RL78/G15-FPB で遊んでみる(2)

RL78/G15-FPB で遊んでみる(1)は、単なる無意味なパターンを動かしているだけのプログラムでしたが、今回は少し実用に近いプログラムです。図 1. に示すように数値等を表示するようにしてみます。　　#1　　#2　　#3　　#4

図 1. LED 表示器の構成

使用する環境及びハードウェアは前回とほぼ同じです。ソフトウェアも大きく変更したのは main.c だけです(12 ビット・インターバル・タイマは 100ms に変更されていますが)。

ここでは、以下の内容を説明します。CSI(SPI)通信については、RL78/G15-FPB で遊んでみる(1)を参照してください。

・(1) MAX7219 を用いた 4 桁の8×8ドットマトリクス LED 表示モジュールの問題点
・(2) 表示用フォント・データの準備
・(3) フォント・データの変換
・(4) main 関数
・(5) 関数の説明
・(6) MAX7219 の制御について

（1）MAX7219 を用いた 4 桁の8×8ドットマトリクス LED 表示モジュールの問題点

今回使用した LED 表示モジュールは、SO パッケージの MAX7219 が LED の下に図 2. に示す写真のように実装されています。このように実装するのは、プリント板の配線を簡単にするためだと考えられます。

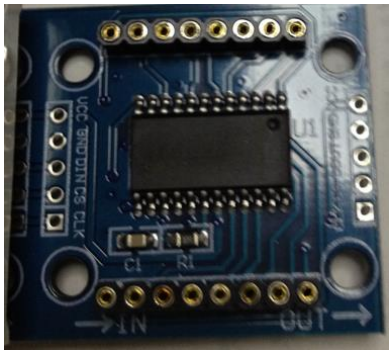

図 2. MAX7219 の実装状態

このためだと思われるのが、図 3.に示すような信号接続になっていることです。そのため、セグメント・データを送信すると、そのデータは横方向に並ぶことになります。8×8 ドット単位で使うのであれば、これでも何も問題はありません。

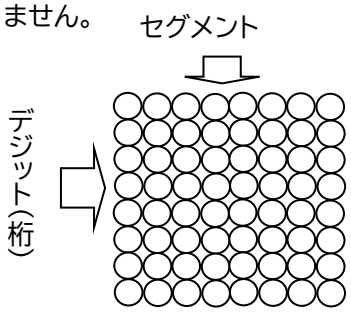

図 3. LED 表示器の信号接続

しかし、文字のフォントを表示する場合には、図 4. に示すような 7×5 ドットの構成になるかと思います。

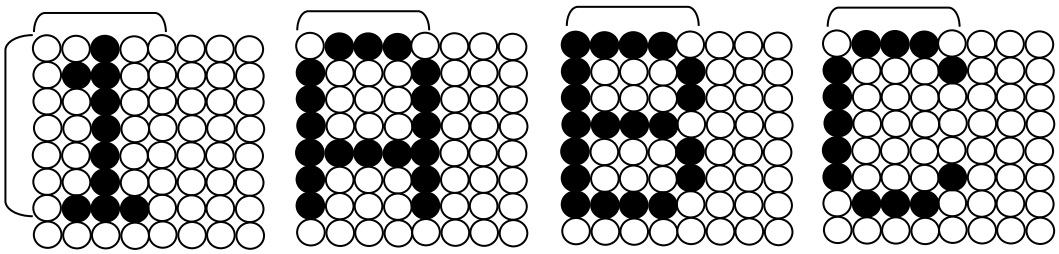

図 4. 表示するフォントの例

7×5 ドットのフォントであれば、32 ドットの幅には 1 ドットの間隔を含めても 5 桁の表示が可能ですが、セグメントが表示の横方向になっているので、表示用フォント・データを作るのが面倒です。RL78/G15 ではコード・フラッシュが 8kB で RAM が 1kB なので、今回は単純に 4 桁の表示だけにとどめておきます。また、フォント・データも横方向のパターンを行の分だけ準備します。

