ETCB Manual

HARDWARE & PROGRAMMING MANUAL

株式会社知能機械研究所

ETCB Manual Version 2.0

http://chinoken.jp
support@chinoken.jp

目次

目次

目次

始めに

このたびは弊社ロボット制御プログラミングボードETCBをお選びいただきまして誠にありがとうございます。本マニュアルをよくお読みいただいた上でご利用なさいますようお願い申し上げます。

注意事項

- ETCBは電気・電子部品でできています。湿気の多いところやほこりっぽい場所などでのご使用にならないでください。同様に濡れた手で扱ったり、電気・電子部品を直接触ったりしてはいけません。
- 仕様に合わない使い方はしないでください。
- 改造をしないでください。
- 動作がおかしいと感じた場合はすぐに使用を止め、すぐに弊社サポートまでご連絡ください。
- Raspberry Piは英国Raspberry Pi財団の登録商標です。

免責事項

ETCBを使用したときのいかなる結果においても弊社では責任を負いません。

同梱品

ETCBには次の部品が付属しています。

- ETCB　1個
- バインド小ねじ M2.6-12　4本
- バインド小ねじ M2.6-4　4本
- 樹脂スペーサΦ2.6L11　4本
- 樹脂スペーサΦ2.6L6　4本
- 2.54ピッチジャンパ　３個
- 2.54ピッチ、13x2ピンソケット　1個

もし足りない場合は下記サポートまでご連絡ください。

別途必要なもの

- ９～１２Vの直流電源またはバッテリー
- シリアル通信用のUSBケーブル（USB-AオスーmicroBオス）
- Raspberry Piと接続する場合は半田と半田ごて

サポート

不具合、修理やご質問は下記メールアドレスへご連絡ください。不具合の詳細（不具合の内容、不具合発生時の状況など）とご連絡先（氏名・メールアドレス）を忘れずお書き添えください。内容によりましては弊社より折り返しご質問させていただくことがあります。あらかじめご了解ください。

support@chinoken.jp

ETCB仕様

概要

ETCBは浮動小数点演算回路をもつ最大72MHzで動作するSTM32F302 MCUを搭載したロボット用制御ボードです。ドーターボードとしてRaspberry Piと接続すると、ETCBボードからRaspberry Pi駆動用の電源を供給できます。またUART、I2C、SPIなどの通信ポートを共有してRaspberry Piと通信できます。

外部ポートとしてADC入力ポート、UARTポート、USBポートなどが準備されておりセンサーを使った自律動作するロボットや遠隔操縦ロボットのプログラミングがすぐに始められます。

市販のサーボモーターが使用できるサーボポートを２系統12ポート用意しています。最大6Aの大電流を流せるので大型のロボットも作れます。さらにサーボポートへの電源供給ON/OFFを切り替えることができますので、省電力モードも実現できます。

AtollicのTrueSTUDIO とSTMicroelectornics社のSTM32CubeMX、およびST-LINK/v2デバッガに対応しているので、組み込み初心者でも比較的容易にプログラム・デバッグができます。プログラムファイルはUSB書き込みに対応していますので、ST-LINK/v2を持っていなくても書き込みができます。

仕様

項目	サブ項目	内容
電源	推奨入力電圧 最大入力電圧	6.6V～12V 16V
	出力電圧	内部3.3V 外部5.0V
	最大出力電流値 最大連続出力電流値	最大2.2A 2A以下
MCU	型番	STMicroelectronics STM32F302C8
	メモリー	16kB SRAM 64kB Flash
	CLOCK	72MHz（最大・標準）
IO	ADコンバータ	・12bit ADC入力ポートx4（カットオフ周波数1.6kHzのローパスフィルター付き） ・ADCリファレンス電圧は3.3Vまたは5.0Vのいずれかを選択可能。ただしADC入力電圧は3.3Vを超えてはならない
	サーボ	・シリアルサーボ、コマンドサーボに対応 ・PWMサーボは使用できません ・サーボポート2系統（UART2、UART3）各6ポート ・信号ライン電圧5.0Vサーボに対応 ・UART2、UART3ポートと排他的利用可能 ・サーボ電圧は入力電源電圧と同じ

		・PA15ピンにてサーボへの電源供給ON/OFF切り替え可能 ・サーボポートは1系統あたり最大連続電流値3A以下を推奨
	USB	・PC接続用USBポートx1（FTDI FT230XS経由で仮想COMポートとして使用） ・USBポートx1（MCUのUSB DP、DM端子と接続）
	UART1	・外部拡張ポート（EXT）、COM1ポートと共有可 ・Raspberry Piとポート共有可能
	UART2 UART3	・UART2ポートおよびサーボポートS7～S12に接続、UART2ポートとS1～S6ポートは排他的利用可能 ・UART3ポートおよびサーボポートS1～S6に接続、UART3ポートとS7～S12ポートは排他的利用可能
	I2C	・I2C1として利用 ・Raspberry Piとジャンパピンにて共有可能 ・I2C_SDAおよびI2C_SCL端子は10kΩでプルアップ済み
	SPI	・SPI1として使用 ・Raspberry Piと接続
	GPIO1～GPIO4	・EXTポートに接続 ・MCU GPIOポートまたはTSC_G2_IO1～4ポートとして使用可能
	GPIO5 GPIO6	・Raspberry Pi GPIO22と接続 ・Raspberry Pi GPIO23と接続
LED	D1	・仮想COMポートを使って、MCU側より送信時に点灯（変更可能）
	D2	・EXTポートに接続 ・フルカラーLED ・MCU端子PA8、PA9、PA10にLED RED、GREEN、BLUEがそれぞれ接続 ・PA8～10はAlternate FunctionとしてTIM_CH1～3に変更できるので、PWM出力可能
書き込み	USB1	・DFUポートをショートしてETCBボードを起動すると、PCからDFU形式のプログラムが書き込み可能
	ST-LINK	・EXTポートの3.3V OUT、GND、SWDIO、SWCLK、NRSTポートを使用してSTMicroelectornics ST-LINK/v2でプログラム・デバック可能

対応Raspberry Pi

ETCBが対応しているRaspberry Piは以下の通りです。ただしいずれのRaspberry PiもETCBが供給可能な電流値を超えて使用はできません。

- Raspberry Pi Model B
- Raspberry Pi Model B+
- Raspberry Pi 2 Model B

- Raspberry Pi 3 Model B

注意事項

- 6.6V以下で使用するとETCBの5V出力端子の電圧が低くなります。そのためアナログセンサに5Vを供給する場合は注意してください。またRaspberry Piを取り付けていた場合は画面上に電圧低下警告が表示されることがあります。
- サーボモーターを取り付けている場合は、サーボモーターが動作するのに十分なバッテリーまたは電源を接続してください。電源の性能により供給電力が不十分だったり急激な電圧低下が起こったりするとMCUがリセットされることがあります。
- 一般的にシリアルサーボやコマンドサーボといった、UART（Universal Asynchronous Receiver Transmitter）通信で動作するサーボモーターのみ使用できます。PWMサーボは使用できません。
- STM32CubeMXやTrueSTUDIOでプログラムを作成する際には、サーボモーター電源をON/OFFできる端子SVO_POW_SW（PA15、「図 2　MCUピン配置」を参照）をGPIO Outputポートとして定義してください。定義しないとサーボモーターに電源が入りません。
- Raspberry Piと接続する13x2ピンソケットはユーザー様が半田付けにて取り付ける必要があります。

外部接続詳細

外観・ポート

下図は基板寸法図とポート位置、MCUピン割り当て、およびEXTポートです。色はポートカラーと合わせています。薄い水色はRaspberry Piの拡張ポートと直接接続されます。

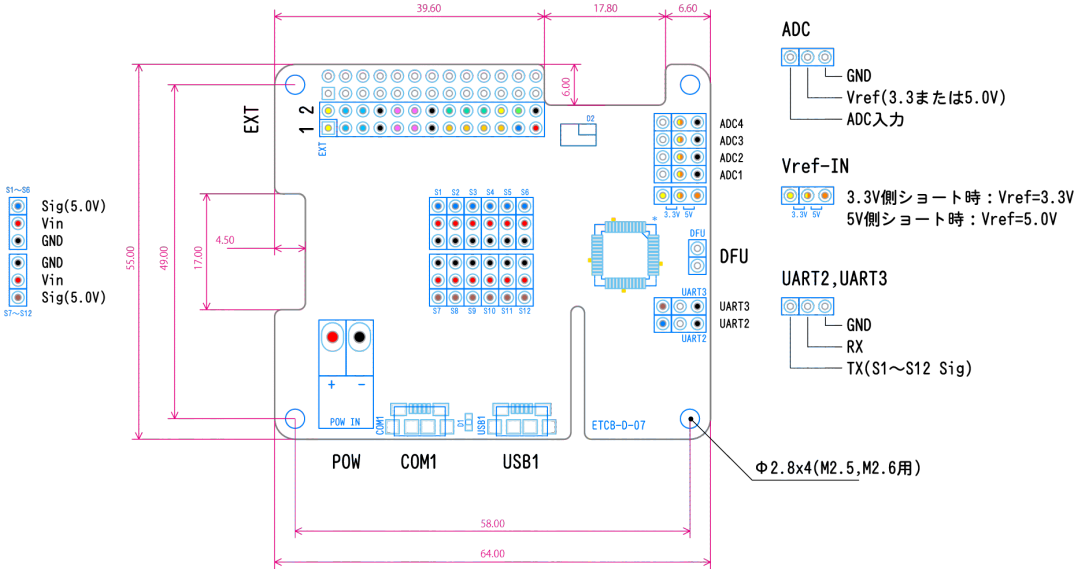


図 1　外形寸法およびポート位置

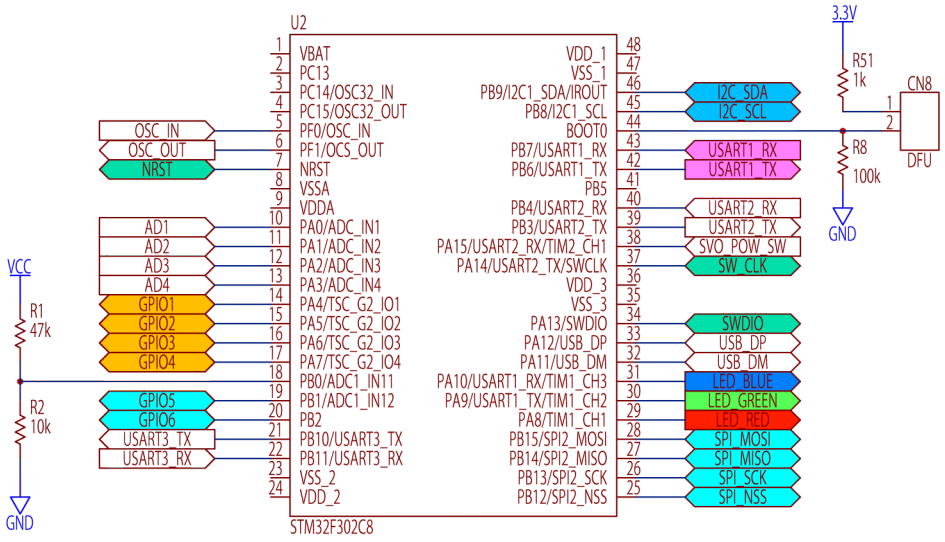


図 2　MCUピン配置

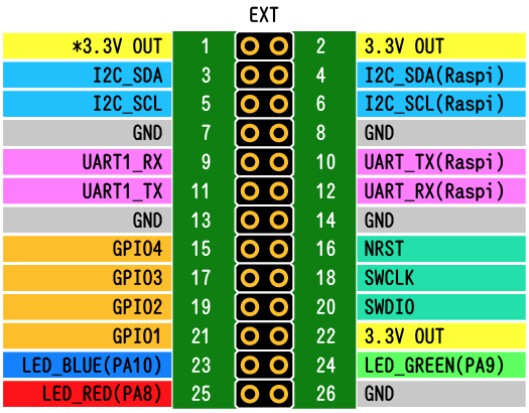


図 3　EXTポート・端子名称

ETCB接続ポート

電源（POW）

ETCBに電源を供給します。

電源コネクタは日本圧着端子製造社製のVHコネクタ（S2P-VH）を使っています。＋とーを間違えないで、使用の範囲内（6.6～12V）の電圧でご利用ください。Li-Fe 2セル、3セルまたはLi-Po2セル、3セルバッテリーが使用できます。サーボ端子には電源電圧がかかりますので、入力電圧とサーボ電圧が正しい組み合わせになっているか確認してください。

COMポート（COM1）

COM1ポートでPCやその他デバイスとシリアル通信ができます。

COM1ポートはMCUのUART1ポートと接続されています。途中にFTDI社USB-シリアル変換ICが入っていますので、PCとUSBケーブルでつなぐとPC側では仮想COMポートとして認識します。FTDI社のデバイスドライバ[1]が必要です。

AD変換ポート（ADC）

ADCポートは12ビットアナログ入力ポートです。

---

[1] FTDI社ホームページ（http://www.ftdichip.com/Drivers/VCP.htm）よりVCP（Viratual COM Port）ドライバをダウンロードしてインストールしてください。

切り替えジャンパで3.3Vまたは5.0V電源をセンサー側に供給します。MCUのリファレンス電圧は3.3Vで入力最大電圧も3.3Vになっているので、5.0Vのセンサーをつないだ場合でもADC入力端子の電圧は3.3Vを超えないように注意してください。

AD1～4ポート（ADC1_IN1～ADC1_IN4）ポートは1.6kHzのローパスフィルターが入っています。

ADC1_IN11ポート（PB0）は電源電圧監視専用ADCポートです。電源電圧を10/57に分圧しています。

UART2・UART3

下記サーボポート（Sポート）と排他的共有ができるシリアルポートです。サーボポートは半二重通信しかできませんが、全二重通信をする場合にUARTポートを使います。サーボポートと違って、信号の電圧レベルは3.3Vですので注意してください。

UART2ポートのTX端子はサーボポートS7～S12に接続されています。UART3ポートのTX端子はS1～S6に接続されています。

S1～S2

市販のシリアルサーボモーターを使うためのポートです。各端子は外側から信号・電源・グランドとなります。

信号レベルは5.0Vです。通信プログラムを作成する場合はMCUポート設定でHalf Duplexに設定してお使いください。UART2、UART3ポートとは排他的使用となります。

EXTポート

UART1（UART_TX・UART_RX）

UARTポートはMCUのUART1ポートに接続されています。またCOMポートにも接続されています。EXTポートのUARTポートとCOMポートは排他的使用となります。

EXTポートのUART1_TXとUART1_RX端子をそのまま使用すると、ETCBのMCUとシリアル通信ができます。Raspberry PiのUART端子もそのまま使えます。

UART1_TXとUART1_RX(Raspi)端子、UART1_RXとUART1_TX(Raspi)端子をジャンパピンでショートするとETCBのMCUとRaspberry Piでシリアル通信ができます。

I2C

MCUのI2C1ポートと接続されています。10kΩでプルアップ済みです。I2Cポートとジャンパピンでショートするとと Raspberry PiのI2Cポートと接続されます。ショートさせないで外部I2Cデバイスに接続することもできます。外部I2Cデバイスと接続するときはRaspberry Piの3.3V端子は使用せず、EXTポート3.3V端子を使用してください。

