修士論文

組み込みプロセッサーに搭載するオリジナルインタプリタ言語の

設計と製作

Design and fabrication of original type interpreter languages in

the embedded processors for personal use

報 告 者

学籍番号: 1215040

氏名: 上田 直也

指 導 教 員

綿森　道夫　准教授

平成 31 年 2 月 12 日

高知工科大学　電子・光システム工学コース

目次

第１章 序論

１-１ 研究の背景

1970 年頃、マイクロプロセッサが個人用コンピュータとして使われ始めた頃はプログラミング言語に機械語が用いられていた。のちに BASIC のような簡略化されたプログラミング言語が使われるようになると、パソコンが個人用に普及するようになった。さらに、ハードディスクが登場し記憶容量が増え、CPU の処理能力が向上するにつれて様々な高級言語が開発され、より多様なアプリケーションが実行できるようになった。

マイクロプロセッサが登場した当時は、その性能は恐ろしく貧弱で、現在の PIC のような組み込み CPU と同等かそれ以下であった。速度も数 MHz 程度で、メモリーも RAM・ROM 合わせて最大で 64kB,中には 4kB や 8kB といった小容量でシステムを組み、販売供給されていた [1]。この様な今となっては劣悪な条件で、当時の人々はこの生まれたばかりの個人用コンピュータ(マイコン)をどのようにして利用していくかに情熱をもって取り組み、それがコンピュータの爆発的な性能向上を促し、現在のコンピュータの時代を作っていた要因となっていると思われる。そしてその過程で、少ないメモリーの中に(ホビーユーザーにとっては最初の高級言語である)BASIC を搭載することがパソコンの普及に大きな拍車をかけたように思えてならない。そして現在においても、今度は組み込み系 CPU にかつてと同様に BASIC を組み込むことができるのではないだろうかと考え、研究を始めることにした。この BASIC を組み込む作業の過程で、当時の 8085 や Z80、6502、6800 に高級言語を搭載するときの人々の思いを感じられたら面白いと想像した。もちろん当時は BASIC という高級言語を C 言語という高級言語で構築するなどという贅沢は到底許されず、当時の熱い思いを完全に再現することは不可能であるが、それでも何かしらの思いは感じられるはずである。また、高級言語に使い慣れた現在では、それを実装するためにハードウェアにはある一定以上の処理能力が求められるが、C 言語等の高級言語は組み込みチップ自身には搭載されておらず、ほかのパソコン等の力を借りてクロス開発しなければならない状況である。しかし、IoT の端末ではできるだけ小規模なハードウェア構成で実装しなければならない場合もある。そこで、組み込み系マイコンに直接搭載して無線制御が簡単にできるプログラミング言語を作り、新たな発展のシーズを探したいと思い本研究を始めた。

1-2 目的

本研究の目的は、マイコン上で動作可能なインタプリタを作成し、インタプリタやコンパイラの構築に対する理論を学ぶことと、Bluetooth や無線 LAN を利用して無線制御を実現することである。その際、3 種類の異なるプロセッサと開発環境でできるだけ同等な機能を有する為に、共通な部分と機種依存する部分、開発環境に依存するする部分に分けて製作する。また、選んだ組み込み系チップはホビーユースとして数年前に最も一般的であった PIC と現在組み込み CPU 利用の中心となりつつある Arduino(今回は Arduino 互換機を利用)、そして急成長している Raspberry Pi Zero W である。開発環境が異なってもコアな部分は同等にし、一方で各チップの個性を活かした開発方法は取り組み応えがあるものである。

最終的な目標として着脱可能な入力装置と出力装置を備え付け、3V の乾電池のみで動作するスモールパーソナルコンピュータシステムの構築を目指す。無線制御を実現する例として Bluetooth や無線 LAN を搭載したデバイスを作成し、インタプリタを各デバイス上で動作させて遠隔操作できるシステムを構築する。また、これらのシステムをリアルタイム OS 上で動作させ、インタプリタからコンパイラに切り替えることが可能か検討する。これらのことはそれぞれのチップの個性を活かす取り組みでもある。

1-3 研究の概要

本研究では、異なるアーキテクチャ開発環境で動作させることを想定したインタプリタの設計を行った。最初に基本命令だけを搭載したインタプリタを作成して Windows PC で動作することを確認し、これを PIC マイコン、ESP32、Raspberry Pi の 3 つのアーキテクチャに移植して動作することを確認した。ここで、各アーキテクチャで共通する機能として I2C コントロール機能を持たせ、I2C デバイスとして重量計を製作した。これは色々なセンサーが I2C ポートを備えているのでそれぞれのユニットに直接 I2C でセンサーを接続し、簡単な BASIC 命令で温度等の測定値を取り込んで出力出来れば Windows PC にはない利点になると考えたからである。また、各チップが持つ特徴を生かすためそれぞれに独自の命令を追加した。

PIC マイコンには、入力装置と出力装置を付けて本体だけでプログラムの変更やデータの表示が行えるようにした。ESP32 では Bluetooth に対応させ、Raspberry Pi 経由でパソコンから遠隔操作を行えるようにしたことで、学内無線 LAN に接続できないデバイスをネットワーク経由で操作できるようにした。ESP32 は Arduino 互換機での開発という意味合いを持つが、標準で無線 LAN と Bluetooth に対応している。今回学内 LAN への直接の接続ができなかったので Raspberry Pi を経由して無線 LAN の SSH で制御を行った。

1-4研究の意義

インタプリタの設計は字句解析や文法チェックで複雑な処理が必要になる。この処理が汎用的になれは別言語への翻訳やソースコードのデータ解析などに応用できる可能性もある。また、複数のアーキテクチャに移植して動作させることはマルチプラットフォームの技術について学ぶことになると考える。また、わずかではあるが、パソコンの黎明期からBASICが使われるようになっていく過程を追体験できたということに対しても意義があったと考えている。

Bluetoothデバイスを Raspberry Piで遠隔操作する実験はインターネットに直接つなぐことのできないデバイスや環境でも IoTが実現可能であることに意義があると考える。このことは、学内 LANの届く範囲に Raspberry Piを設置し、その Raspberry Piと Bluetooth通信が届く範囲で ESP32を設置すれば、Raspberry Piよりも省エネで I/Oピンの自由度が高く装置に組み込むことが容易なESP32でインターネットを通じたリモート制御が可能であることを示している。

第２章インタプリタの設計

２-１言語仕様

製作するインタプリタ言語に求められる要件は、処理能力の低いハードウェアでも本体中に直接プログラムが記述でき、すぐに実行が可能であることと容易に扱える言語仕様であることである。そこで、なじみやすい BASIC の言語仕様を参考に基本命令文を実装した後、独自の命令文を追加した。実際 BASIC インタプリタは、初期の処理能力の低いパソコンにも最初に搭載され、現在のパソコン全盛期の足掛かりとなったことで有名である。

製作するコンパイラはソースコードの実行だけでなく、命令を直接実行できるものとした。直接実行する命令とリストの記述は行番号の有無で区別する方法を取った。行の先頭に行番号がある場合をリストとし、行番号がない場合はその行の命令を直接実行できる。コマンドでモードを切り替える方法も考えたが、テキストエディタなしでリストを編集する場合、行番号が必要になるのでこの方式を取らなかった。組み込み系 CPU に搭載する際は、入力、出力をどの様に構築するかが大きな選択肢として残るので、出来るだけ入力、出力デバイスの選択肢を狭めないためにはこの方法がベストと判断した。

インタプリタの構成は、ソースコードを 1 行読み込んだ時点で字句解析して内部コードに変換し、先頭が番号である場合はバッファに記録する。後で実行命令が入力されるとバッファの先頭から実行するようにした。これにより実行時の字句解析処理を省略することができる中間言語方式である。

ここまでの要件を前提として、機能を以下に示す最低限必要なものに絞ることで確実に動かすことを優先させた。

- 使用するデータ型は int 型整数のみ
- 変数は数を決めてグローバル変数として定義しておく
- 配列はサイズが決められた 1 次元配列

データ型は、プログラムの規模を抑えるために 2byte の整数型のみを扱うことにした。アーキテクチャによっては 4byte 以上で表現できる場合もあるがプログラムを統一するために 2byte で固定することにした。変数はアルファベット 1 文字で表現する方式と複数文字使用できる方式の 2 通りを考えたが、名前を記録する必要がなく必要なメモリーを 26 個に固定できるのでアルファベット 1 文字で表現する方式に統一した。配列は OS を使用していない環境での動的メモリーの確保を避けるため、サイズを固定した 1 次元配列をあらかじめ用意しておく方式にした。

次に使用するキーワードについて考える。キーワードは大文字と小文字を区別しないことにした。ただし、1 つのキーワード内ではどちらかに統一して大文字と小文字の境目で字

句を区切ることができるようにした。これは PIC マイコンに移植することを想定して文字列バッファを節約するために限界までソースコードを詰めて記述できるようにするためであるが、ソースコードの視認性を保つためにスペースで区切ることも可能にした。

ここで、構文を表現する際に用いる BNF(Backus Naur Form)記法について説明する。BNF 記法で数字と英字の定義を表現すると次のようになる [2]。

＜数字＞ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 (2-1)

＜英小文字＞ ::= a | b |c | d | e | f | g | h | i | j | k | l | m | n
| o | p | q | r | s | t | u | v | w | x | y | z (2-2)

＜英大文字＞ ::= A | B |C | D | E | F | G | H | I | J | K | L | M | N
|O | P | Q | R | S | T | U | V | W | X | Y | Z (2-3)

＜英字＞ ::= ＜英大文字＞ | ＜英小文字＞ (2-4)

ここで、「＜」「＞」で囲まれたものを構文要素と呼び、「::=」は左辺を右辺によって定義するという意味であり、「｜」で区切られた要素はそのうちのどれか１つという意味である。数字の場合 0 から 9 のうちのどれか１つなので(2-1)のようになる。英字は(2-4) のように(2-2)と(2-3) を用いて大文字と小文字のうちのどれか１文字を表すことができる。

今回製作するインタプリタでの表記方法を BNF 記法で示す。

＜識別子＞ ::= ＜英大文字＞｛＜英大文字＞｝
| ＜英小文字＞｛＜英小文字＞｝ (2-5)

＜数値＞ ::= ＜数字＞｛＜数字＞｝ (2-6)

＜変数＞ ::= ＜英字＞ (2-7)

＜配列＞ ::= 「@」「(」＜式＞「)」 (2-8)

ここで、「{」「}」で囲まれたものは０回以上繰り返されるという意味である。キーワードや関数などの識別子は連続する大文字又は小文字の英字である(2-5)。数値は整数のみを扱うので、連続した数字の羅列で表すことができる(2-6)。変数はアルファベット１文字で表現し、大文字と小文字を区別しないので(2-7)のようになる。配列は「@」で表し、要素は「(」「)」の中に指定する(2-8)。配列要素の指定は、直接数値を入れる場合や変数などを用いた式を入れる場合があるので＜式＞という構文要素を定義する(2-9)。

＜式＞ ::= ＜項＞｛＜演算子＞＜項＞｝ (2-9)

＜項＞ ::= [−] (＜数値＞ | ＜変数＞ | ＜配列＞
| 「(」＜式＞「)」| ＜代入式＞ | ＜関数＞) (2-10)

＜代入式＞ ::= (＜変数＞ | ＜配列＞)「＝」＜式＞ (2-11)

式は多項演算に対応しているので先頭の項の後ろに演算子と項を交互につなげることができる(2-9)。項は先頭に符号をつけることができ、(2-10)で符号「－」が「[」「]」で囲まれているのは省略可能であることを意味する。演算の優先度は「(」「)」で囲むことで指定できる。代入式は代入演算子「＝」を用いて変数か配列に式の計算結果を代入する(2-11)。この代入式は項として使用することも可能なので式の途中で変数の値を変えることもできる。