（2）表示用フォント・データ

LED 表示モジュールに送信することに対応した表示用フォント・データを準備します（実際には、過去に使ったデータを流用し、いくつか追加しただけです）。

実際のフォント・データは図 5.に示すように、2 次元の配列で 8 バイト（8 行分）のデータを 32 組準備しています（実際には、0x17 以降は消灯状態になっています）。

```
uint8_t const FONT_DATA[32][8] =
{   /* MAX7219用表示データ配列  */
    {0x0E,0x11,0x11,0x11,0x11,0x11,0x0E,0x00},  // 00: ０のフォント
    {0x0E,0x04,0x04,0x04,0x04,0x06,0x04,0x00},  // 01: １のフォント
    {0x1F,0x02,0x04,0x08,0x10,0x11,0x0E,0x00},  // 02: ２のフォント
    {0x0E,0x11,0x10,0x08,0x04,0x08,0x1F,0x00},  // 03: ３のフォント
    {0x08,0x08,0x1F,0x09,0x0A,0x0C,0x08,0x00},  // 04: ４のフォント
    {0x0E,0x11,0x10,0x10,0x0F,0x01,0x1F,0x00},  // 05: ５のフォント
    {0x0E,0x11,0x11,0x0F,0x01,0x01,0x0E,0x00},  // 06: ６のフォント
    {0x02,0x02,0x02,0x04,0x08,0x10,0x1F,0x00},  // 07: ７のフォント
    {0x0E,0x11,0x11,0x0E,0x11,0x11,0x0E,0x00},  // 08: ８のフォント
    {0x07,0x08,0x10,0x1E,0x11,0x11,0x0E,0x00},  // 09: ９のフォント
    {0x11,0x11,0x1F,0x11,0x11,0x11,0x0E,0x00},  // 0A: Ａのフォント
    {0x0F,0x11,0x11,0x0F,0x11,0x11,0x0F,0x00},  // 0B: Ｂのフォント
    {0x0E,0x11,0x01,0x01,0x01,0x11,0x0E,0x00},  // 0C: Ｃのフォント
    {0x07,0x09,0x11,0x11,0x11,0x09,0x07,0x00},  // 0D: Ｄのフォント
    {0x1F,0x01,0x01,0x0F,0x01,0x01,0x1F,0x00},  // 0E: Ｅのフォント
    {0x01,0x01,0x01,0x0F,0x01,0x01,0x1F,0x00},  // 0F: Ｆのフォント
    {0x00,0x06,0x06,0x00,0x06,0x06,0x00,0x00},  // 10: ：のフォント
    {0x00,0x06,0x06,0x00,0x00,0x00,0x00,0x00},  // 11: 。のフォント
    {0x00,0x00,0x00,0x00,0x06,0x06,0x00,0x00},  // 12: °のフォント
    {0x00,0x00,0x04,0x0E,0x04,0x00,0x00,0x00},  // 13: ＋のフォント
    {0x00,0x00,0x00,0x0E,0x00,0x00,0x00,0x00},  // 14: －のフォント
    {0x00,0x20,0x10,0x08,0x04,0x02,0x01,0x00},  // 15: ／のフォント
    {0x00,0x23,0x13,0x08,0x1C,0x1A,0x01,0x00},  // 16: ％のフォント
    {0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},  // 17: □のフォント
```

図 5. フォント・データ

各 8 バイトのデータは、図 6. に示すような構造になっています。ここで、bit の並びは LED 表示モジュール、CSI の転送方向（MSB ファーストに設定）に対応しています。

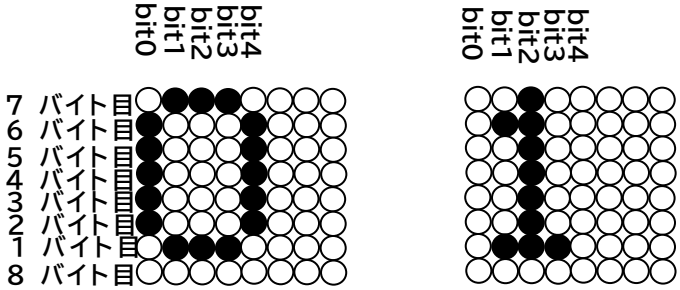

図 6. フォント・データの構成

表示したいデータを使い、0～7 で構造体を読み出して、図 7. に示す各桁用の表示パターン(配列 bit_data)のデータを作成します。

(3) フォント・データの変換

```
volatile uint8_t    bit_data[LEDDIG][8] =↓
{   /* LED の各桁の表示パターンのデータ */↓
    {0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},   /* LED#0のデータ          */↓
    {0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},   /* LED#1のデータ          */↓
    {0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},   /* LED#2のデータ          */↓
    {0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00}    /* LED#3のデータ          */↓
};↓
```