その他通信ポート

SPI

Raspberry PiのSPIと接続済みです。Raspberry PiとMCU間でSPI通信するときに使用します。

GPIO1～4

GPIOポートです。

信号レベルは0～3.3Vになりますので、入力に使う場合は5V信号を入れないでください。Raspberry PiとETCBの両方から出力しないようにしてください。

GPIO5・GPIO6

GPIO5とGPIO6ポートはRaspberry PiのGPIO2、GPIO3ポートとそれぞれつながっています。Raspberry PiとETCBの両方から出力しないようにしてください。

PWM（フルカラーLED）

LED_RED、LED_GREEN、LED_BLUE はそれぞれMCUのPA8（TIM1_CH1）、PA9（TIM1_CH2）、PA10（TIM1_CH3）とつながっています。フルカラーLEDをPWMで表示する場合は、表示間隔（PWM周期）を10～20msにするときれいに表示されます。

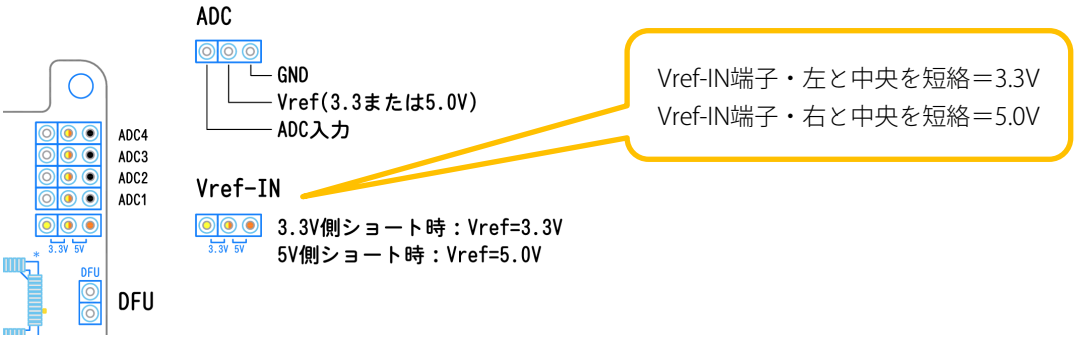
開発環境の構築

本節ではETCBでプログラムを作成したり、デバッグを行ったりするための開発環境のインストールや設定について説明します。

ハードウェアの準備・確認

ADポート入力電圧の切り替え

ETCBのADポート基準（リファレンス）電圧は3.3Vですが、アナログセンサに供給できる電圧は3.3Vまたは5.0Vのどちらか選択できます。お使いのアナログセンサが3.3V系の場合は「外観・ポート」節のVref-IN端子3本の内、左と中央の端子を付属のジャンパでショートします。5.0Vの場合は右と中央の端子をショートします。どちらもショートしていない場合、ADポートの中央の端子はオープンポートとなり、センサーに電源を供給しません。別電源を持つセンサーを使う場合はオープンで使用します。



※基準電圧（3.3Vまたは5.0V）にかかわらず、ADC入力ポートの最大入力電圧は3.3Vです

Raspberry Piとドッキング

ETCBをRaspberry Piとドッキングして使用する場合は付属の13x2ピンソケットをEXTポートの隣の未接続端子に半田付けしてください。

Raspberry Piと通信する場合は、UART通信とI2C通信が使えます。I2C通信をする場合は**エラー! ブックマークが定義されていません。**ページ**エラー! 参照元が見つかりません。**を参考にして、I2C_SDA(3)とI2C_SDA(Raspi)(4)を付属のジャンパでショートします。また、I2C_SCL(5)とI2C_SCL(Raspi)(6)もショートします。UART通信をする場合は同様にUART1_RXとUART1_RX(Raspi)をショートし、UART1_TXとUART1_TX(Raspi)もショートします。

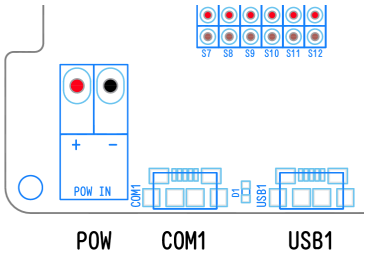
UART1端子はUSBのCOM1とも接続していますので、EXTポートのUART1端子をショートすると、ETCBのMCU、Raspberry Pi、USBのTX、RXが全てつながりますが、それらは同時に通信することはできません。MCUーRaspberry Pi間通信またはMCUーUSB間通信のどちらかのみ使用してください。

I2C、UARTを単独（MCU単独またはRaspberry Pi単独）で使用したいときは、EXTポートから直接通信ラインを引き出して使用してください。

SPIポート、GPIO5・6ポートはRaspberry Piとハードウェア接続していますので、ジャンパは不要です。

別途準備するもの

- 6.6～12V電源。2アンペア以上出力できるACアダプタで動作します。電圧は使用するサーボモーターによって異なります。サーボモーターを使わない場合は12VのACアダプタをおすすめします。コネクタは日本圧着端子製造株式会社のVHR-2Nコネクタを取り付けるか、市販のコード変換アダプタをお使いください。



- USB1ポートからプログラムを書き込むためにはUSB(AオスーミニBオス)ケーブルが必要です。
- ST-LINKを使ったプログラム書き込みやデバッグを行う場合は、24ページの「図４　ST-LINKとの接続」図を参考にしてケーブルで接続してください。一般的なジャンパケーブル（メスーメス）が使用できます。

ソフトウェアのインストール

ETCBではプログラム開発のために以下のソフトウェアを使用します。

- Atollic TrueSTUDIO Lite：　Atollic社の提供する無料の統合開発環境です。
- DfuSe Demonstration：　USBケーブルで（USB1端子）プログラムを書き込むためのツールです。ST-LINKを使用する場合は不要です。
- DFU File Manager：　TrueSTUDIOで作成したプログラム（HEX形式）をDFU形式に変換するツールです。DFU形式に変換するとDfuSe DemonstrationツールでUSB端子からプログラムを書き込めるようになります。
- STM32CubeMX：　STMicroelectronics社の提供する、C言語コードジェネレーター（プログラム生成ソフトウェア）です。マウス操作で簡単にMCUの初期設定やUSB設定コードを生成できます。コードは、Atollic TrueSTUDIOにインポートして使用できます。生成する内容は、ペリフェラル（周辺機器）のポート設定、クロック周波数、タイマー（周期、PWM設定など）、AD変換（ポート登録、変換時間設定）、UART・I2C・SPIなどの通信設定、DMA転送設定、割り込み設

定などほとんどの初期設定項目をマウスで設定し、C言語コードを生成できます。
設定した項目はプロジェクトファイル（ioc形式）として保存できて、後から編集することも可能です。ETCBではサンプルプログラムを可能な限りSTM32CubeMXのプロジェクトファイルで提供する予定です。

Atollic TrueSTUDIOのインストール

開発環境にはAtollic社のTrueSTUDIOを推奨しています。始めにAtollic社サイトからTrueSTUDIO Lite版をダウンロードしインストールしてください。できるだけ最新のものを使用するようにしてください。
ダウンロードサイト：http://timor.atollic.com/resources/downloads

インストールの途中でデバッグ用デバイスのドライバインストール画面が表示されますので、ST-LINKにチェックマークを入れてインストールしてください。

DfuSe Demonstrationのインストール

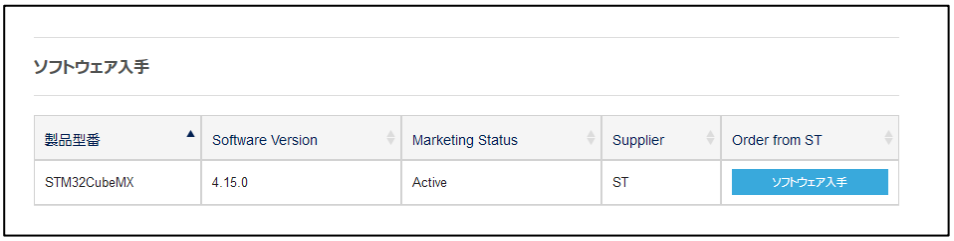
USBケーブルを使った書き込みをするには、TrueSTUDIOで作成したELF形式実行ファイルをDfuSe Demonstrationソフトウェアを使ってDFU形式(Device Firmware Upgrade)に変更する必要があります。DfuSe Demonstrationソフトウェアは下記手順でダウンロードしてください。ST-LINKを持っている場合は、本ソフトウェアは不要です。

1. STMicroelectronics社サイトへアクセス
   http://www.st.com/content/st_com/ja.html
2. 右上の検索キーワード入力欄で「DFU」または「STSW-STM32080」と入力し、検索します。
3. 検索結果一覧の「製品型番」欄で「STSW-STM32080」を探してください
4. 「STSW-STM32080」がリンクになっていますので、クリックして先へ進みます。
5. 表示されたページの一番下にSTSW-STM32080の「ソフトウェア入手」ボタンがありますので、クリックします。
6. ソフトウェアのライセンス契約内容が表示されますので、よく読んでライセンスに合意する場合はACCEPTボタンをクリックします。
7. 氏名およびメールアドレス入力欄が表示されますので、全て記入し「提出」ボタンをクリックします。
8. ダウンロードリンクが指定したメールアドレスに届きますので、DfuSeソフトウェア（ファイル名：en.stsw-stm32080.zip）をダウンロードしてインストールしてください。

STM32CubeMXのインストール

STM32CubeMXはMCUのポート設定やタイマー、割り込み、DMAなどの諸設定をGUIで行うツールです。ソフトウェア開発用のひな形となるソースコードを出力します。下記の手順でダウンロードしてください。

1. STMicroelectronics社サイトへアクセスします。
   http://www.st.com/content/st_com/ja.html
2. 右上の検索キーワード入力欄で「STM32CubeMX」と入力し、検索します。
3. 検索結果一覧の「製品型番」欄で「STM32CubeMX」を探してください。
4. 「STM32CubeMX」がリンクになっていますので、クリックして先へ進みます。
5. 表示されたページの一番下にSTM32CubeMXの「ソフトウェア入手」ボタンがありますので、クリックします（画面はバージョン4.15.0のとき）。



6. ソフトウェアのライセンス契約内容が表示されますので、よく読んでライセンスに合意する場合はACCEPTボタンをクリックします。
7. 氏名およびメールアドレス入力欄が表示されますので、全て記入し「提出」ボタンをクリックします。
8. 記入したメールアドレスにダウンロードリンクが届きますので、DfuSeソフトウェア（ファイル名：en.st32cubemx.zip）をダウンロードしてインストールしてください。

ETCBのMCU用ライブラリのインストール

1. インターネットに接続している状態で、STM32CubeMXを起動してください。
2. メインメニューから、Help＞Install New Libiriesを選択してください。
3. STM32CubeF3 Releasesの欄で、一番新しい「Firmware Package for Family STM32F3」を選択して、Install Nowボタンでインストールしてください。

STM32CubeMXアップデートとライブラリのアップデート

STM32CubeMXのアップデート方法です。STM32CubeMXでは頻繁に本体やライブラリがアップデートされていますので、常に最新版となるようにしてください。

STM32CubeMXのアップデート

1. STM32CubeMXを起動してください。
2. メインメニューの「Help」メニューから「Check for Updates」を選んでください。
3. 「Check Update Manager」ダイアログが表示されますので、下にある「Check」ボタンで最新版の確認をします。
4. アップデートファイルがあった場合は、「Install Now」ボタンでアップデータのダウンロードとインストール準備を行います。
5. 「Install Now」ボタンでファイルのダウンロードが完了すると、再起動するように指示が出ます。いったんSTM32CubeMXを終了してから、管理者権限で起動してください。STM32CubeMXを管理者権限で起動するには、Windowsショートカットアイコンまたはスタートメニューのアイコンを右クリックして、「管理者として実行」を選択してください。
6. 起動時に自動的にアップデートされます。

ライブラリのアップデート

1. STM32CubeMXを起動してください。
2. メインメニューの「Help」メニューから「Check for Updates」を選んでください。
3. メインメニューの「Help」メニューから「Install New Libraries」を選んでください。
4. ダイアログが表示されますので、「STM32CubeF3 Releases」欄から「Firmware Package For STM32 F3」の四角マークをクリックしてチェックマークをいれます。最新のバージョンを選択してください。
5. ダイアログ一番下の「Install Now」ボタンでファームウェアパッケージをインストールします。

プログラム作成と書き込み・実行

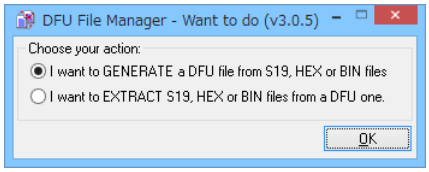
USBケーブルで書き込み

USBケーブルでプログラムを書き込むには、HEX形式ファイルをDFU形式に変換します。DFU形式のファイルをDfuSe Demoソフトウェアで書き込みます。

ST-LINKでプログラムを書き込む場合は24ページ「ST-LINKを使用する場合」を参照してください。

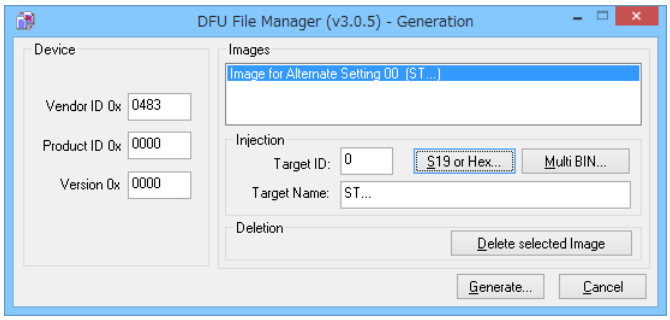
HEXファイルをDFU形式に変換する

1. Dfu file managerを起動します。I want to GENERATE a DFU file from S19, HEX or BIN filesにチェックを入れ、OKボタンを押します。



2. 「S19 or Hex...」ボタンを押して、変換対象のファイルを選択します。今回はサンプルプログラム（LED_TEST.elf.hex）を使用します。

3. Generateボタンを押して、DFUファイルへの変換を実行します。変換後のファイル名の入力が求められますのでDFUファイルの名前（今回はLED_TEST.dfu）を入力して「保存」ボタンを押します。



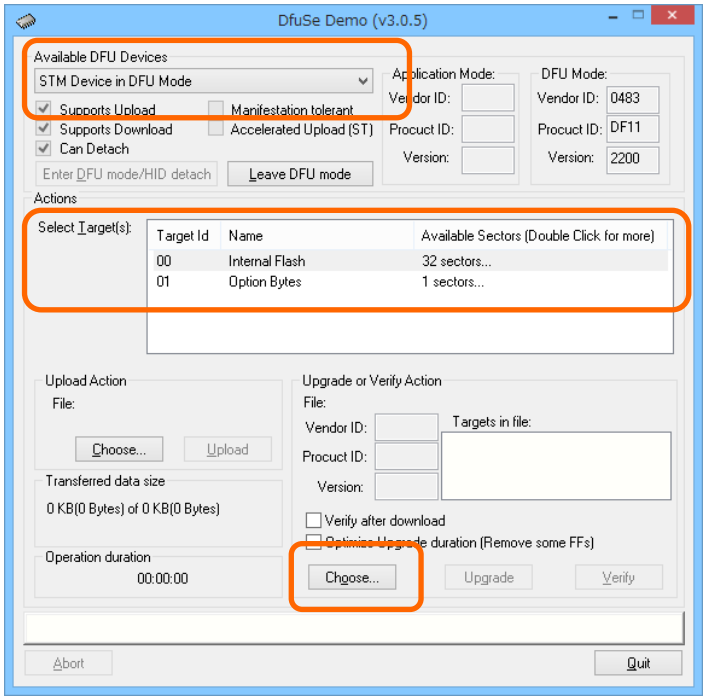
4. ETCBボードのDFU端子をジャンパでショートさせた状態で電源を入れます。ジャンパを刺すためのピンヘッダは付属しておりませんので、ピンヘッダをお客様で半田付けしていただくか、ピンセットのようなものでDFU端子をショートさせてください。半田付けや端子のショートは間違えないように十分気をつけて行うようにしてください。
   USB1のコネクタにマイクロUSBケーブルを取り付けPCと接続します。初回はデバイスドライバ（STM Device in DFU Mode）のインストールが実行されます。

