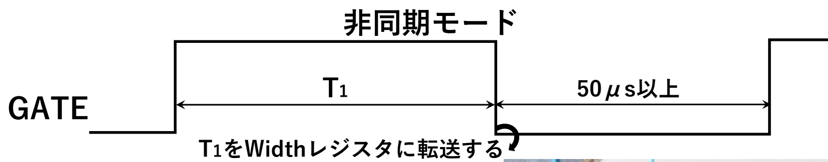
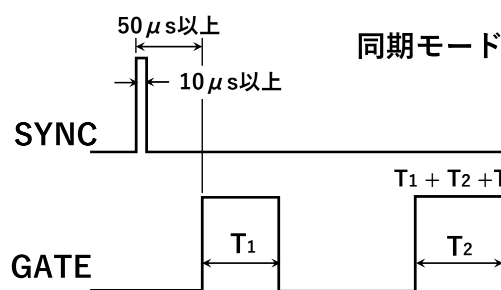
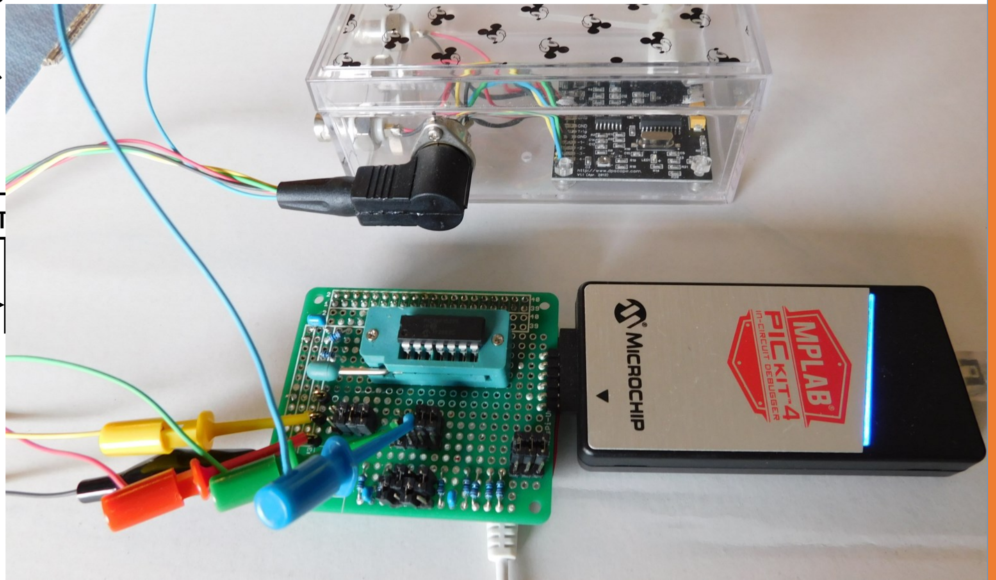
Raspberry Pi 派生プロジェクト

汎用測定チップPICCOLO

PICを使って専用チップを作る







initialize()
- init_time()
- init_versatility()

kground()
- time_command()
- versatile_command()
- voltage_measurement()

PICCOLOファームウエア

all_interrupts()
- i2c_isr()
  - app_layer()
    - read_t_status()
    - read_a_status()
    - write_t_command()
    - write_a_command()
    - read_width()
    - read_count()
    - read_data(m)
- sync_isr()
- period_isr()
- overflow1.isr()

Address	Name	Hex	Decimal	Binary
0192	SSP1CON3	0x40	64	01000000
020C	TMR1	0x0000	0	00000000
020C	TMR1L	0x00	0	00000000
020D	TMR1H	0x00	0	00000000
020E	T1CON	0x02	2	00000010
020F	T1GCON	0x84	132	10000100
0210	T1GATE	0x00	0	00000000
0211	T1CLK	0x05	5	00000101
028C	T2TMR	0x00	0	00000000
028D	T2PR	0xF7	247	11110111
028E	T2CON	0xF0	240	11110000
028F	T2HLT	0x00	0	00000000

7500 8000 8500 9000 9500 10000 10500 11000 11500 12000 12500 13000 13500 14000 14500
Instruction Cycles

著者　秋山忠次

（非売品）

目次

1 Raspberry Pi 派生プロジェクト　汎用測定チップ ...1
1.1 開発の背景...1
1.2 仕様を決める...1
1.2.1 要求仕様の分析...1
1.2.2 ベースにする PIC チップ...1
1.2.3 開発仕様の作成...2
1.3 開発環境を整える...2
1.3.1 開発環境と言語...2
1.3.2 PICkit による書き込みとデバグ...3
1.3.3 Raspberry Pi を利用した信号源...3
1.3.4 開発・評価ボード...3
2 開発仕様と実装設計...4
2.1 開発仕様...4
2.1.1 通信レジスタ...4
2.1.2 通信機能...4
2.1.3 時間測定機能...4
2.1.4 周波数測定機能...4
2.1.5 イベント計数機能...4
2.1.6 電圧測定機能...4
2.2 実装設計 I（測定機能）...4
2.2.1 入出力端子のアサイン...4
2.2.2 パルス幅測定機能...5
2.2.3 周波数測定機能...5
2.2.4 イベント計数機能...6
2.2.5 電圧測定機能...6
2.2.6 I2C 通信...6
2.3 実装設計 II（PIC MPU の使い方）...7
2.3.1 クロック周波数...7
2.3.2 Watch Dog Timer...7
2.3.3 割り込み処理...7
2.3.4 バックグランド処理...7
2.3.5 メモリマップ...8
2.3.6 命令アドレス...8
2.3.7 スタック...8
2.4 実装設計 III（ファームウエア）...8
2.4.1 PIC 基本部...8
2.4.2 パルス幅測定機能...9
2.4.3 周波数測定機能...10
2.4.4 イベント計数／電圧測定機能...12
2.4.5 バックグランド処理...13
2.4.6 I2C 通信機能...13
2.5 モジュール構成...16
3 モジュール作成...17
3.1 コーディング上の注意事項...17
3.1.1 C 言語の使い方について...17
3.1.2 モジュール化する...17
3.1.3 16 ビット SFR へのアクセスはマクロを使う...17
3.1.4 割込みの悪影響を防ぐ...18
3.1.5 割り込み要因の判別に注意する...18
3.1.6 使用する SFR は全て初期化する...18
3.1.7 機能に目が届く大きさにする...19
3.2 PIC 基本部...19
3.2.1 コンフィギュレーシ...19
3.2.2 インクルードファイル...19
基本部モジュール...21
3.3 時間測定機能...22
3.3.1 インクルードファイル...22
3.3.2 時間測定機能モジュール...24
3.4 汎用測定機能...26
3.4.1 インクルードファイル...26
3.4.2 汎用測定機能モジュール...27
2.1 I2C 通信機能...29
2.1.1 インクルードファイル...29
2.1.2 I2C 通信機能モジュール...30
2.2 ビルド結果...32
3. シミュレーションによる検証...33
3.1 シミュレーションによる検証の概要...33
3.1.1 ハードウエアのサポート...33
3.1.2 ブレークポイント...33
3.2 初期化...34
3.3 コマンド解釈...34
3.3.1 時間測定機能コマンド解釈の検証...34
3.3.2 汎用測定機能コマンド解釈の検証...35
3.3.3 電圧測定機能...35
3.4 パルス幅測定機能...36
3.4.1 SYNC 割り込み（同期モード）...36
3.4.2 GATE 割り込み（非同期モード）...36
3.4.3 オーバーフロー割り込み...36
3.5 周波数測定機能...36
3.5.1 パルスゲート回路...36
3.5.2 測定周期割り込み...37
3.5.3 オーバーフロー割り込み...37
3.6 イベント計数機能...37
3.6.1 NCO 割り込み...37
3.6.2 イベント計数...37
3.7 I2C 通信機能...37
3.7.1 I2C 割り込み処理の検証...38
3.8 割り込み処理時間...39
3.9 モニター信号のシミュレーション...39
4. 実チップ検証...40
4.1 Raspberry Pi を使った検証環境構築...40
4.2 シミュレーションでの未検証項目...42
4.3 定周期割り込みと周波数ゲートの検証 42
4.3.1 定周期割り込み...42
4.3.2 周波数ゲート信号の発生...42
4.4 I2C 通信の検証...43
4.4.1 書き込み動作...43

4.4.2 読み取り動作（その 1） 44
4.4.3 読み取り動作（その 2） 44
4.5 時間計測機能の検証 45
4.5.1 パルス幅測定機能...........................45
4.5.2 周波数測定機能 46
4.5.3 周波数測定用ゲート時間................47
4.6 汎用測定機能の検証 47
4.6.1 イベント計数機能...........................47
4.6.2 電圧測定機能 48
4.7 MPU クロック周波数の影響...............49
4.7.1 定周期割り込みの負荷 49
4.7.2 伝送エラーの原因解明 50
4.7.3 最大負荷のイベント計数機能..........51
4.8 PICCOLO チップとしての検証...........52
6. I2C 通信の概要 53
6.1 3 層モデル 53
6.2 物理層 53
6.3 データリンク層 54
6.4 応用層 54
6.5 ユーザー機能 55
6.6 PIC の I2C サポート 55
仕様書 汎用測定チップ PICCOLO 1
概要 1
特長 1
広い動作範囲 1
多様な入力 1
I2C 通信。 1
外形.. 1
端子配列.. 1
電気的特性.. 1
測定機能.. 2
パルス幅測定機能 2
パルス周波数測定機能............................... 2
汎用入力測定機能 2
内部レジスタ 3
レジスタ一覧 3
Status レジスタ 3
Command レジスタ 4
データレジスタ.. 5
あとがきと付録.. 1
あとがき.. 1
付録 1 自律走行車プロジェクト 2
付録 2 プロジェクトファイル......................... 2

利用条件と免責事項

この本の著者は、この本のすべての内容（引用を除く）が、著者オリジナルの著作物であることを主張します。掲載されたプログラムコードを除き、本書の内容を勝手に改変することを一切禁止します。

Raspberry Pi を使った応用を志す人は、この本の内容を自由に活用し、また同じ意図を持つ人に紹介あるいは再配布することができます。

この本に掲載、あるいは添付されたプログラムを、自作や研究目的で、利用したり改造したりすることは自由です。しかし商用目的に流用するには、著者の許諾が必要です。

この本の内容あるいは掲載プログラムを利用したことによって起こった、どのような損害に対しても、著者は責任を持ちません。また著者の過誤や誤謬にもとづく記載があった場合も同様です。

2021 年 6 月 著者

1　Raspberry Pi 派生プロジェクト　汎用測定チップ

1.1　開発の背景

いま、Raspberry Pi を使った自律走行車プロジェクト（付録に挙げた Web ページを参照）を進めています。その中で必要になった、Raspberry Pi だけでは実現できない測定機能を搭載したチップの開発過程を、派生プロジェクトとして記録します。

マイクロコントローラ PIC にファームウエアを組み込むことで実現することにしました。私にとって初めて触れる PIC は、コンピュータというより、所定の機能を実現するための素材です。だから、あまり技巧的なプログラミングはしないで、愚直に機能を記述していくことにします。

この本の中で使う用語を説明しておきます。PIC では命令を実行する部分は『CPU』（実際に使うのは Enhanced Mid-Range CPU）と呼ばれていますが、Raspberry Pi などと区別するため『MPU (micro processing unit)』と記述します。また、PIC で実行するプログラム（あまり柔軟性のあるものではなく、一度書き込んだら変更することは少ない）は『ファームウエア』と呼んで、Raspberry Pi などの『ソフトウエア』と区別することにします。

PIC の周辺ハードウエアの制御や状態を知るのに使う『特殊機能レジスタ』を SFR と略称します。これとは別に、I2C 通信を介して読み書きできるデータを（ファームウエア）レジスタと呼ぶことにします。これには、チップを制御するコマンドレジスタ、チップの状態を表示するステータスレジスタ、測定値を表示する測定値レジスタがあります。

1.2　仕様を決める

1.2.1 要求仕様の分析

要求元である自律走行車プロジェクトで必要になった測定機能は次の二つです。

1. 超音波が障害物まで往復する時間（10cm～1m = 0.6～6ms のパルス幅信号）を測定する。分解能は 1cm（60μs）程度あれば充分
2. 左右動輪の回転センサ出力パルス（2 チャンネル。最大でも 30 パルス/秒程度）を計数する。1m を移動するときのパルス数は 100 程度なので、1000 カウントできれば充分

これを PIC で実現できる機能に翻訳すると、それぞれ次のようになります。

1. 入力信号をゲートにして、クロックパルスを計数するカウンタ。クロック周波数は 100kHz（10μs/パルス）以上あればよいが、実装上は PIC チップの内蔵クロック 500kHz または 125kHz を使うのが望ましい。計数値が 10 万を超えることはないので、カウンタ長は 16 ビットで足りる。
2. パルス周波数が 30Hz 程度と低いので、ハードウエアカウンタでなくても対応できる。100～500μs 程度の周期でポートを読み取るファームウエア処理で充分。カウンタ長は 8 ビットでは不足するので 16 ビットにする。

測定値を読み出すために I2C 通信（スレーブ）機能が必要なので SDA と SCL を加え、I/O ピン（信号数）は少なくとも 5 本が必要です。これに V_{DD} と GND を加えれば、8 ピンの IC でも良さそうです。

しかし、プログラム開発（書き込みとデバグ）用として、さらに 3 本のピンが必要です。I/O と兼用にすることもできますが、デバグの手間が大きいので、専用に割り当てることにしました。必要なピン数は 10 本なので、14 ピンのパッケージが適当です。増えたピンは、機能増強に使います。

1.2.2 ベースにする PIC チップ

PIC は多くの種類が販売されています。Microchip 社が、古い PIC も廃品種にしないという方針を取っているためです。このチップを商品に使うのなら、信頼性と価格（一般的にメモリ容量が大きく、機能数が多いほど高価）を吟味して選ぶところです。しかし PIC の市場価格は、あまりメモリや I/O 数に依存していません。むしろ流通量の影響が大きいようです。趣味のプロジェクトなので、目的に最適なものより、使い勝手のいいものを選びます。今後の別プロジェクトで使うことも考え、チップを標準化（常に同じチップを使う）しておきます。以下の基準でベースチップを選ぶことにしました。

- 14 ピン DIP パッケージで必要な I/O 数がある
- 動作に必要な外付け回路がない、または少ない
- 基板に取り付けた状態でデバグできる
- 3.3V の電源で動作する

- MPU は拡張（Enhanced mid-range）8 ビット
- I2C 通信用のハードウエアがある
- 16 ビット長のカウンタがある
- プログラムメモリは 4kW 以上ある
- データメモリは 1kB 以上ある
- 入手が容易（秋月電子通商で在庫している）

この基準で絞り込んだ中から、いちばん安価だった 15386 ファミリーの PIC16F15325-I/P をエィッヤァと選びました。おもな仕様は以下のとおりです。

項目	仕様
外形	14P DIP
電源電圧	2.3～5.5V
MPU アーキテクチャ	8 ビット RISC
最大クロック周波数	32MHz
命令セット	14 ビット/命令
プログラムメモリ	8kW
データメモリ	1kB
不揮発メモリ	224B（使わない）
I/O	12 本（プログラム・デバグ用を含む）
I2C 通信	1 チャンネル
カウンタ・タイマ	16 ビット×2 本、8 ビット×1 本
A/D 変換	10 ビット
D/A 変換	5 ビット（使わない）

PIC16F15325 のおもな仕様

1.2.3開発仕様の作成

ベースチップにあわせて、開発仕様の概要を決めます。要求仕様どおりにするのではなく、より汎用性の高いチップにするために拡張しました。

I/O ピンに余裕があるので、入力機能を増やしていきます。イベント（低速パルス）入力を 2 点から 4 点に増やしました。A/D 変換機能があるので、この 4 点はアナログ電圧測定もできるようにします。

時間幅測定は、超音波の送信信号と同期をとるモードと、時間幅信号の最後で測定値を取り込むモードを考えました。もう一つ、別の外部パルス信号の周波数測定もできるようにしました。

以上を整理して、開発仕様の概要版とします。

項目	仕様
時間幅測定機能	ゲート入力がアクティブになっている時間を測定
測定クロック	内部クロック（500kHz/125kHz）
測定動作	同期信号、またはゲート信号の終わりでデータ取込
周波数測定機能	入力パルスの周波数を測定
ゲート時間	1 秒、100ms、10ms から選択
自由カウント	外部入力パルスを計数（開始・停止は通信指定）
汎用入力機能	4 点（イベントあるいはアナログ入力を選択）
イベント計数	4 入力のそれぞれを計数
イベント極性	立ち上がり、または立ち上がり・立ち下がり両方
アナログ入力	0～基準電圧(2.048V)の範囲を測定

開発仕様（測定機能のみ）の概要

使い勝手と汎用性を考え、いっぽう実現方法も考慮に入れながら、次章で開発仕様を詳細化していきます。

1.3　開発環境を整える

PIC は内蔵マイクロコントローラ（MPU）がプログラムを実行することで機能を発揮するので、プログラムの開発環境が重要です。

1.3.1 開発環境と言語

Microchip 社が無償で公開している開発環境を PC にインストールして使うことにします。以下の 1．以外は、統合開発環境 MPLAB X IDE として提供されています。1．のコンパイラ（XC8）は、IDE をインストールした後で追加インストールします。インストール方法は説明しないので、他の資料にあたってください。

1. **C 言語コンパイラ**（C プログラムを処理してリンカーに渡せる形式にする）
2. **アセンブラ**（アセンブリ語プログラムを処理してリンカーに渡せる形式にする）
3. **リンカー**（アセンブラ出力を実行可能形式にする）
4. **シミュレータ**（PC 上でプログラムを実行する）
5. **プログラマ**（PIC にプログラムを書き込む）
6. **デバッガ**（実チップ上でプログラムの実行を制御し、動作を確認する）
7. **プロジェクトマネージャ**（プロジェクトに関わるファイルを管理する）
8. **テキストエディタ**（プログラムを編集する。使い慣れた他のエディタを使ってもよい）

C コンパイラには無償版と有償版があり、有償版は強力な最適化（コードを小さくする、実行を早くするなど）ができると言われています。無償版でも一般的な最適化は可能だし、60 日の試用期間中は有償版の最適化も行えます（使いませんでした）。

PICCOLO の開発には、できるだけ C 言語を使うことにします。どうしても必要になった時はアセンブリ語も併用するつもりで始めましたが、結局使わずに済みました。

1.3.2 PICkit による書き込みとデバグ

In-Circuit Serial Programming（ICSP）搭載のチップを選んだので、プログラムの書き込みやデバグに、チップ上の回路が使えます。多少の制約はありますが、安価な PICkit で書き込みとデバグをすることにします。2020 年 11 月時点では PICkit4 が入手できました。ソフトウエアは MPLAB X IDE が使えます。この操作方法を説明すると大変な量になるので省略します。他の資料を参照してください。

1.3.3 Raspberry Pi を利用した信号源

入力機能の検証と評価は、アナログ電圧やパルス信号を与えて行います。I2C 通信の検証用マスターばかりでなく、さまざまなパルス信号を発生する信号源が必要です。ここでは自律走行車でホストとして使う Raspberry Pi を採用しました。

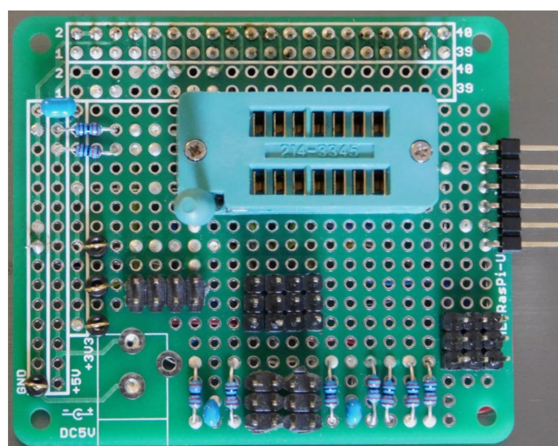
Raspberry Pi には Raspberry Pi OS と pigpio ライブラリをインストールしておきます。include ディレクトリ（作業ディレクトリの下）には、自律走行車プロジェクトの use-pigpio.h、use-time.h をコピーしておきます。

1.3.4 開発・評価ボード

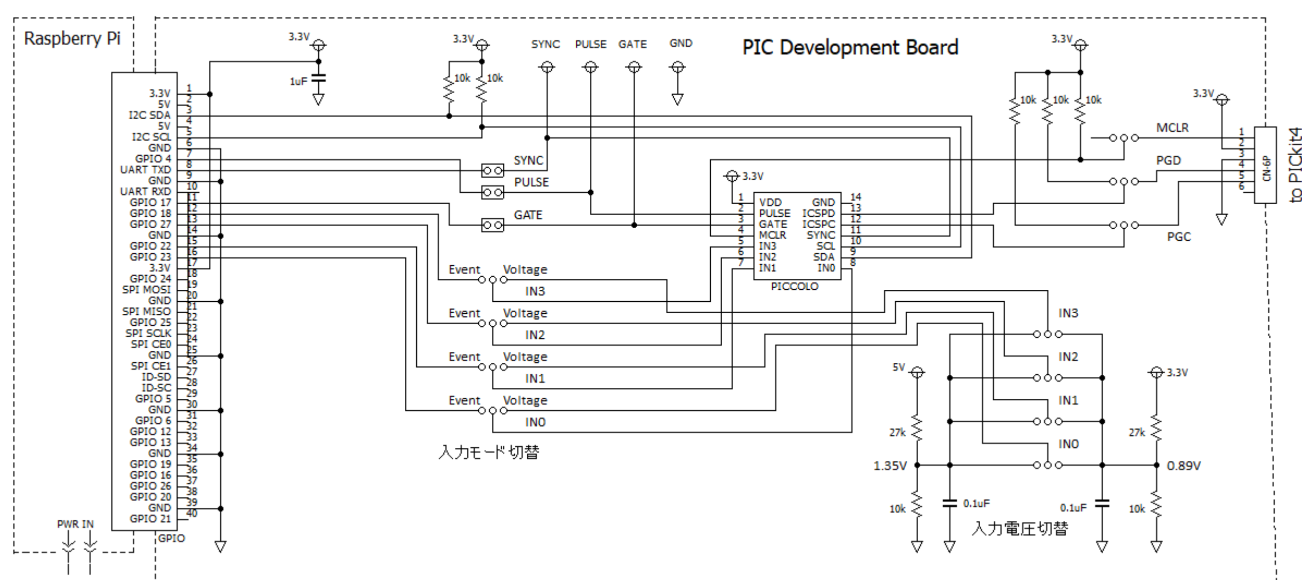
PICCOLO チップの検証と評価には、PICkit の接続、Raspberry Pi との I2C 通信、パルス信号とアナログ電圧の印加ができる必要があります。それほど大規模な回路ではないので、Raspberry Pi 用の拡張基板上に組み上げることにしました。ここで評価を終えておけば、最終使用先である自律走行車のソケットに差し込むだけで使えるようになります。

回路図を下に示します。ジャンパは入力の切り替えを行うために設けました。

電源は Raspberry Pi の 3.3V ラインを使います。PICkit からは電源を供給しません。最終的に、こんな外観になりました。



開発・評価ボードの外観



PICCOLO チップの開発・評価ボード

2　開発仕様と実装設計

2.1　開発仕様

前章で概要を検討した開発仕様を、より厳密なものにしていきます。最終的な開発目標は、この本の最後に「仕様書」としてまとめてあります。

2.1.1 通信レジスタ

PICCOLO チップの機能は全て、I2C 通信でアクセスできるレジスタで表現します。このレジスタはファームウエアで実現するもの（仮想レジスタ）で、ベースチップのハードウエアとして存在するレジスタ（SFR）とは別のものです。以下では、マスターが PICCOLO チップにデータを渡すことを『書き込み』、PICCOLO チップのレジスタからデータを得ることを『読み出し』と表現することにします。

Command レジスタ

PICCOLO チップの機能を設定します（書き込み）。入力の選択、クロックの選択、測定モードなどを設定します。

Status レジスタ

測定状態や（測定値以外の）測定結果を表示します（読み出し）。

測定値レジスタ

測定機能ごとに 16 ビットの測定値を表示します（読み出し）。オーバーフローなどの付加情報は Status レジスタに表示されています。

2.1.2 通信機能

I2C 通信のスレーブ機能を搭載します。アドレスは既知の I2C デバイスと重複しない 0x38 を選びました。

実装を単純にするため、機能の一部を制限します。

- I2C クロックは標準の 100kHz に限定する
- 通信レジスタは書き込み／読み出し専用（コマンドレジスタは読み返せない、ステータスや測定値には書き込めない）
- レジスタ毎に（1 バイトまたは 2 バイト単位で）読み書きする（両コマンドレジスタへの一括書き込み、複数データの一括読み出しはできない）

2.1.3 時間測定機能

GATE 信号のパルス幅を測定します。同期信号 SYNC 後のパルス幅（の合計）を測定するモードと、GATE 信号毎に測定するモードがあります。自律走行車では前者を使います。

2.1.4周波数測定機能

PULSE 信号の周波数を測定します。測定時間は 1 秒、100ms、10ms から選びます。この機能は自律走行車では使用しません。

2.1.5 イベント計数機能

IN0～IN3 信号の変化回数を数えます。自律走行車では 2 点だけ使います（両輪の回転を測定する）。

2.1.6 電圧測定機能

IN0～IN3 信号（イベント計数と共用）の電圧を測定します。測定範囲は 0～2.048V です。この機能は自律走行車では使用しません。

2.2　実装設計 I（測定機能）

測定機能を PIC ベースチップで実現する方法を検討します。

2.2.1 入出力端子のアサイン

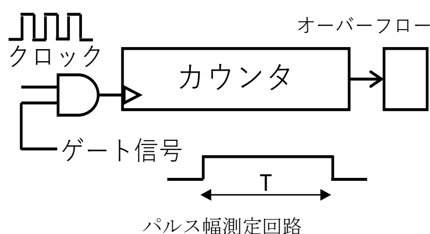
PICCOLO チップの入出力端子（ピン）を次の表のようにアサインします。時間測定機能を RA ポートに、汎用入力を RC ポートに割り振りました。I2C 通信用端子以外は、すべて入力専用です。

ピン番号	PIC ポート名	PICCOLO チップのアサイン
1	V_{DD}	V_{DD}
2	RA5	PULSE（パルス周波数入力）
3	RA4	GATE（時間幅信号）
4	MCLR/RA3	MCLR（システム用）
5	RC5	IN3（汎用入力）
6	RC4	IN2（汎用入力）
7	RC3	IN1（汎用入力）
8	RC2	IN0（汎用入力）
9	RC1	SDA（I2C 通信）
10	RC0	SCL（I2C 通信）
11	RA2	SYNC（時間幅測定の同期信号）
12	RA1/ICSPCLK	ICSPCLK（システム用）
13	RA0/ICSPDAT	ICSPDAT（システム用）
14	V_{SS}	V_{SS}（GND）

PICCOLO チップのピンアサイン

2.2.2 パルス幅測定機能

パルス幅を測定する基本回路は以下のようなものです。ゲート信号が H レベルの間、クロックパルスを計数します。オーバーフローが起こったら、それを記憶しておき、結果に反映します。



パルス幅測定回路

ベースチップには、このゲート機能をもったタイマー#1 があります。オーバーフローは割り込みで捕まえ、T-Status レジスタに反映します。クロックパルスは 500kHz の内蔵クロックをそのまま、あるいは 4 分周したもの（125kHz）を使います。

同期モードでは SYNC 信号で、非同期モードでは GATE 信号の終わりで割り込みをかけ、データを測定値レジスタに転送するとともに、タイマー#1 をクリアします。この間は、ゲート信号が閉じている（L レベルにある）必要があります。

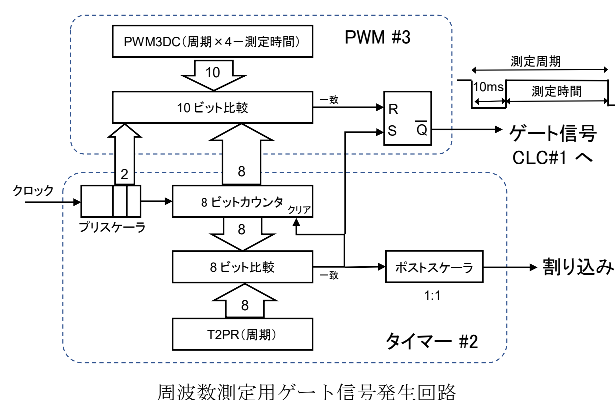
PIC 応用例の多くは、外部信号でタイマー#1 の計数値をいったんキャプチャレジスタ（CCPR）に取り込むようにしています。これは計数中にタイマー#1 を読み出そうとしたとき、2 バイトのデータを読み出す間にカウンタが進んでしまうと、16 ビットデータの整合がとれなくなる（例えば計数値が 0x00ff のとき、下位バイトを読み出した直後に計数が進むと、上位バイトが 0x01 に変わり、16 ビットデータが 0x01ff になってしまう）ことを防ぐための手段です。ベースチップには、これを避けて 16 ビットデータを読み取る手順が用意されているので、単純な「ゲート付きカウンタ」を採用しました。手順については次章の「**コーディング上の注意事項**」の節で説明します。

2.2.3 周波数測定機能

周波数測定に使う回路は、パルス幅測定機能と同じです。クロックにはパルス入力を、ゲート信号には内部で一定のパルス幅信号を作って与えてやります。ベースチップには、タイマー#1 とよく似た機能のタイマー#0 がありますが、こちらにはゲート機能がないので、追加してやることにします。

ゲートを外付けするのでは汎用チップらしくないので、ベースチップが内蔵している構造可変型論理回路 CLC（Configurable Logic Cell）を利用することにします。入力として PULSE 入力（RA5 ポート）と PWM で発生させたゲート信号を選び、その AND を出力させます。タイマー#0 は CLC#1 の出力をクロックとして使うことができます。これでパルス幅測定回路と同じものがつくれました。

ゲート信号はタイマー#2 と PWM 発生回路#3（PWM#3）で作ります。ベースチップには 6 個の PWM 発生回路があります。そのうち PWM#1～#2 はタイマー#1 の計数値を使い、PWM#3～PWM# 6 はタイマー#2 の計数値を使います。 次の図を見てください。



周波数測定用ゲート信号発生回路

タイマー#2 で測定周期を発生させます。クロックをプリスケーラで分周したものを 8 ビットカウンタで計数し、T2PR レジスタの設定値に達したら（一致したら）、カウンタをクリアすることで周期を設定します。このタイミングで MPU に割り込みをかけ、タイマー#0 の計数値を測定値レジスタに転送してから、タイマー#0 をクリアします。

PWM#3 は、割り込み発生後、カウンタ計数値の下にプリスケーラの上位 2 ビットを加えた 10 ビットデータが、PWM3DC に設定した値と一致するまでの時間を発生させます。この時間（約 10ms）を測定周期から引いたものがゲート信号になります。ちょっと奇妙な方法ですが、割り込み後のデータ読み込み時間（ゲート信号が L になっている時間）を確保してから、次の測定を始めるための工夫です。

ゲート時間ごとに、次の表のように設定します。設定項目のうちカッコで囲っているのは、他の設定の結果であることを示しています。

設定項目	1 秒	100ms	10ms
クロック周波数	31.25kHz	31.25kHz	500kHz
プリスケーラ	1/128	1/16	1/64
(1 カウント)	4.096ms	512μs	128μs
T2PR 設定値	247	215	156
(測定周期)	1011.7ms	110.08ms	19.968ms
(PWM 設定単位)	1.024ms	128μs	32μs
PWM3DC 設定値	11	79	311
(PWM 時間)	11.26ms	10.11ms	9.952ms
(ゲート時間)	1000.4ms	99.97ms	10.016ms

ゲート信号を発生するための設定

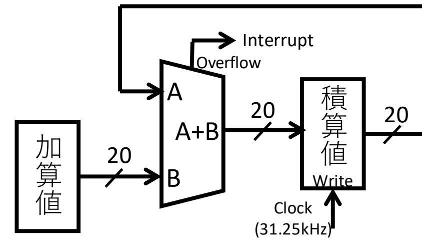
内部発振器の精度は±1%なので、約 1/1000 の精度でゲート時間が決められれば十分です。

2.2.4 イベント計数機能

イベントは低速の事象なので、ファームウエア処理で実現します。ベースチップには、入力ポートの変化で割り込みを発生する機能があるので、これを使えば良さそうに見えます。しかしイベント発生源のハードウエアによっては、イベント発生時に出力が短時間オン・オフを繰り返すことがあり、計数誤差のもとになります。タクトスイッチの場合では、この繰り返し（チャタリングといいます）が 100μs～10ms 程度続きます。自律走行車の光インタラプタは細いスリットを使っているので、回転円盤の縁の影響は受けにくいとは思いますが、念のため 100μs 以内の現象は除去したいと思います。

そこで 100～300μs 程度の定周期割り込みを使い、これより短い過渡現象を無視するようにします。

定周期割り込みの発生にはタイマーを使うのが普通ですが、ここまでにベースチップが持っている 3 つのタイマー#0～#2 を使いつくしてしまいました。ちょっと大仰なのですが、数値制御発信器（Numerically Controlled Oscillator: NCO）というブロックを使います。これは 20 ビットのレジスタと加算器からなり、1 クロックごとにデータを逐次的に加算（加算値が 1 ならカウンタになる）していき、桁上がり（オーバーフロー）で割り込みを発生させます。クロックとして 31.25kHz（32μs）を使い、0x40000 を逐次加算 すると 4 回（128μs）、0x20000 を逐次加算すると 8 回（256μs）で割り込みを発生させられます。



NCO を使った定周期割り込み

割り込みごとにポート入力を読み取り、前回からの変化を調べます。変化があったときは、メモリ上に設けた測定値レジスタを 1 だけ増やします。

2.2.5 電圧測定機能

ベースチップに内蔵されている 10 ビット A/D 変換器を使って電圧を測定します。データシートの説明文からみると、サンプル・ホールド回路と逐次近似方式を使っているようです。測定信号を選んだら 2μs（サンプル時間）程度待ち、1 ビットあたり 1～6μs（設定による）で変換が行われます（この 11.5 倍が逐次近似にかかる）。仮にビット変換時間を 2μs とすると、変換に要する時間は 25μs です。あまり急いで実行する必要はないので、バックグランド処理の中で変換終了を待つことにします。

2.2.6 I2C 通信

I2C 通信プロトコル（通信規約）については、最後の章を参照してください。このうち、物理層とデータリンク層（の一部）を実現するのが Master Synchronous Serial Port（MSSP1）です。

スレーブモードでは、自分のアドレス向けの通信を受信したら、割り込みを起こします。その後 1 バイトのデータを受信あるいは送信の終了まではハードウエアが面倒を見てくれるので、次の割り込みまでに、MPU で必要な処理を行います。

応用層の仕様は限定しています。データを受信した（Command レジスタ宛に限る）ときは、内容を解釈します。2 バイトの読み出しレジスタを指定されたときは、現在値を送信バッファにコピーしてから、順次送信します。こうすることで、送信中に測定データが更新されても、送信データに矛盾が起こる心配がありません。

具体的なファームウエア処理は後の節で説明します。

2.3　実装設計 II（PIC MPU の使い方）

測定機能以外のハードウエア設定と、MPU の使い方を検討します。

2.3.1 クロック周波数

PIC はクロック源を複数持ち、動作クロックを動的に切り替えることができます。これは、組込み機器などで、仕事がないときの消費電力を落とすのに便利です。しかし PICCOLO チップでは、常に測定を行うのが前提なので、クロックは一定にします。

消費電力を小さく保つため、基本クロック FOSC は内部で発生させた 8MHz を使うことにします（1～32MHz から選択）。PIC の MPU はこの 4 周期を 1 サイクルとして、1 命令の処理に 1 サイクル（分岐命令などは 2 サイクル）かかるので、処理能力は 2MIPs（2×10^6 命令/秒）弱ということになります。あとで処理能力が不足することが分かったら、クロックを早くすることを考えましょう。

2.3.2 Watch Dog Timer

Watch Dog Timer は、プログラムの暴走を監視し、期待した処理ができていないときにリセットをかけるものです。PICCOLO チップは（ファームウエアの検証が十分なら）あまり暴走は起こりそうにありません。万一起こったら、ホスト（Raspberry Pi）側で全体をリブートすることにして、Watch Dog Timer は使わないことにします。