図 7. 各桁用の表示パターン

ここには、各桁の LED のフォント・パターンがセットされていることになります。図 6. のデータは通常は、ダイナミック点灯で、表示データを頻繁に書き換えるような場合に使用します。今回のように、ダイナミック点灯の制御は MAX7219 で処理しているので、4 桁の表示を行うだけなら必須ではありません。しかし、フォント・データを色々と修飾(変更)したり、同じ表示器を用いて 5 桁表示を行なったりして、作業領域として使うことを意識して準備しています。

表示したい表示データ(配列 disp_data)は、関数 R_get_FONT を呼び出すことで、この配列(bit_data)にフォント・パターンを生成するために使用されます。必要なら、その後フォント・パターンをいじってください。フォント・データの上位 3 ビットは 0 になっているので、必要ならここにデータを追加して"."や":"を追加することができます。

このようにして、準備した表示データ・パターンは、図 8. MAX7219 に送信する配列 led_data に対応するアドレス情報と組み合わされて格納され、この 8 バイトがまとめて送信されます。

```
volatile uint8_t    led_data[LEDDIG][2] =↓
{   /* MAX7219用表示データ設定テーブル : { LED,データ } */↓
    {Digit0,0x00},                  /* LED#0の設定データ(桁,データ)      */↓
    {Digit0,0x00},                  /* LED#1の設定データ(桁,データ)      */↓
    {Digit0,0x00},                  /* LED#2の設定データ(桁,データ)      */↓
    {Digit0,0x00}                   /* LED#3の設定データ(桁,データ)      */↓
};↓
```

図 8. 送信データの形式

この配列も、絶対に作らないといけない訳ではありません。配列 bit_data のデータを読み出し、桁情報に続けて送信していくことでも対応は可能です。ここでは、各段階でデータをまとめておくことで、動作確認やデバッグ(デバッグと言う工程はないと人もいます)がやり易くなります。考え方や進め方、またプログラミングの手法はいくつもあります。自分に合った手法で進めてください。

（おまけ）

なぜ、LED 表示器ではフォントの横方向を同時に変更するようなスキャン方法になっているのでしょうか。私の想像では、人の目の解像度は横方向が高いので、横方向をできるだけ短い時間差でデータを更新しているのではないかと思っています。もっとも、横方向は 32 ドット程度で、それほどの時間差はないので、フォントの並びは縦でもいいのではないかと思います（その方がプログラムは楽です）。

と、言っても現実のハードウェアは変えられないので、面倒な（楽でない）処理について考えてみます。

図 9. に示すのが操作対象の横方向のデータの並びです。

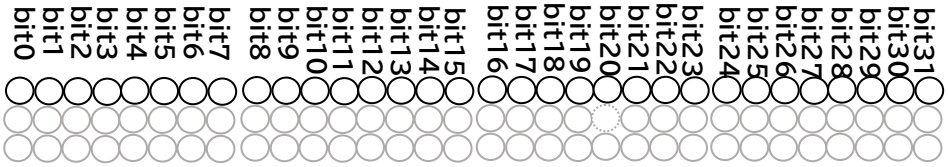

図 9. 操作対象

図 8 に示す 4 バイト（32 ビット）をレジスタ・アドレスに続けて MSB ファーストで送信するので、倍の 64 ビットを送信することで、MAX7219 に表示データを送信します。

ここで、bit31-bit24 が右端の桁、bit23-bit16 が右から 2 つ目の桁、bit15-bit8 が左から 2 つ目の桁、bit7-bit0 が左端の桁の 8×8LED に対応します。

ここに、5 桁のフォントを並べることを考えてみます。図 10. に配置の例を示します。

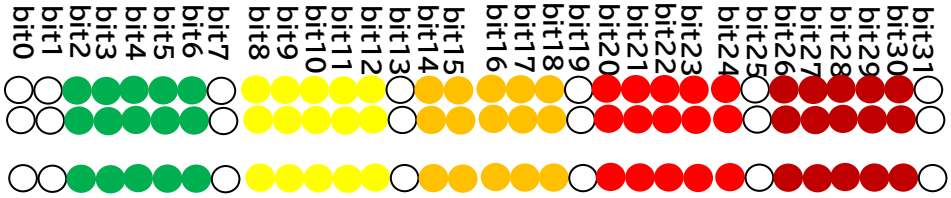

図 10. 5 桁の配置

この例では、bit30-bit26（茶色）が一の桁、bit24-bit20（赤色）が十の桁、bit18-bit14（橙色）が百の桁、bit12-bit8（黄色）が千の桁、bit6-bit2（緑色）が万の桁になります。ここに、図 4. のフォント・データの下位 5 ビットをセットしていくことになります。

