とことんAD変換
(dsPIC33FJ128MC802を用いて)

本稿掲載のWebページ

古橋　武

目次

第 5 章　とことん AD 変換 (dsPIC33FJ128MC802 を用いて) 3

5.1 組み立てと部品 3
5.1.1 組み立て 3
5.1.2 部品 5
5.2 MPLAB® X IDE と MPLAB® XC16 コンパイラ 10
5.3 タイマ 1 による割り込み 17
5.3.1 デバイスコンフィグレーション設定 17
5.3.2 オシレータシステム 19
5.3.3 メイン関数 23
5.3.4 タイマ初期設定関数 24
5.3.5 割り込み処理関数 28
5.3.6 タイマ割り込み処理プログラムのブロック図 30
5.4 I/O ポート 31
5.4.1 出力用ポート 31
5.4.2 入力用ポート 33
5.4.3 タイマ割り込み（ディジタル入力）処理プログラムのブロック図 38
5.5 DA 変換器の使用方法 39
5.5.1 PIC マイコンとの接続 39
5.5.2 DAC.x の設定と実行 39
5.5.3 DAC.c 40
5.5.4 DAConv() 関数 42
5.5.5 SPI モジュールの初期設定 45
5.5.6 DA 変換器 50
5.5.7 SPI の内部クロックを最大に 53
5.5.8 DA 変換プログラムのブロック図 53
5.6 AD 変換モジュール 55
5.6.1 動作例 55
5.6.2 タイマ 1 割り込みによる AD 変換のサンプリング周期設定 56

5.6.3 タイマ3によるAD変換のサンプリング周期設定と変換終了時割り込み　ー タイマ3による1 [kHz] サンプリング ー 67
5.6.4 Autoconversionによる高速サンプリング 71
5.6.5 Debuggerの利用 76
5.6.6 1.1[Msps]の実現 79
5.7 DMA 86
5.7.1 2CH同時サンプリング 86
5.7.2 2CH同時サンプリング＋4サンプル転送後割り込み 100
5.7.3 PIA Mode (2CH同時サンプリング＋4サンプル転送後割り込み) 104
5.7.4 転送データ半分での割り込み（BUFM = 1の試み） 109
5.7.5 Ping-Pongモード 113
5.7.6 Scanモード 118
5.8 索引 123

参考文献 **129**

第5章

とことんAD変換(dsPIC33FJ128MC802を用いて)

5.1 組み立てと部品

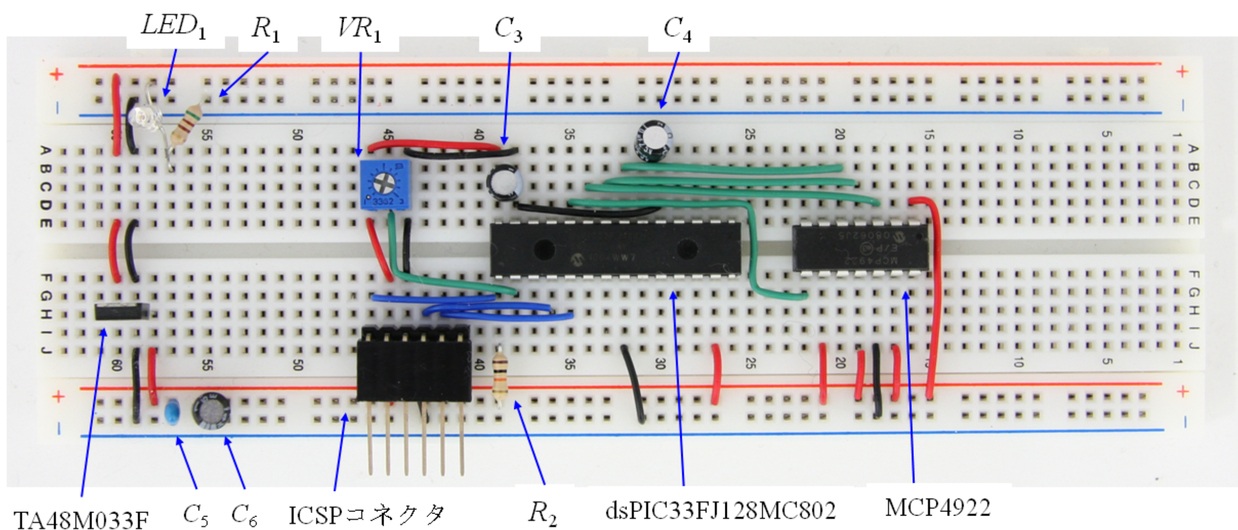


図 5.1: dsPIC33FJ128MC802 を用いた AD 変換モジュールの実験回路

5.1.1 組み立て

本節では AD(Analog/Digital) 変換モジュールの使用方法について解説する．図 5.1 はブレッドボード上に製作した実験回路である．PIC33FJ128MC802 を用いて，内蔵の AD 変換モジュールの実験を行うための回路である．MCP4922 は 12 ビットの DA 変換器であり，マイコン内の値のモニタ用である．本章にて新たに使用する部品は PIC マイコン

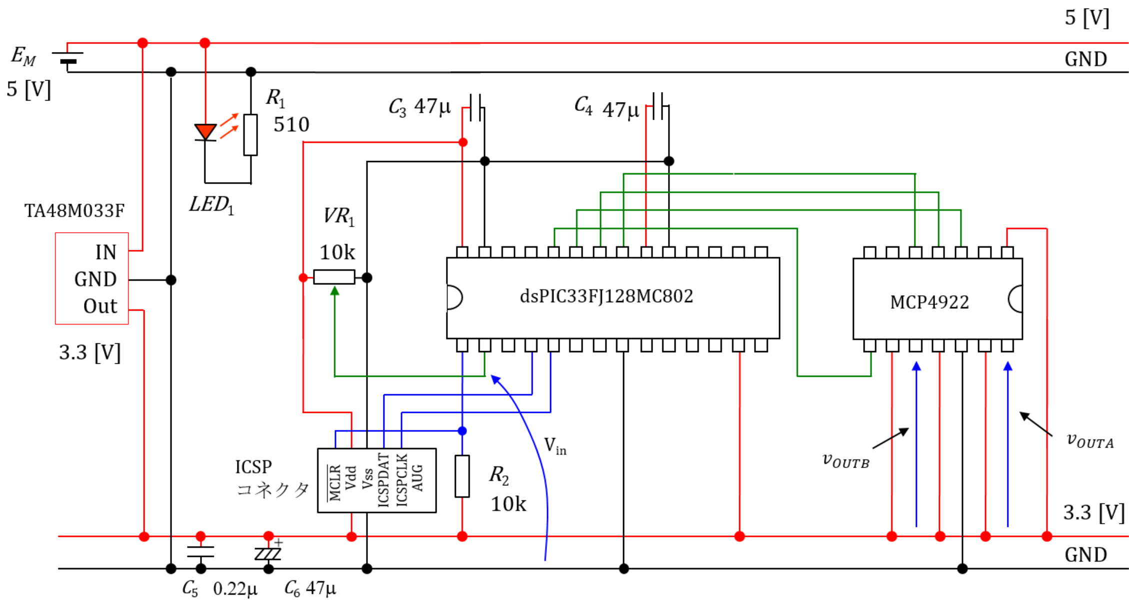


図 5.2: dsPIC33FJ128MC802 を用いた AD 変換モジュールの実験回路図

(PIC33FJ128MC802)，DA 変換器 (MCP4922)，3 端子レギュレータ (TA48M033F)，電解コンデンサ (47 [μF], 35 [V])，とブレッドボードである．

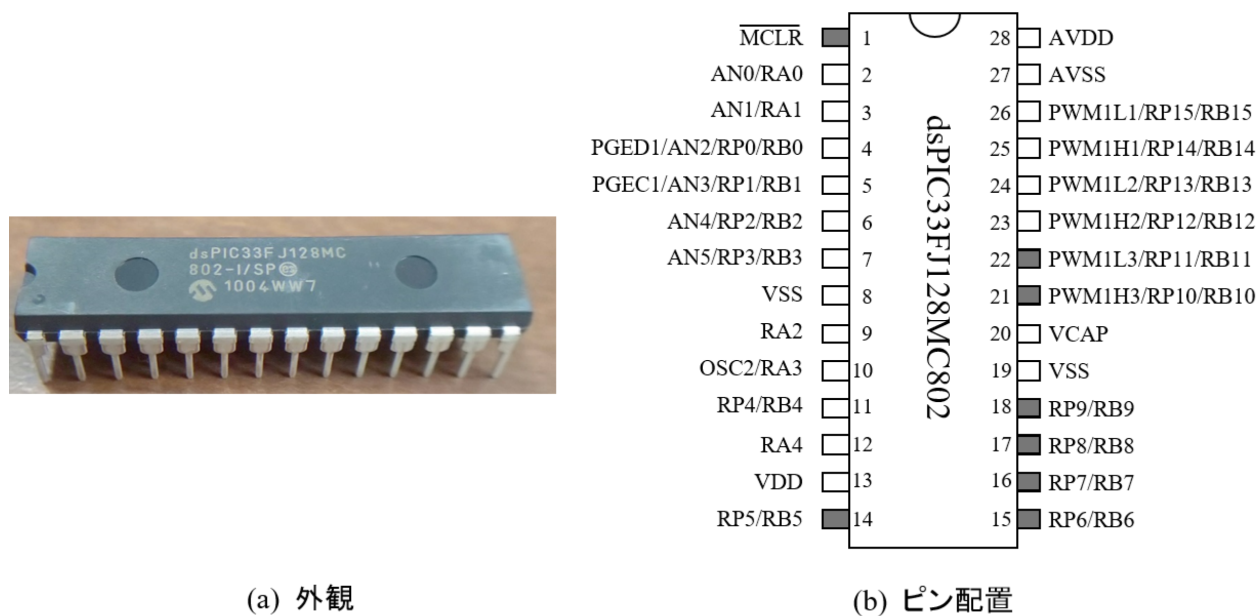
図 5.2 は実験回路の回路図である．表 5.1 にこの実験回路において新たに使用する部品を示す．表には平成 30 年 10 月時点での筆者の部品購入先を記してある．ただし，送料は含まれていない．いずれもネット通販である．電源には充電池 4 本を使用して，約 5 [V] の電圧を得ている．乾電池 3 本で 4.5[V] としても問題なく動く．以降，回路内の各部品について順次解説し，その後に AD 変換モジュールの実験用プログラムについて説明する．

表 5.1: 部品表

品名	型式	個数	単価	値段	入手先の例
PICマイコン	PIC33FJ128MC802-I/SP	1	701	701	チップワンストップ
12ビットD/A変換器	MCP4922	1	250	250	秋月電子通商
3端子レギュレータ	NJU7223F33, 3.3V, 500mA	1	50	50	〃
抵抗	100Ω, 1/4W 100個入り	1	100	100	〃
電解コンデンサ	47 μF, 35 V	3	10	30	〃
ブレッドボード	EIC−102J	1	650	650	〃
			総計	1781	円

5.1.2 部品

PIC マイコン–PIC33FJ128MC802–



(a) 外観　(b) ピン配置

図 5.3: dsPIC33FJ128MC802 のピン配置

図 5.3 には dsPIC33FJ128MC802 の外観とピン配置を示す．このマイコンは 28 ピンからなる．ピン配置図には PIC33FJ128MC802 のピンの機能のうち，本章の AD 変換と本稿のモータドライブに関係のあるものを記してある．VDD には＋ 3.3V の電圧源を接続し，VSS にはグラウンド（GND) を接続する．また，AN0/RA0 のように/記号で併記された機能は，プログラムの設定により使い分けできることを意味する．RPx (x ＝ 0〜15) は Remappable Pin と呼ばれるピンであり，Input Capture, Output Compare, QEI, UART, SPI 等の入出力端子としてソフトウェアにより使い分けることができる．詳細は Microchip 社の Web ページから無料でダウンロードできるデータシート “PIC33FJ128MC802 Data Sheet” の Peripheral Pin Select を参照されたい．出力ピンについては表：OUTPUT SELECTION FOR REMAPPABLE PIN に対応が記載されている．入力ピンについては表：SELECTABLE INPUT SOURCES (MAPS INPUT TO FUNCTION) に入力信号名と RPx 設定用レジスタとの対応が記載され，RPINRx レジスタ (PERIPHERAL PIN SELECT INPUT REGISTER) により入力信号を RPx のいずれに割り当てるかを設定できる．黒く塗られた端子は 5 [V] の信号を受け付けることができる．このマイコンの定格

電圧は3.3 [V] であるが，ディジタル回路の多くが5 [V] 電源を使用しているので，5 [V] 信号を電圧変換することなく，直接接続できる端子が設けられている．

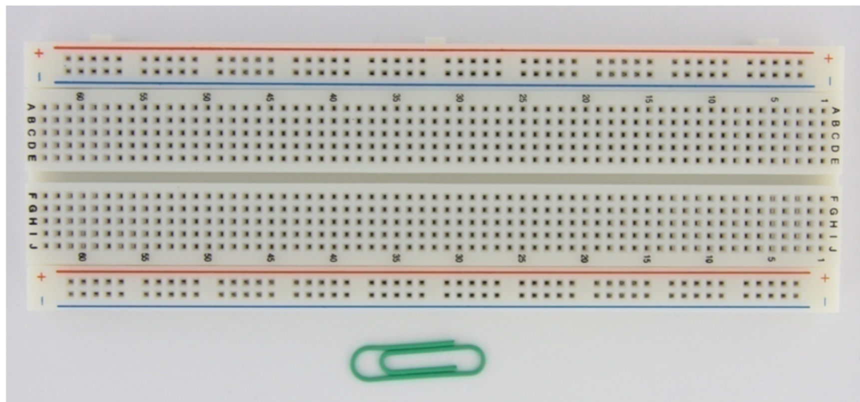
図　5.4: 3端子レギュレータ

3端子レギュレータ

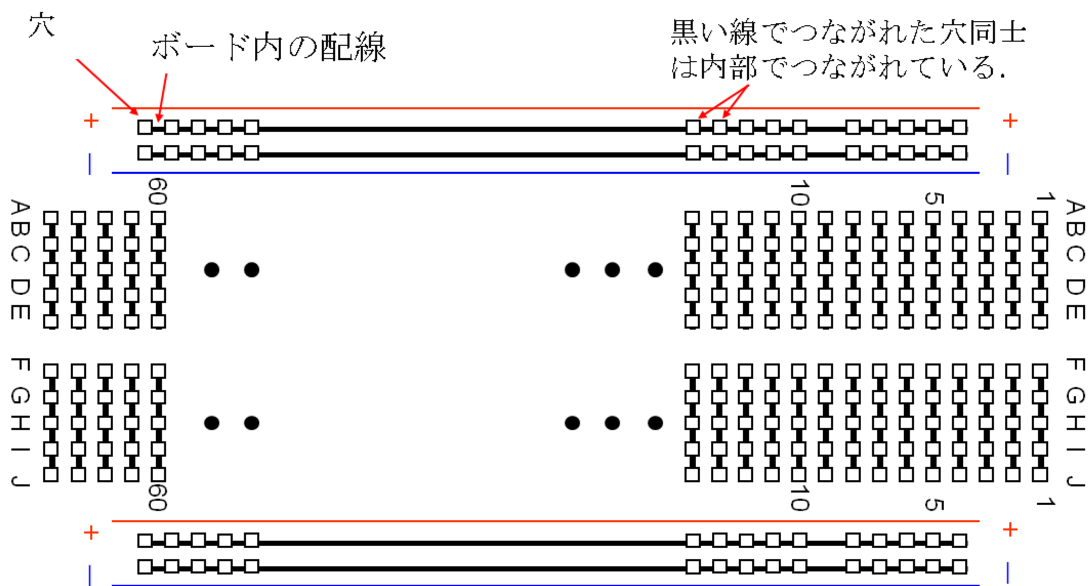
図5.4は3端子レギュレータ TA48M033F の外観とピン配置を示す．データシートによると，出力電圧3.3 [V], 最大入力電圧29 [V]，最大出力電流0.5 [A] である．本稿で用いるマイコンの定格電圧は3.3[V] であるため，電池の組み合わせでこの電圧を実現することはできない．そこで，この3端子レギュレータを用いて3.3[V] の電圧を得ている．IN 端子に電池の＋側を接続し，GND 端子に電池の-側を接続する．　Out 端子をマイコンの VDD 端子（13番ピン）に接続する．

ブレッドボード

図5.5は本章から用いているブレッドボードの外観と穴同士の配線の様子を示す．基本的には図1.24のブレッドボードと，穴の数が異なるだけで，同じ構造である．□の記号が穴であり，穴同士のボード内部のつながりの様子を黒い線で示してある．一番上の2行と一番下の2行の穴はそれぞれ横同士でつながっている．これらは，図5.1，図5.2のように，電源ライン，GND ラインとして使い分けると，回路の配線を電源への配線，GND への配線を簡潔にすることができる．



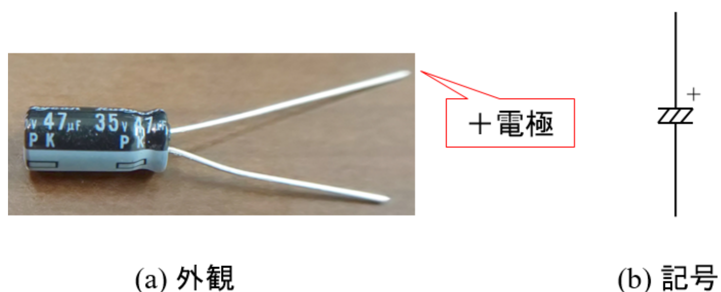
(a) ブレッドボード



(b) ブレッドボードの穴のつながりの様子

図 5.5: ブレッドボード

電解コンデンサ

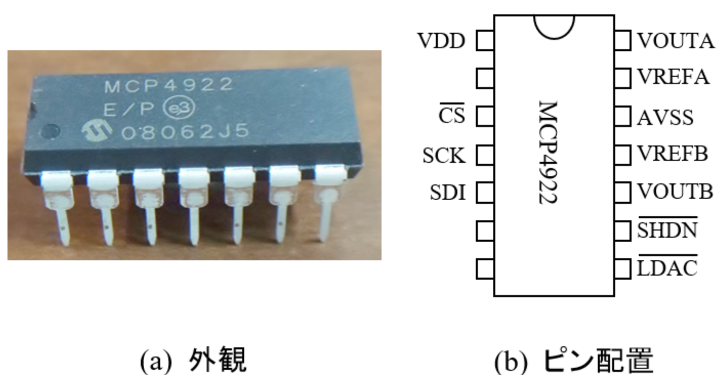


(a) 外観　(b) 記号

図　5.6: 電解コンデンサ

図 5.6 は電解コンデンサの外観写真と記号である．図は 47 [μF]，35 [V] の電解コンデンサである．電解コンデンサには極性がある．足の長い方が+電極である．逆極性に電圧を印加しておくと爆発することがあるので注意を要する．記号には極性を表す+の印がついている．

12 ビット DA 変換器



(a) 外観　(b) ピン配置

図　5.7: 12 ビット DA 変換器

図 5.7 は 12 ビット DA 変換器 MCP4922 の外観写真とピン配置である．電気的仕様の詳細は Microchip 社の Web ページから無料でダウンロードできるデータシート”MCP4922”を参照されたい．PIC マイコンとの接続法，プログラムの詳細は後述する．

ICSP コネクタ

ICSP(In-Circuit Serial Programming) コネクタの製作と PICkit3 の接続方法については本稿第 1 章の図 1.30, 1.31 を参照されたい.

5.2 MPLAB® X IDE と MPLAB® XC16 コンパイラ

本モータドライブノートの第 1 章では統合開発環境に MPLAB® IDE を用い，8 ビットマイコン用の HI-TECH C コンパイラを用いた．第 1 章を書き終えてから 6 年の間が空いてしまった．こういうのを間抜けというのでしょうが，途中まで書いてあったので，思い切って最後まで書き上げて公開しようと思います．どなたかの目にとまって，お役に立てれば幸いです．

本稿では新しい統合開発環境である MPLAB® X IDE (Integrated Development Environment：統合開発環境へと移行し，dsPIC 用のコンパイラとしてはレガシィとなった MPLAB® C30 コンパイラから，最新の MPLAB® XC16 コンパイラへと切り替える．そこで，本節では MPLAB® X IDE の使い方を解説します．知ってるよという方は本節を飛ばしてください．

MPLAB® X IDE は Microchip 社のホームページ → Design → Development Tools → Software Tools for PIC MCUs and dsPIC DSCs → MPLAB® X IDE → Downloads → MPLAB® X IDE v5.05 (2018 年 10 月時点) とたどることでインストーラ (MPLABX-v5.05-windows-installer.exe) をダウンロードできる．このインストーラを立ち上げ，インストーラの推奨通りに Next ボタンを押していくことで，MPLAB® X IDE をインストールできる．無事インストールに成功すれば，C:¥Program Files (x86) のフォルダ内に Microchip という名前のフォルダ，ドキュメントフォルダ内に MPLABXProjects という名前のフォルダが作られている．

同様に，MPLAB® XC16 コンパイラは Microchip 社のホームページ → Design → Development Tools → Software Tools for PIC MCUs and dsPIC DSCs → MPLAB® XC Compilers → Downloads → MPLAB® XC16 Compiler v1.35 (2018 年 10 月時点) とたどることでインストーラ (xc16-v1.35-full-install-windows-installer.exe) をダウンロードできる．このインストーラを立ち上げ，推奨通りに Next ボタンを押していくことで，XC16 コンパイラをインストールできる．無事インストールに成功すると，Microchip フォルダ内に xc16¥v1.35 という名前のフォルダが作られている．

次に，XC16 コンパイラと同じページにある 16-bit dsPIC33, PIC24E, PIC24H MCUs: Legacy Peripheral Libraries のインストーラをダウンロードし，Peripheral Libraries を v1.35 フォルダ内にインストールする．これにより，adc.h, timer.h, spi.h などのヘッダファイルを入手できる．

本稿では，v1.35¥support¥dsPIC33F¥h フォルダ内のヘッダファイル p33FJ128MC802.h と，v1.35¥support¥peripheral_30F_24H_33F フォルダ内のヘッダファイル adc.h, dma.h, spi.h, timer.h を使用する．

本稿で解説するタイマ 1 による割り込み，AD 変換器等のソースコードを，本稿と同じ
モータドライブノート
に圧縮フォルダに入れて掲載してある．この圧縮フォルダには利用するヘッダファイルも入れてある．これらヘッダファイルの中身は，読みやすさを重視して，Microchip 社のオリジナルのヘッダファイルから daPIC33FJ128MC802 が使用する部分のみを抜き出してある．

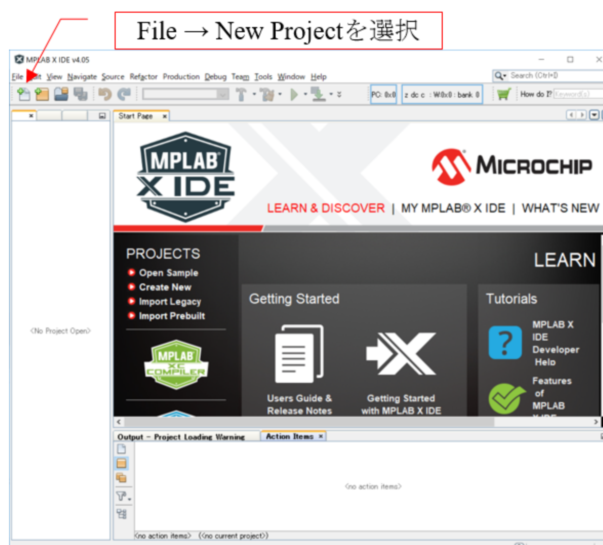


図 5.8: MPLAB® X IDE の立ち上げと New Project の設定

本節では，次節で詳述するタイマ 1 による割り込みに関連するファイルを用いて，MPLAB® X IDE による編集，マイコンへの書き込み方法について記す．

MPLAB® X IDE のアイコンを左ダブルクリックすることで，この統合開発環境を立ち上げることができる．図 5.8 の画面が立ち上がったら，File → New Project と選択する．次に図 5.9 のように進み，デバイスとして dsPIC33FJ128MC802 を選択する．

その後は図 5.10 のように書き込み，デバッグツールとして PICkit3 を選択し，コンパイラに XC16(vx.xx).... を選択する．図 5.11 は次に表示される画面である．あらかじめ作っておいたフォルダ（例えば，MPLABXProjects フォルダ内に dsPIC33FJ128MC802 という名前のフォルダを作っておく）をブラウズし，プロジェクト名を自分で決めて（例えば，Timer1_interrupt とする）入力し，”Set as main project”にチェックを入れて，言語に Shift_JIS を選択する．

以上が完了した段階で，（上の例では dsPIC33FJ128GP802 のフォルダの中に Timer1_interrupt という名前の）フォルダが作られている．図 5.12 のように，このフォルダの中へ

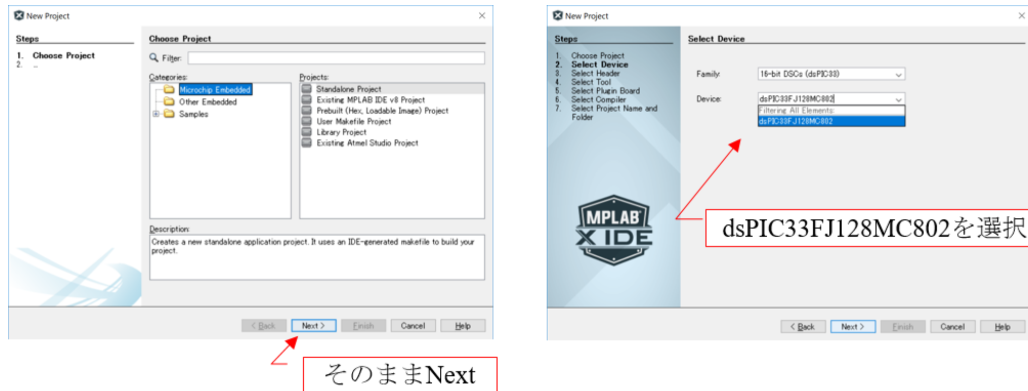


図 5.9: MPLAB® X IDE: Device 選択

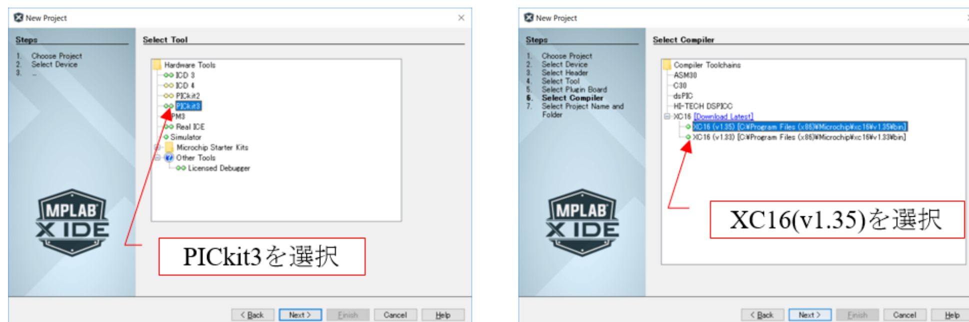


図 5.10: MPLAB® X IDE: tool, compiler 選択

モータドライブノートからダウンロードしておいた「タイマ１による割り込み」フォルダ内のプログラムのソースファイル (Timer1_interrupt.c, timer1.c) およびヘッダファイルの入っているフォルダ (include) をコピーする．

次に，これらのファイルをプロジェクトに追加する．図 5.13 のように，Source Files のフォルダを右クリックし，ADD Existing Item を選択する．そして，同図右のように追加したいソースファイルを選択する．次に，図 5.14 の画面のように，Header Files のフォルダを右クリックして，ヘッダファイルを追加する．

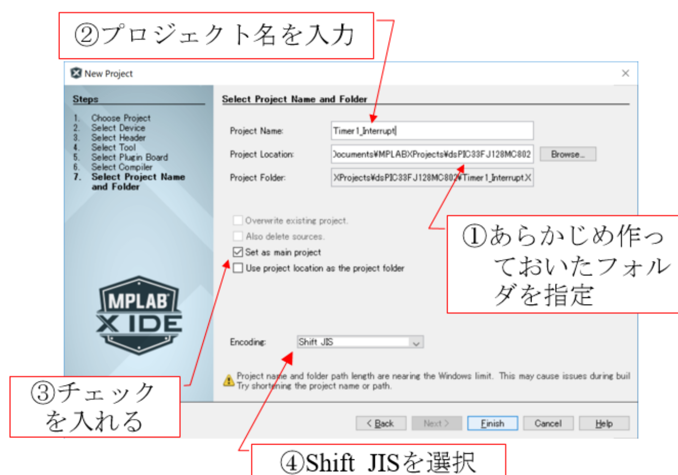


図 5.11: MPLAB® X IDE: project name, folder, 言語選択

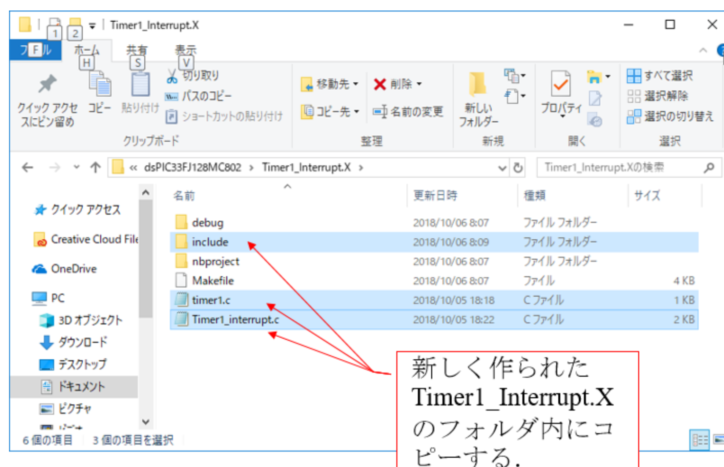


図 5.12: MPLAB® X IDE: ソースファイル，ヘッダフォルダのコピー

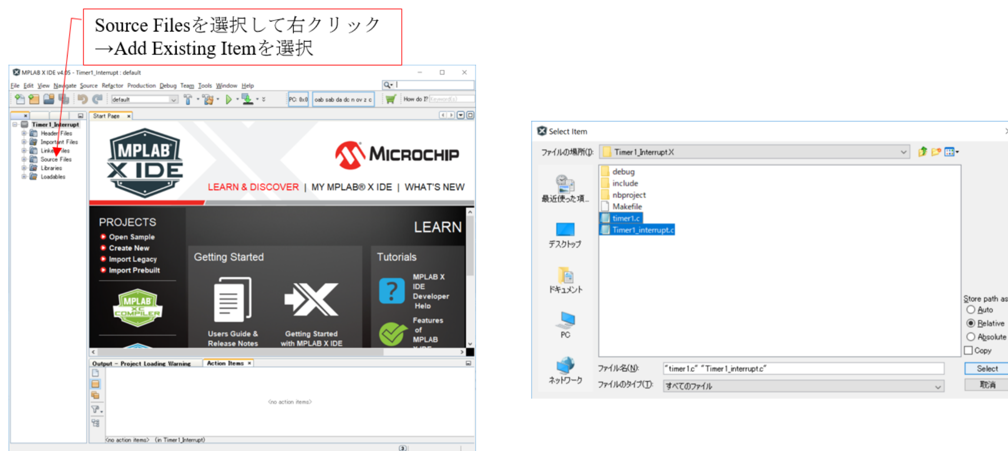


図 5.13: MPLAB® X IDE: Source の追加

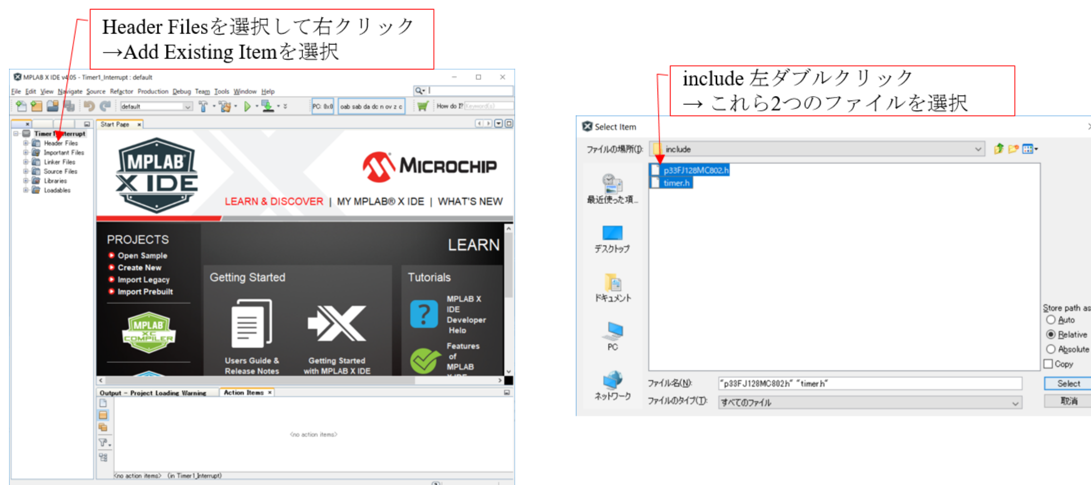


図 5.14: MPLAB® X IDE: Header の追加

以上で MPLAB® X IDE によるプログラムの編集，PIC マイコンへの書き込み準備が完了した．ブレッドボードの電源ラインに 4 本の乾電池 (6[V]) もしくは充電池 (5[V]) の電源を接続して電圧を印加し，ICSP コネクタに PICkit3 を接続してパソコンと USB ケーブルで接続する．そして，図 5.15 のように，画面内の Timer1_Interrupt.c を左ダブルクリックすることで，このファイル内のプログラムを開くことができる．そして，Make and Program Device Main Project ボタンをクリックすれば，同図下の表示が現れて，マイコンの 12 番ピンには図 5.16 に示す波形が出力される．このプログラムはタイマ 1 により 1[kHz] の繰り返し周波数で割り込みモジュールが起動され，同モジュールは RA4 ポートに 1(12 番ピンに約 3 [V]) を出力する．

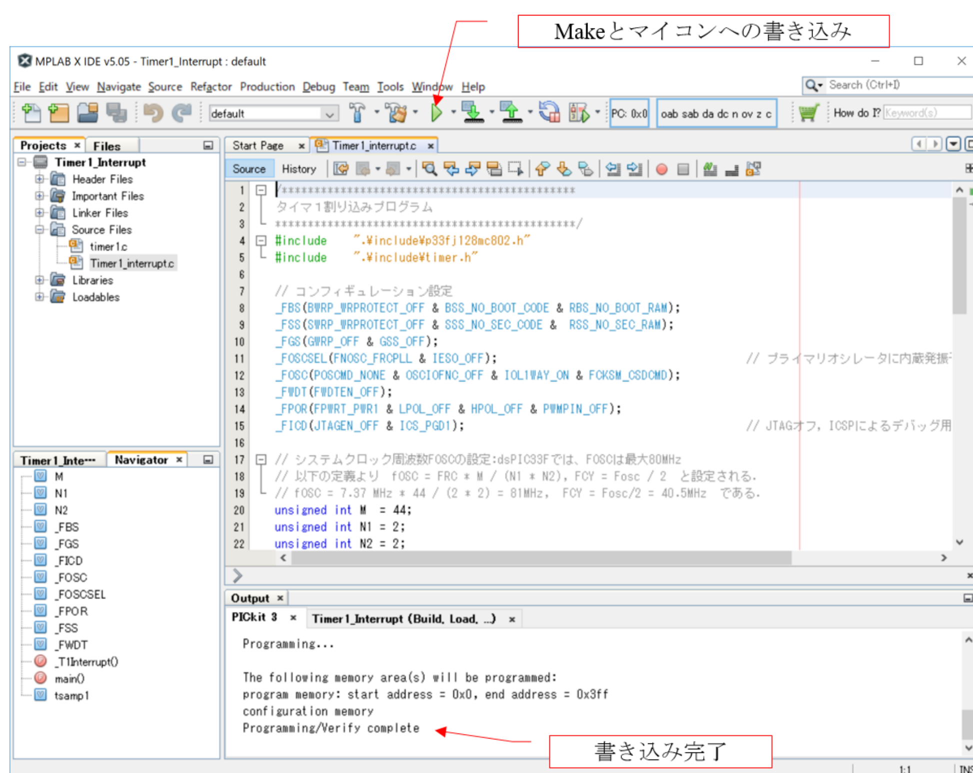


図 5.15: MPLAB® X IDE: Make とマイコンへの書き込み

図 5.16 はオシロスコープ画面のスナップショットであり，画面の横軸は 25[ns/div]，縦軸は 1 [V/div] である．画面の右下には観測波形の繰り返し周波数が 1.00401[kHz] と表示されている．これはタイマ 1 による割り込み周波数の計測値である．また，RA4 ポートに出力されるパルスの幅 (約 3[V] の電圧が出力されている期間）は約 25[ns] である．このプログラムでは，RA4 ポートに 1 を出力する命令を実行した直後に同ポートに 0 を出力する命令を実行している．12 番ピンに約 3[V] の電圧が出力され，0 を出力する命令の

実行直後に 12 番ピンに 0[V] の電圧が出力される．パルス幅より 1 命令の実行所要時間が約 25[ns] であることが分かる．データシートによると Device Instruction クロック周波数 (Device Operating 周波数とも呼ばれている）FCY = FOSC/2 である．1 命令の実行時間は 1/FCY = 1/40.5 [MHz] ≈ 25 [ns]) であり，観測結果はこの値と一致する．

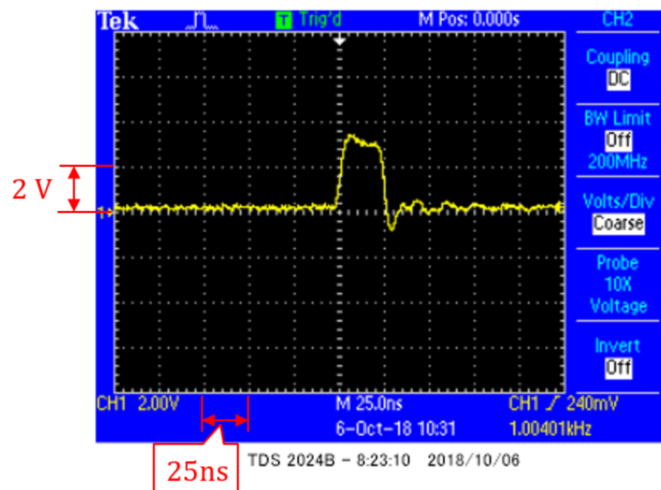


図 5.16: タイマ 1 割り込みプログラムによる 12 番ピンの出力波形

5.3 タイマ１による割り込み

5.3.1 デバイスコンフィグレーション設定

本節では，前節で設定したプロジェクト Timer1_Interrupt を用いて解説を進める．

データシート (PIC33FJ128MC802 Data Sheet) の TIMER1，TIMER2/3 and 4/5 によると PIC33FJ128MC802 には 16 ビットタイマモジュール (Timer1 〜 6) がある．このうち Timer2 と 3，Timer4 と 5 はそれぞれ連結されて 32 ビットタイマとして使うことができる．本節では Timer1 を用いて，1[msec] の周期で割り込みを行うプログラムを示す．割り込みとは，マイコンがあるプログラムを実行中に，そのプログラムの実行を一時中断させて，別のプログラムを実行させることを言う．マイコンに割り込み要求が入ると，マイコンはそれまでのプラグラムの実行を中断して，要求のあった別のプログラムを実行し，その割り込みプログラム終了後に，中断していた元のプログラムの実行を再開する．

```
#include     ".¥include¥p33fj128mc802.h"
#include     ".¥include¥timer.h"

// コンフィギュレーション設定
_FBS(0xFFFF);
_FSS(0xFFFF);
_FGS(0xFFFF);
_FOSCSEL(FNOSC_FRCPLL);          // プライマリオシレータに内蔵発振子を利用し，内蔵PLLを利用する．
_FOSC(OSCIOFNC_ON);              // OSC2 (10番ピン) を通常のI/Oピンとする．
                                 // OFFとするとクロックが10番ピンから出力される．
_FWDT(FWDTEN_OFF);               // ウォッチドックタイマ を SWDTEN ビットによりオン/オフする設定．
                                 // _SWDTEN = 0 でウォッチドッグタイマオフ
_FPOR(FPWRT_PWR128);             // Power-on Reset Timer 128ms
                                 // 電源投入(VDDが立ち上がって)から128msの間，マイコンをリセット
                                 状態にして，内部回路のバイアス電圧などが安定するのを待つ．
_FICD(JTAGEN_OFF & ICS_PGD1);    // JTAGオフ，ICSPによるデバッグ用端子をPGD1(4番ピン)に設定
```

