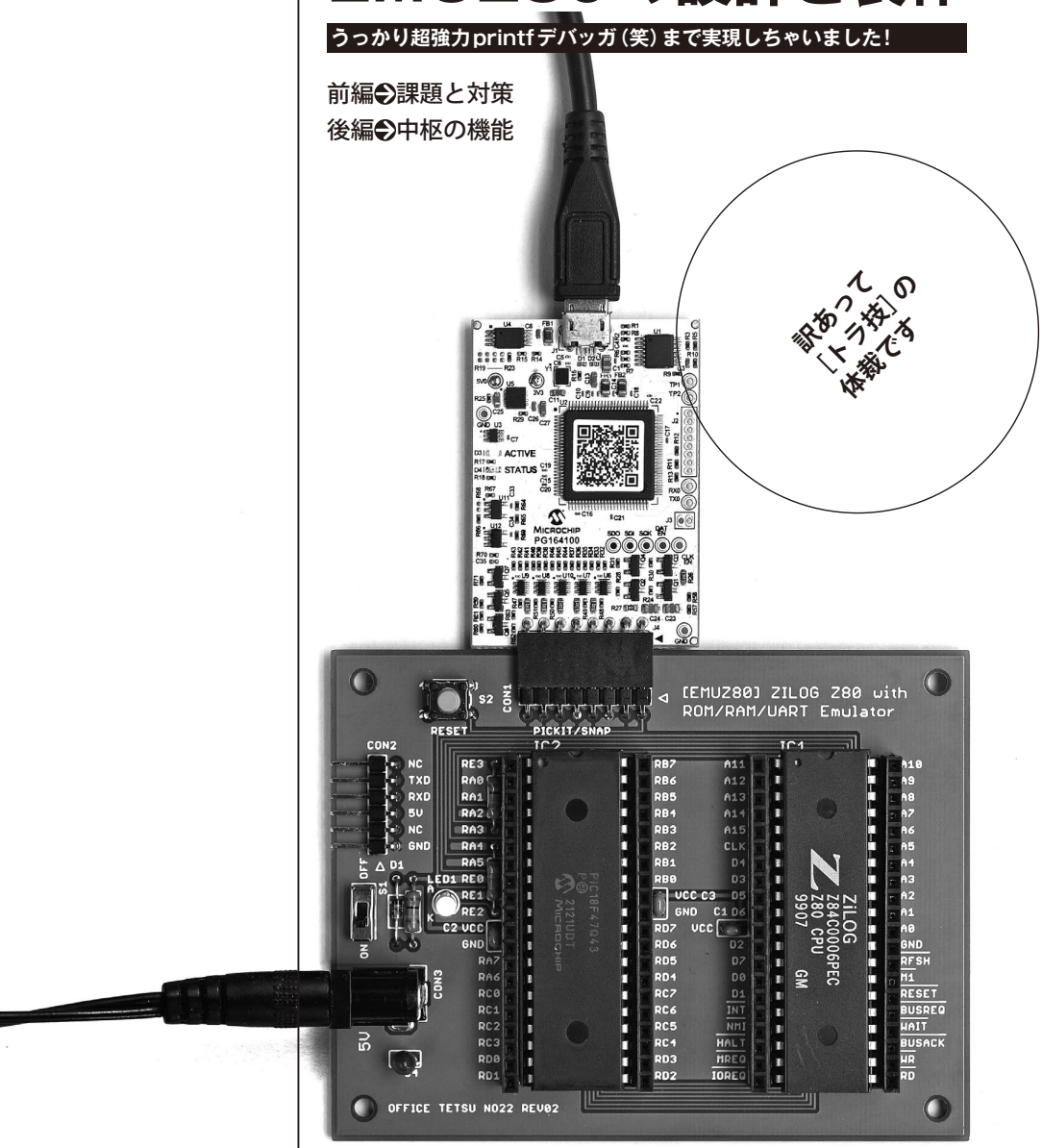
コンピュータの部品数と製作費は
どこまで減らせるだろうか?

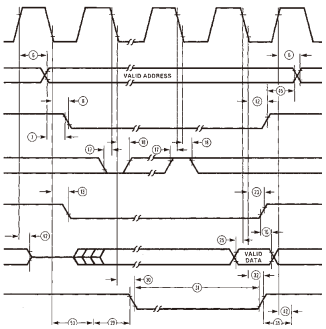
Z80 と PIC18F47Q43 だけで動く

EMUZ80の設計と製作

うっかり超強力printfデバッガ(笑)まで実現しちゃいました!

前編➔課題と対策
後編➔中枢の機能





コンピュータの部品数と製作費はどこまで減らせるだろうか？

訳あって「トラ技」の体裁です

Z80とPIC18F47Q43だけで動く

EMUZ80の設計と製作

うっかり超強力printfデバッガ（笑）まで実現しちゃいました!

前編➲課題と対策

鈴木 哲哉

Z80 コンピュータの周辺回路を PIC18F47Q43 で代替する

電子機器の設計では，よく，目標の働きをなるべく少数の安価な部品で実現するように求められます．仕事だと災難ですが，仕事でなければ一種のゲームです．面白そうなので，適度に困難な目標として，コンピュータへの採用例が豊富な CPU をひとつ選び，過去のどれよりも簡素な設計に挑戦してみました．

製作例の CPU はザイログの Z80（図 1）です．Z80 は 1976 年に発売され，初期のパソコンに広く採用されました．以来，現在まで製造されているロングセラーです．おそらくパソコンよりあとに生まれた世代でも概要くらいは知っているでしょう．そのことは，簡素化を達成したとき自慢する上で誠に好都合です．

Z80 で動くコンピュータの，ごく普通の回路は古い雑誌に掲載されています．Z80 の周辺には，メモリ，端末のインターフェース，クロック源，リセット回路があります．これらをひとつふたつ減らしても芸がありません．徹底した簡素化を目指すなら，全部の働きをひとつのマイコンで置き換えるのが妥当です．

電子工作のコミュニティには往年の CPU を現在のマイコンと 2，3 の IC で動かす先例が多数あります．それらに学んだ結果，マイコンを上手に選び，ファームウェアを頑張れば，2，3 の IC さえ省略して CPU とマイコンだけで動かせる感触が得られました．本当にできるかどうかは，やってみれば判明します．

製作例のマイコンはマイクロチップテクノロジーの PIC18F47Q43（図 2）です．電子工作でよく使われる PIC 系の比較的新しい製品で，速度こそ中庸ですが，

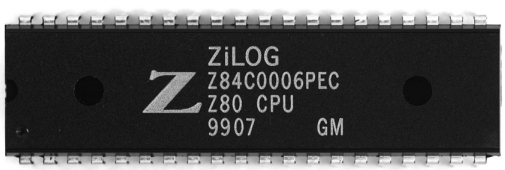


図 1 ●現在でも入手可能な Z80

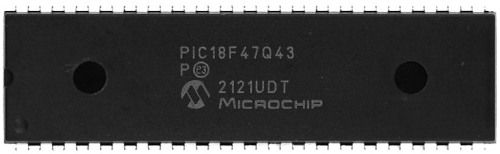


図 2 ●多彩なモジュールを内蔵した PIC18F47Q43

圧倒的に安く，適量のメモリと多彩なモジュールを備えます．中でもモジュールは，工夫しだいで苦境を切り抜ける武器となり，ゲームを盛り上げます．

この種の開発ではファームウェアの書き込み装置や各種の測定器をつないだり外したりします．終盤には端末をつないで動作確認をします．製作例は，そうした作業が便利なように，それらの端子を備え，信号名を印刷した，専用のプリント基板に組み立てました．これを EMUZ80 と呼ぶことにします（図 3）．

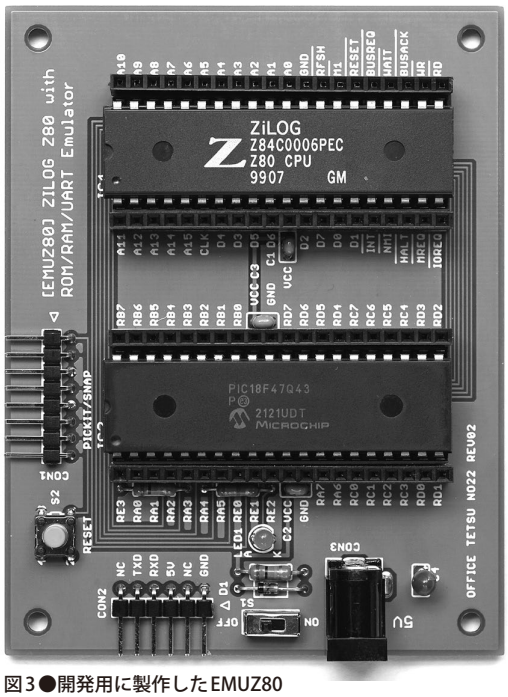


図 3 ●開発用に製作した EMUZ80

EMUZ80 は，現在，いちおう動作しています．Z80 の周辺に IC は PIC18F47Q43 しかなく，部品数はもう減りません．部品代も総額 1000 円強です．だからといって，これが結論ではありません．より効果的なピンをより単純な手順で制御し，より高速に動かすことができたら，それがより正しい結論です．

これから，EMUZ80 の Z80 が PIC18F47Q43 で動く仕組みを述べます．それは無限に考えられるやりかたのうち成功した一例です．エンジニアなら仕事の息抜きに，週末エンジニアだったら腕試しとして，Z80 がより速く動く改善案を考えながら読んでくだされば，きっと楽しい時間を過ごせるものと思います．

単純で明快な回路に最大限の可能性を込める

EMUZ80 の回路は，概略，Z80 と PIC18F47Q43 のピンを 1 対 1 でつないだ格好になります（図 4）．設計の課題は，どれとどれをつなぐかに絞られます．とりわけ PIC18F47Q43 の側は，ファームウェアが最短の手順で動き，なおかつモジュールが所定の機能を発揮できるよう，ピンを慎重に選ぶ必要があります．

PIC18F47Q43 のピンはポート A ～ E に属します．Z80 のアドレスバスとデータバスは，このポートのグループにつなぐのが合理的です．ポート A は何かと便利なので温存し，ポート E は不便で使えませんから，アドレスバスはポート B と D で読み，データバスはポート C につないで既定値を入力とします．

ポート A のピンは，さまざまなモジュールと関係しています．Z80 のピンのうち，ゆくゆく何らかのモジュールで制御したくなりそうなものは，とりあえずポート A につないでおくのが賢明です．実際のところそれをどんな風に使うかは，ファームウェアを書く段階で，順次，決定していこうと思います．

ポート E はピンが 4 本しかない上に機能が単純な入出力に限られるため活躍が期待できません．RE3 は PIC18F47Q43 を手動リセットできる唯一のピンであり，その目的に使っていることだけが必然的な設計です．あとは，ポート A につなげなかった Z80 のピンをしかたなくこちらへつないでいます．

RE0 に Z80 の $\overline{WR}$ をつないだのは単純にそんな事情です．RE1 は Z80 をリセットする $\overline{RESET}$ につながっていて，EMUZ80 の起動時に 1 回，活躍の機会が与えられます．RE2 は Z80 へ割り込み要求する $\overline{INT}$ につながっていますが，当面，常時 1 として割り込みを止める目的で使うことになります．

