【作業内容】

プログラムを完成させる

I2C温度・圧力センサの回路を組む

実際に数値を計測する

【作業項目】

1. プログラムを作成する。今回作成したプログラムを以下に示す。

|  |
| --- |
| 1. #include <stdio.h> 2. #include <stdlib.h> 3. #include <string.h> 4. #include <wiringPi.h> 5. #include <wiringPiI2C.h> 6. // BMP180のI2Cインタフェースのアドレス 7. // 確認方法：gpio i2cdetect 8. #define BMP180ADDR 0x77 9. // レジスタアドレスの定義 10. // キャリブレーションデータレジスタのアドレス 11. // AC1-AC6,B1,B2,MB,MC,MD 全て2バイト（16ビット）長 12. // 先頭アドレス 13. #define CALSADR 0xaa 14. // 終了アドレス 15. #define CALEADR 0xbf 16. // キャリブレーションデータの個数 17. #define CALDATANUM 11 18. // キャリブレーション用のデータ配列参照用 19. #define AC1 0 20. #define AC2 1 21. #define AC3 2 22. #define AC4 3 23. #define AC5 4 24. #define AC6 5 25. #define B1 6 26. #define B2 7 27. #define MB 8 28. #define MC 9 29. #define MD 10 30. // データレジスタのアドレス 31. #define DATAMSB 0xF6 32. #define DATALSB 0xF7 33. #define DATAXLSB 0xF8 34. // コントロールレジスタのアドレス 35. // b7,b6:oss b5:sco b4-b0:measurement 36. // oss->０:lowpower,1:standard,2:highres,3:ultrahighres 37. #define CTRLREG 0xf4 38. // 測定開始コマンドの定義 39. #define TEMP 0x2e 40. #define PRESS0 0x34 41. #define PRESS1 0x74 42. #define PRESS2 0xb4 43. #define PRESS3 0xf4 44. // リセットレジスタ（書き込み専用）のアドレス 45. // 0xb6を書き込むとパワーオンリセット動作 46. #define RESETREG 0xe0 47. // IDレジスタ 48. // 機能チェック用、正常なら読み出すと常に0x55 49. #define IDREG 0xd0 50. // コンパイル方法 51. // gcc -Wall -o bmp180 bmp180.c -lwiringPi 52. // 測定モードの定義（oss） 53. #define LOWPOWER 0 54. #define STANDARD 1 55. #define HIGHRES 2 56. #define ULTRARES 3 57. // BMP180の機能チェック用 58. #define CHECKOK 0 59. #define CHECKNG -1 60. // キャリブレーションレジスタ名のテーブル 61. const char\* calregname[11] = {"AC1", "AC2", "AC3", "AC4", "AC5", "AC6", 62. "B1", "B2", "MB", "MC", "MD"}; 63. // 以下の定数は補正アルゴリズム中にある数値をそのまま入れたもの 64. // 実際の測定値や補正データの代わりに使えば、期待した動作かどうかの検証ができる 65. int testcaldata[11] = {408, -72, -14383, 32741, 32757, 23153, 66. 6190, 4, -32768, -8711, 2868}; 67. int testut = 27898; 68. int testup = 23843; 69. int testoss = 0; 70. // プロトタイプ宣言 71. int get\_press(int ut, int up, int oss, int\* caldata); 72. int get\_temp(int ut, int\* caldata); 73. int get\_raw\_press(int fd, int oss); 74. int get\_raw\_temp(int fd); 75. int check\_bmp180\_function(int fd); 76. int\* read\_caldata(int fd, int\* caldata); 77. int main() { 78. int fd; 79. // デバイスディスクプリタ、デバイスごとにつく番号のようなもの 80. int oss = 1; 81. // 気圧標準測定（2回サンプリング平均） 82. int caldata[CALDATANUM]; 83. // 補正データを格納する配列 84. int ut, up; 85. // 生の温度・圧力値（ADCの変換結果そのもの） 86. int t, p; 87. // 補正した後の温度・圧力値 88. // I2Cインタフェースの初期化 89. fd = wiringPiI2CSetup(BMP180ADDR); 90. if (fd < 0) { 91. printf("I2C初期化エラー！\n"); 92. exit(EXIT\_FAILURE); 93. } 94. // 動作チェック機能を使った動作確認 95. if (check\_bmp180\_function(fd) != CHECKOK) { 96. printf("BMP180の動作確認が不良です。\n"); 97. exit(EXIT\_FAILURE); 98. } 99. // 100. printf("BMP180の動作確認OK\n\n"); 101. read\_caldata(fd, caldata); 102. // 補正データの読み出し 103. ut = get\_raw\_temp(fd); 104. // 温度測定値の読み出し 105. up = get\_raw\_press(fd, oss); 106. // 気圧測定値の読み出し 107. t = get\_temp(ut, caldata); 108. // 補正計算を行って補正した温度を求める 109. // t = get\_temp(testut,testcaldata);のようにすると補正の動作確認ができる 110. printf("気温（補正済み）= %4.1f °C, ", (float)t / 10.0); 111. p = get\_press(ut, up, oss, caldata); 112. // 補正計算を行って補正した気圧を求める 113. //p = get\_press(testut, testup, 114. // testoss,testcaldata);で補正の動作確認ができる 115. printf("気圧（補正済み）= %6.2f hPa\n", (float)p / 100.0); 