DFUファイルの書き込み

1. DfuSe Demoを起動します。「Available DFU Devices」のプルダウンメニューに「STM Device in DFU Mode」と表示されていることを確認してください。何も表示されていない場合は、4の手

順に戻ってDFUモードで起動しなおしてください。
また、DfuSe Demoウィンドウの真ん中にある、「Select Target(s):」の欄に、「Internal Flash　32sectors...」、「Option bytes　1sectors...」とあるか確認してください。表示が違う場合は、DFUモードで起動していません（MCUを認識していない）。この場合も4の手順に戻ってDFUモードで起動しなおしてください。

2. 「Update or Verify Action」の「Choose…」ボタンを押し、転送対象のDFUファイルを選択します。今回は手順3で作成したLED_TEST.dfuを指定します。



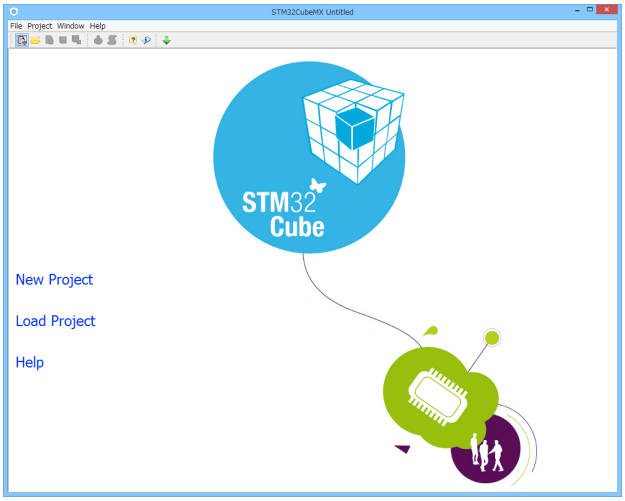
3. Updateボタンを押し、ETCBボードへプログラムを転送します。書き込み中はプログレスバーに進行状況が表示されます。プログレスバーに「Target 00: Upgrade successful !」と表示されたら書き込み完了です。

4. 転送終了後にDFUピンのジャンパを取り外し、電源を入れなおします。

ソースコード編集・新規プログラム作成

本節ではSTM32CubeMXサンプルプロジェクト（テンプレート）を使って、LEDが時間経過に沿ってゆっくりと色が変わっていくプログラムの作成・編集を行います。STM32CubeMXの使い方の詳細は29ページからの「STM32CubeMXでETCBプログラムひな形を作る」を参照してください。

STM32CubeMXでソースコード生成



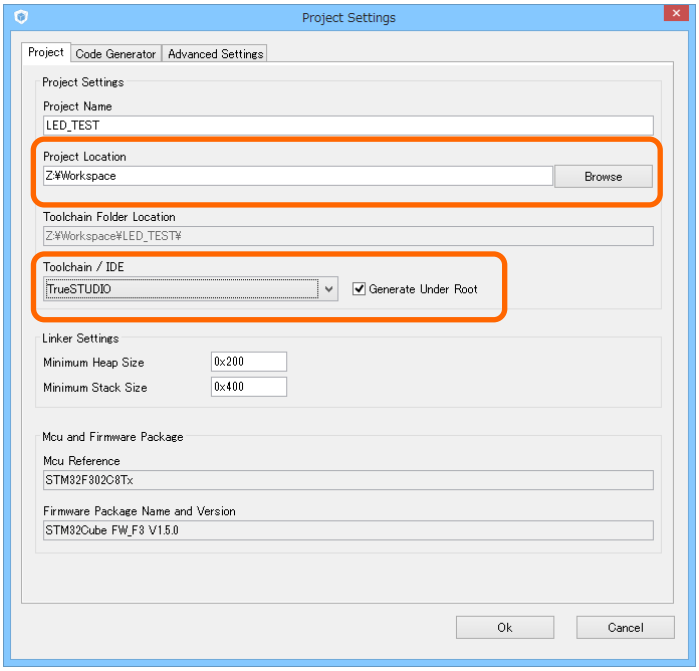
1．ETCB_Turtle_temp.iocをダブルクリックして、「STM32CubeMX」を起動します。
ETCB_Turtle_tempはETCBの基本的な機能が予め設定されているテンプレートファイルです。COM1ポートやAD変換は表の構成となっています。

項目	内容
MPU CLOCK	48MHz
ADコンバータ	・AD1、AD2、AD3、AD4、バッテリー電圧、MPU内部温度の6チャンネルのデータを取得 ・変換後のデータはDMAを用いてメモリーへ転送
UART1	・全二重通信 ・通信速度：115200[bps]、データ幅：8ビット、パリティ：無 ・データの送信：DMA転送可、データの受信：割り込み
UART2_TX(PB3), UART3_TX(PB10)	・半二重通信 ・通信速度：115200[bps]、データ幅：8ビット、パリティ：無 ストップビット：1ビット ・割り込み、DMA：未使用
TIM1	・LED_RED、LED_GREEN、LED_BLUE用のPWM信号を生成 ・PWM周期：15ms ・PWM幅の設定：0～99
TIM2	・15ms周期のカウンタ

	・定期的な処理を組み込む際に使用

2．Projectメニューから「Settings」を選びます。Project Settingsダイアログが表示されますので、Projectタブを選択してProjectに関する情報を下記のように入力します。「Project Location」の欄は任意です。通常はTrueSTUDIOを初回起動したときに作成されるWorkspaceフォルダを指定します。

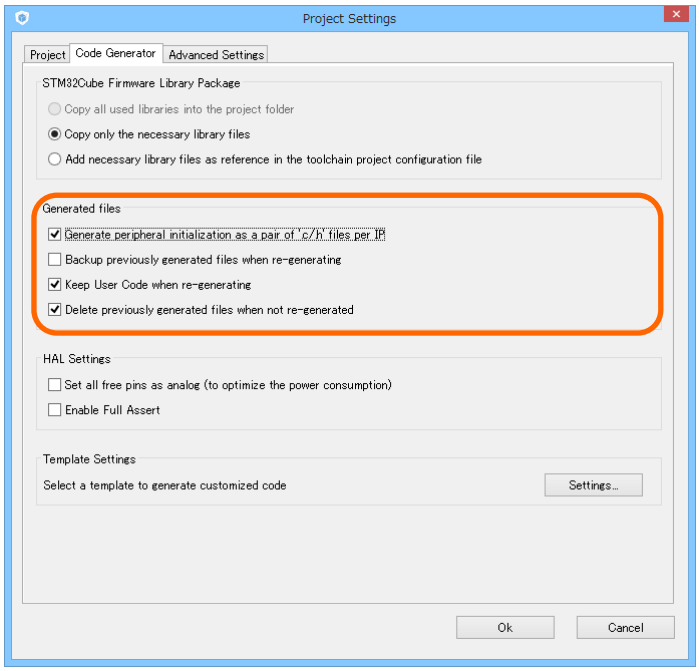
また、「Toolchain / IDE」はTrueSTUDIOを選択してください。



項目	内容
Project Name	LED_TEST
Project Location	任意のフォルダ
Toolchain / IDE	TrueSTUDIO
Generate Under Root	チェックを入れます

3．Code Generatorタブに切り替え、「Copy only the necessary library files」を選択します。また「Generated files」欄で「generate peripheral initialization as a pair of 'C/h' fules per Peripheral」にチ

ェック[2]を入れます。その他の欄はデフォルトで問題ありません（下記の図を参考にしてください）。



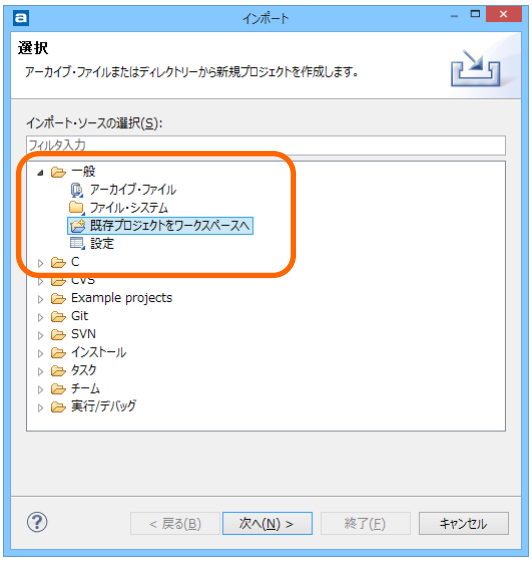
4．Projectメニューから「Generate Code」を選択しソフトウェア開発用のひな形となるソースコードを生成します。ソースコードは２番で指定した、「Toolchain Folder Location」欄に記載されているフォルダに保存されます。

TrueSTUDIOを使ったソースコード編集・デバッグ

1．TrueSTUDIOを起動します。ワークスペース・ランチャー（ダイアログ）が表示されますので、STM32CubeMXで作成したWorkspaceフォルダを指定してください。

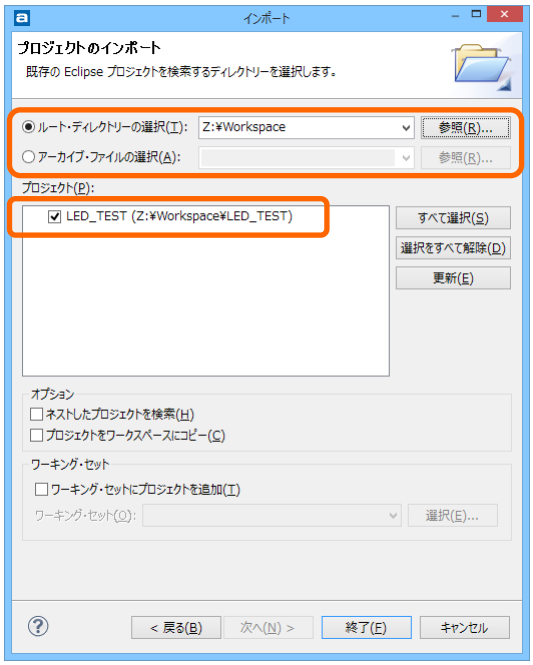
[2] generate peripheral initialization as a pair of 'C/h' fules per Peripheralにチェックを入れると、AD変換などの周辺回路のひな形ソースコードも生成されます。

2．「ファイル」メニューから「インポート」を選択します。インポートダイアログ（選択）が表示されますので、「一般」から「既存プロジェクトをワークスペースへ」を選択し、「次へ」ボタンを押します。



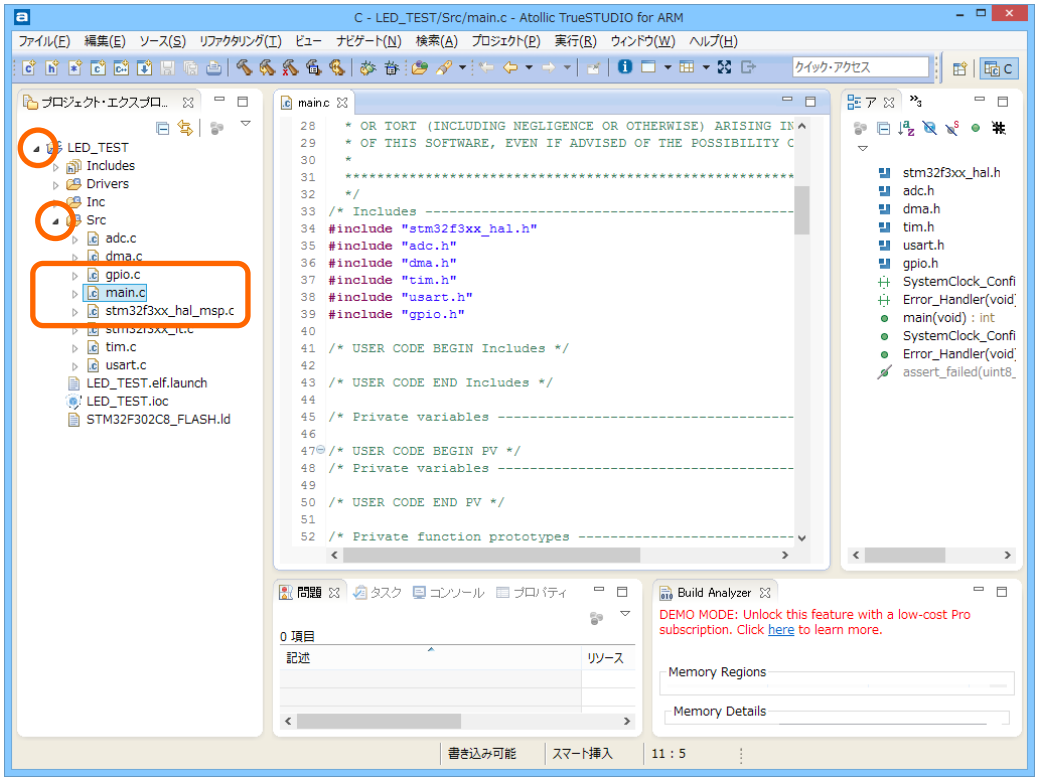
3．インポートダイアログ（プロジェクトのインポート）で「ルート・ディレクトリの選択(T):」を選択し、参照ボタンをクリックしてSTM32CubeMXで作成したWorkspaceフォルダを指定してください。

4．「プロジェクト(P):」欄にWorkspaceにあるLED_TESTプロジェクトが表示されていることを確認して、右下の「終了」ボタンを押します。ボタンを押すとインポートが開始されます。画面上のL



ED_TESTのチェック欄が☑チェックマークでは無くて黒い■マークになっている場合は、オプション欄で「ネストしたプロジェクトを検索」を選択してください。

5．プロジェクト・エクスプローラでLED_TESTの左側の三角マークをクリックしプロジェクトのリストを展開します。Srcの左側の三角マークをクリックしソースファイルのリストを展開します。「main.c」ファイルをダブルクリックしウィンドウに開きます。



6．main.cにLEDの輝度を調整するコードを追記します（赤字の部分）。なお**ユーザーが書き入れるコードは必ず「USER CODE BEGIN」から「USER CODE END」の間**となります。それ以外の場所にコードを書き入れると、STM32CubeMXでコード生成を行ったときにユーザーコードが消える場合があります。