PIC18F47Q43 のピン数不足により，Z80 のマスク不能割り込みと DMA（ダイレクトメモリアクセス）

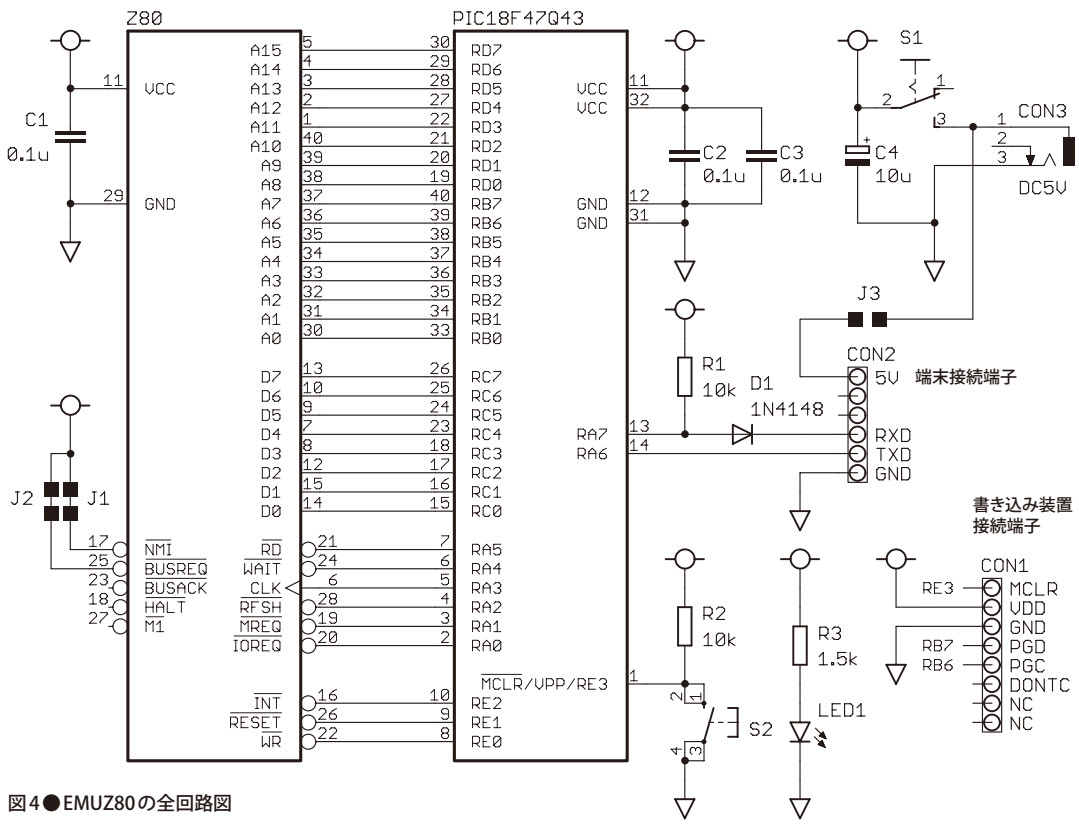


図 4●EMUZ80 の全回路図

は使いません．また，Z80 の命令読み出し動作とホールトを検出しません．これらに関係するピンは，入力なら無効に固定し，出力なら開放としました．特段の問題にはならないだろうと推測しています．

リセット信号とクロックを出して何はともあれ Z80 を始動させる

EMUZ80 は Z80 と PIC18F47Q43 の両端にピンソケットがあり，LED や多機能テスタやロジックアナライザをつないで開発途中の原始的な動きを観測することができます．そうはいっても，Z80 を始動しないことには話になりません．ファームウェアの開発はリセット信号とクロックの生成から始めます．

Z80 は $\overline{\text{RESET}}$ を下げるとリセットし，外面的にすべての動作を止めます．内面的には各部のレジスタに初期値を設定します．この動作を進行，完了させるため，CLK へ最低 3 クロックを与える必要があります．リセットは $\overline{\text{RESET}}$ を上げると解除され，Z80 はその時点のレジスタの値にしたがって始動します．

PIC18F47Q43 にしてみれば Z80 をリセットする手順は RE1 へ 0 と 1 を出力するだけなのでファームウェアは入門書の例題のように書けます．一方，クロックの生成は手段から探さなければなりません．使えそうな仕組みはいくつかあって，そのうち NCO（数値制御発振）モジュールがおあつらえ向きでした．

NCO は定数を繰り返し加算して結果がオーバーフローするごとにパルスを出す発振器です（図 5）．定数は NCO1INC に与えます．加算の頻度は NCO1CLK で選択したクロックに同期します．NCO1PFM によりパルスを分周器へ入れることができて，その場合，デューティー比 50% のクロックが生成されます．

NCO は単純な方形波しか出力しませんが，それで間に合う用途だと利点だらけです．定数と結果は 20 ビット，加算の頻度は最高 64MHz なので，広範囲の比較的正確な周波数が得られます．それを出力するピンは PPS（ペリフェラルピン選択）モジュールでポート A のどこへでも割り当てることができます．

以上を踏まえると Z80 を始動する手順はこうなります（図 6）．まず，$\overline{\text{RESET}}$ を下げて Z80 が不確定な信号で誤動作することを防ぎます．次に，NCO を動かして Z80 の CLK へクロックを出力します．そのまま何かほかの設定をしてクロックが最低 3 発出る時間を稼ぎます．それから，$\overline{\text{RESET}}$ を上げます．

Z80 のクロックは，EMUZ80 を確実に動かす観点から周波数を低めに設定しておくのが無難です．しかし，下手に書いたファームウェアでも絶対に動くほど低いと高速化を目指す熱意に水を差します．そのへん

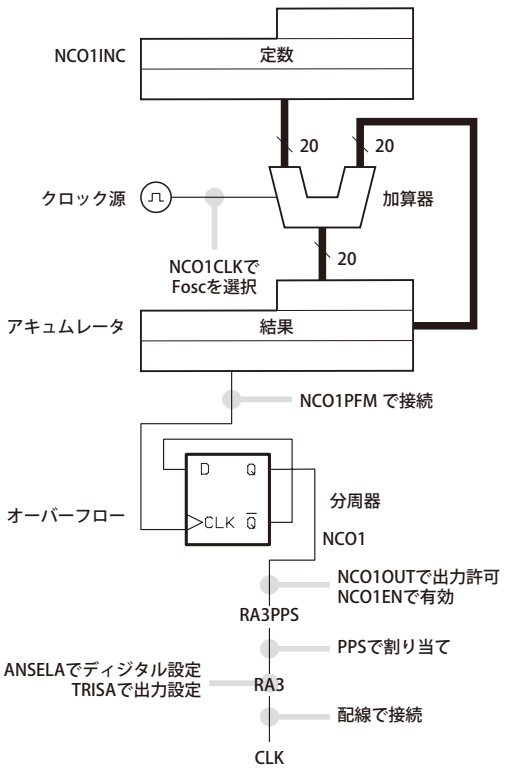


図5●NCOに関連するレジスタ

```
#define Z80_CLK 2500000UL //Max 16MHz

//RESET output pin
ANSELE1 = 0; //Disable analog
LATE1 = 0; //Reset
TRISE1 = 0; //Set as output

//CLK output pin
ANSELA3 = 0; //Disable analog
TRISA3 = 0; //NCO output pin
RA3PPS = 0x3f; //RA3 asign NCO1

//Z80 clock(RA3) by NCO FDC mode
NCO1INC = Z80_CLK * 2 / 61; //Const.
NCO1CLK = 0x00; //Clock source Fosc
NCO1PFM = 0; //FDC mode
NCO1OUT = 1; //NCO output enable
NCO1EN = 1; //NCO enable

//Z80 start
LATE1 = 1; // Release reset
```

図6●Z80を始動させる手順

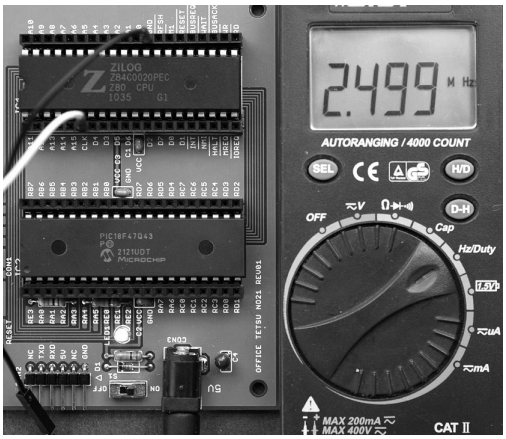


図7●Z80のクロックを確認した例

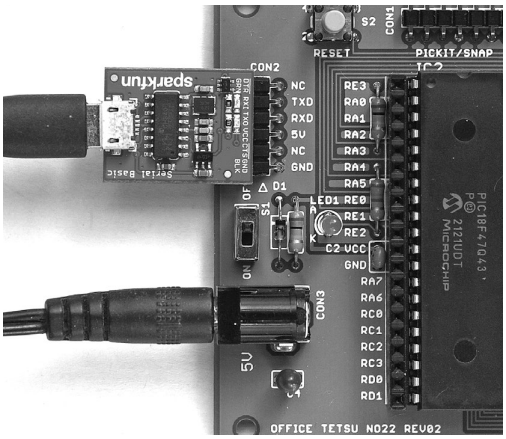


図9●USB-シリアル変換アダプタを取り付けた状態

の塩梅をみて2.5MHzに決めました．これは，無印（高速版でない）Z80の最高動作周波数です．

NCOが出す周波数はシンボルに定義してあり，変更が容易です．EMUZ80が動かなければ下げ，動いたら上げて限界を探ることができます．定数は近似式で計算していて狙った数字ぴったりにはなりませんが，多機能テスタの周波数カウンタで確認したところ，ごく小さな誤差に収まっていました（図7）．

端末インターフェースを作り操作の手段を与える

OSもライブラリもない真っさらな状態から開発を始める場合，できるだけ早い段階で端末がつながるようにするのが定石です．端末はレジスタや変数の内容を覗き見る手段となり，デバッグの効率が上がって開発を加速します．同時に，その仕組みはZ80へ提供する端末インターフェースの基礎となります．

俗にいうシリアルの端末はUART（非同期送受信器）モジュールで制御することができます（図8）．

制御のしかたに凝らなければ，ほとんどのレジスタは初期値のままで大丈夫です．その場合，通信方式は非同期シリアル，データ形式はデータ長8ビット，パリティなし，1ストップビットとなります．

端末はテレタイプのASR-33やDECのVT-100だと粋ですが，実際はUSB-シリアル変換アダプタでパソコンとつないで端末ソフトを操作することになるでしょう．EMUZ80の端末接続端子はArduino Pro Mini 5V用に安く出回っているUSB-シリアル変換アダプタとピンの配置を合わせてあります（図9）．

PIC18F47Q43をはじめとするCMOSのICは，端末をつなぐと無操作を示す信号が電源に回り込み，勝手に動き出すことが知られています．その問題の対策として，PIC18F47Q43の受信用のピンにダイオードと抵抗からなる逆流防止回路を付けています．このとても簡素な回路はネットで教えてもらいました．

通信用クロックは専用の発振器で生成します．この発振器はクロックを数えてU3BRGの値と一致するごとにパルスを出します．U3BRGは16ビットあって比較的正確な周波数が得られます．通信速度は，たい

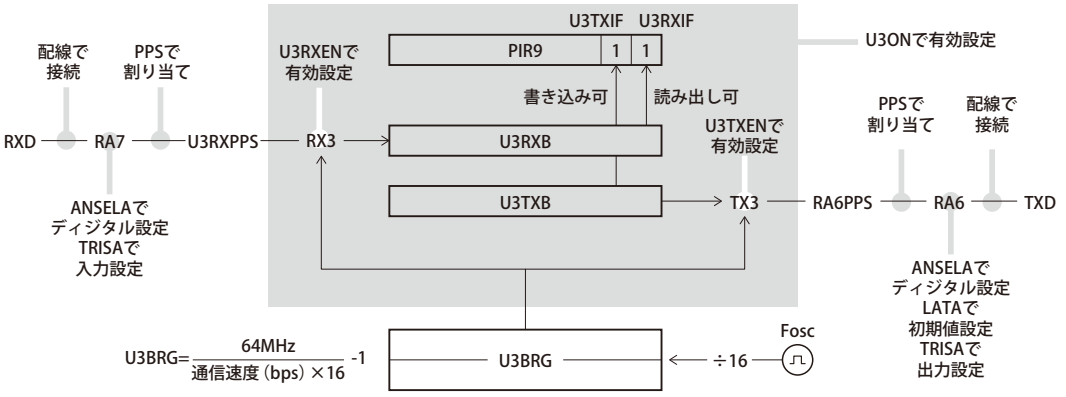


図8●UARTに関連するレジスタ

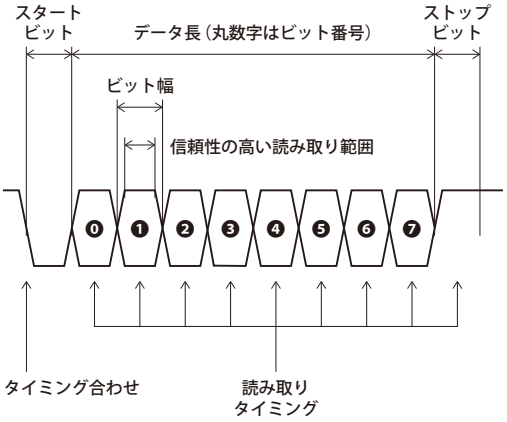


図10●UARTの波形と読み取りタイミング（受信の例）

ていの端末ソフトが初期値としている9600bpsが妥当で，その場合，U3BRGの値は416です．

俗にいうシリアルは1データごとに先頭でタイミングを合わせます．仮に通信用クロックが微妙に違っていても1データの末尾でズレの累積がビット幅を超えなければ支障がありません．特に端末をUSB-シリアル変換アダプタでつなぐ場合，通信線が短くて波形の角が立つためズレに耐性があります（図10）．