(4) main 関数

ハードウェアおよび周辺機能の設定は、「RL78/G15-FPB で遊んでみる(1)」の「(2) CS+環境での使用設定」を参照してください。

main 処理では、12 ビット・インターバル・タイマで 1 秒(100ms×10 回)待った後で、LED2 を点灯してSWの押下待ち状態になったことを示してSWが押されるのを待ちます。SWが押されたら、MAX7219の初期化を行います。4 つのMAX7219の初期化はCS信号をアソートして、レジスタ・アドレス＋設定データのペアを送信し、CS 信号をネゲートします。これを、5つのレジスタに対して行います。これにより、RL78/G15-FPB 側から順に初期化されます。詳細は、「(6) MAX7219 の制御について」を参照してください。

SWが押下されたら、(低速内蔵クロックをカウントする)12 ビット・インターバル・タイマで生成した 100ms を 10 カウントした 1 秒のタイミングをカウントしてその結果を十進数で表示します。このために、得られた十進数を配列 disp_data にセットし、フォント・データに展開して、これを LED 表示器(MAX7219)に送信するだけです。

（5）関数の説明

main.c で使用している API 関数の一覧を以下に示します。

関数名	概要
R_CSI00_Start	CSI00 の動作開始
R_CSI00_Stop	CSI00 の動作停止
R_CSI_Tx	CSI00 データ送信処理を起動し、完了するまで待つ
R_CSI_Tx_start	CSI00 データ送信の起動処理を行う.
R_check_status	CSI00 の通信ステータスを確認する
R_CSI_Init	CSI00 のパラメータ(ステータス)を初期化
R_CSI_wait_ready	CSI00 の送信完了を待つ
R_MAIN_UserInit	main 関数の初期化処理
R_wait_SW	SW の押下を待つ
R_wait_100ms	100ms 単位での時間経過を待つ
R_wait_1s	1s 単位での時間経過を待つ
R_get_FONT	表示データに対応したフォント・データを設定する
R_send_DATA	表示するフォント・データを送信する

また、各 API 関数の仕様を以下に示します。

[関数名]　R_CSI00_Start

関数概要	CSI00 の動作開始のための処理を行う
プロトタイプ宣言	void R_CSI00_Start(void):
機能概要説明	SCK00 のレベルをローに設定し、CSI00 の動作を起動し、割り込み関係を初期化する。
引数	なし
リターン値	なし
概要	なし

[関数名]　R_CSI00_Stop

関数概要	CSI00 の動作を停止する
プロトタイプ宣言	void R_CSI00_Stop(void):
機能概要説明	CSI00 の動作を停止し、出力を停止し、割り込みを禁止する。
引数	なし
リターン値	なし
概要	なし

[関数名]　R_CSI00_Stop

関数概要	CSI00 の動作を停止する
プロトタイプ宣言	void R_CSI00_Stop(void):
機能概要説明	CSI00 の動作を停止し、出力を停止し、割り込みを禁止する。
引数	なし
リターン値	なし
概要	なし

[関数名]　R_CSI_Tx

関数概要	CSI00 でデータを送信する
プロトタイプ宣言	MD_STATUS　R_CSI_Tx(uint8_t　*　const　tx_buf,　uint16_t tx_num);
機能概要説明	CSI00 から、第 1 パラメータで示されたバッファから第 2 パラメータで示された数のデータを送信する。送信が完了したら、戻る。
引数	第 1 引数　　送信データを格納したバッファのアドレス 第 2 引数　　送信するデータのバイト数
リターン値	0x00:MD_OK　　送信を正常完了 0x80:MD_ERROR　　オーバーラン・エラー 0x81:MD_ARGERROR 引数(データ数)エラー
概要	この関数では、MD_ERROR は発生しない