図 5.17: デバイスコンフィギュレーション設定

図 5.17 はヘッダファイルのインクルードとデバイスコンフィギュレーション設定を示す．インクルードするヘッダファイルは p33fj128mc802.h と timer.h である．p33fj128mc802.h は XC16 コンパイラをインストールした際に

C:¥Program Files (x86)¥Microchip¥xc16¥v1.35¥support¥dsPIC33F¥h

フォルダに自動的にダウンロードされている．また，timer.h は XC16 コンパイラのパッチファイル（XC16 コンパイラと同じ Web ページにある）をインストールすると，

C:¥Program Files (x86)¥Microchip¥xc16¥v1.35¥support¥dsPIC33F¥h

内にダウンロードされているので，これらを用いることが出来る．本稿では，筆者が事

前にこれらヘッダファイルを「タイマ1による割り込み ¥include」フォルダ内にコピーしておいたものを用いる例を紹介している．

インクルードするヘッダファイルは Timer1_interrupt.c と同じフォルダ内に include と名付けられたフォルダ内にある．この場合，先頭にピリオドを付けることで，Timer1_Interrupt と同じフォルダ内のフォルダ include を指定できる．各ヘッダファイルの内容については以降必要に応じて触れていく．

デバイスコンフィギュレーションレジスタおよびレジスタ内の各ビットの名前は，データシートの DEVICE CONFIGURATION REGISTER MAP に定義されている．例えば，コンフィギュレーション設定の先頭の _FBS() は，FBS レジスタの値を設定するマクロであり，p33fj128mc802.h のヘッダファイル内で定義されている．設定項目はたくさんあるが，その多くをオフにしてある．ここで重要なのは，FOSCSEL レジスタである．データシートによると，FOSCSEL レジスタ内の第 0〜2 ビットは FNOSC bits と名付けられている．そして，dsPIC33F CONFIGURATION BITS DESCRIPTION によると，FNOSC bits はオシレータソースを選択するビットであることが分かる．p33FJ128MC802.h のヘッダファイルを開くと，

$$\#define\ FNOSC_FRCPLL \quad 0xFFF9 \tag{5.1}$$

と定義されている．0xFFF9 は 16 進数表記であり，2 進数表記では，この数値は 0b1111 1111 1111 1001 である．下 3 桁は 001 であり，データシートの dsPIC33F CONFIGURATION BITS DESCRIPTION から

001 = Internal Fast RC (FRC) oscillator with PLL

である．コンフィギュレーション設定において，

_FOSCSEL(FNOSC_FRCPLL)

と記述することで，マイコンのプライマリオシレータに内蔵発振子を使用し，さらに内蔵 PLL の使用を指定する．

FOSC レジスタの第 2 ビットは OSCIOFNC bits (OSC2 Pin Function bit) と名付けられている．p33fj128mc802.h のヘッダファイルにて

$$\#define\ OSCIOFNC_ON \quad 0xFFFB \tag{5.2}$$

と定義されている．B = 0b1011 なので第 2 ビットは 0 であり，データシートの dsPIC33F CONFIGURATION BITS DESCRIPTION から

0 = OSC2 is general purpose digital I/O pin

である．図 5.3 より，OSC2 は 10 番ピンである．OSC2 を汎用 I/O ピンとして使用する設定である．もし，OSCIOFNC_OFF とすると，

1 = OSC2 is clock output
とあるので，10 番ピンからマイコンのクロック信号を出力する設定となる．
FWDT レジスタの第 7 ビットは FWDTENbit と名付けられている．ヘッダファイルでは

$$\#define\ FWDTEN_OFF \quad 0xFF7F \tag{5.3}$$

とあり，第 7 ビットを 0 にしている．データシートより，ウォッチドッグタイマを RCON レジスタの SWDTEN ビットにより起動／停止できる設定である．本稿ではウォッチドッグタイマは使用しないので，図 5.21 の main() 関数にて

_SWDTEN = 0;

と記述することで，ウォッチドッグタイマをオフにしている．
FPOR レジスタの第 0～3 ビットは FPWRTbits である．

$$\#define\ FPWRT_PWR128 \quad 0xFFFF \tag{5.4}$$

により，111 = Power-on Reset Timer 128 ms と設定している．電源投入（VDD が立ち上がって）から 128ms の間，マイコンをリセット状態にして，内部回路のバイアス電圧などが安定するのを待つ設定である．
FICD レジスタの第 0，1 ビットは ICS bits(ICD Communication Channel Select bits) と名付けられている．ICD は In-Circuit Debugger のことである．本稿では ICD に Pickit3 を使っている．ヘッダファイルにて

$$\#define\ ICS_PGD1 \quad 0xFFFF \tag{5.5}$$

と定義されている．下 2 桁は 11 であり，データシートの dsPIC33F CONFIGURATION BITS DESCRIPTION から
11 = Communicate on PGEC1 and PGED1
である．図 5.3 より，PGEC1 は dsPIC33FJ128MC802 の 5 番ピン，PEGD1 は 4 番ピンである．
_FICD(ICS_PGD1)
と記述することで，ICSP によるデバッグ用端子を 4, 5 番ピンに設定している．

5.3.2 オシレータシステム

図 5.18 はオシレータシステムのブロック図である．前項に述べたように，FNOSC ビットを FRCPLL=0b001 とすることで，システムクロック FOSC に FRC オシレータ + PLL

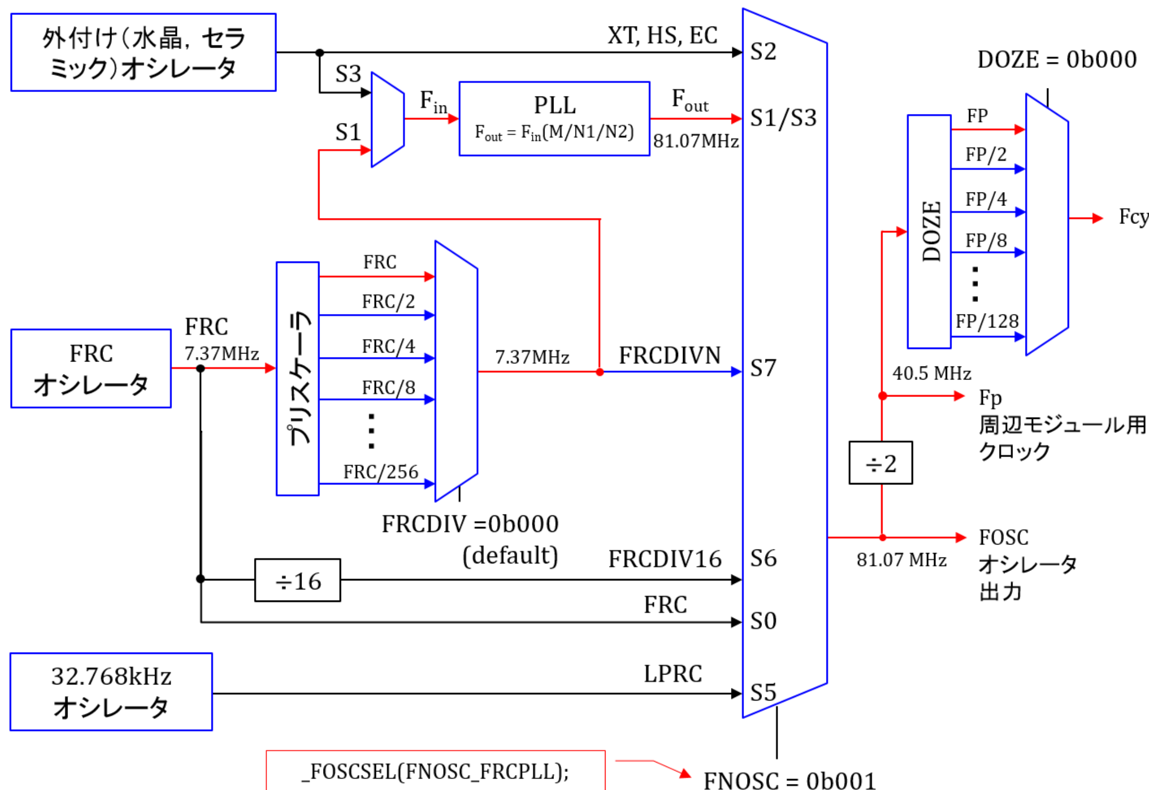


図 5.18: オシレータシステムブロック図

の出力を得る設定としている．FRCDIV ビット（図 5.20）はデータシートの CLOCK DIVISOR REGISTER によると，デフォルト値が 000 である．これは FRC divided by 1 を意味する．

図 5.19 は内蔵 PLL のブロック図である．PLL には内蔵 FRC オシレータの出力がそのまま入力され，その周波数 $F_{in} = 7.37$ [MHz] となる．PLLPRE, PLLDIV,PLLPOST を設定することで，PLL の出力周波数 F_{out} が決められる．図 5.20 は CLKDIV レジスタである．dsPIC33FJ128MC802 のデータシートによると

$$\begin{aligned} \text{PLLPRE} &= \text{0b11111 のとき N1} = 33 \\ &\vdots \\ \text{PLLPRE} &= \text{0b00001 のとき N1} = 3 \\ \text{PLLPRE} &= \text{0b00000 のとき N1} = 2 \end{aligned} \tag{5.6}$$

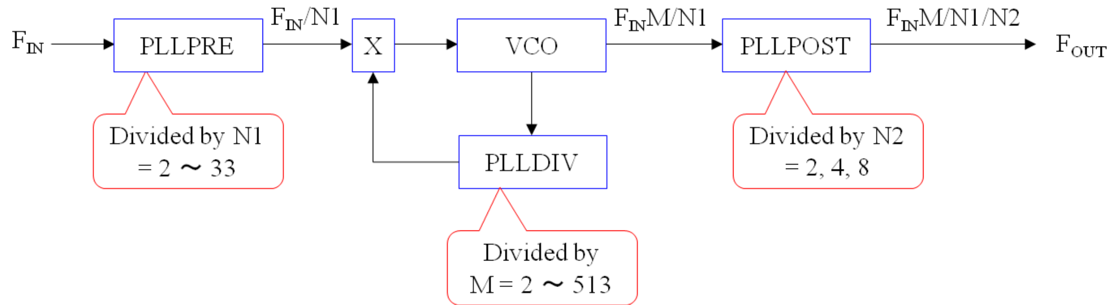


図 5.19: PLL ブロック図

CLKDIV

15	14	13	12	11	10	9	8
ROI	DOZE			DOZEN	FRCDIV		

7	6	5	4	3	2	1	0
PLLPOST		-	PLLPRE				

図 5.20: CLKDIV レジスタ

であり，

$$\begin{aligned} \mathrm{PLLPOST} &= \mathrm{0b11}\text{ のとき }\mathrm{N2} = 8 \\ \mathrm{PLLPOST} &= \mathrm{0b01}\text{ のとき }\mathrm{N2} = 4 \\ \mathrm{PLLPOST} &= \mathrm{0b00}\text{ のとき }\mathrm{N2} = 2 \end{aligned} \tag{5.7}$$

である．また，PLLFBD レジスタより

$$\begin{aligned} \mathrm{PLLDIV} &= \mathrm{0b111111111}\text{ のとき }\mathrm{M} = 513 \\ &\vdots \\ \mathrm{PLLDIV} &= \mathrm{0b000000001}\text{ のとき }\mathrm{M} = 3 \\ \mathrm{PLLDIV} &= \mathrm{0b000000000}\text{ のとき }\mathrm{M} = 2 \end{aligned} \tag{5.8}$$

である．図 5.21 にて，_PLLPRE = 0，_PLLDIV = 42，_PLLPOST = 0 と設定している．

```
//タイマ周期設定
unsigned int    tsamp1 =  40000;
                // Timer1 インタラプト周波数設定  40MHz/tasmp1 = 1kHz
                // init_timer1にて使用

//メイン関数
int main(void)
{ //PLL設定
   _PLLPRE   =  0;      // N1 =  2
   _PLLDIV   = 42;      //  M = 44
   _PLLPOST  =  0;      // N2 = 2
            // FOSC = FRC * M / (N1 * N2), FCY = Fosc / 2
            // FOSC = 7.37 MHz * 44 / (2 * 2) = 81MHz,  FCY = 40.5MHz

   _SWDTEN = 0;                       //ウォッチドッグタイマオフ
   while(OSCCONbits.LOCK !=1 );       // PLLロック待ち

   // ポート初期設定
   TRISA = 0b00000;                   // ポートRA4～RA0  出力ポート設定
   TRISB = 0b0000000000000000;        //        RB15～RB0 出力ポート設定

   // 周辺モジュール初期設定
   init_timer1(tsamp1);               // タイマ1初期設定

   while(1){}                         //メインループ
}
```

図 5.21: メイン関数

これにより，N1 = 2, M = 44, N2 = 2 となる．したがって，

$$
\begin{aligned}
F_{out} &= \frac{F_{in} \times M}{N1 \times N2} \\
&= \frac{7.37[\mathrm{MHz}] \times 44}{2 \times 2} \\
&\approx 81.07[\mathrm{MHz}] \qquad (5.9)
\end{aligned}
$$

と得られる．なお，PLLPRE等はデータシートで用いられている用語であるが，これら用語はヘッダファイルp33fj128mc802.h内にて，前にアンダーバーを付して

$$\#\mathrm{define\ _PLLPRE\ CLKDIVbits.PLLPRE} \qquad (5.10)$$

と定義されている．_PLLPREに値を代入することで，CLKDIVレジスタのPLLPREビットを書き換えることのできる設定が，ヘッダファイル内でなされている．

オシレータ出力FOSCは図5.18の設定により，PLLの出力周波数 F_{out} が取り出され，

FOSC = 81.07[MHz] となる．また，同図の周辺モジュール用クロック Fp は

$$\begin{aligned} \mathrm{Fp} &= \frac{\mathrm{FOSC}}{2} \\ &\approx 40.5[\mathrm{MHz}] \end{aligned} \tag{5.11}$$

となる．Fcy はデータシートにてデバイスインストラクションクロック，デバイスオペレーティングクロック，インストラクションサイクルクロック，システムクロックと呼ばれている．dsPIC33FJ128MC802 のデータシートの SYSTEM CONTROL REGISTER MAP によると，マップの最右列の ALL Resets に 0x3040 とある．すなわち，CLKDIV レジスタのデフォルト値は

CLKDIV = 0x3040 = 0b0011 0000 0100 0000

である．0xAAAA は 16 進数表記であり，0bBBBB BBBB ··· は 2 進数表記である．図 5.20 より，第 11 ビットは DOZEN と命名されたビットであり，デフォルト値は

DOZEN = 0;

である．これにより，ドーズモードはオフとされ，図 5.18 において，

$$\mathrm{Fcy} = \mathrm{Fp} \tag{5.12}$$

と設定される．DOZE（居眠り）モードは CPU の速度を遅くして，マイコンの消費電力を抑えるモードである．

以上により図 5.18 の赤い経路が選択される．

5.3.3 メイン関数

図 5.21 のメイン関数の後半は I/O ポートの設定，タイマ 1 の初期設定，そして，メインループ（無限ループ）である．

図 5.3 より，dsPIC33FJ128MC802 には RA0〜RA4, RB0〜RB15 の全部で 21 個のポートをディジタル信号の入力用もしくは出力用に利用できる．データシートの Parallel I/O Ports によると Data Direction Register (TRISx) の値が 1 のとき，当該ピンは入力用ポートとなり，0 のとき出力用ポートとなる．このプログラムでは RA0〜RA4, RB0〜RB15 の全てのピンを出力用ポートに設定している．

init_timer1() 関数にてタイマ 1 の初期設定を行っている．引数の tsamp1 はメイン関数の上の行にて

$$\text{unsigned int} \quad \text{tsamp1} = 40000; \tag{5.13}$$

により定義されている．この値により，タイマ 1 による割り込み周波数 $f_{interrupt}$ が

$$\begin{aligned} f_{interrupt} &= \frac{\text{FCY}}{\text{tsamp1}} \\ &= \frac{40.5[\text{MHz}]}{40000} \\ &\approx 1[\text{kHz}] \end{aligned} \tag{5.14}$$

と設定される．この仕組みは 5.3.4 項にて詳述する．

while(1){} は () の中が TRUE (=1) である限り {} の中の命令を実行する関数である．{} の中には何も記述されていないので，この while() 関数は，() の中の判定のみを無限に繰り返す．タイマ 1 は tsamp1 の定める周期でこの while() に割り込みをかける．

```
// タイマ1　初期設定
void init_timer1(unsigned int tsamp1)
{
   unsigned int TM1Config = T1_ON & T1_GATE_OFF & T1_PS_1_1&  T1_SOURCE_INT;
   unsigned int TM1IntConfig = T1_INT_PRIOR_6 & T1_INT_ON;

          OpenTimer1(TM1Config, tsamp1-1);
                                  //タイマ1設定　サンプリング周期　(1/40MHz) x  tsamp1  sec
          ConfigIntTimer1(TM1IntConfig);
                                  //タイマ1による割り込み設定
}
```

図 5.22: タイマ 1 初期設定関数 init_timer1()

5.3.4 タイマ初期設定関数

図 5.22 はタイマ 1 の初期設定関数 init_timer1() である．この関数は timer1.c ファイルの中にある．また，図 5.23 は，init_timer1() 関数が使用している OpenTimer1() 関数と ConfigIntTimer1() 関数である．init_timer1() 関数はメイン関数により起動され，引数として tsamp1 = 40000 を引き継ぐ．そして，TM1Config と TM1IntConfig に値を入力して，OpenTimer1() 関数を起動して，タイマ 1 を起動し，ConfigIntTimer1() 関数を起動してタイマ 1 による割り込みを設定する．

```
void OpenTimer1(unsigned int config,unsigned int period)
{
   TMR1  = 0;            /* Reset Timer1 to 0x0000 */
   PR1   = period;       /* assigning Period to Timer period register */
   T1CON = config;       /* Configure timer control reg */
}

void ConfigIntTimer1(unsigned int config)
{
   _T1IF = 0;                        /* clear IF bit */
   _T1IP = (config &0x0007);         /* assigning Interrupt Priority */
   _T1IE = (config &0x0008)>>3;      /* Interrupt Enable /Disable */
}
```

図　5.23: OpenTimer1() 関数と ConfigIntTimer1() 関数

TM1Config に格納する値と意味は以下の通りである．

T1_ON：タイマ1起動

T1_GATE_OFF：タイマ1の外部信号によるゲート操作オフ

T1_PS_1_1：プリスケーラの比率を1 ： 1

T1_SOURCE_INT：タイマのクロックソースを FCY とする．

これらの定数はヘッダファイル timer.h にて次のように定義されている．

```
#define T1_ON            0xffff
#define T1_GATE_OFF      0xffbf
#define T1_PS_1_1        0xffcf
#define T1_SOURCE_INT    0xfffd
```

したがって，

$$\text{T1_ON \& T1_GATE_OFF \& T1_PS_1_1 \& T1_SOURCE_INT} = \text{0b1111 1111 1000 1101} \tag{5.15}$$

である．この値が OpenTimer1() 関数の1番目の引数として，同関数に渡されている．OpenTimer1() 関数ではこの値を config の名前で受けて，T1CON に格納している．ヘッダファイル p33FJ128MC802.h にて，T1CON は TIMER1 CONTROL REGISTER であることが指定されている．データシートの TIMER1 CONTROL REGISTER によると，

T1CON に格納された値より

TON = 1 : Starts 16-bit Timer1

TGATE = 0 : When TCS = 0, gated time accumulation disabled

TCKPS = 00 : Presacle = 1:1

TCS = 0 : Internal clock (FCY)

と設定されたことが分かる．

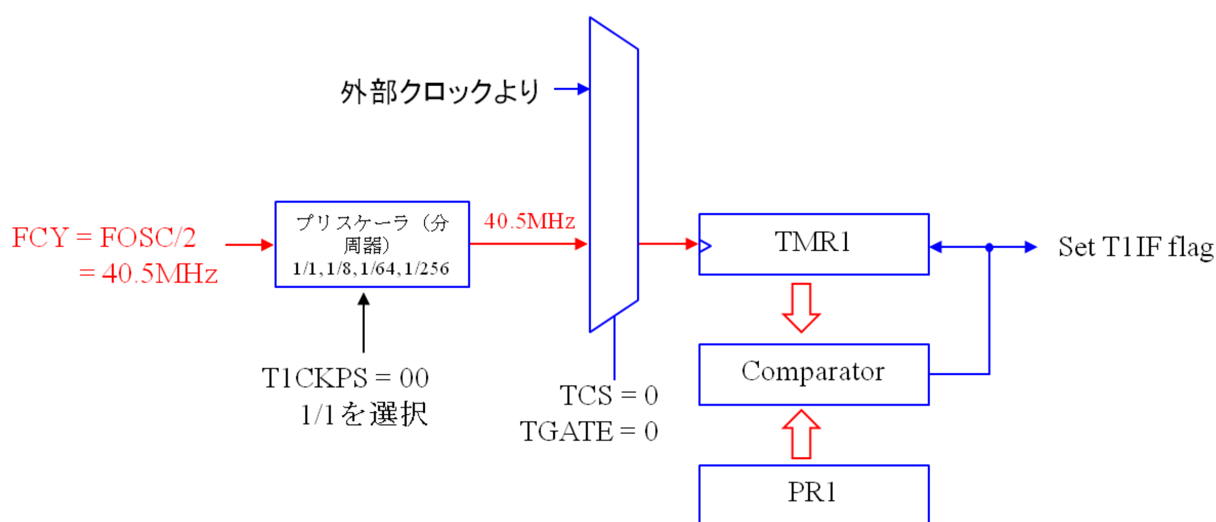


図 5.24: タイマ 1 ブロック図

図 5.24 はタイマ 1 のブロック図である．以上の設定に基づいて，データシートのブロック図から関連部分を抜粋してある．プリスケーラは 1:1 が選択されたので，16 ビットのカウンタ TMR1 には，赤い線の経路を通ってデバイスインストラクションクロック FCY(40.5MHz) が入力される．

OpneTimer1() 関数の 2 番目の引数は tsamp1 - 1 である．OpenTimer1() 関数ではこの値を period の名前で受けて，PR1 に格納している．図 5.24 の Comparator は PR1 の値と TMR1 の値を比較して，両者が一致したとき T1IF flag を 1 にセットし，TMR1 を 0 にリセットする．T1IF ビットはタイマ 1 により割り込がかけられているときに 1，そうでないときに 0 とするフラグである．そして，TMR1 は再びクロックのカウントアップを始める．TMR1 は (5.14) 式の周波数で 0 から tsamp1-1 までのカウントアップを繰り返す．

TM1IntConfig に格納する値と意味は以下の通りである．

T1_INT_PRIOR_6：タイマ1による割り込みの優先度を6とする．(優先度は1〜7)
T1_INT_ON：タイマ1による割り込みオン

これらの定数はヘッダファイル timer.h にて次のように定義されている．

```
#define T1_INT_PRIOR_6      0xfffe
#define T1_INT_ON           0xffff
```

したがって，

$$\text{T1_INT_PRIOR_6} \;\&\; \text{T1_INT_ON} = 0\text{b}1111\ 1111\ 1111\ 1110 \tag{5.16}$$

である．この値が ConfigIntTimer1() 関数の引数として，同関数に渡されている．ConfigIntTimer1() 関数ではこの値を config の名前で受けて，下3ビット (=0b110) を _T1IP に格納し，下から4ビット目 (=0b1) を _T1IE に格納している．ヘッダファイル p33FJ128MC802.h にて，

```
#define _T1IF IFS0bits.T1IF
#define _T1IP IPC0bits.T1IP
#define _T1IE IEC0bits.T1IE
```

と定義されている．IFS0 は INTERRUP FLAG STATUS REGISTER 0 であり，T1IF ビットはタイマ1により割り込がかけられているときに 1，かけられていないときに 0 とするフラグである．IPC0 は INTERRUP PRIORITY CONTROL REGISTER 0 であり，T1IP ビットは 0b111 のとき優先度最高の 7，0b001 のとき最低の 1 に設定される．IEC0 は INTERRUP ENABLE CONTROL REGISTER 0 であり，T1IE ビットは 1 のとき割り込み要求を受け付け，0 のとき受け付けない．したがって，

T1IF = 0 ; 割り込みフラグをクリア（0にする）
T1IP = 0b110 ; 割り込み優先度 6
T1IE = 1 : 割り込み要求受け付け

と設定されたことが分かる．

```
//タイマ1割り込み処理関数
void __attribute__((interrupt, no_auto_psv)) _T1Interrupt(void)
{
        _LATA4 = 1;                    // RA4 オン
        _LATA4 = 0;                    // RA4 オフ
        _T1IF = 0;                     // 割り込みフラグクリア
}
```

図 5.25: タイマ 1 割り込み処理関数

5.3.5 割り込み処理関数

図 5.25 はタイマ 1 割り込み処理関数 T1Interrupt() を示す．タイマ 1 により 1 秒間に 1000 回の頻度でメイン関数に割り込みがかけられ，この T1Interrupt() 関数が起動される．このプログラムは RA4 ポート（図 5.3 の 12 番ピン）を 1 にセットし，その直後に 0 にクリアしている．なお，ヘッダファイル p33fj128mc802.h にて

```
#define _LATA4 PORTAbits.LATA4
```

と定義されている．

T1IF ビットを 0 とすることで，タイマ 1 割り込みフラグを 0 にクリアしている．このフラグは図 5.24 において，タイマ 1 が割り込みを欠けるときに 1 にセットされている．割り込みフラグのクリアにより，再割り込みを受け付け可とする．

割り込み処理関数の名前 _T1Interrupt() は Interrupt vector name の一つである．この名前の一覧はデータシートの INTERRUPT VECTORS の Interrupt Source にある．割り込み処理関数はこのリストの記号（例えば T1, ADC1 など）の前に _（アンダーバー）をつけ，後ろに Interrupt() をつければよい．

__attribute__((interrupt,···)) はこの関数の属性が割り込みが要求が受け付けられたときに起動される関数であることを定義している．その後の no_auto_psv [1]は割り込み処理関数起動時に PSVPAG レジスタの内容の退避とデフォルト値のセットおよび割り込み終了時の回復を行わないことを指定している．この退避，セット，回復には 4 サイクル ($4/FCY \approx 100$ [ns]) を要するので，これらの処理を行わないことで割り込み処理に要する時間を 100[ns] 節約できる．auto_psv の機能は 32k バイトを超える大量の定数データを扱う場合に便利であるが，本稿のモータ制御では必要としないので，no_auto_psv としておく．

[1]psv は，MPLAB® XC16 C Compiler User's Guide によると，program space visibility の略である．const により定数を指定すると，この定数値はプログラム用のフラッシュメモリ内に格納される．データシートによれば dsPIC33FJ218MC802 のフラッシュメモリは 128k バイト，RAM メモリは 16k バイトである．大量の定数を扱う場合にはフラッシュメモリを活用したほうがよさそうである．定数を，例えば，unsigned int 等により変数として宣言すると，この変数は RAM メモリ内に確保される．大量の定数は const 宣言してフラッシュメモリ内に格納することで，少ない RAM メモリを消費することなく，また，外部メモリを使うよりも高速処理できる．ただし，32k バイトを超える定数を扱う場合には，事前に定数データを 32k バイト単位に分割してそれぞれ定数配列宣言をすると，フラッシュメモリ内に各配列が離れて格納される．32k バイト分のアドレスは 15 ビットで表現できる．PSVPAG レジスタの値は 16 ビット以上のアドレスを表現するのに使われる．PSVPAG レジスタのデフォルト値は 0 である．各定数配列のプログラムメモリ内へのアドレス割り当てはコンパイル時に自動的になされる．どう割り当てられたかは，コンパイル結果の output の program memory リストを見れば分かる．プログラムメモリ内に格納されたデータを読み出すには，16 ビット以上のアドレスは PSVPAG レジスタの値を書き換えて指定し，15 ビット以下については通常の配列値の読み出しと同じように指定する．例えば，m[ ] という名前の定数配列の先頭が 0x130ee(=0b0001 0011 0000 1110 1110) のアドレスに割り当てられた場合は，その第 16，15 ビット目（数字の右端が第 0 ビット目であることに注意）の値 0b10 を PSVPAG レジスタにセット (PSVPAG = 0b10 と) し，配列の先頭の値は m[0] とすることで取り出すことができる．割り込み処理関数起動時に auto_psv 指定をすれば，割り込み処理前の PSVPAG の値は退避され，代わりにデフォルト値がセットされ，割り込み終了時に回復される．no_auto_psv 指定では，PSVPAG に割り込み処理前の値がそのまま残り，また，割り込み処理関数内で書き換えられた場合は，割り込み終了後にはその書き換えられた値が引き継がれる．

5.3.6 タイマ割り込み処理プログラムのブロック図

図5.26はタイマ1割り込み処理プログラムのブロック図を示す．タイマ1により1[kHz]の繰り返し周波数でタイマ1割り込み処理関数を起動し，この関数は12番ピンに1/0の波形を出力する．図5.16は12番ピンの出力波形である．

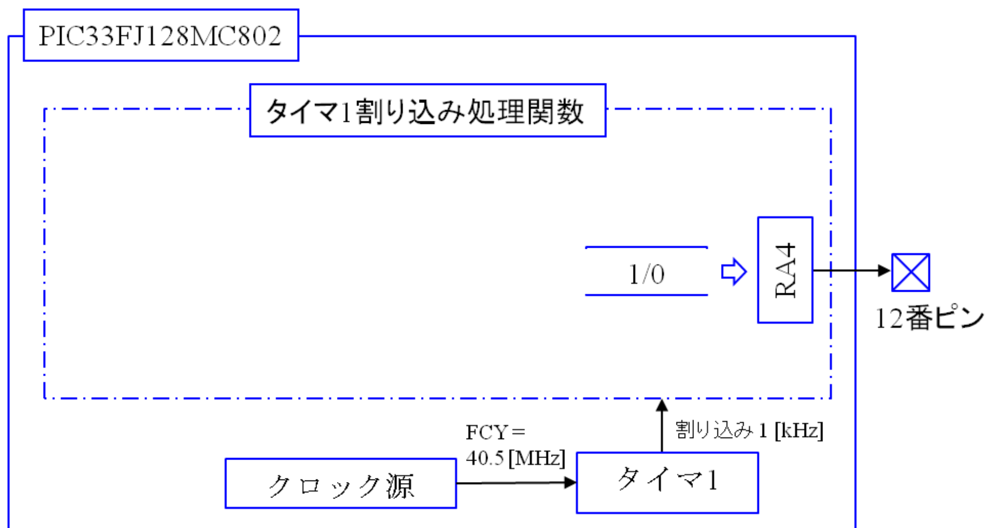


図 5.26: タイマ1割り込み処理プログラムのブロック図

5.4 I/O ポート

5.4.1 出力用ポート

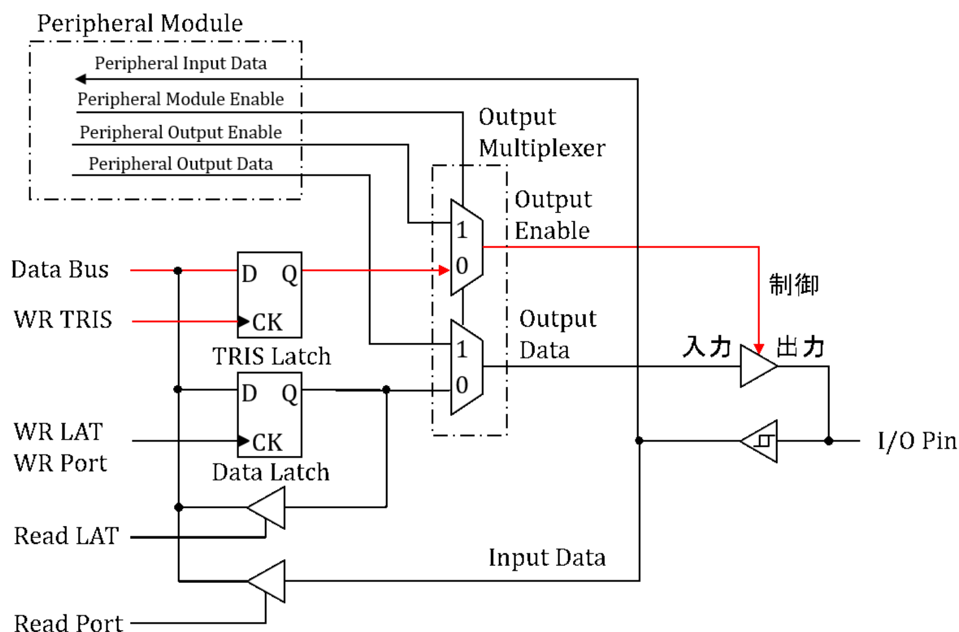


図 5.27: I/O ポートのブロック図（入出力設定）

本節では，I/O ポートについて解説する．図 5.27 は I/O ポートのブロック図である．図 5.3 において，I/O ポートは RA0〜RA4, RB0〜RB15 の 21 個ある．図は 1 個のポートに関するブロック図である．

I/O ピンを，周辺モジュール (Peripheral Module) が出力用として使用する場合は，I/O ポートは出力用としては使用不可となる．この切り替えは図中の一点鎖線で囲まれた Output Multiplexer によりなされる．周辺モジュールの Peripheral Module Enable 信号が “1” となると，Multiplexer は “1” の側に信号の経路を切替える．これにより，Peripheral Output Enable 信号，および，Peripheral Output Data 信号がそれぞれ 3 ステートバッファの制御信号，および，入力信号となる．3 ステートバッファは制御信号が “1” のとき，入力のディジタル信号をそのまま出力する．制御信号が “0” のときは，バッファ出力側を高インピーダンスとして，I/O ピンから切り離す．Peripheral Module Enable 信号が “0”（デフォルト値）の場合は，Output Multiplexer は “0” の側に経路をつなぐ．

図 5.21 において

TRISA = 0b00000;

としていた．0b00000 の最上位ビットが RA4 に，最下位ビットが RA0 に対応している．これにより RA4〜RA0 は出力ポートに設定される．図 5.27 は，Data Direction Register(TRISA) により 3 ステートバッファの制御を行っている経路を朱書きで示してある．TRISA 命令を実行すると，WR TRIS 信号が TRIS Latch 用の D フリップフロップの CK 入力に与えられ，この信号の立ち下がりエッジを捉えて，データバスの信号（0b00000 の中の対応ビットの値）が Q に保持される．そして，Q の値が 3 ステートバッファの制御入力となる．ただし，このままでは TRISA の当該ビットの “0” が制御入力として与えられるが，マイコン内のどこかで，もしくは，プログラムのコンパイル時に 0/1 を反転して，“1” を与える設定となっていると解釈する．データシートに説明はない．

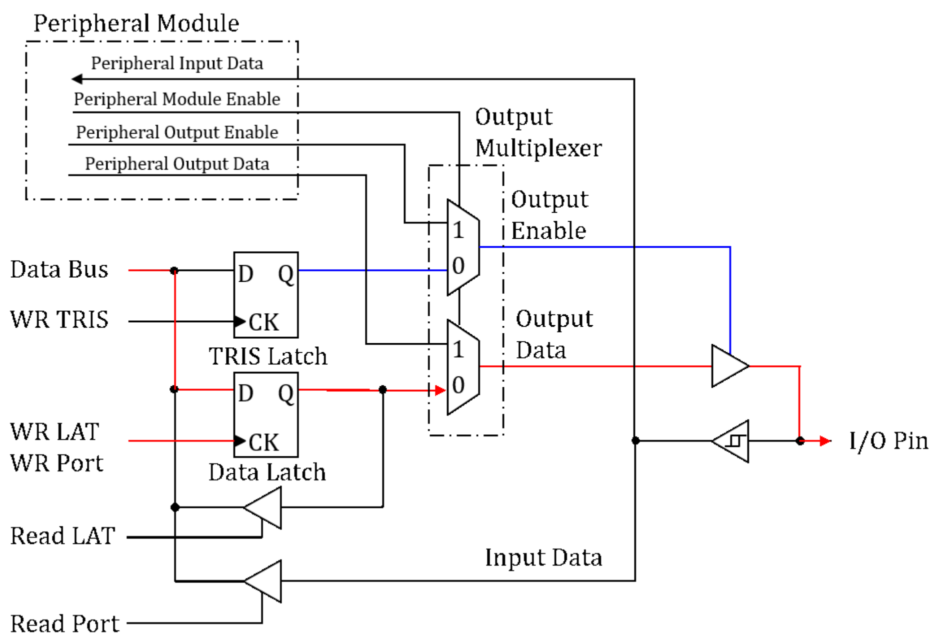


図 5.28: I/O ポートのブロック図（データ出力）

前節の割り込み処理プログラムは RA4（12 番ピン）に 1，0 のディジタル信号を出力した．図 5.28 は I/O ピンにデータを出力する経路を示す．このブロック図を RA4 用の I/O ポートと見なすと，

_LATA4 = 1;

により，WR LAT 信号が Data Latch 用の D フリップフロップの CK 入力に与えられ，こ

れによりデータ信号が Q に保持され，3 ステートバッファの入力が与えられる．この入力信号はバッファを通って，I/O ピンから出力される．この場合，データ信号は 1 のディジタル信号なので，I/O ピン（この場合は 12 番ピン）の電圧が約 3.3 [V] となる．

なお，データシートによると

_RA4 = 1;

としても RA4 ポートへの出力は変わらない．WR PORT 信号が Data Latch 用の D フリップフロップの CK 入力に与えられ，Q にデータ “1” を出力する．

5.4.2 入力用ポート

New Project として，I_O_Ports_Digital_Input を設定してください．そして，

モータドライブノート

に掲載の圧縮フォルダ内の I_O_Ports_Digital_Input フォルダから新たに作られた MPLABX-Projects¥I_O_Ports_Digital_Input.X フォルダ内に I_O_Ports_Digital_Input.c と timer1.c のファイルと include フォルダをコピーしてください．そして，I_O_Ports_Digital_Input.c と timer1.c のファイルを Source Files に付加し，include フォルダ内のヘッダファイルを Header Files に付加してください．

```
//タイマ1割り込み処理関数
void __attribute__((interrupt, no_auto_psv)) _T1Interrupt(void)
{
    _LATA4 = _RA0;          //2番ピンの入力を12番ピンに出
    _T1IF = 0;                  // 割り込みフラグクリア
}

//メイン関数
int main(void)
 {
    （省略）

    TRISA = 0b00001;                    // ポートRA4～RA0  出力ポート
    TRISB = 0b0000000000000000;     //      RB15～RB0 出力ポート
    AD1PCFGL = 0x0001;
                                    // AN0用のポートをディジタル用ポートに設定する.
                                    // デフォルトはアナログポート設定となっている.
    （省略）
}
```