2.3.3 割り込み処理

測定と通信の処理要求には、それぞれのタイミングがあるので、割り込み処理で対応します。

ベースチップは、割り込みが発生すると、5 つの重要なレジスタを、シャドーレジスタという領域に退避してくれます。退避と回復がそれぞれ 1 サイクルで行われるので、非常に便利です。多重割り込みが発生しないので、割り込み処理中に複数バイトのデータを扱っている間に、別の割り込みにデータを書き換えられる心配もありません。しかし、シャドーレジスタは一組しかありません。そのため、割り込みレベルも一つしかなく、割り込み処理ルーチン（Interrupt Service Routine：ISR）の中で、要因を調べて処理しなければなりません。

PICCOLO チップで使用する割り込み要因を次の表にまとめます。目標処理時間（要因が発生してから処理を終えるまで）はおおよその目安で、それ以内に必ず処理できなければいけないというわけではありません。通信の場合は相手を待たせればいいので。それでも、目標処理時間が短いものは優先して処理します。

優先度	割り込み要因	目標処理時間
1	I2C 通信（アドレス受信）	10μs
1	I2C 通信（データ受信）	40μs
1	I2C 通信（データ送信完了）	10μs
1	I2C 通信（通信終了）	40μs
2	SYNC 信号（タイマー#1）	20μs
2	GATE 信号（タイマー#1）	20μs
2	周波数ゲート（タイマー#0）	5ms
3	イベントのチェック（NCO）	50μs
4	タイマー#1 オーバーフロー	50μs
4	タイマー#0 オーバーフロー	50μs

割り込み要因と目標処理時間

ISR の先頭で、優先度の高いものから処理要求を調べていきます。最初の要求を処理したら、いったん ISR を終了して、再割り込みを可能にします。こうすることで、優先度の低い処理をしている間に、優先度の高い処理要求が発生すれば、終了後ただちに処理できるようにできます。そのためには、優先度の低い処理でも、短時間で済ませる必要があります。

割り込み処理の始めでは、優先度順に要因をチェックして、最初に見つかった要因の処理ルーチンへ分岐しますが、その前に要因をクリアしておきます。これは、処理中に同じ割り込みが発生したときのためです。

2.3.4 バックグランド処理

割り込み処理以外の時間には、最も優先度の低い処理を行います。ふつうは自己診断などを組み込むのですが、PICCOLO チップでは省略します。

その代わり、処理時間目標のない、あるいは緩い、二つの処理を行います。途中で割り込まれては困る処理を行っている間は、割り込みを禁止するようにします。

コマンド解釈

Command レジスタに書き込まれたデータに従って、ハードウエアの設定を変更します。できれば ISR の中で処理するのが望ましいのですが、割り込みを禁止する時間を短くするために、込み入った設定はバックグランドで処理します。

電圧測定

入力を切り替えて A/D 変換器を起動し、測定が終わったら測定値レジスタに書き込みます。書き込み中は割り込みを禁止します。

変換終了を待つ間は、コマンドレジスタの変更がないか調べて、変更があったら解釈処理を先に実行します。変更がなければ、変換終了待ちを繰り返します。

2.3.5 メモリマップ

ベースチップには 1kB のデータ用メモリがあります。16 F シリーズの PIC は命令語長の制約で、メモリアドレスは 7 ビットでしか指定できません。そのため、メモリは 128 バイトごとのバンクに分割されていて、バンク番号は別の SFR を通して指定します。13 個のバンクには 80 バイトずつのメモリ（13 番目のみ 48 バイト）と、全バンクからアクセスできる 16 バイトの共通メモリがあります。

128 バイトより広い連続領域を作るため、上の 80 バイトをつなげて別の連続アドレス（0x2000～0x23EF）にマップすることができます。この領域へのアクセスは、16 ビットの SFR にアドレスを書き込むことで、間接的に行います。

PICCOLO チップでは、メモリのどこに、どのデータを置くか、コンパイラに決めさせることにします。

2.3.6 命令アドレス

8kW の命令領域にもバンク構造があります。16 F シリーズの PIC は命令語長の制約で、プログラムの分岐先は 11 ビット（2kW）で指定するので、この境界を越える分岐は面倒です。ベースチップには 4 つのバンクがあります。

PICCOLO チップでは、すべてのファームウエアがバンク#0 に収まると仮定して、コンパイル/リンクを行うことにします。ファームウエアサイズが 2kW を越えるようなら、時間的な制約のない初期化処理などを別バンクに置くことを検討します。

2.3.7 スタック

サブルーチンコールのときには、戻り番地をスタックに保存します。ベースチップには、これ専用の領域（メモリとは別）が 16 段分あります。多くの CPU のように、スタックに動的変数を作るということはできません。PICCOLO ファームウエアでのサブルーチンコールの深さは、10 段以内に収まると予想しています。

2.4 実装設計 III（ファームウエア）

ファームウエア処理の実装設計を行います。多くの機能は次の 3 つの処理で実現されます。

1. ハードウエア（SFR）と内部データの初期化
2. コマンドの解釈と SFR の設定変更
3. 割り込みあるいはバックグランド処理

初期化と設定変更では、ベースチップのハードウエアを制御する特別機能レジスタ（SFR）にデータを設定します。これから、この設定データを決めていきます。以下では、SFR のレジスタ名とビット名はベースチップのものを使いますが、それぞれの詳細は説明を省略します。ベースチップのデータシートを参照してください。

なお、SFR の特定ビットを示すときには、構造体のように「SFR 名.ビット名」と記述しています。C 言語で記述するときには別の方法を使うので、間違わないようにしてください。

以降は、SFR の名前とビットフィールド、C プログラムの変数名と関数名は `OSCCON1` などと固定幅フォントで、デバッガ MPLAB X IDE の機能は *Stimulus* とイタリック（斜体）フォントで表します。

2.4.1PIC 基本部

まず、PIC を使う上での基本的な設定を決めます。

コンフィギュレーション

コンフィギュレーションとは、ファームウエアからはアクセスできない領域にある特別機能レジスタに設定するデータのことで、最初にプログラマで書き込みます。

ベースチップの仕様書に従って、設定を決めていきます。安定した環境下で専用チップとして使うので、あまり複雑な監視や異常処理は必要ありません。

コンフィグレーション項目	設定内容
外部クロック監視	監視しない
クロック周波数変更	ファームウエアで変更が可能
クロック出力	出力しない
電源投入時のクロック	32MHz
デバッガの使用	使用する
スタック異常リセット	リセットしない
信号内部配線網（PPS）	使用する
ゼロクロス検出	使用しない

ブラウンアウト	使用しない
電源投入タイマー	使用しない
ウオッチドッグタイマー	使用しない
低電圧プログラム	使用する
書き込み禁止	すべての領域で禁止しない
ブートブロック	使用しない

PICCOLO チップのコンフィギュレーション

一般的な初期化

一般的な初期化とは、PICCOLO チップの機能とは直接関係しないものの、PIC を使う上で必要な設定のことです。

制御レジスタ名	設定値	設定内容
OSCCON1	0110 0000	クロック HFINTOSC/1 分周
OSCCON3	000x x000	クロック即時変更
OSCEN	0110 0000	500kHz/31.25kHz を使用する
OSCFRQ	0000 0011	HFINTOSC = 8MHz
INTCON	0100 0001	測定回路からの割り込み許可
PMD0	0000 0111	基準電圧以外の回路は不要
PMD2	0100 0111	ADC 以外の回路は不要
PMD3	0011 1011	PWM3 以外の回路は不要
PMD4	1100 0001	MSSP(I2C)以外の回路は不要
PMD5	0001 1100	CLC1 以外の回路は不要

一般的な（測定機能以外の）設定

2.4.2パルス幅測定機能

パルス幅測定機能はほとんどがハードウエア処理です。

初期化

タイマー#1 周辺の SFR を設定します。そのあとで、内部変数である T-Command レジスタと T-Status レジスタを（周波数測定機能の分も）初期化します。

まず、周波数測定機能でも使う I/O ポート A の初期設定をします。ポート A の用途は以下のとおりで、すべて入力です。

ポート番号	信号名	用途
RA0	ICSPDAT	デバグ用
RA1	ICSPCLK	デバグ用
RA2	SYNC	時間幅測定の同期信号
RA3	MCLR	デバグ/リセット用
RA4	GATE	時間幅信号
RA5	PULSE	周波数測定用パルス信号

ポート A の用途

ポート A を上の目的に合うように設定します。

レジスタ名	設定値	設定内容
PORTA	0000 0000	ポート出力（使わない）
TRISA	0011 0111	ポートは全て入力
LATA	0000 0000	書き込み不要（PORTA に書く）
ANSELA	0000 0000	すべてデジタル入力
WPUA	0011 0100	入力信号はプルアップ
ODCONA	0000 0000	出力はプッシュプル(使わない)
SLRCONA	0000 0000	スルーレートは最高速
INLVLA	0011 1111	入力は MOS レベル

ポート A の初期設定

次にタイマー#1 に関連する SFR を初期化します。初期設定値は、T-Command レジスタの初期値を反映したものにします（以降で説明する、他の測定機能についても同様）。

レジスタ名	設定値	設定内容
T1CON	0010 0110	1/4 分周、16 ビットアクセス
T1GCON	1100 0000	測定ゲートを使う
T1CLK	0000 0101	クロックは 500kHz
T1GATE	0000 0000	ゲート信号は PPS から貰う
T1GPPS	0000 0100	GATE(RA4)をゲート信号に
INTPPS	0000 0010	SYNC(RA2)を割り込み源に

パルス幅測定機能の初期化

コマンド解釈

新しいコマンドを現在の T-Command と比較し、変化したビットによって、SFR の設定を変更します。

<table>
<tr><th>設定項目</th><th colspan="4">指令値と設定値</th></tr>
<tr><td>測定クロック</td><td colspan="2">WC == 0</td><td colspan="2">WC == 1</td></tr>
<tr><td>T1CON.CKPS</td><td colspan="2">10 (8μs)</td><td colspan="2">00 (2μs)</td></tr>
<tr><td>ゲートの極性</td><td colspan="2">WP == 0</td><td colspan="2">WP == 1</td></tr>
<tr><td>T1GCON.GPOL</td><td colspan="2">1 (High)</td><td colspan="2">0 (Low)</td></tr>
<tr><td rowspan="2">動作と割り込み
（PIE は 1 が割込）</td><td colspan="2">WE == 0</td><td colspan="2">WE == 1</td></tr>
<tr><td>WM == 0</td><td>WM == 1</td><td>WM == 0</td><td>WM == 1</td></tr>
<tr><td>T1CON.ON</td><td colspan="2">0 （動作停止）</td><td colspan="2">1 （動作）</td></tr>
<tr><td>PIE0.INTE</td><td>0</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>PIE5.TMR1GIE</td><td>0</td><td>0</td><td>0</td><td>1</td></tr>
<tr><td>PIE4.TMR1IE</td><td>0</td><td>0</td><td>1</td><td>1</td></tr>
</table>

パルス幅測定機能の SFR 設定

WE に 1 が書き込まれたときの処理：

- PIR0.INTF、PIR5.TMR1GIF、PIR4.TMR1IF をクリアする
- TMR1 を 0 に初期化する
- Width 測定値レジスタを 0 に初期化する

- T-Status レジスタの RDT ビットと OVT ビットをクリアする
- メモリ上に用意した一回目フラグをセット、オーバーフローフラグをクリアする

  （一回目フラグは、タイミングによっては 1 回目の測定値が半端な値になるので、データを廃棄するために使います。他の測定機能でも、それぞれフラグを用意する）
- 割り込みの設定（PIE0.INTE または PIE5.TMR1GIE、PIE4.TMR1F）は、他の設定がすべて終わってから行う

WE に 0 が書き込まれた時の処理：

- PIE0.INTE または PIE5.TMR1GIE、PIE4.TMR1F をクリアして、割り込みを禁止する
- T1CON.ON をクリアしてカウンタの動作を止める
- T-Status レジスタと Width 測定値レジスタは、以前の値を保持する

割り込み処理

割り込み要因は 3 つありますが、オーバーフロー以外の処理は同じです。

SYNC 割り込み、GATE 割り込みの処理：

- PIR0.INTF または PIR5.TMR1GIF をクリアする
- 一回目フラグがセットされているときは、一回目フラグをクリアして終了する
- TMR1 のデータを Width 測定値レジスタに転送する
- T-Status レジスタの RDT ビットをセットする
- オーバーフローフラグを T-Status レジスタの OVT ビットにコピーする
- TMR1 を 0 に初期化する
- オーバーフローフラグをクリアする

オーバーフロー割り込みの処理：

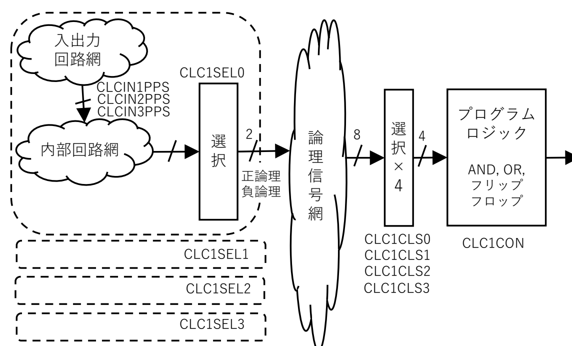
- PIR4.TMR1IF をクリアする
- オーバーフローフラグをセットする

2.4.3周波数測定機能

周波数測定機能もほとんどがハードウエア処理ですが、多くの回路ブロックを使うので、設定が複雑です。

CLC の回路設計

最初に CLC（Configurable Logic Cell）の使い方を決めておきます。CLC（4 つあるうちの 1 番目 CLC1）の回路はおおよそ次のようなものです。CLC1…は、機能や入力を選択する SFR です



CLC1 回路の構成

4 つある選択ブロック（点線で囲ってある）では、内部回路と入力ポートからの信号を選び、正論理と負論理の信号を論理信号網に送ります。次段の選択回路（4 回路ある）では、この 8 種類の入力から AND-OR 回路で選び、プログラムロジックで演算（今回の用途では AND）した結果を出力します。

入出力回路網の選択回路（CLCIN0～CLCIN3）は 4 つあるのですが、データシートによると、なぜか内部回路網は CLCIN1～CLCIN3 しか選べません（誤記かもしれません）。CLCIN1 を選ぶことにします。

初期化

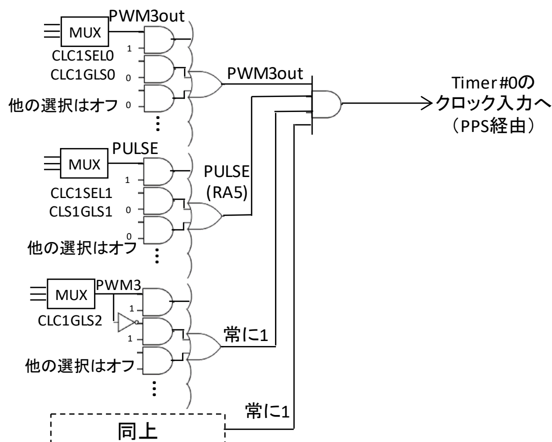
4 つの回路ブロックを初期設定します。T-Command レジスタと T-Status レジスタはパルス幅測定機能が初期化してくれています。

レジスタ名	設定値	設定内容
タイマー#2 設定		
T2CLKCON	0000 0110	クロック 31.25kHz
T2CON	1111 0000	プリスケーラ 1/128 分周
T2HLT	0000 0000	フリーランニングモード
T2RST	0000 0000	外部リセット（設定不要）
T2PR	1111 0111	247 クロックで繰り返し
PWM#3 設定		
PWM3CON	1001 0000	PWM 出力は L レベル
PWM3DCH	0000 0010	10ms 待ち時間設定
PWM3DCL	1100 0000	
CLC#1 設定		
CLC1CON	1000 0010	4 入力 AND
CLC1POL	0000 0000	入力反転なし
CLC1SEL0	0001 0001	入力#0＝PWM3 出力

CLC1SEL1	0000 0001	入力#1＝CLCIN1PPS
CLCIN1PPS	0000 0101	CLCIN1 = RA5(PULSE)
CLC1GLS0	0000 0010	AND 入力#1 = PWM3
CLC1GLS1	0000 1000	AND 入力#2 = PULSE
CLC1GLS2	0000 0011	AND 入力#3 = 常に 1
CLC1GLS3	0000 0011	AND 入力#4 = 常に 1
タイマー#0 設定		
T0CON0	0001 0000	16 ビットカウンタモード
T0CON1	0001 0000	クロック入力＝PPS
T0CKI1PPS	0000 0101	フリー入力 = RA5(PULSE)

周波数測定機能の初期化

CLC 設定のうち、CLC1GLS2 と CLS1GLS3 の設定は分かりにくいかもしれません。4 入力 AND 回路のうち、使わない 2 入力（#3 と#4）は常に 1 である必要があります。同じ入力（ここでは PWM3）と、その反転を入力して OR を取ると、必ず 1 になることを利用しています。



CLC#1 の初期化構成結果

これで、タイマー#2 と PWM#3 で作った測定時間幅パルス PWM3out（負論理）と PULSE 入力の AND がタイマー#0 のクロックになっています。

初期化状態でタイマー#0 は停止させています。T0CKIPPS 信号には PULSE ポートの信号を与えておき、フリーカウント時には T0CON1.T0CS を切り替えるだけで済むようにしました。

コマンド解釈

新しいコマンドを現在の T-Command と比較し、変化したビットによって、SFR の設定を変更します。設定が終了したら、新しいコマンドを T-Command に書き込みます。

設定項目	指令値と設定値			
パルス計数の極性	PP == 0		PP == 1	
CLC1POL.LC1POL	0 （立ち上がり）		1 （立ち下がり）	
動作	PE == 0		PE == 1	
T0CON0.T0EN	0		1	
PIE4.TMR2IE	0		1	
PIE0.TMR0IE	0		1	
PM1:PM0	00	01	10	11
測定時間	フリー	1s	100ms	10ms
T2CLKCON.CS	0101	0110	0110	0101
T2CON.CKPS	110	111	100	110
T2PR	0x9C	0xf7	0xD7	0x9C
PWM3DCH	0x02	0x02	0x13	0x4d
PWM3DCL	0xc0	0xc0	0xc0	0xc0
T0CON1.T0CS	000	111	111	111

周波数測定機能の SFR 設定

PE に 1 が書き込まれた時の処理：

- PIR0.TMR0IF、PIR4.TMR2IF をクリアする
- TMR0 を 0 に初期化する
- Count 測定値レジスタを 0 に初期化する
- T-Status レジスタの RDP ビットと OVP ビットをクリアする
- メモリ上に用意した一回目フラグをセット、オーバーフローフラグをクリアする（両方ともパルス幅測定機能とは別に用意する）
- 割り込みの設定（PIE0.TMR0IE、PIE4.TMR2IE）は、他の設定がすべて終わってから行う

PE に 0 が書き込まれた時の処理：

- PIE0.TMR0IE と PIE4.TMR2IE をクリアして割り込みを禁止する
- T0CON0.T0EN をクリアしてカウンタの動作を止める
- T-Status レジスタの RDP ビットと OVP ビットをクリアする
- T-Status レジスタと Count 測定値レジスタは、以前の値を保持する

割り込み処理

タイマー#2 割り込みの処理：

- PIR4.TMR2IF をクリアする
- 一回目フラグがセットされているときは、一回目フラグをクリアして終了する

- TMR0 のデータを Count 測定値レジスタに転送する
- T-Status レジスタの RDP ビットをセットする
- オーバーフローフラグを T-Status レジスタの OVP ビットにコピーする
- フリーランニングモード以外のときには、TMR0 を 0 に初期化し、オーバーフローフラグをクリアする

オーバーフロー割り込みの処理：

- PIR0.TMR0IF をクリアする
- オーバーフローフラグをセットする

2.4.4 イベント計数／電圧測定機能

イベント計数機能は、数値制御発信器（NCO）からの一定周期割り込みで処理を行います。電圧測定機能はバックグランド処理として実行します。

初期化

初期化では、すべての入力チャンネルはイベント計数測定モードとし、測定は実行しないようにしておきます。NCO はずっと初期設定のままで使い続けます。I/O ポート C の用途は以下のとおりです。

ポート番号	信号名	用途
RC0	SCL	I2C 通信
RC1	SDA	
RC2	IN0	汎用入出力
RC3	IN1	
RC4	IN2	汎用入出力 （デバグ時は信号出力にも使う）
RC5	IN3	

ポート C の用途

下記のように SFR を設定したら、A-Status レジスタ（どの入力も測定値・オーバーフローなし）などの内部変数を初期化します。

レジスタ名	設定値	設定内容
ポート C 設定		
PORTC	0000 0000	仮出力データ
TRISC	1111 1111	ポート C は入力
ANSELC	0000 0000	デジタル入力
WPUC	0011 1100	デジタル入力をプルアップする
ODCONC	0000 0011	I2C のみオープンドレイン出力
SLRCONC	1111 1100	I2C のみ最大スルーレート
INLVLC	1111 1111	CMOS レベル入力
NCO 設定		
NCO1CON	1000 0001	パルス出力で使う
NCO1CLK	0000 0100	31.25kHz クロック
NCO1ACCL	0000 0000	逐次加算値=0x20000 （250μs 周期割り込み）
NCO1ACCH	0000 0000	
NCO1ACCU	0000 0010	
PIE7	0001 0000	割り込み許可
FVR（固定基準電源）の設定		
FVRCON	1000 0010	ADC 用 2.048V を使用
A/D 変換器の設定		
ADCCON0	1110 1101	GND 入力で A/D 変換可能
ADCCON1	1101 0011	2μs/ビット、2.048V 基準
ADACT	0000 0000	自動トリガ禁止（設定不要）

イベント計数/電圧測定機能の初期化

コマンド解釈

新しいコマンドを現在の A-Command と比較し、変化したビットによって、SFR の設定を変更します。設定とデータの初期化が終了したら、新しいコマンドを A-Command に書き込みます。

IN0～IN3 の設定を、SFR のビット 2～5 に反映します（次の表ではビット位置を n、チャンネル番号を m（n = m+2）で表します）。

レジスタ名.ビット名	コマンドの AEm:ENm ビット			
	00	01	10	11
ANSELC.ANSCn	0	1	0	0
WPUC.WPUCn	1	0	1	1

入力ポートの設定

新しいコマンドの入力チャンネル m がイベント計数モード（AEm = 1）、あるいは電圧測定モード（AEm:ENm = 01）のときは、測定値を初期化（Data(m) = 0, OVm:RDm = 00）します。

入力ごとに「一回目フラグ」を持ち、AEm ビットが 1 に変更された時（イベント入力）、このフラグを立てます（電圧測定モードは使用しない）。

入力チャンネル m が測定をしない（AEm:ENm = 00）ときは、デジタル入力モードにしますが、測定は行いません。それまでの測定値は保持し続けます。

イベント計数（割り込み処理）

256μs 周期の割り込みがあったとき、入力チャンネル m ごとに A-Command レジスタの AEm ビットがセットされていたら（イベント計数モード）、以下の処理を行います。

- 入力（PORTC.RCn）を読み取る
- 「一回目フラグ」が立っていたら、フラグをクリアし、読み取った値を保存して終了する

- ENm ビットが 0 のとき、保存してあった値：読み取った値 == 0:1 なら Data(m)に 1 を加える（立ち上がりでカウント）
- ENm ビットが 1 のとき、保存してあった値と読み取った値が違えば Data(m)に 1 を加える（変化でカウント）
- カウントがオーバーフローしたら、OVm ビットをセットする
- RDm ビットをセットする
- 読み取った入力を次回のために保存する

電圧測定（バックグランド処理）

電圧測定機能の実現方法を考えます。入力チャンネル m ごとに A-Command レジスタの AEm:ENm ビットが 0:1（電圧測定モード）だったら、以下の処理を行います。

- ADCON0.CHS（A/D 変換器入力選択）に次の値をセットする

m	測定ポート	ADCCON0.CHS
0	RC2	010010
1	RC3	010011
2	RC4	010100
3	RC5	010101

A/D 変換器の入力選択

- ADCCON0.GO をセットして、変換を開始する
- いったん親プログラムに戻る
- 親プログラム処理から再度呼ばれたら、ADCCON0.GO ビットを調べ、クリアされるまで前のステップから繰り返す（変換終了待ち）
- ADCCON0.GO ビットがクリアされたら、割り込みを禁止する
- ADRESH レジスタを読み取り、上位 6 ビットをクリアして Data（m）レジスタの上位バイトに書き込む
- ADCRESL レジスタのデータを Data(m)レジスタの下位バイトに書き込む
- A-Status レジスタの RDm ビットをセットする
- 割り込みを許可する
- 次のチャンネルに進む

2.4.5 バックグランド処理

バックグランド処理では、コマンドの解釈と電圧測定を同時に実行するため、以下の動作を繰り返します。

- T-Command レジスタに新しいコマンドが書き込まれていたら、その解釈を実行する
- A-Command レジスタに新しいコマンドが書き込まれていたら、その解釈を実行する
- 電圧測定を実行・再開する

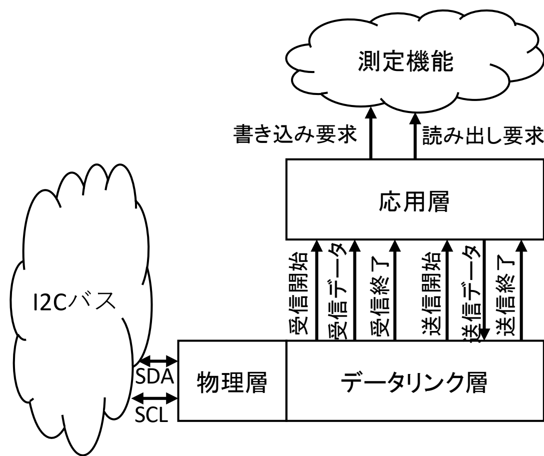
2.4.6 I2C 通信機能

I2C 通信の仕様と、ベースチップのハードウエアで実現できることは、最後の章で説明しています。PICCOLO チップで実現するのは、スレーブ機能です。実装を簡略化するため、以下の制限を行います。

- （物理層の制限）クロック周波数は 100kHz に限定する
- （データリンク層の制限）7 ビットアドレスだけに応答する
- （データリンク層の制限）ジェネラルコール（アドレス 0x00 を使って、全部のスレーブに命令を送ること）には応答しない
- （応用層の制限）データリンク層で一回に伝送するのは、内部レジスタ 1 個分（8 ビットあるいは 16 ビット）のみ
- （応用層の制限）Command レジスタは書き込めるが読み出せない、他のレジスタは読み出せるが書き込めない

応用層とデータリンク層

I2C のファームウエアをデータリンク層と応用層に分けて設計します（物理層はすべてハードウエアで処理される）。シリアル通信では、一回の送受信で転送されるデータを一括して、データリンク層と応用層の間でやり取りするのが一般的です。しかし最後の章で指摘したように、I2C 通信では転送バイト数が（特に読み取り手順では）データリンク層で判断できません。そこで、両者のインターフェースはバイト単位にならざるを得ません。



I2C 通信の実装設計

データリンク層は、受信あるいは送信の開始と終了を応用層に知らせて、1 バイトずつデータを与えたり、取り出したりします。

初期化

ハードウエア（MSSP）を以下のように初期化し、最後に割り込みを許可します。

レジスタ名	設定値	設定内容
SSP1CLKPPS	0001 0000	SCL を RC0 から受け取る
SSP1DATPPS	0001 0001	SDA を RC1 から受け取る
RC0PPS	0001 0101	SCL を RC0 に与える
RC1PPS	0001 0110	SDA を RC1 に与える
SSP1STAT	1000 0000	100kHz
SSP1CON1	0011 0110	7 ビットアドレス、スレーブ
SSP1CON2	0000 0000	クロックストレッチは送信のみ
SSP1CON3	0100 0000	正常受信時のみ ACK を返す
SSP1MSK	1111 1110	アドレスは完全一致のみ処理
SSP1ADD	0111 0000	アドレス 0x38
PIE3	0000 0001	割り込み許可

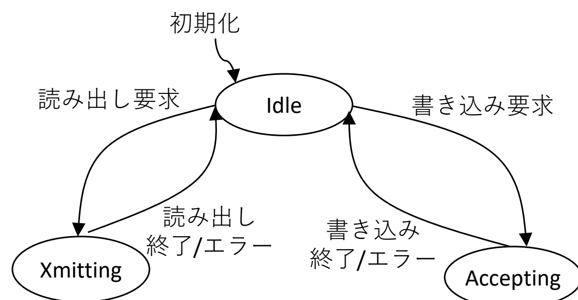
I2C 通信（MSSP）の初期化

データシートに不思議な記述を見つけました。『MSSP を SM バス（しきい値が微妙に異なる）として使うときは、その用途に使えるバッファを装備している RC3/RC4 ポートを使うこと』というものです。それなら、デフォルトのポートを RC3/RC4 にしておいてくれればいいのに、なぜか RC0/RC1 になっています。I2C 通信は 3.3V をオープンドレインで駆動するので、どのポートを使っても問題はないのですが、理由が分かりません。内部配線を決める PPS 回路網では、MSSP 入力は RC0/RC1 がデフォルトになっています。RC0/RC1 への出力信号は表のように設定する必要があります。

データリンク層

データリンク層の動作は複雑なので、ステートマシンにしておきます。状態は Idle、Receiving、Sending の 3 つで、発生した割り込みと内部状態によって動作と状態を決めていきます。

ステートマシンの概要を次に示します。



データリンク層の状態遷移図（概要版）

ステートマシンの設計で考慮すべき項目を拾い出しておきます。

割り込み要因	表示されるフラグ	設計
アドレス受信	SSP1STAT.DA/	Idle 時に処理
データ受信	SSP1STAT.BF	Receiving 時に処理
データ送信完了	SSP1STAT.BF/	Sending 時に処理
STOP 受信	SSPSTAT.P	送受信終了
受信オーバーフロー	SSP1CON1.SSPOV	クリア、受信終了
START 受信	SSP1STAT.S	（起させない設計）
送信データ上書	SSP1CON1.WCOL	（起らない設計）
送信バス衝突	PIR3.BCL1F	（起させない設計）
要チェック項目	**表示されるフラグ**	**チェック・タイミング**
アドレス/データ別	SSP1STAT.DA/	データ受信時
受信確認のチェック	SSP1CON2.ACKSTAT	データ送信後
受信オーバーフロー	SSP1CON1.SSPOV	Receiving 中の割込

ステートマシンの遷移要因

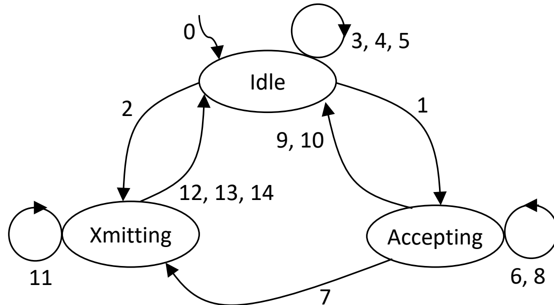
表の中で「起させない」となっている部分は SFR の設定で禁止している要因、「起らない」となってる部分は設計上で対応する要因ですが、念のため、割り込み処理中にチェックします。

ここまでの検討をもとに、データリンク層のステートマシンを設計します。動作については、最後の章「**I2C 通信の概要**」を参照してください。

面倒なのは、『繰返し START』への対応です。START デリミタが状態遷移の要因なのですが、START を検出するたびに割り込みが起こると、処理が重くなってしまいます（アドレスを受信する前の現象なので、自分以外への要求でも割り込みが発生する）。そこで受信があれば、毎回アドレスかどう

かを確認することで代用します。読み出し中に『繰返し START』が発生することは、あまりないと思うのですが、そのときには NACK でデータ送信を終わらせてくれると期待しています。

実装版の状態遷移図は以下のようになりました。



データリンク層の状態遷移図（実装版）

図の中にある数字は、次の状態遷移表の番号です。こうやって状態とイベント、条件ごとに処理を決めることで、設計を行います。

この部分は割り込み処理（ISR）として実装します。Idle と Accepting の処理は似ている部分が多いので、状態遷移表でもまとめています。

番号	始状態	イベント	←のフラグ表現	処理	終状態
0	?	初期化		MSSP ハードウエアと変数を初期設定する	Idle
1	Idle	アドレス+W（書き込み要求）を受信	SSP1STAT.BF & !SSP1STAT.DA & !SSP1SAT.RW	送受信バッファ（SSP1BUF）を空読みする 応用層に受信開始を知らせる	Accepting
6	Accepting				
2	Idle	アドレス+R（読み出し要求）を受信	SSP1STAT.BF & !SSP1STAT.DA & SSP1SAT.RW	送受信バッファ（SSP1BUF）を空読みする 応用層に送信開始を知らせる 応用層から送信データ受け取り、送受信バッファ（SSP1BUF）に入れる クロックストレッチを解除（SSP1CON1.CKP = 1）して送信を開始する	Xmitting
7	Accepting				
3	Idle	STOP を検出	SSP1STAT.P	（何もしない）	Idle
4	Idle	受信オーバーフロー	SSP1CON1.SSPOV	SSP1CON1.SSPOV をクリアする	Idle
5	Idle	データを受信	SSP1STAT.BF & SSP1STAT.DA	（(通常はあり得ない）送受信バッファ（SSP1BUF）を空読みする	Idle
8	Accepting	データを受信	SSP1STAT.BF & SSP1STAT.DA	送受信バッファ（SSP1BUF）のデータを応用層に渡す	Accepting
9	Accepting	STOP を検出	SSP1STAT.P	応用層に受信終了を知らせる	Idle
10	Accepting	受信オーバーフロー	SSP1CON1.SSPOV	SSP1CON1.SSPOV をクリアする 応用層に受信終了を知らせる	Idle
11	Xmitting	送信が完了し、さらなるデータ要求がある	!SSP1STAT.BF & !SSP1CON2.ACKSTAT	応用層から送信データ受け取り、送受信バッファ（SSP1BUF）に入れる クロックストレッチを解除（SSP1CON1.CKP = 1）して送信を開始する	Xmitting
12	Xmitting	送信が完了し、さらなるデータ要求はない	!SSP1STAT.BF & SSP1CON2.ACKSTAT	応用層に送信終了を知らせる	Idle
13	Xmitting	STOP を検出	SSP1STAT.P	応用層に送信終了を知らせる	Idle
14	Xmitting	バス衝突を検出	PIR3.BCL1F	（通常はあり得ない）PIR3.BCL1F をクリアする	Idle
(15)	Any	START を検出	SSP1STAT.S	（割り込み禁止なので起こらない）（何もしない）	遷移しない

データリンク層状態遷移表

応用層

データリンク層からの通知と要求にあわせた動作を設計します。

受信データで読み出しレジスタが指定されたら、そのデータを取り出して送信の準備をします。送信時にさらにデータ要求されたら、測定機能から次のデータを取り出して送るようにもできるのですが、今回の実装では 1 レジスタだけに限定しました。

時間測定、汎用測定モジュールでは、測定値を読み取ったときに「この測定値はもう使ったよ」という意味で、ステータスレジスタの Ready ビットをクリアしておきます。

通知・要求	応用層の処理
受信開始	受信メモリを初期化する
受信データ	A．　１バイト目がコマンドレジスタ番号だったら、２バイト目をレジスタに書き込む。以後は受信データを破棄する B．　１バイト目がステータスレジスタ番号あるいはデータレジスタ番号だったら、該当するレジスタを読み出し、送信メモリに書き込む。以後は受信データを破棄する。 C．　１バイト目が上位以外だったら、そのデータと以後のデータを破棄する
受信終了	２バイト目が与えられていなければ、すべての受信データを破棄する。
送信開始	今回の実装では何もしない
送信データ	A．　送信メモリの先頭データを返す B．　送信メモリが空のときは、ダミーデータを返す
送信終了	送信メモリにデータが残っていたら破棄する

応用層の処理

2.5　モジュール構成

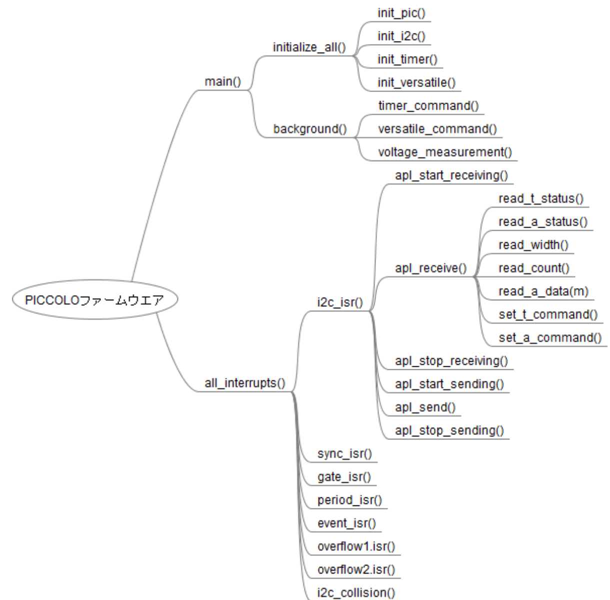
ここまでの実装設計をもとに、ファームウエアのモジュール構成を決めます。機能のまとまりを考えると、二つの測定機能と通信機能、それに基本部に分類できます。それぞれをモジュールとし、インクルードファイルとプログラムファイルで構成します。