[関数名]　R_CSI_Tx_start

関数概要	CSI00 でデータの送信を開始する
プロトタイプ宣言	MD_STATUS　R_CSI_Tx_start(uint8_t　*　const　tx_buf,　uint16_t tx_num);
機能概要説明	CSI00 から、第 1 パラメータで示されたバッファから第 2 パラメータで示された数のデータの送信を開始したら戻る。この関数で開始した送信が完了する前に、再度送信を行おうとすると、オーバーラン・エラーを戻す。
引数	第 1 引数　　送信データを格納したバッファのアドレス 第 2 引数　　送信するデータのバイト数
リターン値	0x00:MD_OK　　送信を正常完了 0x80:MD_ERROR　　オーバーラン・エラー 0x81:MD_ARGERROR 引数(データ数)エラー
概要	

[関数名]　R_check_status

関数概要	CSI00 の通信状況(ステータス)を確認する
プロトタイプ宣言	uint8_t R_check_status(void);
機能概要説明	CSI00 の通信状況を戻す
引数	なし
リターン値	0x00:MD_OK　　送信を正常完了 0x01:ビジー　　CSI00は送信中 0x80:MD_ERROR　　オーバーラン・エラー 0x81:MD_ARGERROR 引数(データ数)エラー
概要	この関数では、MD_ERROR は発生しない

[関数名]　R_CSI_Init

関数概要	CSI00 を初期化する
プロトタイプ宣言	void R_CSI_Init(void);
機能概要説明	CSI00 の通信状況(ステータス:g_status)をクリアする
引数	なし
リターン値	なし
概要	なし

［関数名］　R_CSI_wait_ready

関数概要	CSI00 送信完了を待つ
プロトタイプ宣言	void R_CSI_wait_ready(void);
機能概要説明	CSI00 の通信状況（ステータス：g_status）をチェックして、通信が完了するのを待つ。残り送信データ数が 0 になり、CSI00が通信中でなければ、CS信号をネゲートし、バッファ空き割り込みに設定し、通信ステータスを通信完了にして戻る。
引数	なし
リターン値	なし
概要	なし

［関数名］　R_MAIN_UserInit

関数概要	main 関数の初期化処理
プロトタイプ宣言	void R_MAIN_UserInit(void);
機能概要説明	CSI00 を起動し、ベクタ割り込みを許可して戻る。
引数	なし
リターン値	なし
概要	なし

［関数名］　R_wait_SW

関数概要	SW の押下を待つ
プロトタイプ宣言	void R_wait_SW(void);
機能概要説明	SW が放されている状態から押されるのを待つ
引数	なし
リターン値	なし
概要	なし

［関数名］　R_wait_100ms

関数概要	引数で渡された回数 100ms を待つ
プロトタイプ宣言	R_wait_100ms(uint16_t);
機能概要説明	100ms タイマを起動し、引数で示された回数の割り込みを待つ
引数	待ち合わせる 100ms の回数
リターン値	なし
概要	なし

［関数名］　R_wait_1s

関数概要	引数で渡された秒数だけ待つ
プロトタイプ宣言	void R_wait_1s(uint8_t);
機能概要説明	引数で示された値の 10 倍だけ R_wait_100ms を呼び出す
引数	待ち合わせる秒数
リターン値	なし
概要	なし

［関数名］　R_get_FONT

関数概要	表示データに対応したフォント・データを設定する
プロトタイプ宣言	void R_get_FONT(void);
機能概要説明	配列 disp_data に格納されている表示データに対応するフォント・データを配列 bit_data に設定する
引数	なし
リターン値	なし
概要	なし

［関数名］　R_send_DATA

関数概要	引数で渡された秒数だけ待つ
プロトタイプ宣言	void R_wait_1s(uint8_t);
機能概要説明	CS 信号をアソートし、配列 led_data のフォント・データをレジスタ・アドレスに続けて送信する。4 桁分のデータの送信が完了したら CS 信号をネゲートする処理を 8 行分繰り返す。
引数	待ち合わせる秒数
リターン値	なし
概要	なし

（6）MAX7219 の制御について

MAX7219 へのコマンドや表示データのセットは、「レジスタ・アドレス」+「設定データ／コマンド」の形式の 16 ビット・データを送信することで実現します。1 個の MAX7219 で考えると、図 11. に示すような入力信号になります。