関数は下に示すように識別子と括弧に囲まれた式で構成され、複数のパラメータを持つ関数の場合は「,」で区切る(2-12)。パラメータを持たない関数の場合は括弧だけをつける。

＜関数＞ ::= ＜識別子＞「(」[＜式＞ {「,」＜式>}]「)」　　(2-12)

すべての環境で共通する関数は以下の表2-1に示す2つを実装した。RANDは1つのパラメータを持ち、1から指定した整数までの間の数を返す。SIZEは例外的に識別子のみで使用することが可能で、括弧がない場合はソースコード用メモリーの残りサイズを返し、括弧を付けた場合には使用可能な配列のサイズが返される。本BASIC言語では、独自の関数をプログラム内に定義できる訳ではなく、あくまでもGOSUB、RETURN命令による行番号を利用したサブルーチンが基本的である。そのため関数とは予めインタプリタ設計時に組み込んでおいた組み込み関数のことを意味している。

表2-1 関数

関数名	機能
RAND	乱数
ABS	絶対値
SIZE	ソースコード用メモリーの残りサイズ/配列のサイズ

演算子は加減乗除に加え剰余も実装した。剰余の演算子はC言語と同じ'%'で表現することにした。関係演算子の等号は、代入演算子との区別を前後の計算式からでなく演算子のみで判別できるように明示的に異なる表記方法を取った。この部分がC言語と同じ部分である。論理演算子は字句解析のアルゴリズムを簡略化するためにビット演算と共通の識別子を用いた。

表2-2 演算子

演算子	機能
+、-、*、/、%	加減乗除と剰余
==、 !=、<、 >、<=、 >=	比較演算
&, |, ^	論理演算/ビット演算

次に実装する命令文を定義する。実装する機能は、条件分岐、ループ処理、ジャンプ命令、

サブルーチン、標準入出力の 5 種類である。すべてのプログラムは条件分岐とジャンプがあれば記述できるはずであるのでこれだけの命令の実装で充分である。これらの命令はソースコードの記述時以外に現れたときは即時実行するために 1 行で入力可能である。

条件分岐の定義を(2-13)に示す。IF 文は＜式＞の値が 0 以外のときに後ろの＜文＞を改行するまでの範囲で実行する文法にした。この場合、＜式＞の値が 0 とき処理を次の行に飛ばすだけで良いので＜文＞の範囲を調べるルーチンを省略することができる。条件が偽の場合の処理は GOTO で代用することにした。

＜IF＞ ::= IF ＜式＞｛＜文＞｝ (2-13)

ループ処理として FOR 文を(2-14)に示す。FOR 文は＜代入式＞でカウンタ変数を初期化し、TO の後ろの＜式＞で指定した終了値と一致するまで＜文＞を繰り返し実行する。カウンタ変数は NEXT で加算され、加算値は STEP で指定することができる。STEP は省略可能で、省略した場合の加算値は 1 である。加算値の絶対値が 2 以上の場合は終了値を超えた時点でループと終了する。

＜FOR＞ ::= FOR ＜代入式＞ TO ＜式＞ [STEP ＜式＞] [改行]
｛＜文＞ [改行]｝ NEXT [＜変数＞] (2-14)

ジャンプ命令には(2-15)に示す GOTO 文を用いる。この GOTO 文は＜式＞で指定した行番号に移動して処理を実行する。指定された行番号が存在しない場合はエラーとしてプログラムの実行を終了する。

＜GOTO＞ ::= GOTO ＜式＞ (2-15)

サブルーチンの記述には GOSUB(2-16)と RETURN(2-17)を用いる。GOSUB 文は GOTO 文と同様に＜式＞で指定した行番号に移動して処理を実行するが RETURN で移動元の GOSUB の次の位置に戻ることができる。

＜GOSUB＞ ::= GOSUB ＜式＞ (2-16)
＜RETURN＞ ::= RETURN (2-17)

標準入力から入力された定数を変数や配列に代入する INPUT 文を(2-18)に示す。INPUT 文は実行時に変数名または配列のプロンプトを表示して数値が入力されるのを待つ。INPUT 文と同じ行で別の命令文を継続する場合は「;」で区切ることで INPUT 文の処理を終了できる。

＜INPUT＞ ::= INPUT (＜変数＞|＜配列＞){「,」(＜変数＞|＜配列＞)} [;]　　(2-18)

標準出力に文字列や数値を表示する PRINT 文を(2-19)に示す。PRINT 文は後ろに続く文字列や式の計算結果を表示し最後に改行する。複数の文字列や式を連続して表示する場合は「,」で区切ることで可能となる。PRINT 文と同じ行で別の命令文を継続する場合は INPUT 文と同様に「;」が使用できる。(2-20)の TEXT 文は PRINT 文の自動改行を除いたもので、このインタプリタの基本部分が完成した後に追加した。

＜PRINT＞ ::= PRINT (＜文字列＞|＜式＞){「,」(＜文字列＞|＜式＞)} [;]　　(2-19)
＜TEXT＞ ::= TEXT (＜文字列＞|＜式＞){「,」(＜文字列＞|＜式＞)} [;]　　(2-20)

文字列は以下の(2-21)に示すように定義した。文字列は「“」か「’」で囲むことで使用できる。文字列中に「“」や「’」を含める場合、文字列中のものと異なる記号で囲む必要がある。

＜文字列＞ ::= 「“」{＜英字＞|＜数字＞|「’」}「“」
　　| 「’」{＜英字＞|＜数字＞|「“」}「’」　　(2-21)

尚 PRINT 文などで使用する文字列には、画面制御用のエスケープシーケンスを含めることも可能である(2-22)。これらの処理は実行するアーキテクチャにより異なる。

＜エスケープシーケンス＞ ::= 「¥」(n ｜ r ｜ f ｜ b ｜「¥」)　　(2-22)

この他、表 2-3 に示すプログラムの実行制御に関する命令を追加した。

表 2-3 実行制御命令

キーワード	機能
NEW	ソースリストをすべてクリア
LIST	ソースリストの表示
RUN	プログラムの実行
STOP	プログラムの終了
REM	コメント

NEW はソースリスト及び変数と配列をすべて初期化する。LIST はバッファに保存されているソースリストを表示する命令で、後ろに行番号を付けるとその行番号から先のリストを表示することも可能である。RUN でプログラムを先頭から実行し、ソースリストの終

端へ到達するかソースリスト中の STOP を読み込んだ時点で終了する。REM は行末までの文字をすべてコメントと見なして実行時に無視する。NEW、LIST 及び RUN は暴走を防ぐためにソースコード内には記述できないことにした。

コンパイラの理論 [2]では、BNF 記法を構文図か正規表現に直し、それを元に有限オートマトンに変換する。状態数の最小化を経て可能であれば決定性有限オートマトンに変換するなどして字句解析ルーチンに進む。本研究では、一般的な BASIC 言語の文法を利用したために、この作業が不要であり、後戻りの無い再帰的下向き構文解析が可能であるので(いわゆる LL(1)文法)、終端記号に達するまで、それぞれの文法の構成要素に対して First、Follow、Director を求めていない。完全に力技で字句解析(構文解析を含む)を行う手法とした。これは BASIC が最初にパソコンに搭載された際にも行われていた手法であると考えている。

２-２字句解析

このインタプリタは文字を入力して改行されると、命令の処理を始める。最初は入力された文字列を単語単位で分割し、中間コードに置き換えてバッファに追加する。

最初に文字列から字句を切り出す為にソースコード ２-1 に示す getToken 関数を作成した。

ソースコード ２-1 字句を切り出す関数

```
int getToken(CHAR_T **linePtr, SIZE_T *toklen);
```

ここで、CHAR_T と SIZE_T はアーキテクチャによって型の種類を変更する必要があったのでマクロを使用して定義しており、それぞれ char 型と正の整数型に置き換わる。この関数の引数には読み込む文字列のポインタ linePtr と字句の長さを格納する変数のアドレス toklen を渡す。linePtr が指す文字列のアドレスを字句の先頭に移動し、その字句の長さを toklen が指す変数に代入して字句の種類に応じて定数を返す。字句の種類は数字、識別子、文字列、未定義の文字、行の終端の 5 種類である。識別子には英大文字、小文字、演算子があり、字句はそれぞれで分けるが、同じ定数を返す。定数の値は enum で文字定数として定義した。

ソースコード ２-2 字句の種類を表す定数（抜粋）

```
I_NUM,      //number
I_VAR,      //variable      (キーワード, 演算子の識別で兼用)
I_STR,      //string
I_UNKNOWN,  //未定義の文字
I_EOL       //行末
```

定数名は参考図書 [3]で用いられていた名前のルールに従って先頭に「I_」を付けて文字定数であることを表した。尚これらの文字定数は中間コードで識別コードとして使用する。字句解析においては先程の BNF 記法で記述された文法を利用した。また、実際に 1 文字読み込むルーチンに関しては、機種依存があるのでここでは述べない。

2-3 中間コード

中間コードを格納するバッファはソースコード 2-3 に示すような配列をグローバル変数として宣言して置く。

ソースコード 2-3 中間コードバッファと変数

```
CHAR_T ibuf[IBUF_SIZE];          //中間コードバッファ
CHAR_T listbuf[LIST_SIZE+1];    //リストバッファ
```

これらのサイズはアーキテクチャのメモリーサイズに合わせて変更するのでマクロで定義している。リストバッファには 1 行分ソースコードを中間コードに変換したとき先頭が行番号であった場合に格納される。リストバッファのサイズは PIC マイコンなどで起動時にプログラムを実行するフラグのために 1byte 余分に確保したが最終的に実装には至らなかった。尚、インタプリタ言語の性質上、入力したプログラムは全て RAM 上に保存する必要がある。もちろんプログラムを保存する為に、OS のある Raspberry Pi ではファイルの形式で、PIC では搭載した EEPROM に保存できるが、実行時には全て RAM に転送する必要が生じる。文法に対する幾つかの制限は PIC に搭載し、利用可能な RAM の量によって決定された。

中間コードには字句の種類によって次のルールに従って変換する。

- キーワードや演算子は 1byte の定数に置き換える
  - FOR ⇒ I_FOR、 「==」 ⇒ I_EQ
- 数値は識別子と 2 byte のデータに変換する
  - 512 ⇒ I_NUM 0x00 0x02
- 変数は識別子と 1byte の変数番号に変換する
  - I ⇒ I_VAR 0x08
- 文字列は先頭に識別子と 1byte の文字数を追加
  - “String” = I_STR 0x06 ‘S’ ‘t’ ‘r’ ‘i’ ‘n’ ‘g’

中間コードへの変換はソースコード 2-4 に示す LineToCode 関数で行う。

ソースコード 2-4 文字列を中間コードに変換する関数

```
SIZE_T LineToCode(CHAR_T *str);
```

LineToCode 関数は引数に指定された文字列から getToken 関数で字句を切り出し戻り値に応じて中間コードに変換する処理を行う。

字句が変数の場合、数字文字列を数値に変換してリトルエンディアンで 2byte のデータにし、数値であることを示す識別コードを加えて 3byte のデータとして中間コードバッファに追加する。

字句が識別子の場合は字句が命令文のキーワードか調べ、キーワードであった場合は予めキーワードと対になるように定義して置いた列挙定数に置き換え、キーワードではなく字句の長さが 1 文字の場合は変数と認識する。変数はアルファベットを 0 から 25 までの数値に変換して識別コードを加え、2byte のデータとして中間コードバッファに追加する。字句の長さが 2 文字以上ある場合は未定義の命令としてエラーとなる。

字句が文字列の場合は識別コードと文字列の長さを加えて、文字列はそのまま中間コードバッファにコピーする。このとき文字列の区切りである「"」や「'」は必要ないので除く。

行の終端まで読み終わると行末の識別コードを中間コードバッファに追加して次の処理に移る。インタプリタにおいては、プログラムを解釈してファイルに書き出すという手順を取らないために、中間コード方式を取らなければ、例えば FOR のような命令が、"F","O","R" といった文字列で RAM に保存されてしまい、メモリーの無駄遣いになることが多い。コンパイラと違って最適化処理がないので字句解析の際に一部構文解析も兼ねた中間コードへの変換は有効な手法である。