先に定義されている関数や変数は、記入途中でSHIFTキーとスペースキーを同時押しすると、名称の補完候補が表示されます。

下記コードをコピー＆ペーストすると、全角スペースが挿入されてしまうことがあります。エラー表示で「stray '@' in program」や「stray '¥201' in program」などと表示された場合は、該当のプログラムに全角スペースが混じっていますので、半角スペースなどに書き換えてください。

```
/* Private function prototypes -----------------------------------------------*/
void LED(uint8_t , uint8_t , uint8_t);   // LEDのPWMを設定する関数のプロトタイプ
```

```
  /* USER CODE END PFP */

  <中略>

  /* USER CODE BEGIN 2 */
  /*
   * LED用PWM出力開始
   */
    HAL_TIM_Base_Start(&htim1); // TIM1スタート
    HAL_TIM_PWM_Start(&htim1, TIM_CHANNEL_1); // TIM1_CH1(LED赤) PWM出力開始
    HAL_TIM_PWM_Start(&htim1, TIM_CHANNEL_2); // TIM1_CH2(LED緑) PWM出力開始
    HAL_TIM_PWM_Start(&htim1, TIM_CHANNEL_3); // TIM1_CH3(LED青) PWM出力開始

    uint8_t count = 0 , red, green;
  /* USER CODE END 2 */

  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {
  /* USER CODE END WHILE */
  /* USER CODE BEGIN 3 */
    if( count < 10 ) red = 10*count; // LED(赤)の設定値
    else red = 10*(20-count);
    green = 100 - red; // LED(緑)の設定値

    LED(red, green, 0); // 設定値の反映

    HAL_Delay(100); // 100ms休む
    count = (count+1) % 20; // 変数countの更新
  }
  /* USER CODE END 3 */

}

 <中略>

/* USER CODE BEGIN 4 */
/*
 * RGB明度指定でLEDを点灯する
 * RGB明度は0~100
 */
void LED(uint8_t r, uint8_t g, uint8_t b)
{
  htim1.Instance->CCR1 = (uint32_t)r * 100;
```

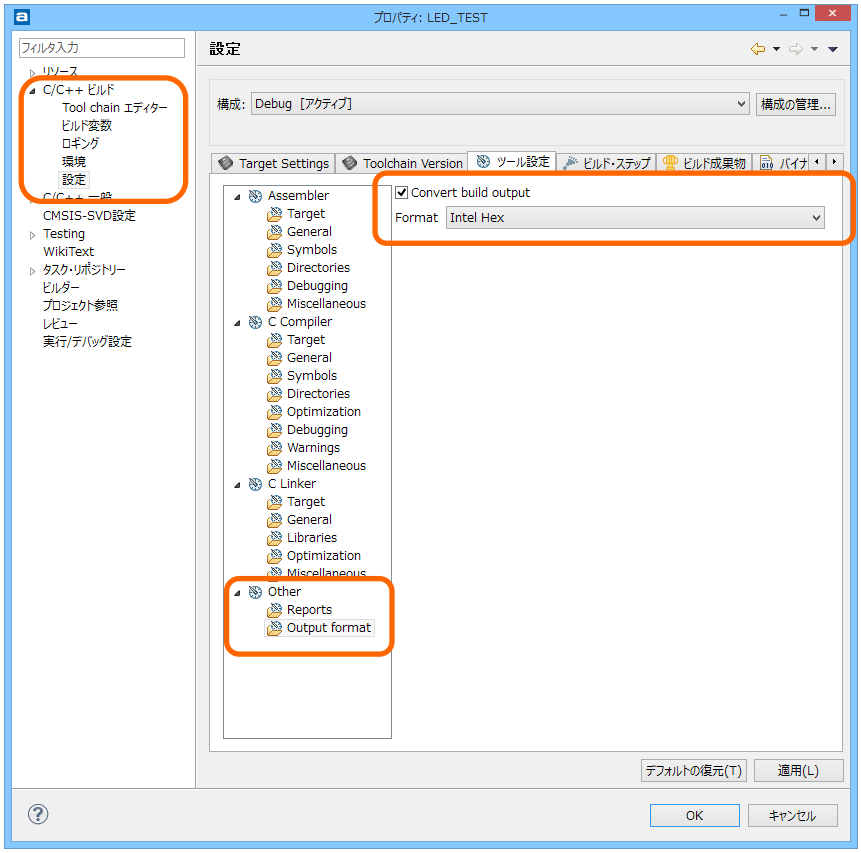
USER CODE BEGIN 4は
main.cの下の方にあります

```
  htim1.Instance->CCR2 = (uint32_t)g * 100;
  htim1.Instance->CCR3 = (uint32_t)b * 100;
}

/* USER CODE END 4 */
```

7．DFU形式ファイルをUSBポート経由で書き込みを行う場合は、プロジェクトの設定を変更する必要があります。ST-LINKを使用してプログラムの書き込み・デバッグを行う場合はこの手順を省略してかまいません。

8．DFU形式ファイルをUSBポート経由で書き込む場合は、標準の設定で生成されないHexファイルを生成する必要があります。まずTrueSTUDIOのプロジェクトメニューからプロパティを選択し、「C/C++ビルド」の左の三角マークをクリックしてビルド設定の項目を展開します。つぎに「設定」を選び、「Other」の「Output format」を選択します。「Convert build output」にチェック



を入れ、Formatは「Intel Hex」を選びます。設定が終わったらOKボタンを押してプロジェクトの変更を反映させます。

9．プロジェクトメニューのプロジェクトのビルドを選択し、プログラムのコンパイルを実行します。デバッグ用ELF形式ファイルと一緒にHEXファイルが生成されます。

ビルドに失敗した場合は、プログラムが間違っていないか確認・再編集をして再度ビルドしてください。

書き込みと実行

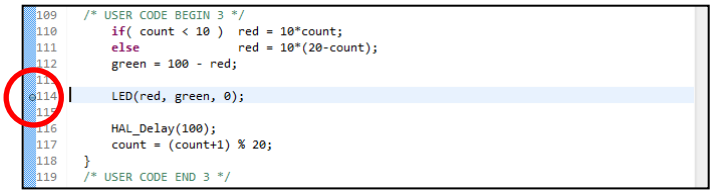
ST-LINKを使用しない場合（USBケーブル書き込み）

「DFU形式ファイルをUSBポート経由で書き込み」で行った手順でビルド後のプログラムをETCBボードへ転送して実行します。

ST-LINKを使用する場合

ST-LINKを使用する場合はデバッグを行うことができます。先ほど作成したプログラムにブレイクポイント(デバッグ時に一時停止する)を設定してみましょう。

1．LED(red, green, 0);が記述されている行を選択し、行番号の左側をダブルクリックします。行番号の左側にブレークポイントが設定されていることを示す○マークが表示されます。ブレークポイントを取り除く場合は再度ダブルクリックします。



2．ST-LINKとPCをUSBケーブルで接続し、ETCBボードとST-LINKを接続した状態で、ETCBボードの電源を入れます。EXTポートでのST-LINK差し込み場所を間違えないようにしてください。ST-LINK側とEXTポートのつなぎ方は次のようになります。

EXTポート番号	SWDIOポート名	ST-LINKコネクタ番号
14	GND	3
16	NRST	15
18	SWCLK	9
20	SWDIO	7
22	3.3V	1

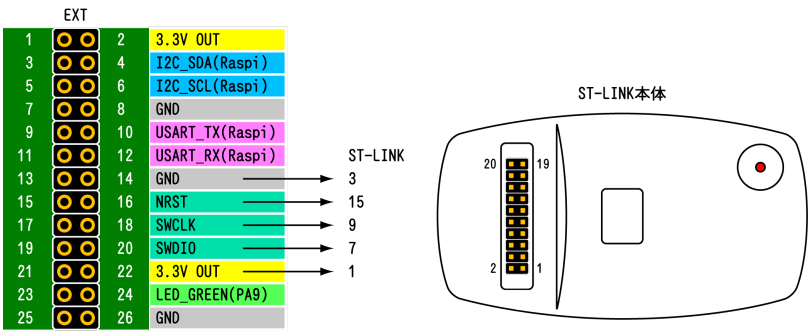
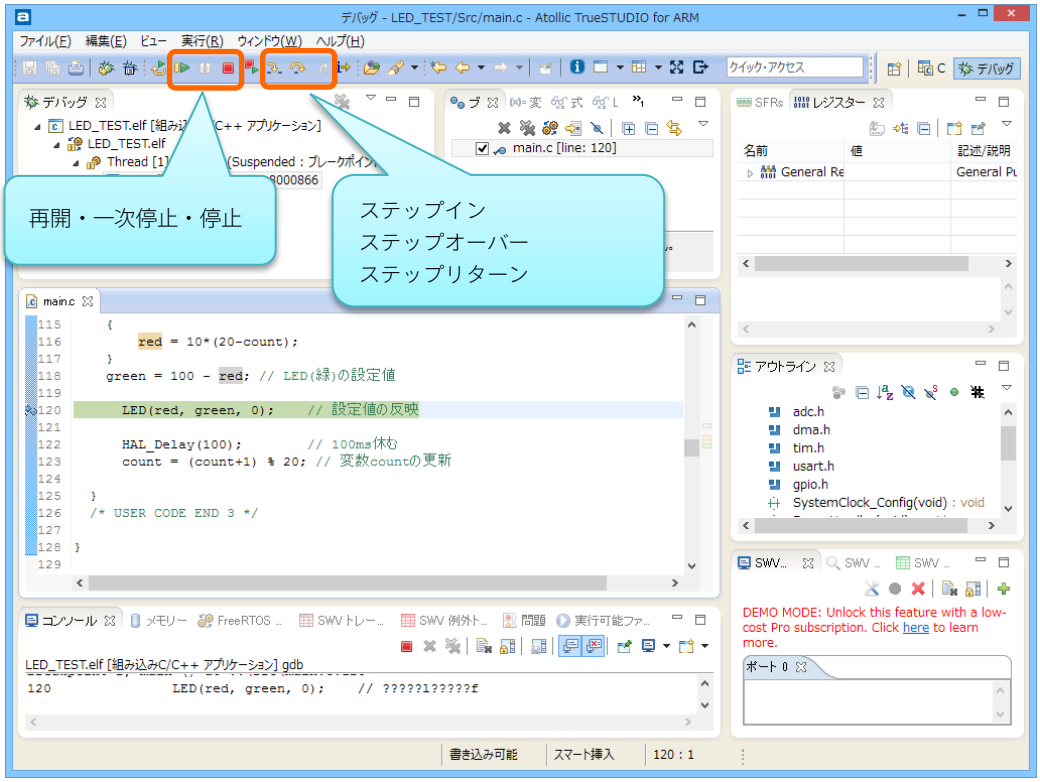


図 4　ST-LINKとの接続

3．実行メニューのデバッグを選択してデバッグモード（パースペクティブとよぶ）に移行します。この時ETCBボードのFlashメモリーへ先ほどビルドしたプルグラムが転送されます。

4．実行メニューの再開（F8）を選択します。プログラムの実行が始まり先ほど設定したブレークポイントの行で実行が一時停止します。
実行メニューでステップオーバー（F6）を選択します。一時停止していた処理（LEDの輝度調整）が実行されETCBボード上のLEDが点灯します。



5．プログラムを再開するたびにLEDの色が変化することが確認できます。またデバッグ画面では変数の値を確認・修正することもできます。

6．デバッグを終了するときは、実行メニューの終了を選択します。

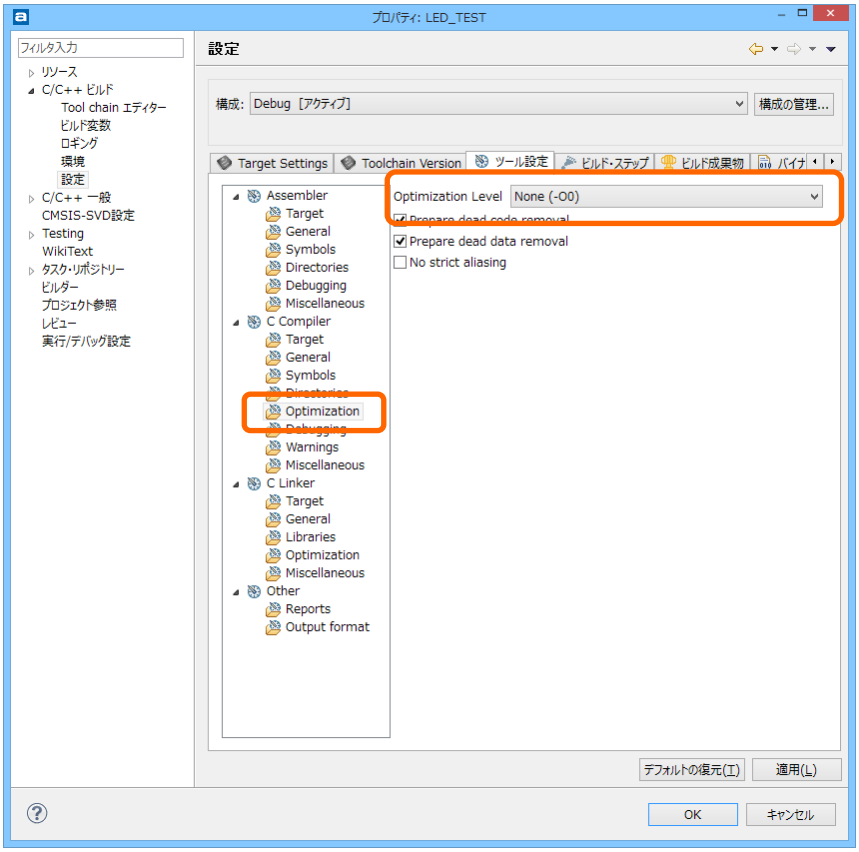
デバッグ命令

再生（再開）	次のブレークポイントまで処理を進めます。次のブレークポイントに処理が移らない場合（ブレークポイントが無い・ブレークポイントを入れた割り込みが発生しなかったなど）は、プログラムは最後まで実行されます。
一次停止	処理を一時的に停止します。一次停止ボタンを押したタイミングで、停止位置が変わります。
停止	プログラムを停止します。

ステップイン	処理行（ブレーク中の位置）が関数だった場合にステップインを実行すると、その関数の中に処理が移ります。
ステップオーバー	次の命令まで処理を進めます。処理行が関数だった場合は、関数を実行してから次の処理行まで進みます。
ステップリターン	処理行が関数の中である場合にステップリターンを実行すると、その関数の処理を全て実行してから関数の呼び出し位置へ処理が移ります。
式ビュー	デバッグパースペクティブのときに「式ビュー」を開いておき、モニタリングしたい変数などを記載しておくと、ブレークポイントで一次停止する度に、そのときの変数の中身を更新・表示します。 デフォルトでは式ビューは開いていますが、ウィンドウメニューのビューから開くこともできます。

デバッグ中にブレークポイントで止まらない場合

プログラムの最適化がセットされている場合、デバッグ中にブレークポイントで止まらないで、その前後の行でプログラムが停止することがあります。その場合はTrueSTUDIOのプロジェクトメニューからプロパティを選択し、下図のように「C/C++ビルド」欄の「設定」を開き、「ツール設定」タブか



ら「C Compiler」＞「Optimization」を開き「Optimization Level」を「None（-O0）」にしてください。

Raspberry Piへの応用

準備

本マニュアルではETCBとRaspberry Pi間のUART通信プログラムの作成について説明しますが、Raspberry Pi側のUART通信プログラムについては扱わないで、代わりにETCBのUSB1端子を使うことにします。