図 5.29: I/O ポートをディジタル入力とするプログラム I_O_Ports_Digtal_Input.c

図 5.29 は I_O_Ports_Digtal_Input.c 内のプログラムの抜粋である．前節の Timer1_Interrupt.c のプログラムとの相違点を朱書きで示してある．

TRISA = 0b00001;

とすることで，RA0（2 番ピン) をディジタル入力用ポートに指定している．図 5.27 を RA0 のブロック図とすると，TRIS Latch 用の Q の値が “0” となり，3 ステートバッファの出力側は高インピーダンスとなって I/O ピンから切り離される．

ここで，データシートからは分り難いのであるが，図 5.3 にあるように，2 番ピンは AN0（AD 変換用のアナログ入力）用も兼ねている．データシートの I/O ポートのブロック図（図 5.27）には説明がないが，Configuring Analog Port Pins に説明があるように，AN5〜AN0 用ポートのデフォルトはアナログポートの設定である．そこで，

AD1PCFGL = 0x0001;

として，最下位ビットが AN0 用なので，これを “1” として，AN0 をディジタルポート設定にする必要がある．詳しくはデータシートの Configuring Analog Port Pins および AD1PCFGL レジスタの稿を参照されたい．

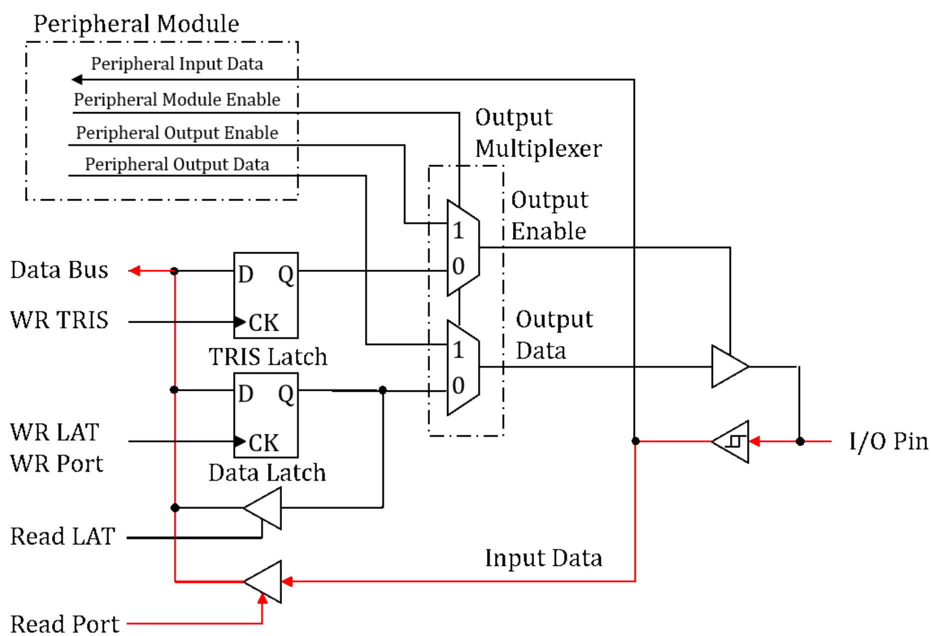


図 5.30: I/O ポートのブロック図（ディジタル入力）

図 5.30 を RA0 ポートのブロック図とすると，同図は

_LATA4 = _RA0;

の右辺の_RA0 命令による 2 番ピンからのディジタル信号読み込み時の，制御信号とデータの経路を示す．Read Port が “1” となることで，一番下の 3 ステートバッファの入力と出力がつながり，2 番ピンのアナログ信号がシュミットトリガを通して 0 or 1 のディジタル信号に変換されて，データバスへと伝えられる．この読み込まれた値は直ちに RA4 の出力用ポートに出力される．

なお，シュミットトリガは耐ノイズ性の高いトリガ回路である．図 5.31 はシュミットトリガの記号と入出力特性を示す．同図左が記号であり，右が入出力特性である．シュミットトリガの出力電圧 v_{out} が約 0 [V] から 約 3.3 [V] へと変化する入力電圧 v_{in1} の閾値と，v_{out} が約 3.3 [V] から 約 0 [V] へと変化する v_{in0} の閾値は図示のように異なっている．このように v_{in} 上昇時と下降時で経路の異なる特性はヒステリシス特性と呼ばれる．図のようなヒステリシス特性を持ったトリガ回路では，入力電圧にノイズが乗っていても，出力電圧は「ばたつかない」．図 5.32 はシュミットトリガの効果のイメージ図である．同図 (a) は $v_{in0} = v_{in1} = 2$ [V] として，ヒステリシス特性を持たないトリガの場合の入力電圧 v_{in} と出力電圧 v_{out} の波形例である．v_{in} にはノイズが載っていて，細かく変化している．入力電圧 v_{in} が閾値電圧 $v_{in0} = v_{in1} = 2$ [V] と何度も交叉するため，出力電圧 v_{out} はそのたびに反転し,「ばたついて」いる．

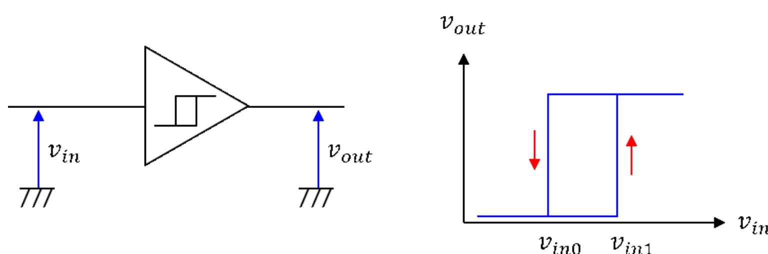


図 5.31: シュミットトリガと入出力特性

図 5.32(b) は，$v_{in0} = 1$ [V]，$v_{in1} = 2$ [V] として，ヒステリシス特性を持せたトリガの場合の入力電圧 v_{in} と出力電圧 v_{out} の波形例である．v_{in} が $v_{in1} = 2$ [V] を超えた時点で v_{out} が 3.3[V] となっている．v_{out} は，v_{in} が $v_{in0} = 1$ [V] を下回らない限り，反転しないため,「ばたつかない」．

図 5.33 は I_O_Ports_Digital_Inputs.c を実行したときの実験波形例である．入力電圧 v_{in}

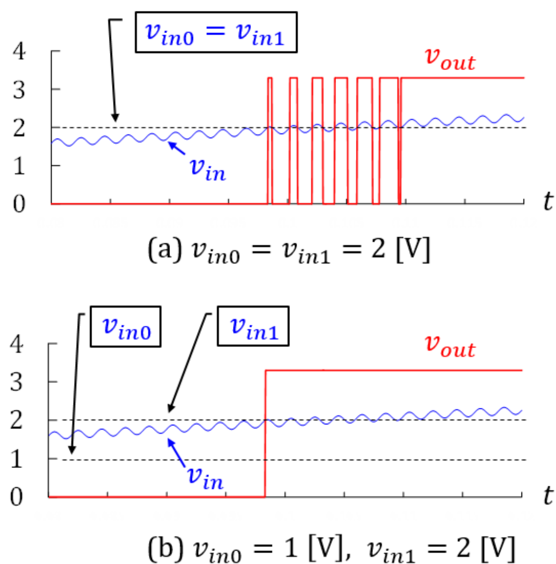


図　5.32: シュミットトリガの効果

を正弦波形としたときの出力電圧 v_{out} の波形である．v_{out} の立ち上がり時の v_{in} の値と，立ち下がり時の v_{in} の値に明確な違いがあることを見て取れる．

なお，以下の命令は，_LATA4 = _RA0; とは異なるので注意されたい．

_LATA4 = _LATA0;

図 5.34 は，_LATA0 を実行時の信号の経路を朱書きで示してある．Read LAT 信号により下から 2 番目の 3 ステートバッファの入出力がつながれ，Data Latch の Q の値がアドレスバスに読み出されている．

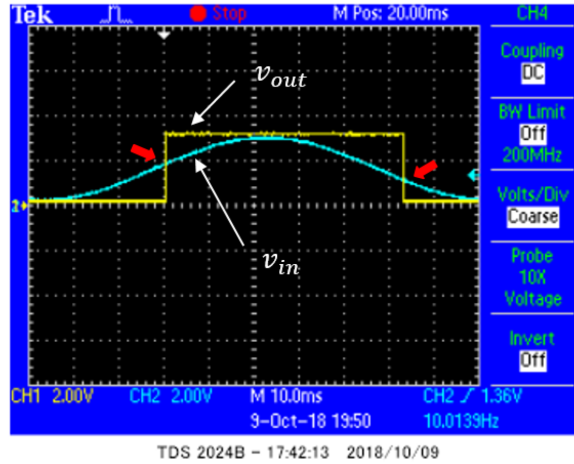


図 5.33: シュミットトリガの入出力波形

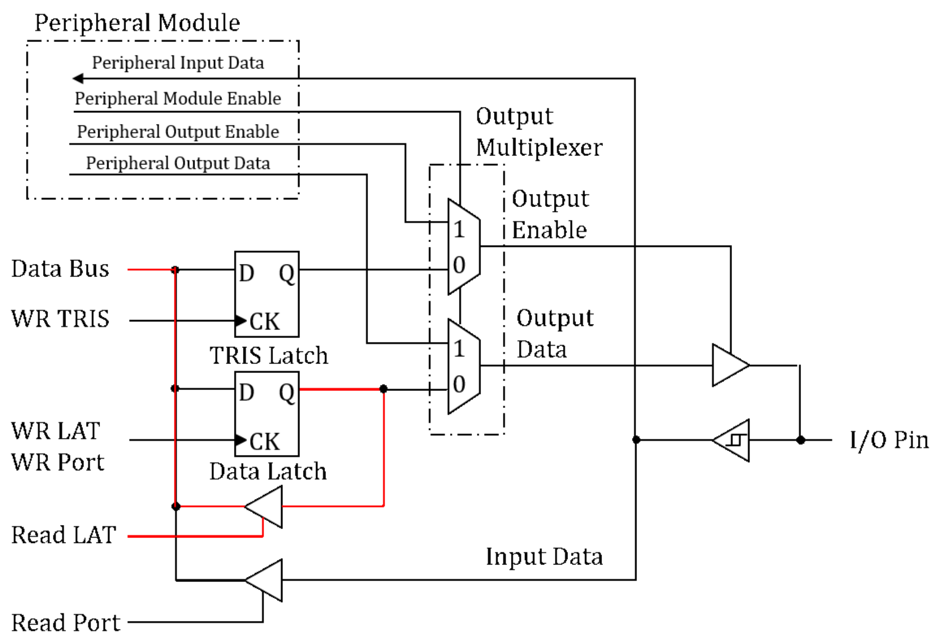


図 5.34: I/O ポートのブロック図（Latch 読み出し）

5.4.3 タイマ割り込み（ディジタル入力）処理プログラムのブロック図

図 5.35 は２番ピンからディジタル値を読み込んで，その結果を１２番ピンに出力するタイマ１割り込み処理プログラムのブロックずである．

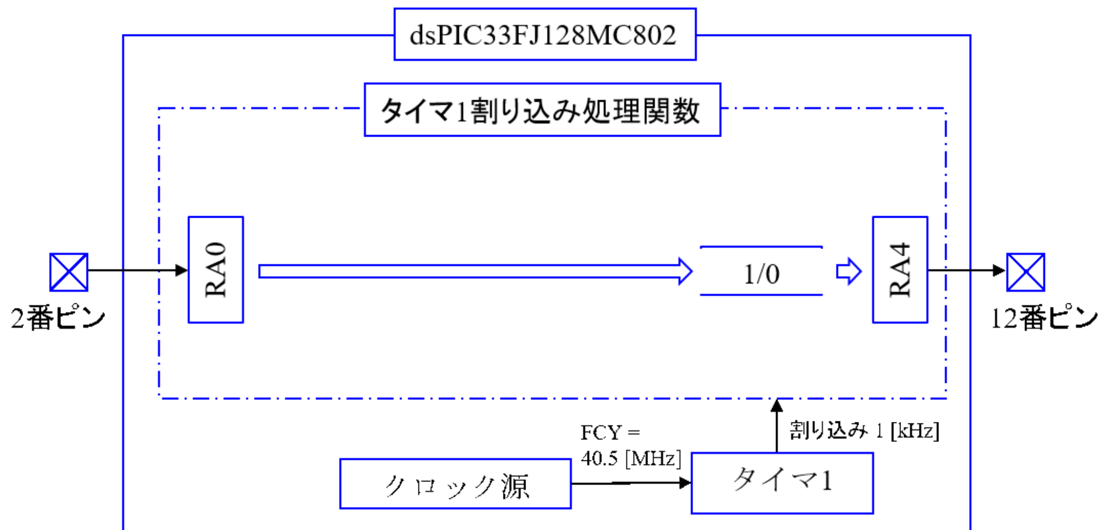


図 5.35: タイマ１割り込み（ディジタル入力）処理プログラムのブロック図

5.5 DA 変換器の使用方法

5.5.1 PIC マイコンとの接続

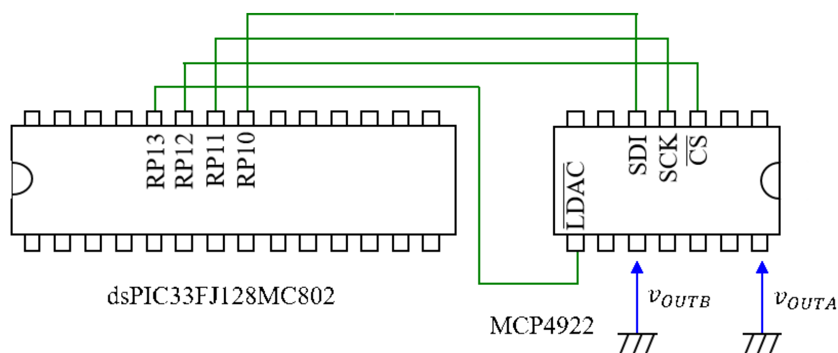


図 5.36: PIC マイコンと DA 変換器の接続

PIC マイコン内の数値のモニタ用として 12 ビット DA 変換器を用いる．PIC マイコンと DA 変換器は SPI(Serial Peripheral Interface) により通信を行う．図 5.36 は PIC マイコンと DA 変換器 (MCP4922) の接続の様子を示す．データシートによると，図 5.3 の RP0〜RP15 の Remappable Pins のいずれも SPI 用に使用できる．図 5.36 では

RP10(21 番ピン)：SDO（データ出力) → SDI（データ入力）へ
RP11(22 番ピン)：SCK (クロック出力) → SCK（クロック入力）へ
RP12(23 番ピン)：CS (チップ選択信号) → $\overline{\text{CS}}$（チップ選択入力）へ
RP13(23 番ピン)：LDAC (ラッチ信号) → $\overline{\text{LDAC}}$（ラッチ DAC 入力）へ

としている．

5.5.2 DAC.x の設定と実行

New Project として DAC.x を設定してください．そして，
モータドライブノート
に掲載の圧縮フォルダ内の DAC フォルダから，新たに作られた MPLABXProjects¥DAC.X フォルダ内に DAC.c, PI802.c, timer1.c のファイルと include フォルダをコピーしてください．そして，DAC.c, PI802.c, timer1.c のファイルを Source Files に付加し，include フォルダ内のヘッダファイル p33FJ128MC802.h, spi802.h, timer.h を Header Files に付加してください．

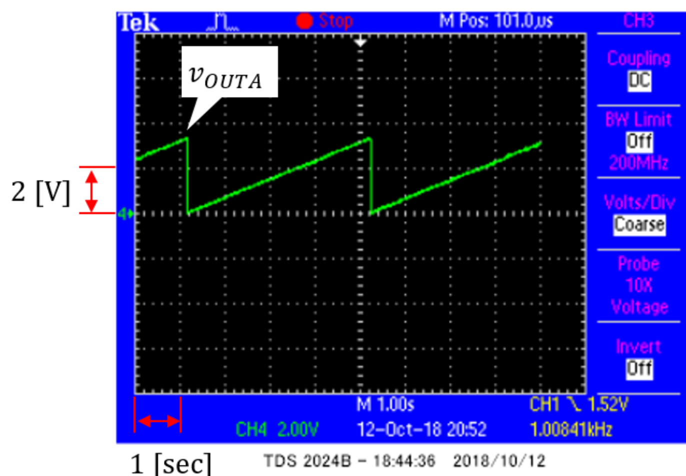


図　5.37: DAC.c 実行時の DA 変換器の出力電圧 v_{outA} の波形例

図 5.37 は DAC.c を実行したときの DA 変換器の 14 番ピンの出力電圧 v_{outA} の波形例である．横軸の 1 目盛りは 1 [sec] であり，ゆっくりと直線的に上昇して，約 3.3 [V] に達すると約 0 [V] にリセットされて再び直線的に上昇する波形が観測される．

5.5.3　DAC.c

図 5.38 は DAC.c 内のメイン関数とタイマ 1 割り込み処理関数のプログラムの抜粋である．図 5.21 の Timer1_Interrupt.c との違いを朱書きで示してある．大きな違いは SPI モジュールの設定（_RP10R = 7, _RP11R = 8, init_SPI() 関数）および，DA 変換器用の関数 (DAConv() 関数) である．

_RP10R = 7; (21 番ピンを SDO1 とする.)
_RP11R = 8;（22 番ピンを SCK1 とする.）

は Remappable ピンの RP10 と RP11 をそれぞれ SDO（データ出力）と SCK（クロック出力）に対応づけています．7 が SDO であること，8 が SCK であることは，dsPIC33FJ128MC802 のデータシートの OUTPUT SELECTION FOR REMAPPABLE PIN に記されている．この表によると，

SDO1：0b00111 = 0d7
SCK1：0b01000 = 0d8

```
unsigned int Vol_com = 0;          // ini_timer1にて使用

//タイマ1割り込み処理関数
void __attribute__((interrupt, no_auto_psv)) _T1Interrupt(void)
{
        Vol_com++;
        DAConv(Vol_com, 0x7000);        // チャネルA選択

        _T1IF = 0;                      // 割り込みフラグクリア
}

//メイン関数
int main(void)
 {
              （省略）

        //SPI用ポート設定
        _RP10R = 7;                     //RP10 → SDO (00111)
        _RP11R = 8;                     //RP11 → SCK (01000)

        //タイマ1モジュールとSPIモジュールの初期設定
        init_timer1(tsamp1);
        init_SPI();

        while(1){}                      //メインループ
}
```

図 5.38: DA 変換器を使うメイン関数とタイマ 1 割り込み処理関数のプログラム

である．なお，同表によると，SPI モジュールは 2 個備わっている．どちらでも使えるので，本稿では SDO1，SCK1 を使用する．SPI 用ポートを設定した後に

```
init_SPI();
```

関数により，SPI モジュールの初期設定を行っている．この関数の詳細は 5.5.5 項にて述べる．

タイマ 1 割り込み処理関数_T1interrupt() では，

```
Vol_com++;
```

により，Vol_com に 1 ずつ足して，

```
DAConv(Vol_com, 0x7000);
```

関数により，Vol_com の値を DA 変換器から出力する．0x7000 は DA 変換器へのコマンドである．5.5.6 項を参照されたい．

5.5.4 DAConv() 関数

```
// SPI ポート設定
#define  SPI_CS      _LATB12                         // D/Aコンバータ Select
#define  SPI_LDAC    _LATB13                         // D/Aコンバータ Load

// DAコンバータルーチン
void DAConv(unsigned int Data, unsigned int Command)
{
        int delay;

        SPI_CS = 0;                                  // CS Low
        SPI1BUF = (0x0FFF & Data) | Command;         // Command + Data送信
        delay = 880;                                 // 約 220 usecディレイ
        while(delay--);                              // 送信終了待ち
        SPI_CS = 1;                                  // CS High
        SPI_LDAC = 0;                                // Loadパルス出力
        SPI_LDAC = 1;
}
```

図 5.39: DAConv 関数のプログラム

DAConv() 関数を図 5.39 に示す．

```
#define SPI_CS      _LATB12
#define SPI_LDAC    _LATB13
```

により，RB12（23 番ピン）ポートに SPI_CS，RB13（24 番ピン）ポートに SPI_LDAC の名前をつけている．

DAConv() 関数を実行すると，DA 変換器 (MCP4922) の入出力に図 5.40 の様な波形が得られる．図は

Data = 0x0F57
Command = 0x7000

の場合の各電圧波形である．

SPI_CS = 0;

を実行すると，$\overline{\mathrm{CS}}$ の値が 1 から 0 となり，マイコンの 23 番ピン（RP12) の出力電圧が 0 [V] となる．このピンとつながる DA 変換器の 3 番ピン（$\overline{\mathrm{CS}}$) も 0 [V] となり，マイコンは DA 変換器 (MCP4922) を選んだ (Chip Select) ことになる．その後に

SPI1BUF = (0x0FFF & Data) | Command;

を実行すると，SPI1BUF レジスタには

$$
\begin{aligned}
(\mathrm{0x0FFF\&Data}) \mid \mathrm{Command} &= (\mathrm{0x0FFF\ \&\ 0x0F57}) \mid \mathrm{0x7000} \\
&= \mathrm{0x7F57} \\
&= \mathrm{0b0111\ 1111\ 0101\ 0111} \qquad (5.17)
\end{aligned}
$$

の値が格納され，SPI モジュールが起動される．図 5.40 の水色の波形のように，マイコンの 22 番ピン (RP11) から DA 変換器の 4 番ピン (SCK) に 16 個のクロック信号が送られる．

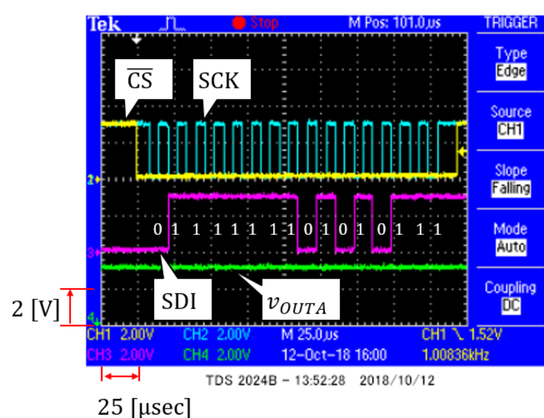


図 5.40: DA 変換器 (MCP4922) の入出力波形例 (V_{OUTA} が高いとき)

SCK の最初の立ち下がりのタイミングでマイコンは SDO(21 番ピン) に SPI1BUF の最上位ビットの値 “0” を出力する．マイコンの SDO は DA 変換器の SDI につながれている．DA 変換器は SCK の直後の立ち上がりのタイミングで SDI の値 “0” を内部レジスタに取り込む．SCK の次の立ち下がりでは，マイコンは SDO に SPI1BUF の 2 番目の上位ビットの値 “1” を出力し，DA 変換器は次の SCK の立ち上がりで SDI の値を取り込む．以降，

マイコンと DA 変換器は，全部で 16 個のデータを 16 個のクロックの立ち下がり／立ち上がりのタイミングを利用して送信／受信する．

DAConv() 関数（図 5.39）は，以上の 16 個のデータの送受信が完了するまで，

```
delay = 880;
while(delay--);
```

により待機する．なお，ここでの待機時間は約 220 [μsec] と大きな時間となっている．これは，波形を見やすくするためにわざと SPI モジュール内のクロックを遅くしてあるためである．この例では SPI モジュール内のクロックを最低速の設定にしてある．このクロックを最高速の設定にすると，待機時間は 700 [nsec] 程度で済む．詳しくは 5.5.7 項を参照されたい．

待機終了後に

```
SPI_CS = 1;
```

として，DA 変換器の選択を終了する．これだけでは，DA 変換器の出力端子 v_{OUTx} に変換結果は出力されない．最後に，

```
SPI_LDAC = 0;
```

とすることで，DA 変換された結果が v_{OUTx} に得られる．これにより，データ転送途中の値ではなく，データ転送完了後の結果を v_{OUTx} に出力させられる．また，2 チャネルのデータを転送後に，同時に出力させることもできる．詳細は 5.5.6 項を参照されたい．

図 5.39 および図 5.40 の例では，

Data = 0x0F57 = 0b0000 1111 0101 0111 = 0d3927
Command = 0x7000

である．Command は 5.5.6 項を参照されたい．

Data の範囲は 0x000〜0xFFF (= 0d0〜0 d 4095) である．DA 変換器の基準電圧 $V_{REF} = 3.3$[V] のとき，

Data = 0x0FFF = 0d4085 のとき $V_{OUTx} = 3.3$[V]

である．よって，Data = 0x0F57 = 0d3927 のとき

$$\begin{aligned} \mathrm{V_{OUTx}} &= \frac{3927}{4095} \times 3.3 \\ &= 3.16[V] \end{aligned} \tag{5.18}$$

である．

図 5.41 は

$$\mathrm{Data} = 0\mathrm{x}0388 = 0\mathrm{b}0000\ 0011\ 1000\ 1000 = 0\mathrm{d}904$$

のときの DA 変換器の各部の波形例である．このとき

$$\begin{aligned} \mathrm{V_{OUTx}} &= \frac{904}{4095} \times 3.3 \\ &= 0.72[V] \end{aligned} \tag{5.19}$$

である．

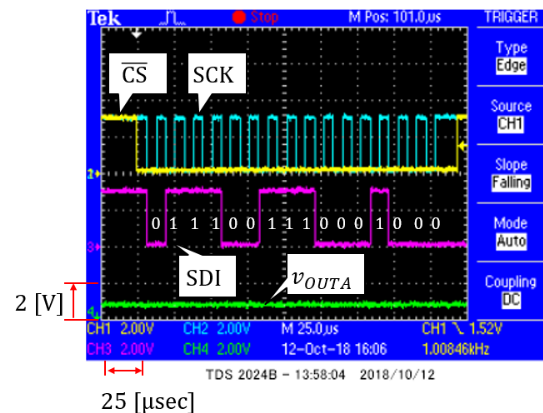


図 5.41: DA 変換器 (MCP4922) の入出力波形例（$\mathrm{V_{OUTA}}$ が低いとき）

5.5.5 SPI モジュールの初期設定

図 5.38 のメイン関数の init_SPI() により，SPI モジュールの初期設定関数を起動していた．図 5.42 は SPI モジュールの初期設定関数のプログラム，図 5.43 は SPI モジュールのブロック図である．

図中の用語は spi802.h に定義されている．このヘッダファイルは

```
void init_SPI(void)
{
  // SPIの初期設定パラメータ クロック20MHz
  unsigned int SPIConfig1= ENABLE_SCK_PIN & ENABLE_SDO_PIN & SPI_MODE16_ON
                 & SPI_SMP_ON & SPI_CKE_OFF & CLK_POL_ACTIVE_LOW
                 & MASTER_ENABLE_ON & SEC_PRESCAL_8_1
                 & PRI_PRESCAL_64_1;
  unsigned int SPIConfig2 = FRAME_ENABLE_OFF & FIFO_BUFFER_DISABLE;
  unsigned int SPIConfig3 = SPI_ENABLE & SPI_IDLE_STOP & SPI_RX_OVFLOW_CLR;

  // Open SPI
  SPI1CON1 = SPIConfig1;        /* Initalizes the spi module */
  SPI1CON2 = SPIConfig2;
  SPI1STAT  = SPIConfig3;       /* Enable/Disable the spi module */
}
```

図 5.42: SPI モジュールの初期設定関数のプログラム

モータドライブノート
に掲載の圧縮フォルダ内の DAC¥include フォルダの中にある．オリジナルは
C:¥Program Files (x86)¥Microchip¥xc16¥v1.35¥support¥peripheral_30F_24H_33F
内の spi.h ファイルである．このファイルからの抜粋を spi802.h とした．

dsPIC33FJ128MC802 のデータシートによると SPI モジュールの設定用レジスタは，SPIxCON1, SPIxCON2, SPIxSTAT の 3 種類がある．x には 1 or 2 が入る．いずれも 16 ビットのレジスタである．図 5.42 において，SPIConfig1 は SPI1CON1 レジスタ設定用の定数である．図 5.44 は SPI1CON1 レジスタを示す．ヘッダファイルとデータシートによると

ENABLE_SCK_PIN = 0xEFFF = 0b1110 1111 1111 1111

DISSCK = 1 : SCK ピンを SPI モジュールでは使用不可とし，
I/O 用ピンとする．
= 0 : SCK ピンを SPI モジュールが利用可とする．

ENABLE_SDO_PIN = 0xF7FF = 0b1111 0111 1111 1111

DISSDO = 1 : SDO ピンを SPI モジュールでは使用不可とし，
I/O 用ピンとする．
= 0 : SDO ピンを SPI モジュールが利用可とする．

SPI_MODE16_ON = 0xFFFF = 0b1111 1111 1111 1111

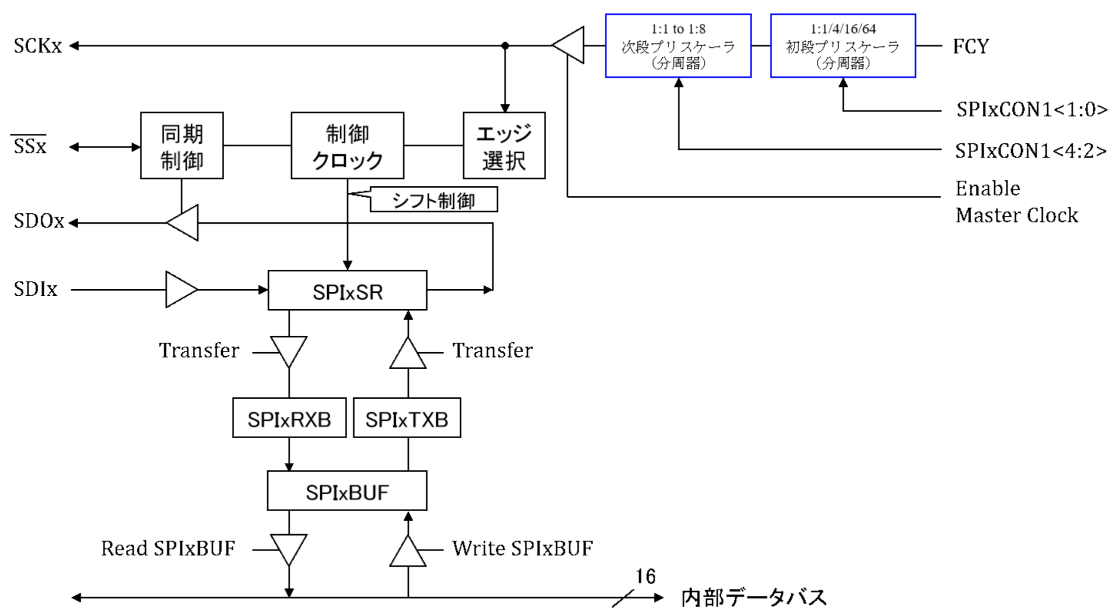


図　5.43: SPI モジュールのブロック図

SPI1CON1

15	14	13	12	11	10	9	8
-	-	-	DISSCK	DISSDO	MODE16	SMP	CKE

7	6	5	4	3	2	1	0
SSEN	CKP	MSTEN		SPRE		PPRE	

図　5.44: SPI1CON1 レジスタ

MODE16 ＝ 1：SPI モジュールが 16 ビットデータを取り扱う．

＝ 0：SPI モジュールが 8 ビットデータを取り扱う．

SPI_SMP_ON ＝ 0xFFFF ＝ 0b1111 1111 1111 1111

SMP ＝ 1：SDI からのデータ読み込みをデータ出力の最後の時点とする．

＝ 0：SDI からのデータ読み込みをデータ出力の中間の時点とする．

これは，SPI モジュールが他のデバイスの SPI モジュールからデータを読み込む場合の設定である．本項では使わない．

SPI_CKE_OFF ＝ 0xFEFF ＝ 0b1111 1110 1111 1111

CKE ＝ 1：クロックがアクティブレベルから Idle レベルへ遷移するとき SDO

にデータを出力する．

= 0：クロックが idle レベルからアクティブレベルへ遷移するときに SDO にデータを出力する．

CLK_POL_ACTIVE_LOW = 0xFFFF = 0b1111 1111 1111 1111

CKP = 1：クロックの 0 をアクティブレベル，1 を idle レベルとする．

= 0：クロックの 1 をアクティブレベル，0 を Idle レベルとする．

MASTER_ENABLE_ON = 0xFFFF = 0b1111 1111 1111 1111

MSTEN = 1：マイコンの SPI モジュールをマスターとする．

= 0：マイコンの SPI モジュールをスレーブとする．

SEC_PRESCAL_8_1 = 0xFFE3 = 0b1111 1111 1110 0011

SPRE = 111：2 段目プリスケーラを 1/1 とする．

= 110：2 段目プリスケーラを 1/2 とする．

= ···

= 000：2 段目プリスケーラを 1/8 とする．

PRI_PRESCAL_64_1 = 0xFFFC = 0b1111 1111 1111 1100

PPRE = 11：初段目プリスケーラを 1/1 とする．

= 10：初段目プリスケーラを 1/4 とする．

= 01：初段目プリスケーラを 1/16 とする．

= 00：初段目プリスケーラを 1/64 とする．

SPIConfig2 は SPI1CON2 レジスタ設定用の定数であるが，使用しない項目の設定なので説明を省略する．

図 5.45 は SPI1STAT レジスタである．SPIConfig3 は SPI1STAT レジスタ設定用の定数である．データシートによると

SPI_ENABLE = 0xFFFF = 0b1111 1111 1111 1111

SPIEN = 1：SPI モジュールを使用可とする．

= 0：SPI モジュールを使用不可とする．

SPI1STAT

15	14	13	12	11	10	9	8
SPIEN	-	SPISIDL	-	-	-	-	-

7	6	5	4	3	2	1	0
-	SPIROV	-	-	-	-	SPITBF	SPIRBF

図 5.45: SPI1STAT レジスタ

SPI_IDLE_STOP = 0xFFFF = 0b1111 1111 1111 1111

SPISIDL = 1 : マイコンが Idle モードのときは SPI モジュールを停止する．

= 0 : マイコンが Idle モードにあるときも SPI モジュールを作動させる．

SPI_RX_OVFLOW_CLR = 0xFFBF = 0b1111 1111 1011 1111

SPIROV = 0 : SPI1BUF のオーバーフローフラグを 0 としておく．

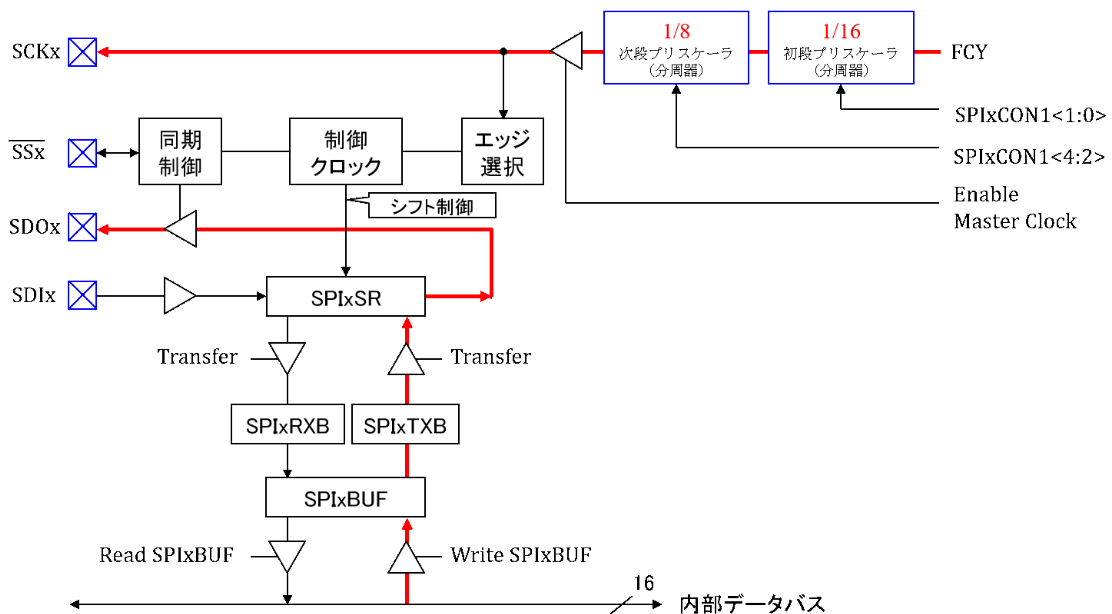


図 5.46: 初期設定後の SPI モジュールのブロック図

以上により，図 5.43 の SPI モジュールは，図 5.46 のように設定される．プリスケーラ

により，SPI モジュール内のクロック周波数を F_{SPI} とすると

$$F_{SPI} = \frac{40.5[MHz]}{8 \times 64} \approx 79[kHz] \tag{5.20}$$

となる．16 ビットのデータ転送には，最低でも

$$\frac{16}{79[kHz]} \approx 200[\mu sec] \tag{5.21}$$

を要する．内部クロックをこのように遅くした理由は，単に SCK などの波形を見やすくするためである．プリスケーラをいずれも 1/1 とすると，転送時間は

$$\frac{16}{40.5[MHz]} \approx 400[nsec] \tag{5.22}$$

程度に短縮できる（実際は 700[nsec] 程度である.）．ブレッドボード上の 40.5 [MHz] の信号をオシロスコープで観測すると，波形が大きく乱れる．DA 変換器の動作に支障は見られないが，説明の簡明さは得られない．

SPI モジュールは，図 5.39 の

```
SPI1BUF = (0x0FFF & Data) | Command;
```

命令の実行により，図 5.46 において，SPIxBUF に Command + Data が書き込まれることで起動される．SPIxBUF の内容は SPIxSR レジスタに転送され，図 5.40 の手順で，1 ビットずつ DA 変換器に転送される．

5.5.6　DA 変換器

図 5.47 は DA 変換器 (MCP4922) のブロック図である．データの経路は A, B の 2 チャネルがある．マイコンと図 5.36 のように接続されている．$\overline{\text{CS}}$ により，Interface Logic が起動され，SCK, SDI を通して，データを受け取る．Input Register x にデータ転送が終了した段階で，$\overline{\text{LDAC}}$ に 0 が入ることで，DACx Register に受信データが一括転送される．その後，string DAC（抵抗分圧方式 DA 変換器）により DA 変換がなされて，オペアンプを通して v_{OUTx} に出力される．マイコンから送られてくるデータは，図 5.39 の DAConv 関数のプログラムより

```
0bcccc dddd dddd dddd
```