2-4 プログラムの記述

中間コードへの変換が終わると入力されたものが即時実行する命令かプログラムかを判

図 2-1 リストバッファの構造

別し、プログラムの場合はリストバッファに格納する。リストバッファの構造は行の追加や削除が容易にできる必要があるので、図 2-1 に示すように各行に 1 行の長さ情報を加えることにした。プログラムの場合の中間コードは先頭の 1byte が数値の識別コードなので、これを長さ情報に置き換えることでデータサイズを変えることなくバッファに保存できる。

最終行には長さを 0 とすることでプログラムがそこで終了していることを表現することができる。

挿入処理は最初に中間コードの行番号からリストバッファ内の挿入位置を検索し、ここで重複する行が見つかった場合はその行から後ろのデータをシフトして削除する。挿入位置を決定するとリストバッファの空き容量を中間コードの長さと比較し、挿入可能であれば後ろのデータをシフトして空いたスペースに中間コードをコピーする。挿入時はシフト処理を 2 回行うことになるが単純なアルゴリズムで任意の行を追加できる。この様に RAM 上にプログラムを行番号形式で保存する方式は、長さ情報を利用して次の場所を検索するので、リストバッファには行番号順に並べ替える必要が生じる。このことが CPU のコストを多く使う要因になっている。

2-5 実行処理

中間コードを実行するにはソースコード 2-5 に示すグローバル変数が必要になる。

ソースコード 2-5 実行処理で使用する変数

```
CHAR_T *cip;    //中間コードポインタ
CHAR_T *clp;    //カレント行の先頭ポインタ
CHAR_T *lstk[LSTK_SIZE];//List スタック領域
unsigned int lstki; //スタックインデクス
```

cip と clp はそれぞれ実行中の中間コードと行のポインタである。lstk は FOR 文や GOSUB 文を実行する際のスタックで使用し、lstki にはスタックのサイズを記録されている。

中間コードの実行では最初に実行制御命令であるかどうかを調べる。ソースコード 2-6 の iCommand 関数は命令が NEW、LIST、RUN の場合にそれぞれの処理をして次の命令が入力されるのを待ち、IF や FOR など文の場合は中間コードを 1 行実行する LineExe 関数を呼び出す。

ソースコード 2-6 命令を実行する関数(一部省略)

```
void iCommand(void){
        cip = ibuf;         //中間コードポインタを先頭に
        lstki = 0;          //スタッククリア

        switch(*cip){
                case I_NEW:
                        メモリー等の初期化
                        break;
                case I_LIST:
                        リストの表示処理
                        break;
                case I_RUN:
                        リストの実行処理
                        break;

                default:
                        LineExe();          //中間コードの実行
                        break;
        }
}
```

LineExe 関数は I_EOL が見つかるまで処理を続けるが、FOR 文や GOTO 文で無限ループ処理を実行した場合は外部からの中止コマンドを受け付ける必要があるので、GetEsc 関数で中止コマンドを受け取ったときにループを抜ける。GetEsc 関数は Windows 用プログラムの場合、kbhit 関数を用いてキーの入力を監視して中止コマンドとして割り当てたキーが入力された場合は 1 を返す。尚 kbhit 関数はほかのアーキテクチャでは使用できないのでマクロを用いて置き換えられるようにしている。

プログラム中でエラーが発生した場合はエラーコードを変数にセットしてプログラムを終了し、エラーコードに割り当てたメッセージを表示する。エラーコードのセットはインタプリタのデバッグのために SetError マクロで行っている。

ソースコード 2-7 中間コードを実行行する関数(一部省略)

```
CHAR_T *LineExe(void){
        SIZE_T lineno;   //行番号
        CHAR_T *lp;      //行ポインタ
        int forindex, vto, vstep;    //カウンタ変数の index, 終了値, 増分
        int ifcond;          //if の条件

        while(*cip != I_EOL){       //行末まで繰り返す

                //強制終了
                if(GetEsc()){
                        SetError(ERR_ESC); break; //ERROR
                }

                //コードの実行
                switch(*cip){
                        case I_GOTO:
                        break;
                        case I_GOSUB:
                        break;
        ------------------------------省略-----------------------------
                        default:
                        SetError(ERR_SYNTAX); break; //ERROR
                }
        }
}
```

コードの実行開始時の cip が指す値には識別コードが格納されており、switch でそれぞれの処理へ分岐し、各処理が終わると cip を次の命令の識別コードの位置に更新する。

1. GOTO

GOTO の処理では分岐先の行番号を取得するとリスト中の該当するアドレスに変換し、clp に格納する(ソースコード 2-8)岐先が存在しない場合はエラーコードをセットして終了し、正常に分岐できた場合は cip を分岐先の行番号情報を飛ばして中間コードの先頭のアドレスにセットする。

ソースコード 2-8 GOTO 文の処理

```
case I_GOTO:
        cip++;   //GOTO のパラメータの位置に進める
        //分岐先の行番号を取得
        lineno = Expression(I_NUM);       //パラメータから分岐先の行番号を取得
        if(basic_err) break;          //ERROR
        lp = GetLinep(lineno);     //分岐先のアドレスに変換
        if(lineno != GetLineNum(lp)){       //分岐先が存在しない場合
                SetError(ERR_ULN); break; //ERROR
        }
        clp = lp; //行ポインタを分岐先へ移動
        cip = clp + 3;//中間コードの先頭
break;
```

分岐先の行番号の取得は値が式として与えられる場合があるので後述する Expression 関数を使用している。basic_err は Expression 関数でエラーが発生したときにエラーコードがセットされるのでエラーを検出すると処理を抜ける。GetLinep 関数はリストから指定された行番号に対応する中間コードの先頭アドレスを返し、対応する行のコードがない場合はリストの終端のアドレスを返す。GetLineNum 関数は指定されたリスト中のアドレスにある中間コードの行番号を返す関数である。

2. GOSUB、RETURN

GOSUB 文は GOTO と同じ手順で分岐先のアドレスを取得し、元の行と中間コードのアドレス及び GOSUB の識別コードをスタックに格納してから clp と cip を更新する(ソースコード 2-9)。

サブルーチンの FOR ループ中に RETURN がある場合は FOR 文のスタックも取り出す必要があるので、スタックの配列 lstk は FOR 文と併用している。

RETURN ではスタックの情報を確認して、GOSUB のデータであれば識別コード、中間コードと元の行のアドレスという順で取り出して clp と cip に戻す処理をする。

スタックに FOR 文のデータが入っていた場合は GOSUB のデータが出るまで読み捨て、GOSUB がなかった場合はエラーコードをセットして終了する(ソースコード 2-10)。

ソースコード 2-9　GOSUB 文の処理(一部省略)

```
case I_GOSUB:
        cip++;   //GOSUB のパラメータの位置に進める
        //分岐先の行番号を取得
        (GOTO と同じ)
        //スタック処理
        if(lstki > LSTK_SIZE - 3){//stack overflow
                SetError(ERR_STKOF); break; //ERROR
        }
        lstk[lstki++] = clp;          //行ポインタ
        lstk[lstki++] = cip;          //中間コードポインタ
        lstk[lstki++] = (CHAR_T*) I_GOSUB;
        clp = lp; //行ポインタを分岐先へ移動
        cip = clp + 3;//中間コードの先頭
break;
```

ソースコード 2-10　RETURN 文の処理

```
case I_RETURN:
        //スタックのチェック
        if(!lstki){           //スタックが空のとき
                SetError(ERR_STKUF); break; //ERROR
        }
        while(I_FOR == (unsigned int)((unsigned int*)lstk[lstki -1])){
                //FOR を強制終了
                lstki -= 6;          //スタックインデックスを Popup
                if(lstki < 3){
                        SetError(ERR_STKUF); break; //ERROR
                }
        }
        --lstki;
        cip = lstk[--lstki];//中間コードポインタ
        clp = lstk[--lstki];//行ポインタ
break;
```

3.FOR、NEXT

FOR 文ではカウンタ変数の変数番号を取得して初期値を代入し、終了値と増分を取得してスタック処理を行う(ソースコード 2-11)。スタックに格納するデータはループの先頭にある文の行と中間コードのアドレス、FOR 文のパラメータ 3 つと FOR 文識別コードである。

ソースコード 2-11　FOR 文の処理(一部省略)

```
case I_FOR:
        cip++;
        //カウンタ変数のセット
        if(*cip != I_VAR){
                SetError(ERR_VAR); break; //ERROR
        }
        forindex = *(cip+1);        //カウンタの変数アドレス
        Expression(I_NUM);          //初期値の代入
        if(basic_err) break;        //ERROR

        終了値、増分値をセット
        オーバフローチェック

        //スタック処理
        if(lstki > LSTK_SIZE - 6){//stack overflow
                SetError(ERR_STKOF); break; //ERROR
        }
        lstk[lstki++] = clp;        //行ポインタ
        lstk[lstki++] = cip;        //中間コードポインタ
        //FOR のパラメータをスタック
        lstk[lstki++] = (CHAR_T*) ((unsigned int*)vto);      //終了値
        lstk[lstki++] = (CHAR_T*) ((unsigned int*)vstep);    //増分値
        lstk[lstki++] = (CHAR_T*) ((unsigned int*)forindex);//カウンタ変数名
        lstk[lstki++] = (CHAR_T*) I_FOR;//スタックの種類
break;
```

NEXT 文では逆の手順でスタックからデータを読み、カウンタ変数に増分値を加算してループの先頭にポインタをセットする処理を行う。

4.IF

IF 文は Expression 関数で条件式を計算し、結果が真の場合に後の処理を続け、偽の場合は cip を行末まで進めて次の行の処理へ進める(ソースコード 2-12)。

ソースコード 2-12 IF 文の処理

```
case I_IF:
        cip++;
        ifcond = Expression(I_NUM);        //真偽を取得
        if(basic_err){
                SetError(ERR_IFWOC); break; //ERROR
        }
        if(ifcond) break;  //真の場合 次の処理へ
case I_REM:
        while(*cip != I_EOL){cip++;}//次の行へ進む
break;
```

次の行へ進む処理は REM と共通しているので break せずに REM の処理に流している。この手法はリ・チン・ワンのパロアルトタイニーBASIC で用いられていたものである [3]。

5.式の処理

Expression 関数では以下のような式でも原則に従って計算できるようにする必要がある

$$a = 12 * (4 + 5) \qquad (2\text{-}23)$$

これを実現するために Expression 関数では再帰呼び出しを行う。式(2-23)を計算する場合、最初に「a」と「=」を読み、「=」を引数に指定して Expression 関数を再帰呼び出しする。ここで「12」と「*」を読んだとき引数の「=」と優先度を比較し、「*」の優先度が高いので「*」を引数に指定してさらに再帰する。次の項を読むとき「(」があるので先に括弧の中を Expression 関数で計算し、計算結果を項とする。次の項が無くなると演算結果を持って関数を抜ける。

Expression 関数をソースコード 2-13 に示す。1 つ目の項は GetValue 関数で取得する。この関数は cip の位置にある識別コードから項の種類を判断し I_NUM ならば後の 2byte を数値に変換、変数や配列の場合は保持している値を返し、括弧の場合はその中を別の式として Expression 関数で計算て結果を返す。

演算子の優先度は、OpePrioirity 関数で演算子の識別コードから優先順位の定数に変換し直前の演算子と比較する。直前の演算子は Expression 関数の引数から得られる。式の先頭の場合は引数に演算子以外の識別コード(I_NUM)をセットして強制的に次の項の読み込