通常ETCBとRaspberry PiでUART通信するためには、EXTポートのUART1_TX/RX端子をジャンパで接続します。同様にETCBのUSB1ポートもMCUのUART1と接続されていますので、USB1ポートから命令を受け取るプログラムを作成すると、そのままRaspberry Piから命令を受け取るプログラムとなります。そこで、本マニュアルでは、USB1（UART1）から命令を受け取り、サーボモーターを動かすプログラムを作成します。また、プログラム作成環境の整備や、ETCBの機能の標準的な使い方なども説明します。

Raspberry Pi側のプログラムについては別紙「ETCB Raspberry Piプログラミングマニュアル」を参照してください。

必要なソフトウェア・デバッガ

- プログラムの作成にはSTM32CubeMX、Atollic TrueSTUDIOが必要です。あらかじめWindows PCへインストールしておきます。できるだけ最新版をご用意ください。またSTM32CubeMXでF3ファミリーのファームウェアもインストールしておいてください。
- USB1ポートからも書き込みはできますが、デバッグできないので、できるだけST-LINK/v2を使うようにしてください。USB1ポートから書き込みするときには、DFU端子にピンヘッダを取り付けるか（半田づけ作業が必要です）、端子をショート出来るものが必要です。
- USB1ポートからサーボモーターのバイナリデータ命令を送るために、CoolTermというターミナルソフトウェアを使います。下記サイトからダウンロード・インストールしてください（ダウンロードしたZIPファイルを適当なフォルダで解凍して、coolterm.exeを実行してください）。
  http://freeware.the-meiers.org/

Raspberry Piとハードウェア接続（UART）

始めにRaspberry PiとETCBを接続する準備をします。本マニュアルでは実際に行う必要はありません。

1．半田付け
付属の2.54ピッチピンソケットをEXTポート横に半田付けしてください。ピンソケットを半田付けするときにソケットと基板に隙間ができないように注意してください。付属のΦ2.6L11樹脂スペーサとネジを使って、あらかじめRaspberry Piに取り付けた状態で半田付けすると失敗が少なくなります。

2．ジャンパピンのとりつけ
図３　EXTポート・端子名称を参考にして、UART_RXとUART_TX(Raspi)を付属のジャンパピンで短絡（ショート）します。同様にUART_TXとUART_RX(Raspi)を短絡します。

3．Raspberry Piにとりつけ
Raspberry PiのGPIOピンとずれないように取り付けてください。取り付け時はETCBとRaspberry Piとの間に必ずΦ2.6L11をはさみこんでください。取り付けたらネジ止めしてください。

STM32CubeMXでETCBプログラムひな形を作る

ETCBはUART、I2C、AD変換、GPIOなどの機能があらかじめハードウェアで決まっていますが、プログラム上で機能の実装を行うと間違えることがあります。このようなPinout設定（端子機能の設定）が簡単に設定できるSTMicroelectronics社のSTM32CubeMXソフトウェアを使用します。STM32CubeMXは同社の提供しているHALライブラリを使って、MCUのPinout設定をGUIメニューで作って、プログラムのひな形を生成することができます。

ここではSTM32CubeMXを使ってETCBの周辺回路（ペリフェラル）設定について、ETCB標準設定例を使って説明します。実際はユーザーが使用する周辺回路だけを設定してかまいません。

使用するサーボモーターは近藤科学社製のICSシリアルサーボモーターです。

プログラムひな形の内容

本例題（ETCB_RobotBasic）ではRaspberry Piからサーボの位置命令を受け取り、ETCBへ取り付けられたサーボモーターへ周期的にコマンドを送信します。その他、AD変換を自動的に行います。また、LEDをPWMで点灯します。また、できるだけDMA転送を使って、MCUへの負担を軽減します

使用するサーボモーターは近藤科学社製のシリアルサーボモーターとします。近藤科学社製のシリアルサーボモーターはICSコマンドで動作しますので、このICSコマンドをサーボモーターに定期的に送ります。Futaba社製シリアルサーボを使用するときは、後述の`IcsSetPos`関数の中身を適当に書き換えてください。

機能	ポート	内容
AD変換	AD1～AD4ポート MCU温度 バッテリー電圧監視	■AD変換は自動的に繰り返し行うように設定 ■変換後の値はDMA転送を使って指定変数へ自動保存する ■MCU温度とバッテリーの電圧監視
UART1	UART非同期通信でRaspberry Piと通信する ※Raspberry Pi側のプログラムはここでは扱いません	■通信プロトコルは、通信速度115200bps、データ長8ビット、1ストップビット、パリティ無し ■Raspberry Piからの命令をUART1受信割り込みで受信 ■Raspberry Piから命令が3ms来なかったら（タイムアウト）、その時点まで受け取ったデータを１命令と判断 ■タイムアウト時間はSYSTICKタイマーで判断 ■命令に対する返事はDMA送信するように設定

		■UART1で受け取ったデータを解析して、目標位置をics_servo構造体のuint16_t dpos変数に保存
SIO	UART2、UART3 サーボモーターの駆動	■uint16_t dposからサーボモーターへの命令（ICSコマンド）を作って、SIOポートから周期的に送信 ■サーボ命令送信周期はTIM2タイマーを使用
LED	PWM(TIM1)を使ったフルカラーLEDの点灯	■PWMを使って、LEDを点灯 ■点灯(PWM)周期は10ms ■PWMカウンタ1カウント当たりのPWM出力時間は0.1msとすると100カウントで100%となりますので、フルカラーLEDのRGBはそれぞれ100段階で設定可能となります。 ■ETCBではPWM端子の極性が逆[3]なので、CH PoralityをLowに設定
ST-LINK	ST-LINKデバッグ対応	■ST-LINKを使ったデバッグができるように設定
SYSTICK	SYSTICK割り込み	■SYSTICKをメニューから選ぶ（STM32CubeMXが自動的に1ms間隔のSYSTICKタイマーを生成）
TIM2	サーボモーターへの周期的命令送信	■TIM2の周期を20msに設定 ■TIM2の周期割り込みが発生したら全サーボへ位置コマンドを送る（20msごとにサーボモーターへコマンド送信）
RCC	クロック周波数	■72MHz設定

STM32CubeMXの起動とプロジェクトの新規作成

STM32CubeMXを起動し、初期画面またはFileメニューから「New Project」（プロジェクトの新規作成）を選択してください。New Projectダイアログが表示されますので、ETCBのMCUに合わせて、各メニューを下記のように選んでください。または例題からプロジェクトを読み込んでもかまいません。

- Series　STM32F
- Lines　STM32F302
- Package　LQFP48

[3] 端子の電圧レベルをHIGHにすると、LEDが消える

- MCUs List　STM32F302C8Tx

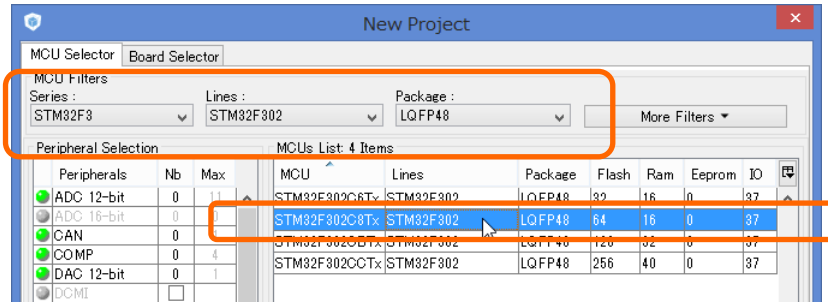


図 5　MCU選択

選択が完了したら、OKボタンを押してください。そうすると電源などの必要最低限のピン設定が完了した画面（図６　MCU初期状態）が表示されます。なお同図でPinoutメニューおよびPinout設定画面の名称は正式名称ではなく、分かりやすく言い換えたものです。正式にはPinoutメニューがPeripheral and Middleware treeで、Pinout設定画面はChip viewと言います。

「Pinout」、「Clock Configuration」、「Configuration」、「Power Consumption Calculator」という４つのタブがありますが、最初はPinoutを選択してください。

Pinout設定（端子設定）

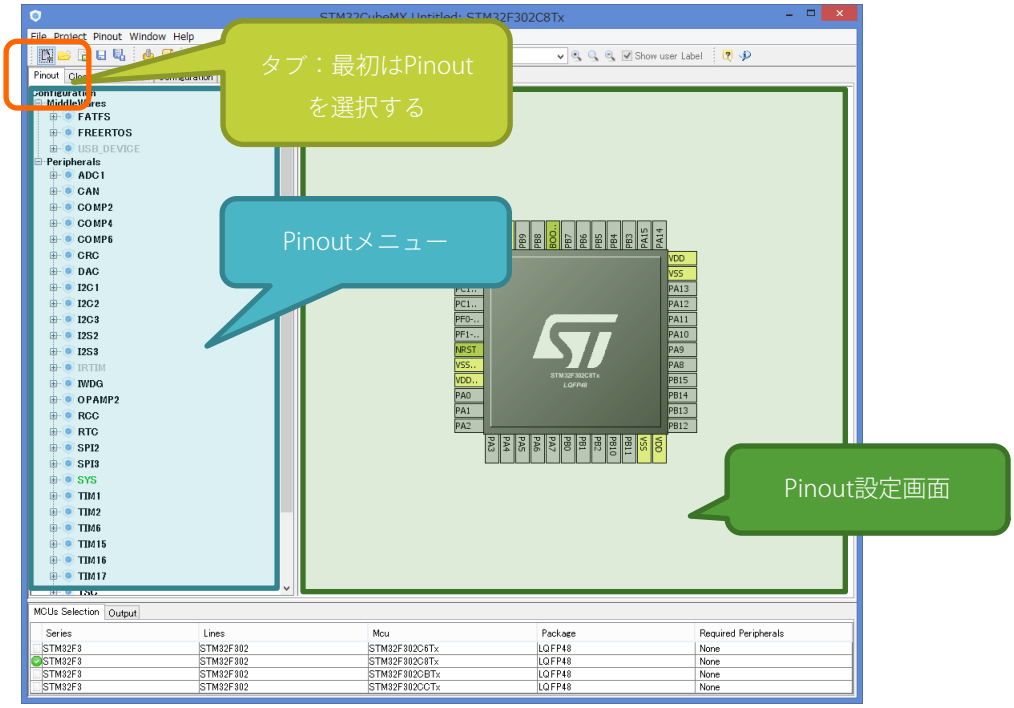


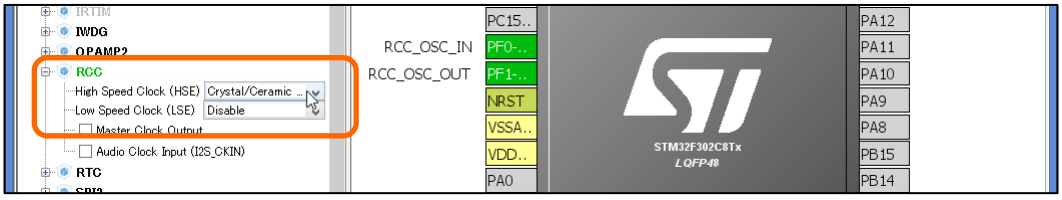
図 6　MCU初期状態

Pinout設定画面はマウスホイールで画像の拡大・縮小ができます。またマウスでMCUの位置をドラッグできます。

RCC設定

RCC設定ではMCUのメインクロックのソースを決定します。

ETCBは8MHzのセラロックが実装されていますので、クロックソースをHSE＝Crystal/Ceramic Resonatorにします。画面左側のPinoutメニューからRCCを選択し、High Speed Clock（HSE）を「Crystal/Ceramic Resonator」にします。RCC設定は必ずしも他の周辺回路設定より先にする必要はありませんが、後から行う「Clock Configuration」の前に設定しておきます。

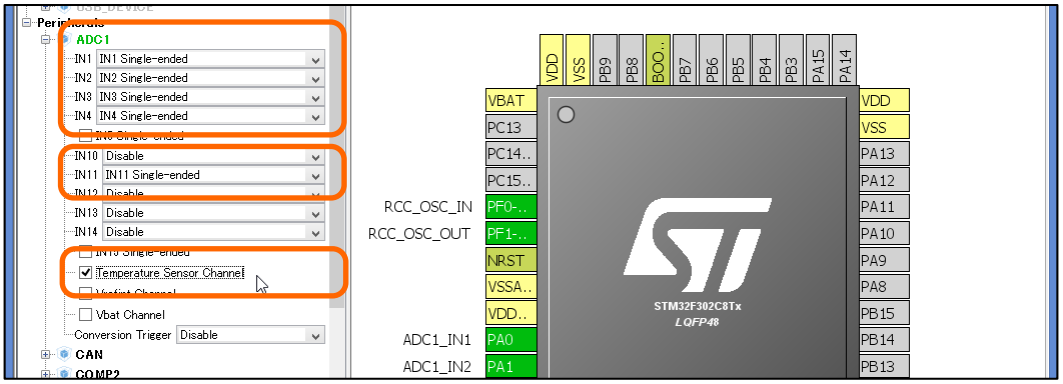


AD変換設定

例題では、main関数などで何もしなくても勝手にAD変換をして、指定した変数へAD値を保存しておくように設定します。具体的には、AD変換をしたあとに自動的にDMA転送[4]を行い、DMA転送が完了すると割り込みが発生するようにセットします。DMA転送完了割り込み関数内では、保存したAD値を実際の値（AD0ならAD値を電圧値に変換）します。

周辺回路（端子機能）をMCUの端子にセットするには、Pinout設定画面で直接端子を指定するか、PinoutメニューのPeripheralsにある「ADC1」欄から、AD1～AD4を有効にします。ETCBはAD変換ポートが４つあり、AD1～AD4ポートはそれぞれPA0～PA3に対応しています。PB0は電源入力端子（バッテリー入力）とつながっているので、PB0端子も使えるようにします。また、MCU内部には温度計が備わっているので、温度計の出力もAD変換して数値化（計測）できるようにします。

PinoutメニューのADC1の各ポートは全て「INx_Single-ended」にしてください。Single-endとはAD変換器１つを１個ずつ変換するという設定です。



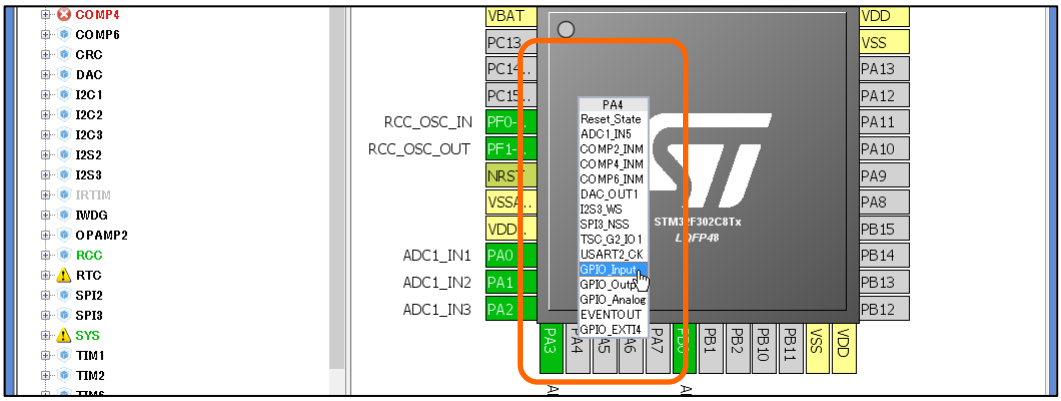
一緒に「Temperature Sensor Channel」にチェックを入れて、温度データもAD変換するように設定します。