UARTの初期設定は初期値でいい部分の設定を省略すると簡単に完了します（図11）．送受信用のピンはポートAから選び，お馴染みの基本的な設定をして，PPSで当該機能へ割り当てます．UARTらしい設定は通信用クロックの周波数を決める手順くらいで，あとはそのまま各部の働きを有効にします．

```
//UART3 Receiver
ANSELA7 = 0; //Disable analog
TRISA7 = 1; //RX set as input
U3RXPPS = 0x07; //RA7->UART3:RX3

//UART3 Transmitter
ANSELA6 = 0; //Disable analog
LATA6 = 1; //Default level
TRISA6 = 0; //TX set as output
RA6PPS = 0x26; //RA6 asign TX3

//UART3 initialize
U3BRG = 416; //9600bps @ 64MHz
U3RXEN = 1; //Receiver enable
U3TXEN = 1; //Transmitter enable

//UART3 start
U3ON = 1; //Serial port enable
```

図11●UARTを初期設定する手順

文字列「hello, world」の出力とエコーバックを試す

UARTでの送受信は次に述べる3本のレジスタを使います．PIR9は受信データが読み出し可能になったときU3RXIF（ビット0）を1とし，送信データが書き込み可能になったときU3TXIF（ビット1）を1とします．その時点で，受信データはU3RXBにあります．送信データはU3TXBへ書き込みます．

EMUZ80は，これら3本のレジスタをZ80のアドレスに割り当てます．PIR9には単独のアドレスを与え，読み出し専用とします．あとの2本は別の共通なアドレスに割り当て，読み出しならU3RXB，書き込みならU3TXBを対象とします．これで，Z80に端末インターフェースが提供されます．

EMUZ80が完成したら，端末はもっぱらZ80が使い，PIC18F47Q43はレジスタの読み書きを仲介するだけです．しかし，開発の過程でレジスタや変数の内容を覗き見る手段としてPIC18F47Q43でも使えれば便利です．そのため，ファームウェアに関数putchと関数getchを仕込みました（図12）．

関数putchは端末へ1バイトのデータを出力します．関数getchは端末から1バイトのデータを入力します．マイクロチップテクノロジーが提供する開発環境は，このふたつを書くことで標準入出力ライブラリの関数群が使えます．たとえば，関数printfで端末に「hello, world」を出力することができます．

```
#include <stdio.h>

//UART3 Transmit
void putch(char c) {
  while(!U3TXIF); //Wait for flag set
  U3TXB = c; //Write data
}

//UART3 Recive
char getch(void) {
  while(!U3RXIF); //Wait for flag set
  return U3RXB; //Read data
}

void main(void) {


  printf("hello, world\r\n");
  while(1) putch(getch());
}
```

図12●端末を使うための関数putch/getchと応用例

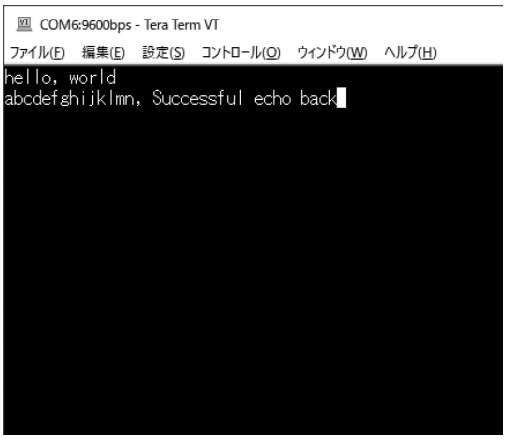


図13●文字列の出力とエコーバックのテストをした例

端末まわりの仕組みは文字列の出力とエコーバックで動作確認しました．文字列の出力は，まあ，儀礼的な手続きです．エコーバックは端末から入力した文字をそのまま出力する処理で，回路の設計からファームウェアの書きかたまで，ひとつでも誤りがあると失敗します．幸い，両方とも成功しました（図13）．

Z80の命令読み出し動作にPIC18F47Q43の応答が間に合わない

EMUZ80の本丸はPIC18F47Q43が内蔵するメモリをZ80へ開放する仕組みです．煩雑な処理となるため，実現の手法が速度に大きく影響します．ただの遅い速いなら我慢するという奥の手がありますが，ある一線を超えて遅いと暴走してどうにもなりません．あらかじめ，その一線を明らかにしておきます．

Z80がいちばん厳しい要求をする命令の読み出し動作を追ってみます（図14）．メモリの処理は$\overline{\text{MREQ}}$の立ち下りから開始されます．要求が命令の読み出しであれば，$\overline{\text{RFSH}}$が1を保ち，$\overline{\text{RD}}$が下がります．Z80は，その約1.5クロックあとに命令を取り込み，$\overline{\text{RD}}$と$\overline{\text{MREQ}}$を上げてこの動作を終了します．

AC特性の規定により，命令はZ80が取り込むより最低50n秒前に確定させておかなければなりません．2.5MHzのクロックで動く無印Z80で計算すると，$\overline{\text{MREQ}}$の立ち下がりから550n秒のうちにデータバスへ命令を乗せる必要があり，間に合わないと暴走します．これが，超えてはいけない一線です．

PIC18F47Q43がやるべきことはざっと次のとおりです．$\overline{\text{RFSH}}$が1の状態で$\overline{\text{MREQ}}$の立ち下がりを待ち，$\overline{\text{RD}}$の0を見てデータバスの方向を出力に設定し，アドレスバスの上位と下位のバイトを読み，それをROMに相当する配列の要素番号に変換し，有効な範囲だと確認したのち内容を取り出します．

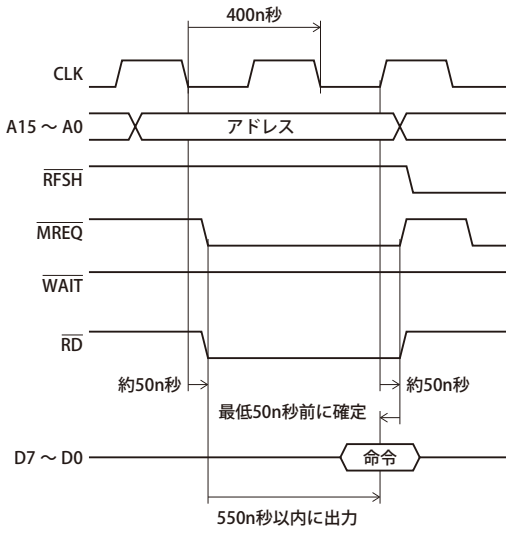


図14●命令の読み出しに関連する信号の推移

PIC18F47Q43は64MHzのクロックで動き，単純な命令は62.5n秒（4クロック），分岐や条件判定をともなう命令は最長250n秒（16クロック）を費やします．メモリの処理は分岐や条件判定を多用しますが，そうでなくても最大8命令しか使えないので，与えられた時間のうちに応答することは無理そうです．

PIC18F47Q43のファームウェアはC言語で書いてコンパイルするのが一般的であり，表向きコードの実体がわかりませんから，メモリの処理に掛かる時間を計算で予測することは困難です．一線を超えて遅いことは明白だとしても，遅さの程度を知るにはロジックアナライザで観測する必要があります（図15）．

観測用のファームウェアはメモリの処理に特化した暫定版です．大きくて誌面に掲載し切れませんが，常識的な手順を書いてあると思ってください．ROMに

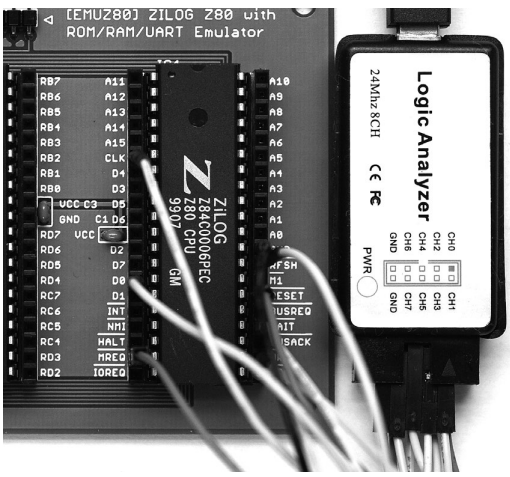


図15●ロジックアナライザを取り付けた状態

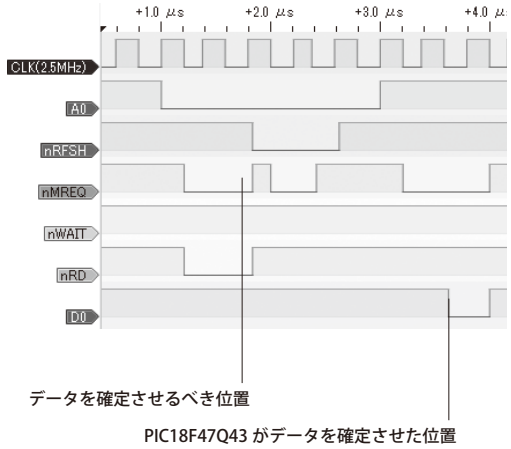


図16●Z80でJR $を実行したときの波形

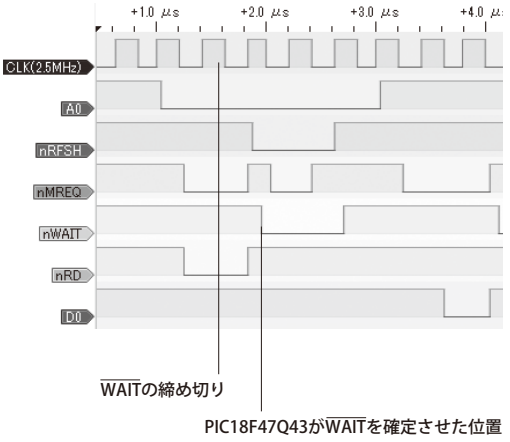


図18●ファームウェアでウェイトを掛けたときの波形

相当する配列は先頭にJR $（この命令への分岐）を持ちます．もしメモリの処理が間に合えば，Z80はリセットを解除したあとJR $を繰り返します．

JR $のコードは0x18と0xfeなので，データが確定するとデータバスのD0が下がります．ロジックアナライザでリセットを解除した直後のD0を見ると，本来，データが確定するべき位置より1800n秒も遅れて下がっています（図16）．観測する価値があるのはここまでです．以降はZ80が暴走します．

ウェイトを掛けて間に合わせたいがウェイト信号が間に合いそうにない

メモリの処理が間に合わないとき，解決のための常套手段がウェイトです．すなわち，$\overline{\text{RFSH}}$が1の状態で$\overline{\text{MREQ}}$が立ち下がったら，次のクロックの立ち下がりより少し前に$\overline{\text{WAIT}}$を下げます．以降，Z80はクロックの立ち下がりごとに$\overline{\text{WAIT}}$を調べ，1となるまで読み書き動作を引き延ばします（図17）．

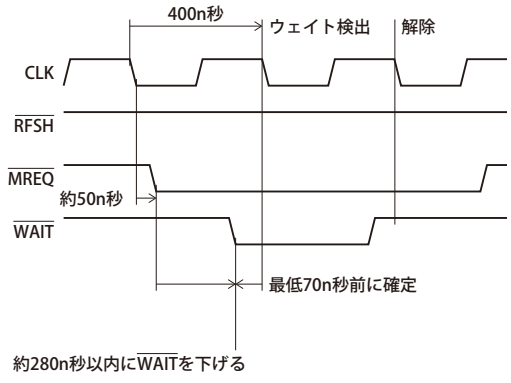


図17●ウェイトに許される時間（波形は1ウェイトの例）

AC特性の規定により，$\overline{\text{WAIT}}$はクロックの立ち下がりより最低70n秒前に確定させておかなければなりません．PIC18F47Q43は，$\overline{\text{MREQ}}$の立ち下がりを見付けたあと280n秒のうちに$\overline{\text{WAIT}}$を下げる必要があります．この時間で実行できるのは最大4命令です．いかにも無理そうですが，確認してみます．

ファームウェアは前述したメモリの処理にウェイトの処理を追加した暫定版で，Z80の信号の動きを常時監視し，入門書の例題のような手順で$\overline{\text{WAIT}}$を下げます．ロジックアナライザで観測すると，いいセンまでは行くものの，間に合っていません（図18）．読み書きは容赦なく進行し，Z80が暴走します．