みへ進める。

ソースコード 2-13 式の処理

```
int Expression(int ope){
        int result;                                     //計算結果
        static int temp, op;        //一時変数(operand | operator)
        result = GetValue();        //1 つ目の項
        while(1){
                if( OpePrioirity(*cip) > OpePrioirity(ope) ){
                                //(次の優先度が高い | 式の先頭)
                        temp = *cip;        //次の演算子
                        cip++;
                        temp = Expression(temp);//次の項を計算
                }else{
                        op = ope;           //演算子を記憶して上の階層に戻る
                        return result;      //現在の項
                }
                result = DyadicOperation(result, temp, op);//二項演算
        }
}
```

優先度が同等以下の場合は DyadicOperation 関数で二項演算を行う。この関数の引数には現在の項、１つ前の項又は先に計算した結果、演算子を与える。

2-6 全体の構成

インタプリタのすべての処理は以下の Basic 関数で完結するようにした。

ソースコード 2-14 Basic 関数

```
int Basic(CHAR_T *form);
```

メイン関数で行う処理はプロンプトを表示して命令文の入力を待ち、1 行文入力されたら Basic 関数を呼び出して処理が終わるとエラーの処理をする操作を繰り返すだけである。

インタプリタの基本的なシステムは basic_com.c にまとめて、環境に合わせて書き換える必要のあるソースファイルと分けた。環境依存のソースファイルには、入出力処理や追加機能の処理を行う iostream.c と文字列の処理やデータ型の定義などの汎用的な関数をまとめた my_function.c である。尚コマンドの追加をする際 basic_com.c を書き換えずにインタプリタのシステムを拡張するために exte.h というヘッダファイルを作成した。exte.h はマク

ロを用いて basic_com.c の中にソースコードを展開する。

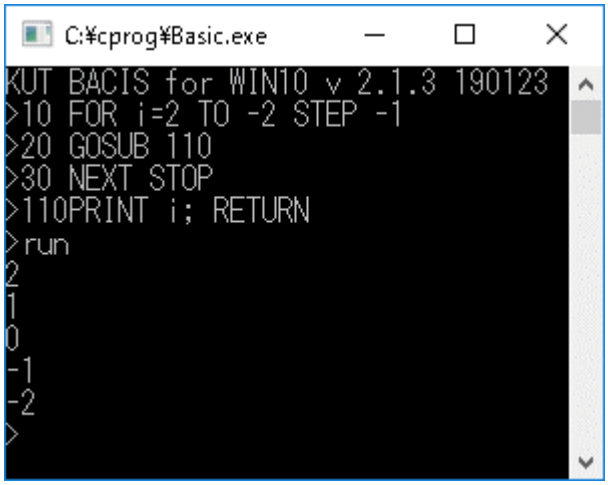

図２-2 実行例１

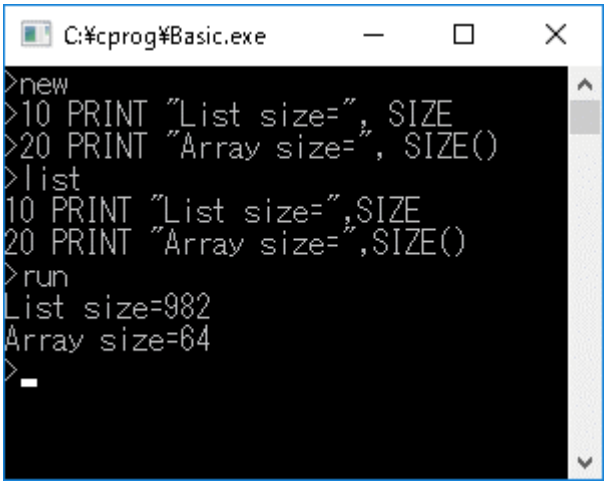

図２-3 実行例２

図２-2 は FOR 文と GOSUB 文の実行例で、最初の行は起動メッセージとしてインタプリタのバージョンを表示している。次の４行はプログラムの入力部分で、STOP 命令と RETURN 命令は直前の命令と同じ行に書き込んだが正常に機能している。プログラムの 110 行目では行番号と PRINT を続けて書いているが数字とアルファベットの境目を正しく識別しているのでエラーにはならない。

図２-3 ではリストバッファを初期化して 2 行のプログラムを書き込んで LIST 命令で表示した後実行した例である。リストバッファは最大で 1023Byte 使用できるように設定しており、図の実行結果では 2 行のプログラムが使用したサイズを除いた残りのサイズを表示している。配列のサイズは 64Byte であることを表している。

２-７行番号を使用しないプログラムの実現

PC や Raspberry Pi ではテキストエディタを使用できるので行番号を必要としないプログラムを実行するコンパイラの設計を考えた。

言語仕様で GOTO や GOSUB の分岐先をラベル名やサブルーチン名で指定出来れば行番号を使用する必要がなくなる。これを実現するには中間コードに変換する際に格納する分岐先のアドレスとラベルやサブルーチンが定義された位置をリンクさせる必要がある。そこでラベルやサブルーチンのアドレスを記録する名前リストを作成して実行時に参照する方式を考えた。

ソースリストを読み込んで中間コードに変換処理をする際、ラベルやサブルーチンの定義を見つけるとその名前とアドレスを名前リストに記録し、GOTO や GOSUB では名前リストの参照する番号を中間コードに加える。逆に GOTO や GOSUB が分岐先の定義より前にあった場合は名前だけを名前リストに記録して置き、ラベルやサブルーチンを見つけた

時にアドレスを追加することで、サブルーチンの定義を自由な場所に置くことができる。

名前リストは以下に示す構造体の配列を用いて、中間コードの生成後際にポインタのリンク処理を行うことにした。

ソースコード 2-15 名前リストの構造

```
typedef struct {
        CHAR_T name[IDENT_NAME_MAX];      //名前
        int num;                          //ラベルやサブルーチンのアドレス
        unsigned int tp :3;               //ラベルかサブルーチンの識別
}Ident_t;
```

ラベルやサブルーチンの言語仕様は以下のように設定した。

<GOTO> ::= GOTO <IDENT> (2-24)

<GOSUB> ::= GOSUB <IDENT> (2-25)

<LABEL> ::= 「:」<IDENT> (2-26)

<SUB> ::= SUB <IDENT> { <文> } RETURN (2-27)

<IDENT> ::= <英大文字> {<英大文字>|<数字>}
| <英小文字> {<英小文字>|<数字>} (2-28)

GOTO と GOSUB は後ろに数字を含む文字列を名前付ける。ラベルの識別子は「:」で表し後ろに名前を付け、サブルーチンの識別子は「SUB」でラベルと同様に後ろに名前を付ける。中間コードに変換する際、SUB には識別コードに RETURN のアドレスを加える。これは GOSUB から呼ばれずに SUB の位置に実行ラインが来た時 RETURN の後ろまで飛ばすためである。

以下のプログラムの実行結果を図 2-4 に示す。サブルーチンや GOTO 文の処理が正しく動作していることが確認できる。

ソースコード 2-16 デモプログラム(test-demo.bas)

```
GOSUB abc
print "a+b=",a+b
GOTO label1
print "a-b=",a-b
:label1
SUB abc
print "a = ", a=12*(c=4+513)
print "b = ", b=12*(4+5)-((4+2)*10)
RETURN
```

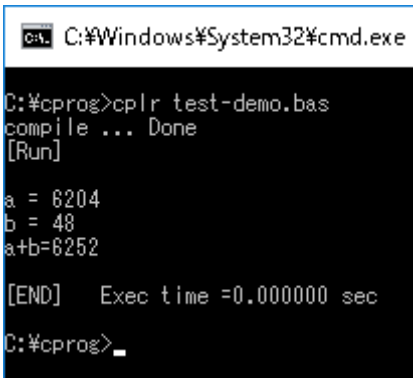

図 2-4 実行結果

第３章 マルチプラットフォーム

３-１異なる環境への対応

作成したインタプリタを搭載するアーキテクチャは PIC マイコン、ESP32、Raspberry Pi Zero W の 3 種類である。

1. PIC マイコン

PIC マイコンを用いた端末の回路図を図３-1 に示す。

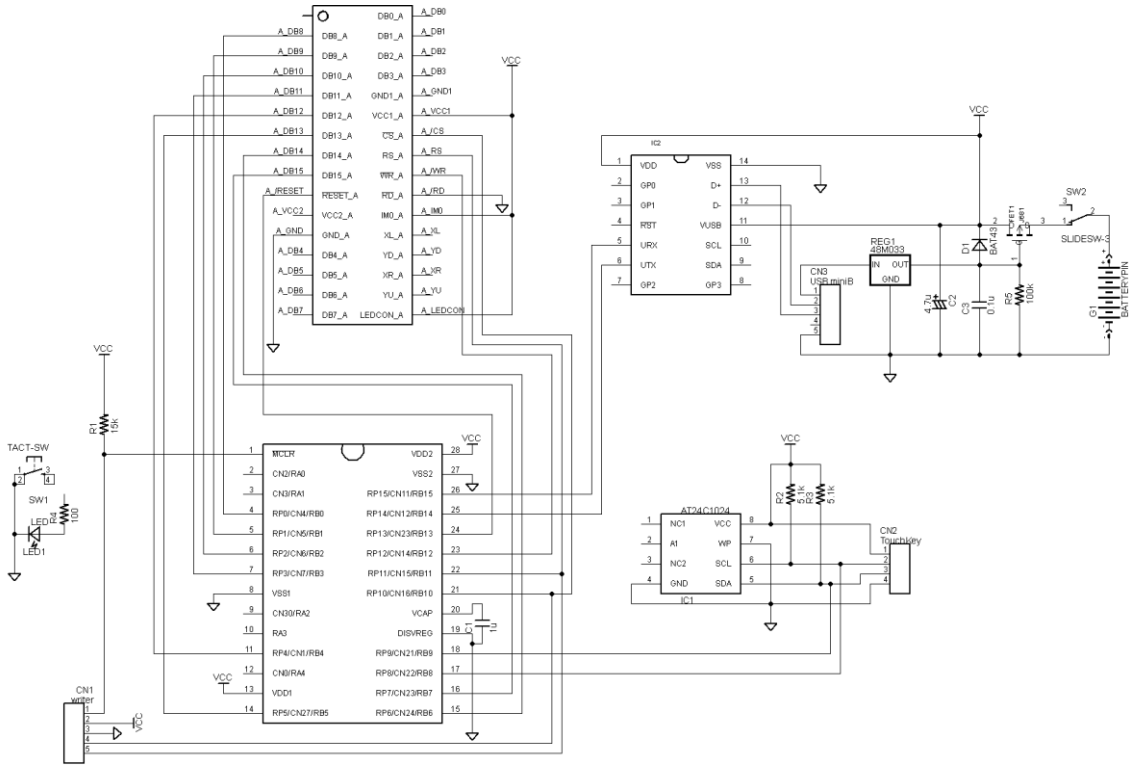

図３-1　PIC マイコンを用いたインタプリタデバイス

PIC マイコンは、他の PIC マイコンよりメモリーが豊富な PIC24FJ64GA002(回路図の左下)を使用し、出力装置はグラフィックカラー液晶表示器 LCM240YP04-07 (回路図の左上)、入力装置にはタッチ式キーボードを I2C 接続できるようにした(回路図の右下のコネクタ)。また、PC と接続して操作できるように USB－UART 変換 IC の MCP2221A を搭載した(回路図の右上)。

タッチ式キーボードは学部の卒業研究で作成した電卓 [4]で使用したものと同じハードウェアに制御アルゴリズムを一部改良して使用した。このキーボードは電卓用の数字と演算ボタン入力用に 25 個のキーしかないのでスマートフォンで用いられているスライド操作の組み合わせによる文字入力ができるようにキーフリップ方式にした。

ESC												LIST
												RUN
			/			C			F			
		@	.	_	B	A	#	E	D	%	BS	BS
			1			2			3			
Home			I			L			O			End
←	→	H	G	(	K	J	&	N	M	)	←	→
			4			5			6			
			R			V			Y			
		Q	P	S	U	T	=	X	W	Z		
			7			8			9			
			‘			0			?			¥f
A/1		Aa	SP		+	-	*	,	.	!	¥r	return