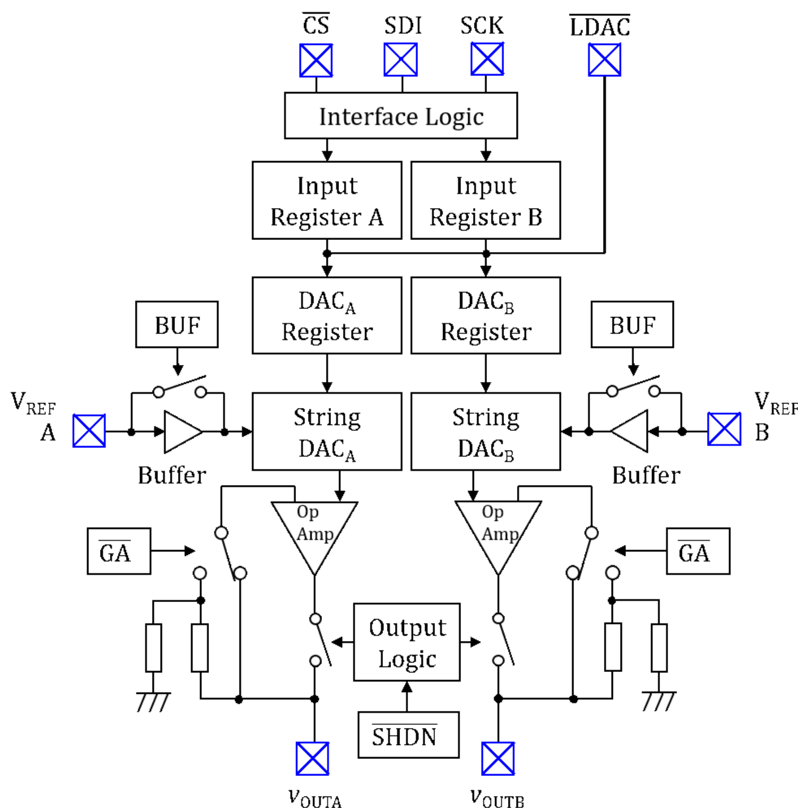


図 5.47: DA 変換器 (MCP4922) のブロック図

という構成である．上位 4 ビットの cccc は Command であり，下位 12 ビットが Data である．図 5.38 のタイマ 1 割り込み処理プログラムによると，Command は 0x7000 であった．したがって，この場合，コマンドは 0b0111 である．

DA 変換器 MCP4922 のデータシートはインターネットより無料でダウンロードできる．このデータシートの WRITE COMMAD REGISTER によると，上位 4 ビットのコマンドは上から

$\overline{\mathrm{A}}$/B : 1 のときチャネル B，0 のときチャネル A 選択
BUF： 1 のとき基準電圧用バッファアンプ使用，0 のとき不使用
$\overline{\mathrm{GA}}$: 1 のときアナログ増幅器のゲイン 1，0 のときゲイン 2
$\overline{\mathrm{SHDN}}$: 1 のとき出力オン，0 のとき出力はハイインピーダンス

である．したがって，コマンドが 0b0111 のとき，

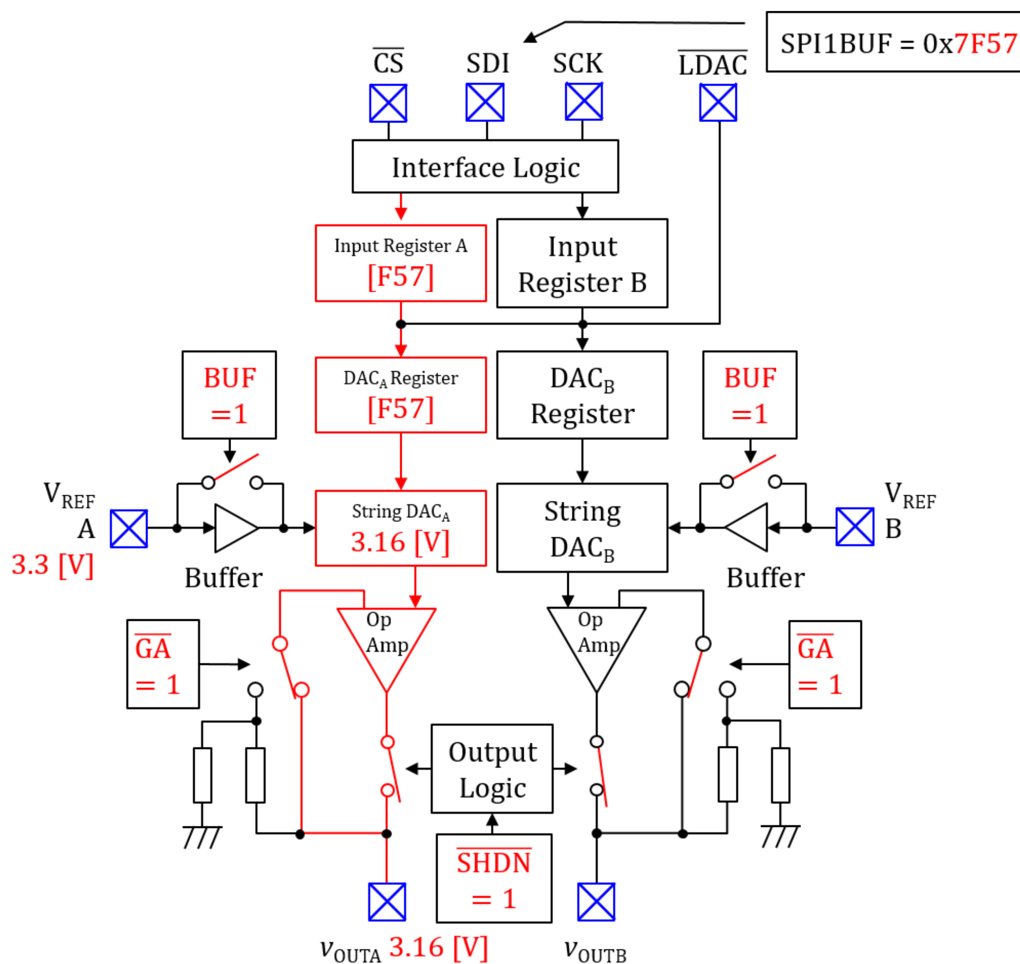


図 5.48: DA 変換器 (MCP4922) のブロック図におけるデータの経路

$\overline{\mathrm{A}}/\mathrm{B} = 0$ チャネル A 選択

$\mathrm{BUF} = 1$ 基準電圧用バッファアンプ使用

$\overline{\mathrm{GA}} = 1$ アナログ増幅器のゲイン 1

$\overline{\mathrm{SHDN}} = 1$ 出力オン

である．図 5.48 は，このときの DA 変換器内のデータの経路を朱書きで示す． BUF, $\overline{\mathrm{GA}}$，$\overline{\mathrm{SHDN}}$ の値に応じて図示のようにスイッチングがなされている．基準電圧 $\mathrm{V_{REF}}$ = 3.3 [V] であり，マイコンから 0x7F57 の Commnad + Data が送られてきたとき，v_{OUTA} から 3.16 [V] のアナログ電圧が出力される．

なお，チャネル B を選択するには，Command を 0b1111 とすればよい．

5.5.7 SPI の内部クロックを最大に

参考に SPI モジュールの内部クロックを最大とした場合の DA 変換器入力の各部波形を図 5.49 に示す．これは，図 5.42 にて，

SEC_PRESCAL_8_1 → SEC_PRESCAL_1_1
PRI_PRESCAL_64_1 → PRI_PRESCAL_1_1

として，図 5.39 にて

delay = 880; → delay = 1;

とすればよい．

緑色の波形は $\overline{\mathrm{LDAC}}$ の電圧 V_{LDAC} である．$\overline{\mathrm{CS}}$ を “1” にリセットした直後に $\overline{\mathrm{LDAC}} = 0$ としている．

SCK は 40.5 [MHz] のクロック信号であり，ブレッドボード上では波形が大きく崩れてしまっていることが分かる．ただし，DA 変換器の動作に支障は見られない．

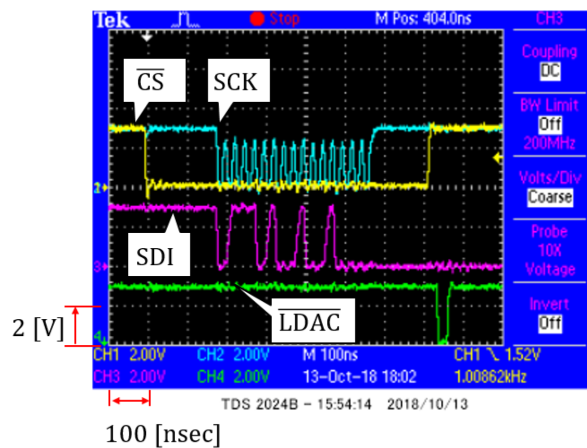


図 5.49: DA 変換器の入力波形（SPI モジュールの内部クロック最大（40.5[MHz]））

5.5.8 DA 変換プログラムのブロック図

図 5.50 は DA 変換プログラム DAC.c のブロック図である．SPI モジュールの内部クロック最大の場合である．

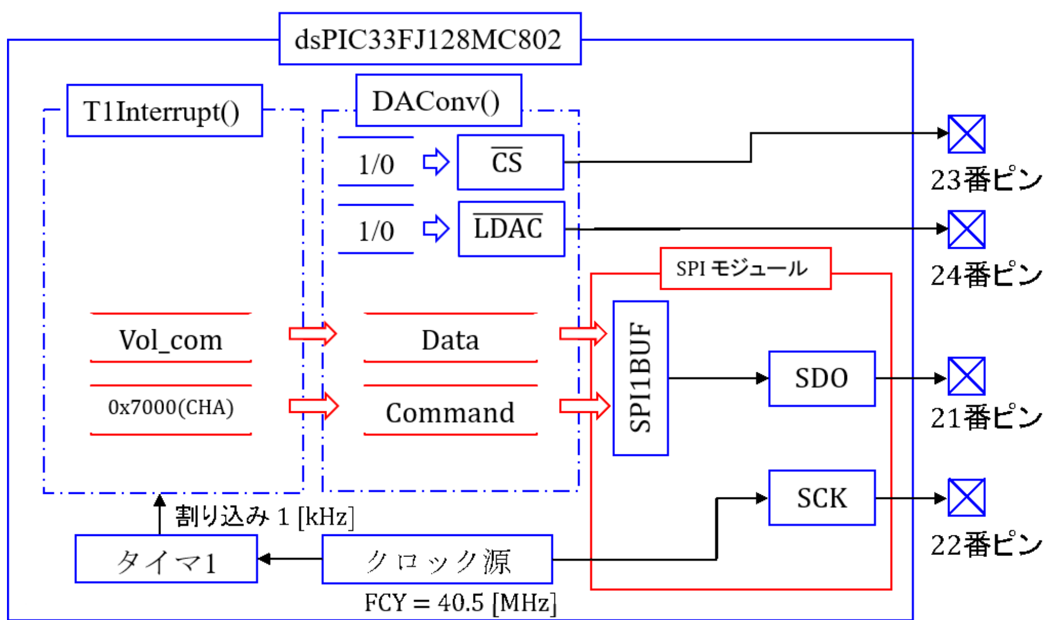
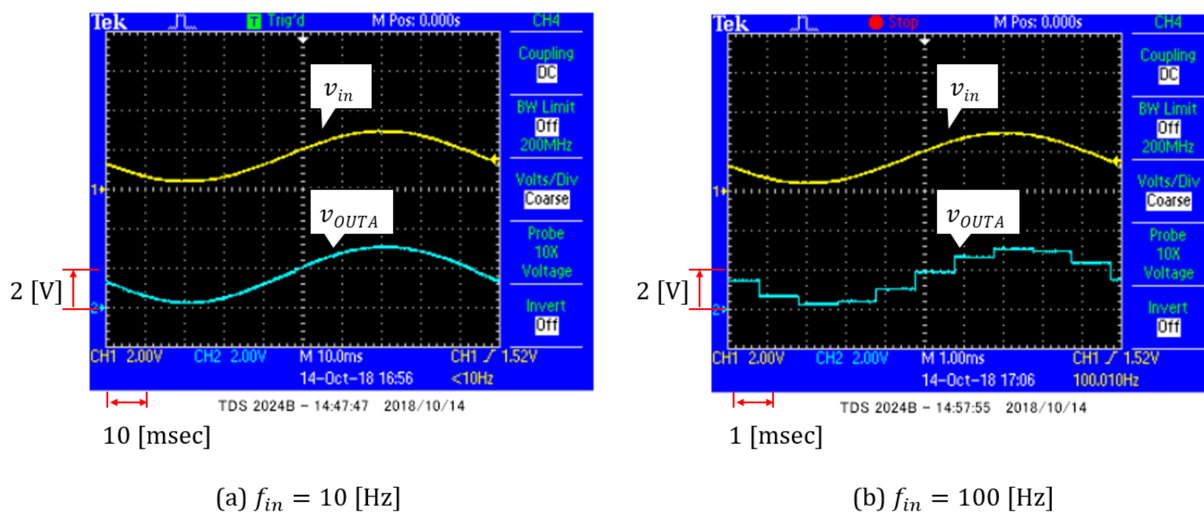


図　5.50: DA 変換プログラム DAC.c のブロック図

5.6 AD 変換モジュール

5.6.1 動作例



(a) f_{in} = 10 [Hz]

(b) f_{in} = 100 [Hz]

図 5.51: AD 変換の入出力波形例

本節から，いよいよ，図 5.2 の全回路を用いて実験を行います．New Project として，ADC を設定してください．そして，

モータドライブノート

に掲載の圧縮フォルダ内の ADC フォルダから新たに作られた MPLABXProjects¥ADC.X フォルダ内に ADConv.c,ADC802.c, SPI802.c, timer1.c のファイルと include フォルダをコピーしてください．そして，ADConv.c,ADC802.c, SPI802.c, timer1.c のファイルを Source Files に付加し，include フォルダ内のヘッダファイルを Header Files に付加してください．

図 5.51 は，図 5.2 において，dsPIC の 2 番ピンと GND 間の電圧 v_{in} に正弦波電圧を印加したときの DA 変換器の出力電圧 v_{OUTA} の波形例を示す．同図 (a) は v_{in} の周波数 f_{in} = 10[Hz] の場合であり，(b) は f_{in} = 100[Hz] の場合である．AD 変換のサンプリング周波数 f_{SAMP} = 1[kHz] であるため，f_{in} = 100[Hz] の場合には，AD 変換後の波形 v_{OUTA} は階段状になっている．

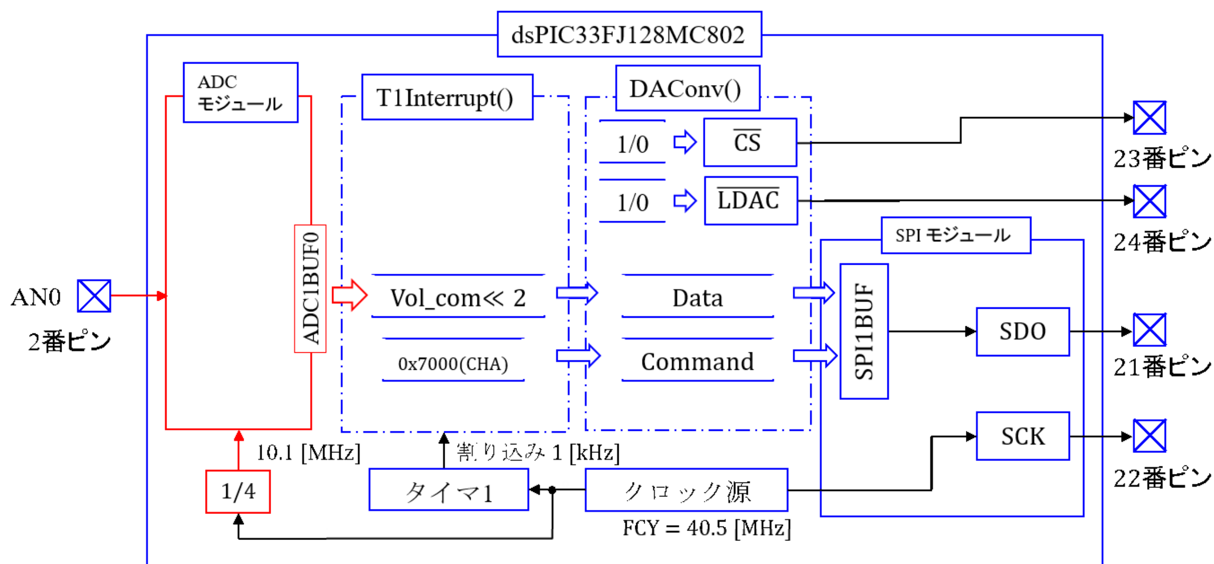


図 5.52: AD 変換プログラム ADConv.c のブロック図

5.6.2 タイマ 1 割り込みによる AD 変換のサンプリング周期設定

図 5.52 は，AD 変換プログラム ADConv.c のブロック図である．図 5.50 の DA 変換プログラムに ADC モジュールを追加している．図 5.53 は AD 変換プログラム ADConv.c のメイン関数とタイマ 1 割り込み処理関数である．メイン関数では

```
init_ADC();
```

により，AD 変換モジュールの初期設定を行っている．後述するように，2 番ピンのアナログ入力（AN0) に対して，10 ビットの AD 変換を自動的に繰り返す設定である．タイマ 1 による割り込み処理の中で

```
Vol_com = ADC1BUF0;
```

により，AD 変換結果が格納されている ADC1BUF0 レジスタから 10 ビットの AD 変換結果を読み出して，Vol_com に格納している．

```
Vol_com = Vol_com<<2;
```

```
//タイマ1割り込み処理関数
void __attribute__((interrupt, no_auto_psv)) _T1Interrupt(void)
{
    Vol_com = ADC1BUF0;
    Vol_com = Vol_com<<2;
    DAConv(Vol_com, 0x7000);    // チャネルA選択

    _T1IF = 0;                  // 割り込みフラグクリア
}

//メイン関数
int main(void)
 {
            （省略）

   init_timer1(tsamp1);    // タイマ1モジュール初期設定
   init_SPI();             // SPIモジュール初期設定
   init_ADC();             // ADCモジュール初期設定

   while(1){}              //メインループ
}
```

図 5.53: AD 変換プログラムのメイン関数，タイマ 1 割り込み処理関数

は，Vol_com を 2 ビット桁上げして，10 ビットデータを 12 ビットデータへと変換している．そして，

DAConv(Vol_com, 0x7000);

により，DA 変換器 (12 ビット) のチャネル A にデータを転送している．

以上のように ADC1BUF0 レジスタから値を読み出すだけで AD 変換結果が得られる設定は，図 5.54 の init_ADC() 関数によりなされている．AD 変換モジュール設定用レジスタは AD1CON1，AD1CON2，AD1CON3，AD1CON4，AD1CHS123，AD1CHS0，AD1PCFGL，AD1CSSL である．AD 変換モジュールは様々な使い方に対応できるように作られているために，初めて使う人には煩雑であるが，慣れれば実に重宝である．

AD1CON1 は AD1CON1 レジスタ設定用の変数である．このレジスタは 16 ビットからなる．図 5.55 は AD1CON1 レジスタ内の各ビットに付けられた名前を示す．図 5.54 の AD1CON1_set の具体的な定数名と値はヘッダファイル ADC.h にて定義されている．これとデータシートを見較べながら，AD1CON1_set による AD 変換モジュールの設定内容を以下にまとめる．

```
void init_ADC(void)
{
   unsigned int AD1CON1_set = ADC_MODULE_ON & ADC_IDLE_CONTINUE
                   & ADC_ADDMABM_SCATTR  & ADC_AD12B_10BIT
                   & ADC_FORMAT_INTG & ADC_CLK_AUTO & ADC_MULTIPLE
                   & ADC_AUTO_SAMPLING_ON & ADC_SAMP_ON;
   unsigned int AD1CON2_set = ADC_VREF_AVDD_AVSS & ADC_SCAN_OFF
                   & ADC_SELECT_CHAN_0  &  ADC_DMA_ADD_INC_1
                   & ADC_ALT_BUF_OFF & ADC_ALT_INPUT_OFF;
   unsigned int AD1CON3_set = ADC_CONV_CLK_SYSTEM
                   & ADC_SAMPLE_TIME_2  & ADC_CONV_CLK_4Tcy;
   unsigned int AD1CON4_set = 0x0;      // 未使用
   unsigned int AD1PCFGL_set = ENABLE_AN0_ANA;
   unsigned int AD1CSSL_set = 0x0;  // 未使用
   unsigned int AD1CHS0_set = ADC_CH0_POS_SAMPLEA_AN0
                                  & ADC_CH0_NEG_SAMPLEA_VREFN;
   unsigned int AD1CHS123_set =  0x0;  // 未使用

   OpenADC802(AD1CON1_set, AD1CON2_set, AD1CON3_set, AD1CON4_set,
                      AD1PCFGL_set, AD1CSSL_set);
   SetChanADC802_10BIT(AD1CHS0_set, AD1CHS123_set);
}
```

図 5.54: AD 変換モジュールの初期設定プログラム

ADC_MODULE_ON = 0xFFFF = 0b1111 1111 1111 1111

ADON = 1 : ADC モジュールをオンにする.

= 0 : ADC モジュールをオフにする.

ADC_IDLE_CONTINUE = 0xDFFF = 0b1101 1111 1111 1111

ADSIDL = 1 : idle モードにおいて ADC モジュールをオフにする.

= 0 : idle モードにおいても ADC モジュールをオンにする.

ADC_ADDMABM_SCATTR = 0xEFFF = 0b1110 1111 1111 1111

ADDMABM = 1 : DMA RAM アドレスを AD 変換順とする.

= 0 : DMA RAM アドレスを CH 毎に別々に設定する.

このプログラムでは DMA は使わないので無関係な設定.

AD1CON1

15	14	13	12	11	10	9	8
ADON	-	ADSIDL	ADDMABM	-	AD12B	FORM	

7	6	5	4	3	2	1	0
SSRC			-	SIMSAM	ASAM	SAMP	DONE

図 5.55: AD1CON1 レジスタ

ADC_AD12B_10BIT = 0xFBFF = 0b1111 1011 1111 1111

AD12B = 1 : 12 ビット AD 変換を選択する．

= 0 : 10 ビット AD 変換を選択する．

ADC_FORMAT_INTG = 0xFCFF = 0b1111 1100 1111 1111

FORM = 0b11 : 符号付き固定小数点.

= 0b10 : 符号なし固定小数点

= 0b01 : 符号付き整数

= 0b00 : 符号なし整数

ADC_CLK_AUTO = 0xFFFF = 0b1111 1111 1111 1111

SSRC = 0b111 : 自動繰り返しサンプリング

= 0b101 : PWM2 同期サンプリング

= 0b100 : タイマ 5 同期サンプリング

= 0b011 : PWM1 同期サンプリング

= 0b010 : タイマ 3 同期サンプリング

= 0b001 : INT0(16 番ピン) 同期サンプリング

= 0b000 : SAMP = 0 によりサンプル & ホールド → AD 変換開始

ADC_MULTIPLE = 0xFFF7 = 0b1111 1111 1111 0111

SIMSAM = 1 : CH0〜CH3 を同時にサンプル & ホールド

= 0 : CH0〜4 を順にサンプル & ホールド ただし，このプログラムでは CH0 しか使わないのでこの設定は無関係．使用チャネルは AD1CON2 の CHPS bits で指定する．

ADC_AUTO_SAMPLING_ON = 0xFFFF = 0b1111 1111 1111 1111

ASAM = 1：AD 変換が終了すると自動的に次のサンプルを開始

= 0：SAMP = 1 によりサンプルを開始

ADC_SAMP_ON = 0xFFFF = 0b1111 1111 1111 1111

SAMP = 1：サンプリング開始

= 0：サンプル & ホールド → AD 変換開始

ASAM = 0 のとき，SAMP = 1 によりサンプリング開始

ASAM = 1 であればハードウェアにより AD 変換が終了すると自動的に次のサンプルを開始

SSRC = 0b000 のとき，SAMP = 0 によりサンプル & ホールド → AD 変換開始

SSRC ≠ 0b000 のとき，ハードウェアによりサンプル & ホールド → AD 変換開始

AD1CON2

15	14	13	12	11	10	9	8
VCFG			-	-	CSCNA	CHPS	

7	6	5	4	3	2	1	0
BUFS	-	SMPI				BUFM	ALTS

図 5.56: AD1CON2 レジスタ

図 5.56 は，AD1CON2 レジスタの各ビットに付けられた名前を示す．図 5.57 の AD 変換モジュールの設定結果を参照しながら，図 5.54 の AD1CON2_set による設定内容を以下にまとめる．

ADC_VREF_AVDD_AVSS = 0x0FFF = 0b0000 1111 1111 1111

VCFG = 0b000：ADREF+ = AVDD, ADREF- = AVSS

= 0b001：ADREF+ = External VREF+, ADREF- = AVSS

= 0b010：ADREF+ = AVDD, REF- = External VREF-

= 0b011：ADREF+ = External VREF+, ADREF- = External VREF-

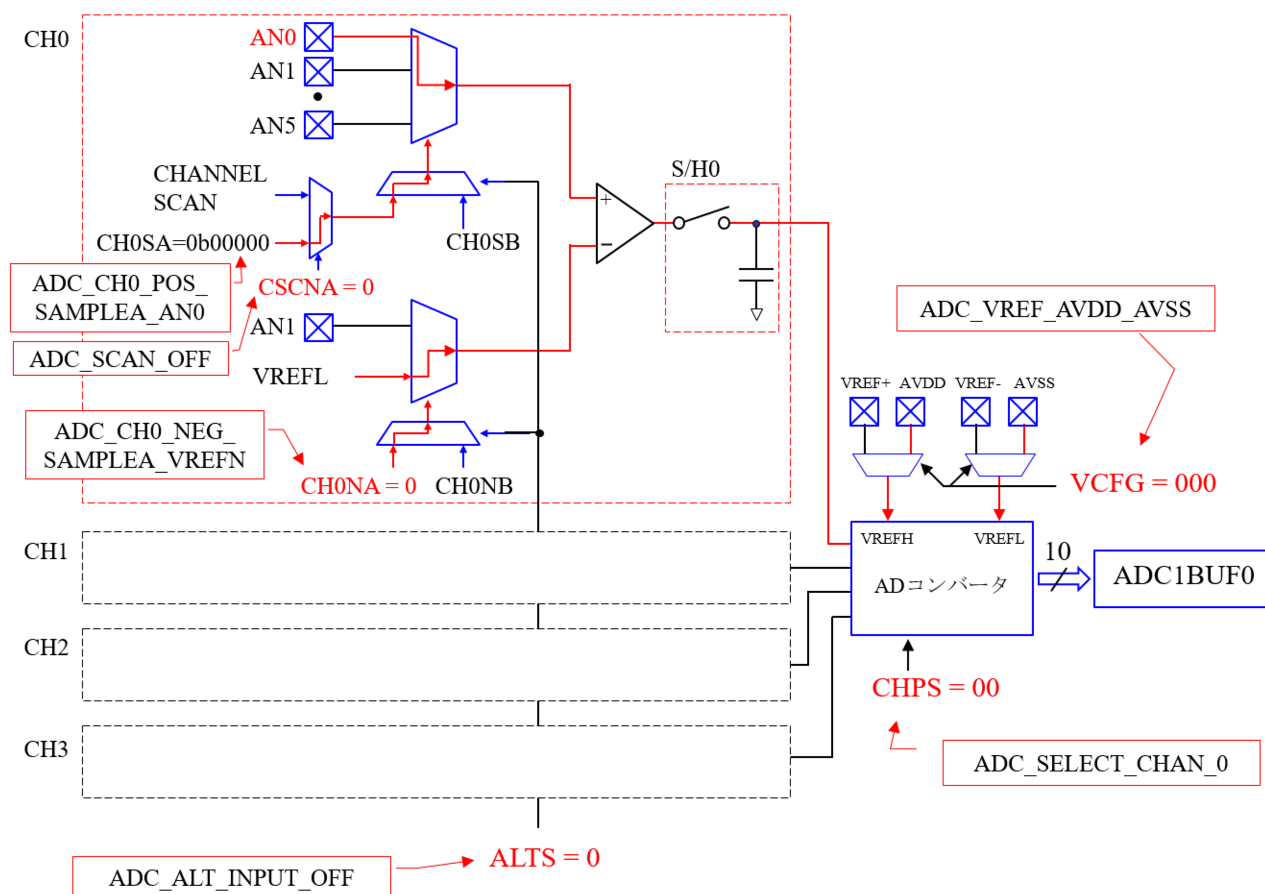


図 5.57: AD 変換モジュールの設定結果

= 0b1xx : ADREF+ = AVDD, ADREV- = AVSS

AD 変換器の基準電圧設定を行う．図 5.57 の右にあるように，基準電圧の高電位側 VREFH を AVDD, 定電位側 VREFL を AVSS に設定する．図 5.3 より，AVDD は 28 番ピン，AVSS は 27 番ピンである．dsPIC33F Family Reference Manual の 16.5 Selecting the Voltage Reference Source によると external voltage reference pins は AN0, AN1 を兼ねることができるとある．

ADC_SCAN_OFF = 0xEBFF = 0b1110 1011 1111 1111

CSCNA = 1: Scan モード使用

= 0 : Scan モード不使用

により，スキャンモードを使わない．Scan モードについては 5.7.6 項を参照されたい．

ADC_SELECT_CHAN_0 = 0xECFF = 0b1110 1100 1111 1111

CHPS = 0b1x : CH0〜CH3 を選択

= 01 : CH0, CH1 を選択

= 00 : CH0 を選択

AD12B = 0 (ADC_AD12B_10BIT) のとき，4 チャネル CH0〜CH3 を利用できる．
AD12B = 1 (ADC_AS12B_12BIT) のとき，CH0 のみを利用できる．

ADC_DMA_ADD_INC_1 = 0xEFC3 = 0b1110 1111 1100 0011

SMPI = 0bxxxx : Scan モードにおいて，スキャンするチャネル数と同じ値を指定する．

= 0b0000 : Scan モードでなければ 0 とする．

DMA の Scan モード (図 5.57 の CHANNEL SCAN，5.7.6 項参照）を利用する場合に，スキャンするチャネル数を入力する．

ADC_ALT_BUF_OFF = 0xEFFD = 0b1110 1111 1111 1101

BUFM = 1 : DMA RAM バッファの半分および全部が満たされた両時点で割り込み

= 0 : バッファ全部が満たされた時点のみで割り込み.

DMA 使用において，AD 変換結果が DMA RAM バッファの半分を満たした時点と全部を満たした時点の両方で割り込み処理を起動する (BUFM = 1) か，全部を満たした時点のみで割り込み処理を起動する (BUFM = 0) かを設定する．

ADC_ALT_INPUT_OFF = 0xEFFE = 0b1110 1111 1111 1110

ALTS = 1 : 図 5.57 にて，(CHCNA, CH0NA,· · ·) と (CH0SB, CH0NB, · · ·) をサンプル毎に切替える．

= 0 : 常に HCNA, CH0NA, · · · を用いる．

どのチャネルも A を選択する設定としている．CH0 では CSCNA = 0 のとき，CH0SA が選択される．また，CH0NA が選択される．さらに，CH0SA = 0b00000 のとき，AN0 がバッファアンプの＋側につなげられる．そして，CH0NA = 0 のとき，VREFL（VCFG = 000 より，AVSS) がバッファアンプの－側につなげられる．

図 5.58 は，AD1CHS0 レジスタである．

AD1CHS0

15	14	13	12	11	10	9	8
CH0NB	-	-	CH0SB				

7	6	5	4	3	2	1	0
CH0NA	-	-	CH0SA				

図 5.58: AD1CHS0 レジスタ

ADC_CH0_POS_SAMPLEA_AN0 = 0xFFE0 = 0b1111 1111 1110 0000

CH0SA = 0b00101：CH0 のバッファアンプの＋側に AN5 をつなげる．

= ⋯

= 0b00000：CH0 のバッファアンプの＋側に AN0 をつなげる．

この設定は，AD1CON2 レジスタの CHCNA = 0 (ADC_SCAN_OFF) のときに有効となる．

ADC_CH0_NEG_SAMPLEA_VREFN = 0xFF7F = 0b1111 1111 0111 1111

CH0NA = 1：CH0 のバッファアンプの－側に AN1 をつなげる．

= 0：CH0 のバッファアンプの－側に VREFL をつなげる．

AD1CON3

15	14	13	12	11	10	9	8
ADRC	-	-	SAMC				

7	6	5	4	3	2	1	0
ADCS							

図 5.59: AD1CON3 レジスタ

図 5.59 は，AD1CON3 レジスタである．図 5.60 の AD 変換モジュールのクロックのブロック図とデータシートを参照しながら，図 5.54 の AD1CON3_set による設定内容を以

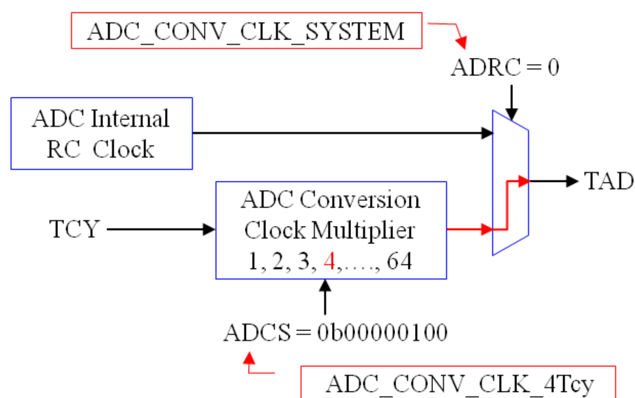


図 5.60: AD 変換モジュールのクロックのブロック図

下にまとめる．

ADC_CONV_CLK_SYSTEM = 0x7FFF = 0b0111 1111 1111 1111

ADRC = 1 : スリープモードのときに ADC 内蔵 RC クロックを利用する．

= 0 : デバイスインストラクションクロック FCY を利用する．

AD 変換モジュールのクロック（周期 TAD）をデバイスインストラクションクロック（周期 TCY = 1/FCY = 1/40.5 [MHz]) から得る設定．

ADC_SAMPLE_TIME_2 = 0xE2FF = 0b1110 0010 1111 1111

SAMC = 0b11111 : 31 TAD

= ⋯

= 0b00010 : 2 TAD

= 0b00001 : 1 TAD

= 0b00000 = 0 TAD

2 × TAD とする設定．ADC Conversion(10-bit Mode) Timing Requirement によると AD1CON1 レジスタの ADC_CLK_AUTO (SSRC = 0b111) の設定においては，TSAMP ≥ 2 TAD としなければならない．

ADC_CONV_CLK_4Tcy = 0xFF03 = 0b1111 1111 0000 0011

ADCS = 0b0011 1111 : TCY(ADCS + 1) = 64TCY = TAD

= ⋯

= 0b0000 0011 : 4TCY = TAD

= 0b0000 0010 = 3TCY = TAD

$$= 0b0000\ 0001 = 2TCY = TAD$$
$$= 0b0000\ 0000 = TCY = TAD$$

ADC Conversion(10-bit Mode) Timing Requirement によると VDD $= 3.0 \sim 3.6$ [V] において TAD > 76 [nsec] と設定しなければならない．TAD は AD 変換器のクロック周期である．この周期設定が ADCS ビットによりなされる．

$$TAD = TCY(ADCS + 1) > 76[nsec] \tag{5.23}$$

である．TCY $= 1/40.5$[MHz] のとき，

$$\begin{aligned} ADCS &> \frac{76[nsec]}{TCY} - 1 \\ &= 76[nsec] \times 40.5[MHz] - 1 \\ &= 2.08 \end{aligned} \tag{5.24}$$

と求められる．そこで，ADCS $= 3$ と設定している．

AD1PCFGL

15	14	13	12	11	10	9	8
-	-	-	-	-	-	-	PCFG8

7	6	5	4	3	2	1	0
PCFG7	PCFG6	PCFG5	PCFG4	PCFG3	PCFG2	PCFG1	PCFG0

図 5.61: AD1PCFGL レジスタ

図 5.61 は，AD1PCFGL レジスタの各ビットに付けられた名前を示す．

ENABLE_AN0_ANA = 0xFFFE = 0b1111 1111 1111 1110

PCFG0 = 1 : 2 番ピンをディジタルモード設定とする．

= 0 : AN0 (2 番ピン）をアナログモード設定とする．図 5.30 のディジタル入力としての使用不可とする．

AD 変換モジュールの設定関数を図 5.62 に示す．これにより，AD 変換モジュールの各レジスタに図 5.54 の設定値を格納する．

```
void OpenADC802(unsigned int config1, unsigned int config2, unsigned int config3,
          unsigned int config4, unsigned int configport, unsigned int configscan)
{
   AD1PCFGL = configport;
   AD1CSSL = configscan;

   AD1CON4 = config4;
   AD1CON3 = config3;
   AD1CON2 = config2;
   AD1CON1 = config1;
}


void SetChanADC802_10BIT(unsigned int channel0, unsigned int channel123)
{
   AD1CHS0 = channel0;
   AD1CHS123 = channel123;
}
```

図 5.62: AD 変換モジュール設定関数

5.6.3 タイマ 3 による AD 変換のサンプリング周期設定と変換終了時割り込み ― タイマ 3 による 1 [kHz] サンプリング ―

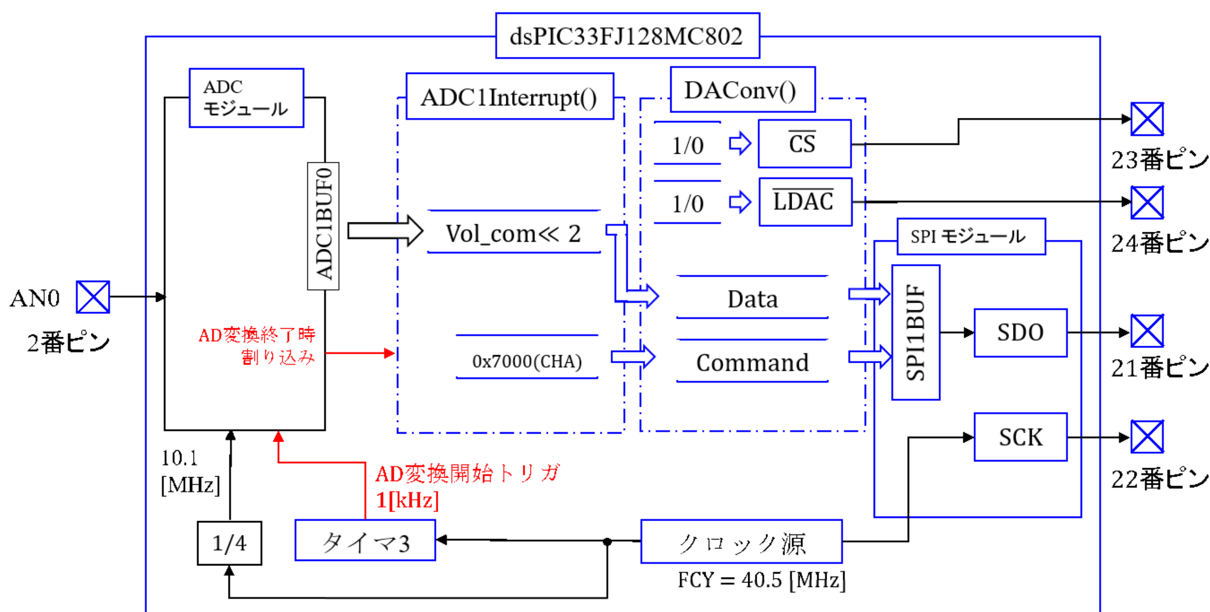


図 5.63: AD 変換終了時割り込みプログラム ADC_ADC1Interrupt.c のブロック図

前項のプログラムは，図 5.54 の AD 変換モジュールの初期設定において，AD1CON1 レジスタの SSRC ビットを ADC_CLK_AUTO (= 0b111) とすることで，自動的に AD 変換を繰り返すモードとしていた．そして，タイマ 1 により 1[ms] 毎に図 5.53 のタイマ 1 割り込み処理関数を起動して，この処理関数がその瞬間に ADC1BUF0 にある AD 変換値を読み出して，DA 変換器にその値を出力していた．