限られた時間でウェイトを掛けるために，通常は，この部分に標準ロジックを使います．典型例はDフリップフロップ（74HC74）と反転ゲート（74HC04）を組み合わせたウェイト回路です（図19）．これで自動的にウェイトが掛かりますし，それはマイコンのファームウェアで解除することができます．

EMUZ80は徹底した簡素化を目指しているので標準ロジックを追加する方法はとれません．でも大丈夫，この期に及んでやっぱりダメでしたとはいいませ

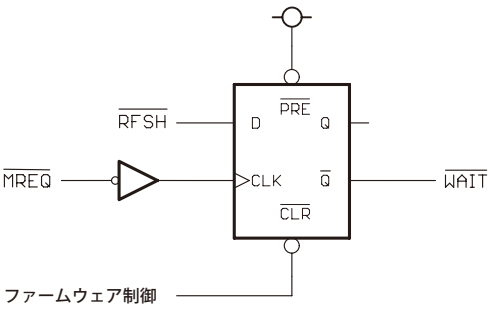


図19●標準ロジックで構成したウェイト回路の典型例

ん．PIC18F47Q43 は CLC（構成可能ロジックセル）モジュールを備えており，これにより標準ロジック並みの速度で応答するウェイト回路が作れます．

CLC のウェイト回路はファームウェアの介在なしにウェイトを掛け，ファームウェアで解除することができます．EMUZ80 のマイコンに PIC18F47Q43 を選んだ決め手は CLC があることです．これで，よく標準ロジックで構成されるウェイト回路と同じものを構成し，問題の解決をはかることにします．

CLC で構成した ウェイト回路の実力

CLC は関係するレジスタが嫌になるくらいたくさんあり，ウェイト回路をひとつ構成するのが大仕事です（図 20）．そこで，マイクロチップテクノロジーが提供する MCC（MPLAB コードコンフィギュレータ）の助けを借りました．MCC はビジュアル表示とマウス操作で C 言語のソースを生成します．

とはいえ，EMUZ80 の開発を始めた時点で MCC はまだ深刻なバグを抱えていました．それを使って細部まで作り込む度胸はなく，ウェイト回路の大筋を描いてすぐ C 言語のソースを生成し，あとは普通にエディタで仕上げました．そんな事情から，CLC は最大 8 組が使えますが，1 組だけを使っています．

CLC は 8 種類のロジックを用意していて，そのうちのひとつを CLCnCON のモードビットで選択します．ウェイト回路の中心に据えたのは 4 番，セットとリセットが付いた D フリップフロップです．便宜上，その 4 本の入力には G1 ～ G4 の OR ゲートが付き，1 本の出力にバッファがあるものとみなします．

OR ゲートと信号を接続する手順は一般論だと際限なく長い話になるため $\overline{\text{MREQ}}$ を G1 へ入れる例で述べます（図 21）．$\overline{\text{MREQ}}$ は電気的に RA1 と接続しています．RA1 は汎用入力に設定し，PPS で CLCIN0 に割り当てます．CLCIN0 は「CLC へ入れるポート A のピン」を意味する符号のひとつです．

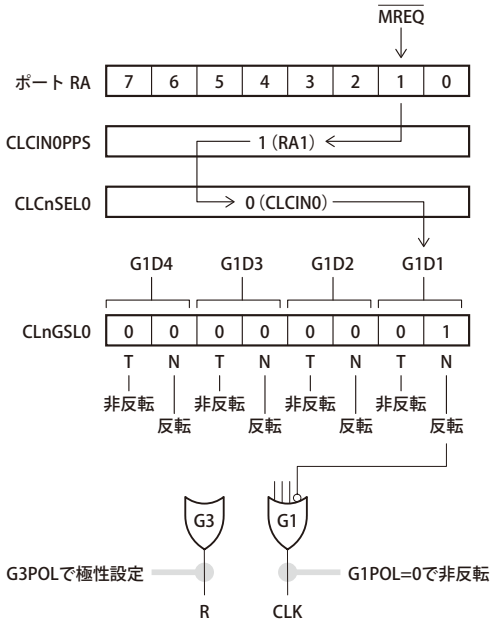


図 21●OR ゲートと信号の接続（G1 と G3 の例）

符号は CLCnSEL0 ～ CLCnSEL3 のどれかに置いておきます．その番号と極性を，OR ゲートと対応した CLnGLS0 ～ CLnGLS3 で選択します．G1 に対応するのは CLnGLS0 で，ここへ 1 を書くと，CLCnSEL0 に置いた符号が反転で接続します．なお，同様の手順で各種モジュールの出力を接続することができます．

OR ゲートの入力が無接続の場合，出力は 0 ですが，CLCnPOL またはそのビット C1POL ～ C4POL で極性を反転させると 1 になります．ウェイト回路の G3 は，一見すると無意味ですが，この操作で一時的

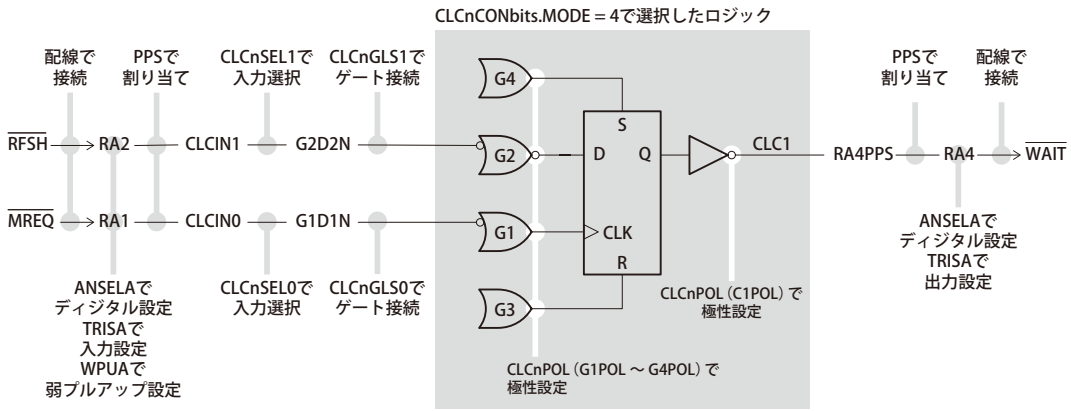


図 20●CLC のウェイト回路と関係するレジスタ

```
//MREQ(RA1) CLC input pin
ANSELA1 = 0; //Disable analog
WPUA1 = 1; //Week pull up
TRISA1 = 1; //Set as input
CLCIN0PPS = 1; //RA1->CLCIN0;

//RFSH(RA2) CLC input pin
ANSELA2 = 0; //Disable analog
WPUA2 = 1; //Week pull up
TRISA2 = 1; //Set as input
CLCIN1PPS = 2; //RA2->CLCIN1;

//WAIT(RA4) CLC output pin
ANSELA4 = 0; //Disable analog
LATA4 = 1; //Default level
TRISA4 = 0; //Set as output
RA4PPS = 1;  //RA4->CLC1;

//CLC logic configuration
CLCSELECT = 0x0; //Use only 1 CLC
CLCnPOL = 0x82; //G2 & OUT inverted

//CLC data inputs select
CLCnSEL0 = 0; //D-FF CLK assign RA1
CLCnSEL1 = 1; //D-FF D assign RA2
CLCnSEL2 = 127; //D-FF S assign none
CLCnSEL3 = 127; //D-FF R assign none

//CLCn gates logic select
CLCnGLS0 = 1; //Connect G1D1N
CLCnGLS1 = 4; //Connect G2D2N
CLCnGLS2 = 0; //Connect none
CLCnGLS3 = 0; //Connect none

CLCDATA = 0; //Clear all CLC outs
CLCnCON = 4; //Select D-FF

//Falling edge inturrupt
CLCnCONbits.INTN = 1; //Int. enable
CLCnCONbits.EN = 1; //CLC enable

//Release wait (D-FF reset)
G3POL = 1;
G3POL = 0;
```

図22●CLCを構成する手順

に1を出力し，Dフリップフロップをリセットして，ウェイトを解除する働きがあります．

CLCの出力は接続の経路がひとつに限られているため設定が比較的簡単です．出力の符号はCLC1で，

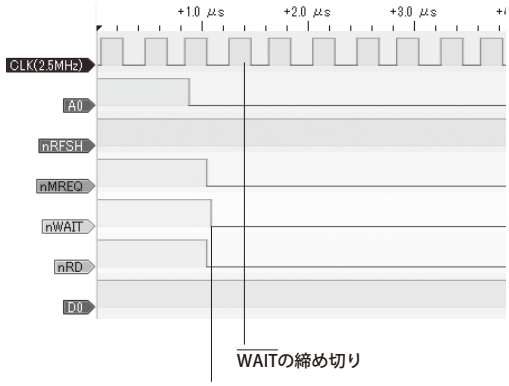


図23●CLCでウェイトを掛けたときの波形

これをCLCnPOLのビット7で反転させ、PPSで実体と結び付けます．出力をピンへつなぐ場合，実体はポートAのどれかです．なお，同様の手順で各種モジュールの入力へつなぐこともできます．

CLCはPIC18F47Q43に埋め込まれたPLD（プログラマブルロジックデバイス）と考えられますし，各種のモジュールとつなげばRaspberry Pi Picoのステートマシンのようにも動かせます．CLCがこれらと決定的に違うのは構成をファームウェアの一部としてごく普通のC言語で書けることです（図22）．

CLCのウェイト回路が期待どおりに動くかどうかはロジックアナライザで調べました．ファームウェアはZ80を始動してすぐウェイトを掛けるだけの暫定版です．ウェイトを解除する手順はあえて書いていないので，うまくウェイトが掛かったら先頭の命令の取り込み動作を継続することが予想されます．

ロジックアナライザの波形を見ると$\overline{\text{MREQ}}$が立ち下がった直後に$\overline{\text{WAIT}}$が下がり，締め切りまで十分な余裕をもってウェイトが掛かります（図23）．以降は$\overline{\text{RD}}$が0，A0も0の状態を維持し，先頭の命令を取り込もうとしていることがわかります．まさに，期待どおりの動作ということができます．

ウェイトを掛けるだけにとどまらないウェイト回路のさまざまな効能

CLCのウェイト回路が$\overline{\text{MREQ}}$の立ち下がりに反応して素早く$\overline{\text{WAIT}}$を下げたことは，ただウェイトがうまく掛かるという以上の意義があります．たとえば，メモリの処理を$\overline{\text{MREQ}}$の立ち下がりではなく$\overline{\text{WAIT}}$の立ち下がりから始めてZ80のリフレッシュが及ぼす悪影響を避けることができます．

Z80は命令を取り込んだ直後のバスが空く時間を利用してDRAMのリフレッシュをします．この機能は

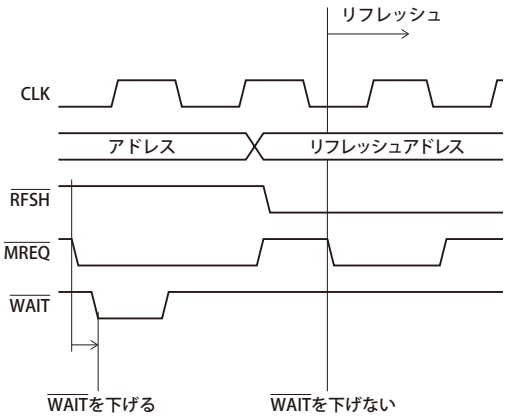


図24●リフレッシュに関連する信号の推移

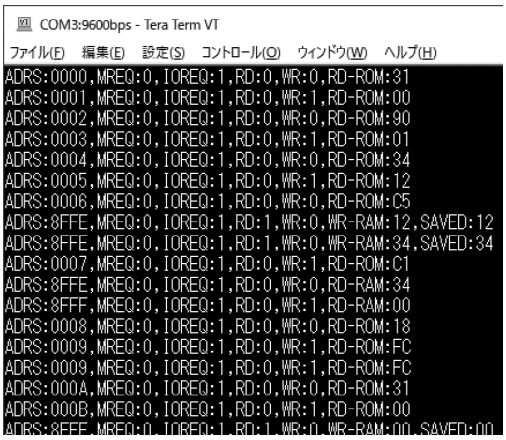


図26●printf デバッガの表示例

数 K バイトの RAM を DRAM で実現していた時代に重宝されました．現在は PIC18F47Q43 でさえ 8K バイトの RAM を内蔵していてリフレッシュが不要です．正しくいえば，反応してはいけません．