図３-2 各キーに割り当てられた文字(アルファベット)

このキーボードの制御アルゴリズムは、最初にタッチされたキーと最後に触れていたキーの座標を読み、スライドした方向と距離を検出する。文字の入力では最初にタッチされたキーの座標とスライド方向で文字を識別し、カーソルキーやバックスペースキーの場合はスライド距離に応じてキー入力の回数を変化させることも可能である。さらにファンクションキーを設定して数字入力用のキーボードに切り替える機能も搭載した。このキーボードは様々なデバイスに接続して使用することを想定し、キーの座標を取得するモードやASCII コードに変換した状態で取得できるモードを搭載し、接続時にコマンドを送って切り替えることができるようにした。

図３-2 のキー割り当てで、右上の「LIST」と「RUN」や下の左から 2 番目の「Aa」はインタプリタで使用するために本体側のプログラムで置き換えたものである。 左下の「A/1」キーが押されたときは、キーボードの設定を数字と記号が入力できるモードに切り替えるように、本体側もプログラムでキーボードにコマンドを送ることでナンバーロックキーのような動作をするようにした。とにかく、キーボードやパソコンなどの入力装置がそばに無いときに、とりあえず、スタンドアローンで入力表示ができる構成を考え、この構成の利用法としてはちょっと持ち運びした際に I2C からのデータが欲しい時などその場でプログラムを組み、実行できることを想定している。

液晶表示器は画面を横向きにした状態で左上を基準とし、右方向に X, 下方向に Y で座標を指定して制御する方式にした。この液晶表示器のコネクタが手に入らなかったので図３-3 に示す 600mil 幅の DIP 変換基板を作成した。

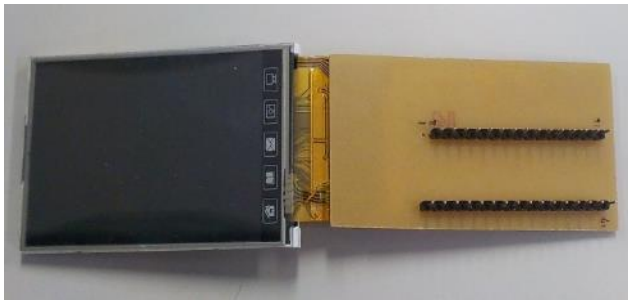

図３-3 DIP 変換基板

色の情報は 3byte 使用するものと 2byte で表現する方式を選ぶことが可能で、緑を 6bit 赤と青をそれぞれ 5bit の 2byte で表現する方式を選択した。データの転送方法には 16bit モードと 8bit モードがあるが、表３-1 画像の表示に必要なピン数と初期化に要する時間を見ると、速度の差が 10%程度しか変わらないのでピン数の少ない 8bit モードで使用することにした。

表３-1 データ転送方式の比較

	16bit モード	8bit モード
画像の表示に必要なピン数	20	12
初期化データの転送時間[ms]	1405	1560

文字は 8×16 の ASCII フォントで、I2C で使用できる EEPROM 内(図３-1 の右下 AT24C1024)に格納し、ASCII コードをフォントのアドレスに変換し EEPROM からデータを読み出して表示する。画像を表示する場合は別の EEPROM を I2C で接続する必要がある。

図３-4 に完成した端末の全体像を示す。

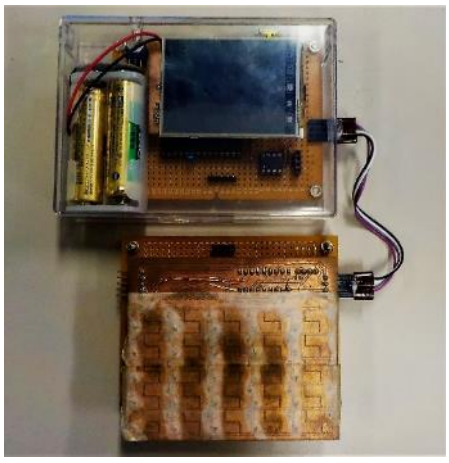

図３-4 端末の全体像

2.　ESP32

ESP32 は Wi-Fi と Bluetooth を内蔵したユニットで、これに電源と USB シリアル通信機能を追加したデバイスモジュールを使用した。

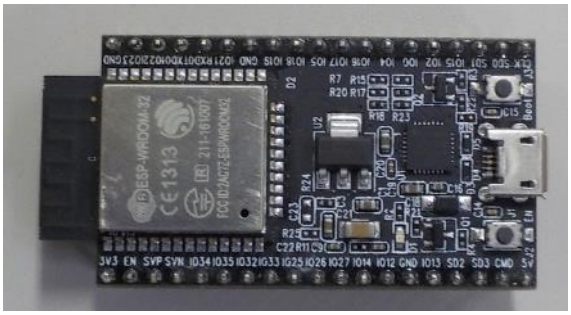

図 3-5 ESP32 Toolkit V2

プログラムの開発環境は Arduino IDE に対応しており、Bluetooth の一部の機能を除いた殆どの機能に対するライブラリが用意されている。すべての機能を使用するには専用の開発環境 ESP-IDF が必要で、操作がやや複雑になり開発に時間が必要だと判断して使用を断念したが、逆に言えば、Arduino IDE を利用して開発したので、ある程度の機能は(Bluetooth と FreeRTOS を利用している部分は駄目であるが)Arduino でも実装可能であることを意味する。

インタプリタは USB シリアルか Bluetooth シリアルを使用してパソコンから入出力を行い操作する。

3.　Raspberry Pi Zero W

Raspberry Pi Zero W は小型でパソコンのような処理能力を持ち GPIO の制御も可能なデバイスで、その中でも特にコンパクトで Wi-Fi と Bluetooth も搭載した Raspberry Pi Zero W を使用することにした(図 3-6)。この Raspberry Pi には Linux OS の Raspbian が入っており、インタプリタもこの OS 上で動作させる。開発環境は、最初は GUI の Code::Blocks を利用していたが、想像以上に重かったので LX Terminal 上で直接 gcc を用いてコンパイルすることで開発を行った。

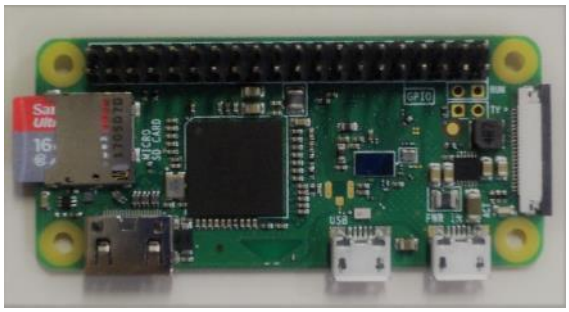

図 3-6 Raspberry Pi Zero-W

3-2 共通する機能

これらのデバイスにインタプリタを移植する際、メモリーのサイズや文字の入出力、乱数の生成方式などの違いを無くす必要がある。そこで、すべてのシステムで共通して使用する basic_com.c では環境に依存する部分をマクロで置き換えられるようにし、iostream.h ファイルで設定することにした。

ソースコード 3-1　データサイズの設定

```
#define IBUF_SIZE 64     //中間コード数
#define LIST_SIZE 1024  //プログラムメモリーサイズ
#define LSTK_SIZE 30     //FOR (5+1), GOSUB (2+1)用スタックサイズ
#define ARAY_SIZE 64     //配列サイズ
#define ESC_CODE 'q'     //ESC key code
```

ソースコード 3-1 は Windows 版の設定である。IBUF_SIZE は 1 行に格納できる中間コードの数で、LIST_SIZE は 1 つのプログラムを格納するメモリーのサイズである。PC や Raspberry Pi では十分なメモリーサイズがあるので制限はないが、他のアーキテクチャと条件を揃えるためにあえて IBUF_SIZE=64、LIST_SIZE=1024 に設定している。尚、行番号を 2byte の符号付き整数で表現しているのでリストの行数には制限がある。

LSTK_SIZE は FOR と GOSUB で使用するスタックのサイズで、FOR は中間コードと行のポインタ、カウンタ変数の変数番号及び終了値と増分値の 5 つに FOR の識別コードを加えた 6byte、GOSUB では中間コードと行のポインタに識別コードを加えた 3byte が必要となるのでそれぞれの倍数となるように設定した。スタックサイズが 30byte では FOR のみの場合で 5 回ネストすることが可能となる。ARAY_SIZE はインタプリタで使用する配列のサイズである。ESC_CODE はプログラムの実行を中止する際に使用するキーで「q」に設定し、PIC マイコン版のタッチ式キーボードでは左上の「Esc」キーに設定した。ただし、シリアル通信で PC から入力する場合は「q」で停止するように「Esc」変換する処理を加えた。

入出力関数はすべてのアーキテクチャで仕様が異なるのでそれぞれの設定が必要である。

ソースコード 3-2　文字入出力関数の設定

```
#define b_Kbhit() kbhit()
#define b_Puts(s) printf("%s",s)
#define b_Putchar(ch) putchar(ch)
#define b_Getchar() getchar()
#define b_Gets(s) gets(s)
```

ソースコード 3-2 は Windows 版の標準入出力関数である。kbhit はキーボード入力を監視し、何らかのキーを押された場合に 0 以外の値を返す関数で conio.h をインクルードする

ことで使用できる。Raspberry Pi ではkbhitに相当する関数がないので、termios.h と fcntl.h をインクルードして fcntl 関数でフラグを操作して未読文字を確認する方法を取った。PIC マイコン場合はタッチ式キーボードのバッファを読み取ることで入力の有無を確認でき、ESP32 では Serial.Available が kbhit の代わりとなる。

乱数についてはシード値の設定が異なる。

ソースコード３-3　各アーキテクチャでの設定(抜粋)

```
#define RAND_SEED srand((unsigned int) time(0));   //Windows、Raspberry Pi
#define RAND_SEED srand(milisec(0));               //PIC24
#define RAND_SEED randomSeed(analogRead(0)); //ESP32
```

ソースコード３-3 はそれぞれのソースファイルからシード値の設定部分を抜粋したものである。Windows、Raspberry Pi はスタンダードライブラリの srand 関数とシステムの時間を取得する time 関数を使用し、PIC マイコンではシード値に 1ms ごとにカウントしている変数の値をセットする。Arduino IDE には randomSeed 関数が用意されているのでアナログ入力の値をセットしている。シード値のセットはシステムを起動してから最初にコマンドを入力されたとき Basic 関数で行われる。

この他、幾つかのアーキテクチャ間で共通させた機能としてプログラムのセーブ機能と一定時間動作を止める時間待ち命令を追加した。

セーブ機能は Windows、Raspberry Pi ではプログラムをファイルに保存し、PIC マイコンでは EEPROM 上に簡易的なファイルシステムを構築して中間コードを保存する。命令は以下の通りである。

表３-2 セーブ機能に関する命令

命令	機能
SAVE "*ファイルパス*"	リストまたは中間コードを保存する
LOAD "*ファイルパス*"	保存されたデータをリストバッファに読み込む
LS ["*ファイルパス*"]	保存されたファイルの表示
RM "*ファイル名*"	ファイルを消す(PIC 専用)
FORMAT "*ファイル名*"	EEPROM を初期化(PIC 専用)

SAVE 命令は Windows と Raspberry Pi の場合はオプションの文字列に指定したファイル名でプログラムを保存することができる。ファイルの中身は中間コードを元のソースリストに変換してテキスト形式で保存されているファイル拡張子に「.bas」を自動的に付ける。また、ファイル名を指定するときに拡張子に「.bin」を付けると中間コードをそのままファイルにして保存する。PIC マイコンではファイル名を 8 文字以内で指定して中間コードを保存する。LOAD 命令は指定したファイルを読み込んで中間コードに変換してリストバッファに格納する。Windows と Raspberry Pi では SAVE 命令と同様に「.bin」を付けた場合は中間コードがそのままリストバッファに展開される。LS はカレントディレクトリ又は