モジュール	ファイル名
基本部モジュール	piccolo.h configuration.h main.c
I2C 通信モジュール	i2c.h i2c_module.c
時間測定モジュール	timers.h timer_module.c
汎用測定モジュール	versatile.h versatile_module.c

モジュールへの分解

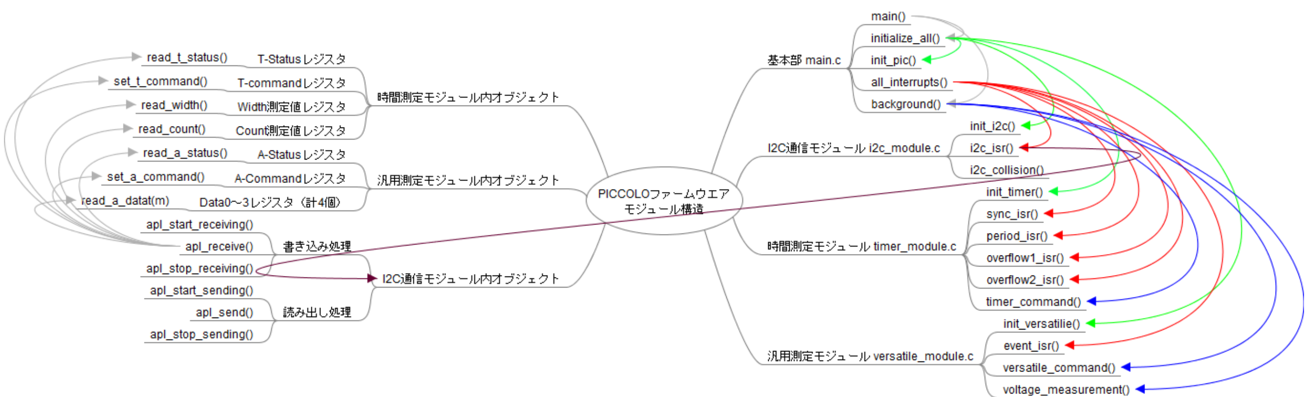
インクルードファイルは、configuration.h を除き、各モジュールの関数プロトタイプと、モジュール固有の定義を記述しています。PIC のコンフィギュレーション定義は、configuration.h にまとめました。

次に、ここまでの設計を関数コールの連鎖としてまとめます。右上の図で上半分がリセット後に順番に実行される処理（初期化とバックグランド処理）、下半分が割り込み処理です。



ファームウエアの関数コール連鎖

これをもとに、関数を各モジュールにまとめなおしたのが次の図です。右半分がモジュールとそこに含まれる関数で、初期化、割り込み、バックグランド処理で呼び出される様子を示しています。左半分は測定機能と I2C 通信機能（応用層）のオブジェクトにアクセスするための関数で、それぞれのモジュールの一部です。



ファームウエアのモジュール構成

3　モジュール作成

この章の作業はすべて、PC にインストールした MPLAB X IDE（＋XC8 コンパイラ）上で行います。プログラムの骨格作りの段階では、使い慣れた他のエディタも使いました。

コーディングは全て C 言語で行いました。最適化レベルがあまり高くないと言われる無償版のコンパイラでは、条件分岐処理のステップ数が、すべてアセンブリ語で記述した場合の 4 倍になるという報告を見かけました。しかし、必要ならクロック周波数（初期設計時点で 8MHz）を 4 倍（32MHz）にすればいいだけなので、C 言語に統一しました。ステートマシンのように複雑な論理を、アセンブリ語で組み上げる手間と労苦を減らすためです。

使用する PIC プログラムコードは、すべてこの章に掲載しています。掲載欄を圧縮するため、機械的にインデントを減らしたり、空行を削除したりしたので、プログラムの構造が見えにくくなってしまいました。巻末に載せたリンクから、このプロジェクトで使用するファイルをすべてダウンロードできるようにしているので、実ファイルの参照をお勧めします。

3.1　コーディング上の注意事項

3.1.1 C 言語の使い方について

コーディング（実際にプログラムを書き下していく作業）上の注意事項です。Microchip 社の C 言語（XC8）で SFR の特定のビットフィールドを操作するには、【SFR 名】`bits.`【ビットフィールド名】と指定します。例えば、`INTCON` レジスタの `PEIE` ビットフィールドは、`INTCONbits.PEIE` で指定できます（チップごとのインクルードファイルで、構造体として定義されている）。これまでの『説明』で使った方法（`INTCON.PEIE`）と違うので間違わないようにしてください（コンパイルエラーが発生します）。8 ビットのレジスタ全体は `INTCON` で指定します。

今回のプロジェクトでは、ひとつ自前のコーディング規則を作りました。プログラムの説明をするコメントは、全て”`/*` ”と“ `*/`”で囲む（昔ながらの）形式で記述しました。オプションの非選択と、検証中にプログラムを一時的に削除するときだけは、”`//`”を使いました。一次的なコメントアウトであることを明示して、戻し忘れを防ぐためです。

3.1.2 モジュール化する

前の章で説明したように、PICCOLO チップのプログラムは 4 つのモジュール（9 個のファイル）で構成することにしました。このうち、コンフィギュレーションファイルを除く 8 個のファイルについて説明します。

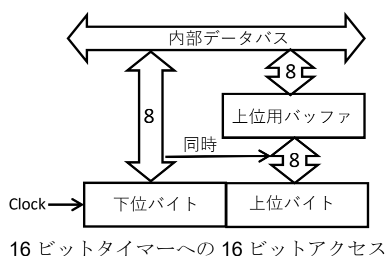
インクルードファイル(xxx.h)は、そのモジュールに関する定義をまとめたもので、コンパイル時のオプション、関数のプロトタイプ宣言、データ型の定義、定数の定義、有用なマクロの定義などを記述しています。

プログラムファイル（yyy.c）に含まれるのは、初期化、割り込み処理（isr)、バックグランド処理を行う関数です。それらが呼び出す下位関数（一部はマクロで定義しています）もソースファイルに含めました。

複数のモジュールで同じデータにアクセスするのに一番簡単な方法は共有メモリです。小規模の実装では、ひとつの変数に対して複数のモジュールから読み書きすることもありますが、あまり推奨される方法ではありません。そこでオブジェクト指向のアプローチとして、モジュール内の変数は全て static（他のモジュールからは直接読み書きできない）とし、アクセスする必要がある時は、そのための関数を呼ぶという方法を採りました。各モジュール内の構造のうち、他のモジュールに開放されるのはインクルードファイルにある関数プロトタイプ宣言だけです。

3.1.3 16 ビット SFR へのアクセスはマクロを使う

タイマー#1 の `T1CON` レジスタには、`RD16` というビットフィールドがあり、これをセットすると、前述の 16 ビット SFR の読み取り問題が解決できるように設計されています。上位バイト用のバッファレジスタが用意されていて、下位バイトを読み取ると、同時に上位バイトがこのバッファレジスタにコピーされます。上位バイトはそのあとにこのバッファレジスタから読み込みます。タイマーに値を書き込むときには、先にこのバッファレジスタに上位バイトを書き込んでおき、下位バイトを書き込むと 16 ビットが同時にタイマー#1 に転送されます。



16 ビットタイマーへの 16 ビットアクセス

ところが、C 言語の unsigned short 変数アクセスが、この順番で行われないので、通常の代入では期待した効果が得られません。そのための関数とマクロを用意しました。タイマー#0 は、16 ビット長のカウンタとして使うと、自動的にこのモードになるので、同じマクロを使います。

同じ機能が数値制御発信器（NCO）にもあります（こちらは 20 ビット）。最後に下位バイトを書き込む必要がありますが、この SFR は初期化のときにしかアクセスしないので、初期化プログラムの中で手順を記述しています。

3.1.4 割込みの悪影響を防ぐ

16 ビットデータを扱うときの問題は、SFR に限りません。PIC16F は 8 ビット MPU（転送は必ず 8 ビットの W レジスタを介して行う）なので、16 ビットデータを一度に扱えません。例えば 16 ビット変数をコピーする C 言語のステートメント `dest = src;` を考えると、アセンブリ語レベルでは次の 4 ステップの転送命令に分解されます。

1. `src` の下位バイト → W レジスタ
2. W レジスタ → `dest` の下位バイト
3. `src` の上位バイト → W レジスタ
4. W レジスタ → `dest` の上位バイト

ステップ 1 とステップ 3 の間に割り込みがあり、その処理内で `src` の値を書き換えられると、`dest` の上位バイトと下位バイトの整合がなくなってしまいます。

幸いベースチップは多重割り込み（割り込み処理中に、もういちど割り込まれること）がないので、割り込み処理と初期化（割り込みが禁止されている）では、この問題が起きません。

バックグランド処理（割り込みで書き換えられる）の中で 16 ビットのデータ処理をするときは、

A. 処理の前に割り込みを禁止し、処理後に割り込みを許可する
B. 処理が長くなるときは、書き換えられる変数を（上の手順 A で）コピーしてから使う

ようにします。

3.1.5 割り込み要因の判別に注意する

ベースチップの割り込みレベルは一つしかないので、`__interrupt` 宣言した関数が要因を判別する必要があります。その判定のもとになるのは、次のような PIC の動作です。

- 要因となる現象が起こると、`PIR(n)` レジスタの該当ビットフィールドがセットされる
- そのとき、`PIE(n)` レジスタの該当ビットフィールドがセットされていると割り込みが発生する

このとき忘れてはならないのは、`PIR(n)` レジスタのビットフィールドは、`PIE(n)` レジスタの設定によらずセットされるということです。測定していないときには、該当する `PIE` レジスタはセットされませんが、それにかかわらず `PIR` レジスタがセットされることがあります。しかもこのビットはファームウエアがクリアしなければなりません。

セットされた `PIR` レジスタは必ずクリアしますが、割り込み処理を実行するかどうかは `PIE` レジスタを見て決めることにしました。

白状すると、最初はこれを見逃していました。シミュレータでは見つからず、実チップで予期せぬ割り込み処理が走っていることが分かったのです。現象を見つけるのも、原因を探すのも難しいので、忘れないようにしたいものです。最初のコンパイル時からコードサイズが大きくなってしまったのは、主にこれと、次に説明する初期化が原因です。

3.1.6 使用する SFR は全て初期化する

ベースチップには、演算に使う『コアレジスタ』とその退避領域を除いて、234 個（表示専用も含む）の特殊機能レジスタ（SFR）があります。これらをすべて覚えて使いこなすのは至難の業です。そこで、使うハードウエアを限定し、その使い方に集中することにしました。

周辺ハードウエアは PMD0～PMD5 レジスタを設定することで、動作を止めることができます。そこで、以下の回路以外は動作を止め、それに関連した SFR は無視できるようにします。

- システムクロック（周辺回路への供給）
- 基準電圧源
- 数値制御発信器（NCO）
- タイマー#0～#2
- A/D 変換器

- PWM3
- MSSP（I2C 通信）
- CLC#1

これらの回路と、RA/RC ポートの SFR が設定の対象です。

SFR は『リセット直後のデフォルト値』を信用せず、すべて初期化しました。C コンパイラが追加した初期化ルーチンが思わぬ設定を行うかもしれないからです。SFR の 8 ビットすべてを初期化し、その後は特定ビットフィールドのセット／クリアを行うようにしました。

3.1.7 機能に目が届く大きさにする

過去のプロジェクトで説明したことですが、プログラムは簡潔で美しい（目が届く範囲に収まっている）ことが大切です。

4 つのモジュールは少し大きめですが、初期化、割り込み処理、バックグランド処理の 3 つに分けてみれば、適当な大きさにまとまったと思います。

3.2 PIC 基本部

PIC ファームウエアを構成する定食メニューには、

1. コンパイラが勝手に付け加える初期化処理
2. ハードウエアのコンフィギュレーション
3. 初期化後に呼ばれる main 関数
4. 割り込み処理

があります。このうち 1 は目に見えないので、2〜4 を作成します。ただし、3 と 4 の実体は各機能モジュールに記述するようにしたので、基本部は、それをコールするだけです。

3.2.1 コンフィギュレーシ

コンフィギュレーションを行うには、次のようにしました。IDE で設定を指定すればコードを自動生成してくれ、「このコードを main 関数の前にコピーしろ」と言われます。基本部モジュールのソースコードと比べて、分量が多くて目障りなので、別ファイルにして、モジュール冒頭でインクルードするようにしました。あまり一般的な手法ではありません。

configuration.h

```
/* configuration.h PICCOLO チップコンフィギュレーション定義
  初版： 2021/3/21 Chuji
 最新版：
MPLAB XC8 コンフィギュレーションツールで自動生成
*/
#ifndef __PICCOLO_CONFIG
// PIC16LF15325 Configuration Bit Settings
// 'C' source line config statements
// CONFIG1
#pragma config FEXTOSC = OFF    // External Oscillator mode selection bits (Oscillator not enabled)
#pragma config RSTOSC = HFINT32 // Power-up default value for COSC bits (HFINTOSC with OSCFRQ= 32 MHz and CDIV = 1:1)
#pragma config CLKOUTEN = OFF   // Clock Out Enable bit (CLKOUT function is disabled; i/o or oscillator function on OSC2)
#pragma config CSWEN = ON       // Clock Switch Enable bit (Writing to NOSC and NDIV is allowed)
#pragma config FCMEN = OFF      // Fail-Safe Clock Monitor Enable bit (FSCM timer disabled)
// CONFIG2
#pragma config MCLRE = ON       // Master Clear Enable bit (MCLR pin is Master Clear function)
#pragma config PWRTE = OFF      // Power-up Timer Enable bit (PWRT disabled)
#pragma config LPBOREN = OFF    // Low-Power BOR enable bit (ULPBOR disabled)
#pragma config BOREN = OFF      // Brown-out reset enable bits (Brown-out reset disabled)
#pragma config BORV = LO        // Brown-out Reset Voltage Selection (Brown-out Reset Voltage (VBOR) set to 1.9V on LF, and 2.45V on F Devices)
#pragma config ZCD = OFF        // Zero-cross detect disable (Zero-cross detect circuit is disabled at POR.)
#pragma config PPS1WAY = OFF    // Peripheral Pin Select one-way control (The PPSLOCK bit can be set and cleared repeatedly by software)
#pragma config STVREN = OFF     // Stack Overflow/Underflow Reset Enable bit (Stack Overflow or Underflow will not cause a reset)
// CONFIG3
#pragma config WDTCPS = WDTCPS_31// WDT Period Select bits (Divider ratio 1:65536; software control of WDTPS)
#pragma config WDTE = OFF       // WDT operating mode (WDT Disabled, SWDTEN is ignored)
#pragma config WDTCWS = WDTCWS_7// WDT Window Select bits (window always open (100%); software control; keyed access not required)
#pragma config WDTCCS = SC      // WDT input clock selector (Software Control)
// CONFIG4
#pragma config BBSIZE = BB512   // Boot Block Size Selection bits (512 words boot block size)
#pragma config BBEN = OFF       // Boot Block Enable bit (Boot Block disabled)
#pragma config SAFEN = OFF      // SAF Enable bit (SAF disabled)
#pragma config WRTAPP = OFF     // Application Block Write Protection bit (Application Block not write protected)
#pragma config WRTB = OFF       // Boot Block Write Protection bit (Boot Block not write protected)
#pragma config WRTC = OFF       // Configuration Register Write Protection bit (Configuration Register not write protected)
#pragma config WRTSAF = OFF     // Storage Area Flash Write Protection bit (SAF not write protected)
#pragma config LVP = ON         // Low Voltage Programming Enable bit (Low Voltage programming enabled. MCLR/Vpp pin function is MCLR.)
// CONFIG5
#pragma config CP = OFF         // UserNVM Program memory code protection bit (UserNVM code protection disabled)
// #pragma config statements should precede project file includes.
// Use project enums instead of #define for ON and OFF.
#define __PICCOLO_CONFIG
#endif
```

3.2.2インクルードファイル

インクルード（ヘッダー）ファイル piccolo.h では、外部インターフェースであるコマンド/ステータス/

測定値レジスタの構造と、その解釈に使うマクロを定義しました。また、測定・通信機能では直接使わない SFR の設定値も定義しています。

また PPS 回路網を変更する前には、ロックを解除する手続きが必要です。しかしデータシートには『プログラム例』が示されているだけで、どこからどこまでが必須なのか分かりません。仕方なく、『プログラム例』の全ての処理を行うマクロを作りました。

piccolo.h

```
/* piccolo.h PICCOLO チップ定義
 初版： 2021/3/21 Chuji
 最新版：2021/6/10
*/
#ifndef __PICCOLO_MODULE
/* ファームウエアの動作切替オプション */
/* CPU クロックを選ぶ */
//#define FAST_CPU      /* CPU クロックを 32MHz にする
*/
#define MID_CPU       /* CPU クロックを 16MHz にする */
/* どちらも定義されていないときは 8MHz。両方定義されていたら 32MHz */
/* I2C 通信の受信漏れを防ぐため、受信時もクロックストレッチを行うオプション */
//#define SAFE_I2C
/* I/O ピンに内部信号を出力させるオプション（検証時のみ）
*/
//#define MONITOR_HW    /* 実チップ上でハードウエアの状態をモニターする場合コメントを外す */
#ifdef MONITOR_HW    /* 以下の一方だけを選ぶこと */
//#define MONITOR_WINDOW       /* RC5 に周波数測定用ゲート信号を出力する */
//#define MONITOR_START    /* RC5 に I2C 通信の D_nA フィールドを出力する */
#endif
//#define FORCE_BP     /* 最適化のために到達しない行に強制的にブレークポイント行を設ける */
#define DUMMY_STATEMENT()    asm("nop")   /* ブレークポイント用のダミー命令 */
/* 関数宣言 */
/* SFR などのビットフィールド処理 */
#define setFlag(x)       x = 1
#define clearFlag(x)    x = 0
#define testFlag(x)      x
/* コマンド/ステータス/データレジスタの構造定義 */
typedef union {
  unsigned char whole_byte;
  struct {
    unsigned  RDT:1;
    unsigned  OVT:1;
    unsigned     :2;
    unsigned  RDP:1;
    unsigned  OVP:1;
    unsigned     :2;
  };
} Tstatus_t;
typedef union {
  unsigned char whole_byte;
  struct{
    unsigned  RD0:1;
    unsigned  OV0:1;
    unsigned  RD1:1;
    unsigned  OV1:1;
    unsigned  RD2:1;
    unsigned  OV2:1;
    unsigned  RD3:1;
    unsigned  OV3:1;
  };
} Astatus_t;
typedef union {
  unsigned char whole_byte;
  struct {
    unsigned  WE:1;
    unsigned  WM:1;
    unsigned  WP:1;
    unsigned  WC:1;
    unsigned  PE:1;
    unsigned  PM0:1;
    unsigned  PM1:1;
    unsigned  PP:1;
  };
} Tcommand_t;
typedef union {
  unsigned char whole_byte;
  struct{
    unsigned  EN0:1;
    unsigned  AE0:1;
    unsigned  EN1:1;
    unsigned  AE1:1;
    unsigned  EN2:1;
    unsigned  AE2:1;
    unsigned  EN3:1;
    unsigned  AE3:1;
  };
} Acommand_t;
/* I2C バスを介して見えるレジスタ番号 */
#define INDEX_T_STATUS      (unsigned char) 0x0
#define INDEX_A_STATUS      (unsigned char) 0x1
#define INDEX_T_COMMAND     (unsigned char) 0x2
#define INDEX_A_COMMAND     (unsigned char) 0x3
#define INDEX_WIDTH         (unsigned char) 0x4
#define INDEX_COUNT         (unsigned char) 0x6
#define INDEX_DATA0         (unsigned char) 0x8
#define INDEX_DATA1         (unsigned char) 0xa
#define INDEX_DATA2         (unsigned char) 0xc
#define INDEX_DATA3         (unsigned char) 0xe
/* レジスタ番号の検査 */
#define isCommand(x)     (!(x & 0xfc)x && (x &
0x2)) /* 0b0000 001x */
#define isStatus(x)      (!(x & 0xfe))
#define isData(x)        (!(x & 0xf1) && )
/* 各レジスタのバイト数 */
#define STATUS_LEN          1
#define COMMAND_LEN         STATUS_LEN
#define DATA_LEN            2
#ifdef MONITOR_HW
#define NUM_CHANNELS    2
#else
#define NUM_CHANNELS    4
#endif
#define getVdata_index(x)   (unsigned char) ((x >>
1) - 4)  /* レジスタ番号から汎用入力データ番号を得る */
#define isVindex(x)         (unsigned char) (x <
NUM_CHANNELS) /* 汎用入力データ番号か調べる */
                       /* x: unsigned char は常に 0
以上 */
/* 実装上定義したデータ型と境界値 */
#define statusRegister  unsigned char
#define commandRegister unsigned char
#define dataRegister   unsigned short
#define initData        (unsigned short) 0
#define initByte        (unsigned char) 0
#define MaxData         (unsigned short) 0xffff
#define initStatus      (unsigned char) 0
/*コンフィギュレーション */
/* SFR 初期設定値（測定機能で使うものはそれぞれのインクルードファイルで記述 */
#define MY_OSCCON1 0b01100000 /* クロック周波数 =
HFINTOSC/1 */
#define MY_OSCCON3 0b00000000 /* ファームウエアによるクロック変更を許可 */
#define MY_OSCEN   0b01100000 /* 500kHz/31.25kHz クロックを使用する */
#ifdef FAST_CPU
#define MY_OSCFRQ  0b00000110 /* HFINTOSC = 32MHz
*/
#else
#ifdef MID_CPU
#define MY_OSCFRQ  0b00000101 /* HFINTOSC = 16MHz
*/
#else
#define MY_OSCFRQ  0b00000011 /* HFINTOSC = 8MHz
*/
#endif
#endif
#define MY_OSCTUNE 0b00000000 /* クロックは較正済周波数 */
```

```
/* 周辺ハードウエアの使用宣言…不要な回路は停止させておく
*/
#define MY_PMD0     0b00000111   /* システムクロックと
基準電圧回路を使う */
#define MY_PMD1     0b00000000   /* タイマーは全て使う
*/
#define MY_PMD2     0b01000111   /* A/D 変換器を使う
*/
#define MY_PMD3     0b00111011   /* PMW3 回路を使う */
#define MY_PMD4     0b11000001   /* MSSP 回路（I2C 通
信用）を使う */
#define MY_PMD5     0b00011100   /* CLC1 回路を使う */
/* 割り込み SFR の初期値 */
#define MY_INTCON   0b01000001   /* 測定回路からの割り
込みを許可、SYNC 立ち上がり検出 */
#define NO_INTERRUPT     0b00000000 /* 初期値：すべて
の割り込み禁止…必要なら各モジュールで再定義する */
#define MY_PIE0     NO_INTERRUPT
#define MY_PIE1     NO_INTERRUPT
#define MY_PIE2     NO_INTERRUPT
#define MY_PIE3     NO_INTERRUPT
#define MY_PIE4     NO_INTERRUPT
#define MY_PIE5     NO_INTERRUPT
#define MY_PIE6     NO_INTERRUPT
#define MY_PIE7     NO_INTERRUPT
#define MY_PIR0     NO_INTERRUPT
#define MY_PIR1     NO_INTERRUPT
#define MY_PIR2     NO_INTERRUPT
#define MY_PIR3     NO_INTERRUPT
#define MY_PIR4     NO_INTERRUPT
#define MY_PIR5     NO_INTERRUPT
#define MY_PIR6     NO_INTERRUPT
#define MY_PIR7     NO_INTERRUPT
/* 割り込み禁止／許可の関数定義 */
#define enableInterrupt()    ei()
#define disableInterrupt()   di()
/* PPS（内部回路網）変更手順 */
#define UnlockPPS() clearFlag(INTCONbits.GIE); \
          PPSLOCK = 0x55; \
          PPSLOCK = 0xAA; \
          clearFlag(PPSLOCKbits.PPSLOCKED)
#define LockPPS()   clearFlag(INTCONbits.GIE); \
          PPSLOCK = 0x55; \
          PPSLOCK = 0xAA; \
          setFlag(PPSLOCKbits.PPSLOCKED)
/* 実ハードウエア上での動作モニターパルス発生 */
#ifdef MONITOR_HW    /* （デバグ用）RC4/5 をモニター出力
に使う */
#define startISR()  setFlag(LATCbits.LATC4) /* ISR
の処理中を示す */
#define endISR()    clearFlag(LATCbits.LATC4)
#define startFunc() setFlag(LATCbits.LATC5) /* 関
数の処理中を示す */
#define endFunc()   clearFlag(LATCbits.LATC5)
#endif
#define __PICCOLO_MODULE
#endif
```

ファイルの冒頭にあるオプションマクロを次の表にまとめました。大部分は検証途中の試行やモニター用途で、最終的な PICCOLO チップで使用するのは MID_CPU オプションだけです。

マクロ	選択項目	#ifdef	#ifndef
FAST_CPU[1]	CPU クロック	32MHz	8MHz
MID_CPU[1]		16MHz	8MHz
SAFE_I2C	クロックストレッチ	書込時でも	読出時のみ
MONITOR_HW[2]	RC4/5 ポート	パルス出力	IN2/IN3
MONITOR_WINDOW	RC5 の出力	周波数ゲート	―
MONITOR_START		I2C の D/A	―
FORCE_BP	BP 用の行	挿入する	挿入しない

オプションマクロの用途

注 1) FAST_CPU と MID_CPU を同時に#define したときは 32MHz。

注 2) MONITOR_WINODW と MONITOR__START は MONITOR_HW を#define したときのみ有効。同時に#define してはならない。

このファイルで MONITOR_HW を#define すると、実チップ上で評価を行うときに、割り込み処理中であることを示すパルス信号を発生してくれます。

FORCE_BP はデバグ用のちょっと特殊な用途です。プログラムの行にブレークポイントを設定しても、その行がコンパイラの最適化で実行されないと、ブレークしません。そんな時に、意味のない命令 DUMMY_STATEMENT()を挿入して、ブレークポイントを設置できるようにするためのマクロです。

基本部モジュール

基本部モジュール main.c には、コンパイラが付け加えた初期化処理に呼ばれる main()関数と、割り込み処理（ISR: Interrupt Service Routine）を記述します。

main()は、各モジュールの初期化ルーチンを呼んだ後、バックグランド処理（コマンドレジスタの解釈と電圧測定）を順番に呼び出していきます。

割り込み処理 all_interrupts()は、ベースチップに多重割り込み機能がないため、すべての割り込みに対応する必要があります。即応を求められる要因から順番に割り込みがあったかチェックし、フラグをクリアしてから各々の ISR を呼び出します。ひとつの割り込み処理を実行するごとに、いったん処理を終了し、次の割り込みを受けられるようにしています。こうすることで、割り込み処理中に、即応を求められる要因が発生しても、すぐに処理を行えるようになります。

main.c

```
/* piccolo.c PICCOLO チップファームウエア基本部
 初版： 2021/3/21 Chuji
 最新版：2021/6/10
*/
#include "include/configuration.h"
#include <xc.h>
#include <stdbool.h>
#include "include/piccolo.h"
#include "include/timers.h"
#include "include/versatile.h"
#include "include/i2c.h"
void initialize_all(void);
void background(void);
void init_pic(void);
void init_isr(void);
/* メインルーチン：リセット後、スタートアップコードの次に実
行される */
int main(void){
  initialize_all();      /* すべての初期化を行ってから
*/
  enableInterrupt();     /* 割り込みを許可して */
  background();          /* バックグランド処理を行う */
```

```
  return 0;              /* ダミー。実際には、この関数は戻らない */
}
/* 割り込み処理 */
/* 優先度順に PIR レジスタの割り込みビットを調べ、セットされていたら
 割り込みビットをクリアしてから該当する ISR を呼び出し、
 いったんバックグランド処理に戻る */
void __interrupt() all_interrupts(void){
#ifdef MONITOR_HW   /* ISR 処理中は RC4 ポートを H にする */
  startISR();
#endif
  if (testFlag(PIR3bits.SSP1IF)){
    clearFlag(PIR3bits.SSP1IF);        /* I2C 通信 */
    i2c_isr();
  } else if (testFlag(PIR0bits.INTF)){     /* SYNC 信号割り込み */
    clearFlag(PIR0bits.INTF);
    if (testFlag(PIE0bits.INTE))
      sync_isr();
  } else if (testFlag(PIR5bits.TMR1GIF)){ /* GATE 信号終了割り込み */
    clearFlag(PIR5bits.TMR1GIF);
    if (testFlag(PIE5bits.TMR1GIE))
      gate_isr();
  } else if (testFlag(PIR4bits.TMR2IF)) { /* 周波数測定ゲート信号終了割り込み */
    clearFlag(PIR4bits.TMR2IF);
    if (testFlag(PIE4bits.TMR2IE))
      period_isr();
  } else if (testFlag(PIR7bits.NCO1IF)){  /* NCO（定周期）割り込み */
    clearFlag(PIR7bits.NCO1IF);
    event_isr();
  } else if (testFlag(PIR0bits.TMR0IF)){  /* 周波数カウンタ（タイマー#0）オーバーフロー */
    clearFlag(PIR0bits.TMR0IF);
    if (testFlag(PIE0bits.TMR0IE))
      overflow0_isr();
  } else if (testFlag(PIR4bits.TMR1IF)){  /* パルス幅カウンタ（タイマー#1）オーバーフロー */
    clearFlag(PIR4bits.TMR1IF);
    if (testFlag(PIE4bits.TMR1IE))
      overflow1_isr();
  } else if (testFlag(PIR3bits.BCL1IF)){  /* I2C スレーブバス衝突（許可はしていない） */
    clearFlag(PIR3bits.BCL1IF);
    i2c_collision();
  }
#ifdef MONITOR_HW
  endISR();
#endif
  /* この時点で割り込み処理から復帰し、次の割り込みが許可される */
}
/* 初期化：各部の初期化を順番に呼び出す */
void initialize_all(void){
  init_pic();         /* PIC の全般的な設定 */
  init_i2c();         /* I2C 通信のハードウエア設定と変数初期化 */
  init_timer();       /* 時間測定機能のハードウエア設定と変数初期化 */
  init_versatile();   /* 汎用測定機能のハードウエア設定と変数初期化 */
  init_isr();         /* すべての割り込みの準備をする */
}
/* バックグランド処理の実行順番制御 -- この関数は決して戻らない（終わらない） */
void background(void){
  while (true){
    timer_command();
    versatile_command();
    voltage_measurement();
  }
}
/* PIC 基本 SFR 設定 */
/* 割り込み禁止状態で呼ばれる */
void init_pic(void){
  OSCCON1 = MY_OSCCON1;   /* 発振周波数 32MHz、クロック周波数 8MHz */
  OSCCON3 = MY_OSCCON3;   /* ファームウエアによるクロック変更を許可 */
  OSCEN  = MY_OSCEN;      /* 500kHz/31.25kHz クロックを使用する */
  OSCFRQ = MY_OSCFRQ;     /* HFINTOSC = 8MHz */
  OSCTUNE = MY_OSCTUNE;   /* クロックは較正済周波数 */
  /* 周辺ハードウエアの使用選択 */
  PMD0   = MY_PMD0;       /* システムクロックと基準電圧回路を使う */
  PMD1   = MY_PMD1;       /* タイマーは全て使う */
  PMD2   = MY_PMD2;       /* A/D 変換器を使う */
  PMD3   = MY_PMD3;       /* PMW3 回路を使う */
  PMD4   = MY_PMD4;       /* MSSP 回路（I2C 通信用）を使う */
  PMD5   = MY_PMD5;       /* CLC1 回路を使う */
}
void init_isr(void){
  /* 割り込み設定の初期化 */
  PIR0   = MY_PIR0;
  PIR1   = MY_PIR1;
  PIR2   = MY_PIR2;
  PIR3   = MY_PIR3;
  PIR4   = MY_PIR4;
  PIR5   = MY_PIR5;
  PIR6   = MY_PIR6;
  PIR7   = MY_PIR7;
  PIE0   = MY_PIE0;
  PIE1   = MY_PIE1;
  PIE2   = MY_PIE2;
  PIE3   = MY_PIE3;
  PIE4   = MY_PIE4;
  PIE5   = MY_PIE5;
  PIE6   = MY_PIE6;
  PIE7   = MY_PIE7;
  INTCON = MY_INTCON;     /* 測定回路からの割り込みを許可 */
}
```