GPIO設定

ETCBのMCUは、PA4～PA7がそれぞれGPIO1～GPIO4に、PB1とPB2がそれぞれGPIO5とGPIO6にハードウェア接続されています。使用する場合はPinout設定画面でPA4～PA7とPB1～PB2をGPIO_OutputまたはGPIO_Inputに設定します。

---

[4] Direct Memory Accessのことで、CPUの演算回路を通さずに直接メモリー間でデータ転送をする方法のこと。CPUリソースを使わないでデータの保存ができます。

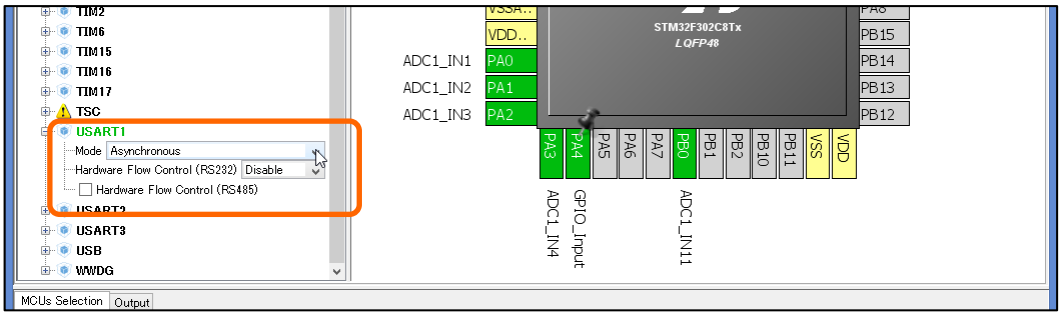


GPIOのInput/OutputはPinoutメニューで設定する項目はありません。

UART1設定

UART1はEXTポートのUARTおよびCOM1（USB micro-Bコネクタ）に接続されています。今回はRaspberry Piから命令を受け取るため、UART1を双方向通信可能にセットして、データを受信したときに受信割り込みが発生するように設定します。返事（UART TX）はDMA転送で行います。DMA転送ではメモリーにデータを書き込むだけで、演算回路を介さずにUARTからデータが出力されます。

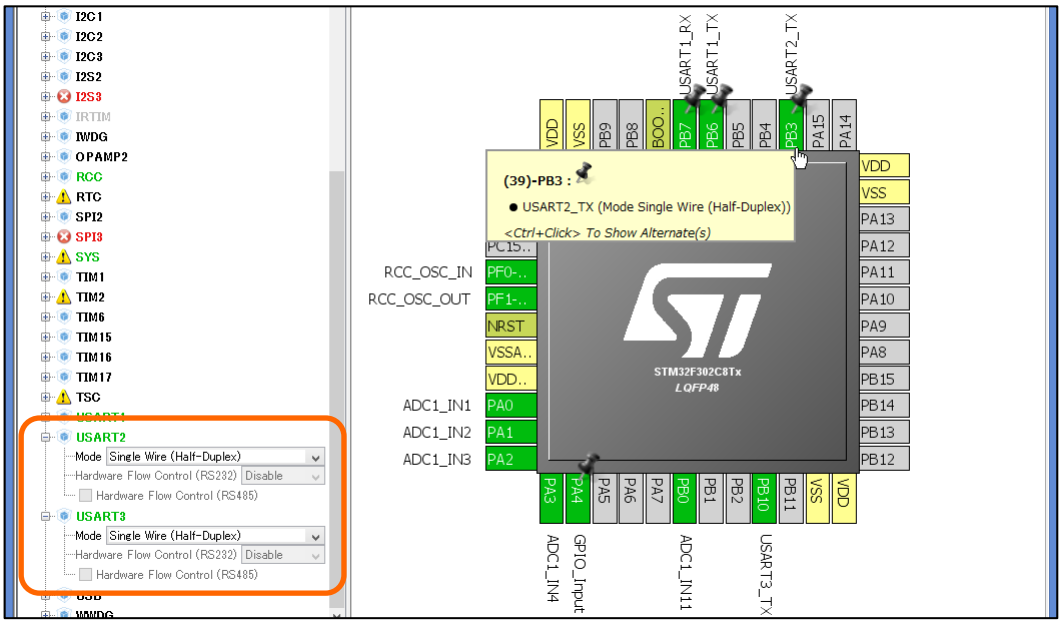
PinoutメニューのUART1を開き、Modeを「Asynchronous」に設定します。Pinout設定画面ではPA9とPA10が自動的にUART1_TXとUART1_RXにセットされますが、ETCBではUART1をPB6とPB7にハード接続していますので、あらためてPB6をUART1_TXにして、PB7をUART1_RXに設定して下さい。



UART2、UART3設定

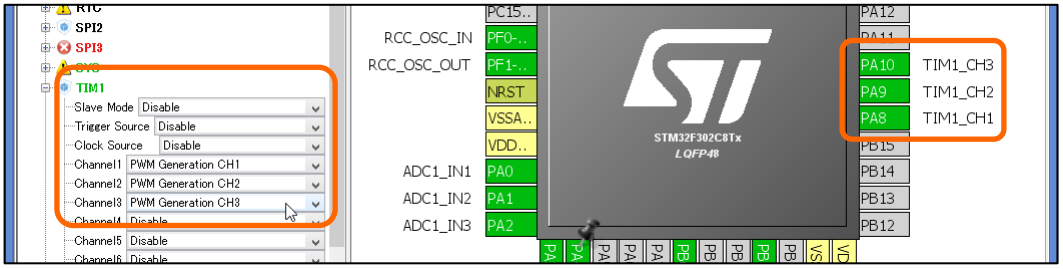
UART2、UART3はそれぞれS7～12とS1～6にハードウェア接続されています。シリアルサーボを動かす場合はUART2とUART3を「Single Wire（Half Duplex）」設定にして、TXポートだけで送受信できるように設定します（送信と受信はプログラム内で切り替えます）。

Pinoutメニューを下図のように設定します。Pinout設定画面での端子設定は自動設定のままにしておきます。



PWM（フルカラーLED）設定

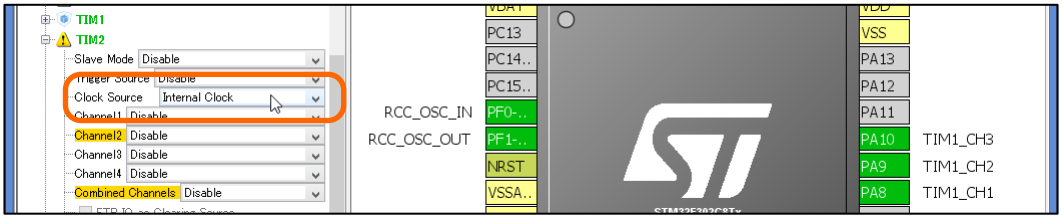
ETCBのフルカラーLEDは赤・青・緑それぞれにPWM（TIM1出力）端子がハードウェア接続されています。PWMのデューティ比を調節することでフルカラー表示ができます。赤・青・緑それぞれはTIM1_CH1、TIM1_CH2、TIM1_CH3端子に接続されていますので、PinoutメニューのTIM1を開き、Channel1～Channel3をそれぞれPWM Generation CH1~3に設定します。PWM出力設定にするとPinout設定画面では自動的にPA8～PA10がTIM1出力設定になります。Clock Sourceの指定は不要です。



タイマー（TIM2）設定

TIM2を使って周期的に割り込みを発生させて、サーボモーターへ命令を送信します。TIM2だけ32ビットタイマーで、残りは16ビットタイマーですので注意してください。

PinoutメニューからTIM2を開き、Clock Source（割り込み発生源）を「Internal Clock」にします。

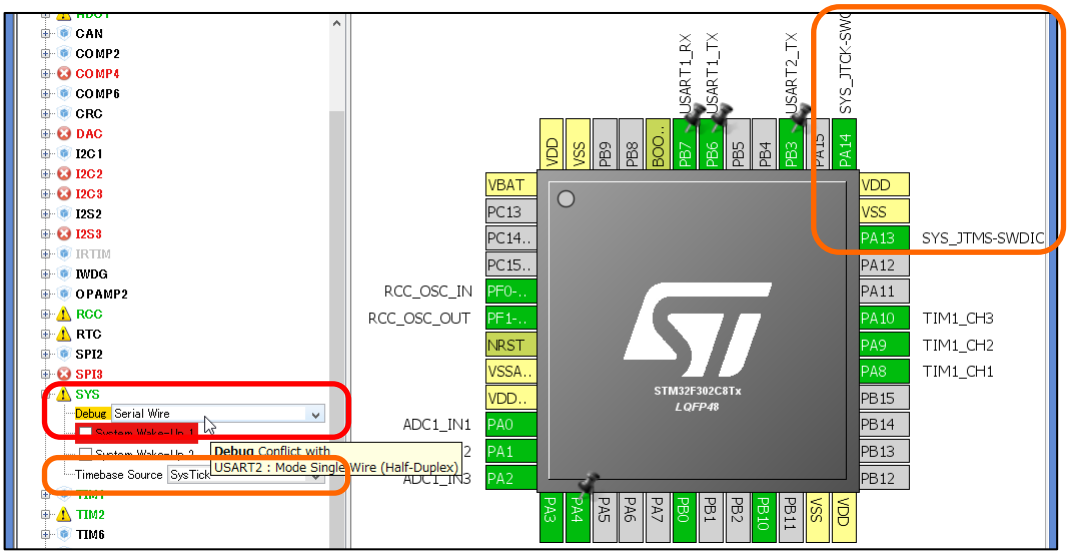


デバッグ設定

最後にST-LINKを使ったデバッグができるように端子設定をします。ST-LINKでデバッグしない場合やSWDIOとSWCLKラインをGPIOで使用したいという場合は不要です。

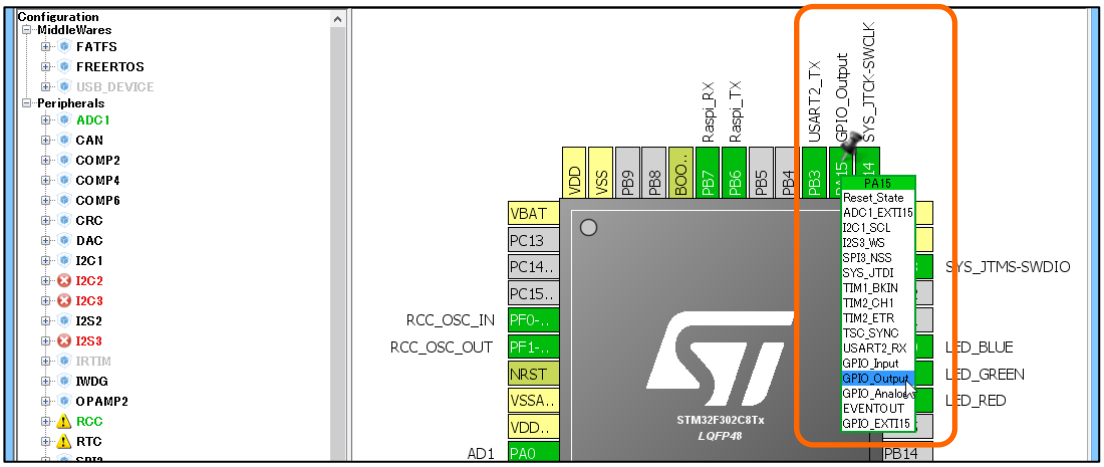
ST-LINKはPinoutメニューのSYSを開き、Debugメニューから「Serial Wire」を選択します。自動的にPA13とPA14がSWDIOとSWCLKにセットされます。

同時にSysTickタイマーを使用できるように、Timebase Sourceメニューを「SysTick」にセットします。SysTickタイマーとは、クロック周波数をMCUが数えて、簡単にタイマーとして使う仕組みです。初期状態では1msタイマーが自動的に設定され、`void HAL_SYSTICK_Callback ()`という割り込みが発生します。SysTickタイマーはその他の割り込みなどに影響され、あまり正確ではありませんので、制度を要求されないタイマーとしてお使いください。



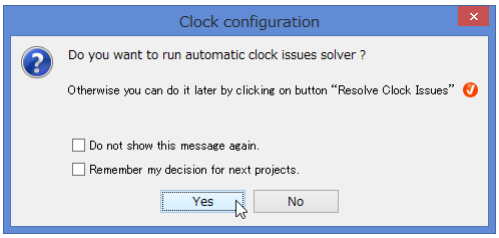
サーボON/OFFスイッチGPIO設定

ETCBはサーボ電源をカットすることができます。回路ではPA15端子をGPIO Outputに設定することでPA15をLow（0V）にすると電源OFF、High（3.3V）にすると電源がONになります。
プログラム上でPA15端子からHighを出力しなくてもデフォルトでサーボの電源は入りますが、PA15端子をGPIO Outputに設定しないと端子電圧が不定となり電源が入りません。Pinout設定画面で直接端子をクリックし、メニューからGPIO Outputに設定してください。



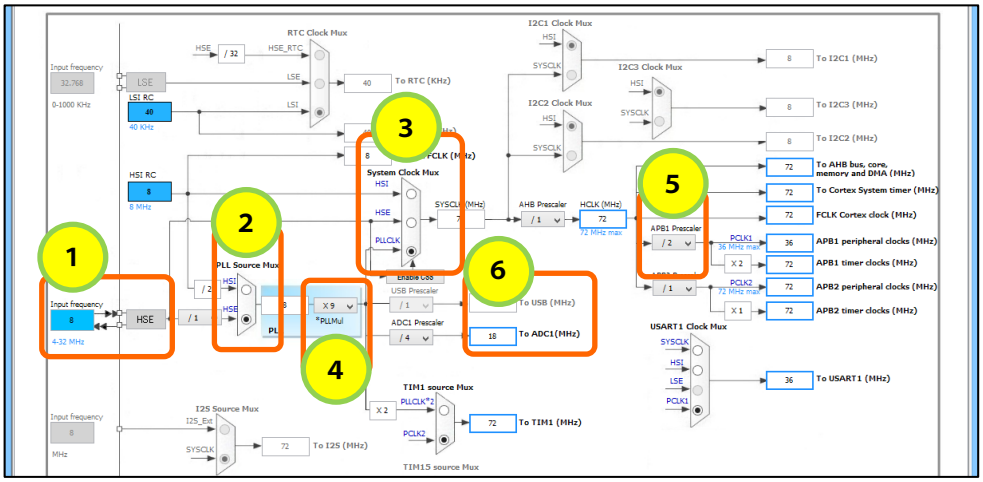
クロック設定

Pinoutの設定が済んだら、MCUのクロックを「Clock Configuration」タブで設定します。最大72MHzまで設定できます。クロックを初めて設定するときは、クロックの自動設定を行うかどうかを答えるダイアログが表示されますので、Yesボタンを押して先に進みます。



クロック設定画面が表示されますので、以下の項目通りに画面上のメニューを選んでください。一番左端にある「Input frequency」のすぐ横に「HSE」と表示されていないときは、Pinoutタブに戻り、RCCの設定をやり直してください。