指定したディレクトリのファイルを一覧表示する命令で、Windows と Raspberry Pi ではスタンダードライブラリの system 関数を用いてシステムコマンドを実行している。PIC マイコンでは I2C バスにリスト保存用の EEPROM を追加して SAVE と LOAD を行えるようにしており、以下に示す構造体を定義してリストのインデックスを EEPROM の前半に

ソースコード 3-4 リストインデックスの構造体

```
typedef struct{
        unsigned char back;      //前のインデックスのアドレス
        unsigned char next;      //次のインデックスのアドレス
        unsigned short siz;      //リストデータサイズ
        unsigned char name[8];   //リストの名前
} List_index ;
```

格納し、後半にリストのデータを中間コード状態で格納する。

List_index 構造体には前後のインデックスのアドレスとリストのサイズと名前の情報を持ち、LS 命令でリストファイルの一覧表示をする際に名前の順で表示するようにした。インデックスのアドレスとリストのデータのアドレスはリンクしており、SAVE や LOAD を実行する際に前後のアドレス情報を変更することでリストファイルの挿入や削除を実現している。尚 EEPROM を最初に使用するときは内部のデータを初期化する必要があるので、PIC マイコン版だけ FORMAT 命令を追加している。ファイルを削除する命令は PC や Raspberry Pi ではエクスプローラーやファイルマネージャーなどがあるので PIC マイコン版にだけ追加した。

wait 機能は、PIC マイコン、ESP32、Raspberry Pi にそれぞれ異なる方法で追加した。

PIC マイコンでは CPU の処理が占領されてしまうことを防ぐために以下に示す関数とタイマー割り込みを用いて時間待ちをする。

ソースコード 3-5 PIC

```
void Wait(unsigned int t){
        dt0 = t;
        while(dt0){}
}
```

ソースコード 3-5 の dt0 はタイマー割り込みで 1ms ごとに 0 になるまでデクリメントする変数で、この変数に待ち時間をセットして 0 になるまで待つ。ESP32 の場合は後述するリアルタイム OS によりタスクが分けられるので Delay 関数を用いても他の機能に影響しない。Raspberry Pi では usleep 関数を用いることで指定した時間だけタスクを他に渡すので CPU の処理が占有せずに時間待ちができる。

3-3 I2C デバイスの作製

PIC マイコン、ESP32、Raspberry Pi で共通して制御できる対象の例として 24bitA/D コンバータを搭載した重量測定器を制作した。重量測定器の回路図を図 3-7 に示す。

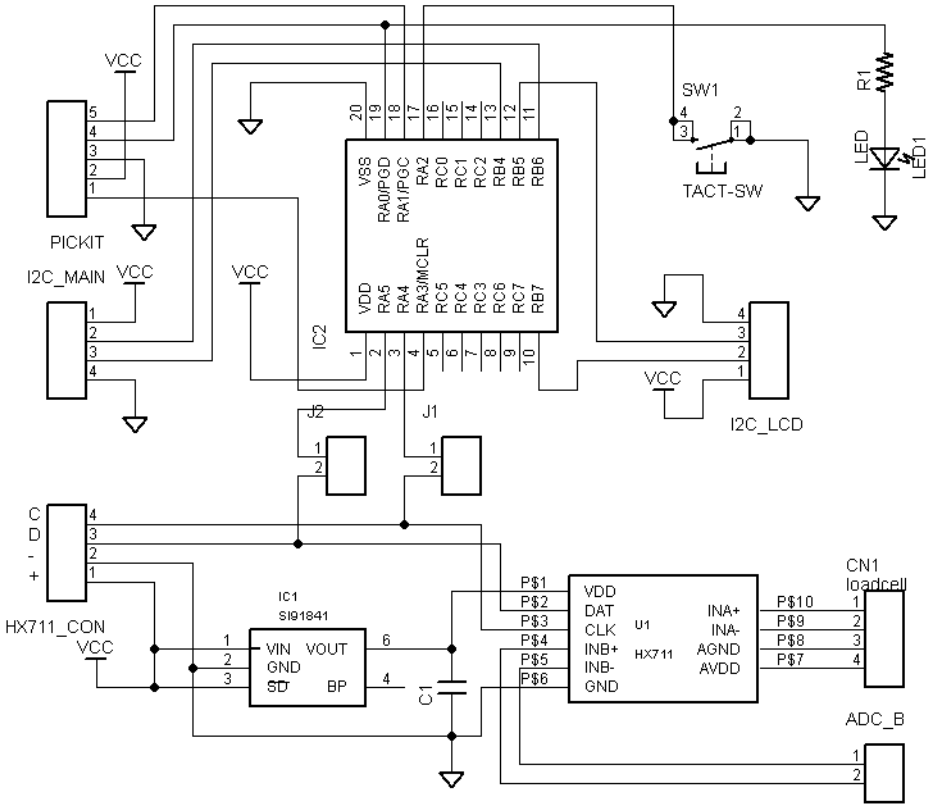

図 3-7 重量測定器の回路図

使用したセンサーはロードセル SC616C (500g)で 24bitA/D コンバータは HX711 を搭載したロードセル用モジュール(回路図の右下)を使用した。HX711 には 2ch の差動入力があり、ポート A には Gain=128、ポート B には Gain=64 のアンプが内蔵されているので、より高い分解能で計測するためにポート A を使用し、回路図の CN1 にロードセルを接続した。また、AD コンバータの電源を安定化させるために低ノイズレギュレーターIC の SI91841DT-285 を使用している(回路図の左下)。制御には I2C ポートを 2 つ使用できる PIC16F1827 を使用し、マスターとなって表示器 (SO1602AWGB-UC-WB)の表示を制御しながらスレーブとして外部からの操作を受け付けるようにした。尚スレーブアドレスは 0x0E とした。

以下の式は、ロードセルの荷重に対する出力電圧の変化量を示す。

$$dv = \frac{V_{OUT}}{LOAD} * VDD \qquad (3\text{-}1)$$

dv は グラム当たりの出力電圧の変化(mV/g)、V_{OUT}は最大重量LOAD(g)を加えたときの電源電圧VDD(V)に対する電圧の変化(mV/V)を表している。SC616C の場合はLOADは 500g、

V_{OUT}は 0.7 mV/V で、電源電圧は HX711 モジュールが生成する基準電圧ではなく SI91841DT-285 の出力を直結しており、実測値が 2.783V であったので,

$dv = 3.8962 \times 10^{-3}$となる。 これは 1g の変化に対して 3.8962μV 電圧が変化するという意味である。ここで、HX711 モジュールの基準電圧は図 3-8 の R1 を 20ｋΩから 10ｋΩに変更したときの値である。

I2C で送るデータは 5byte で、最初の 1byte に測定器の状態を表す情報を送り、次の 4byte で重さを 100 倍した値を送る。

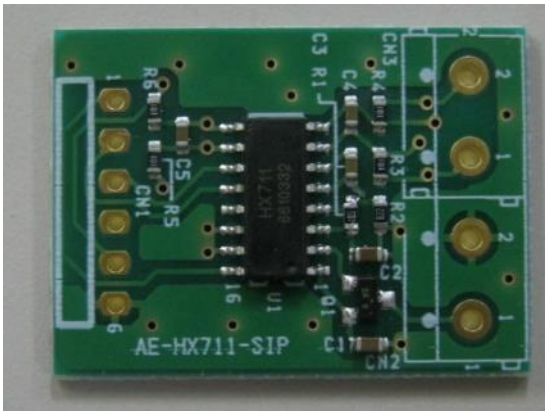

図 3-8 HX711 モジュール

図 3-9 完成した測定器の外観と使用例

図 3-9 は完成した測定器で硬貨の重さを計測している例である。上の行に重さを表示し、下の行にはロードセルの電圧変化の値を表示している。測定結果は±0.05g の範囲で変動するが、0.1g の精度で測定することができた。この値は汎用の電子測りと比較して同じ値であることが求められた。

外部と接続するためのピンソケットは左側面に設置した。

インタプリタで I2C デバイスを制御するために以下の命令を追加した。

表 3-3 I2C デバイスの制御命令

命令	機能
WIRER(*ID*, *size*)	I2C デバイスからデータを読む
WIREW(*ID*, *size*)	I2C デバイスにデータを送る
WIRE	接続されている I2C デバイスの ID を表示

WIRER はデバイスから指定した byte 数のデータを読み配列に格納する関数で、WIREW はデバイスに指定した byte 数のデータを書き込む関数である。引数の ID には I2C デバイスのスレーブアドレスを右シフトして R/W ビットを除いた 10 進数で指定し、size には通信するデータ数を byte 単位で指定する。データを格納する配列は本来ならば専用の配列を用意するべきだが、識別子の都合上使用できる配列が 1 種類しかないためインタプリタの

配列と併用している。

WIRE は接続されている I2C デバイスの ID を一覧表示する命令である。PIC マイコン、と ESP32 では接続されているもののみを表示し、Raspberry Pi ではシステムコマンドを呼び出している関係から、すべてのアドレスの状態が表示される。

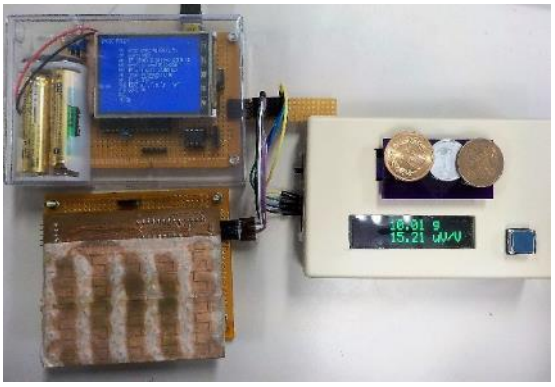

図 3-10 接続例

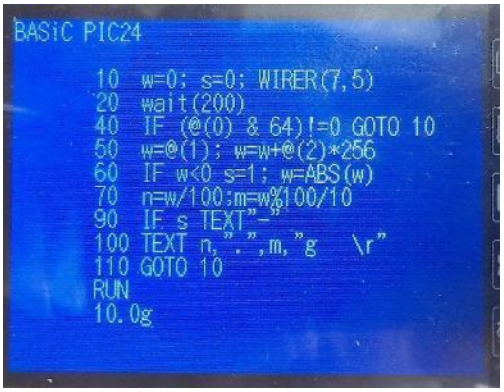

図 3-11 実行結果

図 3-10 は PIC マイコンに接続した例で、電源はインタプリタデバイスから供給している。

図 3-11 は制御プログラムを書いて実行した画面である。最初の行で変数を初期化し、測定器のスレーブアドレスの 7 をセットして 5byte 読む。200ms 待ち、40 の行で測定値が返されなかった場合は最初の行に戻り、測定値が返された場合は次の処理に進む。50 から 70 の行でデータの復元と最下位桁の丸め込みをして TEXT 命令で 1 ラインに繰り返し上書き表示する。表示された値はほとんど変動することなく安定して表示されている。

第４章 環境依存命令

４-１PIC マイコン

PIC マイコンにはグラフィック液晶表示器を搭載したので表４-1 に示すグラフィック系の命令を用意した。

表４-1 グラフィック系の命令

命令	機能
PDOT (*X, Y*)	点をプロットする
PLIN (*X0, Y0, X1, Y1*)	直線をプロットする
LABEL(*X, Y, ”string”, ...*)	任意の位置に文字列を表示
PCOL(*color*)	色の設定
DRAW	背景を消す
GCLS	リセットして通常のコンソール画面にする

PDOT は画面の指定した位置に点をプロットする関数で、PLIN は始点と終点を指定して線を引く関数である。LABEL は始点を任意の座標に設定して文字列を表示する命令で、PRINT 文のように変数や文字列を複数個パラメータにセットできる。これらの命令が持つ引数の座標は左上を原点として右方向に X、下方向に Y が増加する。PCOL は点や線の色を設定する関数で、色は 0 から 7 までの番号で指定して表４-2 に示す色の中から選択する。DRAW は画面を消して背景を黒、前景色を白に設定し、GCLS は元のコンソール画面に戻す命令である。