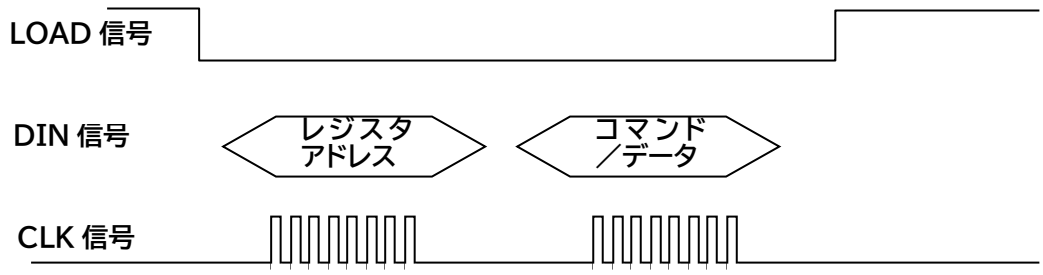

図 11. MAX7219 の信号入力

ここで、注目する必要があるのは、LOAD 信号が立ち上がった時にシフトレジスタにあるデータ（最後に受信したデータ）がその MAX7219 で有効になるということです。そのため、複数の MAX7219 をカスケード接続した場合に、送信したデータがどこまで伝わっていったかを意識しておく必要があります。

4 個の MAX7219 をカスケード接続しているときに、初期設定処理では同じ設定にします。そのために 5 つのレジスタの設定を行いますが、ここは、レジスタ・アドレスとコマンドの組み合わせを 5 組送信しますが、この場合には図 11. に示すように 2 バイトで LOAD 信号を制御します。す

ると、#1の MAX7219 には、そのコマンドが設定されます。次のコマンドを同様に送信すると、#1 には、新しいコマンドが、#2 の MAX7219 には最初のコマンドが設定されます。同様に 3 つ目のコマンドを送信すると、最初のコマンドは#3 の MAX7219 に設定されます。このようにして、最後の(5 番目の)コマンドを送信すると、#1の MAX7219 には、そのコマンドが設定され、#2 の MAX7219 にはその一つ前までにコマンドが設定されます。このように後段になると、一つ前のコマンドが設定されていき、#4 の MAX7219 には 2 番目のコマンドが設定された状態となります。後段の#5 の MAX7219 にコマンドを設定するために、以降は No_OP コマンドを送信して、#4 の MAX7219 に最後の(5 番目の)コマンドまでを送信します。

これに対して、個々の MAX7219 に異なる点灯データを設定するには、図 12.に示すような転送を行います。LOAD 信号をローにして(CS 信号をアサートして)、一番遠い#4 の MAX7219 用のデータから順に最後は一番近い#1 の MAX7219 用のデータを送信し、その後 LOAD 信号をハイにして、シフトレジスタのデータを MAX7219 に設定します。

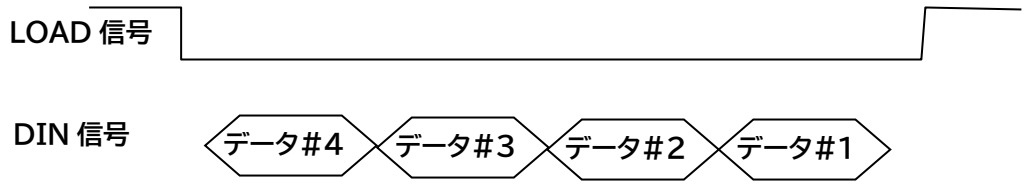

図 12. データ転送時の転送タイミング

これで、4 個の MAX7219 に新しいデータを設定できます。なお、データを変更したくない桁にはデータではなく No_OP コマンドを送信すればいいでしょう。

以上は、MAX7219 のデータ・シートを読んで判断した内容です。実際にプログラムを作成して、期待したように動作しているようなので、間違ってはいないはずです。

最後に、今回は 4 桁のモジュールを使用し、「(1) MAX7219 を用いた 4 桁の8×8ドットマトリクス LED 表示モジュールの問題点」で触れたようにスキャン方向の問題がるので、5 桁で使用するのが面倒だと結論付けしました。1 桁のモジュールも存在するので、これを使用して 90 度回転させて並べるとこの問題は解決できます。この方法を使って、次は、5 桁化を検討してみることにします。

その後は、IICA0 を使ってみることにします。

以上