I2C 通信のステートマシン設計時に検討しておいた、スレーブ衝突の処理（起こらないはず）は、ここの最後に入れてあります。

3.3　時間測定機能

時間測定機能には、パルス幅測定と周波数測定の二つの機能がありますが、両方を一つのモジュールにまとめました。

3.3.1 インクルードファイル

インクルードファイル timers.h では、SFR の初期設定値と、機能別の設定値を定義しています。動作を管理するデータ（1 ビットデータの集まり）を 1 バイトのデータ型 `timer_stat_t` として定義しました。最後にあるのは、ステータス/コマンドレジスタの初期値と、コマンドを解釈するためのマクロです。

timers.h

```
/* timers.h PICCOLO チップ時間・周波数測定モジュール定義
  初版： 2021/3/21 Chuji
 最新版：2021/6/10
*/
#ifndef __TIME_MODULE
#include "piccolo.h"
/* 関数プロトタイプ宣言 */
```

```
/* 初期化 */
void init_timer(void);
/* 割り込み処理 */
void sync_isr(void);
#define gate_isr()  sync_isr()
void period_isr(void);
void overflow0_isr(void);
void overflow1_isr(void);
/* オブジェクトアクセス */
statusRegister read_t_status(void);
void set_t_command(commandRegister);
unsigned short read_width(void);
unsigned short read_count(void);
/* バックグランド処理：コマンド解釈 */
void timer_command(void);
/* このモジュール内で使う型の宣言 */
typedef struct{
  unsigned new_command :1; /* 新しいコマンドを受信したことを示すフラグ */
  unsigned parsing     :1;  /* コマンドの解釈実行中を示すフラグ */
  unsigned sync_mode   :1;  /* パルス幅測定が同期モードであることを示すフラグ */
  unsigned w_notFirst :1;  /* パルス幅測定の一回目フラグ（0 が一回目）*/
  unsigned w_overflow :1;  /* パルス幅測定のオーバーフローフラグ */
  unsigned f_notFirst :1;  /* 周波数測定の一回目フラグ（0 が一回目）*/
  unsigned f_overflow :1;  /* 周波数測定のオーバーフローフラグ */
} timer_stat_t;
#define INIT_TIMER_STATUS   (timer_stat_t) {0} /* 初期化 */
/* SFR 初期値と動作別設定値 */
/* ポートの設定 */
#define MY_PORTA     0b00000000 /* いちおう出力レジスタに 0 を書き込んでおく） */
#define MY_TRISA     0b00110111 /* RA は全て入力 */
#define MY_ANSELA    0b00000000 /* ポートはすべてデジタル入力 */
#define MY_WPUA      0b00110100 /* デジタル入力はプルアップする（プログラム端子を除く） */
#define MY_INLVLA    0b00111111 /* デジタル入力は CMOS レベル */
#define MY_ODCONA    0b00000000 /* ポートは全て入力だから意味がない設定(push/pull) */
#define MY_SLRCONA  0b00000000 /* ポートは全て最大スルーレート */
/* パルス幅計測機能の設定（タイマー#1） */
/* 動作別設定 */
#define T1CLOCK_8us 0b10        /* 500kHz/4 = 8us */
#define T1CLOCK_2us 0b00        /* 500kHz/1 = 2us */
#define T1_GATE_H   0b1         /* パルス幅信号が H レベルの幅を測る */
#define T1_GATE_L   0b0         /* パルス幅信号が L レベルの幅を測る */
/* 初期設定 */
#define MY_T1CON     0b00100110  /* タイマー#1 のクロックは 1/4 分周（非同期） */
#define MY_T1GCON    0b11000000  /* パルス計数にゲート信号を使う */
#define MY_T1CLK     0b00000101  /* 測定クロックは 500kHz */
#define MY_T1GATE    0b00000000  /* PPS 網からゲート信号をもらう */
#define MY_T1GPPS    0b00000100  /* PPS 網で GATE(RA4)をゲート信号にする */
#define MY_INTPPS    0b00000010  /* SYNC(RA2)を割り込み要因にする */
/* 周波数測定機能の設定 */
/* タイマー#2（周期設定）の設定 */
/* 測定時間別設定 */
#define T2CLOCK_1s     0b0110  /* 31.25kHz */
#define T2CLOCK_100ms 0b0110  /* 31,25kHz */
#define T2CLOCK_10ms  0b0101  /* 500kHz */
#define T2CLOCK_FREE  T2CLOCK_10ms
#define T2SCALE_1s     0b111   /* 1/128 */
#define T2SCALE_100ms 0b100   /* 1/16 */
#define T2SCALE_10ms  0b110   /* 1/64 */
#define T2SCALE_FREE  T2SCALE_10ms
#define T2PR_1s        0xf7    /* 247 */
#define T2PR_100ms     0xd7    /* 215 */
#define T2PR_10ms      0x9c    /* 156 */
#define T2PR_FREE      T2PR_10ms
#define PWM3H_1s       0x02    /* パルス幅（データ転送時間）の設定 */
#define PWM3H_100ms    0x13
#define PWM3H_10ms     0x4d
#define PWM3H_FREE     PWM3H_10ms
#define PWM3L_1s       0xc0
#define PWM3L_100ms    PWM3L_1s
#define PWM3L_10ms     PWM3L_1s
#define PWM3L_FREE     PWM3L_1s
#define T0CLOCK_CLC1  0b111   /* TMR0 のクロックは CLC1 の出力（周波数） */
#define T0CLOCK_RA5   0b000   /* TMR0 のクロックは PPS 経由 RA5 ポート（フリーカウント） */
#define CLC_RISE       0b0     /* パルス入力の立ち上がりで計数 */
#define CLC_FALL       0b1     /* パルス入力の立ち下がりで計数 */
/* 初期値 */
/* タイマー#2 で測定タイミングを発生させる */
#define MY_T2CLKCON 0b00000110 /* クロック周波数は 31.25kHz */
#define MY_T2CON    0b11110000 /* プリスケーラは 1/128 分周 */
#define MY_T2HLT    0b00000000 /* フリーランニングモード */
#define MY_T2RST    0b00000000 /* 外部リセット不要 */
#define MY_T2PR     T2PR_1s    /* 247 クロックで繰り返し */
/* PWM#3（ゲート信号発生）の設定 */
/* 測定時間別設定 */
#define PWM3DCH_FREE  0x02
#define PWM3DCL_FREE  0xc0
#define PWM3DCH_1s    0x02
#define PWM3DCL_1s    0xc0
#define PWM3DCH_100ms 0x13
#define PWM3DCL_100ms 0xc0
#define PWM3DCH_10ms  0x4D
#define PWM3DCL_10ms  0xc0
/* 初期値 */
#define MY_PWM3CON  0b10010000 /* PWM 出力は L レベル */
#define MY_PWM3DH   PWM3DCH_1s /* 10ms 待ち時間設定（上位バイト） */
#define MY_PWM3DCL  PWM3DCL_1s /* 10ms 待ち時間設定（下位バイト） */
/* CLC#1（ゲート機能）の設定 */
#define MY_CLC1CON  0b10000010 /* 4 入力 AND ブロックを使う */
#define MY_CLC1POL  0b00000000 /* 入出力反転なし */
#define MY_CLC0SEL0 0b00010001 /* 入力#0 は PWM#3 の出力 */
#define MY_CLC0SEL1 0b00000001 /* 入力#1 は PPS 網からもらう */
#define MY_CLCIN1PPS 0b00000101 /* PPS 網で RA5（PULSE 入力）を入力#1 に供給 */
#define MY_CLC1GLS0 0b00000010 /* AND 入力#1 は PWM#3 */
#ifdef MONITOR_WINDOW
#define SHOW_PWM3  0x0b         /* PPS から PWM3OUT を選択する */
#endif
#define MY_CLC1GLS1 0b00001000 /* AND 入力#2 は PULSE */
#define MY_CLC1GLS2 0b00000011 /* AND 入力#3 は常に 1 */
#define MY_CLC1GLS3 0b00000011 /* AND 入力#4 は常に 1 */
/* タイマー#0（周波数測定用）の設定 */
/* 測定時間別設定 */
```

```
#define T0CLOCK_FREE  0b000 /* T0CKIPPS (PULSE 入力) */
#define T0CLOCK_1s    0b111 /* CLC#1 (ゲート付きPULSE 入力) */
#define T0CLOCK_100ms 0b111 /* CLC#1 (ゲート付きPULSE 入力) */
#define T0CLOCK_10ms  0b111 /* CLC#1 (ゲート付きPULSE 入力) */
/* 初期値 */
#define MY_T0CON0   0b00010000 /* 16 ビットカウンタモード */
#define MY_T0CON1   0b00010000 /* クロック入力は PPSの出力 */
#define MY_T0CKIPPS 0b00000101 /* フリーカウント時の入力は RA5 (PULSE) */
/* ファームウエアレジスタ初期値 */
#define T_INIT_STATUS   (Tstatus_t)  {0} /* 仕様書参照 */
#define T_INIT_COMMAND  (Tcommand_t) {0} /* 仕様書参照 */
/* 機能別のコマンド取り出し */
#define WidthCmd(x)   (x & 0x0f)  /* パルス幅測定コマンド */
#define FreqCmd(x)    (x & 0xf0)  /* 周波数測定コマンド */
#define __TIME_MODULE
#endif
```

3.3.2 時間測定機能モジュール

時間測定機能モジュール timer_module.c は、初期化、割り込み処理、応用層インターフェース、コマンド解釈からなります。

timer_module.c

```
/* timer_module.c PICCOLO チップ時間・周波数測定モジュール
   初版： 2021/3/21 Chuji
  最新版： 2021/6/10
*/
#include <xc.h>
#include <stdbool.h>
#include "include/timers.h"
/* 時間・周波数測定専用データ */
static Tstatus_t t_status;       /* T-Status レジスタ */
static Tcommand_t t_command, t_command_buf; /* T-Command レジスタと受信コマンド */
static timer_stat_t timerStatus;     /* 動作管理変数 */
static dataRegister width_register, count_register; /* 測定値 */
static unsigned char dataH, dataL; /* 16 ビットデータ読み取り用バッファ */
/* 16 ビットカウンタへのアクセス関数 */
#define Read16(dh, dl) {     \
  dataL = dl;  /* 最初に下位バイトを読み取ると、上位バイトがバッファに転送される */ \
  dataH = dh;  /* バッファから上位バイトを読み取る */ \
}
/* 16 ビットデータの再構成 (上の Read16()とペアで使うこと) */
#define Build16()    (dataRegister)(dataH << 8) | dataL
/* 16 ビットカウンタのクリア */
#define Clear16(dh, dl){     \
  dh = initByte;  /* 最初に上位バイト用バッファにデータを書き込む */ \
  dl = initByte;  /* 次に下位バイトに書き込むと、16 ビットデータがカウンタに転送される */ \
}
/* カウンタ SFR と内部変数の初期化 */
/* 割り込み禁止状態で呼ばれる */
```

```
void init_timer(void){
  UnlockPPS();             /* PPS 回路網の変更を許可する */
  /* RA ポートの設定 (時間測定機能共通) */
  PORTA = MY_PORTA;    /* (仮) ポート出力 */
  TRISA = MY_TRISA;    /* RA は全て入力 */
  ANSELA = MY_ANSELA;  /* ポートはすべてデジタル入力 */
  WPUA = MY_WPUA;      /* デジタル入力はプルアップする(プログラム端子を除く) */
  INLVLA = MY_INLVLA;  /* デジタル入力は CMOS レベル */
  ODCONA = MY_ODCONA;  /* ポートは全て入力だから意味がない設定(push/pull) */
  SLRCONA = MY_SLRCONA;    /* ポートは全て最大スルーレート */
  /* パルス幅計測機能の設定 (タイマー#1) */
  T1CON = MY_T1CON;        /* タイマー#1 のクロックは 1/4 分周 */
  T1GCON = MY_T1GCON;      /* パルス計数にゲート信号を使う */
  T1CLK = MY_T1CLK;        /* 測定クロックは 500kHz/2 */
  T1GATE = MY_T1GATE;      /* PPS 網からゲート信号をもらう */
  T1GPPS = MY_T1GPPS;      /* PPS 網で GATE(RA4)をゲート信号にする */
  INTPPS = MY_INTPPS;      /* SYNC(RA2)を割り込み要因にする */
  /* 周波数測定機能の設定 */
  /* タイマー#2 (周期設定) の設定 */
  T2CLKCON = MY_T2CLKCON; /* クロック周波数は 31.25kHz */
  T2CON = MY_T2CON;        /* プリスケーラは 1/128 分周 */
  T2HLT = MY_T2HLT;        /* フリーランニングモード */
  T2RST = MY_T2RST;        /* 外部リセット */
  T2PR = MY_T2PR;          /* 247 クロックで繰り返し */
  /* PWM#3 (ゲート信号発生) の設定 */
  PWM3CON = MY_PWM3CON;    /* PWM 出力は L レベル */
  PWM3DCH = MY_PWM3DH;     /* 10ms 待ち時間設定 (上位バイト) */
  PWM3DCL = MY_PWM3DCL;    /* 10ms 待ち時間設定 (下位バイト) */
  /* CLC#1 (ゲート機能) の設定 */
  CLC1CON = MY_CLC1CON;    /* 4 入力 AND ブロックを使う */
  CLC1POL = MY_CLC1POL;    /* 入出力反転なし */
  CLC1SEL0 = MY_CLC0SEL0; /* 入力#0 は PWM#3 の出力 */
  CLC1SEL1 = MY_CLC0SEL1; /* 入力#1 は PPS 網からもらう */
  CLCIN1PPS = MY_CLCIN1PPS; /* PPS 網で RA5 (PULSE 入力) を入力#1 に供給 */
  CLC1GLS0 = MY_CLC1GLS0;    /* AND 入力#1 は PWM#3 */
  CLC1GLS1 = MY_CLC1GLS1;    /* AND 入力#2 は PULSE */
  CLC1GLS2 = MY_CLC1GLS2;    /* AND 入力#3 は常に 1 */
  CLC1GLS3 = MY_CLC1GLS3;    /* AND 入力#4 は常に 1 */
  /* タイマー#0 (周波数測定用) の設定 */
  T0CON0 = MY_T0CON0;        /* 16 ビットカウンタモード */
  T0CON1 = MY_T0CON1;        /* クロック入力は CLC#1 の出力 */
  T0CKIPPS = MY_T0CKIPPS;    /* フリーカウント時の入力は RA5 (PULSE) */
#ifdef MONITOR_WINDOW   /* 動作確認のため、PWM3 信号を RC5 に出力する */
  RC5PPS = SHOW_PWM3;
#endif
  LockPPS();               /* PPS 回路網の設定を再度禁止する */
  /* 内部変数の初期化 */
  t_status = T_INIT_STATUS;           /* T-Status レジスタ */
```

```
  t_command = T_INIT_COMMAND;         /* T-Command
レジスタ */
  t_command_buf = T_INIT_COMMAND; /* T-Command レジ
スタに設定する新しいデータ */
  timerStatus = INIT_TIMER_STATUS;
  width_register = initData;          /* 時間幅測定値
データレジスタ */
  count_register = initData;          /* 周波数測定値
データレジスタ */
}
/* パルス幅測定割り込み処理 */
/* 割り込み要因には SYNC 割り込みとゲート信号割込みがある
が、処理は同じ
 割り込みフラグは既にクリアされている */
void sync_isr(void){
  if (!testFlag(timerStatus.w_notFirst)){
    setFlag(timerStatus.w_notFirst);    /* 測定開始
後 1 回目のデータは廃棄する */
  } else{
    Read16(TMR1H, TMR1L);        /* 測定値を取り込む
*/
    width_register = Build16(); /* 測定値を再構成する
*/
    setFlag(t_status.RDT);  /* 測定値更新フラグを立て
る */
#ifdef FORCE_BP
    DUMMY_STATEMENT();
#endif
    if (testFlag(timerStatus.w_overflow)){     /*
オーバーフローの反映 */
      setFlag(t_status.OVT);
      clearFlag(timerStatus.w_overflow);
    } else {
      clearFlag(t_status.OVT);
    }
  }
  Clear16(TMR1H, TMR1L);           /* タイマー#1 をク
リアする */
}
/* 周波数測定割り込み */
void period_isr(void){
  if (!testFlag(timerStatus.f_notFirst)){
    setFlag(timerStatus.f_notFirst);    /* 測定開始
後 1 回目のデータは保存しない */
  } else{ /* 2 回目以後は測定値をデータレジスタに保存する
*/
    Read16(TMR0H, TMR0L);        /* 測定値を取り込む
*/
    count_register = Build16(); /* 測定値を再構成する
*/
    setFlag(t_status.RDP);  /* 測定値更新フラグを立て
る */
    if (testFlag(timerStatus.f_overflow)){     /*
オーバーフローの反映 */
      setFlag(t_status.OVP);
    } else {
      clearFlag(t_status.OVP);
    }
  }
  if (testFlag(t_command.PM0) |
testFlag(t_command.PM1)){
    Clear16(TMR0H, TMR0L);     /* フリーランニングモー
ド以外ではタイマー#0 をクリアする */
    clearFlag(timerStatus.f_overflow);
  }        /* フリーランニングモードでは、いったんセットし
たオーバーフローフラグはそのまま */
}
/* タイマー#0（周波数測定）のオーバーフロー処理 */
void overflow0_isr(void){
  setFlag(timerStatus.f_overflow);
}
/* タイマー#1（パルス幅測定）のオーバーフロー処理 */
void overflow1_isr(void){
  setFlag(timerStatus.w_overflow);
}
/* T-Status レジスタの読み取り */
statusRegister read_t_status(void){
  return t_status.whole_byte;
}
/* T-Command レジスタへの書き込み */
void set_t_command(commandRegister command){
```

```
  t_command_buf.whole_byte = command;
  setFlag(timerStatus.new_command);
}
/* パルス幅測定値の読み取り */
dataRegister read_width(void){
  clearFlag(t_status.RDT);  /* データを読み取ったら、
次の測定値を得るまで Ready フラグをクリアする */
  return width_register;
}
/* 周波数測定値の読み取り */
dataRegister read_count(void){
  clearFlag(t_status.RDP);  /* データを読み取ったら、
次の測定値を得るまで Ready フラグをクリアする */
  return count_register;
}
/* パルス幅/周波数測定コマンドの解釈 */
void timer_command(void){
  /* コマンド解釈中に新しいコマンドを受け取っても解釈を続行
し、終了後に再解釈する */
  Tcommand_t new_command;
  disableInterrupt();
    if (testFlag(timerStatus.new_command)) {
      clearFlag(timerStatus.new_command);
      new_command.whole_byte =
t_command_buf.whole_byte;
      setFlag(timerStatus.parsing);
    } else clearFlag(timerStatus.parsing);
  enableInterrupt();
  /* 新しいコマンドを受け取ったら、解釈を始める */
  if (testFlag(timerStatus.parsing)){
    /* パルス幅測定機能の設定処理 */
    if (WidthCmd(new_command.whole_byte) !=
WidthCmd(t_command.whole_byte)){
      /* パルス幅測定の設定に変更があったとき */
      clearFlag(T1CONbits.ON); /* いったんタイマー#1
を止める */
      clearFlag(PIE0bits.INTE); /* すべての割り込を禁
止する */
      clearFlag(PIE5bits.TMR1GIE);
      clearFlag(PIE4bits.TMR1IE);
      clearFlag(PIR0bits.INTF); /* 割り込みフラグをす
べてクリアする */
      clearFlag(PIR5bits.TMR1GIF);
      clearFlag(PIR4bits.TMR1IF);
      /* 変数を初期化する */
      clearFlag(timerStatus.w_overflow);
      clearFlag(timerStatus.w_notFirst);
      /* コマンドによる SFR 設定変更 */
      if (testFlag(new_command.WC)) T1CONbits.CKPS
= T1CLOCK_2us;
      else T1CONbits.CKPS = T1CLOCK_8us; /* クロッ
ク選択 */
      if (testFlag(new_command.WP))
T1GCONbits.GPOL = T1_GATE_L;
      else T1GCONbits.GPOL = T1_GATE_H; /* ゲート信
号の極性選択 */
      if (testFlag(new_command.WE)){    /* 測定を行
うときの処理 */
        clearFlag(t_status.RDT);    /* ステータスと
測定値を初期化する */
        clearFlag(t_status.OVT);
        disableInterrupt();
          width_register = initData;
          Clear16(TMR1H, TMR1L);
        enableInterrupt();
        if (testFlag(new_command.WM)){ /* 非同期モ
ード（ゲート信号の後ろでデータ取り込み） */
          clearFlag(timerStatus.sync_mode);
          setFlag(PIE5bits.TMR1GIE);
        }
        else {                              /* 同期モ
ード（SYNC 信号でデータ取り込み） */
          setFlag(timerStatus.sync_mode);
          setFlag(PIE0bits.INTE);
        }
        setFlag(PIE4bits.TMR1IE); /* オーバーフロー
割り込み許可 */
        setFlag(T1CONbits.ON);    /* タイマー#1 の起
動 */
      }   /* 測定を実行しないときは全ての割り込みを禁止し
たまま */
```

```
    }
    /* 周波数測定機能の設定処理 */
    if (FreqCmd(new_command.whole_byte) !=
FreqCmd(t_command.whole_byte)){
      clearFlag(T0CON0bits.T0EN); /* タイマー#0 を停
止 */
      /* 関連する 2 つの割込みを禁止する */
      clearFlag(PIE0bits.TMR0IE);
      clearFlag(PIE4bits.TMR2IE);
      clearFlag(PIR0bits.TMR0IF);  /* 割り込みフラグ
をすべてクリアする */
      clearFlag(PIR4bits.TMR2IF);
      /* 変数を初期化する */
      clearFlag(timerStatus.f_notFirst);
      clearFlag(timerStatus.f_overflow);
      /* コマンドによる SFR 設定変更 */
      if (testFlag(new_command.PP))
CLC1POLbits.LC1POL = CLC_FALL;
      else CLC1POLbits.LC1POL = CLC_RISE; /* 計数す
る信号変化の設定 */
      /* ゲート時間の設定 */
      if (testFlag(new_command.PM1)){   /* 10ms ま
たは 100ms ゲート */
        if (testFlag(new_command.PM0)) { /* 10ms
*/
          T2CLKCONbits.CS = T2CLOCK_10ms;
          T2CONbits.CKPS = T2SCALE_10ms;
          T2PR = T2PR_10ms;
          PWM3DCH = PWM3H_10ms;
          PWM3DCL = PWM3L_10ms;
          T0CON1bits.T0CS = T0CLOCK_CLC1;
        } else {                  /* 100ms */
          T2CLKCONbits.CS = T2CLOCK_100ms;
          T2CONbits.CKPS = T2SCALE_100ms;
          T2PR = T2PR_100ms;
          PWM3DCH = PWM3H_100ms;
          PWM3DCL = PWM3L_100ms;
          T0CON1bits.T0CS = T0CLOCK_CLC1;
        }
      } else {                   /* 1s ゲートまたはフ
リーランニング */
       if (testFlag(new_command.PM0)) { /* 1s */
          T2CLKCONbits.CS = T2CLOCK_1s;
          T2CONbits.CKPS = T2SCALE_1s;
          T2PR = T2PR_1s;
          PWM3DCH = PWM3H_1s;
          PWM3DCL = PWM3L_1s;
          T0CON1bits.T0CS = T0CLOCK_CLC1;
        } else {                  /* フリーランニン
グ */
          T2CLKCONbits.CS = T2CLOCK_FREE;
          T2CONbits.CKPS = T2SCALE_FREE;
          T2PR = T2PR_FREE;
          PWM3DCH = PWM3H_FREE;
          PWM3DCL = PWM3L_FREE;
          T0CON1bits.T0CS = T0CLOCK_RA5;
        }
      }
      if (testFlag(new_command.PE)){
        /* ステータスを初期化する */
        clearFlag(t_status.OVP);
        clearFlag(t_status.RDP);
        /* 測定値とカウンタを初期化する */
        disableInterrupt();
          count_register = initData;
          Clear16(TMR0H, TMR0L);
        enableInterrupt();
        setFlag(PIE4bits.TMR2IE);   /* タイマー#2 割
り込みを許可 */
        setFlag(PIE0bits.TMR0IE); /* オーバーフロー
割り込みを許可 */
        setFlag(T0CON0bits.T0EN); /* タイマー#0 の起
動 */
      }   /* PE=0 のときは測定値とステータスは前のままに
しておく */
        /* 割り込みも禁止したまま */
    }
    clearFlag(timerStatus.parsing);
    t_command = new_command; /* 現在使用中のコマンド
*/
  }
}
```

初期化 `init_timer()`は、SFR の初期設定と、内部データの初期化を行っています。

割り込み処理のうち測定機能 `sync_isr()`と `period_isr()`は、ほぼ同じ内容です。ハードウエアからの測定終了割込みを受けたら、

- タイマーから測定値を取り出して保存する
- 測定終了をステータスに反映する
- オーバーフローがあったらステータスに反映（管理データの該当ビットはクリア）する
- タイマーを 0 に初期化して次の測定に備える

を行います。

読み出し要求に対しては、保存した測定値を返します。同時にステータスの Ready ビットをクリアします。書き込み要求に対しては、いったんコマンドバッファに保存してフラグを立て、バックグランド処理でコマンドを解釈します。

オーバーフローがあったら、いったん管理データの該当ビットに保存し、測定終了時にステータスに反映します。T-Status レジスタに表示されるオーバーフローと Ready ビットは、データレジスタに保存されている測定値とペアになっているためです。

3.4 汎用測定機能

汎用測定機能では、時間測定機能とは違った形でコマンドとステータスを保持することにしました。ビットマスクとシフトを繰り返すのは煩雑だからです。4 チャンネルある測定入力ごとに構造体データを割り振り、全体で 4 要素の配列にしました。A-Command の分解と A-Status の再構築は、それぞれバックグランド処理（コマンド解釈）と、応用層からのステータス要求があったときに行います。

3.4.1 インクルードファイル

インクルードファイル vesatile.h では、上述のコマンド/ステータスと管理データの構造体を定義しました。SFR の初期設定値と、動作モード毎の設定値も定義しています。

versatile.h

```
/* versatile.h PICCOLO チップ 汎用測定モジュール定義
 初版： 2021/3/21 Chuji
 最新版：2021/6/10
*/
#ifndef __VERSATILE_MODULE
#include "piccolo.h"
/* 関数プロトタイプ宣言 */
/* 初期化 */
void init_versatile(void);
/* 割り込み処理 */
void event_isr(void);
```

```
/* オブジェクトアクセス */
statusRegister read_a_status(void);
void set_a_command(commandRegister);
dataRegister read_a_data(unsigned char);  /* 引数は
4?7(内部レジスタアドレス/2) */
/* バックグランド処理：コマンド解釈 */
void voltage_measurement(void);
void versatile_command(void);
/* このモジュール内で使う型の宣言 */
typedef struct{
  union {
    unsigned stat   :2;
    struct {
      unsigned overflow   :1;
      unsigned ready      :1;
    };
  };
  unsigned event_mode :1; /* イベント計数モード */
  unsigned not_first  :1; /* 一回目フラグ（負論理）
*/
  unsigned previous     :1; /* 前回の入力 */
  unsigned bothEdge   :1; /* 立ち上がり・立ち下がりと
もカウント */
  unsigned voltage_mode   :1; /* 電圧測定モード */
} versatile_t;
typedef struct{
  unsigned new_command    :1; /* 新しいコマンドを受信
したことを示すフラグ */
  unsigned parsing        :1; /* コマンドの解釈実行中
を示すフラグ */
  unsigned converting     :1; /* AD 変換器が動作中を
示すフラグ */
  unsigned current        :3; /* 測定中のアナログチャ
ンネル入力 */
} versatile_status_t;
/* 初期値 */
#define INIT_VERSATILE  (versatile_t) {0}
#define INIT_VERSA_STATUS (versatile_status_t) {0}
#define FIRST_CHANNEL   0 /* 最初の測定チャンネル */
#define CHANNEL_LEN 2   /* A-Status/A-Command レジ
スタのチャンネル毎データ長 */
/* SFR の初期値 */
/* ポート C の設定 */
#define MY_PORTC    0b00000000 /* 出力レベル初期値
（仮） */
#ifdef MONITOR_HW   /* （デバグ用）RC4/5 をモニター出力
に使う */
#define MY_TRISC    0b11001111 /* RC4/5 は出力、他は
は全て入力 */
#define MY_ANSELC   0b00000000 /* ポートはすべてデジ
タル入力 */
#define MY_WPUC     0b00001100 /* デジタル入力はプル
アップする（I2C を除く） */
#define MY_INLVLC   0b11111111 /* デジタル入力は CMOS
レベル */
#define MY_ODCONC   0b00000011 /* RC0/1 はオープンド
レイン出力 */
#define MY_SLRCONC  0b00111100 /* RC0/1 は最大スルー
レート */
#else               /* （最終形態）RC4/5 を IN2/3 とし
て使う */
#define MY_TRISC    0b11111111 /* ポート C は全て入力
*/
#define MY_ANSELC   0b00000000 /* ポートはすべてデジ
タル入力 */
#define MY_WPUC     0b00111100 /* デジタル入力はプル
アップする（I2C を除く） */
#define MY_INLVLC   0b11111111 /* デジタル入力は CMOS
レベル */
#define MY_ODCONC   0b00000011 /* RC0/1 はオープンド
レイン出力 */
#define MY_SLRCONC  0b00111100 /* RC0/1 は最大スルー
レート */
#endif
 /* NCO を定周期割り込み源として使う設定 */
#define MY_NCO1CON  0b10000001 /* パルス出力として使
う */
#define MY_NCO1CLK  0b00000100 /* 31.25kHz クロック
*/
#define MY_NCO1INCL 0b00000000 /* 逐次可算値 */
#define MY_NCO1INCH 0b00000000 /*   250μs 周期で割
り込むには */
#define MY_NCO1INCU 0b00000010 /*   0x20000 に設定す
る */
#ifdef MY_PIE7
#undef MY_PIE7
#endif
#define MY_PIE7     0b00010000 /* 定周期割り込みを許
可する */
 /*  A/D 変換器の設定*/
#define MY_ADCON0   0b11101101 /* A/D 変換器動作、初
期入力は GND */
#define MY_ADCON1   0b11010011 /* 変換速度 2μs/ビッ
ト、基準電圧 2.048V */
#define MY_ADACT    0b00000000 /* 自動トリガを禁止
*/
  /* 基準電圧源の設定 */
#define MY_FVRCON   0b10000010 /* A/D 変換器に
2.048V を供給する */
 /* A/D 変換器入力選択(ADCCON0.CHS) */
#define ADC_IN0  0b010010  /* IN0(RC2) */
#define ADC_IN1  0b010011  /* IN1(RC3) */
#define ADC_IN2  0b010100  /* IN2(RC4) */
#define ADC_IN3  0b010101  /* IN3(RC5) */
#ifdef MONITOR_HW
#define ADC_CHANNELS    {ADC_IN0, ADC_IN1}
#else
#define ADC_CHANNELS    {ADC_IN0, ADC_IN1,
ADC_IN2, ADC_IN3}
#endif
#define CHANNEL_OFFSET  4   /* 内部レジスタアドレス/2
のオフセット */
#define ADC_MASK        0x03ff /* 変換結果の上位 12 ビ
ットを取り出す */
/* ポート C の IN0（RC2）の位置 */
#define IN0_POSITION    _PORTC_RC2_POSITION
#define NEXT_CHANNEL(x) x >>= 1 /* 1 ビットシフトして
再代入する */
#define TestIN(x)   (__bit) x   /* 最下位ビットを取り
出す */
/* ファームウエアレジスタ初期値 */
#define A_INIT_STATUS  (Astatus_t) {0} /* 仕様書参
照 */
#define A_INIT_COMMAND (Acommand_t) {0} /* 仕様書参
照 */
#define __VERSATILE_MODULE
#endif
```

3.4.2汎用測定機能モジュール

汎用測定機能モジュール versatile_module.c では、初期化、応用層インターフェース、コマンド解釈、割り込み処理、バックグランド電圧測定からなります。

versatile_module.c

```
/* vertatile_module.c PICCOLO チップ 汎用測定モジュー
ル
 初版： 2021/3/21 Chuji
 最新版：2021/6/10
*/
#include <xc.h>
#include <stdbool.h>
#include "include/versatile.h"
/* 汎用測定専用データ */
static Acommand_t a_command, a_command_buf; /* 受
信したコマンド */
static versatile_status_t a_status; /* 動作管理用デ
ータ */
static dataRegister Adata_register[NUM_CHANNELS];
/* 測定値 */
```

```
static versatile_t status_reg[NUM_CHANNELS]; /* 内部形式のコマンドとステータス */
const unsigned char adc_channel[NUM_CHANNELS] = ADC_CHANNELS; /* ADC 入力選択データ */
/* 汎用測定用のハードウエアと内部変数の初期化 */
/* 割り込み禁止状態で呼ばれる */
void init_versatile(void){
  /* SFR の初期設定 */
  /* ポートＣの設定 */
  PORTC = MY_PORTC;     /* 出力レベル初期値（仮） */
  TRISC = MY_TRISC;     /* ポートＣは全て入力 */
  ANSELC = MY_ANSELC;   /* ポートはすべてデジタル入力 */
  WPUC = MY_WPUC;       /* デジタル入力はプルアップする */
  INLVLC = MY_INLVLC;   /* デジタル入力は CMOS レベル */
  ODCONC = MY_ODCONC;   /* RC0/1 はオープンドレイン出力 */
  SLRCONC = MY_SLRCONC;/* RC0/1 は最大スルーレート */
  /* NCO を定周期割り込み源として使う設定 */
  NCO1CON = MY_NCO1CON;    /* パルス出力として使う */
  NCO1CLK = MY_NCO1CLK;    /* 31.25kHz クロック */
  NCO1INCU = MY_NCO1INCU; /* 逐次可算値 */
  NCO1INCH = MY_NCO1INCH; /*   250μs 周期で割り込むには */
  NCO1INCL = MY_NCO1INCL; /*   0x40000 に設定する */
  /* 逐次可算値の最下位バイトが最後に設定されなければならない */
  /* 基準電圧源の設定 */
  FVRCON = MY_FVRCON;      /* A/D 変換器に 2.048V を供給する */
  /*  A/D 変換器の設定*/
  ADCON0 = MY_ADCON0;     /* A/D 変換器動作、初期入力は GND */
  ADCON1 = MY_ADCON1;     /* 変換速度 2μs/ビット、基準電圧 2.048V */
  ADACT = MY_ADACT;       /* 自動トリガを禁止 */
  /* 内部変数の初期化 */
  a_command = A_INIT_COMMAND;
  a_command_buf = A_INIT_COMMAND;
  a_status = INIT_VERSA_STATUS;
  for (unsigned char i = 0; i <NUM_CHANNELS; i++){
    status_reg[i] = INIT_VERSATILE;
    Adata_register[i] = initData;
  }
}
/* 定周期割り込み処理 */
void event_isr(void){  /* 割込みではイベント計数のみを行う */
  unsigned char x = PORTC >> IN0_POSITION; /* 入力ポートの取り込み */
  for (unsigned char i = 0; i < NUM_CHANNELS; i++){
    versatile_t si = status_reg[i];
    if (testFlag(si.event_mode)){ /* イベント計数 */
      if (!testFlag(si.not_first)) setFlag(status_reg[i].not_first);
          /* 一回目は処理をスキップ */
      else {  /* 二回目以降はイベントを検出する */
        if ((!testFlag(si.previous) && TestIN(x)) || /* 立ち上がり検出または */
          (testFlag(si.bothEdge) && (testFlag(si.previous) != TestIN(x)))){
                                /* 立ち下がり・立ち上がり両方検出 */
          if (Adata_register[i] == MaxData){ /* オーバーフロー検出 */
            setFlag(status_reg[i].overflow);
            Adata_register[i] = initData;
          } else Adata_register[i]++;
          setFlag(status_reg[i].ready); /* カウンタが進んだら Ready にする */
        }
      }
      status_reg[i].previous = TestIN(x);  /* 今回の入力を保存 */
    }
```

```
    NEXT_CHANNEL(x); /* 次のチャンネルのポートを最下位ビットにする */
  }
#ifdef FORCE_BP
  DUMMY_STATEMENT();
#endif
}
/* A-Status レジスタの読み取り */
statusRegister read_a_status(void){
  statusRegister s = initStatus;
  /* チャンネル毎のステータスを A-Status レジスタ形式に再構成する */
  for (int i = NUM_CHANNELS - 1; i >= 0; i--){
    s |= status_reg[i].stat;
    if (i != 0) s <<= CHANNEL_LEN;
  }
  return s;
}
/* A-Command レジスタへの書き込み */
void set_a_command(commandRegister cmd){
  a_command_buf = (Acommand_t) cmd;
  setFlag(a_status.new_command);
}
/* 汎用測定機能の測定結果読み取り */
dataRegister read_a_data(unsigned char channel){ /* channel = 0 - 3 */
  clearFlag(status_reg[channel].ready);
  return Adata_register[channel];
}
/* 汎用測定コマンド解釈用マクロ */
#define Parse(i, AEm, ENm, ANSCn, WPUCn) \
    if ((new_command.AEm != a_command.AEm) || (new_command.ENm != a_command.ENm)){ /* 変更あり */ \
      if (testFlag(new_command.AEm)){ /* イベント計数モードになった */ \
        clearFlag(status_reg[i].voltage_mode);  /* 電圧測定モードを消す */  \
        setFlag(status_reg[i].event_mode);   /* モード設定 */       \
        status_reg[i].bothEdge = new_command.ENm; /* 計数エッジ */  \
        clearFlag(ANSELCbits.ANSCn); /* ポートはデジタル入力 */  \
        setFlag(WPUCbits.WPUCn); /* ポートをプルアップ */ \
        clearFlag(status_reg[i].overflow);    \
        clearFlag(status_reg[i].ready);       \
        clearFlag(status_reg[i].not_first);  /* 一回目 */       \
        disableInterrupt(); \
          Adata_register[i] = initData; /* 測定値をクリア */  \
        enableInterrupt(); \
      } else if (testFlag(new_command.ENm)) { /* 電圧測定モード */ \
        clearFlag(status_reg[i].event_mode);   /* イベントモードを消す */       \
        setFlag(status_reg[i].voltage_mode);  /* 測定モード設定 */  \
        setFlag(ANSELCbits.ANSCn); /* ポートアナログ入力 */  \
        clearFlag(WPUCbits.WPUCn); /* ポートはプルアップしない */ \
        clearFlag(status_reg[i].overflow);    \
        clearFlag(status_reg[i].ready);       \
        clearFlag(status_reg[i].not_first);  /* 一回目 */       \
        disableInterrupt(); \
          Adata_register[i] = initData; /* 測定値をクリア */  \
        enableInterrupt(); \
      } else { /* 測定停止（測定値とフラグは前のまま） */ \
        clearFlag(status_reg[i].event_mode);   /* イベントモードを消す */       \
        clearFlag(status_reg[i].voltage_mode);  /* 電圧測定モードを消す */  \
        clearFlag(status_reg[i].ready);       \
        clearFlag(ANSELCbits.ANSCn); /* ポートはデジタル入力 */  \
```

```
        setFlag(WPUCbits.WPUCn); /* ポートをプルアッ
プ */ \
      } \
    }   /* マクロ終了 */
/* 汎用測定用コマンドの解釈 */
void versatile_command(void){
  /* コマンド解釈中に新しいコマンドを受け取っても解釈を続行
し、終了後に再解釈する */
  Acommand_t new_command;
  disableInterrupt();
    if (testFlag(a_status.new_command)){
      clearFlag(a_status.new_command);
      new_command.whole_byte =
a_command_buf.whole_byte;
      setFlag(a_status.parsing);
    } else clearFlag(a_status.parsing);
 enableInterrupt();
  if (testFlag(a_status.parsing)){
    Parse(0, AE0, EN0, ANSC2, WPUC2)
    Parse(1, AE1, EN1, ANSC3, WPUC3)
#ifndef MONITOR_HW
    Parse(2, AE2, EN2, ANSC4, WPUC4)
    Parse(3, AE3, EN3, ANSC5, WPUC5)
#endif
    a_command = new_command;   /* 現在使用中のコマン
ド */
  }
}
/* バックグランド処理で電圧を測定する */
void voltage_measurement(void){
  if (testFlag(a_status.converting)){ /* AD 変換器が
動作中なら終了を確認する */
    if (!testFlag(ADCON0bits.GOnDONE)){ /* 変換が終
了していたら */
      disableInterrupt();
        Adata_register[a_status.current] = ADRES &
ADC_MASK; /* 変換結果を取り出して */

setFlag(status_reg[a_status.current].ready); /*
Ready ビットを立てる */
      enableInterrupt();
      clearFlag(a_status.converting); /* 変換手順の
終了 */
      /* 次のチャンネルを選ぶ */
      if(++a_status.current >= NUM_CHANNELS)
a_status.current = FIRST_CHANNEL;
    }
  /* 変換動作中でなければ */
  } else {
    /* 次のチャンネルが電圧測定モードだったら、入力を設定
して AD 変換を始める */
    if
(testFlag(status_reg[a_status.current].voltage_mod
e)){
      ADCON0bits.CHS =
adc_channel[a_status.current]; /* 入力を設定する */
      setFlag(ADCON0bits.GOnDONE);
      setFlag(a_status.converting);
    /* そのチャンネルが電圧測定モードでなければ、次のチャ
ンネルに */
    } else {
      if(++a_status.current >= NUM_CHANNELS)
a_status.current = FIRST_CHANNEL;
    }
  }
}
```