Z80 の信号を見るとリフレッシュの期間は $\overline{\text{RFSH}}$ が下がっています（図 24）．そこで EMUZ80 のウェイト回路（標準ロジックの回路例も同じです）は，$\overline{\text{RFSH}}$ が 1 の状態で $\overline{\text{MREQ}}$ が立ち下がったときだけ $\overline{\text{WAIT}}$ を下げ，$\overline{\text{RFSH}}$ が 0 の状態で $\overline{\text{MREQ}}$ が立ち下がっても反応しないように構成してあります．

$\overline{\text{RFSH}}$ と $\overline{\text{MREQ}}$ と $\overline{\text{WAIT}}$ がそのような関係で動くことは Z80 のかわりにスイッチと LED を取り付けて確認しました（図 25）．ファームウェアは CLC でウェイト回路を構成しておいてウェイトが掛かったら 1 秒後に解除する暫定版です．この格好悪い検査装置は CLC を構成する各段階で役立ってくれました．

ウェイト回路がリフレッシュに反応しない構成なので，PIC18F47Q43 は，その信号を利用して正しい期間にメモリの処理をすることができます．理屈をいうと開始のタイミングが微妙に遅れますが，問題にするほどでないことが，すでに判明しています．あとの処理がラクになる分，差し引きで速くなるでしょう．

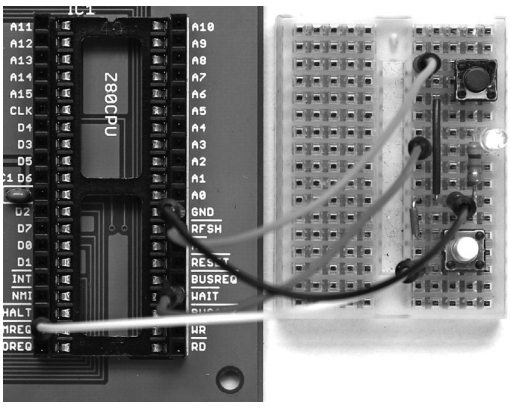


図25●ウェイト回路に取り付けたスイッチと LED

もうひとつ，Z80 は，いったんウェイトが掛かったら解除するまで読み書きの状態を維持することも判明しています．その間，PIC18F47Q43 は動作を続けるので，レジスタや変数の内容を拾って関数 printf で表示する，俗称 printf デバッガが動きます（図 26）．これは先々の開発で頼もしいツールとなります．

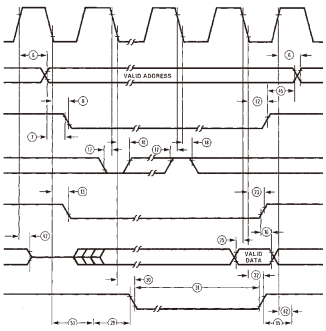
以上で，Z80 の周辺に必要な機能と問題になりそうな部分の対策があらかた揃いました．残された課題は PIC18F47Q43 が内蔵しているメモリを Z80 へ開放する仕組みだけです．もっとも，それは EMUZ80 の核心であり，あと少しの誌面ではまとめられません．今回は，ここで一段落つけようと思います．

すでに実現した機能や対策は，今後の開発で要所へ散りばめることになります．あるものは時系列の適切な位置に配置し，あるものはここぞというところで実行します．いずれにしろ，断片的な動作を確認したからといって，それで終わりとはなりません．役割を把握し，体系的に整理しておくことが必要です．

次回は，現状の全体像を明確にし，説明が足りなかったかもしれないところを補足した上で，メモリの処理を組み立て，EMUZ80 を完成させます．さらに，伝統的な Z80 のコンピュータにならって BASIC を動かし，ベンチマークテストを実施して，安定性や速度の水準を調べます．どうぞご期待ください．

なお，EMUZ80 の基本的な操作方法を説明した技術資料，ファームウェアの MPLAB X IDE プロジェクト，BASIC のソースとコード，プリント基板の頒布情報などを用意し，私のブログの記事『EMUZ80 が完成』（リンクをクリックすると移動します）でご案内しています．必要に応じ，ご覧ください．

すずき・てつや



コンピュータの部品数と製作費は
どこまで減らせるだろうか？

訳あって「トラ技」の体裁です

Z80とPIC18F47Q43だけで動く

EMUZ80の設計と製作

うっかり超強力printfデバッガ（笑）まで実現しちゃいました!

後編➲中枢の機能

鈴木 哲哉

古典的なコンピュータの回路を現在の技術で作り直してみる

かつてコンピュータの自作マニアは部品がぎっしり詰まった基板を巧みに組み立てました．それはもう見事なものですが，職場に元自作マニアの先輩がいて過去の製作物を繰り返し自慢されると3回目くらいから受け答えに困ります．EMUZ80は，そんなときの自慢返しに打って付けなコンピュータです（図1）．

EMUZ80はザイログのZ80と少数の安価な部品で構成されています．Z80は現役の製品ですが，発売が1976年ですから，先輩のCPUと同年代か，ことによっては先輩のCPUもZ80です．それがスカスカの基板で動いているところを見たら，時間の流れとはこういうものかと認識を新たにしてくれるでしょう．

EMUZ80に乗っている能動的な部品はZ80とマイクロチップテクノロジーのPIC18F47Q43だけです．かつてZ80の周辺に使われていた部品は，あらかたPIC18F47Q43が代替しています（図2）．そのうちメモリシステムを除く部分の仕組みは，前回すでに説明していますが，改めて要点をまとめておきます．

リセット回路はRE0を汎用出力に設定してZ80の$\overline{\text{RESET}}$へつないであります．PIC18F47Q43は起動してすぐRE0に0を出力してZ80をリセットし，ほかに必要な処理をして所定の時間を潰したのち，1を出力して解除します．Z80は所定の時間で各部のレジスタへ初期値を設定し，解除した直後に始動します．

クロック源はPIC18F47Q43のNCO（数値制御発振）モジュールで作りました．NCOは単純な方形波しか出しませんが，広範囲の比較的正確な周波数が得られます．これでZ80のCLKへ2.5MHzを与えます．周波数は変更が容易なので，Z80が動かなければ下げ，動いたら上げて限界を探ることができます．

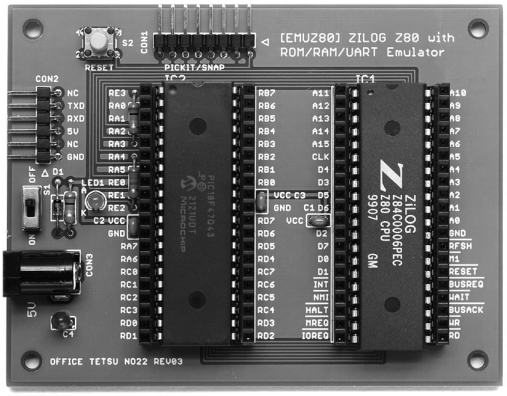


図1●開発用に製作したEMUZ80

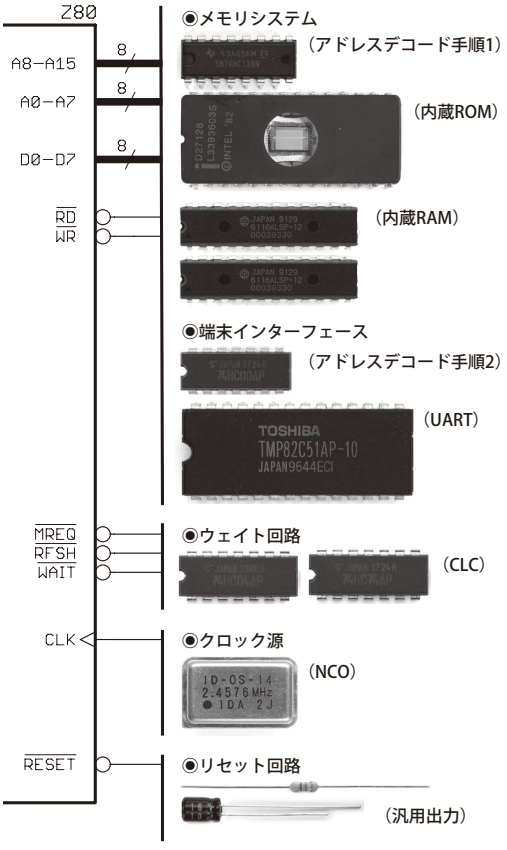


図2●PIC18F47Q43が代替する部品（括弧は代替機能）

PIC18F47Q43 の速度は現在の尺度で中庸ですが，Z80 の信号に逐一反応できる水準には届いていないため，Z80 にウェイトを掛けておいて処理します．ウェイト回路は PIC18F47Q43 の CLC（構成可能ロジックセル）モジュールで構成しました．メモリシステムと関係が深いので，詳細はのちほど言及します．

端末インターフェースは PIC18F47Q43 の UART（非同期送受信器）モジュールで作りました．送受信に関係するレジスタは，U3TXB（送信バッファ），U3RXB（受信バッファ），PIR9（フラグ）の 3 本です．これらをメモリシステムの一部に組み込むことで，Z80 から端末を制御できるようになります．

EMUZ80 を Z80 のコンピュータとして動作させるには，あとひとつメモリシステムが必要です．この部分で PIC18F47Q43 は，Z80 の求めに応じ，内蔵のメモリやレジスタを開放します．実現の方法はいろいろあって，現時点では最適解を判断しかねるため，さしあたり無難に動いてわかりやすい一例を示します．

CLC の D フリップフロップで構成したウェイト回路の動作原理

メモリシステムの構想は，Z80 のいわゆる M1 サイクルに当てはめて実現可能性を検討しました（図 3）．M1 サイクルは通常より半クロック短い命令読み出し時間と，反応してはいけない DRAM のリフレッシュが連続します．いわば厄介ごとの宝庫なので，ここがうまくいけば，ほかもきっとうまくいきます．

EMUZ80 のメモリシステムはウェイト回路で M1 サイクルをはじめとする各種期間の問題を乗り切ります．ウェイト回路の第一義的な目的はメモリの読み書きに必要な時間を稼ぐことです．ウェイトが掛かっていれば時間に厳しい M1 サイクルの命令読み出しを通常どおりの手順で実行することができます．

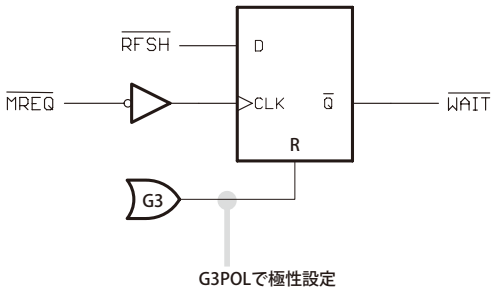


図 4●CLC で構成したウェイト回路の等価回路

ウェイト回路はまた M1 サイクルのリフレッシュを遣り過ごす手段となります．この期間はメモリの読み書きをしないためウェイトを掛けない仕組みになっています．ちょうどいいので，ウェイトをきっかけにメモリの処理を始めてリフレッシュを完全に無視します．ここが勘どころですから，詳細を説明します．

CLC のウェイト回路は D フリップフロップを中心に据えた比較的単純な構成（図 4）で次の 3 点を実現します．Z80 がメモリの処理を要求したらウェイトを掛けること，ただし，要求が DRAM のリフレッシュだったらウェイトを掛けないこと，そして，必要な処理をしたのちウェイトを解除できることです．