AD 変換を 1[ms] に 1 回実行させ，AD 変換終了時に割り込みをかける方法がある．

New Project として，ADC_ADC1Interrupt を作ってください．そして，圧縮フォルダ内の ADC_ADC1Interrupt フォルダから，新たに作られた MPLABXProjets¥ADC_ADC1-Interrupt.X フォルダ内に ADC_ADC1Interrupt.c, ADC802.c, SPI802.c, timer3.c のファイルと inclued ファイルをコピーしてください．そして，ADC_ADC1Interrupt.c, ADC802.c, SPI802.c, timer3.c のファイルを Source Files に付加し，indlude フォルダ内のヘッダファイルを Header Files に付加してください．

図 5.63 は AD 変換終了時に main() 関数に割り込みをかけ，ADC1Interrupt() 関数を起動するプログラムのブロック図である．AD 変換モジュールにはタイマ 3 が AD 変換開始トリガをかける．

```
//タイマ周期設定
const unsigned int      tsamp1 =  40000;
          // Timer3 AD変換繰り返し周波数設定  40MHz/tasmp1 = 1kHz

//メイン関数
int main(void)
 {
     （省略）

     init_SPI();
     init_timer3();
     init_ADC();

     while(1){}          //メインループ
}
```

図　5.64: AD 変換終了時割り込み処理のメイン関数

```
// タイマ3  設定
void init_timer3()
{
   extern const unsigned int tsamp1;
              // ADC_ADC1Interrupt.cにて定義されている.

   unsigned int TM3Config = T3_ON & T3_GATE_OFF & T3_PS_1_1
                                        &  T3_SOURCE_INT;
   OpenTimer3(TM3Config, tsamp1-1);
              //タイマ1設定  サンプリング周期  (1/40MHz) x  tsamp1  sec
}
```

図　5.65: AD 変換の繰り返し周波数設定用タイマ 3 関数

図 5.64 はメイン関数の抜粋である．図 5.53 に対して，タイマ 3 の初期化関数 init_timer3() を用いている点が異なる．なお，

const unsigned int　tsamp1 = 40000;

のように，const を付して，tsamp1 をグローバル定数として定義している．この定義を受けて，図 5.65 のタイマ 3 関数では

extern const unsigned int tsamp1;

として，init_timer3() 関数の外部で定義されている tsamp1 を使うことを宣言している．このプログラムではタイマ 3 関数による割り込みは行わないので，割り込みの設定は無い．

図 5.66 は AD 変換モジュールの初期設定関数の抜粋である．また，図 5.67 は割り込み設定関数である．図 5.54 の AD 変換モジュールの初期設定では，AD1CON1_set = ADC_CLK_AUTO として，自動的に AD 変換を繰り返すモードにしていた．図 5.66 では，

```
//AD変換モジュール初期設定関数
void init_ADC(void)
{
    unsigned int AD1CON1_set = ADC_MODULE_ON & ADC_IDLE_CONTINUE
            & ADC_ADDMABM_ORDER & ADC_AD12B_10BIT
            & ADC_FORMAT_INTG & ADC_CLK_TMR & ADC_MULTIPLE
            & ADC_AUTO_SAMPLING_ON & ADC_SAMP_ON;
    unsigned int AD1CON2_set = ADC_VREF_AVDD_AVSS & ADC_SCAN_OFF
            & ADC_SELECT_CHAN_0 &  ADC_DMA_ADD_INC_1
            & ADC_ALT_BUF_OFF & ADC_ALT_INPUT_OFF;
    unsigned int AD1CON3_set = ADC_CONV_CLK_SYSTEM
            & ADC_SAMPLE_TIME_2 & ADC_CONV_CLK_4Tcy;
    unsigned int AD1CON4_set = 0x0;  // 未使用
    unsigned int AD1PCFGL_set = ENABLE_AN0_ANA;
    unsigned int AD1CSSL_set = 0x0;  // 未使用
    unsigned int AD1CHS0_set = ADC_CH0_POS_SAMPLEA_AN0
            & ADC_CH0_NEG_SAMPLEA_VREFN;
    unsigned int AD1CHS123_set = 0x0; // 未使用
    unsigned int ADCIntConfig =  ADC_INT_PRI_7 & ADC_INT_ENABLE;

    OpenADC802(AD1CON1_set, AD1CON2_set, AD1CON3_set, AD1CON4_set,
                AD1PCFGL_set, AD1CSSL_set);
    SetChanADC802_10BIT(AD1CHS0_set, AD1CHS123_set);
    ConfigIntADC802(ADCIntConfig);
}
```

図　5.66: AD 変換モジュール初期設定関数（AD 変換終了時割り込み処理）

AD1CON1_set = ADC_CLK_TMR

に変更している．adc802.h のヘッダファイルによると

ADC_CLK_TMR = 0xFF5F = 0b1111 1111 0101 1111

SSRC = 0b111 : 自動繰り返しサンプリング
= 0b101 : PWM2 同期サンプリング
= 0b100 : タイマ 5 同期サンプリング
= 0b011 : PWM1 同期サンプリング
= 0b010 : タイマ 3 同期サンプリング
= 0b001 : INT0(16 番ピン) 同期サンプリング
= 0b000 : SAMP = 0 によりサンプル & ホールド → AD 変換開始

である．タイマ 3 のブロック図は，このプログラムの使用の範囲では，図 5.24 のタイマ 1 のブロック図において，TMR1, PR1, T1IF を TMR3, PR3, T3IF に置き換えたものと同じである．クロックによりカウントアップされる TMR3 の値と，PR3 レジスタ（Tsamp1 - 1 の値が格納されている．）の値が一致したときに，図 5.57 の AD 変換モジュールにおいてアナログ値をサンプル & ホールドして，AD コンバータによりディジタル値へと変

```
//AD変換モジュール割り込み設定関数
void ConfigIntADC802(unsigned int config)
{
      IFS0bits.AD1IF = 0;                    /* Clearing the Interrupt Flag bit */
      IPC3bits.AD1IP = config & 0x07;        /* Setting Priority */
      IEC0bits.AD1IE = (config & 0x08)>>3;   /* Setting the Interrupt enable bit */
}
```

図 5.67: AD 変換終了時割り込み設定関数

IPC3

15	14	13	12	11	10	9	8
-	-	-	-	-	DMA1IP		

7	6	5	4	3	2	1	0
-	AD1IP			-	U1TXIP		

図 5.68: IPC3 レジスタ

換を開始する．

AD 変換終了時に割り込みを行うために，

ADCintConfig = ADC_INT_PRI 7 & ADC_INT_ENABLE

により，割り込みを設定している．adc802.h のヘッダファイル，データシートの INTERRUPT PRIORITY CONTROL REGISTER 3 によると

ADC_INT_PRI_7 = 0xFFFF = 0b 1111 1111 1111 1111

AD1IP = 111：優先度 7（最高）

= ···

= 001：優先度 1（最低）

= 000：割り込みをしない．

と優先度を定義している．

図 5.68 は IPC3 レジスタである．AD1IP bits は，IPC3 レジスタの第 4〜6 ビットにある．ADC_INT_PRI_7 とは桁がずれている．図 5.67 の ConfigIntADC802() 関数において

IPC3bits.AD1IP = config & 0x07;

として，ADC_INT_PRI_7 の下 3 桁を取り出して，AD1IP bits に格納している．なお，このプログラムでは割り込みは 1 つしかないので，優先度がいくつであっても関係ない．複数の割り込みを使い分ける場合に，この優先度設定が重要となる．

```
//AD変換終了時割り込み処理関数
void __attribute__((interrupt, no_auto_psv)) _ADC1Interrupt(void)
{
    unsigned int i;
      i = ADC1BUF0<<2;
      DAConv(i, 0x7000);          // チャネルA選択

    _AD1IF = 0;                   // 割り込みフラグクリア
}
```

図 5.69: AD 変換終了時割り込み処理関数

ADC_INT_ENABLE = 0xFFFF = 0b1111 1111 1111 1111

AD1IE = 1：割り込み可とする．

= 0：割り込み不可とする．

の割り込みを可とする設定は，

IEC0bits.AD1IE = (config & 0x08)>>3;

により，第 4 ビットの値を取り出して，一桁目に移動させ，IEC0 レジスタに格納することで行っている．

AD 変換終了時に，AD 変換モジュールはメイン関数に割り込みをかけて，図 5.69 の割り込み処理関数を起動する．そして，アナログ値のサンプリングを開始して，タイマ 3 からの次のトリガ信号を待つ．割り込み処理関数は AD 変換結果が格納されている ADC1BUF0 レジスタから，変換結果を読み出して，DA 変換器に転送する．その後に，AD1IF フラグを 0 にクリアして，次の割り込みを受け付け可能として，割り込み処理を終了する．プログラム処理はふたたびメイン関数へと戻って，while() 文の無限ループが繰り返される．

このプログラムの実行結果は図 5.51 と全く同じである．

5.6.4 Autoconversion による高速サンプリング

New Project として，ADC_ADC1Interrupt_autoconversion を作ってください．そして，圧縮フォルダ内の ADC_ADC1Interrupt_autoconversion フォルダから，新たに作られた MPLABXProjets¥ADC_ADC1Interrupt_autoconversion.X フォルダ内に ADC_ADC1-Interrupt_autoconversion.c，ADC802.c のファイルと inclued ファイルをコピーしてください．そして，ADC_ADC1Interrupt_autoconversion.c，ADC802.c のファイルを Source Files に付加し，indlude フォルダ内のヘッダファイルを Header Files に付加してください．

データシートによると 10 ビット AD 変換器は最高で 1.1 [Msps] の変換速度を実現でき

るとある．実際にこの性能が出せるのか本項で挑戦する．まず，図 5.70 のように，AD 変換モジュールの初期設定を変更する．図 5.66 の初期設定プログラムからの変更箇所を朱色で示す．

ACD_CLK_TMR → ADC_CLK_AUTO

と変更し，これと ADC_AUTO_SAMPLING_ON により，AD 変換モジュールを自動で繰り返し AD 変換を行う設定にする．また，(5.24) 式より，ADCS > 2.08 であった．そこで，前々項，前項では ADCS $= 3$ として，

ADC_CONV_CLK_4Tcy

を選定していた．本項では，データシートの制約をわずかに下回ることになるが，ADCS $= 2$，

ADC_CONV_CLK_3Tcy

として，どこまで速いサンプリングができるか調べる．

```
//AD変換モジュール初期設定関数
void init_ADC(void)
{
   unsigned int AD1CON1_set = ADC_MODULE_ON & ADC_IDLE_CONTINUE
                              & ADC_ADDMABM_ORDER  & ADC_AD12B_10BIT
                              & ADC_FORMAT_INTG & ADC_CLK_AUTO & ADC_MULTIPLE
                              & ADC_AUTO_SAMPLING_ON & ADC_SAMP_ON;
   unsigned int AD1CON2_set = ADC_VREF_AVDD_AVSS & ADC_SCAN_OFF
                              & ADC_SELECT_CHAN_0  &  ADC_DMA_ADD_INC_1
                              & ADC_ALT_BUF_OFF & ADC_ALT_INPUT_OFF;
   unsigned int AD1CON3_set = ADC_CONV_CLK_SYSTEM
                              & ADC_SAMPLE_TIME_2  & ADC_CONV_CLK_3Tcy;
   （未使用）
}
```

図 5.70: Autoconversion による高速サンプリング用 AD 変換モジュール設定

図 5.71 は AD 変換終了時の割り込みにより起動される関数である．図 5.53 および図 5.69 の割り込み処理関数では，DAconv() 関数を実行して，DA 変換器に AD 変換結果を出力していた．本項では DAConv() 関数は用いない．DA 変換器へのデータ転送に要する時間は図 5.49 より約 750 [ns] である．AD 変換における 10 ビット × 3Tcy と同程度であるが，実測すると DAConv() 関数の実行に約 1.1[μ s] かかってしまい，高速サンプリングのモニタとしては使えないことが分かった．そこで，図 5.71 の割り込み処理関数では，AD 変換が終了してこの関数が起動される度に vol_data[i] に $i = 0$ から順に AD 変換結果を格納することにした．データが 100 個を超えた場合は，$i = 0$ にリセットして，データを上書きしている．i は割り込み処理関数の外で

```
unsigned int i = 0;
unsigned int vol_data[1000];

//AD変換終了時割り込み処理関数
void __attribute__((interrupt, no_auto_psv)) _ADC1Interrupt(void)
{
    _LATA4 = 1;                    // RA4(12番ピン)に 1 を出力する.

    vol_data[i] = ADC1BUF0;        // vol_data[i] にAD変換結果を格納する.
    i++;
    if(i>=100)
    {
        i = 0;
    };

    _LATA4 = 0;                    // RA4(12番ピン)に 0 を出力する.

    _AD1IF = 0;            // 割り込みフラグクリア
}
```

図 5.71: Autoconversion による高速サンプリング用 AD 変換割り込み処理関数

unsigned int i =0;

により，グローバル変数として宣言することで，割り込み処理関数が終了しても，i の値が保持される設定としている．vol_data[] の値も同様にグローバル変数として宣言している．

また，AD 変換のサンプリング周波数を知るために

_LATA4 = 1;

_LATA4 = 0;

により，割り込み処理関数実行中は RA4(12 番ピン) に 1 を出力ことにする．プログラム実行中にこの 1 の出力期間を計測したところ，約 370 [ns] であった．もし，1.1 [Msps] のサンプリングが実現できた場合，1 サンプリングの所要期間は 909 [ns] であり，割り込み処理関数の実行時間 370 [ns] は十分に短く，高速 AD 変換の妨げとならない．

図 5.72 は Autoconversion による高速サンプリングプログラムのブロック図である．AD 変換モジュールにて AD 変換が終了する毎にメイン関数の while() 文（無限ループを実行している.）に割り込みがかけられ，割り込み処理関数が起動される．割り込み処理関数は AD 変換結果が格納されている ADC1BUF0 レジスタからデータを読み出して，vol_data に格納する．そして，その処理の間，1 の値を 12 番ピンに出力する．

RA4(12 番ピン) の 1/0 出力の繰り返し周波数（AD 変換モジュールのサンプリング周波数）を計測したところ，960.34 [kHz] が得られた．図 5.73 は，高速サンプリングの結

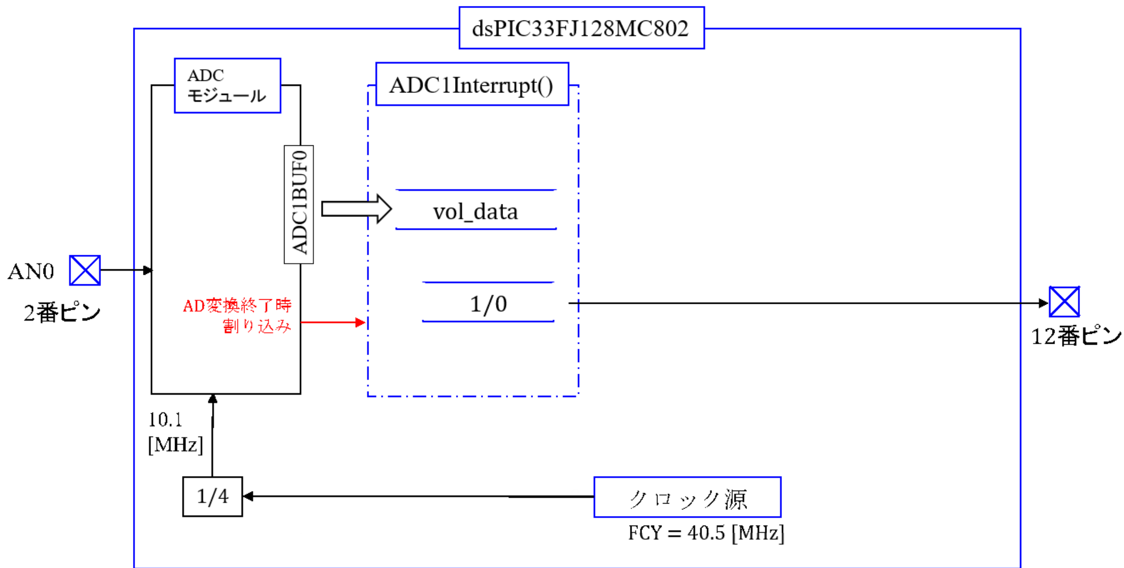


図 5.72: Autoconversion による高速サンプリングプログラムのブロック図

果，vol_data に格納されていたデータを，csv 形式で取り出して Excel によりグラフ化した結果である．信号発生器により 10[kHz] の正弦波信号を発生させ，マイコンの AN0(2 番ピン) に印加して，プログラムを実行した．Vol_data には 100 個のサンプル値が格納される．100 個のデータのサンプルに要する時間 $T_{100samp}$ は

$$\begin{aligned} T_{100samp} &= \frac{100}{960.34[\mathrm{kHz}]} \\ &= 104.1[\mu\mathrm{s}] \end{aligned} \tag{5.25}$$

である．一方，10 [kHz] の信号の 1 周期の時間 $T_{10\mathrm{kHz}}$ は

$$\begin{aligned} T_{10kHz} &= \frac{1}{10[\mathrm{kHz}]} \\ &= 100[\mu\mathrm{s}] \end{aligned} \tag{5.26}$$

である．図 5.73 では 57 番目のデータと 58 番目のデータの間に段差が見られる．これは，57 番目のデータが vol_data に格納された時点でマイコンが一時停止されたことによる．$T_{10\mathrm{kHz}} < T_{100samp}$ であるため，マイコンは信号の 1 周期より多く（約 4 個分）のデータをサンプルしている．

1.1[Msps] のサンプリング周波数は実現できなかった．そこで，その理由を以下に考察する．コンフィギュレーション設定にて

_FOSC(OSCIOFNC_OFF)

とすることで，OSC2 (10 番ピン) にシステムクロックを出力できる．このクロック周波

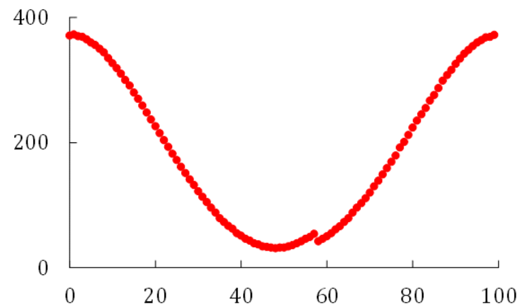


図 5.73: 高速サンプリングによる AD 変換結果 (vol_data の値)，信号周波数 = 10 [kHz], サンプリング周波数 = 960[kHz]

数を計測したところ，40.33 [MHz] であった．これより，1 回の AD 変換に要するシステムクロック数 ADclk は

$$\begin{aligned} \mathrm{ADclk} &= \frac{40.33[\mathrm{MHz}]}{960.34[\mathrm{kHz}]} \\ &= 42 \end{aligned} \tag{5.27}$$

であることが分かった．

システムクロックの周期を Tcy とすると，AD 変換モジュールのクロック周期 TAD は，ADC_CONV_CLK_3Tcy より，

TAD = 3 Tcy

である．データシートの ADC CONVERSION (10-BIT MODE) TIMING CHARACTERISTIC) によると，SSRC = 111 (ADC_CLK_AUTO), TSAMP = 2TAD (ADC_SAMPLE_TIME_2) のとき，AD 変換に要するクロック周期 $\mathrm{T_{ADConv}}$ は

$$\begin{aligned} \mathrm{T_{ADConv}} &= \mathrm{2TAD(TSAMP) + 0.5TAD + 10TAD(10\ ビット\ AD\ 変換) + 1.5TAD} \\ &= \mathrm{14TAD} \\ &= \mathrm{42Tcy} \end{aligned} \tag{5.28}$$

と求まる．実験結果のクロック数 42 はこのデータシートから読み取れるクロック数と合っていることがわかった．

したがって，本項の AD 変換のやり方では 1.1[Msps] のサンプリング周波数は実現できない．

5.6.5　Debugger の利用

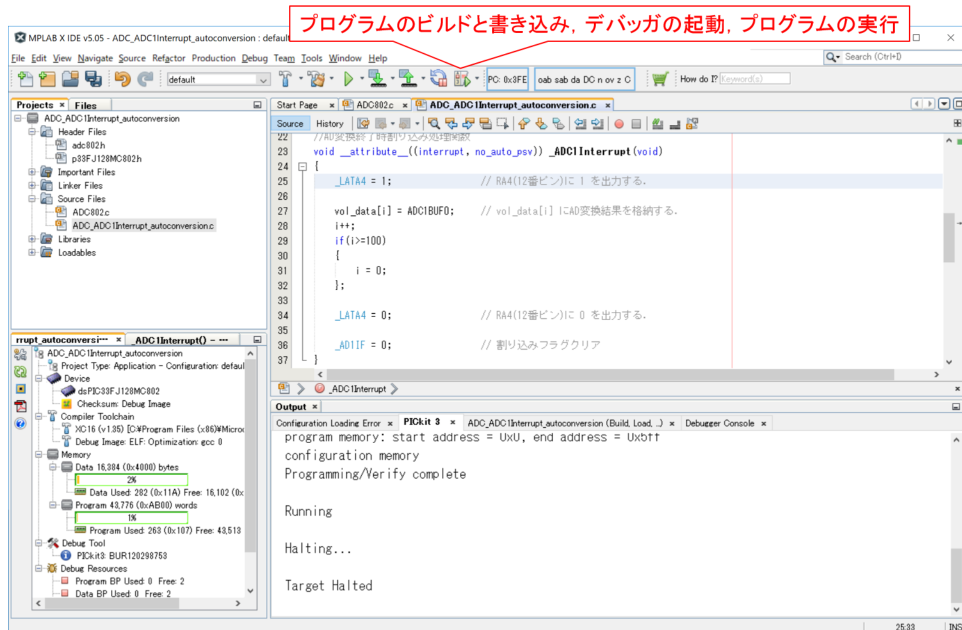


図　5.74: デバッガの起動

前項の vol_data の値は，MPLAB X IDE のデバッガ機能を利用した．図 5.74 に示すボタンを左クリックすることで，プログラムのビルドとマイコンへの書き込み，デバッガの起動，プログラムの実行を行うことができる．図 5.75 の一時停止ボタンにより，プログラム実行を一時停止にすると，vol_data の値を読み出すことが出来る．図中の Variables のタブを左クリックすると，図 5.76 の表示データの入力画面に切り替わる．vol_data と入力し，図 5.77 のように，＋ボタンを左クリックすることで vol_data の一覧を見ることができる．ただし，データは 16 進数形式で表示されている．vol_data を右クリックすると，メニューウィンドウが現れる．Diplay Value Column As → Decimal と進んで左クリックすると，データの表示形式が図 5.78 のように 10 進数に切り替わる．もう一度 vol_data を右クリックし，Export Data → CSV File → Displayed Format と選択して，左クリックすることで，CSV 形式でデータをパソコン等にセーブできる．

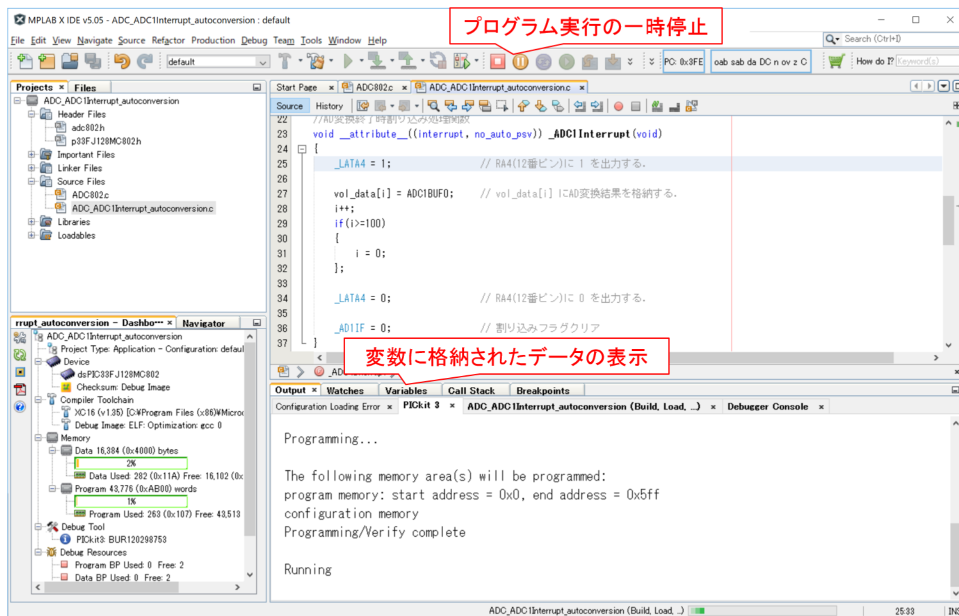


図　5.75: デバッガの一時停止

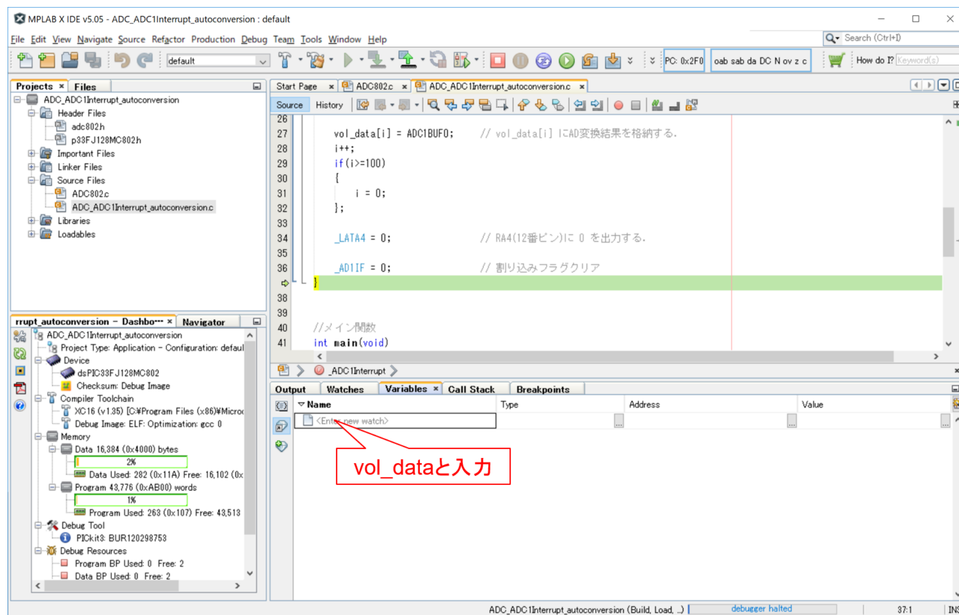


図　5.76: 表示データの指定

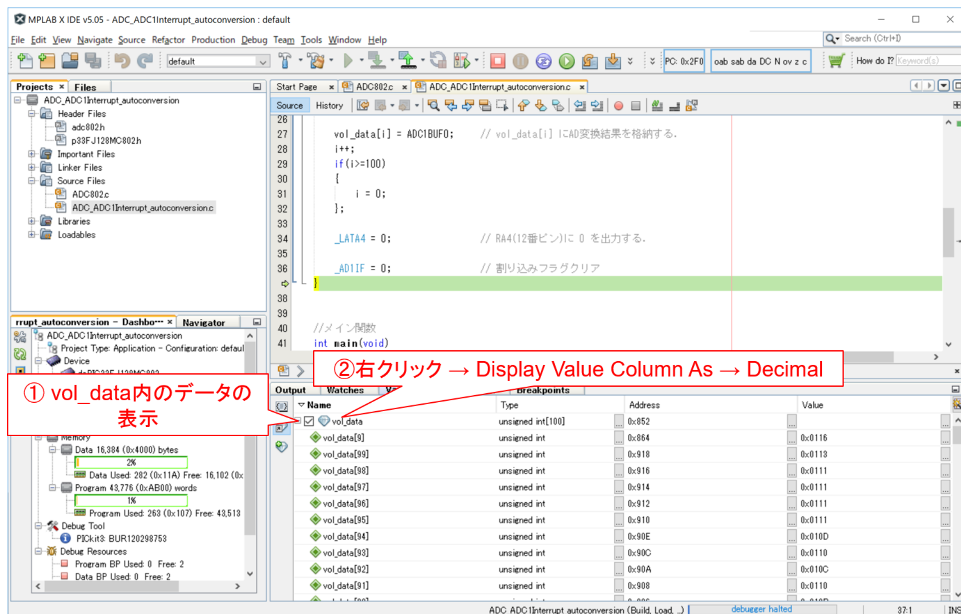


図　5.77: 表示データ形式を 10 進数に指定

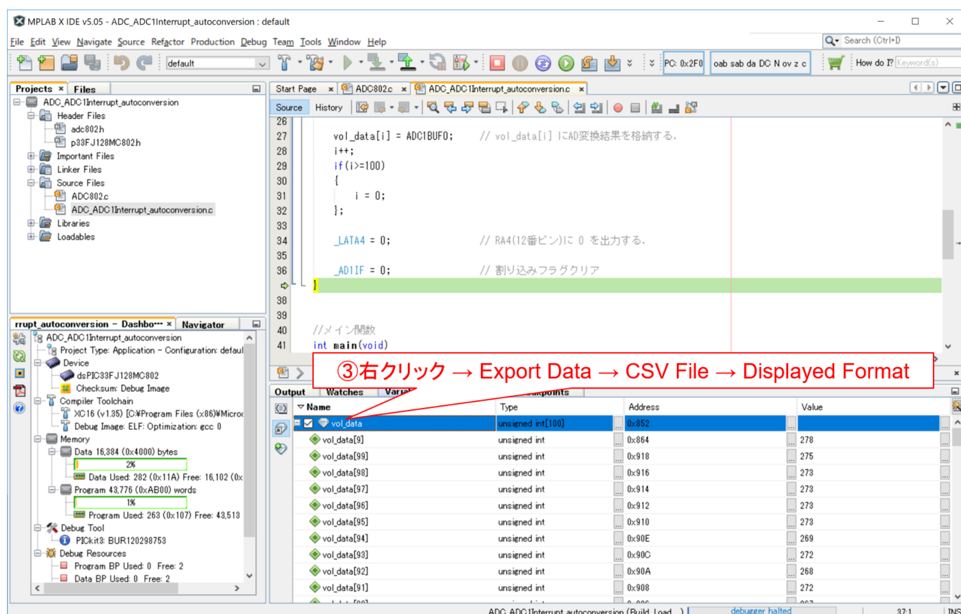


図　5.78: CSV ファイルへ出力

5.6.6 1.1[Msps] の実現

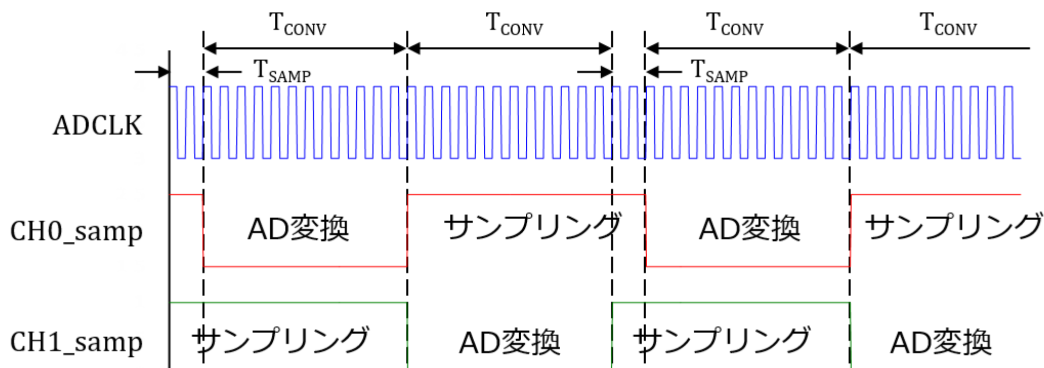


図 5.79: タイミングチャート (2CH sequential sampling)

前項では960 [kHz] のサンプリング周波数しか実現できないかった．dsPIC33FJ128MCX のデータシートを目を皿のようにして読んでも説明を見つけられない．そこで，dsPIC33F Family Reference Manual - Section 16. Analog-to-Digtal Converter(ADC) をネットよりダウンロードし，MULIPLE CHANNELS WITH SEQUENTIAL SAMPLING のタイミングチャートを基に，2CH をシーケンシャル（1CH ずつ順番）にサンプリングして autoconversion する場合のタイミングチャートを描くと図 5.79 が得られた．ここで，ADCLK は図 5.60 のブロック図の出力（周期 TAD のクロック）である．CH0_samp が 1 のとき，チャネル CH0 のサンプル & ホールド回路 S/H0（図 5.82 参照）のスイッチがオンとなり，入力のアナログ電圧がコンデンサに印加される．dsPIC33FJ128MCX のデータシートの ADC CONVERSION (10-BIT MODE) TIMING REQUIREMENTS によると，この電圧が安定するための待ち時間 $\mathrm{T_{SAMP}}$ を，

$$\mathrm{T_{SAMP}} \geq 2\mathrm{TAD} \tag{5.29}$$

としなければければならないとある．CH0_samp が 0 になると，スイッチがオフとなり，その瞬間のアナログ電圧がコンデンサに保持される．そして，その直後の ADCLK の立ち下がり (0.5 クロック後) から AD 変換が開始される．そして，この立ち下がりを含む 10 個の立ち下がり信号を受けて 10 ビットのディジタル値が得られる．その後，1.5 クロックを経て AD 変換を終了する．AD 変換に要するクロック数は 12，所要時間 $\mathrm{T_{CONV}}$ は

$$\mathrm{T_{VONV}} = 12\mathrm{TAD} \tag{5.30}$$

である．この終了後に CH0_samp は再び 1 となり，サンプル & ホールド回路のスイッチが閉じられてアナログ電圧がコンデンサに印加される．また，同時に CH1_samp の値が 0 となり，CH1 の AD 変換が開始される．各チャネルの AD 変換の終了毎に，割り込み処理関数を起動して ADC1BUF0 の値を読み出す．したがって,2CH の AD 変換に要する時間 $\mathrm{T_{AD2CH}}$ は

$$\mathrm{T_{AD2CH}} = (2 + 12 + 12)\mathrm{TAD} = 26\mathrm{TAD} \tag{5.31}$$

となる．両チャネルに同じ入力信号を印加すれば，実質 13TAD のサンプリング周期を得ることができる．このとき，AD 変換のサンプリング周波数 $\mathrm{f_{ADsamp}}$ は

$$\begin{aligned}\mathrm{f_{ADsamp}} &= \frac{40.36[\mathrm{MHz}]}{13 \times 3\,\text{クロック}} \\ &= 1.035[\mathrm{Msps}] \end{aligned} \tag{5.32}$$

である．1.1 [Msps] にもう少しで届きそうである．

dsPIC33F Family Reference Manual - Section 16. Analog-to-Digtal Converter(ADC) に，以下の文章を見つけた．

When using multiple Sample/Hold channels with sequential sampling, programming SAMC for zero clock cycles results in the fastest possible conversion rate.