初期化 `init_versatile()` では SFR の初期化に続き、内部データを初期化しています。

ステータスなどの管理情報は、入力チャンネル毎の変数 `status_reg[]` に分けて管理しています。応用層からステータスを求められたときは、内部形式のデータから A-Status レジスタを再構成します。他のインターフェースは時間測定機能と同様です。

コマンド解釈では、新しいコマンドを受け付けたら、入力チャンネル毎にコマンドを分解して、`status_reg[]` に変換します。マクロ `Parse()` を使ったのは、ビットフィールドの指定を簡単にするためで、プログラム量は増えますが妥協しました。

定周期割り込み処理（イベント計数）

定周期割り込み処理 `event_isr()` では、入力チャンネル毎に処理を行います。イベント計数モードに設定されているときは、該当入力ポートを読み取って前回値と比較して、立ち上がり（あるいは立ち上がり・立ち下がり）があったときはカウンタを進めます。オーバーフローがあったときは、ステータスに即時反映します。

バックグランド処理（電圧測定）

電圧測定はバックグランド処理として実行しますが、A/D 変換処理に時間がかかるので、2 つの状態を設けます。なお、ソースファイルでは、以下の説明と逆の順番で記述しています。

1. A/D 変換中でなければ、次のチャンネルを調べ、電圧測定モードのときは A/D 変換器の入力に接続して、変換を開始する。この時点で、いったん親ルーチンに戻り、コマンド解釈に時間を与える。電圧測定モードでないときも、いったん戻る。
2. A/D 変換中だったら、変換が終わったか確認する。終わっていたら、結果を保存し、ステータスの Ready ビットをセットする。このとき、測定値とステータスを保存する間は、割り込みを禁止しておき、両者が矛盾しないようにする。

2.1 I2C 通信機能

I2C 通信はもっとも即応性を求められる機能です。

2.1.1 インクルードファイル

インクルードファイル i2c.h では、MSSP ハードウエアの初期化データ、通信でつかうデータ型やマクロ、ステートマシンの定義を行っています。

i2c.h

```
/* i2c.h PICCOLO チップ I2C 通信モジュール定義
 初版： 2021/3/21 Chuji
 最新版：2021/6/10
*/
#ifndef __I2C_COMMUNICATION_MODULE
#include "piccolo.h"
#include "timers.h"
#include "versatile.h"
/* 関数プロトタイプ宣言 */
/* 初期化 */
```

```
void init_i2c(void);
/* 割り込み処理 */
void i2c_isr(void);
void i2c_collision(void);
/* 16 ビットデータの分解 */
#define getHigh(x)  (unsigned char) ((x & 0xff00)
>> 8)
#define getLow(x)   (unsigned char) (x & 0xff)
/* 奇数データの検出 */
#define isOdd(x)    (x & 0x01)
/* 専用データの構造とサイズ */
#define I2C_BUFSIZE (unsigned char) 2   /* 送受信バ
ッファの大きさ。これ以上のサイズは対応しない */
#define BUFFER_HEAD (unsigned char) 0   /* 送受信バ
ッファの先頭インデックス */
#define BUFFER_DAT  (unsigned char) 1   /* 次のイン
デックス */
#define DUMMY_DAT   (unsigned char) 0xff   /* ダミ
ーデータ */
#define NULL_DATA   (unsigned char) 0   /* データが
ない時のデータ数 */
#define EMPTY_DATA  (unsigned char) 0   /* 送受信バ
ッファが空であることを表す */
/* ステートマシンの状態 */
#define I2C_IDLE        (unsigned char) 1
#define I2C_ACCEPTING   (unsigned char) 2
#define I2C_TRANSMITTING     (unsigned char) 4
#define I2C_NUM_STATES  (unsigned char) 3  /* 状態
の総数 */
#define RESTART_MASK    (I2C_IDLE | I2C_ACCEPTING)
#define isI2CRestartable(x) (x & RESTART_MASK)
/* 応用層の状態 */
#define APL_IDLE        I2C_IDLE
#define APL_RECEIVING   I2C_ACCEPTING
#define APL_SENDING    I2C_TRANSMITTING
#define APL_NUM_STATES  I2C_NUM_STATES
#define APL_STOP        1 /* 応用層が受信を終了した
*/
#define APL_CONTINUE    0 /* 応用層が次のデータを求め
ている */
/* SFR の初期値 */
#define MY_SSP1STAT 0b10000000 /* 通信速度 100kHz */
#define MY_SSP1CON1 0b00110110 /* 7 ビットアドレス、
スレーブ動作 */
#ifdef SAFE_I2C
#define MY_SSP1CON2 0b00000001 /* 送受信時両方でクロ
ックストレッチ */
#else
#define MY_SSP1CON2 0b00000000 /* クロックストレッチ
は送信時のみ */
#endif
#define MY_SSP1CON3 0b01000000 /* 正常受信時は ACK を
返す */
#define MY_SSP1MSK  0b11111110 /* アドレスが完全一致
時のみ処理 */
#define MY_SSP1ADD  0b01110000/* スレーブアドレス
0x38 */
#ifdef MY_PIE3
#undef MY_PIE3
#endif
#define MY_PIE3     0b00000001 /* MSSP 割り込みを許可
*/
/* 送信時の遅延量 */
#ifdef FAST_CPU
#define TRANSMIT_DELAY  8
#else
#ifdef MID_CPU
#define TRANSMIT_DELAY  2
#else
#define TRANSMIT_DELAY  1
#endif
#endif
/* SDA/SCL をポートにマップする（入力・出力をそれぞれ指定
する必要がある）*/
#define PPS_FOR_SCL 0x10
#define PPS_FOR_SDA 0x11
#define PPS_SCL     0x15
#define PPS_SDA     0x16
#define __I2C_COMMUNICATION_MODULE
#endif
```

2.1.2 I2C 通信機能モジュール

ソースファイルは、応用層プログラム、MSSP ハードウエアの初期化、データリンク層プログラム（割り込み処理）の順番で記述しています。

i2c_module.c

```
/* i2c_module.c PICCOLO チップ  I2C 通信モジュール
 初版： 2021/3/21 Chuji
 最新版：2021/6/10
*/
#include <xc.h>
#include "include/i2c.h"
/* I2C 通信専用データ */
/* 応用層が使うデータ */
static unsigned char APL_state = APL_IDLE;
static unsigned char sendBuffer[I2C_BUFSIZE];   /*
送信バッファ */
static unsigned char sendData = EMPTY_DATA;    /*
送信バッファ内のデータバイト数 */
static unsigned char sendPointer = BUFFER_HEAD; /*
送信バッファ中の次に送るデータ */
static unsigned char receiveData = EMPTY_DATA;  /*
受信バッファ内のデータバイト数 */
/* データリンク層が使うデータ */
static unsigned char i2cState = I2C_IDLE; /* ステー
トマシンの状態 */
static unsigned char recData;          /* 受信データ
の読み取りバッファ */
static unsigned char delay_count;   /* 送信時の遅延
を発生させるためのカウンタ */
/* 応用層関数/マクロ（このモジュール内でのみ使用する） */
#define setBuffer8(sdata){ /* ローカル関数  8 ビットデ
ータを送信バッファに入れる */ \
  sendBuffer[BUFFER_HEAD] = sdata;    \
  sendData = STATUS_LEN;  \
  sendPointer = BUFFER_HEAD;  \
}
/* ローカル関数   リトルエンディアン 16 ビットデータを送信バ
ッファに入れる*/
static void setBuffer16(unsigned short sdata){
  sendBuffer[BUFFER_HEAD] = getHigh(sdata);
  sendBuffer[BUFFER_DAT]  = getLow(sdata);
  sendData = DATA_LEN;
  sendPointer = BUFFER_HEAD;
}
/* 受信開始 */
#define apl_start_receiving() { \
  APL_state = APL_RECEIVING; \
  receiveData = BUFFER_HEAD; \
}
/* 受信終了 */
#define apl_stop_receiving() {  \
  APL_state = APL_IDLE;   \
  receiveData = BUFFER_HEAD;    \
}
/* 1 バイト受信 */
static void apl_receive(unsigned char dat) {
  static unsigned char reg_index; /* 書き込みたいコ
マンドレジスタ番号 */
  if (APL_state == APL_RECEIVING){
    if (receiveData == BUFFER_HEAD){ /* 1 バイト目の
受信 */
      reg_index = dat; /* 最初の受信データ（レジスタ番
号）を保存する */
      receiveData = BUFFER_DAT; /* 次に受信するデータ
はコマンド */
      if ((dat == INDEX_T_COMMAND) || (dat ==
INDEX_A_COMMAND))
        return;  /* コマンドレジスタの場合は、次のデータ
を待つ */
      if (dat == INDEX_T_STATUS){
        setBuffer8(read_t_status()); /* T-Status レ
ジスタの読み取り */
      } else if (dat == INDEX_A_STATUS){
        setBuffer8(read_a_status()); /* A-Status レ
ジスタの読み取り */
      } else if (dat == INDEX_WIDTH) {
```

```
        setBuffer16(read_width()); /* Width レジスタの読み取り */
      } else if (dat == INDEX_COUNT) {
        setBuffer16(read_count()); /* Count レジスタの読み取り */
      } else { /* Data0 - Data3 レジスタの読み取り */
        unsigned char areg = getVdata_index(dat); /* 汎用入力データレジスタ番号に変換 */
        if (isVindex(areg) & !isOdd(dat)){  /* 該当するレジスタ番号が存在すればデータを返す */
          setBuffer16(read_a_data(areg));
        } else {
          /* 該当するレジスタ番号がないときはデータを無視する */
          receiveData = BUFFER_HEAD;
          sendData = EMPTY_DATA;
          APL_state = APL_IDLE;
        }
        return; /* レジスタの読み出しが終了したのでもう受信データを受け付けない */
      }
    } else if (receiveData == BUFFER_DAT){ /* 2 バイト目の受信 */
      if (reg_index == INDEX_T_COMMAND){
        set_t_command(dat);  /* T-Command レジスタへの書き込み */
      } else if (reg_index == INDEX_A_COMMAND){
        set_a_command(dat);  /* A-Command レジスタへの書き込み */
      }
      receiveData = BUFFER_HEAD;
      APL_state = APL_IDLE; /* 受信動作終了 */
      return;
    }
    return;
  /* 3 バイト以上受信しても無視する */
  }
};
/* 送信開始 */
#define apl_start_sending(){    \
  APL_state = APL_SENDING;    \
}
/* 送信終了 */
#define apl_stop_sending() {    \
  APL_state = APL_IDLE;    \
  sendData  =  EMPTY_DATA; \
  sendPointer = BUFFER_HEAD;   \
}
/* 1 バイト送信 */
static unsigned char apl_send(void){
  if (APL_state == APL_SENDING){
    if (sendData-- > EMPTY_DATA){
      /* バッファにデータが残っていれば送信する */
      return(sendBuffer[sendPointer++]);
    } else {
      /* いつまででも送信できるようにする--- sendData は unsigned */
      sendData = EMPTY_DATA;
      return (DUMMY_DAT); /* 残っていなければダミーデータを送信する */
    }
  }
  return (DUMMY_DAT); /* 値を要求されているので、ダミーデータを返す */
}
/* I2C 通信で使う SFR の初期化 */
/* 割り込み禁止状態で呼ばれる */
void init_i2c(void){
  /* SFR の初期設定 */
  SSP1STAT = MY_SSP1STAT;  /* 通信速度 100kHz */
  SSP1CON1 = MY_SSP1CON1;  /* 7 ビットアドレス、スレーブ動作 */
  SSP1CON2 = MY_SSP1CON2;  /* クロックストレッチは送信時のみ */
  SSP1CON3 = MY_SSP1CON3;  /* 正常受信時は ACK を返す */
  SSP1MSK  = MY_SSP1MSK;   /* アドレスが完全一致時のみ処理 */
  SSP1ADD  = MY_SSP1ADD;   /* スレーブアドレス 0x38 */
  /* SCL/SDA の入出力ポートを RC0/RC1 に指定する */
```

```
  UnlockPPS();
  SSP1CLKPPS = PPS_FOR_SCL; /* SCL 入力を RC0 ポートから受ける */
  SSP1DATPPS = PPS_FOR_SDA; /* SDA 入力を RC1 ポートから受ける */
  RC0PPS = PPS_SCL;        /* RC0 ポートに SCL を出力 */
  RC1PPS = PPS_SDA;        /* RC1 ポートに SDA を出力 */
  LockPPS();
}
#define transmit_a_byte()    {    \
  SSP1BUF = apl_send();   \
  for (delay_count =1; delay_count <=TRANSMIT_DELAY; delay_count++) ; \
}
/* I2C 割り込み処理 ＝ データリンク層の機能 */
void i2c_isr(void){
#ifdef MONITOR_START /* モニター信号を発生する */
  if(testFlag(SSP1STATbits.D_nA)) setFlag(LATCbits.LATC5);
  else clearFlag(LATCbits.LATC5);
#endif
  /* ステートごとの状態遷移 */
  if (isI2CRestartable(i2cState)){ /* IDLE/ACCEPTING では Start を確認する */
    if (testFlag(SSP1STATbits.BF)){ /* 受信している */
      recData = SSP1BUF;  /* 受信したのは自分のアドレスかデータ */
      if (!testFlag(SSP1STATbits.D_nA)){ /* アドレスを受信した(START) */
        if (testFlag(SSP1STATbits.R_nW)) { /* 読み出し要求だったら */
          i2cState = I2C_TRANSMITTING;  /* 状態遷移#2, #7 */
          apl_start_sending();
          transmit_a_byte();
#ifndef SAFE_I2C
          setFlag(SSP1CON1bits.CKP); /* クロックストレッチを解除する */
#endif
        } else{ /* 書き込み要求だったら */
          i2cState = I2C_ACCEPTING;   /* 状態遷移#1, #6 */
          apl_start_receiving();     /* 受信バッファの準備 */
        }
      } else { /* 受信したのはデータ */
        if (i2cState == I2C_ACCEPTING)
          apl_receive(recData); /* 受信する（状態遷移#8） */
        /* IDLE 状態でデータを受信することはない（状態遷移#5） */
      }
    } else if (testFlag(SSP1STATbits.P)){ /* STOP を検出した（状態遷移#3, #9） */
      if (i2cState == I2C_ACCEPTING){
        apl_stop_receiving();
        i2cState = I2C_IDLE;
      }
    } else if (testFlag(SSP1CON1bits.SSPOV)){ /* 受信オーバフロー（状態遷移#4, #10） */
        clearFlag(SSP1CON1bits.SSPOV);
        i2cState = I2C_IDLE;
        if (i2cState == I2C_ACCEPTING) apl_stop_receiving(); /* 状態遷移$10 */
    }
  } else {    /* TRANSMITTING */
    if (!testFlag(SSP1STATbits.BF)){ /* 送信が正常に終了したとき */
      if (!testFlag(SSP1CON2bits.ACKSTAT)){
        /* マスターが次のデータを要求している（状態遷移#11） */
        transmit_a_byte();
#ifndef SAFE_I2C
        setFlag(SSP1CON1bits.CKP); /* クロックストレッチを解除する */
#endif
      } else {  /* マスターが次のデータを要求していない（状態遷移#12） */
```

```
        i2cState = I2C_IDLE;
        apl_stop_sending();
      }
    } else if (testFlag(SSP1STATbits.P)){ /* STOP
を検出した（状態遷移#13） */
      apl_stop_sending();
      i2cState = I2C_IDLE;
    }
  }
#ifdef SAFE_I2C
  if(!testFlag(SSP1CON1bits.CKP))
    setFlag(SSP1CON1bits.CKP); /* クロックストレッチ
を解除する */
#endif
}
void i2c_collision(void){ /* I2C 送信中のバス衝突検出
（状態遷移#14） */
  i2cState = I2C_IDLE;
}
```

応用層

応用層は6つの『関数』で定義されていますが、送受信の開始と終了は、処理が単純なのでマクロとして記述しました。実質的な『関数』は2つで、1バイトずつの送受信データを処理します。

機能限定の影響を受けているのは、受信処理部です。受信1バイト目が読み出し専用レジスタの場合は、すぐに返信データを作り、送信メモリに入れます。書き込み専用レジスタの場合は、次のデータを待ってから各機能に送りつけます。この段階で処理を終了してしまうので、応用層のステートはデータリンク層のステートと一致していない時間帯が発生します。

送信処理では、送信メモリから順番にデータを取り出し、空になっても要求があればダミーデータを返します。この関数は、送信メモリの用意さえあれば何バイトでも送れる構造になっています。

データリンク層

データリンク層は、割り込み処理（ISR）として実装しています。ステートマシンをそのまま実装しているので、詳しい説明は不要だと思います。

2.2 ビルド結果

ここまでに書き終わったファームウエアをビルド（コンパイル&リンク）してみます。コンパイルエラーが出なければ、どのくらいメモリを使うかが分かります。

項目	消費量	メモリ容量	目標消費量
プログラムメモリ	1,622W	8,192W	2,048W 以内
データメモリ	64B	1,024B	96B 以内

メモリ使用量（コーディング終了時）

第一版では、プログラムメモリもデータメモリも、目標である単一バンク内に収まっていました。検証中に出てきた問題を解消した結果、最終的には以下のようになりました。この本に掲載しているのは、最終版です。

項目	消費量	メモリ容量	目標消費量
プログラムメモリ	1,874W	8,192W	2,048W 以内
データメモリ	67B	1,024B	96B 以内

メモリ使用量（最終版）

3. シミュレーションによる検証

前章ですべてのファームウエア作成が（いちおう）終わりました。Raspberry Pi を使った開発プロジェクトでは、Python や C 言語のプログラムは、PC 上で論理をシミュレーション検証してから、実チップに移しました。今回もできるだけ同じ手順を踏みます。しかし MPLAB X IDE は、スタブなどを使ってモジュール毎に検証するのに便利な環境ではありません（SFR への読み書きがプログラムの大きな部分を占めるため、Linux 上でシミュレートするのは、面倒が多い）。そこでファームウエアをすべてコンパイル・リンクしたうえで、各モジュールの機能をひとつずつ動かしていくというアプローチを取りました。

シミュレータの設計者はそれなりに努力しているようですが、初期化時に「enable していないハードウエアの設定をするのはおかしい」というワーニングに出会ったときは驚きました。あらかじめ機能だけ設定しておき、使うときに enable しようとしていたからです。ほかにもワーニングの文が意味不明だったり、ミススペルがあったりしました。

3.1 シミュレーションによる検証の概要

3.1.1ハードウエアのサポート

ベースチップの周辺ハードウエア（使用するもの）のうち、シミュレータ（v5.45）がサポートしているのは以下の範囲です。I2C 通信で使う MSSP 以外は含まれているようです。

周辺ハードウエア	使用するモジュール	サポート
ADC	Versatile （電圧測定）	○
CLC1	Timer （周波数測定）	○
INT	ISR	○
MSSP	I2C	×
NCO	Versatile （イベント計数）	○
OSC	main	○
PORTA	Timer	○
PORTC	Versatile	○
PWM3	Timer （周波数測定）	○
Pull Up	Timer, Versatile	○
TMR0	Timer （周波数測定）	○
TMR1	Timer （パルス幅測定）	○
TMR2	Timer （周波数測定）	○

シミュレータのサポート状況

サポートの方法、というよりハードウエアの動作をシミュレートする方法には 2 種類があります：

1. ハードウエアの動作を回路どおりにシミュレートする
2. SFR に外部から値を与えることで動作を模擬する

このうち方法 2. は、SFR にデータを手動で与える（マニュアル・インジェクション）以外に、値を読み出そうとするたびにファイルなどから与える（ファイル・インジェクション）方法などがあります。ファームウエアの論理は確認できますが、ハードウエアの動作制御を含めた検証はできません。今回はマニュアル・インジェクションを多用しています。

方法 1. はタイマーによる計数動作などを含めて検証できますが、PC の負荷が非常に大きいという問題があります。それに加え、検証が十分行われていない（品質が低い）シミュレーションモデルがありました（本文中で説明しています）。実チップで動作する設定にしても、シミュレーションでは不可解な応答を示すことがあります。とくにクロック回路は、PIC の世代が新しくなるたびに実装が複雑になっており、シミュレーションモデルが追い付いていません。いちおうシミュレーションモデルは正しいと『信じて』検証を進め、どうしても解決できないときは、方法 2. で論理を検証してから実チップで試すというアプローチをとりました（この本の中では【未検証項目 n】と注記している）。

3.1.2 ブレークポイント

プログラムの実行状況を監視・変更するため、ブレークポイントを設定して、プログラムの実行を停止することができます。ブレークポイントはソースプログラムの行、マシンコードのアドレス（C 言語からは使えない）、メモリへのアクセス、およびそれらの組み合わせに設定できます。もっとも PICCOLO チップではメモリへアクセスするプログラムが限定されているので、実際に使うのは行ブレークポイントだけです。

具体的な操作方法は、別の資料（ユーザーガイドなど）を参照してください。

3.2 初期化

main()関数の initialize_all()の次の行にブレークポイントを設定して実行すると、すべての初期化が終わった時点で停止します。

ここで *SFR* 一覧を見ると、新たに設定しなおされた *SFR* の設定値が赤字で表示されています。ここまで実装設計として設定内容をじっくり検討してきたので、この時点で設定値が正しいか調べても、検証効果はあまりありません（ミスタイプしか見つからない）。ハードウエアの動作が期待どおりでない（そうならないのが望ましい）ことが分かった時点まで先延ばしにします。

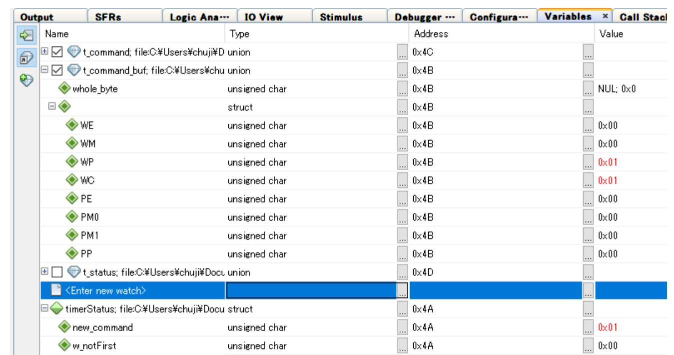
一部のハードウエアのステータスを示すレジスタがすべて 0 を示しており、明らかに動作をシミュレートしていないものがあることが分かりました。これらは要注意です。

3.3 コマンド解釈

次にコマンド解釈部の論理を検証します。いちいちコマンドを与えて結果を調べる（マニュアル・インジェクション）ので、手間がかかるし、自動化もできそうにありません。忍耐強く、繰り返しました。

timer_command()の最初の行（関数名ではなく、その次の if testFlag(…) で始まる行）にブレークポイントを設定すると、バックグランド処理で何度も呼ばれることが確認できます。その次の行（clearFlag(…)）にもブレークポイントを設定しても、その行には到達しません。新しいコマンドを受け取ったときだけ実行されるからです。

変数表示（*Variables*）を呼び出すと（右上の図を見てください）、timerStatus.new_command はセットされていません（表示は 0x00）。このビットをセット（ダブルクリックして、変数値を 0x01 に変更する）してからステップ実行すると、次の行に到達することが分かります。さらにステップ実行すると、上のビットがクリアされ、その次の行に移ります。さらに実行を続けても、新しいコマンド t_command_buf の内容が現在の設定と同じだから、何も起きません。



Name	Type	Address	Value
t_command; file:C:¥Users¥chuji¥D	union	0x4C	
t_command_buf; file:C:¥Users¥chu	union	0x4B	
whole_byte	unsigned char	0x4B	NUL; 0x0
	struct	0x4B	
WE	unsigned char	0x4B	0x00
WM	unsigned char	0x4B	0x00
WP	unsigned char	0x4B	0x01
WC	unsigned char	0x4B	0x01
PE	unsigned char	0x4B	0x00
PM0	unsigned char	0x4B	0x00
PM1	unsigned char	0x4B	0x00
PP	unsigned char	0x4B	0x00
t_status; file:C:¥Users¥chuji¥Docu	union	0x4D	
<Enter new watch>			
timerStatus; file:C:¥Users¥chuji¥Docu	struct	0x4A	
new_command	unsigned char	0x4A	0x01
w_notFirst	unsigned char	0x4A	0x00

コマンドの WP と WC をセットし、コマンド解釈部に与える

コマンドを与えるために t_command_buf の内容を変更するとき、ビットフィールド表示にすれば、間違いが少なくなります。上の図では WP ビットと WC ビットをセットし（赤字で表示されている）、timerStatus.new_command もセットしています。

ここまででコマンド解釈部の検証手順ができ上りました。

1. バックグランド処理の最初にブレークポイントを設定して、プログラムの実行を止める。
2. 新しいコマンドを command_buf に書き込み、new_command をセットしてから、実行を再開する。
3. 次に実行が止まったとき、SFR や変数が設計どおりに設定されているか確認する。

という手順です。コマンド解釈部では、コマンド→SFR／変数という変換をプログラムで記述しています。その論理を検証するため、すべての変換が正しく（設定どおりに）行われていることを確認しなければなりません。

3.3.1 時間測定機能コマンド解釈の検証

まず、timer_command()の検証を行います。コマンドの t_command_buf.WP と t_command_buf.WC、それに timerStatus.new_command をセットして実行すると、次のブレークポイントでは、T1CON.CKPS と T1GCON.GPOL の設定が変わり、クロック分周比とゲート極性がコマンドどおりに選択されたことが分かります。

WP と WC をクリアし、new_command をセットして実行すれば、設定が元に戻ります。これで WP と WC のビットフィールドの解釈が検証できました。二つのビットの処理は独立しているので、同時に検証しても構いません。

Address	Name	Hex	Decimal	Binary
0192	SSP1CON3	0x40	64	01000000
020C	TMR1	0x0000	0	00000000
020C	TMR1L	0x00	0	00000000
020D	TMR1H	0x00	0	00000000
020E	T1CON	0x22	34	00100010
020F	T1GCON	0xC0	192	11000000
0210	T1GATE	0x00	0	00000000
0211	T1CLK	0x05	5	00000101
028C	T2TMR	0x00	0	00000000
028D	T2PR	0xF7	247	11110111
028E	T2CON	0xF0	240	11110000
028F	T2HLT	0x00	0	00000000

WP=1, WC = 1↓　↑WP=0, WC=0

Address	Name	Hex	Decimal	Binary
0192	SSP1CON3	0x40	64	01000000
020C	TMR1	0x0000	0	00000000
020C	TMR1L	0x00	0	00000000
020D	TMR1H	0x00	0	00000000
020E	T1CON	0x02	2	00000010
020F	T1GCON	0x84	132	10000100
0210	T1GATE	0x00	0	00000000
0211	T1CLK	0x05	5	00000101
028C	T2TMR	0x00	0	00000000
028D	T2PR	0xF7	247	11110111
028E	T2CON	0xF0	240	11110000
028F	T2HLT	0x00	0	00000000

同じ手順で、**実装設計 III（ファームウエア）**の節で設計した SFR 設定値になっているか確認していきます。パルス幅測定機能の検証結果を下の表に示します。変数 timerStatus の w_notFirst ビットと w_overflow ビット、さらに測定値を入れる width_counter も初期化されていることも確認します。

変数と SFR		入力値と処理結果			
入力	t_command_buf.WE	0	0	1	1
	t_command_buf.WM	0	1	0	1
結果	T1CON.ON	0	0	1	1
	PIE0.INTE	0	0	1	0
	PIE5.TMR1GIE	0	0	0	1
	PIE4.TMR1IE	0	0	1	1

パルス幅測定機能のコマンド解釈

周波数測定機能の方は、設定する SFR が多いので、ちょっと面倒です。この本のページ数の制約から、個々のコマンドと確認箇所を列挙できませんが、「周波数測定機能の SFR 設定」という表にある、すべての設定を検証することが肝心です。

3.3.2 汎用測定機能コマンド解釈の検証

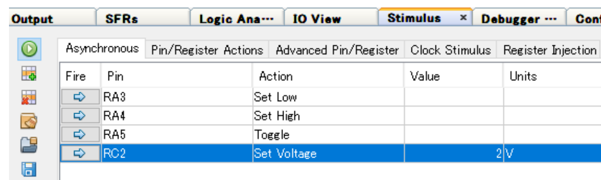
汎用機能コマンド解釈 versatile_command()も同じように検証します。入力チャンネル（IN0～IN3）ごとに指定を変え、それぞれ SFR（ANSELC と WPUC）の正しいチャンネル位置が設定されることを確認します。また、チャンネル毎に用意した、コマンド・ステータスの構造体 status_reg[]の各ビットフィールドも調べます。

変更の順番によっては、イベント計数モードが解除されたとき、status_reg[].bothEdge がセットされたまま残ることがあります。次にイベント計数モードになる時に再設定されるので、問題はありません。

これでコマンド解釈機能の検証ができました。これ以降は（実チップを使って I2C 通信経由でコマンドを与えることができるようになるまで）、同じ手順でコマンドを与え、ハードウエアの設定を変えます。そうすることで、（シミュレーションで）ハードウエアを動作させながら検証していきます

3.3.3 電圧測定機能

次に、バックグランド処理として実行される電圧測定機能を検証します。a_command_buf を AE0:EN0=0:1 と設定して解釈させれば、IN0 チャンネル（RC2）をアナログ入力端子として電圧を測定し始めます。*Simulator*→*Stimulus*（刺激）→ *Asynchronous* 表示を開き、RC2 端子に電圧を与える（*Set Voltage*）ように設定します。電圧は V あるいは mV で与え、左端の➡ボタンを押すと、電圧が印加されます（値を変更するたびに➡ボタンを押す）。



IN0（RC2）端子にアナログ電圧を加える

汎用測定機能モジュール（versatile.c）にある関数 voltage_measurement()で、先頭行と変換結果を取り込む行にブレークポイントをかけておきます。

先頭行で（入力チャンネルの選択と変換待ちで）何回かブレークしてから、変換結果の取り込みが行われる個所でブレークします。次のブレークまで進んだら、取り込み結果（データと ready フラグ）が書き換えられていることを確認します。

ここでシミュレーションモデルの最初の問題が発覚しました。フルスケール（出力コードが 0x3f になる入力電圧）を 2.048V に設定しているのに、これが電源電圧 3.3V になっていました！　SFR の設定値を再確認しましたが、誤っていません。実チップで確認することにしました。【未検証項目 1】

入力ポートを変えながら、それぞれのデータとして反映されることを確認しておきます。

3.4 パルス幅測定機能

タイマー#1 のシミュレーションモデルは思ったように動作せず、GATE（RA4）信号がゲート信号として使えませんでした（RA4 入力が T1GCON.GVAL にも、モニター信号 *T1G* にも反映されない）。設定を見直しても間違いが見つからないので、マニュアル・インジェクションで割り込み処理の論理だけを検証しました。【未検証項目 2】

t_command_buf.WE をセットしてコマンドを実行させると、同期モードの測定が開始されます。

3.4.1 SYNC 割り込み（同期モード）

割り込み許可 PIE0.INTE と PIE4.TMR1IE がセットされていることを *SFR* 表示で確認します。この表示画面で割り込みフラグ PIR0 をダブルクリックすると値が変更できます。最下位（PIR0.INTE）をセット（値の欄に 1 を入力する）して、sync_isr() の 1 行目にブレークポイントを設定します。シミュレーションを再開すると、上のブレークポイントに到達するので、割り込みが模擬できました。PIR0 はクリアされています。

この割り込み時点では、一回目の測定（測定値が正しくないかもしれない）なので、フラグの変更と TMR1 の初期化をするだけで、すぐに抜けてしまいます。

次のブレークでは、もう一度 PIR0 の割り込みフラグを立て、TMR1 に適当な 2 バイトデータを入力します。sync_isr()の後のブレークで調べると、測定値（width_counter）には上で与えたデータが複写され、ステータスレジスタの Ready フラグ t_status.RDT がセットされています。TMR1 は 0 に初期化されます。

これで同期モードでのデータ取り込みが確認できました。

3.4.2 GATE 割り込み（非同期モード）

さらに t_command_buf.WM もセットしてコマンドを実行させると、非同期モードの測定が開始されます。割り込み許可 PIE0.INTE はセットされず、その代わり PIE5.TMR1GIE（ゲート信号の最後で割り込み）がセットされています。

SFR 表示画面で PIR5 の最下位に 1 を入力してシミュレーションを再開すると、同じように sync_isr()でブレークします。あとは同期モードと同様に動作を調べます。

3.4.3 オーバーフロー割り込み

測定中に PIE4.TMR1IE をセットして、オーバーフロー割り込みを起こさせます。そのあとで SYNC 割り込みや GATE 割り込みを発生させると、ステータスレジスタのオーバーフローフラグ t_status.OVT がセットされます。

3.5 周波数測定機能

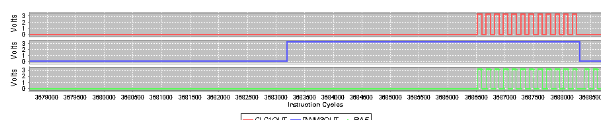
周波数測定では、多くの周辺回路を使っているので、動作確認を慎重に行います。幸いなことに、一部でシミュレーションモデルが使えたので、信号波形も確認できました。最初に t_command_buf を介して 10ms の周波数測定（PE=1, PM1:PM0=11）を設定しておきます。

ちなみに測定時間を 100ms や 1s にすると、ゲート信号が発生しません。タイマー#2 の計数値も増加しません。どうも 31.25kHz のクロックを扱うことができないようです（このクロックのない PIC があるためか？）。【未検証項目 3】

3.5.1 パルスゲート回路

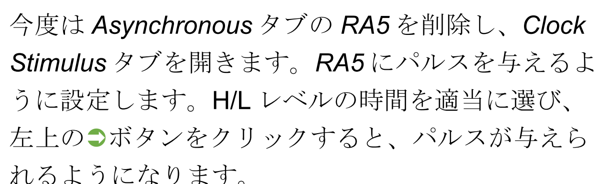
パルスゲート回路の動作を確認するため、シミュレータの *Logic Analyzer* を開き、表示データとして *RA5*（PULSE 入力）、*PWM3OUT*（測定ゲート信号）、*CLC1OUT*（ゲート回路出力＝タイマー#0 入力）を選んでおきます。

Stimulus 表示を開き、*Asynchronous* タブで、*RA5* 入力をトグル（刺激するたびに ON/OFF する）できるようにします。シミュレーションを再開し、適当なタイミングで一時停止させると、*PWM3OUT* が 10ms 毎に変化するのが分かります。*PWM3OUT* が H のとき、*RA5* 入力をトグルします。バックグランド処理にブレークポイントを設けて、シミュレーションを再開します。ブレークするごとに *RA5* をトグルさせると、下のような結果になります。*PWM3OUT* が H のときだけ、入力パルスが CLC1 で切り出されています。このとき *SFRs* を調べると、*RA5* を H にするたびに TMR0 が 1 ずつ増加していることが確認できます。



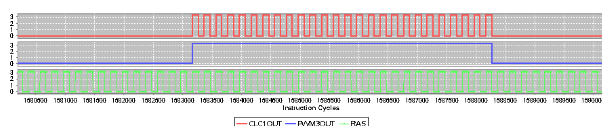
周波数測定（手動入力）の動作波形

今度は *Asynchronous* タブの *RA5* を削除し、*Clock Stimulus* タブを開きます。*RA5* にパルスを与えるように設定します。H/L レベルの時間を適当に選び、左上の➲ボタンをクリックすると、パルスが与えられるようになります。