D フリップフロップそのものは，CLK の立ち上がりで D の値を記憶し，反転して $\overline{Q}$ へ出す回路です．

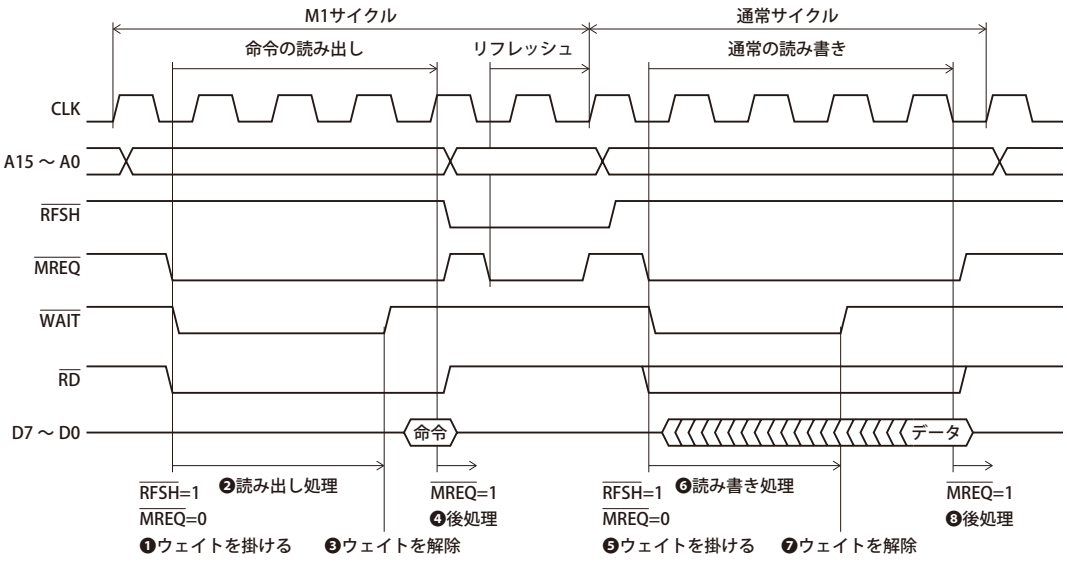


図 3●メモリシステムの動作（メモリの処理が 1 ウェイトで完了すると仮定した例）

$\overline{\text{MREQ}}$ の反転信号を CLK へ入れ，D に $\overline{\text{RFSH}}$，$\overline{\text{Q}}$ に $\overline{\text{WAIT}}$ をつなぐと，Z80 がメモリの処理を要求したとき，通常の読み書きでウェイトが掛かり，リフレッシュの期間はウェイトなしの信号を出します．

D フリップフロップは R が立ち上がるとリセットし，$\overline{\text{Q}}$ が 1 を出してウェイトを解除します．R には OR ゲートの G3 がつながっていて，通常その出力は 0 ですが，ファームウェアで極性を反転すると 1 になります．必要な処理を終えたら，この方法で R を立ち上げ，ウェイトを解除することができます．

ウェイト回路がリフレッシュに反応しないので，メモリシステムはウェイトを起点にメモリの処理を始めます．ウェイトが掛かっているため，処理に時間の制限はありません．完了したらウェイトを解除します．おそらく，何らかの後処理が必要になるでしょう．その起点は $\overline{\text{MREQ}}$ の立ち上がりが適当です．

処理の性質で使い分ける ポーリングとベクタ割り込み

EMUZ80 は $\overline{\text{WAIT}}$ の立ち下がりでメモリの処理を開始し，$\overline{\text{MREQ}}$ の立ち上がりで後処理を開始します．信号の変化を捉えて処理へ移る方法はポーリングとベクタ割り込みがあります．もうひとつ旧式の割り込みがありますが，コンフィグレーションビットの初期値が無効の設定なので，ないものとみなします．

応答が速いのはポーリングです．ポーリングは信号の状態をファームウェアで常時監視します．この間，ほかのことができないため，信号の変化を予測して直前から待ち伏せし，なるべく短い時間で済ませます．まだ手探り状態の EMUZ80 で確実な予測は無理なので，比較的わかりやすい後処理にのみ応用します．

ポーリングは，それと意識するかどうかはともかく日常的にやっているでしょうから，わざわざ説明するのが気恥ずかしい思いです．$\overline{\text{MREQ}}$ の立ち上がりで後処理へ移る典型的な手順は，$\overline{\text{MREQ}}$ とつながった RA1 が確実に 0 と予測される位置へ while(!RA1); を記述し，続けて後処理を記述します（図 5）．

一方，$\overline{\text{WAIT}}$ の立ち下がりはベクタ割り込みで捉えてメモリの処理を開始します．ベクタ割り込みは

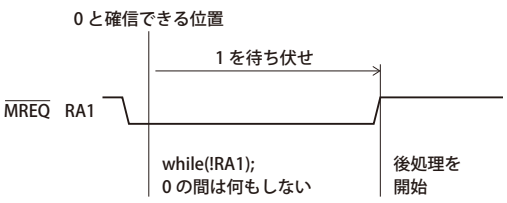


図5●ポーリングで$\overline{\text{MREQ}}$の立ち上がりを調べる手順

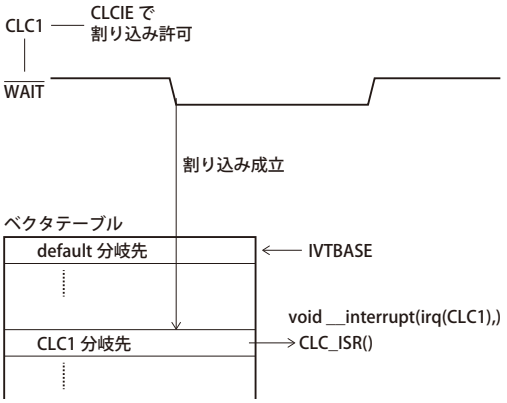


図6●ベクタ割り込みの構造

VIC（ベクタ割り込み制御）モジュールのベクタテーブルにしたがい，割り込み要因ごとに専用の関数を呼び出します（図 6）．$\overline{\text{WAIT}}$ の立ち下がりでは CLC と結び付けた関数が呼び出されることになります．

ベクタテーブルの内容は，マイクロチップテクノロジーが提供する開発環境だと，呼び出される関数のほうに記述子 __interrupt を付けることで自動的に設定されます．関数の名前は自由です．そもそもほかの位置から参照されません．CLC の割り込みは，割り込み要因にちなみ，関数 CLC_ISR と名付けました．

ベクタ割り込みの設定は，最低，割り込み要因の割り込みを許可すれば完了し，しいていうなら，ほかにベクタテーブルの位置を指定できます（図 7）．ベクタテーブルはロックされていて，変更の前後で解除と再ロックが必要です．面倒なわりに利点が少ないため，初期値（0x000008）のままでもいいでしょう．

```
//IVT setup sequence (Example)
IVTLOCK = 0x55;
IVTLOCK = 0xAA;
IVTLOCKbits.IVTLOCKED = 0x00; //Unlock

IVTBASE = 0x000008; //Setup (Default)

IVTLOCK = 0x55;
IVTLOCK = 0xAA;
IVTLOCKbits.IVTLOCKED = 0x01; //Lock

//CLC VI enable
CLC1IF = 0; //Clear interrupt flag
CLC1IE = 1; //Enable CLC1 interrupt

//Global interrupt enable
GIE = 1;
```

図7●ベクタ割り込みを設定する手順

ベクタ割り込みが旧式の割り込みより優れた点は割り込み要因ごとに専用の関数を呼び出すことですが，現状，EMUZ80 の割り込みは CLC だけなので，そこは事実上無意味です．EMUZ80 がうまく動いたあと改良を試みる段階で，さらにいくつかの割り込みを使うことになったら，便利さを実感するでしょう．

PIC18F47Q43 が Z80 に提供するメモリシステムの構造

PIC18F47Q43 は 128K バイトの ROM（プログラムメモリ）と 8K バイトの RAM（データメモリ）を内蔵しています．ファームウェアの開発が進むにつれて判明することですが，PIC18F47Q43 自身が使うメモリはガッカリするほど少量です．おかげで，Z80 に実用上の不足がない容量を開放することができます．

ネットに散在する Z80 のアプリケーションを移植しやすいように，Z80 から見えるメモリマップは往年のコンピュータによくある構成と似せました（図 8）．すなわち，ROM はアドレスの先頭から 16K バイト，RAM は 0x8000 から 4K バイト，端末の制御に使うレジスタは 0xE000 と 0xE001 に配置します．

往年のコンピュータと異なるのは端末関連のレジスタが入出力マップではなくメモリマップにあることです．もし入出力マップに割り当てると Z80 が $\overline{\text{IOREQ}}$ を立ち下げた時点でもウェイトを掛ける必要がありますが，CLC のウェイト回路にはもう $\overline{\text{IOREQ}}$ を入れる余地がなく，それはできませんでした．

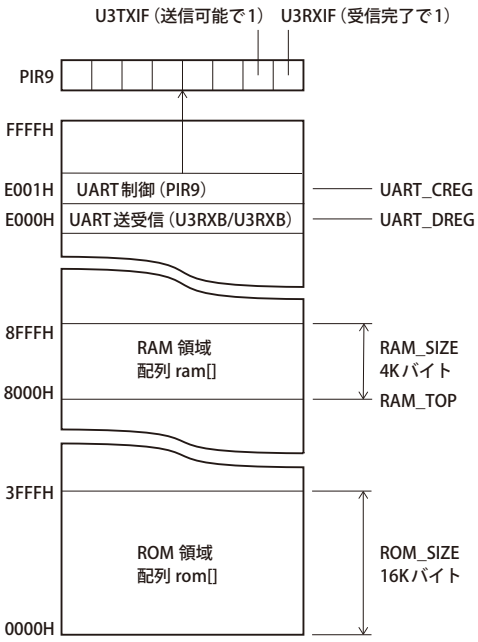


図8●Z80から見たメモリマップ

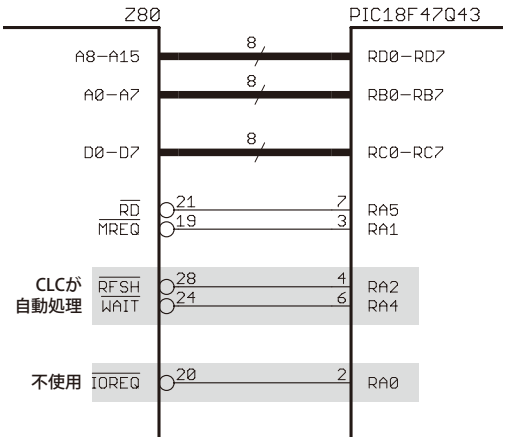


図9●メモリシステムに関係するピン

苦し紛れの産物とはいえ読み書きの対象がすべてメモリマップにあることは処理の手順をわかりやすくします．関係する信号は，アドレス（A0 ～ A15），データ（D0 ～ D7），$\overline{\text{RD}}$，$\overline{\text{MREQ}}$ に絞られます．ほかの信号は CLC のウェイト回路が監視して自動的に反応するか，まったく使用しません（図 9）．

PIC18F47Q43 が内蔵するメモリのうち Z80 に開放する容量は配列 rom と配列 ram に確保しておきます（図 10）．配列 rom はアプリケーションを置く場所

```
#define ROM_SIZE 0x4000 //16K bytes
#define RAM_SIZE 0x1000 //4K bytes
#define RAM_TOP 0x8000 //RAM top address
#define RAM_END RAM_TOP+RAM_SIZE
#define UART_DREG 0xE000 //Data REG
#define UART_CREG 0xE001 //Control REG

//Z80 ROM equivalent
const unsigned char rom[ROM_SIZE] = {
    0x18, 0xfe, // JR $
};

//Z80 RAM equivalent
unsigned char ram[RAM_SIZE];

//Address Bus
union {
    unsigned int w; //16 bits Address
    struct {
        unsigned char l; //Address low
        unsigned char h; //Address high
    };
} ab;
```

図10●Z80のROM、RAM、アドレスに相当する定義

ですが，当面，JR $（この命令への分岐）を置いて動作確認に備えます．要所のアドレスやメモリの容量は容易に変更できるようシンボルに定義しました．

PIC18F47Q43 は Z80 が出力する 16 ビットのアドレスを 8 ビットのポート B と D で入力します．アドレスを保持する変数は 8 ビットずつ 2 回の入力で 16 ビットとなるように宣言しています．この変数は通用範囲がグローバルです．現時点でそうする必要はありませんが，デバッグと将来の改良に備えています．

関数 CLC_ISR が一手に引き受けるメモリシステムの読み書き手順

関数 CLC_ISR は $\overline{\text{WAIT}}$ の立ち下がりで呼び出され，Z80 の要求を調べてメモリマップの読み書きを実行します．呼び出された時点の状況を整理しておきます（図 11）．Z80 の要求を表す信号はすべて確定し，ウェイトが掛かっていて変化しません．データバスは既定値により PIC18F47Q43 から見て入力です．

処理の手順は次のとおりです（図 12）．Z80 の要求が何であれアドレスは必要なので，まずそれを取得します．次に Z80 の $\overline{\text{RD}}$ を調べ，PIC18F47Q43 から見て入力か出力かを判定します．この判定に $\overline{\text{WR}}$ は使いません．$\overline{\text{WR}}$ は確定が遅いため，Z80 のファミリー IC（たとえば Z80 SIO）でさえ使っていません．