SAMC = 0b0000 ($\mathrm{T_{SAMP}} = 0$) としてみよう．図 5.80 は $\mathrm{T_{SAMP}} = 0$ が実現できたとした場合のタイミングチャートである．最初のサンプル & ホールド回路のコンデンサ電圧は安定しないままにディジタル値への変換が開始されてしまうが，2 回目以降は別のチャネルの AD 変換が行われている期間がそのチャネルの $\mathrm{T_{SAMP}}$ であるため，電圧安定に十分な時間が確保される．

$\mathrm{T_{SAMP}} = 0$ とできれば，2CH の AD 変換に要する時間 $\mathrm{T_{AD2CH}}$ は

$$\mathrm{T_{AD2CH}} = (12 + 12)\mathrm{TAD} = 24\mathrm{TAD} \tag{5.33}$$

となり，両チャネルに同じ入力信号を印加することで，AD 変換のサンプリング周波数 f_{ADsamp} を

$$\begin{aligned} f_{\mathrm{ADsamp}} &= \frac{40.36[\mathrm{MHz}]}{12 \times 3\,\text{クロック}} \\ &= 1.121[\mathrm{Msps}] \end{aligned} \tag{5.34}$$

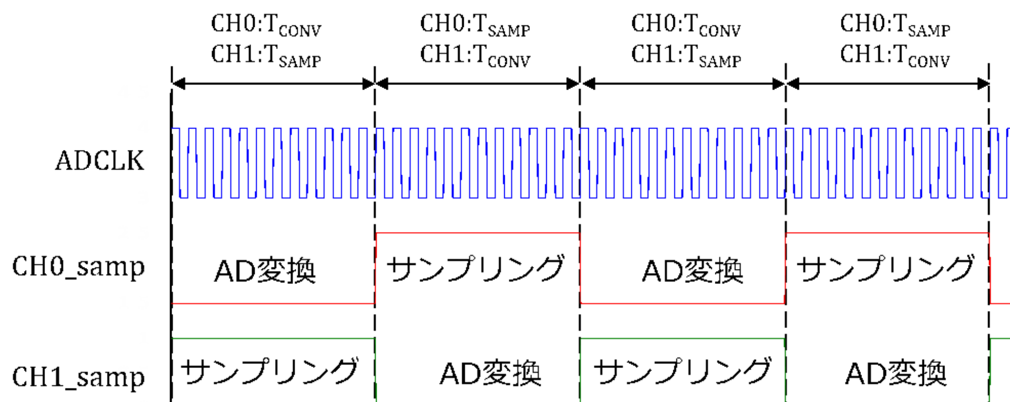


図 5.80: タイミングチャート (2CH sequential sampling, $T_{SAMP} = 0$)

とできるかもしれない！

New Project として，ADC_MUL_1.1Msps を作ってください．そして，圧縮フォルダ内の ADC_MUL_1.1Msps フォルダから，新たに作られた MPLABXProjets¥ADC_MUL_1.1Msps.X フォルダ内に ADC_MUL_1.1Msps.c, ADC802.c のファイルと inclued ファイルをコピーしてください．そして，ADC_MUL_1.1Msps.c, ADC802.c のファイルを Source Files に付加し，indlude フォルダ内のヘッダファイルを Header Files に付加してください．

図 5.81 は，2CH の sequential sampling と autoconverion を実行するための AD 変更モジュール設定を示す．

ADC_CLK_AUTO (SSRC = 0b111)

により，AD 変換開始のトリガ無しで，自動的に AD 変換を繰り返すモード (autoconversion) にする．

ADC_SELECT_CHAN_01 (CHPS = 0b01)

により，CH0 と CH1 を選択する．

ADC_SAMPLE_TIME_0 (SAMC = 0b00000)

により，$T_{SAMP} = 0$ とする．

ADC_CONV_CLK_3Tcy (ADCS = 0b00000010)

により，TAD = 3Tcy とする．

ADC_CH0_POS_SAMPLEA_AN1 (CH0SA = 0b00001)

により，CH0 に AN1 を割り当てる．

ADC_CH123_POS_SAMPLEA_0_1_2 (CH123SA = 0b0)

```
void init_ADC(void)
{
   unsigned int AD1CON1_set = ADC_MODULE_ON & ADC_IDLE_CONTINUE
                              & ADC_ADDMABM_ORDER  & ADC_AD12B_10BIT
                              & ADC_FORMAT_INTG & ADC_CLK_AUTO & ADC_MULTIPLE
                              & ADC_AUTO_SAMPLING_ON & ADC_SAMP_ON;
   unsigned int AD1CON2_set = ADC_VREF_AVDD_AVSS & ADC_SCAN_OFF
                              & ADC_SELECT_CHAN_01  &  ADC_DMA_ADD_INC_1
                              & ADC_ALT_BUF_OFF & ADC_ALT_INPUT_OFF;
   unsigned int AD1CON3_set = ADC_CONV_CLK_SYSTEM
                              & ADC_SAMPLE_TIME_0  & ADC_CONV_CLK_3Tcy;
   unsigned int AD1CON4_set = 0x0;       // 未使用
   unsigned int AD1PCFGL_set = ENABLE_AN0_ANA & ENABLE_AN1_ANA;
   unsigned int AD1CSSL_set = 0x0;  // 未使用
   unsigned int AD1CHS0_set = ADC_CH0_POS_SAMPLEA_AN1
                              & ADC_CH0_NEG_SAMPLEA_VREFN;
   unsigned int AD1CHS123_set = ADC_CH123_POS_SAMPLEA_0_1_2
                              & ADC_CH123_NEG_SAMPLEA_VREFN;

   unsigned int ADCIntConfig =  ADC_INT_PRI_7 & ADC_INT_ENABLE;

   OpenADC802(AD1CON1_set, AD1CON2_set, AD1CON3_set, AD1CON4_set,
                          AD1PCFGL_set, AD1CSSL_set);
   SetChanADC802_10BIT(AD1CHS0_set, AD1CHS123_set);
   ConfigIntADC802(ADCIntConfig);
}
```

図 5.81: 2CH の sequential sampling + autoconversion による最高速サンプリング用 AD 変換モジュール設定

により，CH1 に AN0, CH2 に AN1，CH3 に AN2 を割り当てる．ここで，AN1 は CH0 と CH2 の両方に割り当てられているが，この設定では CH2, CH3 は使用しない．

また，図 5.54 の設定から変わらない設定ではあるが，本項の sequential & autoconversion に関わる 2 つの設定を以下に記す．

ADC_MULTIPLE　(SIMSAM = 0)

により，CH0〜CH4 を順にサンプル & 変換する．ただし，CHPS bits により，CH0 と CH1 のみを選択している．

ADC_AUTO_SAMPLING_ON　(ASAM = 1)

により，AD 変換が終了すると自動的に次のサンプリングを開始する．

図 5.82 は，以上の設定を反映した AD 変換モジュールのブロック図である．CH0 が AN1 に，CH1 が AN0 に接続され，サンプル & ホールド (S/H) 回路を通して AD コンバータに接続されている．S/H 回路の切り替えと AD コンバータの実行は図 5.80 のタイミングチャートの通りになされると期待する．ただし，$T_{SAMP} = 0$ の設定通りになるの

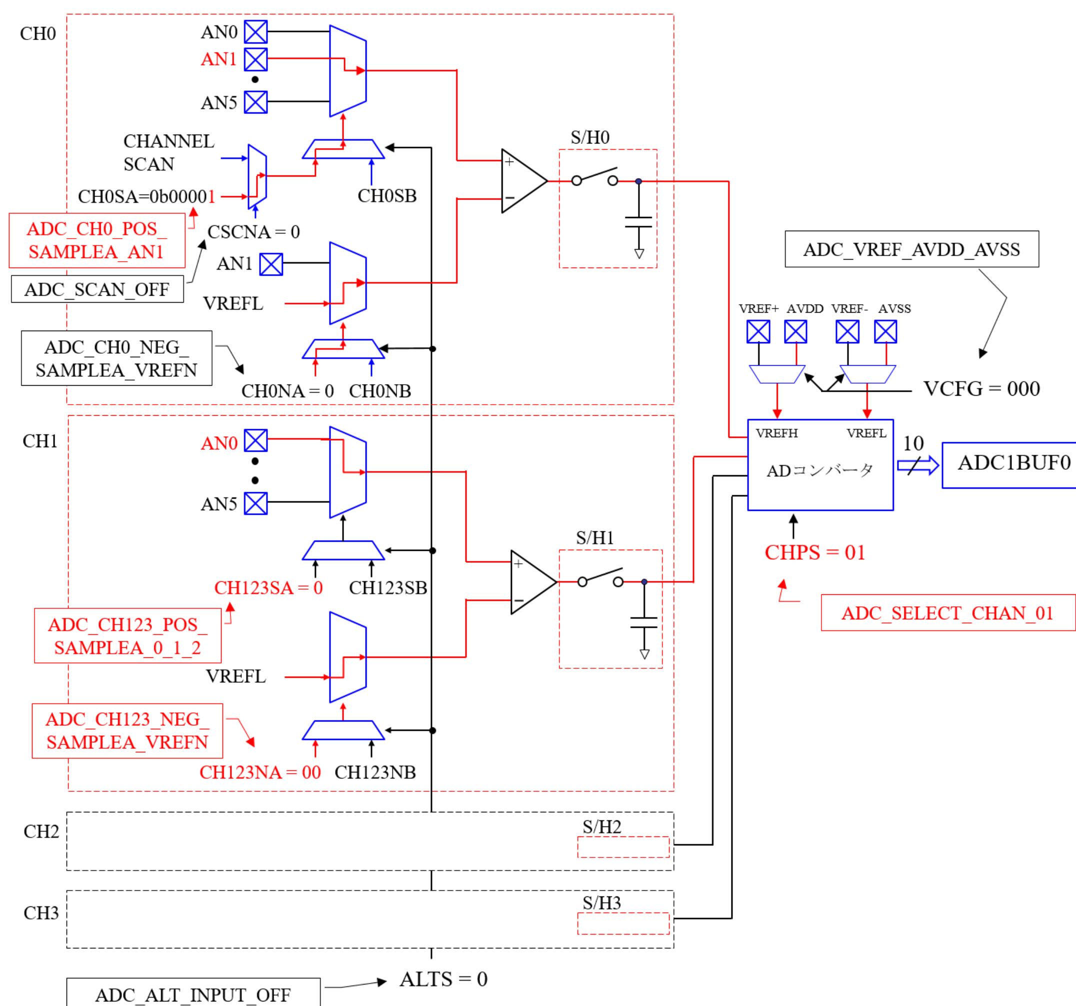


図 5.82: AD 変換モジュールのブロック図 (2CH サンプリング時)

かはわからない．データシートの英文は，SAMC をゼロに設定すれば，最高速の AD 変換を実現できると記載されているだけである．

なお，割り込み処理関数は，図 5.71 と同じである．

実行結果は，システムクロック周波数 FCY = 40.36 [MHz] のとき，

$T_{SAMP} = 1.0762$ [Msps]

であった．ほぼ，1.1[Msps] を実現できた！

サンプリング周期 T_{ADsamp} は

$$
\begin{aligned}
T_{ADsamp} &= \frac{40.36[\mathrm{MHz}]}{3 \times 1.0762[\mathrm{Msps}]} \\
&= 12.5\mathrm{TAD}
\end{aligned}
\qquad (5.35)
$$

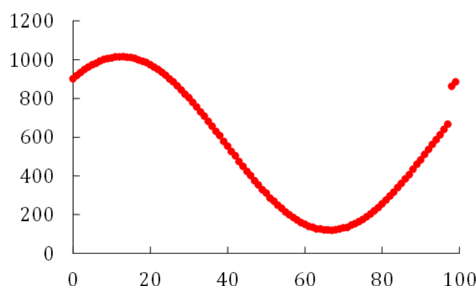


図 5.83: 最高速サンプリングによる AD 変換結果 (vol_data の値)，信号周波数 = 10 [kHz], サンプリング周波数 = 1.076 [MHz]

であった．さらに逆算すると，$\mathrm{T_{SAMP}}$ は，

$$\begin{aligned} \mathrm{T_{SAMP}} &= 2 \times \mathrm{T_{ADsamp}} - 24\mathrm{TAD} \\ &= \mathrm{TAD} \end{aligned} \tag{5.36}$$

であった．SAMC = 0 と設定しても，$\mathrm{T_{SAMP}} = 0$ とはならなかった．

図 5.83 は，vol_data に格納されていたデータを，csv 形式で取り出して Excel によりグラフ化した結果である．信号発生器により 10[kHz] の正弦波信号を発生させ，マイコンの AN0, AN1(2, 3 番ピン) に印加して，プログラムを実行した．Vol_data には 100 個のサンプル値が格納される．100 個のデータのサンプルに要する時間 $T_{100samp}$ は

$$\begin{aligned} T_{100samp} &= \frac{100}{1.0762[\mathrm{Msps}]} \\ &= 92.9[\mu\mathrm{s}] \end{aligned} \tag{5.37}$$

である．一方，10 [kHz] の信号の 1 周期の時間 T_{10kHz} は 100[μs] である．図 5.83 では 98 番目のデータと 99 番目のデータの間に段差が見られる．$T_{\mathrm{10kHz}} > T_{100samp}$ であるため，マイコンは信号の 1 周期より（約 7 個分）少ないデータをサンプルしている．

さらに高速化を図るには，CH0〜CH4 に同じ信号を入れて，sequential & autoconversion により AD 変換を実行する設定がある．このとき，$\mathrm{T_{SAMP}} = \mathrm{TAD}$ は変わらないので，4CH の AD 変換に要する時間 $\mathrm{T_{AD4CH}}$ は

$$\mathrm{T_{AD4CH}} = (1 + 4 \times 12)\mathrm{TAD} = 49\mathrm{TAD} \tag{5.38}$$

となり，AD 変換のサンプリング周波数 f_{ADsamp} を

$$
\begin{aligned}
f_{\mathrm{ADsamp}} &= \frac{40.36[\mathrm{MHz}] \times 4\mathrm{CH}}{49 \times 3 \text{ クロック}} \\
&= 1.098[\mathrm{Msps}] \qquad (5.39)
\end{aligned}
$$

とできる．

5.7 DMA

5.7.1 2CH 同時サンプリング

DMA のメリット

PIC マイコンによる AC モータドライブにおいて電流制御系を構成するには，2 相のモータ電流を同時計測する必要がある．dsPIC33FJ128MC802 の AD 変換結果の格納用バッファは ADC1BUF0 のみである．前項の最高速サンプリングでは，各チャネルの AD 変換終了毎に割り込み処理関数が起動され，この関数が，言い換えれば，CPU が AD 変換結果を読み出していた．ADC1BUFx が 2 個あれば，2 相の電流の AD 変換終了後に一回の割り込み処理関数の起動で済む．ADC1BUF が 1 個しか無いのには理由がある．DMA (Direct Memory Access) は，CPU を介さないで，周辺モジュールと DMA RAM 間のデータ転送を行う機構である．2CH のアナログ入力を同時サンプリングする場合，AD 変換モジュールのサンプル & ホールドは，2CH 同時にスイッチをオフとして，アナログ電圧を保持する．そして，AD 変換器は，まず，CH0 の電圧をディジタル値に変換する．変換結果はバッファADC1BUF0 に格納される．DMA は変換終了後即座にバッファから DMA RAM に変換結果を転送する．その後，CH1 の電圧の変換，転送がなされる．2CH の変換結果の転送後に DMA が CPU に割り込みをかけ，CPU はこれら変換結果を DMA RAM から読み出して，例えば，これら変換結果がモータ電流であれば，電流指令値と比較して 3 相インバータへの指令電圧を生成する．なお，データシートの DATA MEMORY MAP によると，PIC33FJ128MC802 には，DMA RAM が 1kword （1024 × 16 ビット）ある．最大 1024 個の AD 変換値を，CPU を介さずに貯めることができ，その間，CPU は AD 変換に一切関わる必要が無い．

DMA の仕組み

図 5.84 は，dsPIC33FJ128MC802 のデータシートの TOP LEVEL SYSTEM ARCHITECTURE USING A DEDICATED TRANSACTION BUS の抜粋である．AD 変換モジュールは DMA Ready Peripheral（DMA 対応周辺モジュール）の 1 つである．DMA が無いマイコンでは，CPU Peripheral DS Bus と SRAM X-Bus のみを持ち，周辺モジュールと SRAM の間のデータ転送は CPU が担う．DMA があるマイコンでは，DMA DS Bus と DMA RAM が設けられている．周辺モジュールと DMA RAM 間のデータ転送は，CPU を介することなく，DMA コントローラが担う．このマイコンの DMA にはチャネ

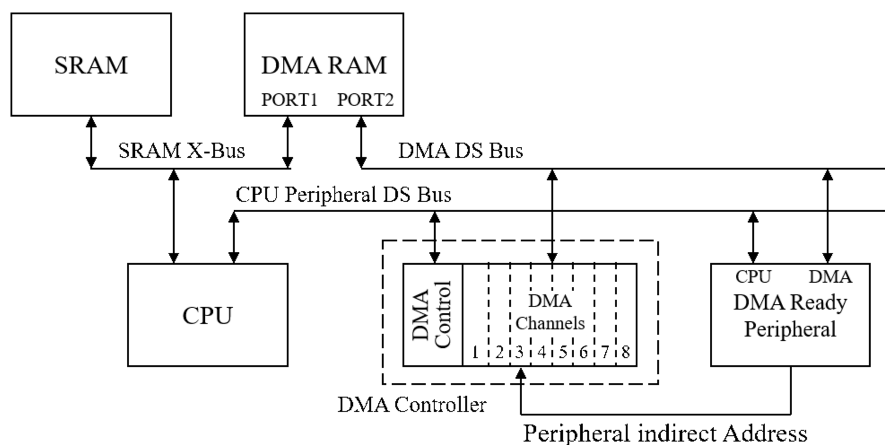


図 5.84: DMA コントローラとデータバスの構成

ルが 8 つあり，それぞれが 1 つずつ周辺モジュールを担当することが出来る．dsPIC33F Family Reference Manual の DMA Data Transfer Example によると，例えば，AD 変換終了時に AD 変換モジュールから担当チャネルに DMA データ転送要求が出され，当該チャネルは ADC1BUF0 からデータを読み出して，DMA RAM に転送する．DMA RAM の格納番地情報もこのチャネルが管理している．

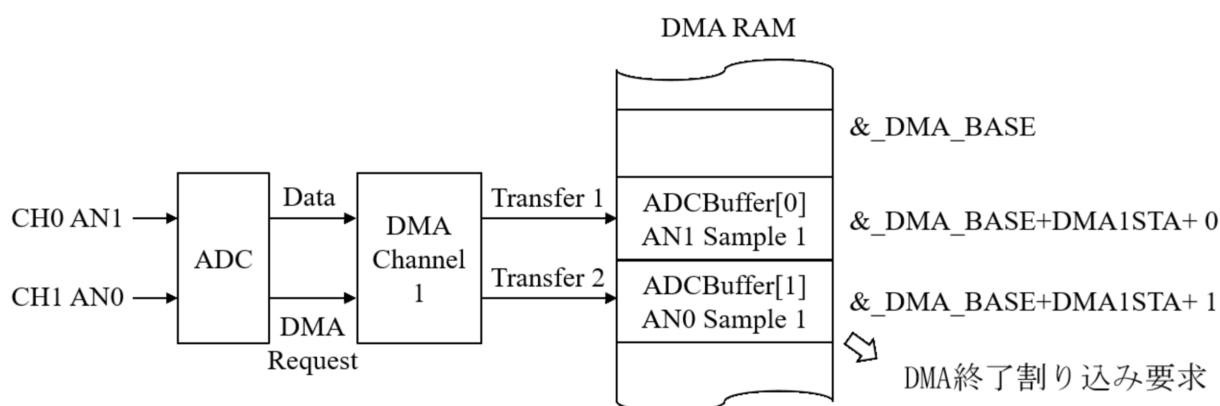


図 5.85: DMA1Interrupt.c における 2CH データの DMA RAM への転送・格納の様子

New Project として，ADC_DMAInterrupt を作ってください．そして，圧縮フォルダ内の ADC_DMAInterrupt フォルダから，新たに作られた MPLABXProjets¥ADC_DMAInterrupt.X フォルダ内に ADC_DMAInterrupt.c，ADC802.c，DMA802.c，SPI802.c，timer3.c のファ

イルと include ファイルをコピーしてください．そして，xxx.c のファイルを Source Files に付加し，include フォルダ内のヘッダファイルを Header Files に付加してください．

このプログラムが DMA RAM を利用する様子を図 5.85 に示す．2 チャネル CH0(AN1), CH1(AN0) の入力データは DMA Channel 1 を通して DMA RAM に続けて転送される．そして，2 つのデータ転送後にメイン関数に割り込みがかけられ，割り込み処理関数 DMA1Interrupt() 関数が起動される．割り込み処理関数は DMA RAM のデータを読み出して DMA に転送する．DMA RAM の転送先アドレスはベースとなる DMA_BASE アドレスに DMA1STA レジスタの値を加えて決められている．

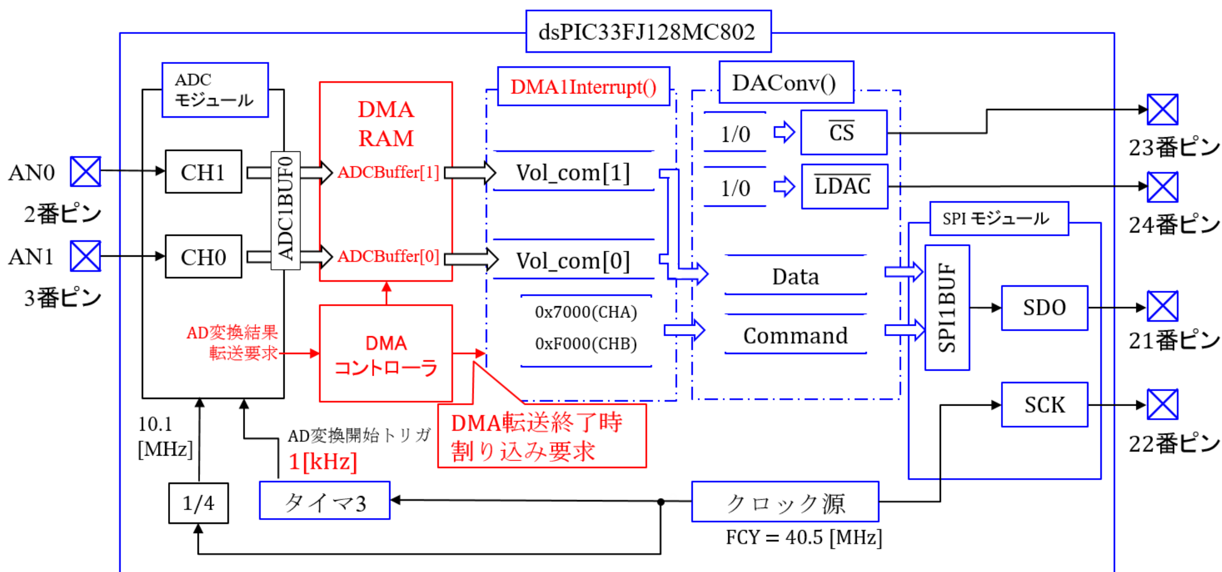


図 5.86: ADC_DMA1Interrupt プログラムのブロック図

DMA 利用 2CH 同時サンプリングプログラム

図 5.86 は ADC_DMAInterrupt.c のプログラムのブロック図である．ADC モジュールはタイマ 3 の 1 [kHz] 毎のトリガにより起動される．CH0 が 3 番ピンの AN1 に，CH1 が 2 番ピンの AN0 に割り当てられている．タイマ 3 のトリガにより，ADC モジュールは図 5.82 のサンプル & ホールド S/H0, S/H1 のスイッチをオフとしてサンプルを終了し，その後 CH0, CH1 の順に AD 変換を行い，バッファADC1BUFF0 に格納する．DMA コントローラは各 AD 変換終了毎にバッファ内のデータを，DMA Channel 1 を通して，ADCBuffer[0], [1] にそれぞれ格納する．そして，DMA コントローラは 2 つのデータを転送終了後に，メイン関数に割り込みをかけて，割り込み処理関数 DMA1Interrupt() 関数を起動する．割

り込み処理関数は ADCBuffer[0] を DMA RAM から読み出して Vol_com[0] に格納し，コマンド 0x7000 を付して SPI モジュールを通して DA 変換器に転送する．Vol_com0] の値は図 5.36 の DA 変換器の v_{OUTA} に出力される．引き続き，ADCBuffer[1] → Vol_com[1]，0xF000 + Vol_com[1] → v_{OUTB} と処理を行う．

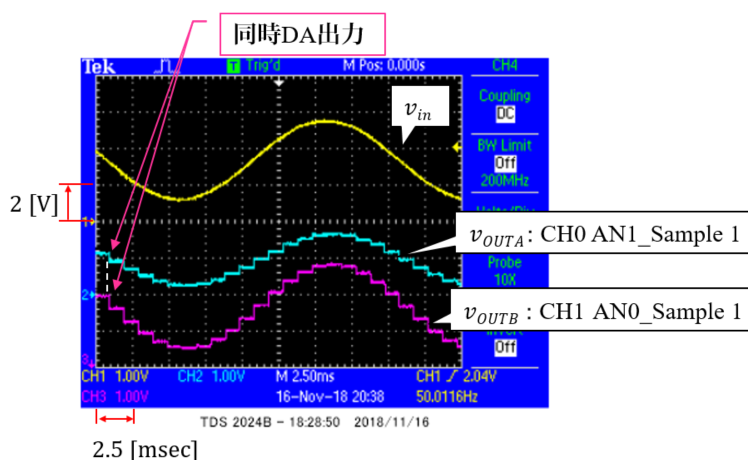


図 5.87: ADC_DMA1Interrupt プログラムの波形

図 5.87 は ADC_DMA1Interrupt.c のプログラムにより得られた波形例を示す．黄色の波形が AN0 への入力電圧 v_{in} である．周波数 $f_{in} = 50$ [Hz] である．AN1 には v_{in} を分圧して約半分の電圧を入力している．同時サンプリングを行い，同時 DA 出力（図 5.101 の DASimulOut() 関数) により，両出力電圧波形は位相が一致している．タイマ 3 のトリガ周期 $\mathrm{T_{tmr3}} = 1$ [ms] である．2CH の AD 変換の所要時間 $\mathrm{T_{AD2CH}}$ は (5.31) 式より 26TAD である．このプログラムでは，TAD = 4Tcy と設定しているので，

$$
\begin{aligned}
\mathrm{T_{AD2CH}} &= 26\mathrm{TAD} \\
&= 26 \times 4\mathrm{Tcy} \\
&= 26 \times \frac{4}{40.36[\mathrm{MHz}]} \\
&= 2.6[\mu\mathrm{s}]
\end{aligned}
\tag{5.40}
$$

である．DMA 転送に要するクロック数はデータシートにタイミングチャートが見当たらないため残念ながら不明であるが，経験上，連続 AD 変換の速度を遅らせることは無いようである．また，SPI モジュールによるデータ転送 (DAConv() 関数の実行) には，実測すると約 2.2 [μs] を要している．したがって，AD 変換から DA 変換器へのデータ転送までのトータル時間約 2.8 [μs] は，タイマ 3 のトリガ周期（1[ms]）よりはるかに短い．図

5.87 の v_{OUTA}, v_{OUTB} の波形には入力電圧の 1 周期 (= 20 [ms]) 間に 20 個の階段状波形が見られる．

```
//タイマ周期設定
const unsigned int       tsamp1 =  40000;
                         // Timer3 AD変換サンプリング周波数設定  40MHz/tasmp1 = 1kHz
signed int Vol_com[2];

// DMA設定
#define  NUMSAMP 2
unsigned int ADCBuffer[NUMSAMP] __attribute__((space(dma)));
                                        // DMAによりA/D変換結果を格納する配列の宣言

void init_SPI(void);
void init_ADC(void);
void init_DMA1(unsigned int a);
void DAConv(unsigned int a, unsigned int b);
void DASimulOut(void);

//DMA1割り込み処理関数
void __attribute__((interrupt, no_auto_psv)) _DMA1Interrupt(void)
{
    Vol_com[0] = ADCBuffer[0]<<2;
    DAConv(Vol_com[0], 0x7000);          // チャネルA選択
    Vol_com[1] = ADCBuffer[1]<<2;
    DAConv(Vol_com[1], 0xF000);          // チャネルB選択
    DASimulOut();

    _DMA1IF = 0;                         // 割り込みフラグクリア
}
```

図 5.88: DMA 利用 2CH 同時サンプリングプログラム (パラメータ設定，割り込み処理関数)

図 5.88，5.89 は，DMA 利用 2CH 同時サンプリングプログラムのパラメータ設定，割り込み処理関数，そしてメイン関数である．5.6.3 項の ADC_ADC1Interrupt.c からの変更点を朱書きで示す．新たに DMA 設定と DMA1 割り込み処理関数が追加されている．

#define NUMSAMP 2

により，DMA での 2 個のデータ転送が終了した時点で割り込みがかけられ，DMA1Interrupt() 関数が起動される設定である．

unsigned int ADCBuffer[NUMSAMP] __attribute__((space(dma)));

は，DMA による AD 変換結果の転送先の配列として ADCBuffer[] を宣言している．2 個のデータを格納するために，配列数は 2 である．space(dma) と指定することで，DMA RAM 内に配列が設定される．Microchip 社の提供する XC16 コンパイラ用に決められた

```
//メイン関数
int main(void)
{   //PLL設定
    _PLLPRE  = 0;                 // N1 = 2
    _PLLDIV  = 42;                //  M = 44
    _PLLPOST = 0;                 // N2 = 2
              // FOSC = FRC * M / (N1 * N2), FCY = Fosc / 2
              // FOSC = 7.37 MHz * 44 / (2 * 2) = 81MHz,  FCY = 40.5MHz

    _SWDTEN = 0;                  //ウォッチドッグタイマオフ

    TRISA = 0b00000011;           //  RA0.1  入力  RA1～RA4 出力ポート
    TRISB = 0b0000000000000000;   //  RB0～RB15 出力ポート

    //SPI用ポート設定
    _RP11R = 8;                   //RP11 → SCK (01000)
    _RP10R = 7;                   //RP10 → SDO  (00111

    init_SPI();
    init_timer3();
    init_ADC();
    init_DMA1(NUMSAMP);

    while(1){}                    //メインループ
}
```

図 5.89: DMA 利用 2CH 同時サンプリングプログラム (メイン関数)

表現法である．

DMA1 による割り込みにより DMA1Interrupt() 関数が起動される．この関数では，DMA による転送先 ADCBuffer[] から値を順次読み出して，DA 変換器に転送している．

DASimulOut();

は，図 5.47 の DA 変換器の Input Register A, B に 2 個のデータを転送後に A, B チャネル同時に DA 変換結果を出力させる関数である．

_DMA1IF = 0;

により，割り込み時に 1 とセットされていた割り込みフラグを 0 にリセットし，DMA1 による次の割り込みを受付け可とする．

図 5.89 のメイン関数での変更点は，

TRISA = 0b00000011;

により，RA0, 1(2, 3 番ピン) を入力ポートに設定することと，

ini_DMA1(NUMSAMP);

により，DMA1 の初期設定関数を実行していることのみである．

図 5.90 は AD コンバータの設定関数である．図 5.66 との相違点を朱書きで示してある．

```
// ADコンバータ初期設定
void init_ADC(void)
{
    unsigned int AD1CON1_set = ADC_MODULE_ON & ADC_IDLE_CONTINUE
                             & ADC_ADDMABM_ORDER & ADC_AD12B_10BIT
                             & ADC_FORMAT_INTG & ADC_CLK_TMR & ADC_SIMULTANEOUS
                             & ADC_AUTO_SAMPLING_ON & ADC_SAMP_ON;
    unsigned int AD1CON2_set = ADC_VREF_AVDD_AVSS & ADC_SCAN_OFF
                             & ADC_SELECT_CHAN_01 & ADC_DMA_ADD_INC_1
                             & ADC_ALT_BUF_OFF & ADC_ALT_INPUT_OFF;
    unsigned int AD1CON3_set = ADC_CONV_CLK_SYSTEM
                             & ADC_SAMPLE_TIME_2 & ADC_CONV_CLK_4Tcy;
    unsigned int AD1CON4_set = ADC_DMA_BUF_LOC_1;
    unsigned int AD1PCFGL_set = ENABLE_AN0_ANA & ENABLE_AN1_ANA;
    unsigned int AD1CSSL_set = 0x0; // 未使用
    unsigned int AD1CHS0_set = ADC_CH0_POS_SAMPLEA_AN1
                             & ADC_CH0_NEG_SAMPLEA_VREFN;
    unsigned int AD1CHS123_set = ADC_CH123_POS_SAMPLEA_0_1_2
                             & ADC_CH123_NEG_SAMPLEA_VREFN;

    OpenADC802(AD1CON1_set, AD1CON2_set, AD1CON3_set, AD1CON4_set, AD1PCFGL_set,
                            AD1CSSL_set);
    SetChanADC802_10BIT(AD1CHS0_set, AD1CHS123_set);
}
```

図 5.90: DMA 利用 2CH 同時サンプリングプログラム (AD コンバータ設定)

図 5.91 は AD1CON1 レジスタである．図 5.55 の再掲である．

ADC_ADDMABM_ORDER = 0xFFFF = 0b1111 1111 1111 1111

ADDMABM = 1 : DMA RAM アドレスを AD 変換順とする．

(例：図 5.85)

= 0 : DMA RAM アドレスを CH 毎に別々に設定する．

(例:図 5.106, 図 5.117)

ADDMABM = 1 (ADC_ADDMABM_ORDER) のとき，DMA1CON レジスタ (図5.95) の AMODE = 0b01 or 0b00 (Register Indirect with/without Post-Increment Mode) とする．

ADDMABM = 0 (ADC_ADDMABM_SCATTER) のときは，AMODE = 0b10 (Peripheral Indirect Addressing Mode) とする．また，同時に AD1CON4 レジスタ (図 5.93) の DMABL により各 ANx に対して DMA RAM のバッファサイズを指定する．

ADC_SIMULTANEOUS = 0xFFFF = 0b1111 1111 1111 11111

SIMSAM = 1 : CH0〜CH4 を同時にサンプル & ホールドする．

AD1CON1

15	14	13	12	11	10	9	8
ADON	-	ADSIDL	ADDMABM	-	AD12B	FORM	

7	6	5	4	3	2	1	0
SSRC			-	SIMSAM	ASAM	SAMP	DONE

図 5.91: AD1CON1 レジスタ

AD1CON2

15	14	13	12	11	10	9	8
VCFG			-	-	CSCNA	CHPS	

7	6	5	4	3	2	1	0
BUFS	-	SMPI				BUFM	ALTS

図 5.92: AD1CON2 レジスタ

= 0：CH0～CH4 を順にサンプル & ホールドする．

図 5.92 は AD1CON2 レジスタである．図 5.56 を再掲する．

ADC_DMA_ADD_INC_1 = 0xEFC3 = 0b1110 1111 1100 0011

SMPI = 0b0000：Channel Scan Mode でなければ 0 とする．

= 0bxxxx：Channel Scan Mode(5.7.6 項参照) において，スキャンするチャネル数と同じ値を指定する．

ADC_SELECT_CHAN_01 = 0xEDFF = 0b1110 1101 1111 1111

CHPS = 0b1x：CH0～3 を選択する．

= 0b01：CH0, 1 を選択する．

= 0b00：CH0 を選択する．

図 5.93 は AD1CON4 レジスタである．

ADC_DMA_BUF_LOC_1 = 0xFFF8 = 0b1111 1111 1111 1000

DMABL = 0b111：バッファ 128 ワード (16 ビット) を各 ANx に割り当てる．

= 0b110：バッファ 64 ワードを各 ANx に割り当てる．

AD1CON4

15	14	13	12	11	10	9	8
-	-	-	-	-	-	-	-

7	6	5	4	3	2	1	0
-	-	-	-	-	DMABL		

図 5.93: AD1CON4 レジスタ

= · · ·
= 0b001：バッファ 2 ワードを各 ANx に割り当てる.
= 0b000：バッファ 1 ワードを各 ANx に割り当てる.

ADDMABM = 0 (ADC_ADDMABM_SCATTER) のときは AMODE = 0b10 (DMA1_PERIPHERAL_INDIRECT) とし，この DMABL により各 ANx に割り当てるバッファサイズを指定する.

unsigned int AD1CHS0_set = ADC_CH0_POS_SAMPLEA_AN1
& ADC_CH0_NEG_SAMPLEA_VREFN;

により，CH0 の正入力に AN1 を割り当てている．そして，

unsigned int AD1CHS123_set = ADC_CH123_POS_SAMPLEA_0_1_2
& ADC_CH123_NEG_SAMPLEA_VREFN;

により，CH1, 2, 3 の各正入力に AN0, 1, 2 を割り当てている．AN1 が CH0 と CH2 で重複しているが，このプログラムでは CH0 と CH1 しか使わないので問題は無い.

なお，図 5.66 では ConfigIntADC802() 関数により AD 変換終了時の割り込み設定がなされていたが，このプログラムでは，DMA によるデータ転送終了時に割り込みを行うため，割り込み設定は図 5.94 の DMA1 初期設定関数にてなされる.

DMA1 初期設定関数は引数 NUMSAMP を持つ．これは，ADC_DMA1Interrupt.c 関数にてグローバル定数として宣言されている．また，

extern unsigned int ADCBuffer[]__attribute__((space(dma));

により，DMA RAM 内の配列 ADCBuffer[] が外部変数として宣言されている.

DMA1_MODULE_ON などの定数は DMA802.h のヘッダファイル内で定義されている．この定義と dsPIC33FJ128MC802 のデータシートを参照しながら，DMA の各レジスタの設定を以下に見ていく.

```
//DMA1初期設定関数
void init_DMA1(unsigned int NUMSAMP)
{
    extern unsigned int ADCBuffer[ ] __attribute__((space(dma)));

    unsigned int DMA1CON_set = DMA1_MODULE_ON  & DMA1_SIZE_WORD
                               & PERIPHERAL_TO_DMA1 & DMA1_INTERRUPT_BLOCK
                               & DMA1_NORMAL  & DMA1_REGISTER_POST_INCREMENT
                               & DMA1_CONTINUOUS;
    unsigned int DMA1REQ_set = 0b00001101;      // チャネル 1 をADC1 担当とする.
    unsigned int DMA1STA_set = __builtin_dmaoffset(ADCBuffer);
                                                // DMA RAM primary Start Addres register設定
    unsigned int DMA1PAD_set = (volatile unsigned int)&ADC1BUF0;
    unsigned int DMA1CNT_set = NUMSAMP -1;

    unsigned int DMA1IntConfig =  DMA1_INT_PRI_4 & DMA1_INT_ENABLE;

    OpenDMA802_1(DMA1CON_set, DMA1REQ_set, DMA1STA_set, DMA1PAD_set, DMA1CNT_set);
    ConfigIntDMA802_1(DMA1IntConfig);
}
```

図 5.94: DMA 利用 2CH 同時サンプリングプログラム (DMA1 初期設定)

図 5.95 は DMA1CON レジスタを示す.

DMA1_MODULE_ON = 0xFFFF = 0b1111 1111 1111 1111

CHEN = 1 : DMA1 チャネルを使用可とする.

= 0 : DMA1 チャネルを使用不可とする.

DMA1_SIZE_WORD = 0xBFFF = 0b1011 1111 1111 1111

SIZE = 1 : 転送するデータサイズをバイト (8 ビット) とする.

= 0 : 転送するデータサイズをワード (16 ビット) とする.

PERIPHERAL_TO_DMA1 = 0xDFFF = 0b1101 1111 1111 1111

DIR = 1 : DMA RAM から周辺モジュールへデータ転送を行う.

= 0 : 周辺モジュールから DMA RAM へデータ転送を行う.

DMA1_INTERRUPT_BLOCK = 0xEFFF = 0b1110 1111 1111 1111

HALF = 1 : NUMSAMP の半分のデータが転送されたときに割り込みを起動する.
ただし，NUMSAMP が奇数であれば NUMSAP+1 の半分とする.

= 0 : NUMSAMP のデータ全てが転送されたときに割り込みを起動する.

DMA1_NORMAL = 0xF7FF = 0b1111 0111 1111 1111

NULLW = 1 : DIR = 0 のとき，DMA RAM へデータ転送後に
周辺モジュールのレジスタを 0 にリセットする.