RA5 ポートへクロック信号を与える

RA5 ポートへ与えられたクロック信号が、PWM3 のゲート信号で切り出され、タイマー#0 に与えられることが分かります。



周波数測定（パルス入力）の動作波形

3.5.2 測定周期割り込み

PWM3OUT が L になったときに調べると、`TMR0` がクリアされています。その前の計数値が変数 `count_register` に転送され、ステータスレジスタ `t_status` の `RDP` ビット（事前にクリアしておく）がセットされていることが分かります。

3.5.3 オーバーフロー割り込み

ちょっと変則的な手法ですが、測定時間を 100ms に設定すると、PWM3 の出力が H または L に固定されてしまいます。H に固定されたときに、パルス入力を与えるようにします。適当な時間を置きながらシミュレーションをブレークさせると、`TMR0` がカウントアップを続けていることが分かります。計数値が 0xffff を越えた瞬間にオーバーフロー割り込みが発生し、`timerStatus.f_overflow` がセットされます。

このままでは測定が終了しないので、*SFR* 表示で `PIR4.TMR2IF` をセットし、強制的に測定値を取り込ませます。一回目の計数では取り込みが行われません。二回目以降は計数値が取り込まれ、ステータスレジスタの `t_status.RDP` と `t_status.OVP` がセットされます。

3.6　イベント計数機能

汎用測定機能のうち、イベント計数は 250μs ごとの割り込みを使って、ファームウエアでカウンタを増やしています。そのための割り込みは数値制御発信器（NCO）で作っているのですが、NCO のシミュレーションモデルが動作してくれず、積算器が増えていきません。31.25kHz のクロックを使っているせいだと推測して、別のクロック源（500kHz, 8MHz, CLC1 出力）を試したのですが、全滅でした。気にはなるのですが、実チップまで検証を持ち越します。【未検証項目 4】

3.6.1 NCO 割り込み

NCO のオーバーフローを、割り込みフラグ `PIR7.NCO1IF` を立てることで模擬します。最初に割り込み許可 `PIE7.NCO1IE` がセットされていることを確認しておきます。

検証する入力チャンネル（ここでは IN0（*RC2*）を使って説明する）にパルスを与えます。周波数測定で *RA5* ポートにパルスを与えたのと同じ方法で、*RC2* にパルスを与える用意をします。間隔は、1 パルスの間にバックグランド処理のブレークが 6 回くらいかかるようにしておきます。

3.6.2 イベント計数

バックグランド処理にブレークポイントを設けてから、入力チャンネル *IN0* を（立ち上がり）イベント計数モード（`AE0=1, EN0=0`）に設定します。

Logic Analyzer で *RC2* を表示させます。入力パルスが与えられているはずです。ブレークごとに `PIR7.NCO1IF` で割り込みを模擬して、次のブレークで測定値 `Adata_register[0]`を調べます。連続する割り込みの間に *RC2* が L→H と変化したときだけ測定値が増えているのが確認できます。`EN0` も 1 に設定すると、H→L 変化でも測定値が増えます。

ブレーク中に測定値 `Adata_register[0]`に 0xfffe を書き込んでから割り込みを模擬すると、2 回目にオーバーフローが起こり、フラグが立ちます。

ここまでの確認を、IN1～IN3 についても行います。入力パルスを与えるポートも変更し、正しいポートの計数が行われ、混信（対応しないポートを数えてしまう）もないことを確認します。そうなるように設計したことは忘れ、第三者の目で検証することが大事です。

3.7　I2C 通信機能

I2C 通信を実行する MSSP 回路のシミュレーションモデルは実装されていないということでした。そこで、どこまでシミュレーションできるか確認しました。その結果、

1. MSSP で割り込み要因（例えば START）を検出したビット（SSP1STAT.S）を立てても、割り込みは発生しない
2. 受信バッファ（SSP1BUF）を読み取ると自動的にクリアされるはずの SSP1STAT.BF は、クリアされない
3. SSP 割り込み（PIR3.SSP1IF）をセットすると、割り込みが起こる

ことが分かりました。つまり、SSP1 レジスタに必要な情報を与えてから、割り込みをかければファームウエアの検証が（ちょっと煩雑な手法ですが）できることが分かりました。

3.7.1 I2C 割り込み処理の検証

応用層の機能は、データリンク層が必要とするときに直接呼ばれる（あるいはデータリンク層プログラムにマクロで埋め込まれている）ので、別々に検証するより、まとめて調べた方が分かりやすいです。

デーリンク層の状態（i2cState）は、I2C_IDLE（1）から始まっています。この状態から、さまざまな受信状態をシミュレートしていきます。

START デリミタ検出

SSPSTAT.S をセットして割り込みをかけます。I2C_IDLE 状態では何も起こりません。

書き込み動作

書き込み動作は一回のバス動作（トランザクション）で終了します。

手順	操作
a	SSP1BUF に 0x70（アドレス 0x38+Write）を書き込み、SSP1STAT レジスタのビットフィールドを DA=0（アドレス）、RW/=0（Write）、BF=1（データ受信）を設定して割り込むと、状態が I2C_RECEIVING（1）に遷移する
b	SSP1STAT.DA をセットし、SSP1BUF に 0x03（A-command レジスタのアドレス）を与えてから割り込むと、応用層がコマンド受信の準備を始める
c	SSP1BUF に A-Command レジスタに与えるデータを設定し、割り込むと、変数 a_command_buf に複写され、a_status.new_command がセットされる
d	実行を再開しバックグランド処理を回せば、汎用入力機能の設定ができている（この部分は既に検証済み）
e	SSP1STAT.BF と SSP1STAT.DA をクリアし、SSP1STAT.P をセットして割り込むと、状態は I2C_IDLE に戻る

書き込み動作の検証手順

上の手順で T-Command レジスタも設定できることを確認します。(b)から後で START/STOP 検出、受信オーバーフローまたはアドレス受信があると、状態が I2C_IDLE に戻り、受信データは（それ以後に受信したデータも）破棄されます。

読み出し動作

読み出し動作は、読み取るレジスタのアドレス書き込みと、読み取ったデータの送信という二回のトランザクションからなります。

手順	操作
a	（書き込みトランザクション） SSP1BUF に 0x70（アドレス 0x38+Write）を書き込み、SSP1STAT レジスタのビットフィールドを DA=0（アドレス）、RW/=0（Write）、BF=1（データ受信）を設定して割り込むと、状態が I2C_RECEIVING（1）に遷移する
b	SSP1STAT.DA と SSP1STAT.BF をセットし、SSP1BUF に 0x00（T-Status レジスタのアドレス）を与えてから割り込むと、応用層が sendBuffer[0]に t_status をコピーしてくれる
c	STOP（SSP1STAT.P=1）割り込みで書き込みを終了する（状態は I2C_IDLE に戻る）
d	（読み出しトランザクション） SSP1BUF に 0x71（アドレス 0x38+Read）を書き込み、SSP1STAT レジスタのビットフィールドを DA=0（アドレス）、RW/=1（Read）、BF=1（データ受信）を設定して割り込むと、状態が I2C_SENDING（2）に遷移する。割り込みの前にプログラムの状態遷移する箇所（i2cState = I2C_SENDING;）にブレークポイントを設けておく
e	上のブレークポイントで送受信バッファを空に（SSP1STAT.BF をクリア）しておく（送信を始められるようにする）。また未受信クロックストレッチを許可（SSP1CON1.CKP をクリア）しておく。
f	実行を再開すると、データリンク層が応用層から送信データを受け取り、SSP1BUF に収納する。次にクロックストレッチを解除（SSP1CON1.CKP をセット）して、送信を開始するので、SSP1BUF の内容（T-Status）を確認する
g	送受信バッファを空に（SSP1STAT.BF をクリア）し、マスターの受信を終了（SSP1CON2.ACKSTAT をセット）して送信完了を模擬する。この状態で割り込みをかけると、状態が I2C_IDLE に戻る

読み出し動作の検証手順

ここまでで T-Status レジスタの読み出しが確認できました。同じ手順で A-Status レジスタ（アドレス 0x01）の読み出しも検証します。こんどは汎用測定機能が配列 status_reg[]として持っているデータから A-Status レジスタを再構成するので、配列のデータを変えながら確認する必要があります（この検証は、Linux など C 言語が走る環境でモジュール評価を行う方が、手間がかかりません。）

手順(g)で、SSP1CON2.ACKSTAT をセットする代わりにクリアし、割り込みをかけると、次のデータ（データがないので 0xff）を SSP1BUF に入れてきます。何回でも繰り返すことを確認したら、手順(g)で読み取りを終了させます。

次に読み出しアドレスを 0x04～0x0e（測定値）にして、読み出し手順を実行します。こんどは 2 バイ

トが読み出せます。測定値変数の値を変えて動作を確認し、1 バイト目は上位バイトが、2 バイト目に下位バイトが送信されることを確認しておきます。

最後に読み出しアドレスを存在しないレジスタ（奇数あるいは 0x10 以上）を指定すると、データが存在しない（常に 0xff が返される）ことを確認すれば、読み出し動作の検証は終わりです。

3.8 割り込み処理時間

MPLAB X IDE には *Stop Watch* という機能があって、ブレークポイント間の時間（サイクル数）を測定することができます。割り込み処理 `all_interrupts()` の最初と最後にブレークポイントを設定して、処理時間を調べておきます。

計数回路は全て（イベント計数は 4 チャンネルとも）使用状態にして、2 回目以降の処理時間を調べました。実装設計時にあげた、目標処理時間（目安）も転記してあります。

割り込み要因	処理サイクル数	換算処理時間	目標処理時間
I2C 通信	46	23μs	10～40μs
SYNC/GATE	43	21.5μs	20μs
周波数ゲート	65	32.5μs	5ms
イベント計数	241	121μs	50μs
オーバーフロー	43-48	21～24μs	50μs

割り込み処理にかかった時間

大部分は目標に近い時間内で処理できていますが、イベント計数だけは長い時間がかかっています。この処理中は、優先度の高い割込み要求があっても待たされてしまうので、他の時間測定の不感時間や通信の応答時間が長くなってしまいます。

これは設計時の盲点でした。優先度の低い処理なので、確実に実行できるように設計したのが仇になってしまいました。`event_isr()` のコードを見直して、配列の参照をポインタに変更しても改善しません。そのくらいの知恵は、C 言語の最適化で処理されてしまっているようです。

そこで、選択肢としては

1. クロックを 4 倍にする（消費電力は増える）
2. 仕様にある応答時間・不感時間を長くする（チップの遅い動作を許容する）

くらいしかありません。1. を選ぶときは、ファイル piccolo.h で `FAST_CPU`（クロックを 2 倍にするときは `MID_CPU`）を#define してコンパイルします。

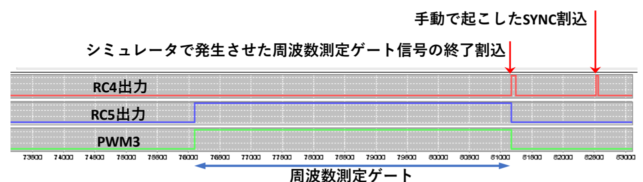
3.9 モニター信号のシミュレーション

ファームウエアを実チップで実行する前に、モニター信号が発生するか確認しておきます。piccolo.h の冒頭にある `#define MONITOR_HW` のコメントを外すと、次のような動作をモニターする信号が I/O ピンに発生します。

1. ISR（割り込みサービス）が処理をしている間、RC4 ポートが H レベルになる。
2. 周波数測定のゲートが開いている間、RC5 ポートが H レベルになる

シミュレーションで時間幅測定（同期モード）と周波数測定（10ms）を行うようにし、RC4、RC5、PWM3 出力（内部信号で、実チップでは見えない）を *Logic Analyzer* に表示します。

`all_iterrupts()` の先頭にブレークポイントを設定しておくと、次の画面のような信号（左半分）が記録できます。バックグランド処理にもブレークポイントを設けると、RC4 にパルスが発生していることがわかります。信号が少し進んだら、手動で SYNC 割り込みを発生（`PIR0.INTF` をセットする）させると、RC4 にもうひとつパルスが現れます。



モニター信号のシミュレーション

これで割り込み処理期間を実チップの外から観測できることが分かりました。どの割り込みを処理しているかは、他の信号との相関で判別できます。`startISR()` を割り込み要因を判定する `if` 文の中に入れることで、その ISR だけを観測するようにもできます。

これで、実チップで割り込み処理の実行時間や、どのくらい忙しいのかを知ることができます。

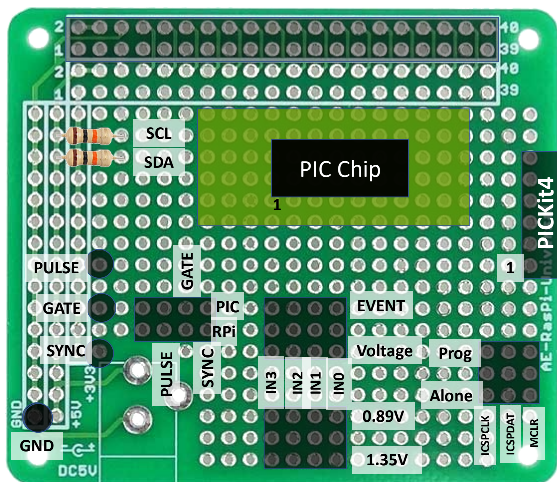
RC5 ポートに周波数ゲート信号を取り出したのは、シミュレーションでは 100ms と 1 秒の信号発生が検証できなかったからです。

4.　実チップ検証

4.1　Raspberry Pi を使った検証環境構築

開発・評価ボードの組み立て

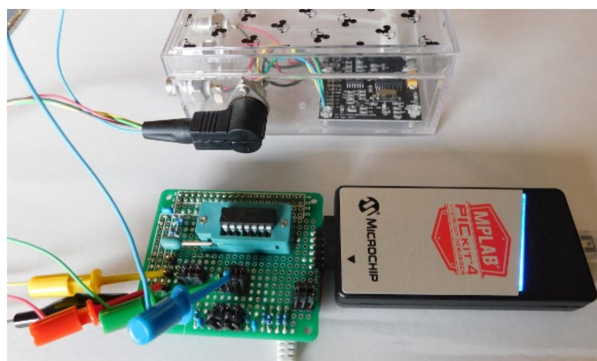
この本の最初の方で設計した、PIC 開発・評価ボードを組み立ててから検証します。信号のチェックピンとジャンパの選択は、下の図のようにしました。



開発・評価ボードのジャンパ選択

簡易型ロジックアナライザ

実チップでの検証では、パルスを与えたり、チップの内部動作をピンに出力させたりするので、ロジックアナライザがあると便利です。USB で PC につなぎ、PC の画面に信号を表示する簡易型ロジックアナライザを用意しました。安価モデルですが、けっこう役に立ちました。



開発・評価ボードに PICKit4 と簡易型ロジアナ（上）をつなぐ

ホストの準備

ホストになる Raspberry Pi には、PC からリモートログインできるようにしておきます。この章に出てくる検証用のプログラムはすべて（シェルスクリプトを除いて）Python で記述しています。作業ディレクトリの下にある include には、use-pigpio.h と use-time.h がコピーしてあるはずです。同じディレクトリに、GPIO のポートを定義する gpio.h を作成します。このファイルは use-pigpio.h がインクルードしてくれます。

gpio.h

```
/* BCM2835 GPIO 定義
   初版:  2014/12/28 Chuji
   最新版:  2021/5/12 PIC 開発ボード用
   注：pigpio デーモンの使用を前提とする
 */
#ifndef __GPIO
/* GPIO ポートの用途は以下のとおり
  GPIO0 (pin27):  ID_SD --- EEPROM は使わない
  GPIO1 (pin28):  ID_SC --- EEPROM は使わない
  GPIO2 (pin03):  I2C SDA *** I2C バス
  GPIO3 (pin05):  I2C SCL *** I2C バス
  GPIO4 (pin07):  PULSE *** 周波数入力信号
  GPIO5 (pin29):
  GPIO6 (pin31):
  GPIO7 (pin26):
  GPIO8 (pin24):
  GPIO9 (pin21):
  GPIO10(pin19):
  GPIO11(pin23):
  GPIO12(pin32):
  GPIO13(pin33):
  GPIO14(pin08):  SYNC *** パルス幅測定用同期信号
  GPIO15(pin10):
  GPIO16(pin36):
  GPIO17(pin11):  GATE *** パルス幅信号
  GPIO18(pin12):  EVENT3 *** イベント信号#3
  GPIO19(pin35):
  GPIO20(pin38):
  GPIO21(pin40):
  GPIO22(pin15):  EVENT1 *** イベント信号#1
  GPIO23(pin16):  EVENT0 *** イベント信号#0
  GPIO24(pin18):
  GPIO25(pin22):
  GPIO26(pin37):
  GPIO27(pin13):  EVENT2 *** イベント信号#2
*/
/* GPIO 信号名の指定 */
#define GPIO_I2C   1 /* I2C チャンネル #1 */
#define GPIO_SDA   2
#define GPIO_SCL   3
#define GPIO_SYNC  14 /* 時間測定用信号 */
#define GPIO_PULSE  4
#define GPIO_FREQ  GPIO_PULSE
#define GPIO_GATE   17
#define GPIO_EVENT0  23 /* 汎用（イベント）測定用信号
*/
#define GPIO_EVENT1  22
#define GPIO_EVENT2  27
#define GPIO_EVENT3  18
#define __GPIO
#endif
```

作業ディレクトリに戻って、pigpio ライブラリを起動するシェルスクリプトを用意し、実行可能（`chmod a+x run_gpio`）にしておきます。

run_gpio

```
sudo pigpiod -s 10
```

このシェルスクリプトは、Raspberry Pi が立ち上がったら自動的に実行するよう設定しておくと便利です。

次にテスト用の信号を発生するライブラリを作成します。

pic_board.py

```
/* pic_board.py PIC 評価ボード用に GPIO を初期化する
                 各種パルスを発生させる
  初版: 2021/5/12 Chuji
  最新版:
*/
#define BCM2835
#define PICCOLO_I2C  0x38
#include "include/use-pigpio.h"
#include "include/use-time.h"
#include "include/piccolo.h"
/* 初期化 親プログラムは戻り値を pi という名前で保存すること
*/
def openPIC(): /* PIC 評価ボードへの出力を初期化する */
  pi = GPIO_OPEN()
  pi.GPIO_MODE(GPIO_SYNC, GPIO_OUT)
  pi.GPIO_MODE(GPIO_PULSE, GPIO_OUT)
  pi.GPIO_MODE(GPIO_GATE, GPIO_OUT)
  pi.GPIO_MODE(GPIO_EVENT0, GPIO_OUT)
  pi.GPIO_MODE(GPIO_EVENT1, GPIO_OUT)
  pi.GPIO_MODE(GPIO_EVENT2, GPIO_OUT)
  pi.GPIO_MODE(GPIO_EVENT3, GPIO_OUT)
  pi.GPIO_WRITE(GPIO_SYNC, GPIO_LOW)
  pi.GPIO_WRITE(GPIO_PULSE, GPIO_LOW)
  pi.GPIO_WRITE(GPIO_GATE, GPIO_LOW)
  pi.GPIO_WRITE(GPIO_EVENT0, GPIO_LOW)
  pi.GPIO_WRITE(GPIO_EVENT1, GPIO_LOW)
  pi.GPIO_WRITE(GPIO_EVENT2, GPIO_LOW)
  pi.GPIO_WRITE(GPIO_EVENT3, GPIO_LOW)
  return pi
#define closePIC()  pi.GPIO_CLOSE()
#define SYNC_WIDTH  20 /* us */
#define FASTEST    0  /* 最大速度のパルス出力 */
#define SLOW_PULSE  0.01 /* 20ms 周期 */
/* SYNC パルスを発生させる */
#define sync()  pi.GENERATE_PULSE(GPIO_SYNC,
SYNC_WIDTH, GPIO_HIGH)
/* GATE パルスをのパルス幅を設定する gate(x) : x はパルス
幅(ms) */
#define gate(x)  pi.set_PWM_dutycycle(GPIO_GATE,
100 * x)
#define GATE_FREQUENCY  5  /* 毎秒 5 回 */
#define GATE_RANGE  20000  /* 設定単位は 0.01ms */
/* GATE パルスを繰り返し発生させる */
def create_gate(pi):
  pi.set_PWM_frequency(GPIO_GATE, GATE_FREQUENCY)
  pi.set_PWM_range(GPIO_GATE, GATE_RANGE)
  gate(0)
/* 指定したポートにソフトウエアパルスを発生させる */
def a_pulse(pi, port, gap):
  pi.GPIO_WRITE(port, GPIO_HIGH)
  if gap > 0:
    sleep(gap)
  pi.GPIO_WRITE(port, GPIO_LOW)
/* 指定したポートにソフトウエアパルス列 (n) を発生させる
*/
def generate_pulse(pi, port, n, gap):
  for i in range(n):
    a_pulse(pi, port, gap)
    if gap > 0:
      sleep(gap)
#define fast_pulse(n)  generate_pulse(pi,
GPIO_PULSE, n, FASTEST)
#define slow_pulse(port, n)  generate_pulse(pi,
port, n, SLOW_PULSE)
/* 周波数発生端子 PULSE に指定周波数のパルスを発生させる */
#define frequency(f)  pi.hardware_clock(GPIO_FREQ,
1000 * f) /* f in kHz */
/* 指定ポートに低速パルスを発生させる */
def slow_event(port, f): /* f: in Hz */
  pi.set_PWM_frequency(port, f)
  pi.set_PWM_range(port, 100)
  pi.set_PWM_dutycycle(port, 50)
```

```
/* 16 ビットデータのエンディアン修正 */
#define bswap(x)  ((x & 0xff00) >> 8) | ((x &
0x00ff) << 8)
```

親プロジェクトでも使っている、C プリプロセッサの出力を整形するプログラム cleanfile.py と、検証プログラムを実行するシェルスクリプト run_python を用意し、実行可能にしておきます。

cleanfile.py

```
while True:
  try:
    s = input()
    if s != "":
      if s[0] != "#":
        print (s)
  except EOFError:
    break
```

run_python

```
cpp $1 | python cleanfile.py >tmp.py; python
tmp.py
```

開発・評価ボードの検証

開発・評価ボードと信号発生ライブラリの検証を行っておきます。次のプログラムの(1)〜(5)のコメントを一つずつ外して実行すると、各種のパルス信号が発生することが分かります。(5)の GPIO_EVENT0 のところを EVENT1〜EVENT3 に変更して、混信がないことを確認しておきます。

test_board.py

```
/* PIC/PICCOLO 開発・評価ボードとライブラリの検証プログラ
ム
  初版:2021/5/13
  最新版:
*/
#include "pic_board.py"
pi = openPIC()
/* (1) パルス出力 (ハードウエアクロック) */
/*
for f in range(5, 11):
  print(f, 'kHz .....')
  frequency(f)
  sleep(2)
*/
/* (2) パルス出力 (低速ソフトウエアパルス) のパルス数を変
える */
/*
for i in range(1, 6):
  fast_pulse(i)
  sleep(1)
*/
/* (3) 同期信号 SYNC の発生 (パルス幅 20us) */
/*
for i in range(5):
  sync()
  sleep(1)
*/
/* (4) パルス幅 (1-10ms を ms 単位で指定する)信号の発生
*/
/*
create_gate(pi)
for i in range(1, 11):
```

```
  gate(i)
  sleep(1)
*/
/* (5) イベント(イベントパルスの数を変える)の発生 */
/*
for i in range(1, 6):
  print(i)
  slow_pulse(GPIO_EVENT3, i)
  sleep(1)
*/
closePIC()
```

簡易型ロジックアナライザで SYNC, PULSE, GATE, EVENT0 の波形を調べたところ、以下のような結果になりました。(2)と(5)のパルスレートは、ホストに使った Raspberry Pi ZERO が pigpio デーモンと通信するオーバーヘッドが影響しています。

No	評価内容	結果
(1)	PULSE に 5～10kHz のパルス発生	5～10kHz の周波数発生
(2)	PULSE に 1～5 個のパルス発生	560pps でパルス発生
(3)	SYNC 信号の発生	20μs のパルス発生
(4)	GATE に 1～10ms のパルス幅発生	1～10ms のパルス発生
(5)	EVENT0 に 1～5 個のパルス発生	45pps でパルス発生

書き込み動作の検証手順

最後に 0.89V と 1.35V が発生していることを電圧計（テスタ）で確認すれば終わりです。

このボードとライブラリを使って、PICCOLO チップの動作を確認して行きます。PICKit4 や MPLAB X IDE の使い方の説明は省略しています。他の資料を参照してください。

ベースチップをソケットに装着し、PICKit4 を使ってファームウエアをダウンロードしてから、デバグモードで動作の確認を行います。

4.2 シミュレーションでの未検証項目

前章で（シミュレーションモデルの問題のため）検証できなかったのは、次の 5 点（前章の登場順に挙げてある）でした。これを中心に検証を進め、測定値の妥当性もある程度確認するのがこの章の目的です。

1. AD 変換器の基準電圧が 2.048V にならない
2. タイマー#1 のゲート回路が動作せず、計数できない
3. タイマー#2 と PWM3 で作る周波数測定用のゲート信号（100ms と 1 秒）が発生しない
4. NCO の演算が進まず、250μs の定周期割り込みが発生しない
5. I2C 通信を行う MSSP のモデルがなく、動作が検証できない

全体の手順は以下のとおりです、まずリセット直後から動作する（はずの）定周期割り込みを確認しておきます（未検証項目 4）。次に周波数測定のゲート信号発生とデータ転送割込みを確認します（未検証項目 3）。一番やっかいな I2C 通信（未検証項目 5）を次に検証します。未検証項目 1 と 2、それに測定値の妥当性確認は、I2C 通信を介して測定値が読み取れるようになってから行います。

4.3 定周期割り込みと周波数ゲートの検証

最初のうちは PIC プログラム piccolo.h で `MONITOR_HW` を#define した状態て検証を行います。しばらくの間、CPU クロックは 8MHz で動作させます。

4.3.1 定周期割り込み

【未検証項目 4】RC4 ポートは割り込み処理中だけ H レベルになります。チップの立ち上げ直後に、下のような波形が得られました（20μs/div)。

定周期割り込み処理の観測

最初から走っているのは、イベント計数用の定周期（250μs）処理です。シミュレータでは動作が確認できませんでしたが、250μs ごとに割り込みが発生しています。観測用にポートを使っているので、入力は 2 チャンネルだけですが、(イベント計数はしない設定で）処理時間は 80μs 程度でした。

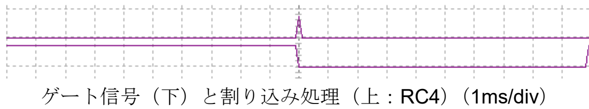
ときどき実行を止めて調べると、*SFR* 表示で `NCO1ACC` レジスタ（20 ビット長）の値が変化していることが確認できます。

これで定周期割り込みが検証できました。きちんと計数できているか調べるのは、I2C 通信ができるようになってからにします。他の割り込みを検証するあいだ、この割り込みは禁止（実行を停止させてから、*SFR* 表示で `PIE7.NCO1IE` をクリア）しておきます。

4.3.2 周波数ゲート信号の発生

【未検証項目 3】もうひとつ、シミュレーションではできなかった検証を行います。周波数測定機能のゲート信号を発生させてみます。いったんプログラムを停止（pause）させ、*Variables* 表示から `t_command_buf` の `PE`, `PM0`, `PM1` ビットと、`timerStatus.new_command` をセットします。プロ

グラムを再開すると、シミュレーションでも確認したように、10ms のゲート信号が発生し、RC5 ポートに出力されます。



ゲート信号（下）と割り込み処理（上：RC4）（1ms/div）

10ms のゲート信号（H レベル）の後ろに割り込みが発生していることが分かります。10ms 弱の休止時間（L レベル）のあと、次のゲート信号が始まっています。

PM1：PM0 を変更すれば、100ms（1:0）と 1 秒（0:1）も発生させられます。ただし、割り込み中信号は 30μs 以下なので、観測は難しいかもしれません。ゲート信号の立ち下がりで観測すれば、割り込みが発生していることが確認できます。

ゲート時間が正確かどうかは、I2C 通信ができるようになってから確認します。

4.4 I2C 通信の検証

【未検証項目 5】これ以降の検証を簡単にするため、最初に I2C 通信を確立しておきます。

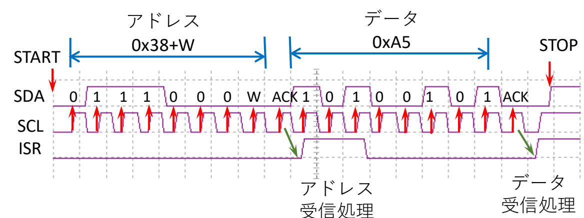
4.4.1 書き込み動作

いちばん単純な処理として、1 バイトを書き込む動作を確認します。次のプログラムは、PICCOLO チップに 0xA5 というデータを書き込みます。PICCOLO チップの応用層は、このデータをレジスタ番号として解釈しようとしますが、存在しないレジスタなので、無視します。

test_i2c.py

```
/* PIC/PICCOLO I2C 通信機の検証プログラム
   初版：2021/5/17
   最新版：
*/
#include "pic_board.py"
pi = openPIC()
piccolo = pi.OPEN_I2C(PICCOLO_I2C)
pi.I2C_WRITE_DEVICE(piccolo, 0xa5)
pi.CLOSE_I2C(piccolo)
closePIC()
```

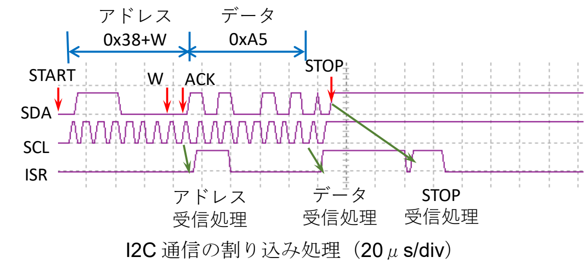
SDA, SCL それに割り込み処理 ISR（RC4）を記録できるようにして、SDA の立ち下がりでトリガをかけると、次のような波形が観測できます（観測機器の制約から、START デリミタは記録できません）。



I2C 書き込み動作の観測（10μs/div）

この後プログラムを止めて、受信バッファ SSP1BUF を読み取ってみると、たしかに 0xA5 を受信していることが確認できます。

記録を遅くすると、処理の全体が見えてきます。

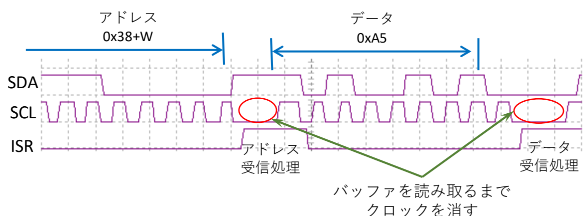


I2C 通信の割り込み処理（20μs/div）

MSSP は自アドレスを受信すると ACK を返し、割り込みを発生します。データを受信したときと、STOP を検出したときも同様で、それぞれ ISR が呼び出されていることが分かります。

一見したところ、ISR は最初に目論んだとおりの時間内に処理ができています。しかし、他の割り込み処理中に受信すると、それが終わるまで受信処理ができません。その結果、受信バッファを読み取る前に、次のデータを上書きしようとするエラーが起こり、通信が異常終了してしまいます。定周期割り込みが、いちばん大きな原因になります。

CPU クロックを変える前に、安全策を試しました。受信時もクロックストレッチ（一時的にクロック SCL を消してしまう）ができるようにして、受信バッファを読み取るまで、次の送信を待たせます。PIC ファームウエア（piccolo.h）で SAFE_I2C を #define すれば、この機能が使えるようにしてあります。その結果を次に示します。



受信時クロックストレッチを入れた書き込み動作（10μs/div）

受信バッファが空になってから、データ転送が始まっています。通常はクロック一個か二個分を待たせるだけですが、他の割り込み処理中はストレッチを

続けるので、通信エラーは起きません（いったん受信バッファを空にしたら、i2c_ISR()の処理の最後から MSSP が受信できるようになる）。

最後に、実際にデータを書き込んでみます。次の検証プログラムは、10ms の周波数測定を始めるコマンドを書き込みます。

```
test_i2c_write.py
```

```
/* PIC/PICCOLO I2C 通信機能の検証プログラム ... コマンド書き込み
   初版：2021/5/17
   最新版：
*/
#include "pic_board.py"
pi = openPIC()
piccolo = pi.OPEN_I2C(PICCOLO_I2C)
pi.I2C_WRITE_BYTE(piccolo, 0x02, 0x00)
pi.CLOSE_I2C(piccolo)
closePIC()
```

書き込みが成功すれば、RC5 ポートに 10ms のゲート信号が発生するはずです。この動作を確認しました（結果は掲載していません）。

4.4.2 読み取り動作（その 1）

次にレジスタを読み取ってみます。

```
test_i2c_read.py
```

```
/* PIC/PICCOLO I2C 通信機能の検証プログラム ... データ読み取り□
   初版：2021/5/17
   最新版：
*/
#include "pic_board.py"
pi = openPIC()
piccolo = pi.OPEN_I2C(PICCOLO_I2C)
reg = input('Register # to read? ...')
pi.I2C_WRITE_DEVICE(piccolo, int(reg))
input('Read register #4 content...')
(n, dat) = pi.I2C_READ_DEVICE_BLOCK(piccolo, 2)
/* print(dat) */
print('Data read from Piccolo: 0x%2x, 0x%2x' %
(dat[0], dat[1]))
pi.CLOSE_I2C(piccolo)
closePIC()
```

このプログラムは、レジスタ番号の書き込みと、データの読み出しを別々のタイミングで行っています。実行前に、いったん PICCOLO チップの動作を停止し、Variable 表示からデータを書き換えておくと、そのデータが読み出せるか確認することができます。動作を再開して、上のプログラムを実行すると、指定したレジスタの内容が読み取れます。必ず 2 バイト読み出そうとするので、T-Status レジスタを指定すると、2 バイト目にはダミーデータ(0xff)が読み出されます。

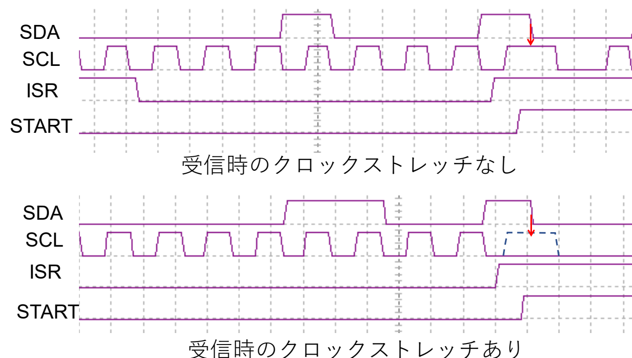
4.4.3 読み取り動作（その 2）

次に『繰返し START（「**I2C 通信の概要**」の章を参照）』を使って、Write/Read をまとめて行ってみます。SAFE_I2C を#define していないときは、指定したレジスタが読み取れますが、#define すると PICCOLO チップがデータを送りだしてくれません（読み出し値は 0xff）。

```
test_i2c_register.py
```

```
/* PIC/PICCOLO I2C 通信機能の検証プログラム ... レジスタの読み出し
   初版：2021/5/17
   最新版：
*/
#include "pic_board.py"
pi = openPIC()
piccolo = pi.OPEN_I2C(PICCOLO_I2C)
reg = input('Register to read: ')
dat = pi.I2C_READ_WORD(piccolo, int(reg))
print('Data = 0x%4x' % dat)
pi.CLOSE_I2C(piccolo)
closePIC()
```

通信波形（次の図（10μs/div）を見てください）を調べると、とんでもないことが分かりました。START デリミタは、SCL が H レベルのときに SDA を H→L と変化させることで表現します。ところが Raspberry Pi は、受信時クロックストレッチで SCL が消えたことに気が付かず、SDA を変化させてしまっています。そのせいで MSSP が『繰返し START』だと認識せず、書き込みが続いているとして受信動作（SDA がプルアップされているので常に H）を続けてしまいます。



受信時クロックストレッチで START が消えてしまう？

マスター（Raspberry Pi）は、クロックストレッチがあることに備えて、SCL ラインを監視することを求められているのですが……原因は分かりません。これはマスターの問題なので、「こういうマスターもいる」と割り切って対策を考えます。可能な対策は、

1. 『繰返し START』は対応せず、マスターは Write/Read を別々に実行する
2. 受信時クロックストレッチを行わない

くらいです。1. はマスターに対して「そっちのせいだから対応してよ」と求めるわけですが、あまり『スレーブ』らしくありません。2. を選択するうえでの問題を検討します。