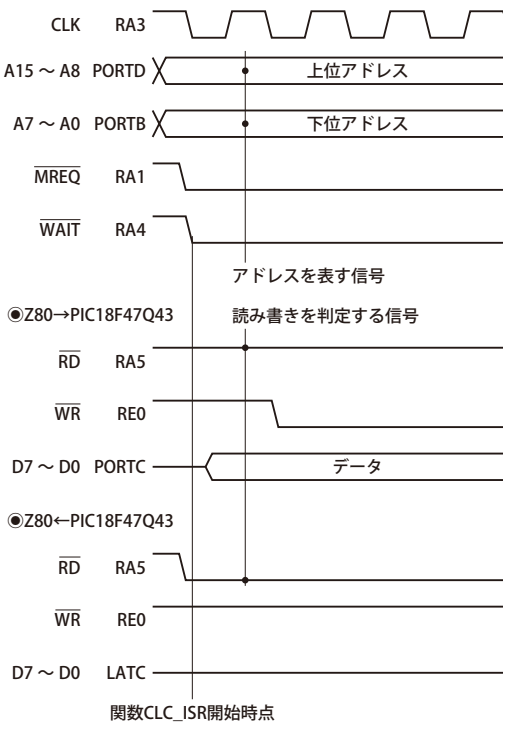


図 11●関数 CLC_ISR が呼び出された時点の状況

```
//Called at WAIT falling edge
void __interrupt(irq(CLC1),) CLC_ISR(){
  CLC1IF = 0; //Clear interrupt flag

  ab.h = PORTD; //Read Address high
  ab.l = PORTB; //Read Address low

  //Z80 memory write cycle
  if(RA5) {
    if((ab.w >= RAM_TOP) &&
      (ab.w < RAM_END)) //RAM
      ram[ab.w - RAM_TOP] = PORTC;
    else if(ab.w == UART_DREG) //U3TXB
      U3TXB = PORTC; //U3 TX buffer
    //Release wait (D-FF reset)
    G3POL = 1;
    G3POL = 0;
    return;
  }

  //Z80 memory read cycle
  TRISC = 0x00; //DATA-BUS -> Z80
  if(ab.w < ROM_SIZE) //ROM
    LATC = rom[ab.w];
  else if(ab.w < RAM_TOP) //Empty
    LATC = 0xff; //Invalid data
  else if(ab.w < RAM_END) //RAM
    LATC = ram[ab.w - RAM_TOP];
  else if(ab.w == UART_CREG) //PIR9
    LATC = PIR9; //U3 flag
  else if(ab.w == UART_DREG) //U3RXB
    LATC = U3RXB; //U3 RX buffer
  else //Empty
    LATC = 0xff; //Invalid data
  //Release wait (D-FF reset)
  G3POL = 1;
  G3POL = 0;

  //Post processing
  while(!RA1);
  TRISC = 0xff; //Set as input
}
```

図 12●関数 CLC_ISR の内容

$\overline{\text{RD}}$ が 1 だと Z80 の要求は書き込み，PIC18F47Q43 から見て入力です．データは少し遅れて Z80 がデータバスに乗せます．関数 CLC_ISR が呼び出されるまで適度な遅れがあるので，データバスにあたるポート C にはすでにデータが存在します．以降，アドレスをもとに RAM とレジスタの処理を振り分けます．

アドレスがRAMの範囲にあったら配列ramの要素番号に換算してデータを保存します．書き込み用レジスタを指している場合はPIC18F47Q43のU3TXBへ保存します．ROMの範囲や読み出し用レジスタの場合は，そもそも書き込むことができないので無視します．最後にウェイトを解除して戻ります．

$\overline{RD}$が0だとZ80の要求は読み出し，PIC18F47Q43から見て出力です．データはPIC18F47Q43が持っていて，出力先はデータバスにあたるポートCです．データバスは既定値が入力ですから，事前に出力へ切り替えておきます．以降，アドレスをもとにROMとRAMとレジスタの処理を振り分けます．

アドレスがROMの範囲にあったら配列romの要素番号に換算して該当するデータをデータバスに出力します．RAMの範囲だったら配列ramで同様に処理します．レジスタを指している場合，アドレスによりPIC18F47Q43のPIR9またはU3RXBを出力します．無効なアドレスは0xFFを出力しておきます．

それからウェイトを解除して最後のもうひと仕事をします．データバスは入力が既定値ですが，出力に切り替えたままです．この状態はZ80が読み出しを終えるまで維持する必要があり，まだ戻せません．状況が整うのを待って入力に切り替えるのが，メモリシステムの重要な後処理ということになります（図13）．

データバスの方向が問題になるのはZ80が読み出しの次に書き込みをする場合です．PIC18F47Q43が出力にしたままZ80が書き込みを始めるとデータが衝突します．少なくともどちらかは入力でなければなりません．まさか切り替えを忘れることはないでしょうが，切り替えが間に合わない事態は想定されます．

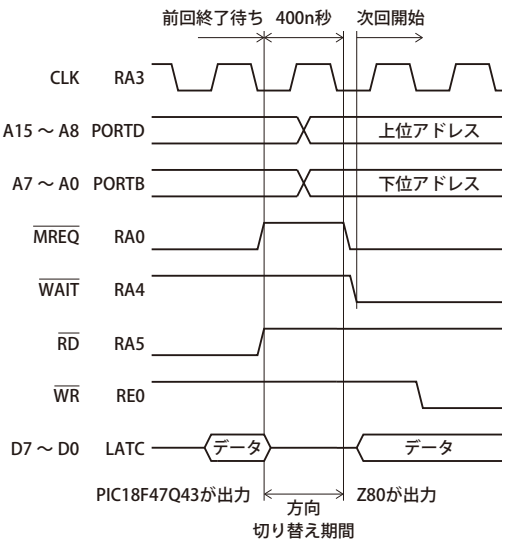


図13●メモリシステムの後処理

PIC18F47Q43がデータバスを入力に切り替えてもいい期間は限られます．Z80が読み込みを完了して$\overline{MREQ}$を上げた直後から次に$\overline{MREQ}$を下げるまで，最短で1クロック，2.5MHzだと400n秒です．ウェイトは解除しているので（解除しないと$\overline{MREQ}$が上がりません），これが正味の時間です．

一刻も早く反応するため$\overline{MREQ}$の立ち上がりをポーリングで捉えて後処理へ移ります．後処理とはいうものの，現状では与えられた時間でどのくらいのことができるか定かでないため，最低限の必要な処理にとどめます．すなわち，データバスを入力へ切り替え，あとは何もせずに関数CLC_ISRから戻ります．

ロジックアナライザとprintfデバッガで動作確認

EMUZ80のファームウェアはソースの記述からプログラムメモリへの書き込みまで全工程をマイクロチップテクノロジーの開発環境で行いました．統合開発環境はMPLAB X IDE，コンパイラはXC8の無償版です．XC8は無償版でもレベル2までの最適化をしてコンパイルすることができます．

書き込み装置はSnapを使いました．書き込みにあたってはZ80を一時的に取り外してください．そうでないとアドレスバスで衝突が起きます。大きな声ではいえませんがザイログのZ80は衝突に強いようです．正直に白状しますと小さな問題は全部をつないだまま書き直し，その場で修正しました（図14）．

図14●Snapとロジックアナライザを取り付けた状態

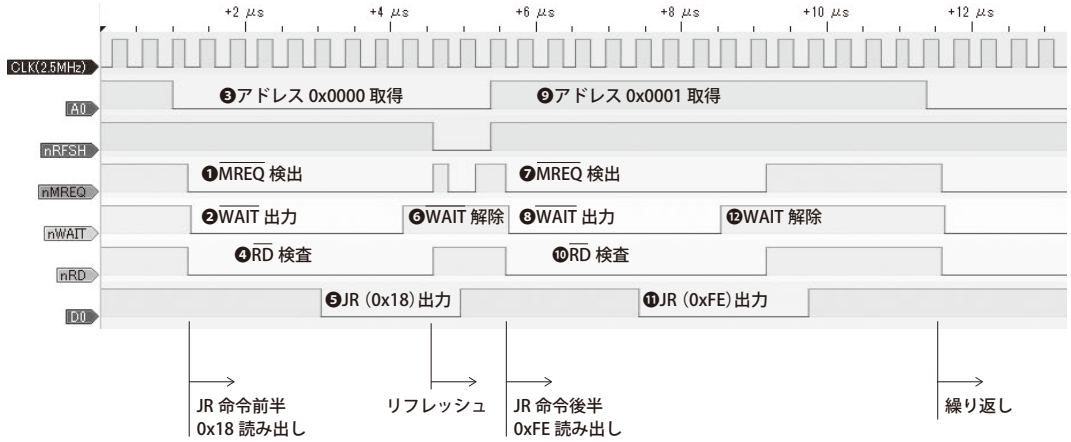


図15●Z80でJR $を実行したときの波形とPIC18F47Q43の主要な動作

EMUZ80は，現状，配列romの先頭にJR $があり，電源を入れるとZ80がリセットしてJR $を繰り返します．その様子は，見た目ではわからないため，ロジックアナライザで観測します．少数の信号しか観測できない安価なロジックアナライザでも工夫しだいで動作確認に十分な役割を果たします（図15）．

アドレスは0x0000と0x0001を往復するので，アドレスバスのA0が0と1を交互に示します．JR $のコードは0x18と0xfeですから，データバスのD0はデータが確定するたびに0となります．ウェイトが掛かるタイミングやリフレッシュの期間が無視される様子はそのような推測をするまでもなく明白です．

これでメモリシステムの基本的な仕組みに誤りがなく，少なくともROMの開放に成功していることがわかりました．引き続き，RAMの開放と後処理を調べます．両方がわかる最短の手順はINC (HL) ループです．JR $にかえてそういうコードを配列romの先頭に置き，Z80に実行させてみます（図16）．

INC (HL) はHLが指し示すアドレスの値をインクリメントします．HLにRAMのアドレスを入れておけば，その値が増えていきます．インクリメントの動作を細かく分解すると，値を読み出し，1を加え，書き戻します．ですから，後処理でデータバスの方向がうまく切り替わっているかどうかもわかります．

```
//Z80 ROM equivalent
const unsigned char rom[ROM_SIZE] = {
  0x21,0x00,0x80, //LD  HL,8000H
  0x34,           //INC (HL)
  0x18,0xFD,      //JR  $-1
};
```

図16●配列romに置いたINC (HL) ループのコード

動作確認には俗称printfデバッガを使います．平たくいうとただの関数printfなのですが，それをメモリの処理に書き加えることで，読み書きの動作を細大漏らさず捉え，状態の推移を端末で見ることができます（図17）．これは高価な開発装置，ICE（インサーキットエミュレータ）の働きに相当します．

```
inc ×
History
  //Z80 memory write cycle
  if(RA5) {
    if((ab.w >= RAM_TOP) &&
       (ab.w < RAM_END)) //RAM
      ram[ab.w - RAM_TOP] = PORTC;
    else if(ab.w == UART_DREG) //U3TXB
      U3TXB = PORTC; //U3 TX buffer
  printf("WR ADRS:%04X,DATA:%02X¥r¥n", ❶
          ab.w,ram[ab.w - RAM_TOP]);
  //Release wait (D-FF reset)
  G3POL = 1;
  G3POL = 0;
  return;
  }

  //Z80 memory read cycle
  TRISC = 0x00; //DATA-BUS -> Z80
  if(ab.w < ROM_SIZE) //ROM
    LATC = rom[ab.w];
  else if(ab.w < RAM_TOP) //Empty
    LATC = 0xff; //Invalid data
  else if(ab.w < RAM_END) //RAM
    LATC = ram[ab.w - RAM_TOP];
  else if(ab.w == UART_CREG) //PIR9
    LATC = PIR9; //U3 flag
  else if(ab.w == UART_DREG) //U3RXB
    LATC = U3RXB; //U3 RX buffer
  else //Empty
    LATC = 0xff; //Invalid data
  printf("RD ADRS:%04X,DATA:%02X¥r¥n",ab.w,PORTC); ❷
  //Release wait (D-FF reset)
  G3POL = 1;
  G3POL = 0;
```

図17●関数CLC_ISRに追加したprintfデバッガ（❶と❷）

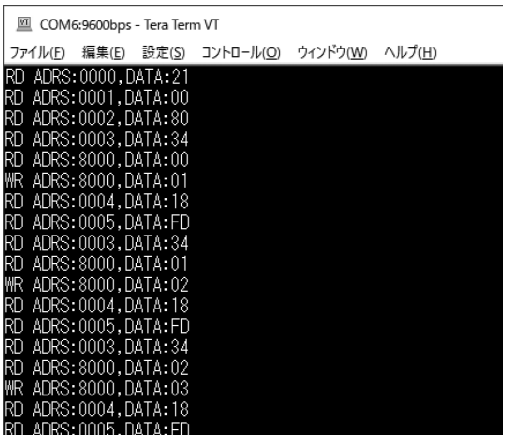


図18●printfデバッガの表示例

printfデバッガでINC (HL)ループの動作を調べると電源を入れてすぐHLにアドレスが入り，以降は，その指し示す値を読み出し，1を加え，書き戻すという動作を繰り返しています（図18）．期待したとおりの結果であり，RAMの開放と後処理がともにうまくいっていると判断することができます．