DMA1CON

15	14	13	12	11	10	9	8
CHEN	SIZE	DIR	HALF	NULLW	-	-	-

7	6	5	4	3	2	1	0
-	-	AMODE		-	-	MODE	

図 5.95: DMA1CON レジスタ

= 0：DMA RAM へデータ転送後に周辺モジュールのレジスタを 0 に リセットしない．

DMA1_REGISTER_POST_INCREMENT = 0xFF0F = 0b1111 1111 0000 1111

AMODE = 0b10：Peripheral Indirect Addressing Mode

= 0b01：Register Indirect without Post-Increment Mode

= 0b00：Register Indirect with Post-Increment Mode

Register Indirect with Post-Increment Mode は，図 5.85 の様に，先頭アドレスを DMA_BASE アドレス + DMA1STA レジスタの値とする．DMA による 1 つ目のデータ転送後に，アドレスを 1 つ増やして，２つ目のデータ転送先とする．

Peripheral Indirect Addressing Mode (AMODE = 0b10 = DMA1_PERIPHERAL_INDIRECT) は，図 5.106 のように CHx 毎に別々のアドレスを割り当てる．AD1CON1 レジスタの ADDMABM = 0 (= ADC_ADDMABM_SCATTER) とし，同時に AD1CON4 レジスタの DMABL = 0bxxx (= ADC_DMA_BUF_LOC_(2^{xxx}) により，各 ANx に対して DMA RAM のバッファサイズを指定する．

DMA1_CONTINUOUS = 0xFFF0 = 0b1111 1111 1111 0000

MODE = 11：単発，Ping-Pong モード起動

= 10：連続，Ping-Pong モード起動

= 01：単発，Ping-Pong モード不起動

= 00：連続，Ping-Pong モード不起動

連続モードは，DMA によるデータ転送を，DMA Request がある度に自動的に繰り返す．Ping-Pong モードは 5.7.5 項にて解説する．

図 5.96 は DMA1REQ レジスタである．

IRQSEL = 0b00001101

DMA1REQ

15	14	13	12	11	10	9	8
FORCE	-	-	-	-	-	-	-

7	6	5	4	3	2	1	0
-	IRQSEL						

図 5.96: DMA1REQ レジスタ

データシートの DMA CHANNEL TO PERIPHERAL ASSOCIATIONS によると，DMA1 に ADC1 の変換結果転送を担当させる．

DMA1STA

15	14	13	12	11	10	9	8
STA<15:8>							

7	6	5	4	3	2	1	0
STA<7:0>							

図 5.97: DMA1STA レジスタ

図 5.97 は DMA1STA レジスタである．このレジスタに DMA RAM 内に宣言された配列 ADCBuffer[] の先頭アドレスを書き込む．この値は，図 5.85 の様に，DMA によるデータ転送時に DMA RAM 内の ADCBuffer[] のアドレス指定に用いられる．

DMA1PAD

15	14	13	12	11	10	9	8
PAD<15:8>							

7	6	5	4	3	2	1	0
PAD<7:0>							

図 5.98: DMA1PAD レジスタ

図 5.98 は DMA1PAD レジスタである．&ADC1BUF0 により，AD 変換結果が格納されている ADC1BUF0 レジスタのアドレスを DMA1PAD レジスタに格納している．なお，volatile unsigned int の volatile は「揮発性の」という意味であり，このアドレスが指示

しているレジスタ(この場合はADC1BUF0)の値は，周辺モジュール(この場合はAD変換器)により書き換えられることがあることを宣言している．この宣言によりコンパイラは，ADC1BUF0の値を利用するプログラムに対して，ADC1BUF0の値が書き換えられないことを前提とした最適化は適用しない．プログラムによっては，書き換えられないことを前提とした最適化（この場合はおせっかい）により，プログラムが誤作動してしまうことがある．ただし，本項のプログラムでは

unsigned int DMA1PAD_set = (unsigned int)&ADC1BUF0

と，volatileを付けなくても支障は無い．また，データシートのDMA CHANNEL TO PERIPHERAL ASSOCIATIONSより，直接

unsigned int DMA1PAD_set = 0x0300

と設定しても支障なく動く．無償のXC16コンパイラを使った場合しか試していない．

DMA1CNT

15	14	13	12	11	10	9	8
-	-	-	-	-	-	CNT<9:8>	

7	6	5	4	3	2	1	0
CNT<7:0>							

図 5.99: DMA1CNTレジスタ

図5.99はDMA1CNTレジスタである．データ転送(Transfer)の回数を指定する．NUMSAMP回のデータ転送が終了した時点で，DMAコントローラは割り込み処理を起動する．

図5.100はDMA1設定関数とインタラプト設定関数である．DMA1設定関数では，図5.94のDMA1初期設定関数にて設定された値を各レジスタに格納する．DMA1インタラプト設定関数では，DMA1による割り込み処理の起動を可能にしている．

図5.101は，DA変換器において2CH同時出力設定を行うプログラムである．DASimulOut()関数は，図5.39のDAConv関数からDA変換器の出力トリガの命令を抜き出したものである．DA変換器に2CHのデータを転送後にDASimulOut()関数を起動することで，DA変換器から2CH(v_{OUTA}, v_{OUTB})を同時に出力させることが出来る．

```
// DMA1設定
void OpenDMA802_1(unsigned int config, unsigned int req, unsigned int sta, unsigned int pad,
                  unsigned int cnt)
{
    DMA1CON = config;
    DMA1REQ = req;
    DMA1STA = sta;
    DMA1PAD = pad;
    DMA1CNT = cnt;
}

// DMA インタラプト設定
void ConfigIntDMA802_1(unsigned int config)
{
    IFS0bits.DMA1IF = 0;                      /* Clearing the Interrupt Flag bit */
    IPC3bits.DMA1IP = config & 0x07;          /* Setting Priority */
    IEC0bits.DMA1IE = (config & 0x4000)>>14;  /* Setting the Interrupt enable bit */
}
```

図 5.100: DMA 利用 2CH 同時サンプリングプログラム (DMA1 設定，DMA1 インタラプト設定)

```
// SPI ポート設定
#define    SPI_CS     _LATB12          // D/Aコンバータ Select
#define    SPI_LDAC   _LATB13          // D/Aコンバータ Load

// DAコンバータ　各種変数モニター用
void DAConv(unsigned int Data, unsigned int Command)
{
    int delay;
    SPI_CS = 0;                    // CS Low
    SPI1BUF = Data | Command;      // Command + Data送信
    delay = 1;                     // 約 xx usecディレイ
    while(delay--);                // 送信終了待ち
    SPI_CS = 1;                    // CS High
}

// DAコンバータ同時出力トリガ
void DASimulOut(void)
{
    SPI_LDAC = 0;                  // DAコンバータ出力トリガ
    SPI_LDAC = 1;
}
```

図 5.101: DMA 利用 2CH 同時サンプリングプログラム (DA2CH 同時出力設定)

5.7.2 2CH 同時サンプリング＋ 4 サンプル転送後割り込み

前項では，AD 変換器が 2CH 同時サンプル & ホールドして，順次 AD 変換し，DMA が各 CH の AD 変換終了後に DMA RAM へ転送した．そして，2CH のデータ転送後に割り込み処理を起動した．本項では，2CH×2 サンプル = 4 サンプルのデータ転送後に割り込み処理を起動する DMA のプログラムについて解説する．

New Project として，ADC_DMAInterrupt_NUMSAMP4 を作ってください．そして，圧縮フォルダ内の ADC_DMAInterrupt_NUMSAMP4 フォルダから，新たに作られた MPLABX-Projets¥ADC_DMAInterrupt_NUMSAMP4.X フォルダ内に ADC_DMAInterrupt_NUMSAMP4.c, ADC802.c, DMA802.c, SPI802.c, timer3.c のファイルと include ファイルをコピーしてください．そして，xxx.c のファイルを Source Files に付加し，include フォルダ内のヘッダファイルを Header Files に付加してください．

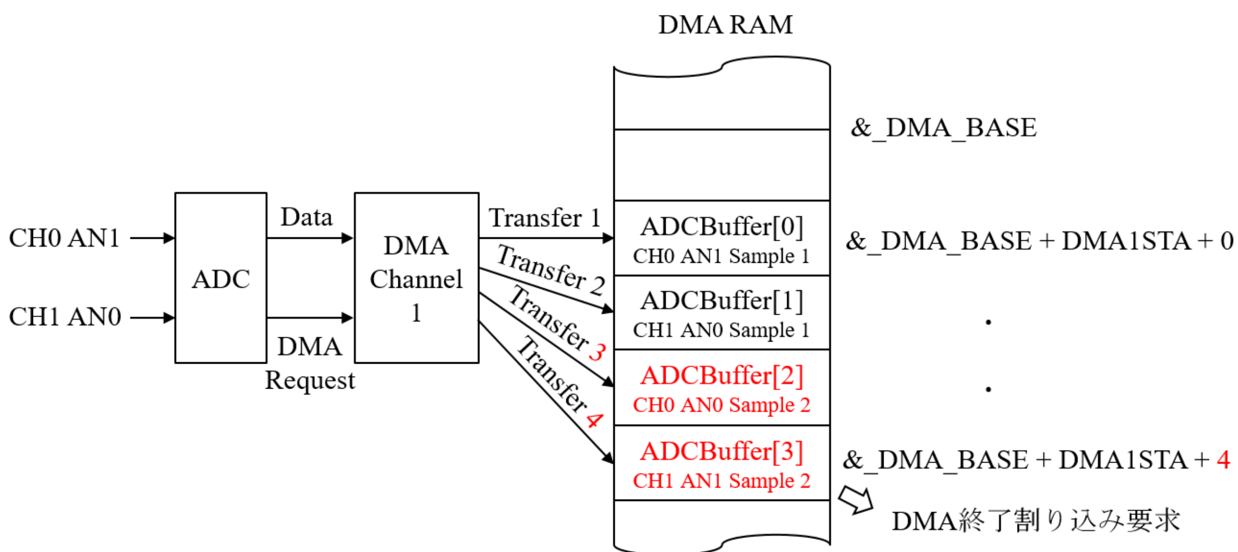


図 5.102: DMA によるデータ転送の様子 (2CH 同時サンプリング+4 サンプル転送後割り込み)

図 5.102 は 4 サンプル転送後に割り込み処理を起動する場合の DMA によるデータ転送と DMA RAM 内のバッファ割り当ての様子を示す．図 5.85 との相違点を朱書きで示す．ADCBuffer[] が要素数 4 に拡張されている．DMA により，CH0，CH1 の AD 変換結果の転送が 2 回繰り返された後に割り込み処理が起動されている．Sample 1，2 はそれぞれ 1, 2 回目のサンプル & ホールドを意味する．

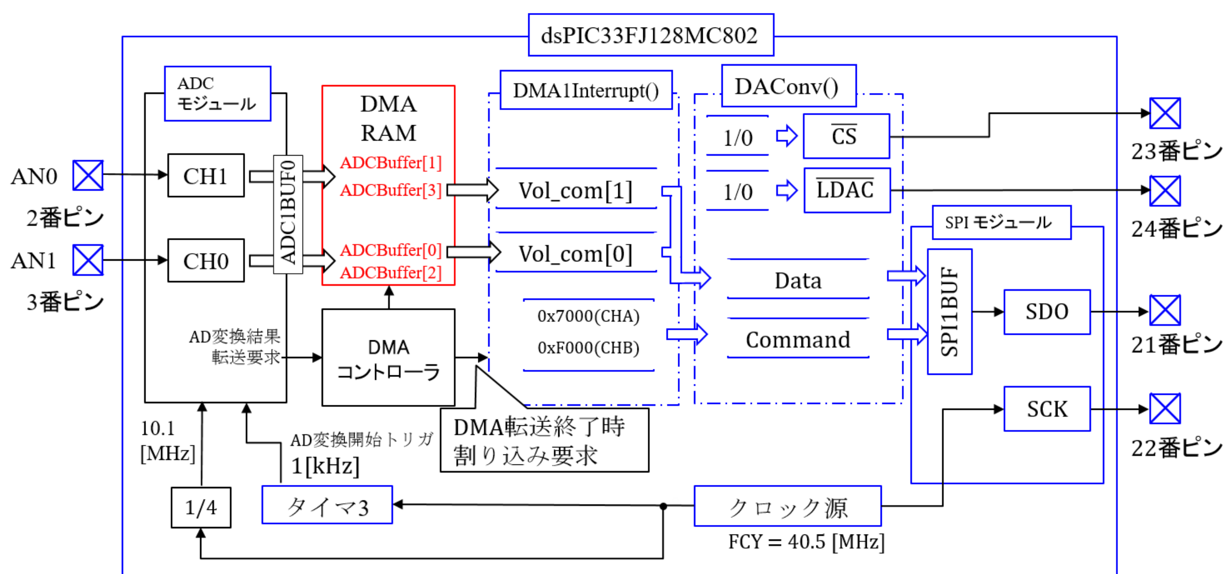


図 5.103: ADC_DMA1Interrupt_NUMSAMP4.c プログラムのブロック図 (2CH 同時サンプリング+4 サンプル転送後割り込み)

図 5.103 は，ADC_DMA1Interrupt_NUMSAMP4.c のプログラムのブロック図である．図 5.86 からの変更点を朱書きで示してある．DMA RAM には ADCBuffer[0]～ADCBuffer[3] に AD 変換結果が格納されている．DMA1Interrupt() 関数では，ADC1Buffer[0](CH0, AN1 の信号) を Vol_com[0] に代入し，DA 変換器の CH A(v_{OUTA}) に出力し，ADCBuffer[3](CH1, AN0 の信号) を Vol_com[1] に代入し，CH B(v_{OUTB}) に出力している．

図 5.104 は，ADC_DMA1Interrupt_NUMSAMP4.c のプログラムの抜粋である．主な変更箇所は

#define NUMSAMP 4

と，DMA1 割り込み処理関数 DMA1Interrupt() における

Vol_com[1] = ADCBuffer[2]<<2

のみである．NUMSAMP により，DMA RAM 内の配列 ADCBuffer[] のサイズを指定している．また，図 5.99 の DMA1CNT レジスタの CNT bits（割り込み処理起動までのデータ転送回数）も指定している．

図 5.105 は，実験波形例である．黄色の波形が AN0 への入力電圧 v_{in} である．周波数 f_{in} = 50 [Hz] である．AN1 には v_{in} を分圧して約半分の電圧を入力している. すなわち，

$$v_{\mathrm{AN1}} = \frac{v_{\mathrm{AN0}}}{2} \tag{5.41}$$

```
// DMA設定
#define  NUMSAMP 4
signed int ADCBuffer[NUMSAMP] __attribute__((space(dma)));
                                  // DMAによりA/D変換結果を格納する配列の宣言

//DMA1割り込み処理関数
void __attribute__((interrupt, no_auto_psv)) _DMA1Interrupt(void)
{
       Vol_com[0] = ADCBuffer[0]<<2;
       DAConv(Vol_com[0], 0x7000);              // チャネルA選択
       Vol_com[1] = ADCBuffer[3]<<2;
       DAConv(Vol_com[1], 0xF000);              // チャネルB選択
       DASimulOut();

      _DMA1IF = 0;                              // 割り込みフラグクリア
}
```

図 5.104: ADC_DMA1Interrupt_NUMSAMP4.c プログラム (2CH 同時サンプリング+4 サンプル転送後割り込み)

である．青の波形が DA 変換器の CH A の電圧 v_{OUTA} であり，赤の波形が CH B の電圧 v_{OUTB} である．それぞれ，AN1, AN0 の入力電圧を AD 変換→ DA 変換した結果である．青の波形（AN1) の振幅が赤の波形（AN0) の約半分となっている．

2CH 同時サンプリングで 4 サンプル後に波形が 1 回 DA 変換器から出力されるので，出力波形のサンプリング周波数 $f_{OUTsamp}$ は

$$\begin{aligned} f_{OUTsamp} &= \frac{1[\mathrm{kHz}]}{2} \\ &= 500[\mathrm{Hz}] \end{aligned} \tag{5.42}$$

である．入力信号の周波数 $f_{in} = 50$ [Hz] なので，入力信号の 1 周期に 10 回の波形更新 (10 個の階段) が見られる．

v_{OUTA}(ADCBuffer[0](AN1)) の方が v_{OUTB}（ADCBuffer[3](AN0)) よりも位相が遅れて見える．ADC モジュールは，タイマ 3 の 1 [kHz] 毎のトリガにより起動され，トリガ毎に 2CH の入力電圧を同時にサンプル & ホールドし，順次 AD 変換する．DMA は AD 変換結果を DMA RAM に転送し，4 回のデータ転送後に割り込み処理を起動する．したがって，ADCBuffer[0](AN1) 内の信号は，ADCBuffer[3](AN0) 内の信号より，(ADC モジュールの） 1 サンプル（$\frac{1}{1[\mathrm{kHz}]} = 1$ [ms]) 分早い時刻のサンプル値である．そして，DA 変換器では，ADCBuffer[0](AN1) の値を 1[ms] 遅らせて，ADCBuffer[3](AN0) の値と同時に表示している．結果として青の波形が 1[ms] に相当する位相だけ遅れて観測されている．

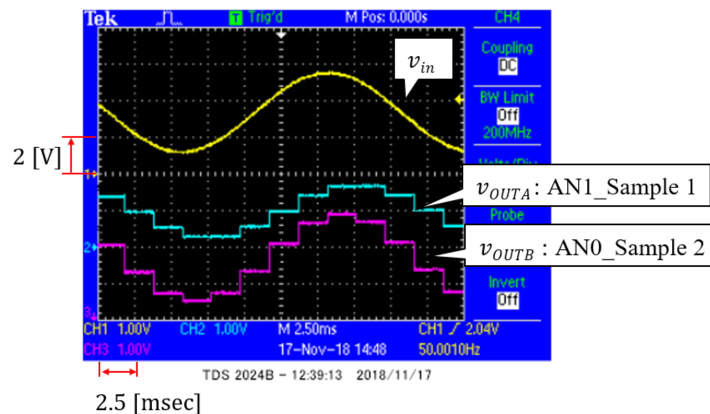


図 5.105: ADC_DMA1Interrupt_NUMSAMP4.c プログラムの波形 (2CH 同時サンプリング+4 サンプル転送後割り込み)

以上から，ADC モジュールは 1 [kHz] のサンプリング周波数で 2CH の AD 変換を実行し，DMA は 4 サンプルのデータ転送後に割り込み処理を起動していることが分かる．

5.7.3 PIA Mode (2CH同時サンプリング + 4サンプル転送後割り込み)

DMA RAMのバッファの使い方をPIA (Peripheral Indirect Addressing) Modeとする例を解説する．

New Projectとして，ADC_DMAInterrupt_PIAを作ってください．そして，圧縮フォルダ内のADC_DMAInterrupt_PIAフォルダから，新たに作られたMPLABXProjetsY=ADC_DMAInterrupt_PIA.Xフォルダ内にADC_DMAInterrupt_PIA.c, ADC802.c, DMA802.c, SPI802.c, timer3.cのファイルとincludeファイルをコピーしてください．そして，xxx.cのファイルをSource Filesに付加し，includeフォルダ内のヘッダファイルをHeader Filesに付加してください．

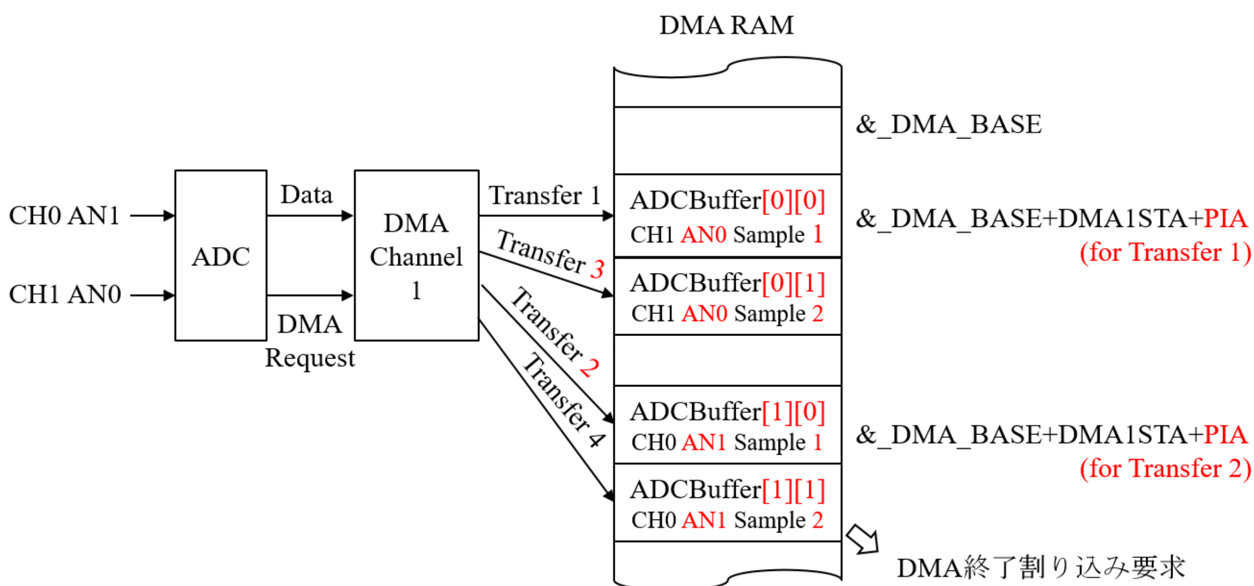


図 5.106: DMAによるデータ転送の様子 (2CH同時サンプリング + 4サンプル転送後割り込み + PIAモード)

図5.106はPIAモードによる，DMA RAMのバッファ利用の様子を示す．図5.102との違いを朱書きで示す．主な違いは，PIAモードでは，DMA RAM内でアナログ入力毎にデータがブロックに分けられて格納されることである．割り込み処理関数がDMA RAMから例えばANxのデータを取り出す際に，アドレスが連続しているため，プログラミングが容易となり，割り込み処理も効率的となる．図5.102のようにAD変換された順にデータがDMA RAMに格納されている場合では，アドレス指定に条件分岐などが必要となる．

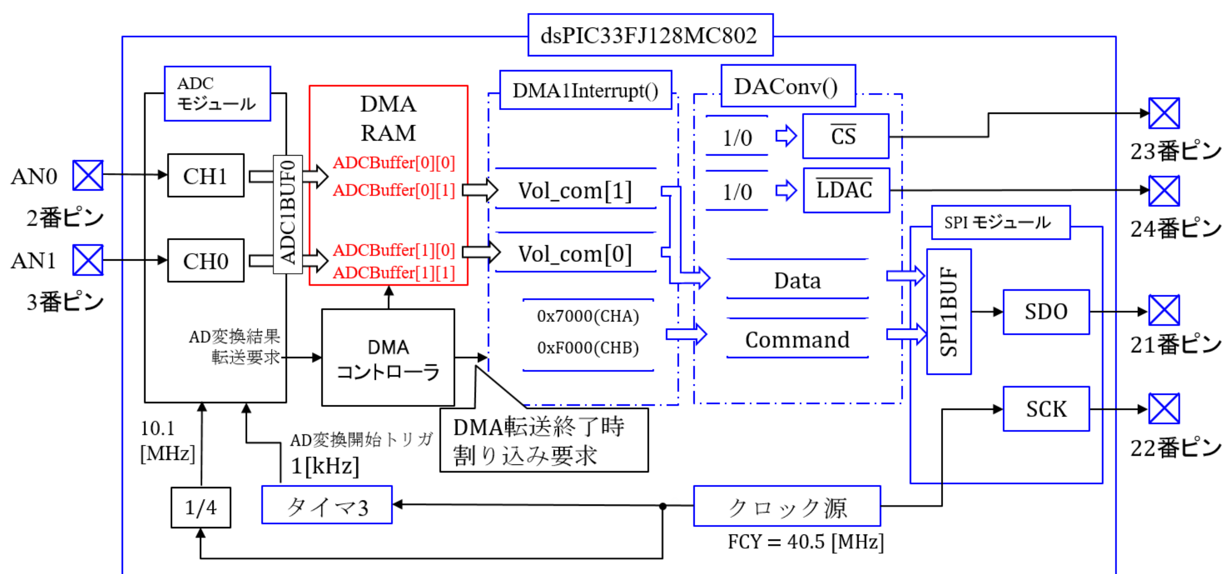


図 5.107: ADC_DMA1Interrupt_PIA.c プログラムのブロック図 (2CH 同時サンプリング + 4 サンプル転送後割り込み + PIA モード)

図 5.107 は，ADC_DMA1Interrupt_PIA.c のプログラムのブロック図である．図 5.103 からの変更点を朱書きで示してある．AN0，AN1 の 1 サンプル目の変換結果はそれぞれ ADCBuffer[0][0]，ADCBuffer[1][0] に，2 サンプル目の結果はそれぞれ ADCBuffer[0][1]，ADCBuffer[1][1] に格納される．

図 5.108 は ADC_DMA1Interrupt_PIA.c のプログラムの抜粋である．

unsigned int ADCBuffer[2][2]

として，2 次元の配列を宣言している．ADCBuffer[1][0](CH0，AN1 の 1 サンプル目の信号) を Vol_com[0] に代入し，DA 変換器の CH A(v_{OUTA}) に出力している．また，ADCBuffer[0][1](CH1，AN0 の 2 サンプル目の信号) を Vol_com[1] に代入し，CH B(v_{OUTB}) に出力している．

図 5.109 は，AD コンバータの初期設定関数である．図 5.90 からの変更点を朱書きで示してある．AD1CON1 レジスタにおいて

ADC_ADDMABM_SCATTER = 0xEFFF = 0b1110 1111 1111 1111

ADDMABM = 1：DMA RAM アドレスを AD 変換順とする．

(例：図 5.85)

```
// DMA設定
#define  NUMSAMP 4
unsigned int ADCBuffer[2][2] __attribute__((space(dma)));
                                        // DMAによりA/D変換結果を格納する配列の宣言

//DMA1割り込み処理関数
void __attribute__((interrupt, no_auto_psv)) _DMA1Interrupt(void)
{
    Vol_com[0] = ADCBuffer[1][0]<<2;
    DAConv(Vol_com[0], 0x7000);                 // チャネルA選択
    Vol_com[1] = ADCBuffer[0][1]<<2;
    DAConv(Vol_com[1], 0xF000);                 // チャネルB選択
    DASimulOut();

    _DMA1IF = 0;                                // 割り込みフラグクリア
}
```

図 5.108: ADC_DMA1Interrupt_PIA.c プログラム (2CH 同時サンプリング + 4 サンプル転送後割り込み + PIA)

= 0 : DMA RAM アドレスを CH 毎に別々に設定する．

と，PIA モードに合わせた設定をしている．

ADC_DMA_BUF_LOC_2 = 0xFFF9 = 0b1111 1111 1111 1001

DMABL = 0b111 : バッファ 128 ワード (16 ビット) を各 ANx に割り当てる．

= 0b110 : バッファ 64 ワードを各 ANx に割り当てる．

= · · ·

= 0b001 : バッファ 2 ワードを各 ANx に割り当てる．

= 0b000 : バッファ 1 ワードを各 ANx に割り当てる．

として，各アナログ入力 ANi に対する DMA RAM バッファを ADCBuffer[i][0], ADCBuffer[i][1] ($i = 0, 1$) の 2 ワードとしたことに合わせて設定している．

図 5.110 は，DMA の初期設定関数である．

DMA1_PERIPHERAL_INDIRECT = 0xFF2F = = 0b1111 1111 0010 1111

AMODE = 0b10 : Peripheral Indirect Addressing Mode

= 0b01 : Register Indirect without Post-Increment Mode

= 0b00 : Register Indirect with Post-Increment Mode

として，PIA モードを設定している．

```
// ADコンバータ初期設定
void init_ADC(void)
{
	unsigned int AD1CON1_set = ADC_MODULE_ON & ADC_IDLE_CONTINUE
				& ADC_ADDMABM_SCATTR & ADC_AD12B_10BIT
				& ADC_FORMAT_INTG & ADC_CLK_TMR & ADC_SIMULTANEOUS
				&ADC_AUTO_SAMPLING_ON & ADC_SAMP_ON;
	unsigned int AD1CON2_set = ADC_VREF_AVDD_AVSS & ADC_SCAN_OFF
				& ADC_SELECT_CHAN_01 &  ADC_DMA_ADD_INC_1
				& ADC_ALT_BUF_OFF & ADC_ALT_INPUT_OFF;
	unsigned int AD1CON3_set = ADC_CONV_CLK_SYSTEM
				& ADC_SAMPLE_TIME_2 & ADC_CONV_CLK_4Tcy;
	unsigned int AD1CON4_set = ADC_DMA_BUF_LOC_2;
	unsigned int AD1PCFGL_set = ENABLE_AN0_ANA & ENABLE_AN1_ANA;
	unsigned int AD1CSSL_set = 0x0; // 未使用
	unsigned int AD1CHS0_set = ADC_CH0_POS_SAMPLEA_AN1
				& ADC_CH0_NEG_SAMPLEA_VREFN;
	unsigned int AD1CHS123_set = ADC_CH123_POS_SAMPLEA_0_1_2
				& ADC_CH123_NEG_SAMPLEA_VREFN;

	OpenADC802(AD1CON1_set, AD1CON2_set, AD1CON3_set, AD1CON4_set,
				AD1PCFGL_set, AD1CSSL_set);
	SetChanADC802_10BIT(AD1CHS0_set, AD1CHS123_set);
}
```

図 5.109: ADC_DMA1Interrupt_PIA.c の AD コンバータ初期設定関数 (2CH 同時サンプリング + 4 サンプル転送後割り込み + PIA)

図 5.111 は実験波形である．図 5.105 と同様の波形が得られている．AN1 と AN0 の位相のずれ方も同様である．

```
void init_DMA1(unsigned int NUMSAMP)
{
      extern signed int ADCBufferA[] __attribute__((space(dma)));
                                        // DMAによりA/D変換結果を格納する配列の宣言

      unsigned int DMA1CON_set = DMA1_MODULE_ON  & DMA1_SIZE_WORD
                  & PERIPHERAL_TO_DMA1 & DMA1_INTERRUPT_BLOCK
                  & DMA1_NORMAL  & DMA1_PERIPHERAL_INDIRECT
                  & DMA1_CONTINUOUS;
      unsigned int DMA1REQ_set = 0b00001101;          // チャネル 1 をADC1 担当とする.
      unsigned int DMA1STA_set = __builtin_dmaoffset(ADCBufferA);
                                                  // DMA RAM primary Start Addres register設定
      unsigned int DMA1PAD_set = (volatile unsigned int)&ADC1BUF0;
      unsigned int DMA1CNT_set = NUMSAMP -1;
      unsigned int DMA1IntConfig =  DMA1_INT_PRI_4 & DMA1_INT_ENABLE;

      OpenDMA802_1(DMA1CON_set, DMA1REQ_set, DMA1STA_set,
                     DMA1PAD_set, DMA1CNT_set);
      ConfigIntDMA802_1(DMA1IntConfig);
}
```

図 5.110: ADC_DMA1Interrupt_PIA.c の DMA 初期設定関数 (2CH 同時サンプリング + 4 サンプル転送後割り込み + PIA)

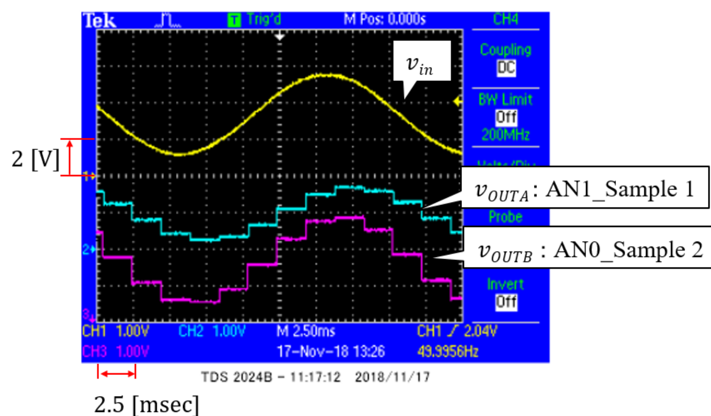


図 5.111: ADC_DMA1Interrupt_PIA.c プログラムの波形 (2CH 同時サンプリング + 4 サンプル転送後割り込み + PIA)

5.7.4 転送データ半分での割り込み（BUFM ＝ 1 の試み）

DMA において，AD 変換結果が DMA RAM バッファの半分を満たした時点と全部を満たした時点の両方で割り込み処理を起動する設定 (BUFM = 1) を試してみる．

New Project として，ADC_DMA_BUFM を作ってください．そして，圧縮フォルダ内の ADC_DMA_BUFM フォルダから，新たに作られた MPLABXProjets¥ADC_DMA_BUFM.X フォルダ内に ADC_DMA_BUFM.c, ADC802.c, DMA802.c, SPI802.c, timer3.c のファイルと include ファイルをコピーしてください．そして，xxx.c のファイルを Source Files に付加し，include フォルダ内のヘッダファイルを Header Files に付加してください．

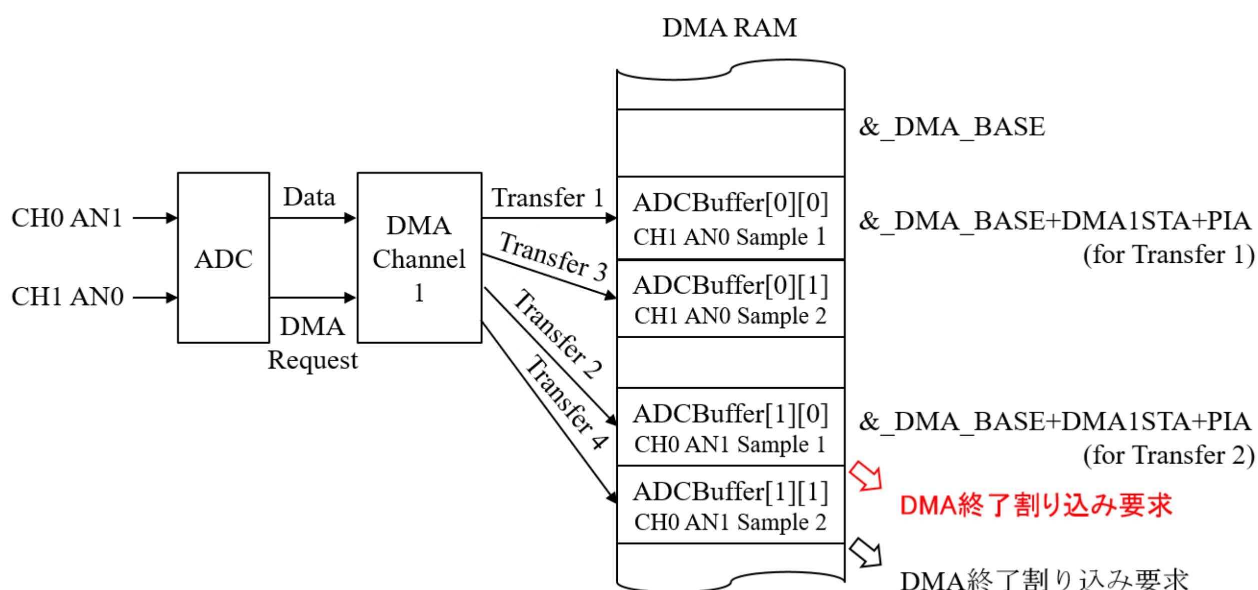


図 5.112: DMA によるデータ転送の様子 (2CH 同時サンプリング ＋ 2 サンプル転送後割り込み ＋ PIA モード ＋ BUFM =1)

図 5.112 は PIA モードにおいて，AD1CON2 レジスタの BUFM = 1 とした場合のデータ転送の様子を示す．図 5.106 との違いを朱書きで示す．違いは，DMA RAM へのデータ転送が半分まで済んだ時点でも割り込み処理が起動されることである．サンプルサイズが大きい場合には，半分までのサンプルが済んだ時点でサンプル値を読み出して処理をした方が，処理時間の制約などに対処できる使い方があると推定される．

図 5.113 は，ADC_DMA1Interrupt_PIA.c のプログラムのブロック図である．図 5.107 からの変更点を朱書きで示してある．DMA RAM へのデータ転送が半分まで済んだ時点でも割り込み処理が起動される．

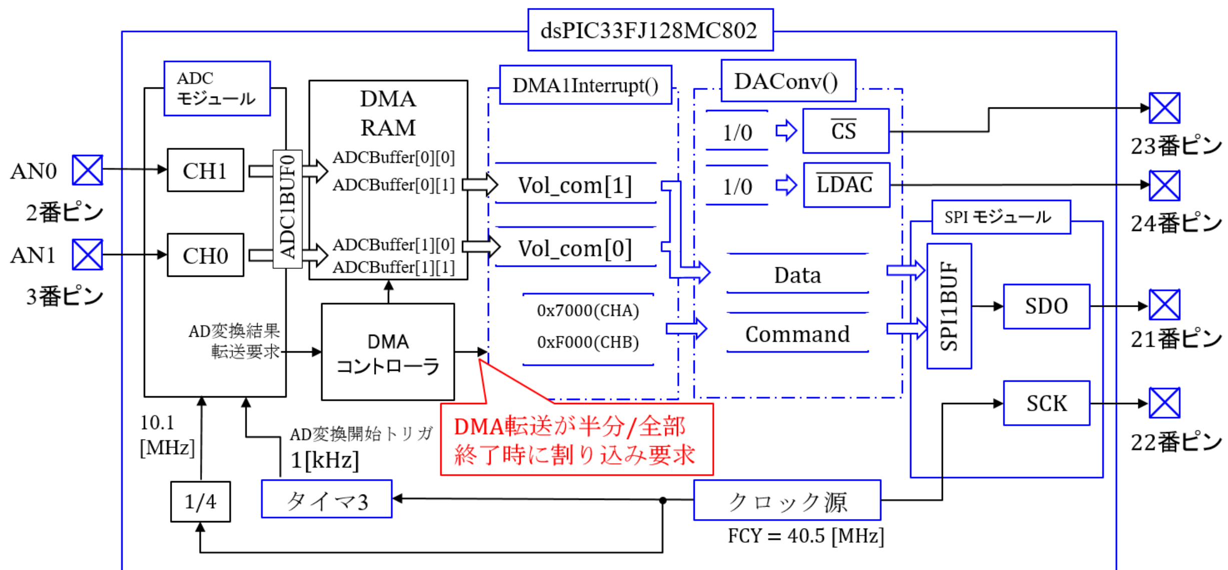


図 5.113: ADC_DMA_BUFM.c プログラムのブロック図 (2CH 同時サンプリング + 2 サンプル転送後割り込み + PIA モード + BUFM = 1)

.

図 5.114 は ADC_DMA_BUFM.c のプログラムの抜粋である．図 5.108 からの変更点は

```
#define NUMSAMP 2
```

のみである．データを2つ DMA RAM に転送した時点で割り込み処理を起動する設定である．

図 5.115 は，AD コンバータの初期設定関数である．図 5.109 からの変更点を朱書きで示してある．AD1CON2 レジスタにおいて

ADC_ALT_BUF_ON = 0xEFFF = 0b1110 1111 1111 1111

BUFM = 1 : DMA RAM バッファの半分および全部が満たされた両時点で割り込み

= 0 : 全部が満たされた時点のみで割り込み.

