- is better use enum\_encoding:

45 -- ATTRIBUTE enum\_encoding : string;

46 -- ATTRIBUTE enum\_encoding of State\_type : TYPE IS "sequential";

47

48

49 -- Constants for special states (the first and the last state)

50 CONSTANT Reset : State\_type := Num0; -- The fisrt state.

51 CONSTANT Max\_count : State\_type := Num9; -- The last state.

52

53 BEGIN

54 ---------- State Register: the only clocked block ------------------------------

55 --------------------------------------------------------------------------------

56 --------- The "memory" of the system:

57 --------- the next events (after a CLK rising edge) will depend on past events

58 --------- and the current input values

59

60 state\_register: PROCESS (CD,CLK, next\_state)

61 BEGIN

62

63 IF CD = '1' THEN -- reset the state memory

64 present\_state <= Reset;

65 ELSIF (CLK'EVENT and CLK = '1') THEN

66 -- Synchronous register (D-type flip-flop)

67 present\_state <= next\_state;

68 END IF;

69

70 END PROCESS state\_register;

71

72 ---------- CC1: Combinational system for calculating next state ----------------

73 --------------------------------------------------------------------------------

74 CC\_1: PROCESS (present\_state, CE)

75 BEGIN

76 IF CE = '0' THEN

77 next\_state <= present\_state; -- count disable

78

79 ELSE -- a simple state up count

80 CASE present\_state IS

81 WHEN Num0 =>

82 next\_state <= Num1 ;

83 WHEN Num1 =>

84 next\_state <= Num2 ;

85 WHEN Num2 =>

86 next\_state <= Num3 ;

87 WHEN Num3 =>

88 next\_state <= Num4 ;

89 WHEN Num4 =>

90 next\_state <= Num5 ;

91 WHEN Num5 =>

92 next\_state <= Num6 ;

93 WHEN Num6 =>

94 next\_state <= Num7 ;

95 WHEN Num7 =>

96 next\_state <= Num8 ;

97 WHEN Num8 =>

98 next\_state <= Num9 ;

99 WHEN Num9 =>

100 next\_state <= Num0 ;

101 END CASE ;

102

103 END IF;

104

105 END PROCESS CC\_1;

106

107 ---------- CC\_2: combinational system for calculating outputs ------------------

108 --------------------------------------------------------------------------------

109 CC\_2: PROCESS (present\_state, CE)

110 BEGIN

111 -- The terminal count output

112 IF ((present\_state = Max\_count) AND (CE = '1') ) THEN

113 TC10 <= '1';

114 ELSE

115 TC10 <= '0';

116 END IF;

117

118 -- And now just copying the present state to the output:

119 CASE present\_state IS

120 WHEN Num0 =>

121 Q <= "0000";

122 WHEN Num1 =>

123 Q <= "0001";

124 WHEN Num2 =>

125 Q <= "0010";

126 WHEN Num3 =>

127 Q <= "0011";

128 WHEN Num4 =>

129 Q <= "0100";

130 WHEN Num5 =>

131 Q <= "0101";

132 WHEN Num6 =>

133 Q <= "0110";

134 WHEN Num7 =>

135 Q <= "0111";

136 WHEN Num8 =>

137 Q <= "1000";

138 WHEN Num9 =>

139 Q <= "1001";

140 END CASE ;

141

142 END PROCESS CC\_2;

143

144 -- Place here other logic if necessary

145

146 END FSM\_like;

147

148

149

150

151

152

153