116. return 0; 117. } 118. int get\_press(int ut, int up, int oss, int\* caldata) 119. // 測定した温度と気圧データから、気圧の補正計算を行う関数 120. // 戻り値は補正した気圧の値 121. { 122. long X1 = (ut-caldata[AC6])\*caldata[AC5]/32768; 123. long X2 = caldata[MC]\*2048/(X1+caldata[MD]); 124. long B5 = X1+X2; 125. long B6 = B5-4000; 126. X1 = (caldata[B2]\*(B6\*B6/4096))/2048; 127. X2 = caldata[AC2]\*B6/2048; 128. long X3 = X1+X2; 129. long B3 = (((caldata[AC1]\*4+X3)<<oss)+2)/4; 130. X1 = caldata[AC3]\*B6/8192; 131. X2 = (caldata[B1]\*(B6\*B6/4096))/65536; 132. X3 = ((X1+X2)+2)/4; 133. unsigned long B4 = caldata[AC4]\*(unsigned long)(X3+32768)/32768; 134. unsigned long B7 = ((unsigned long)up-B3)\*(50000>>oss); 135. long p; 136. if(B7<0x80000000){ 137. p=(B7\*2)/B4; 138. }else{ 139. p=(B7/B4)\*2; 140. } 141. X1 = (p/256)\*(p/256); 142. X1 = (X1\*3038)/65536; 143. X2 = (-7357\*p)/65536; 144. p += (X1+X2+3791)/16; 145. return p; 146. } 147. int get\_temp(int ut, int\* caldata) 148. // 温度測定値と補正データを引数にとって補正した温度を求める関数 149. // 戻り値は補正した温度（整数計算のため、真値の10倍になっているはず） 150. { 151. long X1 = (ut-caldata[AC6])\*caldata[AC5]/32768; 152. long X2 = caldata[MC]\*2048/(X1+caldata[MD]); 153. long B5 = X1+X2; 154. long t = (B5+8)/16; 155. return t; 156. } 157. int get\_raw\_press(int fd, int oss) 158. // 気圧の測定値を求める関数 159. // ossで測定時の変換回数を指定する 160. { 161. int m,l, x; 162. // MSB, LSB, XLSBを入れる変数 163. int up; 164. // 計算して求めた値を入れる変数 165. // ossの範囲は0から3まで 166. if (oss < 0) oss = 0; 167. if (oss > 3) oss = 3; 168. wiringPiI2CWriteReg8(fd, CTRLREG, PRESS0 + (oss << 6)); 169. // 変換開始 170. // 変換時間待ち、ossの値（変換回数=2^oss）によって待ち時間が異なる 171. switch (oss) { 172. case 0: 173. delay(5); 174. break; 175. case 1: 176. delay(8); 177. break; 178. case 2: 179. delay(14); 180. break; 181. default: 182. delay(26); 183. } 184. // ここにデータレジスタからMSB, LSB, XLSBを読み出すコードを書く 185. // 気圧は3バイトのデータ値から計算することになるので注意 186. m = wiringPiI2CReadReg8(fd, DATAMSB); 187. l = wiringPiI2CReadReg8(fd, DATALSB); 188. x = wiringPiI2CReadReg8(fd, DATAXLSB); 189. // ここに読み出したm, l, xから値を計算するコードを書く 190. up = ((m<<16) + (l<<8) + x) >> (8-oss); 191. return up; 192. } 193. int get\_raw\_temp(int fd){ 194. int m, l; 195. // 読み出したMSB, LSBを入れる変数 196. int ut; 197. // 計算で求めた測定温度を入れる変数 198. wiringPiI2CWriteReg8(fd, CTRLREG, TEMP); 199. // 温度の測定開始 200. delay(5); 201. // 変換時間待ち、最大変換時間は4.5ms 202. // ここにデータレジスタからMSB, LSB, XLSBを読み出すコードを書く 203. // 気圧は2バイトのデータ値から計算することになるので注意 204. m = wiringPiI2CReadReg8(fd, DATAMSB); 205. l = wiringPiI2CReadReg8(fd, DATALSB); 206. //ここに読み出したm, lから値を計算するコードを書く 207. ut = (m<<8) + l; 208. return ut; 209. } 210. int check\_bmp180\_function(int fd) 211. // BMP180の機能チェックを行う関数 212. { 213. int r; 214. if (wiringPiI2CReadReg8(fd, IDREG) == 0x55){ 215. r = CHECKOK; 216. }else{ 217. r = CHECKNG; 218. } 219. return r; 220. } 221. int\* read\_caldata(int fd,int\* caldata) 222. // キャリブレーションデータの読み出し関数 223. { 224. int i; int l,m; 225. printf("キャリブレーション値を読み込みます。\n"); 226. for (i = 0; i < CALDATANUM; i++){ 227. m = wiringPiI2CReadReg8(fd,CALSADR+i\*2); 228. l = wiringPiI2CReadReg8(fd,CALSADR+i\*2+1); 229. caldata[i] = (m<<8) + l; 230. // 符号なし16ビットに変換 231. if ((i != AC4) && (i != AC5) && (i != AC6)){ 232. // AC4, AC5, AC6は符号付き16ビットデータなので対処が必要 233. // その対処をここに書く 234. caldata[i] = (signed short)caldata[i]; 235. } 236. } 237. return caldata; 238. } |