として，AD 変換結果が DMA RAM バッファの半分を満たした時点と全部を満たした時点の両方で割り込み処理を起動する．

```
// DMA設定
#define  NUMSAMP 2
unsigned int ADCBuffer[2][2] __attribute__((space(dma)));
                                        // DMAによりA/D変換結果を格納する配列の宣言

//DMA1割り込み処理関数
void __attribute__((interrupt, no_auto_psv)) _DMA1Interrupt(void)
{
        Vol_com[0] = ADCBuffer[1][0]<<2;
        DAConv(Vol_com[0], 0x7000);                  // チャネルA選択
        Vol_com[1] = ADCBuffer[0][1]<<2;
        DAConv(Vol_com[1], 0xF000);                  // チャネルB選択
        DASimulOut();

        _DMA1IF = 0;                                 // 割り込みフラグクリア
}
```

図 5.114: ADC_DMA_BUFM.c プログラム (2CH 同時サンプリング + 2 サンプル転送後割り込み + PIA + BUFM = 1)

```
// ADコンバータ初期設定
void init_ADC(void)
{
        unsigned int AD1CON1_set = ADC_MODULE_ON & ADC_IDLE_CONTINUE
                                & ADC_ADDMABM_SCATTR & ADC_AD12B_10BIT
                                & ADC_FORMAT_INTG & ADC_CLK_TMR & ADC_SIMULTANEOUS
                                &ADC_AUTO_SAMPLING_ON & ADC_SAMP_ON;
        unsigned int AD1CON2_set = ADC_VREF_AVDD_AVSS & ADC_SCAN_OFF
                                & ADC_SELECT_CHAN_01 &  ADC_DMA_ADD_INC_1
                                & ADC_ALT_BUF_ON & ADC_ALT_INPUT_OFF;
        unsigned int AD1CON3_set = ADC_CONV_CLK_SYSTEM
                                & ADC_SAMPLE_TIME_2 & ADC_CONV_CLK_4Tcy;
        unsigned int AD1CON4_set = ADC_DMA_BUF_LOC_2;
        unsigned int AD1PCFGL_set = ENABLE_AN0_ANA & ENABLE_AN1_ANA;
        unsigned int AD1CSSL_set = 0x0; // 未使用
        unsigned int AD1CHS0_set = ADC_CH0_POS_SAMPLEA_AN1
                                & ADC_CH0_NEG_SAMPLEA_VREFN;
        unsigned int AD1CHS123_set = ADC_CH123_POS_SAMPLEA_0_1_2
                                & ADC_CH123_NEG_SAMPLEA_VREFN;

        OpenADC802(AD1CON1_set, AD1CON2_set, AD1CON3_set, AD1CON4_set,
                                AD1PCFGL_set, AD1CSSL_set);
        SetChanADC802_10BIT(AD1CHS0_set, AD1CHS123_set);
}
```

図 5.115: ADC_DMA_BUFM.c の AD コンバータ初期設定関数 (2CH 同時サンプリング + 2 サンプル転送後割り込み + PIA + BUFM = 1)

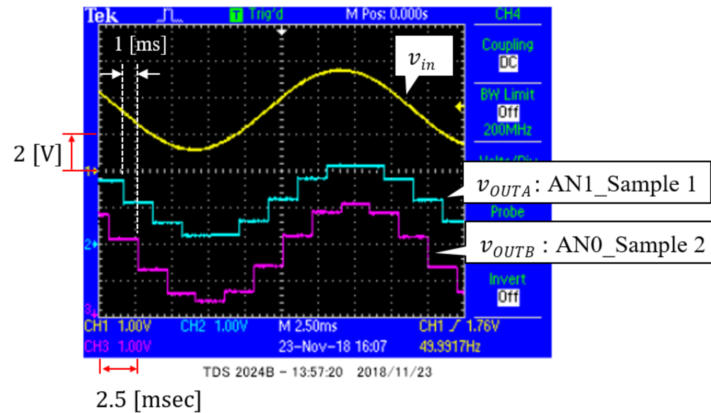


図 5.116: ADC_DMA_BUFM.c プログラムの波形 (2CH 同時サンプリング + 2 サンプル転送後割り込み + PIA + BUFM = 1)

図 5.116 は実験波形である．図 5.111 の実験波形との違いは，AN0 と AN1 の位相にずれが無く，AN0 と AN1 の段差のある時点が 1 [ms] ずれている点である．このことから，ある時点における AN0, AN1 の変換結果が ADCBuffer[0][0], ADCBuffer[1][0] に格納され，次の時点（1 [ms] 後）における AN0, AN1 の変換結果が ADCBuffer[0][1], ADCBuffer[1][1] に格納されていることを確認できる．割り込み処理関数は 1 [ms] 毎に起動され，そのつど，ADCBuffer[1][0], ADCBuffer[0][1] の値を DA 変換器に転送するが，それぞれのバッファの値は 2[ms] に 1 回しか更新されないため，段差のある時点が 1 [ms] ずれて観測されている．

5.7.5 Ping-Pong モード

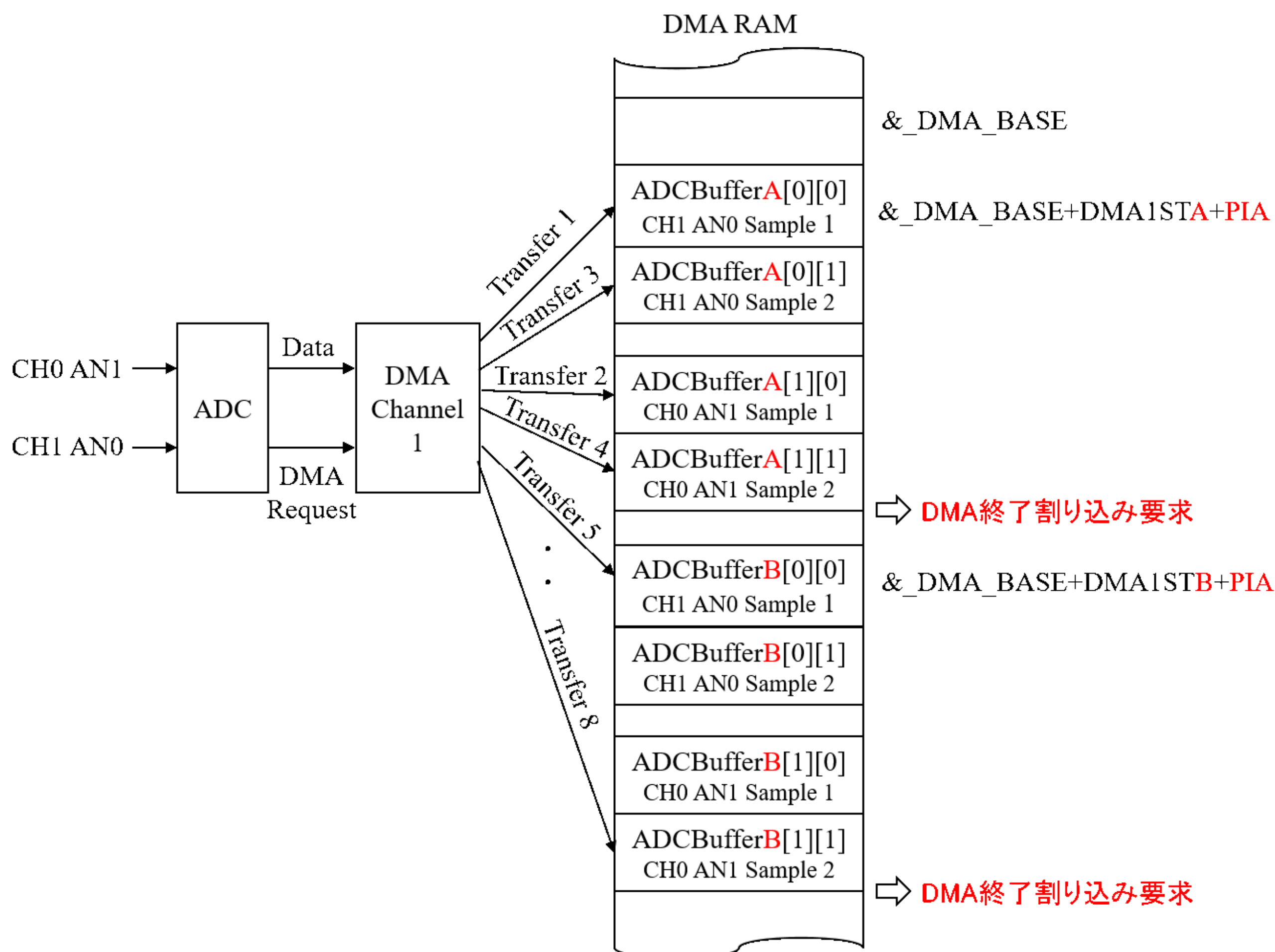


図 5.117: DMA によるデータ転送の様子 (2CH 同時サンプリング + 4 サンプル転送後割り込み + Ping-Pong モード)

DMA には Ping-Pong と呼ばれるモードがある．DMA RAM に A, B の 2 つのブロックを設ける．DMA が A ブロックにデータ転送している間に，CPU が B ブロック内のデータを読み出して処理を行う．そして，A ブロックへのデータ転送が終了したら，DMA は引き続き B ブロックへのデータ転送を行う．その間，CPU は A ブロックのデータを読み出して処理を行う．Ping-Pong のように，2 つのブロック間を行ったり来たりしながら転送/処理を進める．

New Project として，ADC_DMA_Ping_Pong を作ってください．そして，圧縮フォルダ内の ADC_DMA_Ping_Pong フォルダから，新たに作られた MPLABXProjets¥ADC_DMA_

Ping_Pong.X フォルダ内に ADC_DMA_Ping_Pong.c, ADC802.c, DMA802.c, SPI802.c, timer3.c のファイルと include ファイルをコピーしてください．そして，xxx.c のファイルを Source Files に付加し，include フォルダ内のヘッダファイルを Header Files に付加してください．

図 5.117 は図 5.106 の 2CH 同時サンプリング ＋ 4 サンプル転送後割り込みの設定を Ping-Pong モードに切替えた場合のデータ転送の様子を示す．図 5.106 との違いを朱書きで示す．主な違いは，DMA RAM のバッファが A, B の 2 ブロック設けられていることである．今，一定期間内に n 個のサンプル値（AD 変換結果）を転送・処理することを考えよう．Ping-Pong モードでは，データの転送先を 2 つ用意しておいて，DMA はまず，n 個のサンプル値を A ブロックに転送する．n 個の転送が済んだら，次の期間の n 個のサンプル値は B ブロックに転送する．その次の n 個は再び A ブロックへ ⋯ とブロックを交互に切替えて n 個ずつの転送を繰り返す．CPU は DMA がデータ転送中でないブロックからデータを読み出して処理を行う．一方，5.7.4 項の BUFM=1 のモードは $\frac{n}{2}$ 個の転送終了毎に割り込み処理を起動する．

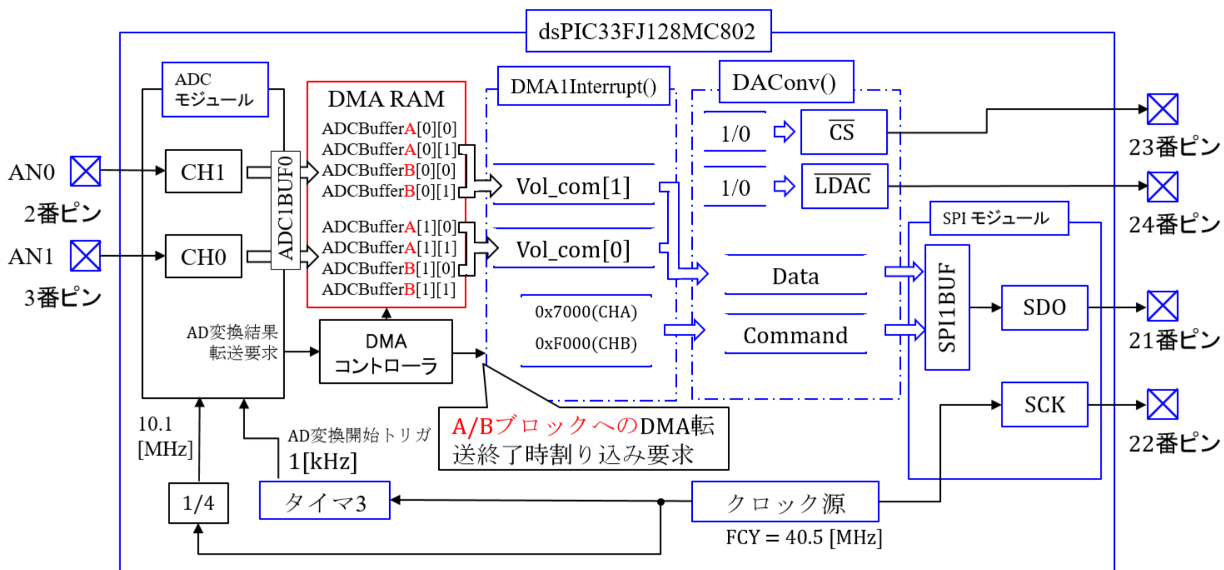


図 5.118: ADC_DMA_Ping_Pong.c プログラムのブロック図 (2CH 同時サンプリング ＋ 4 サンプル転送後割り込み ＋ Ping-Pong モード)

図 5.118 は，ADC_DMA_Ping_Pong.c のプログラムのブロック図である．図 5.107 からの変更点を朱書きで示してある．

図 5.119 は ADC_DMA_Ping_Pong.c のプログラムの抜粋である．図 5.108 からの変更

```
// DMA定数，バッファ，変数設定
#define  NUMSAMP 4
unsigned int ADCBufferA[2][2]__attribute__((space(dma), aligned(8)));
unsigned int ADCBufferB[2][2]__attribute__((space(dma), aligned(8)));
                                              // DMAによりA/D変換結果を格納する配列の宣言
unsigned int DmaBuffer = 0;

//DMA1割り込み処理関数
void __attribute__((interrupt, no_auto_psv)) _DMA1Interrupt(void)
{
   unsigned int vAN0out, vAN1out;
   if(DmaBuffer == 0)
   {
             vAN0out = ADCBufferA[0][1]<<2;
             vAN1out = ADCBufferA[1][0]<<2;
   }else {
             vAN0out = ADCBufferB[0][1]<<2;
             vAN1out = ADCBufferB[1][0]<<2;
   }
   DmaBuffer ^= 1;

   DAConv(vAN1out, 0x7000);               // チャネルA選択
   DAConv(vAN0out, 0xF000);               // チャネルB選択
   DASimulOut();

   _DMA1IF = 0;                           // 割り込みフラグクリア
}
```

図 5.119: ADC_DMA_Ping_Pong.c プログラム (2CH 同時サンプリング + 4 サンプル転送後割り込み + Ping-Pong モード)

点を朱書きで示してある．違いは，ADCBuffer に A, B が設けられていることと，

unsigned int DmaBuffer = 0;

により，フラグ用の変数を宣言して，DMA1Interrupt() 関数が読み出す DMA RAM のブロックを切替えていることである．

図 5.120 は，DMA の初期設定関数である．図 5.110 からの変更点を朱書きで示してある．

DMA1_CONTINUOUS_PING_PONG = 0xEFF2 = 0b1111 1111 1111 0010

MODE = 11 : 単発，Ping-Pong 起動

= 10 : 連続，Ping-Pong モード起動

= 01 : 単発，Ping-Pong モード不起動

= 00 : 連続，Ping-Pong モード不起動

```
void init_DMA1(unsigned int NUMSAMP)
{
   extern signed int ADCBufferA[ ] __attribute__((space(dma)));
   extern signed int ADCBufferB[ ] __attribute__((space(dma)));

   // ADC1 to DMA1 Config
   unsigned int DMA1CON_set = DMA1_MODULE_ON  & DMA1_SIZE_WORD
                              & PERIPHERAL_TO_DMA1 & DMA1_INTERRUPT_BLOCK
                              & DMA1_NORMAL  & DMA1_PERIPHERAL_INDIRECT
                              & DMA1_CONTINUOUS_PING_PONG;
    unsigned int DMA1REQ_set = 0b00001101;      // チャネル 1 をADC1 担当とする.
    unsigned int DMA1STA_set = __builtin_dmaoffset(ADCBufferA);
    unsigned int DMA1STB_set = __builtin_dmaoffset(ADCBufferB);

                                                    // DMA RAM primary Start Addres register設定
    unsigned int DMA1PAD_set = (volatile unsigned int)&ADC1BUF0;
    unsigned int DMA1CNT_set = NUMSAMP -1;

    OpenDMA802_1(DMA1CON_set, DMA1REQ_set, DMA1STA_set, DMA1STB_set,
                               DMA1PAD_set, DMA1CNT_set);

   unsigned int DMA1IntConfig =  DMA1_INT_PRI_4 & DMA1_INT_ENABLE;

   ConfigIntDMA802_1(DMA1IntConfig);
}
```

図 5.120: ADC_DMA_Ping_Pong.c の DMA 初期設定関数 (2CH 同時サンプリング + 4 サンプル転送後割り込み + Ping-Pong)

と，Ping-Pong モードを指定している．

図 5.121 は DMA 設定関数である．変更点を朱書きで示してある．この関数の引数の数が変わったので，dma.h のヘッダファイル内でのこの関数の宣言文にも変更を加えてある．

図 5.122 は実験波形である．図 5.111 と同様の波形が観測されている．

```
// DMA設定
void OpenDMA802_1(unsigned int config, unsigned int req, unsigned int sta, unsigned int stb,
                  unsigned int pad, unsigned int cnt)
{
    /* configures the DMA0 CONTROL register */
    DMA1CON = config;
    DMA1REQ = req;
    DMA1STA = sta;
    DMA1STB = stb;
    DMA1PAD = pad;
    DMA1CNT = cnt;
}


(dma.h)
/* DMA Function Prototypes */
void OpenDMA802_1(unsigned int a, unsigned int b, unsigned int c, unsigned int d, unsigned int e,
                  unsigned int f);
```

図 5.121: ADC_DMA_Ping_Pong.c の DMA 設定関数 (2CH 同時サンプリング + 4 サンプル転送後割り込み + Ping-Pong)

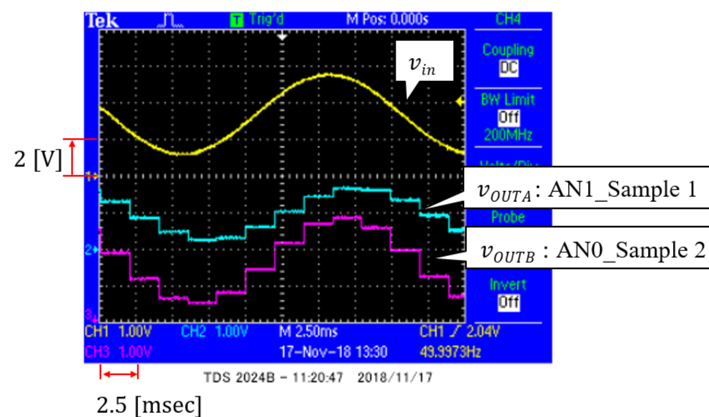


図 5.122: ADC_DMA_Ping_Pong.c プログラムの波形 (2CH 同時サンプリング + 4 サンプル転送後割り込み + Ping-Pong)

5.7.6 Scan モード

New Project として，ADC_DMA1Interrupt_Scan_4CH を作ってください．そして，圧縮フォルダ内の ADC_DMA1Interrupt_Scan_4CH フォルダから，新たに作られた MPLABX-Projets¥ADC_DMAInterrupt_Scan_4CH.X フォルダ内に ADC_DMA1Interrupt_Scan_4CH.c, ADC802.c, DMA802.c, SPI802.c, timer3.c のファイルと include ファイルをコピーしてください．そして，xxx.c のファイルを Source Files に付加し，include フォルダ内のヘッダファイルを Header Files に付加してください．

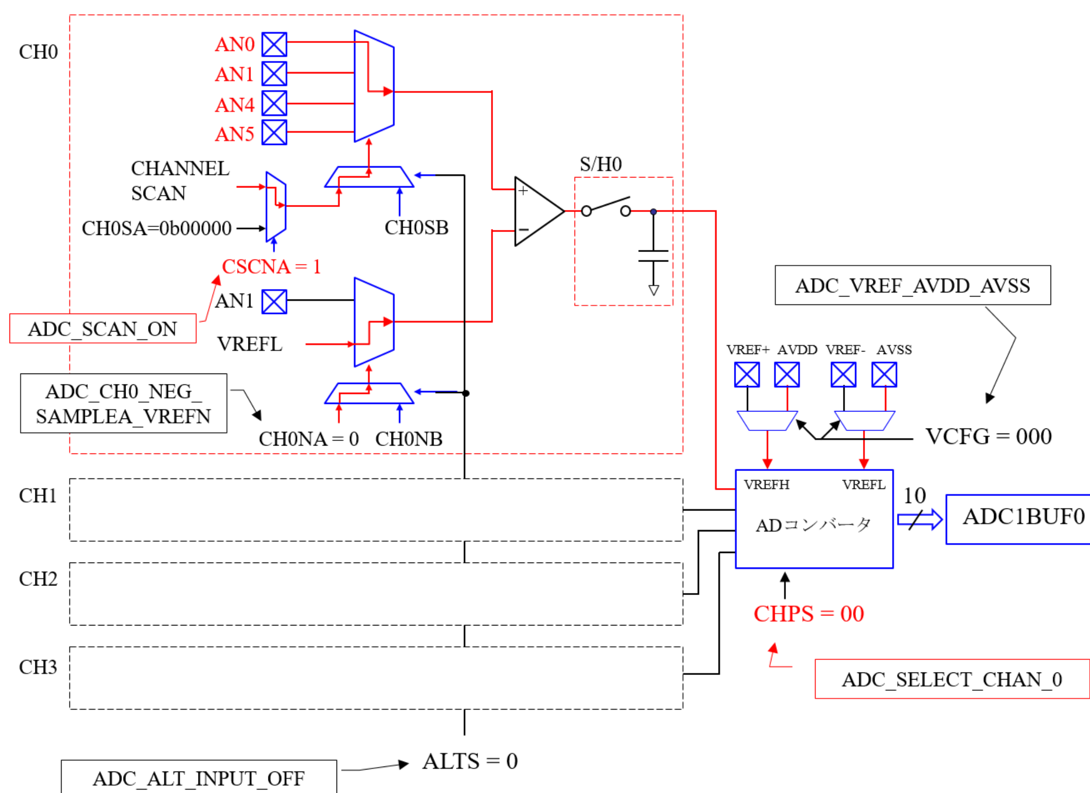


図 5.123: ADC_DMA1Interrupt_Scan_4CH.c の AD 変換モジュールの設定結果

DMA には Scan モードがある．図 5.3 より，dsPIC33FJ128MC802 には AD 変換用のアナログ入力入力 AN0〜AN5 がある．図 5.123 は，この内，AN0, AN1, AN4, AN5 の 4 つのアナログ入力を Scan する場合の AD 変換モジュールの設定の様子を示す．また，図 5.124 は，AD 変換モジュールの初期設定関数である．関連箇所を朱書きで示してある．図 5.125 に AD1CON2 レジスタを再掲する．Scan 機能は CH0 のみに備えられている．そこで，AD1CON2 レジスタにて

```
// ADコンバータ初期設定
void init_ADC(void)
{
        unsigned int AD1CON1_set = ADC_MODULE_ON & ADC_IDLE_CONTINUE
                        & ADC_ADDMABM_ORDER & ADC_AD12B_10BIT
                        & ADC_FORMAT_INTG & ADC_CLK_TMR
                        & ADC_AUTO_SAMPLING_ON & ADC_MULTIPLE & ADC_SAMP_ON ;
        unsigned int AD1CON2_set = ADC_VREF_AVDD_AVSS & ADC_SCAN_ON
                        & ADC_SELECT_CHAN_0 &  ADC_DMA_ADD_INC_4
                        & ADC_ALT_BUF_OFF & ADC_ALT_INPUT_OFF;
        unsigned int AD1CON3_set = ADC_CONV_CLK_SYSTEM
                        & ADC_SAMPLE_TIME_2 & ADC_CONV_CLK_4Tcy;
        unsigned int AD1CON4_set = ADC_DMA_BUF_LOC_1;
        unsigned int AD1PCFGL_set =  ENABLE_AN0_ANA & ENABLE_AN1_ANA
                        & ENABLE_AN4_ANA & ENABLE_AN5_ANA;
        unsigned int AD1CSSL_set = 0b0000000000110011;
        unsigned int AD1CHS0_set = ADC_CH0_NEG_SAMPLEA_VREFN;
        unsigned int AD1CHS123 =  0xFFFF;

        OpenADC802(AD1CON1_set, AD1CON2_set, AD1CON3_set, AD1CON4_set,
                        AD1PCFGL_set, AD1CSSL_set);
        SetChanADC802_10BIT(AD1CHS0_set, AD1CHS123_set);
}
```

図 5.124: ADC_DMA1Interrupt_Scan_4CH.c の ADC 初期設定関数 (Scan モード)

ADC_SELECT_CHAN_0 = 0xECFF = 0b1110 1100 1111 1111

CHPS = 0b1x : CH0〜CH3 選択

= 0b01 : CH0, CH1 選択

= 0b00 : CH0 選択

として，CH0 を選定する．

ADC_SCAN_ON = 0xEFFF = 0b1110 1111 1111 1111

CSCNA = 0b1 : Scan モードとする．

= 0b0 : Scan モードとしない．

により，Scan モードを選定する．この選定を受けて，

ADC_DMA_ADD_INC_4 = 0xEFCF = 0b1110 1111 1100 1111

SMPI = 0b0000 : Scan モードでなければ 0 とする．

= ⋯

= 0b0011 : Scan モードにおいて，スキャンするチャネル数を 4 とする．

により，Scan するチャネル数を 4 に設定する．さらに，図 5.126 の AD1CSSL レジスタ

AD1CON2

15	14	13	12	11	10	9	8
VCFG			-	-	CSCNA	CHPS	

7	6	5	4	3	2	1	0
BUFS	-	SMPI				BUFM	ALTS

図　5.125: AD1CON2 レジスタ

AD1CSSL

15	14	13	12	11	10	9	8
-	-	-	-	-	-	-	CSS8

7	6	5	4	3	2	1	0
CSS7	CSS6	CSS5	CSS4	CSS3	CSS2	CSS1	CSS0

図　5.126: AD1CSSL レジスタ

にて

CSS = 0b000110011

とすることで，スキャンするアナログ入力が AN5, AN5, AN1, AN0 であることを指定する．図 5.124 では，その他の関連箇所も朱書きで示してある．

図 5.127 は Scan モードにおけるデータ転送の様子を示す．CH0 において，AN0, AN1, AN4, AN5 のアナログ入力が順次 AD 変換され，その結果が DMA RAM に逐次転送される．4 入力のスキャンが終了した時点で割り込み処理が起動される．

図 5.128 は，ADC_DMA1Interrupt_Scan_4CH.c のプログラムのブロック図である．また，図 5.129 は ADC_DMA1Interrupt_Scan_4CH.c のプログラムの抜粋である．タイマ 3 からのトリガ 1 回につき 1 つのアナログ入力 ANx の AD 変換が行われる．

tsamp1 = 10000

により AD 変換の周波数を 4 [kHz] に設定することで，4 入力のスキャンを 1 [kHz] の周波数で実行する設定である．スキャンが 1 回終わる毎に割り込み処理関数 DMA1Interrupt() 関数が起動される．この関数では DMA RAM 内の ADCBuffr[ ] から各アナログ入力の変換値を読み出して，それぞれ v_{ANx} に格納している．v_{ANx} はスキャン 20 回分のデータを格納することとしているため，インデックス $i \geq 20$ となった時点で，$i = 0$ にリセットして，また v_{ANx} 配列の先頭から上書きすることを繰り返す．

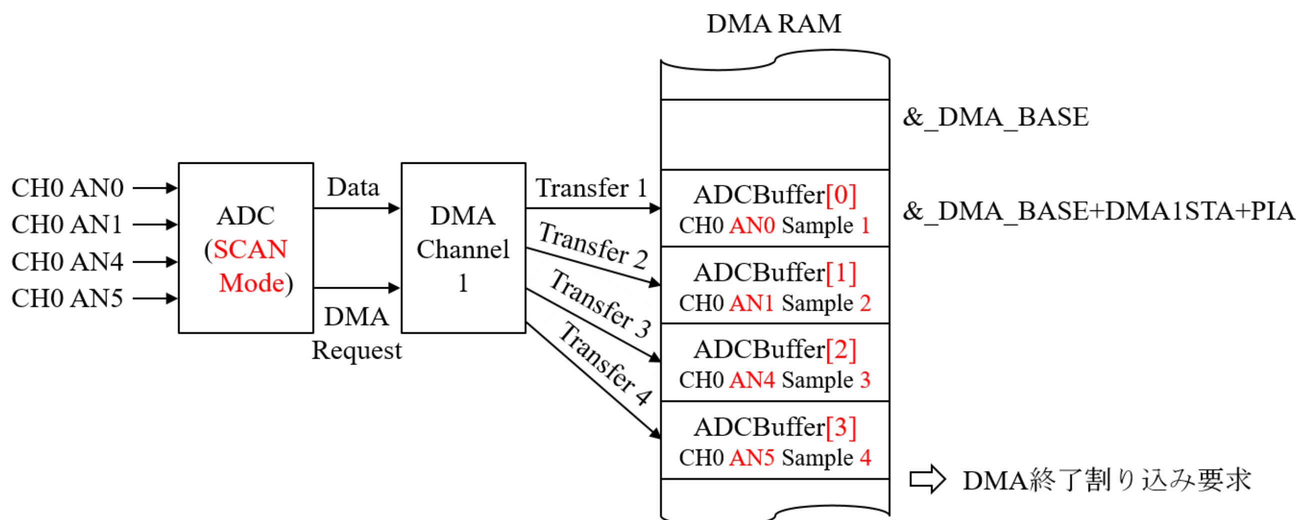


図 5.127: DMA によるデータ転送の様子 (4CH Scan モード)

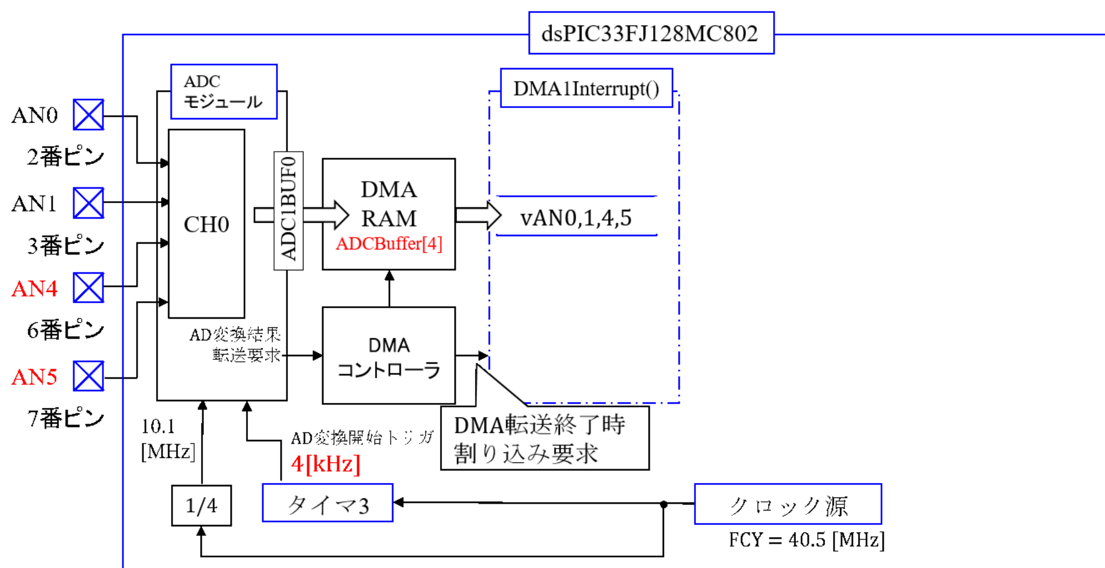


図 5.128: ADC_DMA1Interrupt_Scan_4CH.c プログラムのブロック図 (Scan モード)

```
//タイマ周期設定
const unsigned int      tsamp1 =  10000;                    // AD変換周波数設定  40MHz/tasmp1 = 4kHz
//DMAパラメータ設定
signed int vAN0[20], vAN1[20], vAN4[20], vAN5[20];      // vAN0, vAN1, vAN4, vAN5 格納用
#define  NUMSAMP 4
signed int ADCBuffer[NUMSAMP] __attribute__((space(dma)));
                                                         // DMAによりA/D変換結果を格納する配列の宣言
unsigned int i;

//DMA1割り込み処理関数
void __attribute__((interrupt, no_auto_psv)) _DMA1Interrupt(void)
{
        _LATA4 = 1;
        vAN0[i] = ADCBuffer[0];
        vAN1[i] = ADCBuffer[1];
        vAN4[i] = ADCBuffer[2];
        vAN5[i++] = ADCBuffer[3];
        if(i >= 20)
        {
            i = 0;
        }
        _LATA4 = 0;

       _DMA1IF = 0;                                      // 割り込みフラグクリア
}
```

図 5.129: ADC_DMA1Interrupt_Scan_4CH.c プログラム (Scan モード)

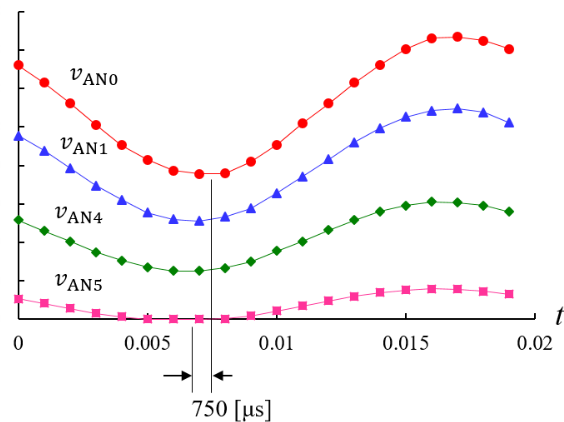


図 5.130: ADC_DMA1Interrupt_Scan_4CH.c プログラムの波形 (Scan モード)

図 5.130 は実験波形である．5.6.5 項と同様にして，MPLAB X IDE のデバッガ機能を利用した．実験では 50 [Hz] の正弦波信号を AN0, AN1, AN4, AN5 に印加した．ただし，振幅をおよそ 1 : 0.75 : 0.5 : 0.25 とした．デバッガにより，マイコンを実行 → Halt 状態にして，v_{ANx} を読み出し，CSV ファイルにセーブした後，エクセルによりグラフに表した．その際，$v_{\mathrm{AN0}}, v_{\mathrm{AN1}}, v_{\mathrm{AN4}}$ にそれぞれ異なる定数を加えて，各波形が縦方向に離れるようにした．各波形の最小点に着目すると，250 [μs] ずつずれている．これより，

$$\frac{1}{4[\mathrm{kHz}]} = 250[\mu\mathrm{s}] \tag{5.43}$$

間隔で各入力のスキャンが実行されたことを確認できる．

5.8　索引

索引

12 ビット DA 変換器, 39
3 ステートバッファ, 31

$\overline{\text{A}}$/B, 51
AD12B, 59
AD1CHS0 レジスタ, 62
AD1CON1 レジスタ, 57, 91
AD1CON2 レジスタ, 93, 118
AD1CON3 レジスタ, 63
AD1CON4 レジスタ, 93
AD1IE, 71
AD1IF, 71
AD1IP, 70
AD1PCFGL レジスタ, 34, 65
ADC CONVERSION (10-BIT MODE) TIMING REQUIREMENTS, 79
ADC1BUF0 レジスタ, 56
ADCBuffer[], 91
ADCS, 64
ADDMABM, 58, 92, 105
ADON, 58
ADRC, 64
ADSIDL, 58
AD 変換, 3
ALTS, 62
AMODE, 96, 106
ASAM, 60
attribute, 29
BUFM, 62
autoconversion, 71
BUF, 51
BUFM, 110

CH0NA, 63
CH0SA, 63
CHEN, 95
CHPS, 62, 93, 119
CKE, 47
CKP, 48
CLKDIV, 20
CLOCK DIVISOR REGISTER, 20
ConfigIntTimer1(), 24
Configuring Analog Port Pins, 34
const, 68
CPU Peripheral DS Bus, 86
CS, 39
CSCNA, 61, 119
CSS, 120

DAC Register, 50
DAConv, 41
DAConv() 関数, 42
Data Direction Register, 32
Data Direction Register, 23
Data Latch, 32
DATA MEMORY MAP, 86
DA 変換器, 39
Debugger, 76
DEVICE CONFIGURATION REGISTER MAP, 18
DIR, 95

DISSCK, 46
DISSDO, 46
DMA, 86
DMA Channel, 88
DMA CHANNEL TO PERIPHERAL ASSOCIATIONS, 97
DMA Data Transfer Example, 87
DMA RAM, 86
DMA Ready Peripheral, 86
DMA1CNT レジスタ, 98
DMA1CON レジスタ, 95
DMA1IF, 91
DMA1Interrupt() 関数, 90
DMA1PAD レジスタ, 97
DMA1REQ レジスタ, 96
DMA1STA レジスタ, 97
DMA1 初期設定関数, 94
DMA1 設定関数, 98
DMA1 割り込み処理関数, 90
DMA_BASE アドレス, 88
DMABL, 93, 106
DMA 対応周辺モジュール, 86
DMA コントローラ, 86
DOZEN, 23
dsPIC33F CONFIGURATION BITS DESCRIPTION, 18
D フリップフロップ, 32

extern, 68

Fcu, 23
FNOSC, 18
FNOSC_FRCPLL, 18
FORM, 59
FOSC, 18, 19
FOSCSEL, 18
FOUT, 20
FPOR, 19
FPWRT, 19
FRCDIV, 20
FRCPLL, 18
FR オシレータ, 20
FWDT, 19
FWDTEN, 19

$\overline{\text{GA}}$, 51

HALF, 95

I/O ピン, 31
I/O ポート, 31
ICD, 19
ICS, 19
ICSP, 9
ICS_PGD1, 19
IEC0, 27
IEC0 レジスタ, 71
IFS0, 27
init_tiemr1, 23
init_SPI(), 41
Input Register, 50
Interface Logic, 50
INTERRUP ENABLE CONTROL REGISTER 0, 27
INTERRUP FLAG STATUS REGISTER 0, 27
INTERRUP PRIORITY CONTROL REGISTER 0, 27
INTERRUPT PRIORITY CONTROL REGISTER, 70

Interrupt source, 28
Interrupt vector name, 28
INTERRUPT VECTORS, 28
IPC0, 27
IPC3, 70
IPC3 レジスタ, 70

LDAC, 39

MCP4922, 39
MODE, 96, 115
MODE16, 47
MPLAB® X IDE, 10
MPLAB® XC16 コンパイラ, 10
MSTEN, 48
MULIPLE CHANNELS WITH SEQUENTIAL SAMPLING, 79

no_auto_psv, 29
NULLW, 95

OpenTimer1(), 24
OSCIOFNC, 18
Output Multiplexer, 31
OUTPUT SELECTION FOR REMAPPABLE PIN, 40

Parallel I/O Port, 23
PCFG0, 65
Peripheral Indirect Addressing Mode, 96, 104
Peripheral Libraries, 10
Peripheral Module Enable, 31
Peripheral Output Data, 31
Peripheral Output Enable, 31
PGEC1, 19
PGED1, 19
PIA モード, 104
Ping-Pong モード, 113
PLL, 20
PLLDVI, 20
PLLFBD, 21
PLLPOST, 20
PLLPRE, 20
PPRE, 48
PR1, 26
Project ADC, 55
Project ADC_ADC1Interrupt, 67
Project ADC_ADC1Interrupt_autoconversion, 71
Project ADC_DMA1Interrupt_Scan_4CH, 118
Project ADC_DMA_BUFM, 109
Project ADC_DMAInterrupt, 87
Project ADC_DMAInterrupt_NUMSAMP4, 100, 104
Project ADC_DMA_Ping_Pong, 113
Project ADC_MUL_1.1Msps, 81
Project DAC, 39
Project IO_Ports_Digital, _Input33
Project Timer1_Interrupt, 11
psv, 29
PSVPAG, 29

RCON, 19
Read LAT, 36
Register Indirect with Post-Increment Mode, 96
Remappable Pin, 5
Remappable Pins, 39

Remappable ピン, 40
RPINRx レジスタ, 5
RPx, 5

SAMC, 64
SAMP, 60
Scan モード, 118
SCK, 39
SDI, 39
SDO, 39
sequential sampling, 80
Serial Peripheral Interface, 39
$\overline{\text{SHDN}}$, 51
SIMSAM, 92
SIMSAM, 59
SIZE, 95
SMP, 47
SMPI, 62, 93, 119
space(dma), 90
SPI, 39
spi.h, 46
SPI1BUF, 43
SPI1BUF レジスタ, 50
SPI1STAT レジスタ, 48
spi802.h, 45
SPI_CS, 42
SPIEN, 48
SPI_LDAC, 42
SPIROV, 49
SPISIDL, 49
SPIxCON1, 46
SPIxCON2, 46
SPIxSR レジスタ, 50
SPIxSTAT, 46
SPI モジュール, 45
SPRE, 48
SRAM, 86
SRAM X-Bus, 86
SSRC, 59
string DAC, 50
SWDTEN, 19
SYSTEM CONTROL REGISTER MAP, 23

T1CON, 25
T1IE, 27
T1IF, 28
T1IF, 26, 27
T1Interrupt(), 28
T1IP, 27
T1IE, 27
T1IP, 27
TAD, 65
TIMER1 CONTROL REGISTER, 25
TMR1, 26
TRIS, 32
TRIS Latch, 32
TRISx, 23

VCFG, 60
VDD, 5
volatile, 97
VSS, 5

WR LAT, 32
WR PORT, 33
WR TRIS, 32
WRITE COMMAND REGISTER, 51

XC16 C Compiler User's Guide, 29

アナログポート設定, 34
インストラクションサイクルクロック, 23
インタラプト設定関数 (DMA), 98
オペアンプ, 50
グローバル変数, 73
システムクロック, 19, 23
周辺モジュール用クロック, 23
周辺モジュール, 31
出力ポート, 32
シュミットトリガ, 35
シーケンシャルサンプリング, 79
タイマ 1 割り込み処理関数, 28
TIMER2/3 and 4/5, 17
TIMER1, 17
抵抗分圧方式 DA 変換器, 50
ディジタル入力用ポート, 34
ディジタルポート設定, 34
デバイスインストラクションクロック, 23
デバイスオペレーティングクロック, 23
デバッガ, 76
データシート, 5, 8, 17
同時サンプリング, 86
内蔵 PLL, 18
内蔵発振子, 18
ヒステリシス特性, 35
ピン配置, 5
プライマリオシレータ, 18
連続モード (DMA), 96
割り込み, 17

参考文献

[1] 後閑哲也「電子制御・信号処理のための dsPIC 活用ガイドブック」技術評論社，2006.

[2] 古橋武「パワーエレクトロニクスノート」コロナ社，2008.

[3] 古橋武「モータドライブノート」

[4] 古橋武「ラジオノート」

平成 30 年 11 月 25 日

著者
古橋　武
名古屋大学工学研究科情報・通信工学専攻
名古屋大学名誉教授（令和 2 年 4 月 1 日より）

本稿の内容は，著作権法上で認められている例外を除き，著者の許可なく複写することはできません．