アドレス受信時の割り込み処理は、SCL の 1～2 クロック分で終わります。その直後に別の割り込みが入っても、10 クロック（100μs）以内に処理を終えていれば、次のデータ（レジスタ番号）を取り損なうことはありません。それを過ぎると、SSP1BUF を読み取る前にデータ転送が終わり、受信オーバーフローが起きてしまいます。それを避けるには、すべての ISR を 100μs 以内に終わるよう設計する必要があります。イベント計数にかかる処理が 241 サイクルなので、CPU クロックは 8MHz では足りず、16MHz か 32MHz にする必要があります。

この受信オーバーフローによる通信障害を、クロック周波数でどれだけ回避できるかは最後に確認します。とりあえずは定周期割り込みを禁止して、8MHz で検証を進めます。

4.5　時間計測機能の検証

4.5.1 パルス幅測定機能

【未検証項目 2】シミュレータでは検証できなかった、パルス幅測定機能を検証します。Raspberry Pi から発生させたゲート信号のパルス幅を測定します。

非同期モード

与える信号が単純で精度が出しやすい非同期モード（GATE 信号が L になった直後に測定値を取り込む）を検証します。

test_width.py

```
/* PIC/PICCOLO パルス幅測定機能の検証プログラム -- 非同期型
  初版：2021/5/27
  最新版：
*/
#include "pic_board.py"
pi = openPIC()
piccolo = pi.OPEN_I2C(PICCOLO_I2C)
#ifdef HIGH_RES
pi.I2C_WRITE_BYTE(piccolo, T_COMMAND, WIDTH_ENABLE
| WIDTH_ASYNC | WIDTH_ACTIVEH | WIDTH_CLOCK2us)
#else
pi.I2C_WRITE_BYTE(piccolo, T_COMMAND, WIDTH_ENABLE
| WIDTH_ASYNC | WIDTH_ACTIVEH | WIDTH_CLOCK8us)
#endif
create_gate(pi)
gate(0)
sleep(0.5)
for i in range(1, 10):
  gate(10*i)
  sleep(0.5)
  dat = pi.I2C_READ_WORD(piccolo, WIDTH_DATA)
  count = ((dat & 0xff) << 8) | ((dat & 0xff00) >>
8)
#ifdef HIGH_RES
  wid = count * 2/1000
#else
  wid = count * 8/1000
#endif
  print('Width = %3d ms, Measured = %3d ms, Data
= %4d  (dat = 0x%4x)' % (10*i,wid, count, dat))
pi.CLOSE_I2C(piccolo)
closePIC()
```

上のプログラムを走らせると、パルス幅を変えながら測定が行われます。GATE 信号が H→L と変化した直後に約 25μs の割り込み処理が発生することが分かりました。測定クロックは 8μs（デフォルト）と 2μs（HIGH_RES）の両方で試しました。

```
$ ./run_python test_width.py
Width =  10 ms, Measured =  10 ms, Data = 1255
(dat = 0xe704)
Width =  20 ms, Measured =  20 ms, Data = 2510
(dat = 0xce09)
Width =  30 ms, Measured =  30 ms, Data = 3765
(dat = 0xb50e)
Width =  40 ms, Measured =  40 ms, Data = 5021
(dat = 0x9d13)
Width =  50 ms, Measured =  50 ms, Data = 6276
(dat = 0x8418)
Width =  60 ms, Measured =  60 ms, Data = 7534
(dat = 0x6e1d)
Width =  70 ms, Measured =  70 ms, Data = 8788
(dat = 0x5422)
Width =  80 ms, Measured =  80 ms, Data = 10042
(dat = 0x3a27)
Width =  90 ms, Measured =  90 ms, Data = 11301

$ (dat = 0x252c)
cpp -DHIGH_RES test_width.py | python
Width =  10 ms, Measured =  10 ms, Data = 5019
(dat = 0x9b13)
Width =  20 ms, Measured =  20 ms, Data = 10038
(dat = 0x3627)
Width =  30 ms, Measured =  30 ms, Data = 15061
(dat = 0xd53a)
Width =  40 ms, Measured =  40 ms, Data = 20082
(dat = 0x724e)
Width =  50 ms, Measured =  50 ms, Data = 25098
(dat = 0x a62)
Width =  60 ms, Measured =  60 ms, Data = 30122
(dat = 0xaa75)
Width =  70 ms, Measured =  70 ms, Data = 35141
(dat = 0x4589)
Width =  80 ms, Measured =  80 ms, Data = 40174
(dat = 0xee9c)
Width =  90 ms, Measured =  90 ms, Data = 45182
(dat = 0x7eb0)
```

GATE 信号はハードウエア PWM で発生させているので、安定しているはずです。測定値は 1%くらいずれているようですが、どちらに原因があるか分かりません。確認には精度の良い測定器が必要なので、とりあえず放置します。これで時間幅測定機能の検証が（精度を除いて）できました。

同期モード

同期モードでは、GATE 信号が L になってから発生する SYNC で割り込みがかかり、データの取り込みが行われます。

```
test_sync.py
```

```
/* PIC/PICCOLO パルス幅測定機能の検証プログラム（同期モード）
  初版：2021/5/27
  最新版：
*/
#include "pic_board.py"
pi = openPIC()
piccolo = pi.OPEN_I2C(PICCOLO_I2C)
pi.I2C_WRITE_BYTE(piccolo, T_COMMAND, WIDTH_ENABLE
| WIDTH_SYNC | WIDTH_ACTIVEH | WIDTH_CLOCK8us)
sleep(1)
a_pulse(pi, GPIO_SYNC, FASTEST)
for i in range(20):
  t = i * 0.01
  a_pulse(pi, GPIO_GATE, t)
  a_pulse(pi, GPIO_SYNC, 0)
  sleep(0.5)
  dat = pi.I2C_READ_WORD(piccolo, WIDTH_DATA)
  count = ((dat & 0xff) << 8) | ((dat & 0xff00) >>
8)
  width = count * 8/1000
  print('Width = %3d ms, Measured = %3d ms, Data
= %4d  (dat = 0x%4x)' % (10*i, width, count, dat))
pi.CLOSE_I2C(piccolo)
closePIC()
```

結果は以下のとおりです。このプログラムのパルス幅は、sleep()で決めており、pigpio コールのオーバーヘッドも含まれているので精度が良くありません。

```
$ ./run_python test_sync.py
Width =   0 ms, Measured =   0 ms, Data =    81
(dat = 0x5100)
Width =  10 ms, Measured =  11 ms, Data = 1424
(dat = 0x9005)
Width =  20 ms, Measured =  21 ms, Data = 2679
(dat = 0x770a)
Width =  30 ms, Measured =  31 ms, Data = 3936
(dat = 0x600f)
Width =  40 ms, Measured =  41 ms, Data = 5193
(dat = 0x4914)
Width =  50 ms, Measured =  51 ms, Data = 6442
(dat = 0x2a19)
Width =  60 ms, Measured =  61 ms, Data = 7705
(dat = 0x191e)
Width =  70 ms, Measured =  71 ms, Data = 8986
(dat = 0x1a23)
Width =  80 ms, Measured =  81 ms, Data = 10219
(dat = 0xeb27)
Width =  90 ms, Measured =  91 ms, Data = 11473
(dat = 0xd12c)
Width = 100 ms, Measured = 101 ms, Data = 12731
(dat = 0xbb31)
Width = 110 ms, Measured = 111 ms, Data = 13979
(dat = 0x9b36)
Width = 120 ms, Measured = 121 ms, Data = 15242
(dat = 0x8a3b)
Width = 130 ms, Measured = 131 ms, Data = 16493
(dat = 0x6d40)
Width = 140 ms, Measured = 142 ms, Data = 17783
(dat = 0x7745)
Width = 150 ms, Measured = 152 ms, Data = 19007
(dat = 0x3f4a)
Width = 160 ms, Measured = 161 ms, Data = 20247
(dat = 0x174f)
Width = 170 ms, Measured = 172 ms, Data = 21522
(dat = 0x1254)
Width = 180 ms, Measured = 182 ms, Data = 22773
(dat = 0xf558)
Width = 190 ms, Measured = 192 ms, Data = 24034
(dat = 0xe25d)
```

4.5.2 周波数測定機能

次に周波数測定機能を検証します。ゲート信号と割り込みが発生していることは、既に確認しています。

10kHz から 90kHz までのパルス列を発生させ、周波数（ゲート時間内のパルス数）を測定します。テストプログラムでは、pigpio ライブラリの hardware_clock()関数を使って、kHz 帯のパルスを発生させています。1 秒のゲート時間では 64kHz 以上でオーバーフローが起こるので、その分（65,536）を足して表示するようにしました。

```
test_freq.py
```

```
/* PIC/PICCOLO 周波数測定機能の検証プログラム
  初版：2021/5/27
  最新版：
*/
#include "pic_board.py"
pi = openPIC()
piccolo = pi.OPEN_I2C(PICCOLO_I2C)
#ifdef F100ms
pi.I2C_WRITE_BYTE(piccolo, T_COMMAND, FREQ_ENABLE
| FREQ_100ms)
#else
#ifdef F1s
pi.I2C_WRITE_BYTE(piccolo, T_COMMAND, FREQ_ENABLE
| FREQ_1s)
#else
pi.I2C_WRITE_BYTE(piccolo, T_COMMAND, FREQ_ENABLE
| FREQ_10ms)
#endif
#endif
for i in range(1, 10):
  frequency(10*i)
  sleep(2.5)
  dat = pi.I2C_READ_WORD(piccolo, FREQ_DATA)
  stat = pi.I2C_READ_BYTE(piccolo, T_STATUS)
  stat = STATUS(stat, FREQ_MASK, FREQ_POS)
  count = ((dat & 0xff) << 8) | ((dat & 0xff00) >>
8)
  if (stat & OVERFLOW):
    count += 65536
  print('Freq = %2d kHz, Count = %6d (dat =
0x%4x)' % (10*i, count, dat))
pi.CLOSE_I2C(piccolo)
closePIC()
```

検証結果を次に示します。ゲート時間は上から順番に 10ms、100ms、1s です。おおむね±0.1%+数カウント程度の誤差に収まっていることが分かります。データシートによると、ベースチップの内部クロック源は工場で較正されており、常温下での精度は±2%以内ということです。許容範囲の結果だと思います。

```
$ ./run_python test_freq.py
Freq = 10 kHz, Count =    102 (dat = 0x6600)
Freq = 20 kHz, Count =    203 (dat = 0xcb00)
Freq = 30 kHz, Count =    303 (dat = 0x2f01)
Freq = 40 kHz, Count =    405 (dat = 0x9501)
Freq = 50 kHz, Count =    506 (dat = 0xfa01)
Freq = 60 kHz, Count =    606 (dat = 0x5e02)
Freq = 70 kHz, Count =    707 (dat = 0xc302)
Freq = 80 kHz, Count =    809 (dat = 0x2903)
Freq = 90 kHz, Count =    910 (dat = 0x8e03)

$ cpp -DF100ms test_freq.py |python
Freq = 10 kHz, Count =   1002 (dat = 0xea03)
Freq = 20 kHz, Count =   2002 (dat = 0xd207)
```

```
Freq = 30 kHz, Count =   3004 (dat = 0xbc0b)
Freq = 40 kHz, Count =   4005 (dat = 0xa50f)
Freq = 50 kHz, Count =   5006 (dat = 0x8e13)
Freq = 60 kHz, Count =   6006 (dat = 0x7617)
Freq = 70 kHz, Count =   7008 (dat = 0x601b)
Freq = 80 kHz, Count =   8010 (dat = 0x4a1f)
Freq = 90 kHz, Count =   9007 (dat = 0x2f23)

$ cpp -DF1s test_freq.py |python
Freq = 10 kHz, Count =  10005 (dat = 0x1527)
Freq = 20 kHz, Count =  20006 (dat = 0x264e)
Freq = 30 kHz, Count =  30015 (dat = 0x3f75)
Freq = 40 kHz, Count =  40020 (dat = 0x549c)
Freq = 50 kHz, Count =  50026 (dat = 0x6ac3)
Freq = 60 kHz, Count =  60037 (dat = 0x85ea)
Freq = 70 kHz, Count =  70059 (dat = 0xab11)
Freq = 80 kHz, Count =  80057 (dat = 0xb938)
Freq = 90 kHz, Count =  90046 (dat = 0xbe5f)
```

4.5.3 周波数測定用ゲート時間

時間測定機能の検証が済んだところで、本来は不要だが趣味的な検証をしてみましょう。周波数測定用ゲートのパルス幅を、パルス幅測定にかけます。

test_width_loop.py

```
/* PIC/PICCOLO パルス幅測定機能/周波数測定機能の検証プログラム
  初版：2021/5/27
  最新版：
*/
#include "pic_board.py"
pi = openPIC()
piccolo = pi.OPEN_I2C(PICCOLO_I2C)
pi.I2C_WRITE_BYTE(piccolo, T_COMMAND, 0x7B)
sleep(1)
dat = pi.I2C_READ_WORD(piccolo, WIDTH_DATA)
count = bswap(dat)
wid = count * 2/1000
print('Width = 10 ms, Measured = %2.3f ms, Count = %4d)' % (wid, count))
pi.I2C_WRITE_BYTE(piccolo, T_COMMAND, 0x5B)
sleep(2)
dat = pi.I2C_READ_WORD(piccolo, WIDTH_DATA)
count = bswap(dat)
ov = pi.I2C_READ_BYTE(piccolo, T_STATUS)
if (ov & 2):
  count += 65536
wid = count * 2/1000
print('Width = 100 ms, Measured = %3.3f ms, Count = %5d)' % (wid, count))
pi.I2C_WRITE_BYTE(piccolo, T_COMMAND, 0x33)
sleep(3)
dat = pi.I2C_READ_WORD(piccolo, WIDTH_DATA)
count = bswap(dat)
ov = pi.I2C_READ_BYTE(piccolo, T_STATUS)
if (ov & 2):
  count += 65536
wid = count * 8/1000
print('Width = 1000 ms, Measured = %4.3f ms, Count = %6d)' % (wid, count))
pi.CLOSE_I2C(piccolo)
closePIC()
```

ジャンパーワイヤーを使って、RC5 ピン（周波数測定用ゲート信号）と GATE ピン（PICCOLO 側）を接続します。

```
$ ./run_python test_width_loop.py
Width = 10 ms, Measured = 10.144 ms, Count = 5072)
Width = 100 ms, Measured = 100.480 ms, Count = 50240)
Width = 1000 ms, Measured = 1004.544 ms, Count = 125568)
```

ゲート時間の設計値は、10.016ms、99.97ms、1000.4ms でした。同じクロックから両方の信号を作っているのにもかかわらず、少し誤差が大きすぎると思います。ただ、それなりの測定器が必要だし、分析に時間をかけすぎると、元プロジェクトに戻れません。元プロジェクトの要求精度は満たしているので、ペンディングにしておくことにします。

4.6 汎用測定機能の検証

測定機能検証の最後は、汎用測定機能です。今のところ、IN0 と IN1 の 2 チャンネルだけが検証対象です。他の割り込みを検証するため、定周期割り込みを禁止していたら、`PIE7.NCO1IE` をセットして割り込みを可能にします。

4.6.1 イベント計数機能

pigpio ライブラリのソフトウエア PWM を使って、IN0（40Hz）と IN1（50Hz）にイベント信号を与え、計数させてみます。

test_event.py

```
/* PIC/PICCOLO 汎用測定機能（イベント計数）の検証プログラム
  初版：2021/5/27
  最新版：
*/
#include "pic_board.py"
def event_count():
  for i in range(30):
    e0 = pi.I2C_READ_WORD(piccolo, VERSATILE0)
    e0 = bswap(e0)
    e1 = pi.I2C_READ_WORD(piccolo, VERSATILE1)
    e1 = bswap(e1)
#ifdef CHANNEL4
    e2 = pi.I2C_READ_WORD(piccolo, VERSATILE2)
    e2 = bswap(e2)
    e3 = pi.I2C_READ_WORD(piccolo, VERSATILE3)
    e3 = bswap(e3)
    print ('%2d:  CH0 = %4d, CH1 = %4d, CH2 = %4d, CH3 = %4d' % (i, e0, e1, e2, e3))
#else
    print ('%2d:  CH0 = %4d, CH1 = %4d' % (i, e0, e1))
#endif
    sleep(1)
pi = openPIC()
piccolo = pi.OPEN_I2C(PICCOLO_I2C)
pi.I2C_WRITE_BYTE(piccolo, A_COMMAND, 0) /* 初期化 */
slow_event(GPIO_EVENT0, 40)
sleep(0.5)
slow_event(GPIO_EVENT1, 50)
#ifdef CHANNEL4
slow_event(GPIO_EVENT2, 20)
sleep(0.5)
slow_event(GPIO_EVENT3, 25)
#endif
print('Rising edge detection ----')
#ifdef CHANNEL4
pi.I2C_WRITE_BYTE(piccolo, A_COMMAND, 0xaa)
#else
pi.I2C_WRITE_BYTE(piccolo, A_COMMAND, 0x0a)
#endif
sleep(1)
event_count()
pi.I2C_WRITE_BYTE(piccolo, A_COMMAND, 0)
sleep(0.5)
print('Both edge detection ----')
#ifdef CHANNEL4
pi.I2C_WRITE_BYTE(piccolo, A_COMMAND, 0xff)
```

```
#else
pi.I2C_WRITE_BYTE(piccolo, A_COMMAND, 0x0f)
#endif
sleep(1)
event_count()
pi.CLOSE_I2C(piccolo)
closePIC()
```

開発・評価ボードの汎用入力の EVENT 側に、ショートピンを差し込んでから実行すると次のようになりました。

```
$ ./run_python test_event.py
Rising edge detection ----
 0:  CH0 =   40, CH1 =   50
 1:  CH0 =   80, CH1 =  100
 2:  CH0 =  121, CH1 =  150
 3:  CH0 =  161, CH1 =  201
 4:  CH0 =  201, CH1 =  251
 5:  CH0 =  241, CH1 =  301
 6:  CH0 =  281, CH1 =  351
 7:  CH0 =  322, CH1 =  402
 8:  CH0 =  362, CH1 =  452
 9:  CH0 =  402, CH1 =  502
Both edge detection ----
 0:  CH0 =   81, CH1 =  100
 1:  CH0 =  161, CH1 =  201
 2:  CH0 =  241, CH1 =  301
 3:  CH0 =  322, CH1 =  402
 4:  CH0 =  402, CH1 =  503
 5:  CH0 =  483, CH1 =  603
 6:  CH0 =  563, CH1 =  704
 7:  CH0 =  644, CH1 =  804
 8:  CH0 =  724, CH1 =  905
 9:  CH0 =  804, CH1 = 1005
```

立ち上がり計数で、チャンネル 0 は 40 ずつ、チャンネル 1 は 50 ずつ増えています。立ち上がり・立ち下がり計数ではその倍になっています。サンプリング間隔を sleep() で決めているので、少し誤差が出ています。

正確にイベントを数えているか確認するため、発生させるイベント数を指定する方法でも試してみます。

test_num_event.py

```
/* PIC/PICCOLO 汎用測定機能（イベント計数）の検証プログラム -- イベント数指定
  初版：2021/5/27
  最新版：
*/
#include "pic_board.py"
def event_count():
  for i in range(100):
    slow_pulse(GPIO_EVENT0, 40)
    slow_pulse(GPIO_EVENT1, 50)
    e0 = pi.I2C_READ_WORD(piccolo, VERSATILE0)
    e0 = bswap(e0)
    e1 = pi.I2C_READ_WORD(piccolo, VERSATILE1)
    e1 = bswap(e1)
    print ('%2d:  CH0 = %4d, CH1 = %4d' % (i, e0,
e1))
    sleep(1)
pi = openPIC()
piccolo = pi.OPEN_I2C(PICCOLO_I2C)
pi.I2C_WRITE_BYTE(piccolo, A_COMMAND, 0) /* 初期化
*/
sleep(0.5)
print('Rising edge detection ----')
pi.I2C_WRITE_BYTE(piccolo, A_COMMAND, 0x0a)
event_count()
pi.I2C_WRITE_BYTE(piccolo, A_COMMAND, 0) /* 初期化
*/
sleep(0.5)
print('Both edge detection ----')
pi.I2C_WRITE_BYTE(piccolo, A_COMMAND, 0x0f)
event_count()
pi.CLOSE_I2C(piccolo)
closePIC()
```

こんどは発生させたイベントの数だけ、計数が進んでいるのが確認できました。

```
$ ./run_python test_num_event.py
Rising edge detection ----
 0:  V0 =   40, V1 =   50
 1:  V0 =   80, V1 =  100
 2:  V0 =  120, V1 =  150
 3:  V0 =  160, V1 =  200
 4:  V0 =  200, V1 =  250
 5:  V0 =  240, V1 =  300
 6:  V0 =  280, V1 =  350
 7:  V0 =  320, V1 =  400
 8:  V0 =  360, V1 =  450
 9:  V0 =  400, V1 =  500
Both edge detection ----
 0:  V0 =   80, V1 =  100
 1:  V0 =  160, V1 =  200
 2:  V0 =  240, V1 =  300
 3:  V0 =  320, V1 =  400
 4:  V0 =  400, V1 =  500
 5:  V0 =  480, V1 =  600
 6:  V0 =  560, V1 =  700
 7:  V0 =  640, V1 =  800
 8:  V0 =  720, V1 =  900
 9:  V0 =  800, V1 = 1000
```

最後に全部の汎用入力を使えるようにします。PICCOLO チップのプログラム（piccolo.h）で #define MONITOR_HW をコメントアウトしてから、IN0～IN3 の動作を確認したら、イベント計数機能の検証は終わりです。

```
$ cpp -DCHANNEL4 test_event.py |python
Rising edge detection ----
 0:  CH0 =   40, CH1 =   50, CH2 =   20, CH3 =
25
 1:  CH0 =   80, CH1 =  101, CH2 =   41, CH3 =
50
 2:  CH0 =  121, CH1 =  151, CH2 =   61, CH3 =
75
 3:  CH0 =  161, CH1 =  201, CH2 =   81, CH3 =
101
 4:  CH0 =  202, CH1 =  252, CH2 =  101, CH3 =
126
 5:  CH0 =  242, CH1 =  302, CH2 =  121, CH3 =
151
 6:  CH0 =  282, CH1 =  353, CH2 =  141, CH3 =
176
 7:  CH0 =  323, CH1 =  403, CH2 =  162, CH3 =
202
 8:  CH0 =  363, CH1 =  454, CH2 =  182, CH3 =
227
 9:  CH0 =  403, CH1 =  504, CH2 =  202, CH3 =
252
10:  CH0 =  444, CH1 =  555, CH2 =  222, CH3 =
277
```

4.6.2 電圧測定機能

【未検証項目 1】シミュレータではフルスケールが基準電圧 2.048V ではなく、なぜか 3.3V になっていました。開発・評価ボードの汎用入力選択ジャンパをすべて電圧側にし、チャンネル毎の電圧を互い違いに選んで、次のプログラムを実行します。

```
test_adc.py
```

```
/* PIC/PICCOLO ADC 変換結果の検証プログラム
   初版：2021/5/27
   最新版：
*/
#include "pic_board.py"
Aregister = [VERSATILE0, VERSATILE1, VERSATILE2,
VERSATILE3]
pi = openPIC()
piccolo = pi.OPEN_I2C(PICCOLO_I2C)
pi.I2C_WRITE_BYTE(piccolo, A_COMMAND, 0x55)
sleep(1)
print('Status = ', pi.I2C_READ_BYTE(piccolo,
A_STATUS))
for i in range(0, 4):
   dat = pi.I2C_READ_WORD(piccolo, Aregister[i])
   adc = ((dat & 0xff) << 8) | ((dat & 0xff00) >>
8)
   voltage = 2.048 * (adc/0x400)
   print('Channel: %d  Data = 0x%4x  Voltage
= %1.2f (V)' % (i, dat, voltage))
pi.CLOSE_I2C(piccolo)
closePIC()
```

結果の一例を次に示します。確かに 2.048V がフルスケールになっていることが分かります。Status 170（0xaa）は、4 チャンネルともデータが Ready であることを示しています。電圧選択ジャンパを適当に入れ替え、入れ替えたチャンネルの電圧が変わることを確認しておきます。

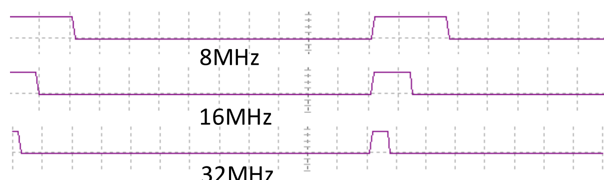
```
$ ./run_python test_adc.py
Status =  170
Channel: 0  Data = 0xc001  Voltage = 0.90 (V)
Channel: 1  Data = 0xae02  Voltage = 1.37 (V)
Channel: 2  Data = 0xbf01  Voltage = 0.89 (V)
Channel: 3  Data = 0xae02  Voltage = 1.37 (V)
```

この試験をしたときの 5V 電源の電圧は 5.12V でした。抵抗分割した電圧は 1.38V なので、納得できる結果です。この分割抵抗は、元プロジェクトの自律走行車で電池電圧を測定する回路と同じものです。可変抵抗にして、入力電圧を変えながら測定できるようにしておけば良かったのかもしれません。

4.7 MPU クロック周波数の影響

4.7.1 定周期割り込みの負荷

ここまで棚上げにしてきた、定周期割り込みの処理時間が I2C 通信に与える影響を評価します。最初に `MONITOR_HW` を#define して、リセット直後の ISR 処理時間（RA4 出力）をモニターします。MPU クロック周波数を変えたときの波形を次に示します（20μs/div）。



MPU クロック周波数で定周期割り込みの処理時間が変わる

波形観測のため汎用入力は 2 チャンネルだけになっています。最大 4 チャンネルのときは、この 2 倍近くの処理時間がかかります。処理時間の目標の目安は 50μs なので、16MHz でもなんとか（I2C 通信の処理も早くなるため）間に合いそうです。

I2C 通信割り込みの直前に、長い定周期割り込み処理が始まると、受信データの取り損ないが起こり、マスターから見ると「返事がない」というエラーに繋がってしまいます。これが起こっているか確認するため、実際にイベント計数をさせながら、測定値の読み出しを行います。試験の前に`#define MONITOR_HW` をコメントアウトして、開発・評価ボードのジャンパで IN2/IN3 にもイベント信号が伝わるようにしておきます。

4 チャンネルにイベント信号を与えるようにして、測定値を test_event.py で読み出します。

確かに MPU クロック 8MHz では、ときどき通信エラーが起こります。

```
$ cpp -DCHANNEL4 test_event.py | python
Rising edge detection ----
 0:  CH0 =    40, CH1 =   51, CH2 =   20, CH3 =
25
 1:  CH0 =    81, CH1 =  101, CH2 =   40, CH3 =
50
Traceback (most recent call last):
  File "<stdin>", line 127, in <module>
  File "<stdin>", line 89, in event_count
  File "/usr/local/lib/python3.7/dist-
packages/pigpio.py", line 2897, in
i2c_read_word_data
    return _u2i(_pigpio_command(self.sl,
_PI_CMD_I2CRW, handle, reg))
  File "/usr/local/lib/python3.7/dist-
packages/pigpio.py", line 1011, in _u2i
    raise error(error_text(v))
pigpio.error: 'I2C read failed'
```

そこで MPU クロック周波数を 16MHz に変えてみたところ、奇妙な現象に出くわしました。通信エラーは起こらなくなったのですが、計数値が減少しているように見えるところ（次の記録の赤字で示した個所）があります。それなのに、その後は何ごともなかったように増えていきます。良く調べてみると、赤字の個所は、本来のデータより 128 だけ少なく見えていることが分かります。つまりバイトデータの MSB が 1→0 に化けているのです。原因調査を始めました。

```
cpp -DCHANNEL4 test_event.py | python
Rising edge detection ----
 0:  CH0 =    40, CH1 =   50, CH2 =   20, CH3 =
25
 1:  CH0 =    81, CH1 =  101, CH2 =   40, CH3 =
50
 2:  CH0 =   121, CH1 =   23, CH2 =   60, CH3 =
75
 3:  CH0 =    33, CH1 =   73, CH2 =   80, CH3 =
101
 4:  CH0 =    74, CH1 =  252, CH2 =  101, CH3 =
126
```

```
 5:  CH0 =   114, CH1 =   302, CH2 =   121, CH3 =
151
 6:  CH0 =   282, CH1 =   353, CH2 =    13, CH3 =
48
 7:  CH0 =   323, CH1 =   403, CH2 =   161, CH3 =
73
 :
```

4.7.2 伝送エラーの原因解明

こういう時の対処法は二つあります。

1. 通信の仕様書やチップのデータシートの読み間違えがないか、確認する。
2. 現象を再現させ、現象を記録して、何が起こっているか推測する。

まず 1. として資料を読み直して誤解がないか調べました。SFR の設定間違いも疑ってみました。調べた限りでは、何のヒントも見つかりませんでした。

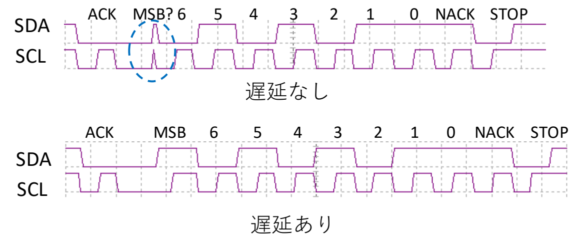
そのあとは 2. として、現象を再現する環境を作ります。次のプログラムは、Data0 レジスタを繰り返し読み出して、最初と違った値だったら停まるようになっています。いったん PICCOLO チップを止めて、*Variables* 表示から `Adata_register[0]` の値（初期値は 0x0000）を 0x89AB（下位バイトの MSB が 1 であればよい）に変更します。PICCOLO チップの動作を再開させ、I2C ライン（SDA と SCL）を観測します。

test_i2c_repeat.py

```
/* PIC/PICCOLO I2C 通信機能の検証プログラム ... レジスタの読み出し
  初版：2021/5/17
  最新版：
*/
#include "pic_board.py"
#define TEST_REG  8
def read_i2c(pi, reg):
  dat = pi.I2C_READ_WORD(piccolo, int(reg))
  dat = bswap(dat)
  sleep(0.2)
  return dat
pi = openPIC()
piccolo = pi.OPEN_I2C(PICCOLO_I2C)
d1 = read_i2c(pi, TEST_REG)
for i in range(1000):
  d2 = read_i2c(pi, TEST_REG)
  print('%3d:  Data = 0x%4x' % (i, d2))
  if d2 != d1:
    break;
pi.CLOSE_I2C(piccolo)
closePIC()
```

テストプログラムが止まったとき、最後の記録が異常を記録しているはずです。次の記録の上側（『遅延なし』）がその結果です。前のバイトの伝送が終わった（ACK）あと、クロックストレッチで SCL が消されているはずなのに、クロックパルスの最後の方が消えきれずに出てきているようです。その直後に SDA が L になり（ドライブしているのがマスター側かスレーブ側か分からない）、それが原因で伝送エラーが起きているらしいことが分かります。

試みにクロックストレッチを終わらせる（`SSP1CON1.CKP` をセットする）のを、数 MPU サイクルだけ遅らせてみます（下図の『遅延あり』）。こんどは SCL パルスが完全に一個分消えて、伝送が正常に行われます（1 ビット分だけ後ろにずれる）。



ビット化けを捕捉した記録（上）と、遅延で回避した記録（下）

MPU クロック周波数を 32MHz にすると、もっと顕著に発生するので、遅延ステップ数を増やしてやります。実際には、実験で正常動作を確認したより長めの遅延を取るようにしました。

重要な知見：「クロックストレッチは、SCL パルス 1 個分以上の時間が経ってから解除すること」

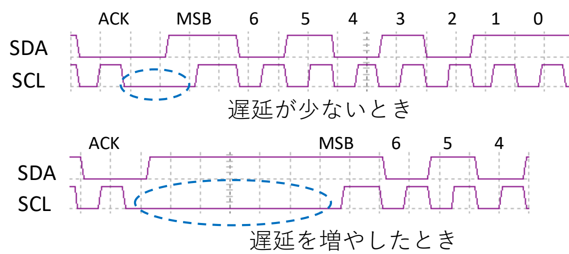
なぜこんな現象が起こったのか？

資料やインターネットを調べた限り、こんな現象の記述は見つかりませんでした。ここから先は推測です。

仮説:「マスターである Raspberry Pi は、スレーブがクロックストレッチを行ったことを、SCL の 1 パルス分経過した後でないと検知できない（しない理由があるのかもしれない？）。そのため、この時間より前にクロックストレッチを解除すると、SCL 検出の論理が狂ってしまい、伝送エラーを起こす（どちらが SDA を L に引っ張ってしまうのかは不明）」

そう考えると、前に出てきた「受信時にもクロックストレッチを行うと、『繰返し START』デリミタが発生しない」という問題も、同じ原因だと推測できます。

遅延をどんどん増やしていくと、2 パルス目も半欠け信号になってしまう懸念がありました。ところが、次の図で示すように、SCL パルスはデジタル的に消えていきます。解除後最初のパルス幅が少し広いのは、自動的に次のパルスとつなぎ合わせているせいだと思います。



SCL の 2 パルス目以後は、「パルス欠け」は起こらない

どうして、こんな現象が今まで知られていなかったのでしょうか？　先例を調べてみると、次のようなことが原因で、クロックストレッチの解除までに時間がかかっていたのではないかと推測できます。

A. PIC のクロック周波数を（消費電流低減のため）低く抑えることが多い
B. 送信データの準備に時間がかかっていた（今回は、あらかじめ応用層が用意していたデータを受け取るだけ）
C. 送信前に次のような手順を踏んでいた。SSP1STAT.BF がクリアされるまで待ち、送信データを SSP1BUFF へ書き込んだ後に衝突がなかったか SSP1CON1.WCOL を調べてから、クロックストレッチを解除していた

データシートを読む限り、SSP1STAT.BF は 8 ビット目（LSB）の転送終了時にクリアされ、割り込みが発生するのは ACK を受信したあとです。上の手順 C. は不要だと思うのですが、これで時間を稼いでうまくいったことから、そのまま使いまわされてきたのではないでしょうか。

4.7.3 最大負荷のイベント計数機能

伝送エラー問題が対処できたので、改めて負荷の影響を調べます。先に掲げた test_event.py を使い、負荷を最大にするため、4 入力チャンネルすべてをイベント計数モードにします。それぞれに異なる周波数のパルスを加えて結果を読み取ります。MPU クロック周波数が 16MHz のときの例を示します。

```
$ cpp -DCHANNEL4 test_event.py | python
Rising edge detection ----
 0:  CH0 =   40, CH1 =   51, CH2 =   20, CH3 =
25
 1:  CH0 =   80, CH1 =  101, CH2 =   40, CH3 =
50
 2:  CH0 =  121, CH1 =  152, CH2 =   60, CH3 =
76
 3:  CH0 =  161, CH1 =  202, CH2 =   80, CH3 =
101
 4:  CH0 =  201, CH1 =  253, CH2 =  101, CH3 =
126
 5:  CH0 =  242, CH1 =  303, CH2 =  121, CH3 =
151
 6:  CH0 =  282, CH1 =  353, CH2 =  141, CH3 =
177
 7:  CH0 =  323, CH1 =  404, CH2 =  161, CH3 =
202
 8:  CH0 =  363, CH1 =  454, CH2 =  181, CH3 =
227
 9:  CH0 =  403, CH1 =  505, CH2 =  201, CH3 =
252
10:  CH0 =  444, CH1 =  555, CH2 =  222, CH3 =
278
11:  CH0 =  484, CH1 =  606, CH2 =  242, CH3 =
303
12:  CH0 =  524, CH1 =  656, CH2 =  262, CH3 =
328
13:  CH0 =  565, CH1 =  707, CH2 =  282, CH3 =
353
14:  CH0 =  605, CH1 =  757, CH2 =  302, CH3 =
378
15:  CH0 =  645, CH1 =  808, CH2 =  323, CH3 =
404
16:  CH0 =  686, CH1 =  858, CH2 =  343, CH3 =
429
17:  CH0 =  726, CH1 =  909, CH2 =  363, CH3 =
454
 :
```

4 入力チャンネルがそれぞれ約 40、50、20、25 ずつ増えていくのが分かります。サンプリング間隔を sleep() で作っているので、精度が良くないのは当然です。通信エラーが起こらず、測定値が単調に増加していることが確認することが大切です。MPU クロック周波数を 32MHz にして、同じことを確認します。