表４-2 色の番号と値

番号	色	値	番号	色	値
0	BLACK	0x0000	4	MAGENTA	0xf81f
1	RED	0xf800	5	YELLOW	0xffe0
2	LIME	0x07e0	6	CYAN	0x07ff
3	BLUE	0x001f	7	WHITE	0xffff

ソースコード 4-1　グラフィックプログラムのリスト

```
10 DRAW a=0
40 FOR i=0 TO 6
50 FOR x=-100 TO 100 GOSUB 110 NEXT
60 PCOL(i+1); a=a+20
70 NEXT
80 PAUSE STOP
110 y=x*x/50
120 PLIN(x+100, 120, x+120+a, y)
130 RETURN
```

図 4-1 表示例

図 4-1 はソースコード 4-1 に示すリストを実行して図形を表示した例である。DRAW 命令で画面の背景を黒くして描画エリアを初期化し、同じ行で変数の初期化も行っている。表示している図形は 2 次関数と座標(x,0)を結ぶ線の軌跡で、図形の描画をサブルーチンにして、メインのループで色を変えながら GOSUB で呼びだしている。

4 - 2 ESP32

ESP32 のモジュールには GPIO 端子があるので表 4-3 に示す GPIO 制御命令を追加した。

これらの命令は Arduino IDE の pinMode、digitalWrite、digitalRead、analogRead 関数を用いて容易に実現できた。PMOD は Pin に GPIO ピンの番号をセットして mode で入力か出力を指定する。mode の値には定数「IN」と「OUT」が使用できる。GPR と GPAR は指定した GPIO ピンの値を返す関数で、GPW は value に 1 か 0 をセットして GPIO ピンの出力を制御できる。

表 4-3 GPIO 制御命令

命令	機能
PMOD(*Pin*, *mode*)	GPIO ピンの入出力モードを設定
GPR(*Pin*)	GPIO ピンの値を読む
GPW(*Pin*, *Value*)	GPIO ピンを出力する
GPAR(*Pin*)	GPIO ピンのアナログ値を読む

ESP32 モジュールの GPIO ピンは機能が限られており入力専用のピンや使用できないピンがあり、間違った設定が入力された場合に警告文を表示させる機能が必要だが、プログラムの肥大化を避けるために付けなかった。

ESP32 ではタイマー割り込みが使用できるのでタイマー割り込みルーチンを使用しで 4 桁の 7segLED をダイナミック制御するアプリケーションを搭載することにした。図 4-2 に ESP32 モジュール委接続するデバイスの回路図を示す。回路にはジョイスティックとスイッチも追加した。

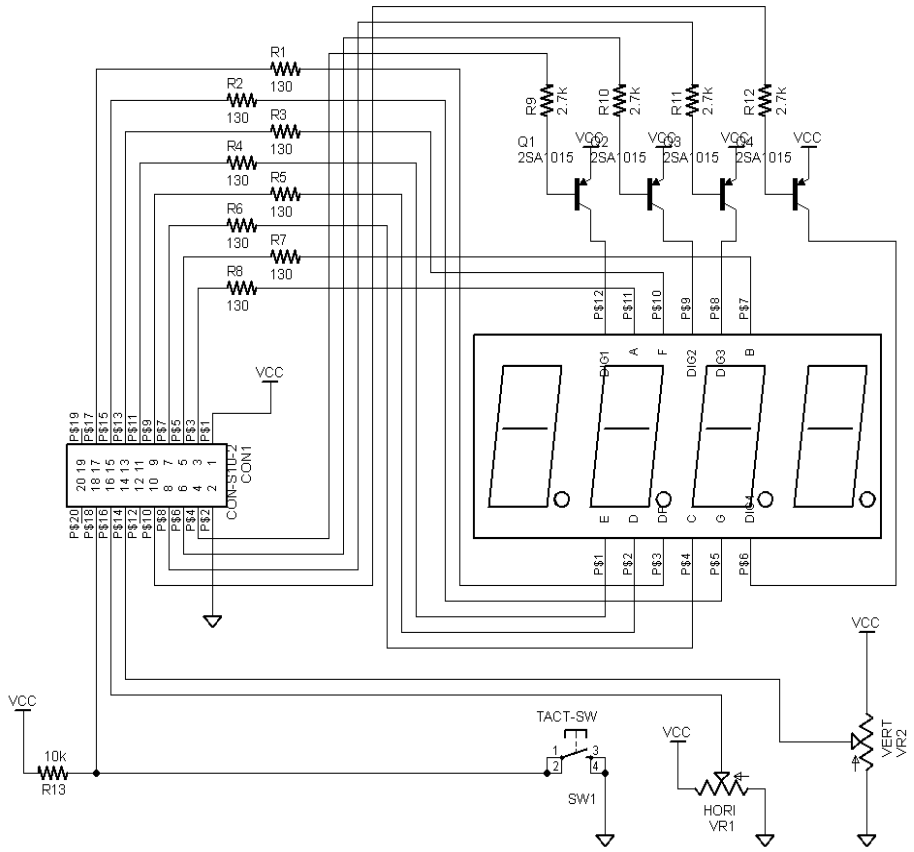

図 4-2 追加デバイスの回路図

表 4-4　7seg LED とスイッチの制御命令

命令	機能
SEGEN (*mode*)	7seg LED の on/off 設定
SEGSET (*Value*)	7seg LED に表示する値をセット
SWSET (*Pin*)	スイッチを接続したピンの設定
SWREAD(*type*)	設定スイッチの操作を検出する

表 4-4 は追加したデバイス用の制御命令である。SEGEN は引数に 1 をセットすると 7segLED のピンモードの設定を行い表示用の変数に格納された値を表示し、0 をセットすると 7segLED の点灯制御を停止する。SEGSET は表示用の変数に値を格納する命令で値を変更するまで表示し続ける。

SWSET と SWREAD はスイッチ操作検出を容易にするための命令である。SWSET は

指定したピンの値を監視し、SWREAD で値の変化を検出するための命令で、SWREAD は Type で立ち上がり、立下り又はその両方の 3 種類を 1 から 3 の定数で指定し、この命令が実行されるごとにサンプリングして変化を検出したときに 1 を返す命令である。

以下に完成したデバイスと動作例を示す。

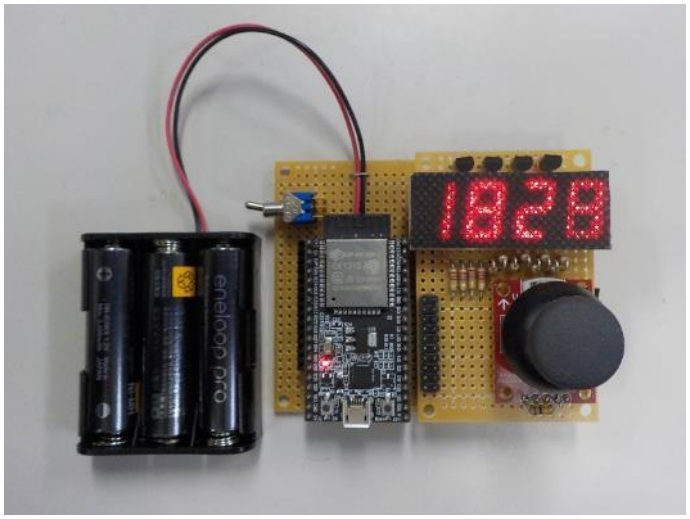

図 4-3　完成したデバイスの動作例

4 - 3 Raspberry Pi

Raspberry Pi には 40 ピンのコネクタがあり、そのうちの 28 本を GPIO として使用できるので ESP32 と同様に GPIO 制御命令を追加した。Raspberry Pi の GPIO はアナログ入出力に対応していないので ESP32 と共通しているのは PMOD、GPR、GPW の 3 つである。

Raspberry Pi で GPIO を制御する方法には 2 通りあり、1 つ目はデバイスドライバーを用いる方法で GPIO へのアクセス速度が 2 つ目の方法に比べて遅くなるが安全な制御が可能で、2 つ目の方法は GPIO の制御レジスタを直接操作する方法で、アクセス速度が高速で複数のポートを同時に操作できるが仮想メモリー空間から物理メモリーへ変換する操作が必要になる [5]。

ここではデバイスドライバーを用いて制御することにした。操作手順は次のとおりである。

1. /sys/class/gpio/export に操作する GPIO のポート番号を書き込む
2. /sys/class/gpio/gpio *ポート番号*/direction　に“in”か”out”を書き込む
3. /sys/class/gpio/gpio *ポート番号*/value　で GPIO の値を読み書きする
4. /sys/class/gpio/unexport に操作を終了する GPIO のポート番号を書き込む

手順 1 と 2 の操作をまとめると PMOD 命令の動作となり、3 の操作で GPR と GPW 命令の動作を実現できる。手順 4 は GPIO の制御を終了する際に必要な操作で、プログラムの終了時などで使用する PEND といる命令を追加した。

第５章 無線制御の構築

５-１ ESP32 を用いた Bluetooth 接続と Free-RTOS の適用

Arduino IDE では ESP32 の Bluetooth 機能の内で Bluetooth シリアル通信のライブラリには対応していたので端末の入出力に使用することにした。BluetoothSerial.h をインクルードすると SerialBT クラスで通常の Serial クラスと同様に available や print といったメソッドが使用できる。USB と Bluetooth のどちらからでも操作するためにソースコード５-1 のような関数を作成した。

ソースコード５-1 Bluetooth と USB の両方から１文字受け取る関数

```
byte Serial_Getchar(){
        byte res=0;
        if(SerialBT.available()) res = SerialBT.read();
        if(Serial && Serial.available()) res = Serial.read();
        return res;
}
```

ESP32 では Free-RTOS が動作しており、Bluetooth や Wi-Fi の機能を別タスクで実行している [6]。

そこでインタプリタプログラムをタスクで実行することにした。タスクはソースコード５-2 に示す関数で作成する。

ソースコード５-2 タスク生成関数

```
xTaskCreatePinnedToCore(vTask_BASIC, "BASIC", //タスクのポインタと名前
                        0x2000,       //スタックのサイズ
                        NULL,         //タスクのパラメータ
                        10,           //タスクの優先レベル
                        NULL,         //タスクの参照用ハンドル
                        0);           //戻り値
```

vTask_BASIC 関数にメインループ処理を置いておき、xTaskCreatePinnedToCore でタスクを生成すると自動的にタスクスケジュールが実行される。1 つだけ修正しなければならなかったのは文字が入力されるまで待つループにタスクディレイを入れて定期的に別のタスクに処理を移す必要があったことである。

５-２ Raspberry Pi の無線 LAN 接続

Raspberry Pi を PC から操作するために学内無線 LAN に接続することを試みた。学内無線 LAN は認証プロトコルが TTLS 方式でユーザー名とパスワードによる認証が必要で

あるが、Raspberry Pi の GUI アプリケーションではユーザー名とパスワードを入力することができない。そこで wpa_supplicant の設定を直接操作して学内無線 LAN に接続することができた。

設定方法は、/etc/wpa_supplicant/wpa_supplicant.conf をテキストエディタで開いて以下の記述を加える

```
network={
        ssid="kut"
        scan_ssid=1
        key_mgmt=WPA-EAP
        proto=WPA2
        pairwise=CCMP
        group=CCMP
        eap=TTLS
        phase2="auth=PAP"
        identity="ユーザー名"
        password="パスワード"
}
```

変更を保存したら以下に示すように kill all コマンドで wpa_supplicant を停止し、次のコマンドで再起動することで設定が有効になり学内無線 LAN に接続される。(「＞」はプロンプトである)