②　I2C温度・圧力センサの回路を組む。回路図を図1に示す。

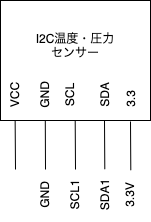
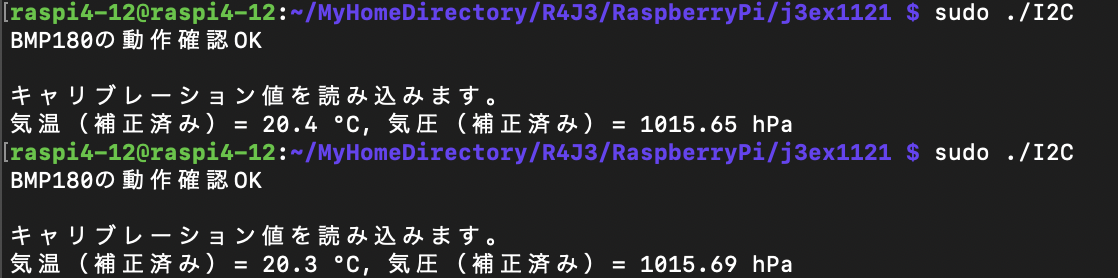


図1：I2C温度・圧力センサーの回路図

③ gcc -Wall -o I2C I2C.c -lwiringPiを実行し、プログラムをコンパイルする。

④ sudoで実行する。実行した結果を下記に示す。



温度と気圧は補正された値が出力されている。

【作業時間】

・作業時間：90分

・報告書作成時間：30分