Z80で端末を制御する3つのサブルーチン

端末インターフェースの動作確認は，これまでに述べた方法が使えません．ロジックアナライザは端末の試験的な操作をやる前にサンプル数が上限へ達してしまいます．printfデバッガはPIC18F47Q43とZ80が端末を奪い合って停止します．ですから，普通にZ80のテストプログラムを作って動かしてみます．

動作確認は端末から入力した文字をそのまま出力するエコーバックで十分ですが，テストプログラムは思うところがあって丁寧に書きました（図19）．こういう場合の通例で，最初に「HELLO, WORLD」を出力します．ゆくゆくはアプリケーションで使えるように，要所の手順をサブルーチンにまとめました．

端末の制御は，最低，1文字出力と1文字入力ができれば実現します．また，その中の入力可否判定を独立させると［Ctrl］+［c］で実行を中断する仕組みなどが作れます．以上の3つを流用可能なサブルーチンで用意しました．流用可能とは，戻り値を除いて，Z80のレジスタが書き換わらないことを意味します．

テストプログラムはJR $やINC（HL）ループに比べてやや大きいため，命令一覧表でコードを調べる方法だと集中力を欠いてよく間違えます．エディタでソースを書き，アセンブラでコードに変換しました．そのコードは，自作のツールでコンマ区切りの文字列に変換し，配列romへコピーアンドペーストしました．

```
;       HELLO, WORLD AND ECHO BACK
;       TARGET: Z80 ON EMUZ80
;
UARTDR  EQU   0E000H     ;DATA REG
UARTCR  EQU   0E001H     ;CONTROL REG
ROMTOP  EQU   0000H      ;ROM TOP
RAMTOP  EQU   8000H      ;RAM TOP
RAMSIZ  EQU   1000H      ;RAM SIZE
STACK   EQU   (RAMTOP+RAMSIZ)
;
        ORG   ROMTOP
        LD    SP,STACK   ;SET STACK
        LD    HL,MESSAGE ;POINT TOP
HELLO:  LD    A,(HL)     ;GET A CHAR
        CP    00H        ;END?
        JR    Z,ECHO     ;IF YES, BREAK
        CALL  PUTCH      ;PUT CHAR
        INC   HL         ;POINT NEXT
        JR    HELLO      ;REPEAT
;
ECHO:   CALL  GETCH      ;GET CHAR
        CALL  PUTCH      ;PUT CHAR
        JR    ECHO       ;REPEAT
;
PUTCH:  PUSH  AF         ;STORE CHAR
PCST1:  LD    A,(UARTCR) ;READ STATUS
        BIT   1,A        ;CHECK TX FLAG
        JR    Z,PCST1    ;IF 0, REPEAT
        POP   AF         ;RESTORE CHAR
        LD    (UARTDR),A ;TX CHAR
        RET
;
KBHIT:  BIT   0,A        ;CHECK RX FLAG
        RET
;
GETCH:  LD    A,(UARTCR) ;READ STATUS
        CALL  KBHIT      ;CHECK RX FLAG
        JR    Z,GETCH    ;IF 0, REPEAT
        LD    A,(UARTDR) ;GET CHAR
        RET
;
MESSAGE DB    'HELLO, WORLD',0DH,0AH,0
;
        END   0000H
```

図19●端末インターフェースのテストプログラム

Z80は，1文字出力と1文字入力と入力可否判定のサブルーチンでPIC18F47Q43のレジスタを読み書きします．また，サブルーチンを呼び出す前後でアドレスをRAM（スタック）へ退避し，復帰させます．う

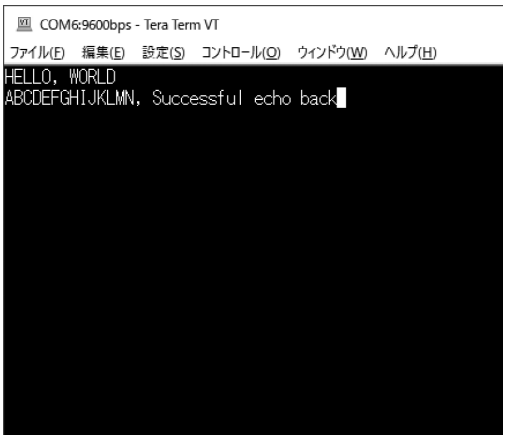


図20●テストプログラムの実行例

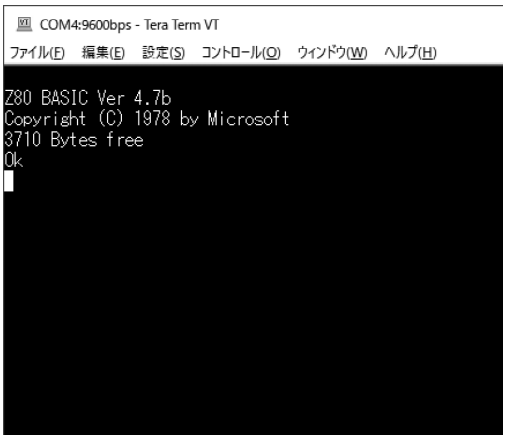


図22●Grant's BASICの起動メッセージ

まく動けば，コードを置いたROMとともにレジスタとRAMの開放にも成功したことになります．

結果は期待したとおりでした（図20）．端末をつないで電源を入れると最初に「HELLO, WORLD」が表示され，以降，エコーバックを繰り返します．これでメモリシステムの動作確認が成功し，ファームウェアが完成しました．この時点でEMUZ80は，まだアプリケーションがないコンピュータに相当します．

BASICの移植に成功して ひとかどのコンピュータが完成

EMUZ80で何かひとつアプリケーションを動かすとしたら，Grant Searleさんが配布し，世界中で愛用されているGrant's BASICがいいでしょう．これは実質的にマイクロソフトBASICです．かつてのパソコンとほぼ同じことができるため，比較して，長所と欠点を含めたEMUZ80の特徴が明らかになります．

Grant's BASICの元ネタはナスコのコンピュータキットNascom-2のBASICです（図21）．ソースはナスコの機関誌で公開されました．Grant's BASICは，それをもとに，機種依存部分の大半を削除し，端末だけで動くように改編してあります．EMUZ80にとって機能的な過不足がなく，しっくり収まります．

Grant's BASICの移植は端末の1文字出力と1文字入力と入力可否判定を書き替える程度で完了します．つまり，EMUZ80のテストプログラムから記述を流用してすぐに動かすことができます．RAMの容量は自動判別され，起動メッセージに，まあそのくらいかなという感じの空き容量が表示されます（図22）．

BASICの働きを隅々まで検証することは無理なので，自作マニアの間で流通しているASCIIART.BASを走らせてみます（図23）．このプログラムは，重い

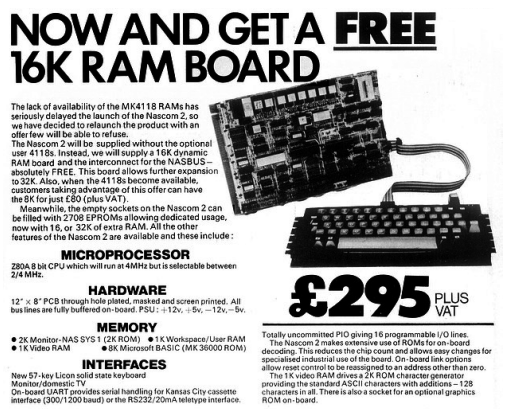


図21●Nascom-2の概要（1980年の雑誌広告より抜粋）

```
10 FOR Y=-12 TO 12
20 FOR X=-39 TO 39
30 CA=X*0.0458
40 CB=Y*0.08333
50 A=CA
60 B=CB
70 FOR I=0 TO 15
80 T=A*A-B*B+CA
90 B=2*A*B+CB
100 A=T
110 IF (A*A+B*B)>4 THEN GOTO 200
120 NEXT I
130 PRINT " ";
140 GOTO 210
200 IF I>9 THEN I=I+7
205 PRINT CHR$(48+I);
210 NEXT X
220 PRINT
230 NEXT Y
```

図23●ASCIIART.BASの内容

図24●ASCIIART.BASの実行結果

浮動小数点計算，2文字めで区別される変数，FOR～NEXTからGOTOで抜ける制御構造といった嫌らしい記述を含み，ささいな問題でよく停止します．

ASCIIART.BASが完走すると端末に文字でマンデルブロ集合のフラクタル図形が描かれます．EMUZ80のGrant's BASICは無事に最後まで実行してOkを出しました（図24）．少なくとも機能的には，ひとかどのコンピュータに仕上がっていることが証明されました．では，速度のほうはどうでしょうか．

実をいうとASCIIART.BASの目的は，端末に面白い図形を描いて見せることではないしBASICの動作確認でもありません．自作マニアの関心はもっぱら実行時間にあり，実質的なベンチマークテストとして，大勢がスコアを発表しています．それらと比較することでEMUZ80の速度を考察します（図25）．

EMUZ80と同じくZ80を採用して端末で操作する形式のコンピュータは，普通のメモリだとクロックが2.5MHzのとき約480秒で，あとは周波数どおりに高速化します．EMUZ80は1461秒ですからファームウェアで1/3に低速化しています．換言すれば，ファームウェアの改善で高速化する余地があります．

現状のファームウェアはEMUZ80の考えかたを端的に実現したもので，とりわけメモリシステムは最短のわかりやすい手順で構成されています．幸い，より速いものと比較しなければ遅いとは感じない水準なので，これを標準ファームウェアと位置付けた上で，将来へ向けて，改善の腹案を列挙しておきます．

機種/クロック周波数	設計	実行時間
EMUZ80/2.5MHz	電脳伝説	1461秒
SBCZ80/2.4576MHz	電脳伝説	489秒
SBCZ80/4.9152MHz	電脳伝説	245秒
Z80-MBC/4MHz	Just4Fun	293秒
Z80-MBC2/8MHz	Just4Fun	150秒
KBC-Z84015S/9.8304MHz	共立電子産業	122秒
AKI-80/9.8304MHz	秋月電子通商	122秒

図25●主要なZ80採用機のASCIIART.BAS実行時間

ハンダごてなしに挑戦できるEMUZ80の新しい目標

すぐに試せる改善の方法はZ80に与えるクロックの周波数を上げることです．やってみると，安定して動作する上限は4MHz，スコアは1389秒で，周波数を上げたわりに速度が上がっていません．このくらいだと無印Z80を使えない欠点のほうが目立ってしまうため，これ単独では，効果的な対策といえません．

速度を劇的に向上させそうなのはCLCです．CLCは先例が乏しく，恐る恐る1組だけ使いましたが，うまく動いた以上は複数組を使いこなしたいところです．たとえば，アドレスをもとにROMとRAMとレジスタの処理を振り分ける処理は，CLCでアドレスデコーダを構成すれば瞬時に完了するでしょう．

プログラミングの腕前に自信がある人はファームウェアをC言語ではなくアセンブリ言語で書くか，大筋をC言語で書いてここぞというところにアセンブリ言語を埋め込む方法で手順の無駄をなくすことができます．さらに，C言語では書けない巧妙な手順を組み立てて速度を上げられる可能性もあります．

当初，EMUZ80の目標は最少の安価な部品でコンピュータを完成させることでした．それをどうにか達成したら意図せずまたファームウェアを改善して速度を上げる目標が生まれました．今度は純粋に知恵の勝負であり，ハンダごてがいりません．プログラミングにだけ関心がある人でも参加していただけます．

EMUZ80の動作原理は，Z80にとどまらず，Z80と似たバスを持つさまざまなCPUへ応用できると思います．実際，現時点でモトローラのMC68008とつないだ試作機が「HELLO, WORLD」を表示するところまで出来上がっています．こちらの方面もなかなか面白いと思うので，よろしければ挑戦してください．

なお，EMUZ80の基本的な操作方法を説明した技術資料，ファームウェアのMPLAB X IDEプロジェクト，BASICのソースとコード，プリント基板の頒布情報などを用意し，私のブログの記事『EMUZ80が完成』（リンクをクリックすると移動します）でご案内しています．必要に応じ，ご覧ください．

すずき・てつや