いずれかのメニューを選んだときに、他の項目が赤く表示された場合は、その選択は動作しない設定ですので、選択をやり直してください（例えばPLLMulをx10以上にした場合など）。



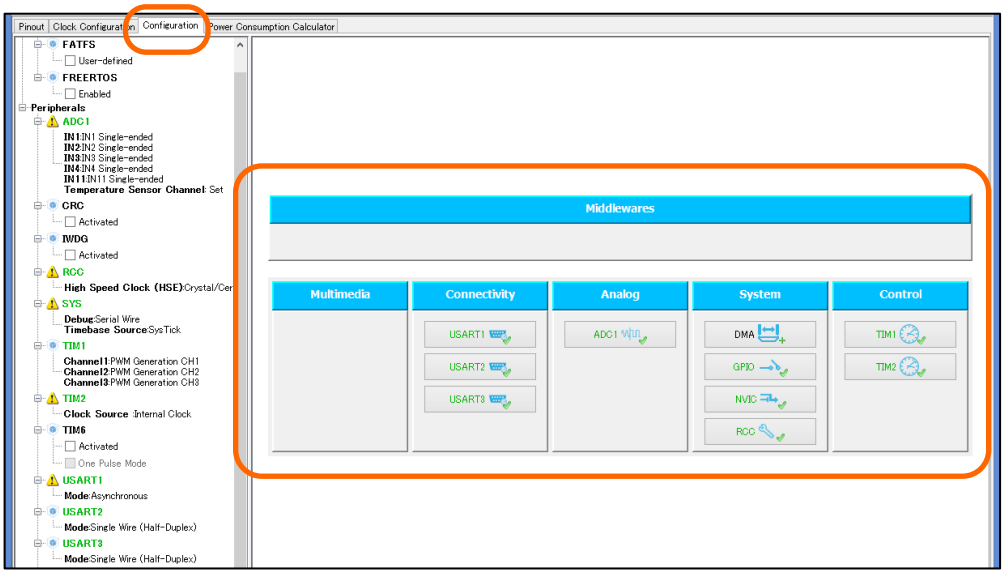
①	Input frequency	HSEとなっていることを確認。そうで無い場合はRCCを再設定
②	PLL Source Mux	HSEを選択（●マークが入る）
③	System Clock Mux	PLLCLKを選択（●マークが入る）
④	PLLMul	x9（8[MHz] x 9=72MHzにセットされます）
⑤	APB1 Prescaler	APB1クロックはメインクロックの半分で動作するため、「/ 2」を選択します。
⑥	ADC1 Prescaler	AD変換は最長で601.5Cycle（601.5 / 72000000 = 8.5[us]）が選択できます。ADC1の変換速度が速すぎると、変換後の割り込み発生頻度が高くなり他の処理ができなくなる事があります。ETCBは最大72MHzで動作しますので、Prescalerで変換時間を1/4くらいにしておいた方が良いでしょう。

最終的なクロックが72MHzとなるようにしてください。クロックを低く設定すると消費電力が下がりますが、タイマー設定も変わりますので注意してください。

Configuration

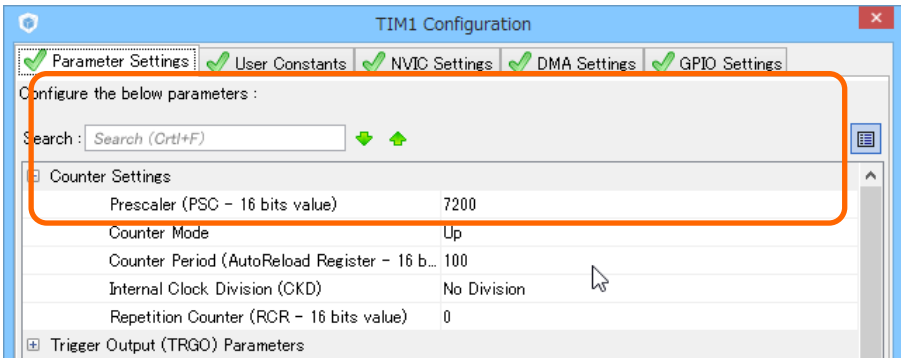
ConfigurationタブではPinout設定画面で設定した各周辺回路の詳細な設定や、割り込みなどの設定を行います。設定の簡単なものから説明します。

最初にConfigurationタブを選択すると、Multimedia、Connectivity、Analog、System、Controlの５つのカテゴリが表示され、それぞれに周辺回路名が表示されたボタンがあります。設定したい機能のボタンを押すと設定ダイアログが表示されます。



TIM1（フルカラーLED）周期設定

まずLED点灯用にPWM信号をTIM1で周期的に発生させます。ここではPWM周期（TIM1の周期）を10ms、分解能（LEDのRGB光の強弱）を100段階で調整できるようにします。まずTIM1の周期はメインクロックをPrescalerで割った数値となりますので、10msの周期を100段階で調整するには、タイマーの1カウントが0.1ms（0.1ms×100段階=10ms）にする必要があります。

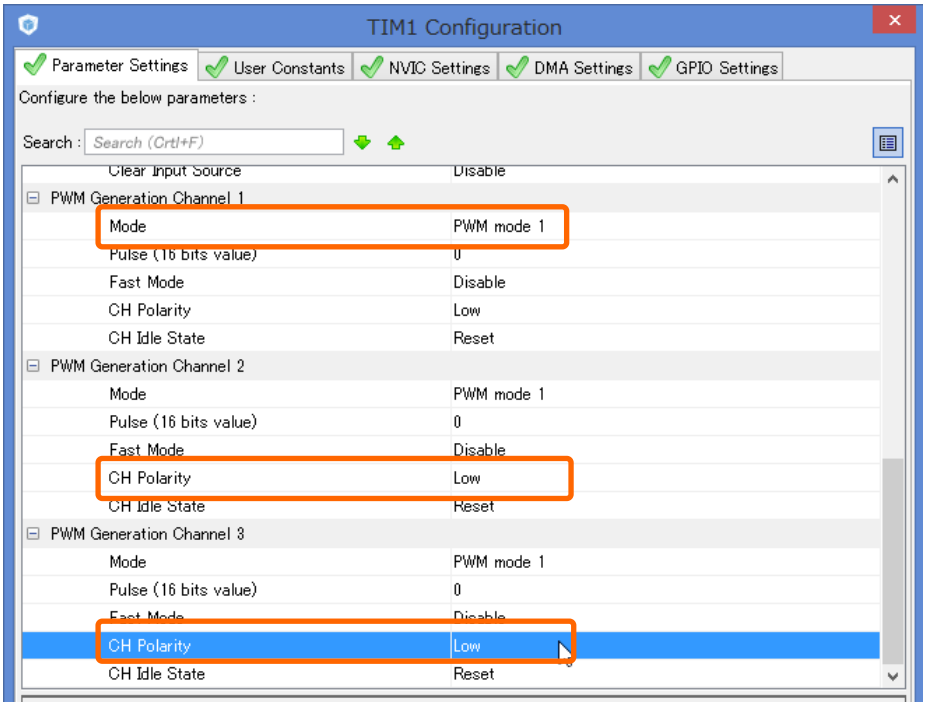


今回はMCUのクロックは72MHzですので、Prescalerが1の場合は1カウント当たりの時間は1/72,000,000秒となります。これを1カウントあたり0.1ms（10msの1/100）にするにはPrescalerを7200とすると、7,200/72,000,000秒=0.0001秒＝0.1msとなります。

次にCounter Period（周期カウント）を100にして、PWM周期を0.1ms×100=10msとします。

実際のプログラムではRepetition Counterの値をセットしてデューティ（RGB色の明るさ）を調整します（後述）が、ここでは0にしておきます（PWM信号を出力しない）。

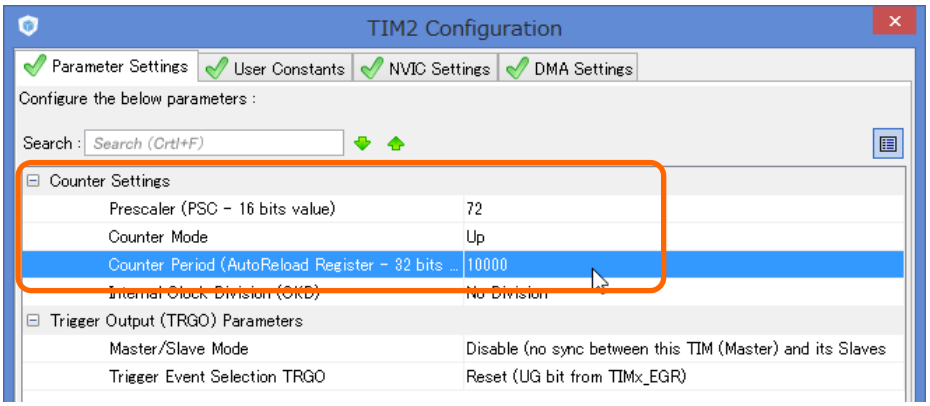
ETCBのLEDは端子を0V（Low）で点灯しHighで消灯するように、極性が逆になっています。そこでPWM Generation Channel（1～3）の全てのCH Polarityを「Low」にして、PWM出力がONのときはLow電圧（0V）を出力し、OFFのときはHigh電圧（3.3V）を出力させます。デューティ比を大きくするとLEDが明るく点灯します。タイマーをPWM出力に設定すると周期的に割り込みを発生させなくても自動的に信号が出力されます。極性を逆にしないとデューティを大きくするとLEDが暗く点灯します。

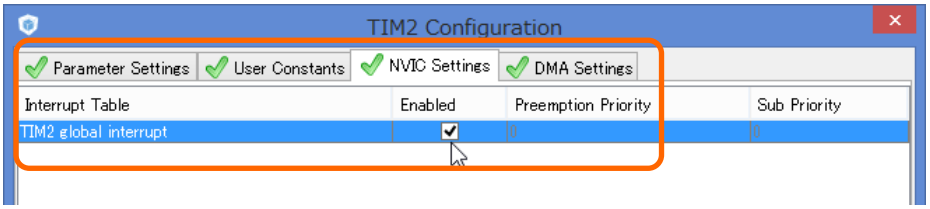


TIM2

TIM2タイマーはサーボに適当な周期で命令を出すために使用します。今回は周期を20msに設定しますが、あとからタイマーの周期を細かく設定できるように、Prescalerを72、Counter Periodを20000にセットします（周期20ms、分解能0.001ms）。

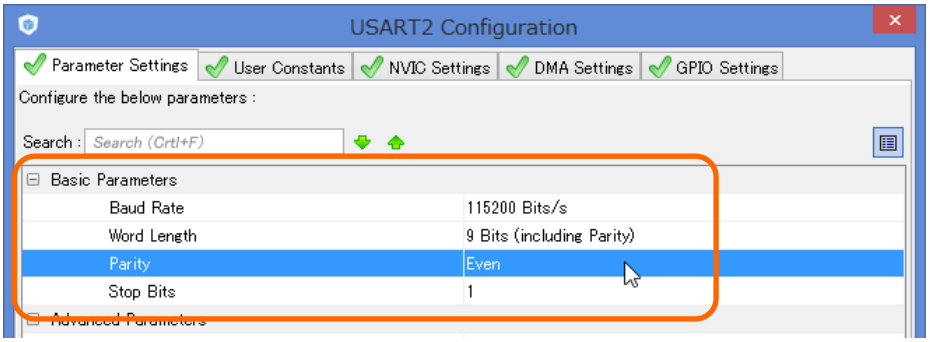
TIM2タイマーで20ms毎に割り込みを発生させるために、「NVIC Settings」タブを開き、TIM2 Global Interruptにチェックマークを入れます。





UART2とUART3の通信プロトコル設定

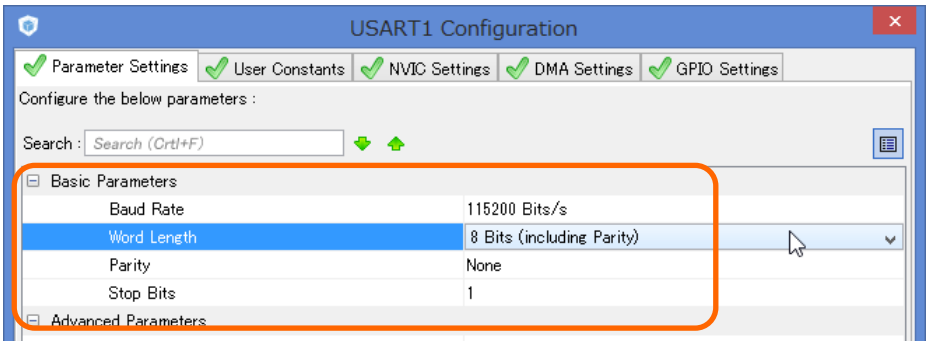
UART2とUART3を使ってサーボモーターへ命令を送ります。今回は近藤科学社製のICSシリアルサーボモーターが動くように設定しますので、通信速度115200bps・データ長9ビット・パリティEVEN（偶数）・1ストップビットにセットします。データ長はパリティビットを含みますので、偶数・奇数パリティを使う場合はデータ長8ビットに1ビット追加してください。



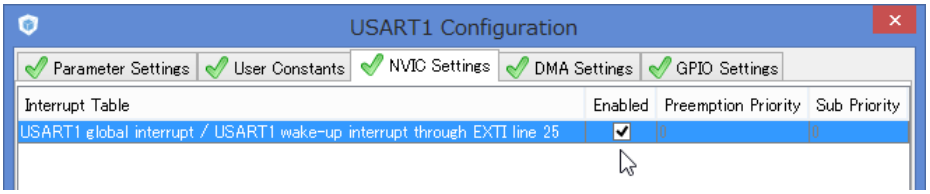
UART1（RX割り込み・TX DMA転送）

UART1はRaspberry Piからの通信を受信したときに割り込みを発生させて、コマンド受信完了後、結果を返すようにします。返事をするときはDMA転送を使って、できるだけMCU演算回路を使わない制御を行います。本例題では設定のみで実際は返事をしません。