次にイベント計数の精度を確認します。こんどはソフトウエアでパルスを与え、パルス数をきちんと計数できていることを確認します。時間がかかるので、試験は 2 チャンネルだけにしました。

test_num_event.py

```
/* PIC/PICCOLO 汎用測定機能（イベント計数）の検証プログラム -- イベント数指定
   初版：2021/5/27
   最新版：
*/
#include "pic_board.py"
def event_count():
  for i in range(100):
    slow_pulse(GPIO_EVENT0, 40)
    slow_pulse(GPIO_EVENT1, 50)
    e0 = pi.I2C_READ_WORD(piccolo, VERSATILE0)
    e0 = bswap(e0)
    e1 = pi.I2C_READ_WORD(piccolo, VERSATILE1)
    e1 = bswap(e1)
    print ('%2d:  CH0 = %4d, CH1 = %4d' % (i, e0,
e1))
    sleep(1)
pi = openPIC()
piccolo = pi.OPEN_I2C(PICCOLO_I2C)
pi.I2C_WRITE_BYTE(piccolo, A_COMMAND, 0) /* 初期化
*/
sleep(0.5)
print('Rising edge detection ----')
pi.I2C_WRITE_BYTE(piccolo, A_COMMAND, 0x0a)
event_count()
pi.I2C_WRITE_BYTE(piccolo, A_COMMAND, 0) /* 初期化
*/
sleep(0.5)
print('Both edge detection ----')
pi.I2C_WRITE_BYTE(piccolo, A_COMMAND, 0x0f)
event_count()
pi.CLOSE_I2C(piccolo)
closePIC()
```

結果の一部を次に示します。あたりまえと言われるかもしれませんが、40 と 50 ずつ増えていることを確認します。

```
$ ./run_python test_num_event.py
Rising edge detection ----
 0:  CH0 =   40, CH1 =   50
 1:  CH0 =   80, CH1 =  100
 2:  CH0 =  120, CH1 =  150
 3:  CH0 =  160, CH1 =  200
 4:  CH0 =  200, CH1 =  250
 5:  CH0 =  240, CH1 =  300
 6:  CH0 =  280, CH1 =  350
 7:  CH0 =  320, CH1 =  400
 8:  CH0 =  360, CH1 =  450
 9:  CH0 =  400, CH1 =  500
10:  CH0 =  440, CH1 =  550
 :
```

MPU クロック周波数は 16MHz で良さそうです。

4.8 PICCOLO チップとしての検証

検証の最後として、デバッガを外して動作を確認するため、MPLAB X IDE からベースチップに最終版ファームウエアを書き込みます。

PICKit4 を取り外して、Raspberry Pi と開発・評価ボードだけにします。ICSP 関連のジャンパは Stand Alone 側（プルアップ）に接続しておきます。

この状態で、この章で行った次の検証をすべて（モニター信号を使うものを除く）行えば、PICCOLO チップの開発は完了です。

検証項目	検証プログラム	備考
I2C 通信機能	test_i2c_register.py	
時間幅測定機能	test_width.py test_sync.py	分解能を変更
周波数測定機能	test_freq.py	ゲート時間 100ms/1s
イベント計数機能	test_event.py test_num_event.py	test_event.py では 4 チャンネル動作も試す
電圧測定機能	test_adc.py	

最終検証項目

検証が終わった PICCOLO チップは、自律走行車プロジェクトで使えます。

6. I2C 通信の概要

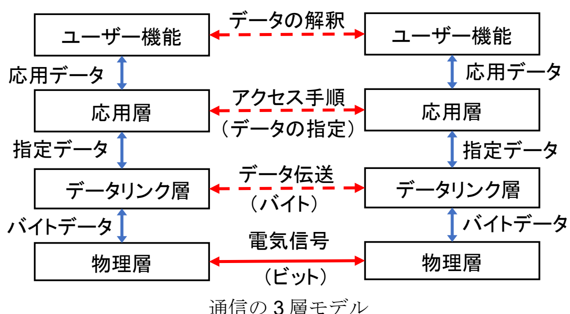
I2C 通信のプロトコル（通信規約）を、私なりの理解に基づいて説明します。

I2C バスは、オランダのフィリップス社（現在の NXP 社）が、基板上に実装された IC 間でデータ交換を行うために提案した技術（Inter-IC bus = I^2C bus）です。「マスター・スレーブ」方式と呼ばれる通信で、マスター（自律走行車の場合は Raspberry Pi）がスレーブ（ここでは PICCOLO チップ）に指令を送ることで、通信を実現しています。I2C バスの仕様は、NXP 社から「I^2C バス仕様およびユーザーマニュアル」として公開されており、最新版は Rev.6（2014 年）です。

もともとの I2C バスでは複数のマスターが存在できる（マルチマスターといいます）ようになっていますが、それに関しては一項目（『繰返し START』）以外、詳しい説明を省略します。

6.1 3 層モデル

I2C バスの説明では、いきなり波形やビットからデータの意味まで出てくることが多く、混乱してしまいます。私は下図のような層に分けて理解するようにしています。マスターとスレーブの中の機能をおのおの 3 層の通信機能とユーザー機能（データの作成・解釈などを行う）に分けて考え、それぞれの間で「何か」を交換することでユーザー機能を支えています。



通信の 3 層モデル

最下位の物理層は、電気信号を決める規則です。扱うのは電圧や負荷、波形、信号の本数などです。

その上のデータリンク層は、どのデバイスからどのデバイスにデータを送るのかを決める規則です。これに付随して、物理層に信号を送る手順（これを媒体アクセス制御といいます）も指定します。扱うデータの単位はバイトです。

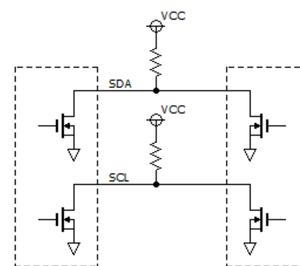
応用層は、どんなデータ（長さや構造）を送る（あるいは交換する）かを決める手順です。

その上にあるのは、I2C 通信の「ユーザー」とも呼ばれ、データを作ったり、解釈したりします。

では各層を下から順番に見ていきましょう。

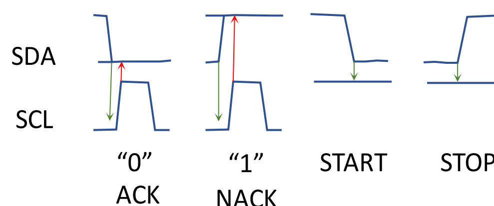
6.2 物理層

I2C 伝送路はオープンドレイン（またはオープンコレクタ）型の共用バスです。10kΩ以下の抵抗でプルアップされ、送信側はトランジスタをオンにすることで L レベルを送信します。オフにすれば H レベルになります。いちどに複数のトランジスタが送信しようとすると、おかしな波形になってしまいますが、これはデータリンク層で制御します。バスが使われていないときは全てのトランジスタがオフになっています（H レベル）。



オープンドレインバス

信号線は 2 本で、データ SDA とクロック SCL があります。ビット信号を送るとき、SDA は SCL が L レベルにある間に変化し、受信側は SCL の立ち上がりで読み込みます（下図の左側）。伝送の開始（START）と終了（STOP）は、SCL が H レベルにある間に SDA を変化させることで表現します（デリミタと呼ばれる信号ですが、I2C バスの慣習では『コンディション』と言います）。通常のクロック周波数（1 ビットの伝送速度）は 100kHz（100kbps）です。さらに高速の 400kHz や超高速（最大 3.4MHz）という仕様もありますが、伝送路に厳しい条件があります。



物理層の伝送信号

ACK と NACK は受信側が SDA を L レベルに引っ張る（あるいは引っ張らない）ことで、受信ができた（できなかった）ことを知らせる、データリンク層の信号です。これ以外に、スレーブ側が SCL を強制的に L レベルにする（クロックを「消して」しまう）ことで、応答の準備をする時間を稼ぐ、データリンク層の手順もあります。

データリンク層からの送信要求データや、受信通知データは、バイト単位でやり取りされます。物理層では、これを時系列のシリアルデータ（MSB が最初に伝送される）としてバスに載せます。

なお派生技術として、物理層の仕様を別の値にして、もっと長い距離を伝送できるようにした SM（System Management）バスなどもあります。

6.3　データリンク層

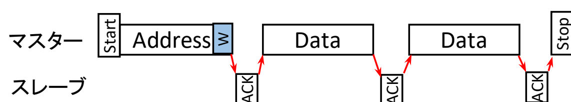
データリンク層では、START/STOP のデリミタと、バイト単位のデータと、その受信確認（ACK/NACK）を交換します。

START のあと、最初にマスターが伝送するデータは、アドレス情報（Address）です。このうち上位 7 ビットでスレーブのアドレスを、最下位ビットで読み出し／書き込みを指定します。資料によっては、7 ビットのアドレスに最下位ビットを付け加え、0x38（0b0111000）ではなく 0x70（0b01110000）がアドレスだと記述していることがあるので、注意してください。

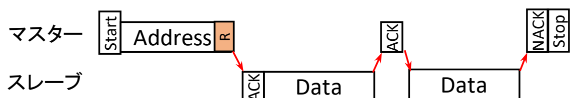
スレーブアドレスは 128 個（一部が特殊用途に指定されている）のうち、0x08～0x7B が使えます。10 ビットのアドレスもありますが、説明を省略します。

Address バイトの最下位ビットは、0（W）が書き込み、1（R）が読み出しを指定します

書き込み手順



読み取り手順



データリンク層プロトコル

スレーブにデータを書き込むときには、アドレス（＋W）を指定して、スレーブが ACK を返したら、最初のデータを送ります。送信が終わったら、STOP デリミタを送ります。

データを読み取るときは、R ビットをセットします。スレーブは ACK を返した後、バスにデータを送ります。データの準備に時間がかかる時は、SCL を強制的に L にして時間を稼ぎます（クロックストレッチという方法です）。伝送クロックは全てマスターが供給するのですが、それを「消す」ために、スレーブも SCL ドライブ回路を備えています。マスターは必要なデータを受信したら、「これ以上は要らない」という意味を示すため、ACK のかわりに NACK を返すのが一般的です。

マルチマスターの場合、他のマスターにバスを明け渡すのは、STOP です。ところが複数回の書き込み・読み出しの間に、他のマスターにバスを使わせないことが必要になる場合があります。そこで、STOP を送信せずに、もう一度 START から始めるという『繰返し START（repeated start）』という手順も仕様に含まれています。

I2C バス仕様では一般的な通信プロトコルと違って、伝送されるデータの長さ（バイト数）が表に出てきません。そのため、マスターが STOP しない限り、スレーブは 100 バイトだろうが、1M バイトだろうが、受信あるいは送信を行わなければなりません（NACK を使って以後の受信を拒否する手順はありますが）。データリンク層にとって、バイト数は上位の阿吽の呼吸で決められ、マスター側のデータリンク層に伝えられます。

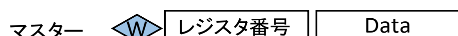
6.4　応用層

実をいうと I2C バスの規定には応用層が含まれていません。連続的に読み書きされるデータ列の解釈を、すべてユーザー機能に任せるのなら、それでいいわけです。事実、そういう実装をしている IC（例えば A/D 変換器 MPC3425 など）もあります。

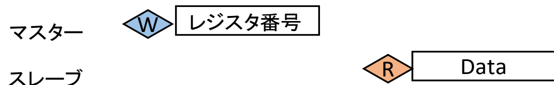
一般的な応用層では、スレーブ内に「内部レジスタ」を定義して、それに対して読み書きを行います。AQM シリーズの液晶表示器は特殊なプロトコルを持っていますが、「コントロールバイト」をレジスタ番号と解釈して使うことができました。

Raspberry Pi の I2C バスインターフェースを通してデータをデバイスへ読み書きする手順を応用層と解釈すると、プロトコルは次の図のようになります。

書き込み手順



読み取り手順



応用層プロトコル

書き込むときは、レジスタ番号とデータ（長さはレジスタで決まっている）を送りつけます。読み出すときは、まずレジスタ番号を送りつけ、次のデータリンク層の読み取り手順でレジスタのデータを読み出します。データリンク層の『繰返し START』は、この手順をまとめて（バスの使用権を他のマスターに渡すことなく）行うためのものです。

6.5 ユーザー機能

ユーザー機能で重要なのは、各内部レジスタのデータにどういう意味があるかということです。

スレーブ（PICCOLO）側では、仕様書に従って、Command レジスタの設定を解釈・実行します。また、測定結果を Status レジスタと測定値レジスタに反映します。

マスター（Raspberry Pi）側では、仕様書に従ってスレーブの機能を設定し、測定値を読み取って利用します。

エンディアン

ここで問題になるのが、CPU のエンディアンです。これは、複数バイトのデータをメモリに格納する方法のことです。インテル系の CPU では、アドレスの若い方に下位バイトを（リトルエンディアン）、モトローラ系の CPU では上位バイトを（ビッグエンディアン）収納します。

ビッグエンディアン

アドレス	
0	データ1の上位バイト
1	データ1の下位バイト
2	データ2の上位バイト
3	データ2の下位バイト

リトルエンディアン

アドレス	
0	データ1の下位バイト
1	データ1の上位バイト
2	データ2の下位バイト
3	データ2の上位バイト

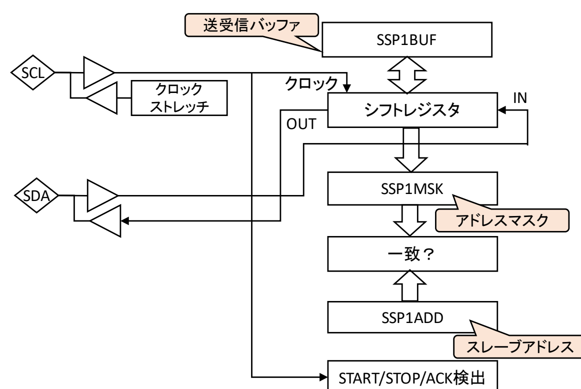
エンディアン（2 バイトデータの場合）

応用層は複数バイトのデータを伝送するとき、上位バイトを最初に送ります。受け取った側でその順番にメモリに書き込むと、ビッグエンディアンになっています。Raspberry Pi で採用している ARM コアのアーキテクチャはリトルエンディアンなので、バイトの入れ替えが必要です。pigpio ライブラリではワード（2 バイト）データを読み書きする機能がありますが、バイトの入れ替えはしてくれないので、アプリケーションで行う必要があります。ちょっと不親切な気がしますね。

6.6 PIC の I2C サポート

ベースチップとして採用した PIC16F15325 には、I2C 通信の物理層の全てとデータリンク層の一部を実行してくれるハードウエア（MSSP: Master Synchronous Serial Port）があります。一度設計した MSSP は、多くの PIC で繰り返し採用されているはずですが、ベースチップ以外の PIC を使う場合は念のためデータシートで確認してください。おそらく違いが出てくるのは、複数の MSSP を持った PIC に SSP1 だけでなく SSP2 があることくらいだと思いますが。

MSSP の機能概要を下図に示します。



ベースチップの MSSP 機能概要

SDA の受信ビット列はシフトレジスタでバイトデータに変換されてから、送受信バッファ（`SSP1BUF`）へ書き込まれます。送信時には送受信バッファのデータをシフトレジスタに書き込み、シフトアウトしたビット列を送信します。

受信データがアドレスのときは、アドレスレジスタ（`SSP1ADD`）に収納されているスレーブアドレスと比較して、一致したら割り込みを発生します。このとき一部のビットをマスクして、ある範囲のアドレスすべてを検出できるようにもできます。

ACK の発生と検出、START/STOP 検出による割り込みもハードウエアが行ってくれます。送信データを準備する時間を作る、クロックストレッチも行えます。

スレーブ動作時の MSSP からの割り込み要因には次のようなものがあり、それぞれのフラグを確認して処理します。PIR3.SSP1IF、PIR3.BCL1F、SSPCON1.SSPOV、SSP1STAT.BF 以外のフラグは、処理が終わればハードウエアが自動的にクリアしてくれます。

割り込み要因	表示されるフラグ	必要な処理
MSSP 割り込み	PIR3.SSP1IF	要因毎の処理
START 検出	SSP1STAT.S	処理不要
アドレス一致	SSP1STAT.DA/	バッファ読み取り
データ受信	SSP1STAT.BF	バッファ読み取り
データ送信完了	SSP1STAT.BF	バッファ書き込み
STOP 確認	SSP1STAT.P	通信終了
バス衝突	PIR3.BCL1F	送信終了
受信オーバーフロー	SSP1CON1.SSPOV	通信終了
送信データの上書き	SSP1CON1.WCOL	フラグのクリア

MSSP からの割り込み要因

ちょっと面倒なのが送受信バッファのステータス SSP1STAT.BF です。データが受信されたときは 1 にセット（バッファにデータがある）され、送信が終わったときはクリア（バッファが空になった）されます。いま送信中なのか（0 なら送信完了）、受信中なのか（1 ならデータ受信完了）によって、判断を切り替える必要があります。PICCOLO チップではステートマシンの状態で判断しています。が、SSP1CON レジスタの R/Ŵ ビットを使う方法もあります。

MSSP 自身が一つのステートマシンになっており、そのステートは SSP1CON レジスタに反映されています。このレジスタのビットパターンだけで動作を決める方法もあり、ネット上で見られる多くの実装例が採用しています。例えば液晶の表示メモリのような、比較的単純なデータ構造を扱うのには、これで充分だし、早い応答も望めます。

PICCOLO チップで、ファームウエアのステートマシンを作ったのは、応用層を意識したからです。書き込まれたデータに対して、内部のデータ構造を反映した処理を行うために、何バイト目のデータなのか知る必要があったのです。もっとも遷移条件の大部分が SSP1CON レジスタなので、MSSP のステートを『再現』しているのに近いというのが実情です。

仕様書　汎用測定チップ PICCOLO

概要

汎用測定チップ PICCOLO は、多様な入力を持つ低消費電力型の汎用入力デバイスである。パルス幅、パルス数、アナログ電圧を測定し、I2C バスを介して測定値を読み取ることができる。

特長

広い動作範囲

- 電源電圧：2.3V ～5.5V
- 動作温度：-40℃～85℃
- クロック内蔵（8MHz）

多様な入力

- 入力ピン数：6
- パルス幅測定（1 点）：16 ビット、分解能 2μs または 8μs
- パルス周波数測定（1 点）：16 ビット、ゲート時間は 10ms～1s またはコマンド書き込みによる
- イベント計数（4 点）：16 ビット
- アナログ電圧測定（4 点）：12 ビット分解能（0 ～2.048V）

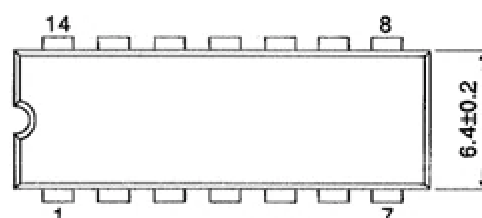
イベント計数とアナログ電圧測定は、入力ピンごとに一方を選択できる。

I2C 通信。

- I2C スレーブ（アドレス 0x38）
- 100kHz クロック、7 ビットアドレス

外形

- 14 ピン DIP パッケージ



端子配列

端子番号	端子記号	I/O	説明
1	VDD	-	電源（2.3V～5.5V）
2	PULSE	I	パルス入力信号（パルス数を測定する）
3	GATE	I	パルス幅信号（パルス幅を測定する）
4	MCLR/	P	（システム用）
5	IN3	I	汎用入力#3（イベントパルスまたはアナログ電圧）
6	IN2	I	汎用入力#2（イベントパルスまたはアナログ電圧）
7	IN1	I	汎用入力#1（イベントパルスまたはアナログ電圧）
8	IN0	I	汎用入力#0（イベントパルスまたはアナログ電圧）
9	SDA	I/O	I2C 通信データ SDA
10	SCL	I/O	I2C 通信クロック SCL
11	SYNC	I	パルス測定開始信号
12	ICSPC	P	（システム用）
13	ICSPD	P	（システム用）
14	GND	-	GND

（注）3 本のシステム用端子はデバイスのプログラムに使うもので、使用時には 10kΩでプルアップすること。

電気的特性

項目	最小値	推奨値	最大値	単位	条件
周囲温度（T_a）	-40		+85	℃	
電源電圧（V_{DD}）	2.5	3.3	5.5	V	
消費電流（I_{DD}）			1.7	mA	V_{DD} = 3.3V

測定機能

パルス幅測定機能

GATE 信号がアクティブ（H または L レベルから選択する。デフォルトは H）である時間を測定する。測定クロックは 125kHz（分解能 8μs）または 500kHz（分解能 2μs）から選択（デフォルトは 125kHz）できる。カウンタ長は 16 ビットで、オーバーフローがあっても下位 16 ビットは有効。オーバーフローは T-Status レジスタに反映される。Width データレジスタに収納された測定値を I2C 通信で読み出すことができる。T-Command レジスタの WE ビットに 1 が書き込まれたとき、測定値を初期化する。WE ビットに 0 を書き込むと測定は終了するが、それまでの測定値とステータスは保持されている。
測定動作には以下の 2 つのモードがある。

同期モード

同期モードでは SYNC 信号の後から測定が行われる。GATE 信号は何回アクティブになってもよい（測定値は加算される）。SYNC 信号の立ち上がりで測定値が Width データレジスタに収納され、カウンタは初期化される。この処理のため、SYNC 信号の立ち上がり後 50μs の間、GATE 信号をアクティブにしてはならない。

非同期モード

非同期モードでは、GATE 信号がアクティブでなくなったときに、測定値が Width データレジスタに収納され、カウンタは初期化される。この処理のため、GATE 信号が再びアクティブになるまで、50μs 以上の時間を置かなければならない。Width データレジスタには最新の測定値のみが保存されている。

パルス周波数測定機能

PULSE 入力信号の変化（L→H または H→L。デフォルトは L→H）回数を数える。測定時間は 10ms、100ms、1 秒、あるいは不定（I2C 通信で開始と終了を制御する）。Count データレジスタに収納された測定値を I2C 通信で読み出すことができる。T-Command レジスタの PE ビットに 1 が書き込まれたとき、測定値を初期化する。PE ビットに 0 を書き込むと測定は終了するが、それまでの測定値とステータスは保持されている。
測定動作には以下の 2 つのモードがある。

周波数モード
周波数モードでは、指定した測定時間（10ms（最大周波数 6.5MHz）、100ms（最大周波数 650kHz）、1 秒（最大周波数 65kHz）から選択）の間、PULSE 入力パルスを計数し、Count レジスタに結果を収納する。測定時間の間に、約 10ms の休止時間がある。

フリーカウントモード
フリーカウントモードでは、I2C 通信で指令された時間の間、PULSE 入力を計数する。計数値は 20ms 毎に Count データレジスタに転送される。書き込んだ PE ビットの処理に最大 50μs かかるため、書き込み後 50μs 以内にパルス信号を与えてはならない。

汎用入力測定機能

汎用入力 IN0～IN3 は、それぞれイベント（論理）信号、あるいは 0～2.048V のアナログ信号の入力端子として使うことができる（デフォルトではどちらも実行しない）。

イベント計数モード
イベント計数モードの汎用入力は、入力が変化（L→H、または両方の変化。デフォルトは L→H）した回数を 16 ビット長で計数する。コマンドレジスタの AEx ビットに 1 を書き込むと、計数値は 0 に初期化される。計数値は常に Data0～3 データレジスタに反映されるので、いつでも I2C 通信で読み出すことができる。AEx:ENx に 0:0 を書き込めば計数は止まるが、それまでの計数値やステータスは読み出せる。
計数は汎用入力を 250μs でサンプリングすることで行われるので、イベントの継続時間はこれより長くなければならない。

アナログ電圧測定モード
アナログ電圧測定モードの汎用入力は、12 ビットのデジタルデータに変換され、順次 Data0～3 データレジスタに収納される。測定は非同期で行われるので更新間隔は一定しないが、10ms を越えることはない。

内部レジスタ

レジスター覧

PICCOLOはI2C通信でアクセスできる16個の内部レジスタを持つ。Commandレジスタ以外は読み出し専用である。StatusレジスタとCommandレジスタは1バイトごとに読み書きするが、Dataレジスタは2バイト同時に（ワードとして）読み出すこと。これは、各バイトの読み出しの間にデータが更新されてしまうことを防ぐために必要である。

レジスタへの読み書きは、レジスタ単位で行うこと。アドレスが連続しているからといって、一度のI2C通信で複数のレジスタにアクセスしてはならない（データレジスタの2バイト読み出しを除く）。

アドレス	レジスタ	用途
0x00	T-Status	測定の状態を表示する
0x01	A-Status	
0x02	T-Command	測定機能を設定する
0x03	A-Command	
0x04	Width (high byte)	パルス幅測定の結果を収納する
0x05	Width (low byte)	
0x06	Count (high byte)	パルス計数の結果を収納する
0x07	Count (low byte)	
0x08	Data 0 (high byte)	汎用入力#0の測定結果を表示する
0x09	Data 0 (low byte)	
0x0A	Data 1 (high byte)	汎用入力#1の測定結果を表示する
0x0B	Data 1 (low byte)	
0x0C	Data 2 (high byte)	汎用入力#2の測定結果を表示する
0x0D	Data 2 (low byte)	
0x0E	Data 3 (high byte)	汎用入力#3の測定結果を表示する
0X0F	Data 3 (low byte)	

Status レジスタ

T-Status レジスタ（アドレス 0x00）	7	6	5	4	3	2	1	0
	—	—	OVP	RDP	—	—	OVT	RDT
デフォルト値（電源投入時）	0	0	0	0	0	0	0	0

ビット位置	デフォルト値	名称	説明
0	0	RDT	パルス幅測定の状態を示す 0：測定が実行（あるいは終了）していない 1：前回のデータ読み取り後に測定値が更新された
1	0	OVT	パルス幅測定データの有効性を示す 0：データは正常に測定された 1：測定中にオーバーフローがあった
[3:2]	00		未使用
4	0	RDP	パルス（周波数）測定の状態を示す 0：測定が実行（あるいは終了）していない 1：前回のデータ読み取り後に測定値が更新された
5	0	OVP	パルス（周波数）測定データの有効性を示す 0：データは正常に測定された 1：測定中にオーバーフローがあった
[7:6]	00		未使用

A-Status レジスタ（アドレス 0x01）	7	6	5	4	3	2	1	0
	OV3	RD3	OV2	RD2	OV1	RD1	OV0	RD0
デフォルト値（電源投入時）	0	0	0	0	0	0	0	0

ビット位置	デフォルト値	名称	説明
0	0	RD0	汎用入力#0 測定の状態を示す（アナログ電圧測定モードでのみ有効。イベント計数モードでは常に 0） 0：測定が実行（あるいは終了）されていない 1：前回のデータ読み取り後に測定値が更新された
1	0	OV0	汎用入力#0 測定データの有効性を示す 0：データは正常に測定された 1：測定中にオーバーフローがあった
2	0	RD1	汎用入力#1 測定の状態を示す（表示内容は#0 と同じ）
3	0	OV1	汎用入力#1 測定データの有効性を示す（内容は#0 と同じ）
4	0	RD2	汎用入力#2 測定の状態を示す（表示内容は#0 と同じ）
5	0	OV2	汎用入力#2 測定データの有効性を示す（内容は#0 と同じ）
6	0	RD3	汎用入力#3 測定の状態を示す（表示内容は#0 と同じ）
7	0	OV3	汎用入力#3 測定データの有効性を示す（内容は#0 と同じ）

Command レジスタ

T-Command レジスタ（アドレス 0x02）	7	6	5	4	3	2	1	0
	PP	PM1	PM0	PE	WC	WP	WM	WE
デフォルト値（電源投入時）	0	0	0	0	0	0	0	0

ビット位置	デフォルト値	名称	説明
0	0	WE	パルス幅測定の実行を選択する 0：測定を実行しない 1：測定を実行する
1	0	WM	パルス幅測定の動作モードを選択する。 0：同期モード 1：非同期モード
2	0	WP	パルス幅測定用ゲート信号の極性を選択する 0：H レベルのパルス幅を測定する 1：L レベルのパルス幅を測定する
3	0	WC	パルス幅測定の内部クロックを選択する 0：125kHz（8μs 単位） 1：500kHz（2μs 単位）
4	0	PE	パルス計数の実行を選択する 0：計数を実行しない 1：計数を実行する
[6:5]	00	PM1:PM0	パルス計数の動作モードを選択する 00：フリーカウントモード 01：周波数モード（ゲート時間 1 秒） 10：周波数モード（ゲート時間 100ms） 11：周波数モード（ゲート時間 10ms）
7	0	PP	パルス計数の入力極性を選択する 0：立ち上がりの回数を計数する 1：立ち下がりの回数を計数する

A-Command レジスタ（アドレス 0x03）	7	6	5	4	3	2	1	0
	AE3	EN3	AE2	EN2	AE1	EN1	AE0	EN0
デフォルト値（電源投入時）	0	0	0	0	0	0	0	0

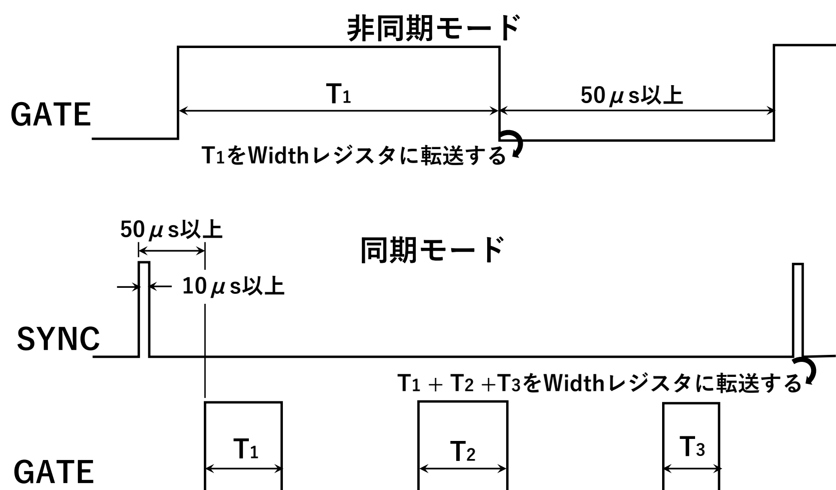
ビット位置	デフォルト値	名称	説明
[1:0]	00	AE0:EN0	汎用入力測定#0 の動作を選択する 00：入力測定を実行しない 01：アナログ電圧測定モードを実行する 10：イベント入力の立ち上がり回数を計数する 11：イベント入力の変化回数を計数する
[3:2]	00	AE1:EN1	汎用入力測定#1 の動作を選択する （設定内容は#0 と同じ）
[5:4]	00	AE2:EN2	汎用入力測定#2 の動作を選択する （設定内容は#0 と同じ）
[7:6]	00	AE3:EN3	汎用入力測定#3 の動作を選択する （設定内容は#0 と同じ）

データレジスタ

アドレス	データ名	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x04	パルス幅測定結果	WidthF～Width0															
0x06	パルス入力計数結果	CountF～Count0															
0x08	汎用入力#0 測定結果	Data0F～Data00															
0x0A	汎用入力#1 測定結果	Data1F～Data10															
0x0C	汎用入力#2 測定結果	Data2F～Data20															
0x0E	汎用入力#3 測定結果	Data3F～Data30															
デフォルト値（電源投入時）		0000 0000 0000 0000															

各測定値は 16 ビット長（電圧測定の時に限り 12 ビットで、上位ビットには 0 が入る）の整数で、測定中にオーバーフローが起こったときは Status レジスタの該当ビットがセットされる。

パルス幅測定機能のタイミング



あとがきと付録

あとがき

PIC は初体験という状態から、どうにか PICCOLO チップを作り上げることができました。この本はその過程をまとめたものです。いかにも最初からキチンと設計して、何の支障もなく出来上がったような印象を与えてしまったかもしれませんが、実際は調査→試行→問題分析→問題点修正というサイクルを何度もくりかえしたのです。データシートを読み直し、先行者の経験談を探しては、考え込む日々でした。その経験をもとに、同じ道を歩もうという人たちに役立ちそうなヒントをまとめておきます。

開発のステップ

ベースチップとして採用した PIC16F15325 のデータシート（英語版）は 558 ページもある膨大なもので、隅から隅まで読み通して、使いこなすことはできません。こんなアプローチを選びました。

1. 最初に PIC のアーキテクチャを知る： 素人らしく、まずレジスタ構成や割り込み、周辺ハードウエアの制御方法に関する知識を集めました。この段階では、データシートよりネット上のものを含む文献が役に立ちました。
2. チップの仕様と実現方法を検討する： データシートにある周辺ハードウエアの機能概要を調べ、何が PICCOLO チップの実現に使えるか検討しました。タイマーの使用は当然として、CLC とか NCO といった、なじみの薄いハードウエアが使えそうだと分かりました（実装設計 I)。
3. 実装設計をする： 使う予定のハードウエアだけ（データシートのページ数で 1/3 以下）に絞り、使い方と SFR の設定内容を決めました実装設計 III)。
4. プログラムは一気に書き下す： 実装設計に従って、C ソースコードを作成しました。しばらく使っていなかったせいで起こす文法まちがいは、開発環境のおかげで、すぐに解消できました。
5. シミュレーションで論理を検証する：機能の大部分を割り込み処理として実現しています。実際に割り込みが起こる条件を設定して、その動作を追うには、シミュレーションが適しています。ハードウエアの動作までは調べられないときでも、論理検証を済ませておくことで、次のステップが簡単になりました。
6. 実チップの中を覗く： PIC の動作を調べるため、内部の信号や状態をピンに出力して、ロジアナで観測しました。割り込みの応答や処理時間などを知るのに役立ちました。
7. Raspberry Pi から使う： Raspberry Pi から入力信号を与え、測定結果を読み取れるようにしました。動作の検証が簡単になりました。

皆さんにとって、何かの参考になれば幸いです。

応用範囲をどう広げるか

PICCOLO チップのように、上位プロセッサの手足として動作するチップの用途は、もっとありそうです。PIC の特長を生かすとすると、

1. Raspberry Pi より早い応答（1ms 以下）が求められる機能を PIC に処理させてから、結果をゆっくり取り出す
2. Raspberry Pi から少し離れた場所でデータ収集や操作をする
3. PIC だけなら超低消費電力であることを利用して監視をする

などです。具体的な用途は次のようなものが考えられます。

分類	用途
1	● ロータリーエンコーダーや回転センサの読取り ● パルスモーターの駆動と制御
2	● 筐体の温度監視と冷却ファンの回転数制御
3	● 電池の電圧監視と低下時のシャットダウン要求 ● 時間やセンサを監視し、必要な時に Raspberry Pi を起動する

PIC 応用チップの用途例

基本部と I2C 通信モジュール（他モジュールを呼ぶ部分などの改造は必要）はそのまま使い、新しいモジュールを用意すれば良さそうです。それぞれを初期化、割り込み処理、バックグランド処理から組み上げる方法は、そのまま踏襲すればいいので。

付録 1 自律走行車プロジェクト

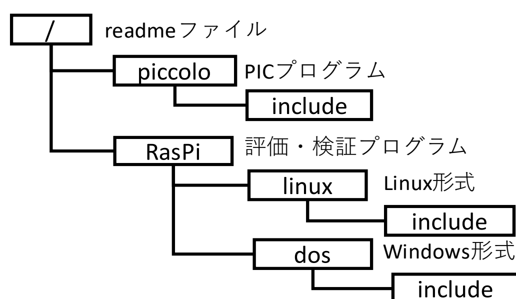
PICCOLO チップのユーザーである自律走行車プロジェクトについては、下の URL をご覧ください。。

https://www.akiyama-tokyo.net/electronics/raspbuggy.html

付録 2 プロジェクトファイル

このプロジェクトで作成したファイルをまとめてダウンロードできるようにしてあります。学習や研究目的であれば、自由に改造や再配布をしてもらって構いません。ただし著者は、プログラムの実行によって起こった、どのような事態にも責任は負いません。

ファイルは ZIP 圧縮されており、以下のようなディレクトリ構成になっています。piccolo ディレクトリには PIC 用の C プログラムが、RasPi ディレクトリには Raspberry Pi で使う評価プログラムが収納されています。RasPi の下にある linux ディレクトリには Raspberry Pi で動作するプログラムが、dos ディレクトリには、Windows 環境で読める形式に変換したファイルが収納されています。



配布ファイルのディレクトリ構成

プロジェクトファイルのダウンロードは以下から:

https://www.akiyama-tokyo.net/electronics/piccolo.zip

Raspberry Pi 派生プロジェクト

汎用測定チップ PICCOLO

PIC を使って専用チップを作る

2021 年 6 月

著者・発行者　秋山忠次 (chuji@akiyama-tokyo.net)

非売品

Raspberry Pi を使った応用を志す人は、本書の複写・複製・再配布を自由に行えます。ただし、無断で商業目的に利用することは、著者の権利侵害になります。

©Chuji Akiyama 2021