＞ sudo kill all wpa_supplicant

＞ sudo wpa_supplicant -i wlan0 -c /etc/wpa_supplicant/wpa_supplicant.conf &

これによって、学内無線 LAN 経由で PC から Raspberry Pi を SSH などによる操作が可能になった。

次に Raspberry Pi から Bluetooth で ESP32 に接続する設定を行った。ESP32 モジュールとのペアリングは bluetoothctl や GUI アプリケーションで可能である。ペアリングを完了すると以下のコマンドでデバイスアドレスとチャンネルの確認を行った。

＞ sudo sdptool search SP

ここで Serial Port というサービスがない場合は以下のコマンドで追加する必要がある。

＞ sudo sdptool add SP

次にシリアルコンソールから操作するために/dev/rfcomm*にデバイスの ID を紐づけする。rfcomm0 にチャンネル 2 の ID＝24:0A:C4:XX:XX:XX を紐づける場合

＞ sudo rfcomm bind 0 24:0A:C4:XX:XX:XX 2

最後に minicom に-s オプションを付けて起動し、ボーレートの設定をすると ESP32 モジュールに接続できる。尚、ここで Ctrl-A Z とキーを入力して設定画面を開いて改行コードに CR を追加する必要がある。また、ファイル転送機能で一度にソースコードを送る場

合、Ctrl-A Z T とキーを入力して送信時に 1 文字当たり 10ms 程度のディレイを入れることで通信エラーを軽減させた。

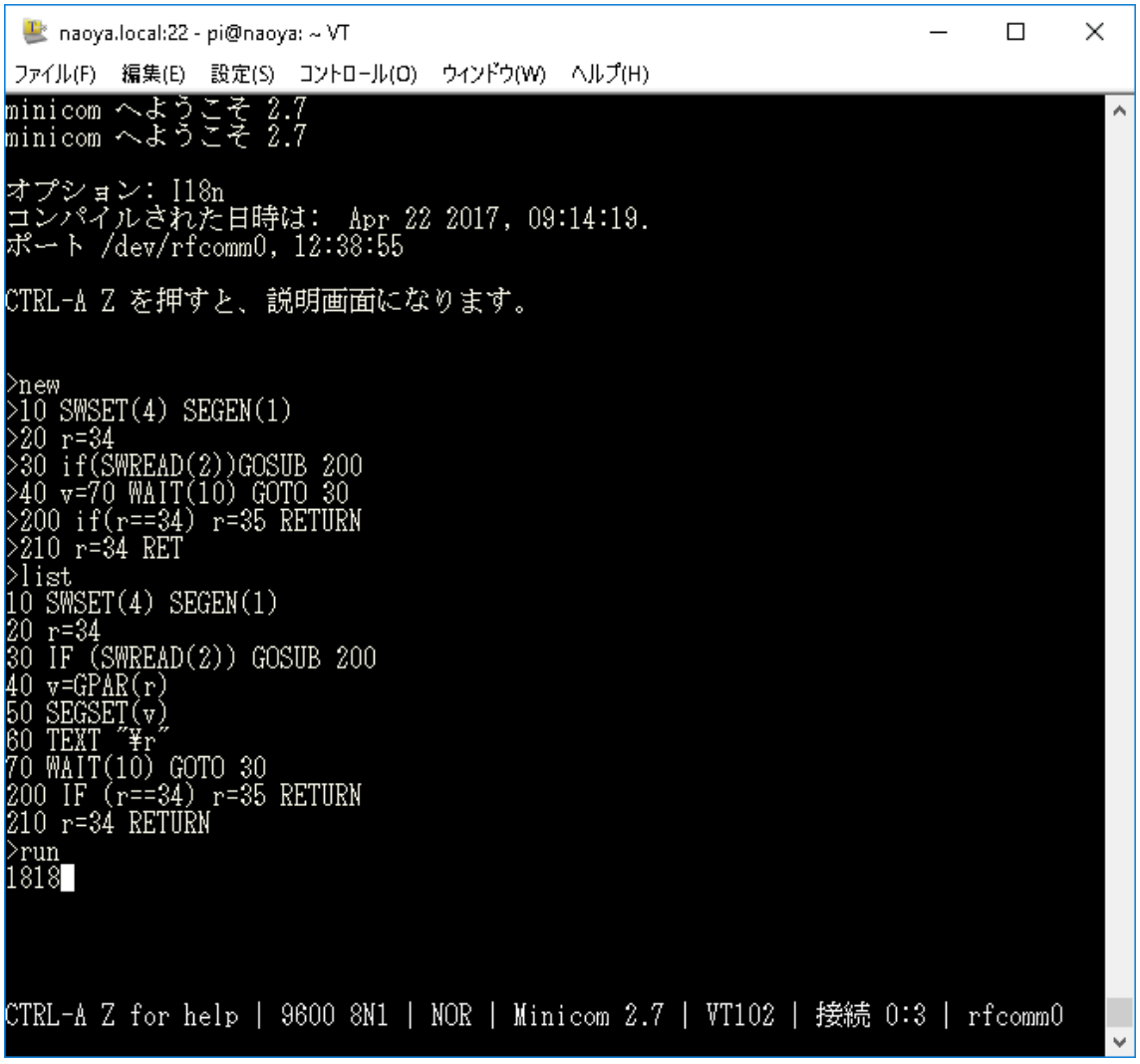

図 5-1 minicom の実行画面

図 5-1 は minicom を用いて Raspberry Pi から ESP32 を操作している例である。プログラムは Raspberry Pi に保存して置いたファイルを minicom のファイル送信機能で転送した。転送は、Ctrl-A Z S を押して送信するファイルのファイル名を指定することでデータが送られる。図では一部の行が表示されていないが LIST 命令を実行すると正しく転送されていることが確認できる。送信時に正しく表示をさせる場合は、改行の送信時に 100ms 以上のディレイを設定することで解決できた。

5-3 電池駆動方式の開発とシステム化

ESP32 モジュールを完全に独立させて動作させるために電池で駆動させることを試みた。ESP32 モジュールは USB からの電源を内部で 3.3V に落として駆動しているので、充電

式電池を 3 本使用しロードロップレギュレーターIC(NJM-2845DL1-33)で 3.3V を作り、ESP32 モジュールの電源ラインに接続することで駆動できることを確認した。

Raspberry Pi に関しても USB から 5V の電源を供給して動作させているが、実態は 3 端子レギュレータで 3.3V に電圧を落として使っているに過ぎない。ESP32 と異なり Raspberry Pi にはロードロップの 3.3V の 3 端子レギュレータが搭載されているので、直接 3.3V+0.3V(程度)、すなわち 3.6V 以上を 5V ラインに供給すれば 3.3V の電圧が内部でつくられ、無事動作することを確認した。そこで今回は公称電圧 1.2V のニッケル水素 2 次電池を 3 本直列にして 3.6V を作り、この電圧で Raspberry Pi を動作させることにした。選んだ Raspberry Pi は Zero W であり、Raspberry Pi3 に比べると大幅に消費電力が少ないので十分電池駆動が可能である。以上のことをまとめて 1 つのシステムを構築する。

次に本学内の無線 LAN を使って SSH で Tera Term が使える Windows PC に接続する。Raspberry Pi Zero W に搭載した BASIC はコマンドライン対応の CUI タイプなので SSH 接続で十分である。この SSH 接続を使って Raspberry Pi 上で GPIO を操作し 3 色 LED の点灯消灯を行う。Raspberry Pi は Linux OS 上で BASIC が起動しているので同時に複数のプログラムをマルチタスクで動かすことができる。この性質を利用して 2 つ目の SSH コンソールで I2C 接続した重量計を動かし、3 つ目の SSH コンソールで Bluetooth 接続した ESP32 とダイナミック 7segLED を制御する。この様子を図 5-2 と図 5-3 に示す。この様に Raspberry Pi Zero W、ESP32 を、無線 LAN を介して離れたところにある Windows PC で制御することが可能になった。より広い活用方法があるはずである。

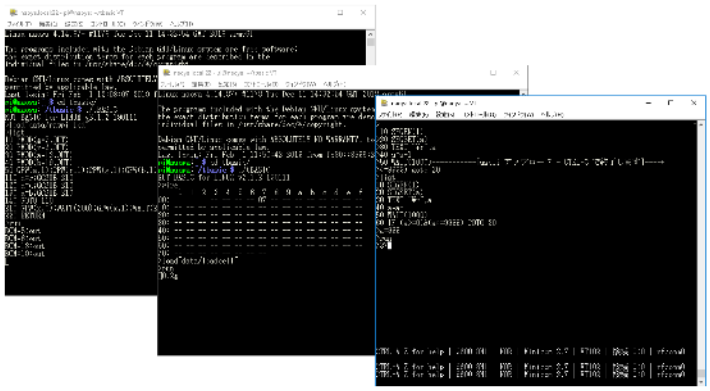

図 5-2 3 種類のプログラム制御画面

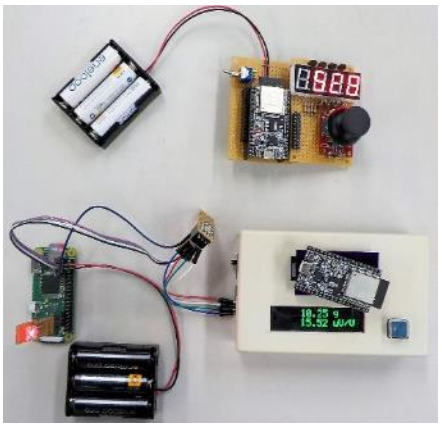

図 5-3 動作例

第６章 まとめ

本研究では PIC マイコン、ESP32、Raspberry Pi の３つのアーキテクチャで動作可能なインタプリタを作成し実際に動作することを確認できた。また、24bitAD コンバータを用いた重量計では 0.1g（1μV）の精度で測定が可能で、I2C 端末に測定結果を表示させることもできた。

Raspberry Pi では学内無線 LAN への接続と Bluetooth による ESP32 との接続を実現したことによって、無線 LAN に接続されていない ESP32 モジュールを Raspberry Pi 経由で PC から遠隔操作することが可能になった。

インタプリタの設計では機能を簡素化して確実に動作させることを優先させたので、現時点ではセンサーの値を処理して表示する程度の機能しか持たないのであまり実用的とは言えない。しかし PIC24 のプログラムメモリーでも空き容量が十分にあるので、行番号の廃止や変数を任意の名前にするといったことも実現できると思われる。

参考文献

[1] 鈴木哲哉, 古典電脳物語 8085,Z80,CP/M,タイニーBASIC... マイコン黎明期のコンピュータの自作法, 東京都千代田区: 株式会社ラトルズ, 2006.

[2] 中田育男, コンパイラ ー作りながら学ぶー, 東京都: 株式会社 オーム社, 2017.

[3] 鈴木哲哉, タイニーBASIC を C で書く, 東京都: ソシム株式会社, 2016.

[4] 上田直也, 卒業論文 ”*タッチ入力式オリジナル電卓の設計と製作*”, 2017.

[5] インターフェース編集部、桑野雅彦, お手軽 ARM コンピュータ ラズベリー・パイで I/O, 東京都: CQ 出版社, 2014.

[6] 石岡之也, “ESP32 標準搭載！注目 IoT マイコン OS「Free-RTOS」をはじめる,” *インターフェース,* 第 10 月号, p. 156, 2018.

謝辞

今回の修士論文の作成にあたり、終始に丁寧で熱心なご指導とご教示を賜りました高知工科大学大学院基盤工学専攻電子・光システム工学コース 綿森 道夫准教授に心からの感謝を申し上げます。本研究では綿森道夫准教授のお力添えなしでは完成に至ることはなかったと思います。副指導教員の星野 孝総准教授、橘　昌良教授にも感謝を申し上げます。

また、高知工科大学大学院基盤工学専攻電子・光システム工学コース在学中 本研究実験遂行や学生生活、その他様々な面で終始ご厚意頂きました、山本 利水教育講師や皆様には重ねて感謝の意を述べさせて頂きます。