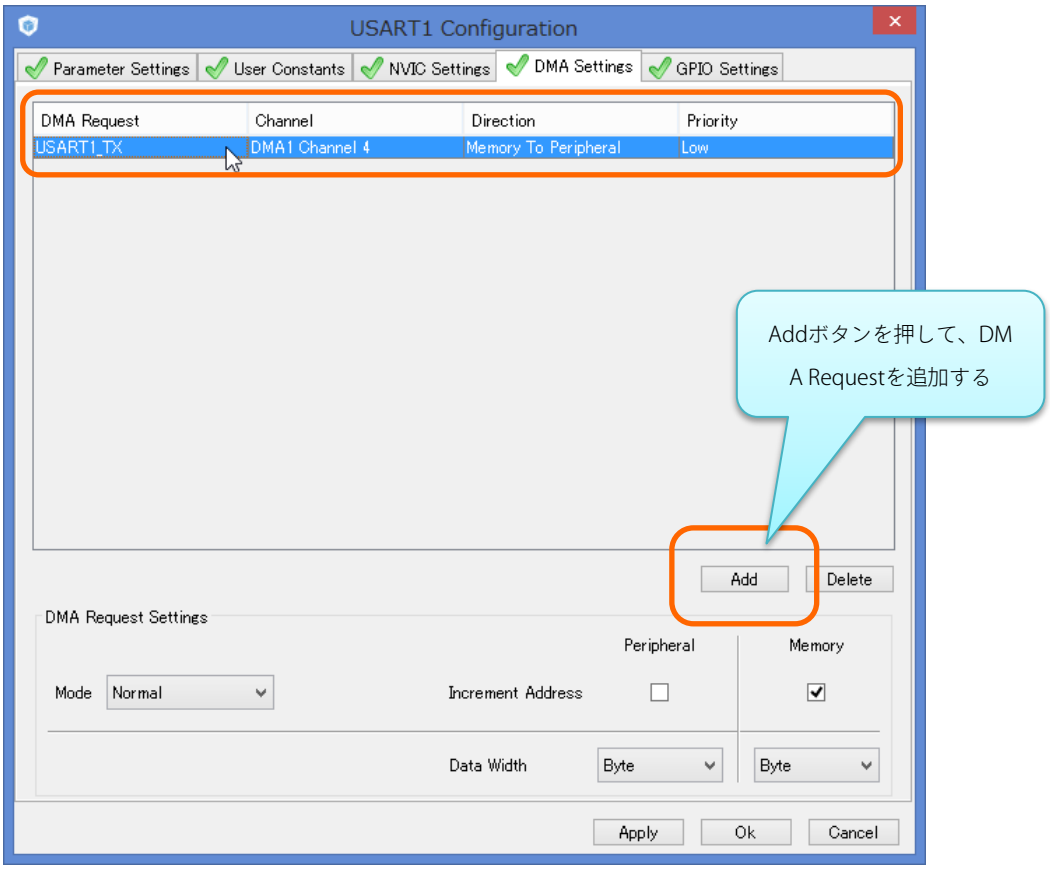
ConfigurationタブでConnectivityカテゴリのUSART1ボタンを押すと、下記のようにUSART1 Configurationダイアログが表示されますので、UART2、UART3同様に通信プロトコルを設定します。通信速度115200bps・データ長8ビット・パリティNone（なし）・ストップビット1にセットします。



つぎにUSART1 Configurationダイアログの NVIC Settingsタブを開き、USART1 global interruptの「Enabled」にチェックマークを入れ、受信割り込みを有効にします。



つぎにDMA Settingsタブを開き、右下にある「Add」ボタンを押します。画面のDMA Request欄にメ



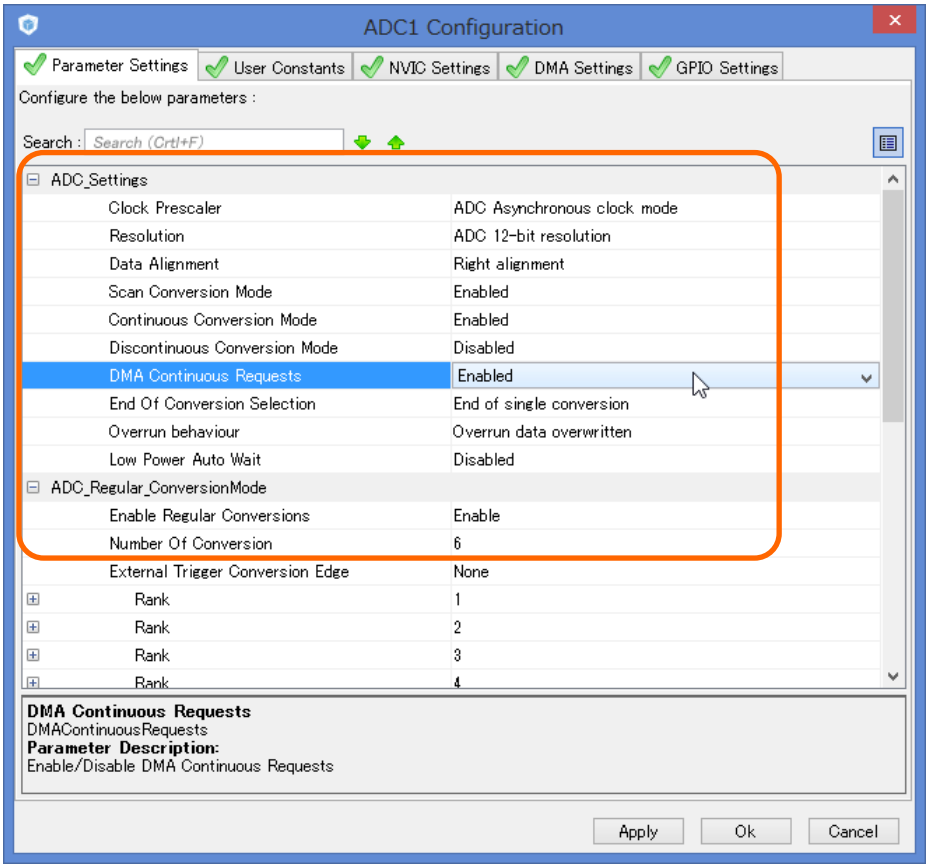
ニューが１列追加されますので、「Select」欄から「USART1_TX」を選択します。DMAチャンネル（DMA1 Chennel4）やDirection（転送方向・Memory To Peripheral）は自動設定のままで問題ありません。MCUが忙しくないときに転送すれば良いので、Priorityは「Low」のままにしておきます。また、ダイアログの下にある「Data Width」も、Peripheral＝「Byte」、Memory＝「Byte」（バイトデータをバイトデータのまま送信）にしておきます。

これで、指定する変数にデータを書き込みDMA転送を登録するだけでUART1よりデータが自動的に送信されます。

AD変換設定（変換後DMA転送＋DMA転送完了割り込み）

AD変換は各チャンネルのAD変換終了後に、変換データ（12ビットAD値）をメモリーへDMA転送します。全てのチャンネルがDMA転送完了したら転送完了割り込みを発生させて、変換データを割り込み関数の中で実際の値（電圧、温度、バッテリ電圧）に変更します。

ConfigurationタブのAnalogカテゴリにあるADC1ボタンを押します。つぎにADC_Regular_Conversion Mode欄のNumber Of Conversions（AD変換するポートの数）を6（AD1～4、電源、温度で合計6個）にします。



Number Of Conversions の数だけRankができあがりますので、下記のように設定してください。

- Rank1～4　Rank1=Channel1、Rank2＝Channel2、Rank3＝Channel3、Rank4=Channel4
- Rank5　Channel11（電源）
- Rank6　Channel Temperature Sensor
- Sampling Time　61.5Cycle～601.5Cycle[5]（全てのRank）

ADC_Settings欄は下記のように設定してください。

- Scan Conversion Mode　Enabled（自動変換・ソフトウェアトリガー[6]ではない）
- Continuous Conversion Mode　Enabled（連続変換）
- End Of Conversion Selection　End of sequence conversion（チャンネル毎に変換）

上記設定を行うと、MCUが自動的にAD変換[7]を行い、変換後のデータを指定された変数へ自動的に転送（データの更新）しますので、ユーザーは指定した変数を読み取るだけで良いことになります。

つぎにDMA転送設定を行います。DMA転送タブを開き、「Add」ボタンを押してDMA Request項目を追加します。Select欄をADC1にセットします。ChannelとDirectionとPriorityは初期状態のままで問題ありません。

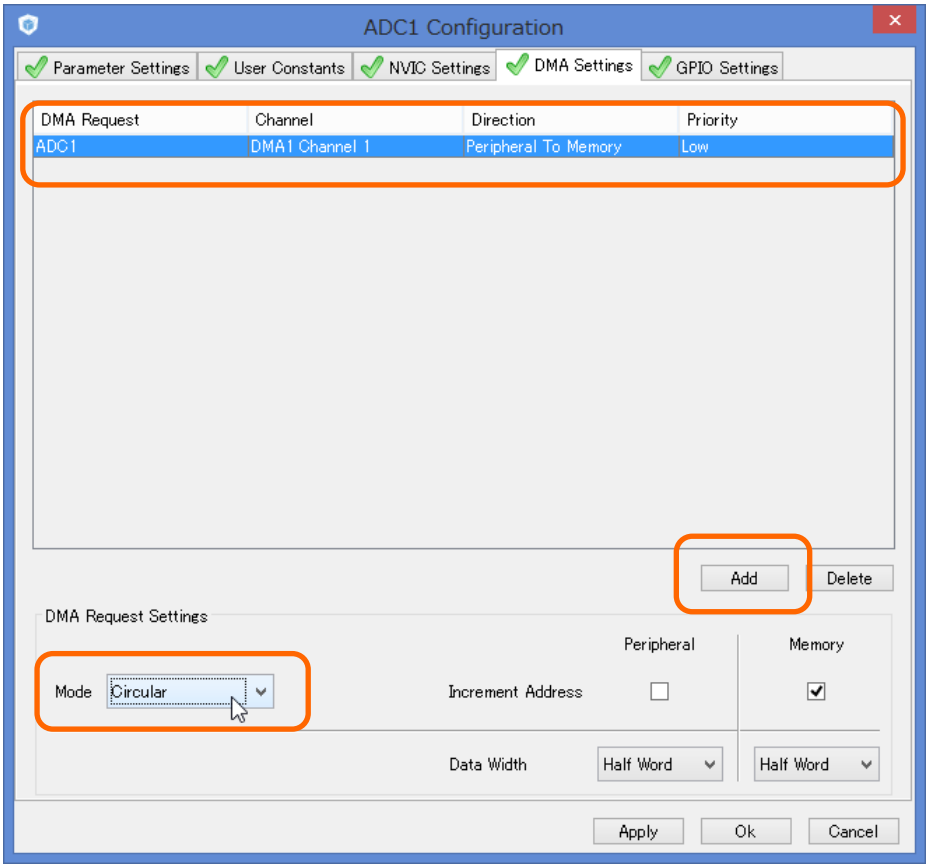
ダイアログ右下にあるData Width欄はPeripheral＝「Half Word」Memory＝「Half Word」にしておきます。Half Wordは16ビットのことで、ETCBではAD変換値が12ビットの解像度があるため、データを格納するメモリーは16ビット用意しておくという意味です。データを格納するメモリーはプログラムで設定します。

[5] 1Cycleは1/72MHzの時間がかかります。サンプリング周期を短くすると、他の処理に移れなくなる場合があり、変換精度も悪くなりますので、601.5Cycleを選択した方が良いでしょう。

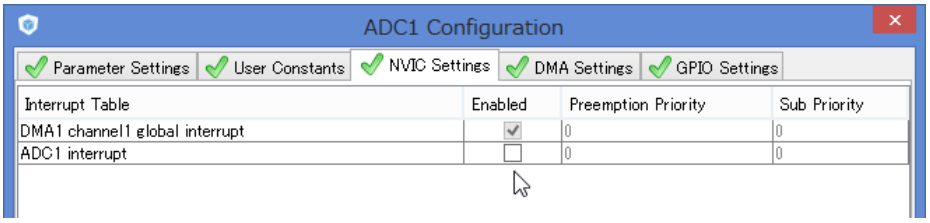
[6] ソフトウェアトリガー：プログラムで変換を開始すること。

[7] 通常はソフトウェアトリガーで繰り返し変換を行う必要があります。

また、「DMA Request Settings」のModeを「Circular」にセットして、自動的に繰り返しDMA転送が行われるようにします。



つぎにNVIC Settingsタブを開き、DMA1 channel1 global interruptのEnabled欄にチェックマークが入れてください。Enabledのときは DMA転送完了時に割り込みが発生します。ADC1 interruptのEnabled欄にチェックマークを入れるとAD変換が完了する度に割り込みが発生してしまい、多チャンネルを変換している場合は割り込み発生頻度が高くなり、他の機能が実行されなくなる恐れがあります。今回はチェックしないでください。



割り込み優先順位設定

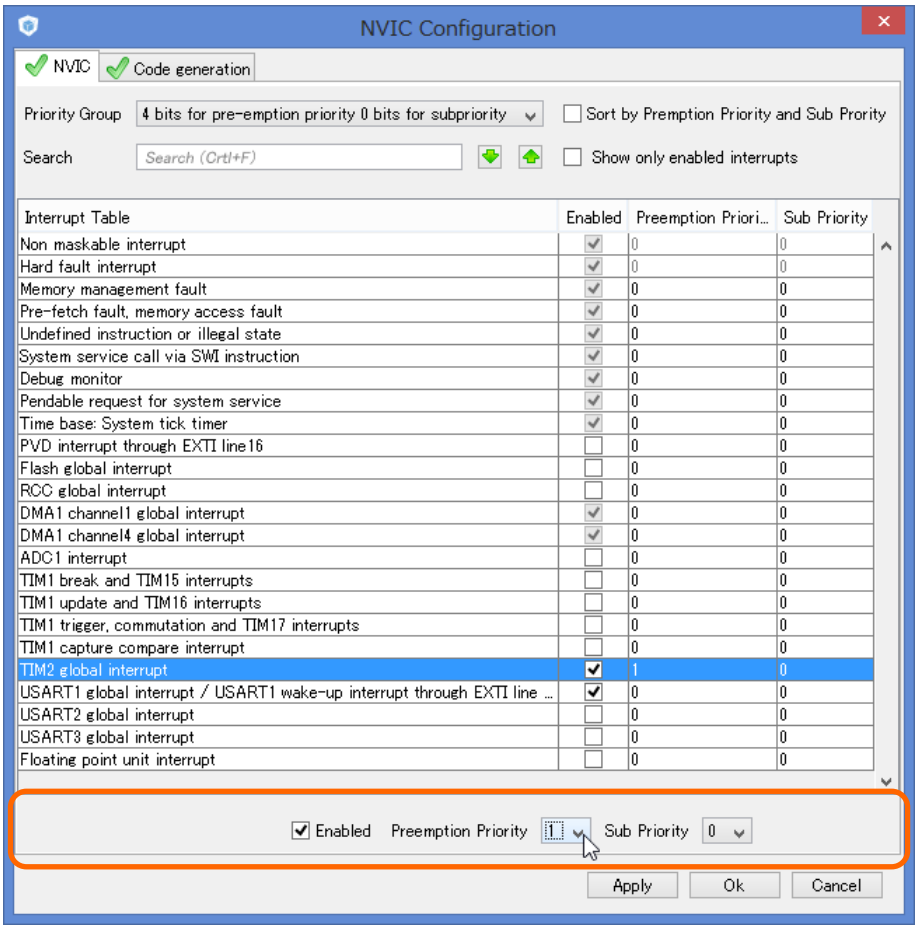
割り込み関数の優先順位を設定します。優先順位の高い割り込みを処理している最中は、優先順位の低い割り込みは実行されません。逆に優先順位の低い割り込みを処理している間でも、優先順位の高

い割り込みが発生すると、いったん優先順位の高い方へ処理が移動し、それが終わると元の割り込み位置へ処理が戻ります。

UARTの受信割り込みを最優先にしないと、データの取りこぼしが起きることがありますので、UART受信割り込みは最優先に設定します。SystemカテゴリからNVICボタンを押して、「NVIC Configuration」ダイアログを表示します。今回はTIM2のタイマー周期割り込みとUART1受信割り込みを使いますので、USART1受信割り込みの優先度を0、TIM2割り込みを1にしています。

Priority Group

割り込みの優先順位の付け方をグループ（プリエンプションとサブプライオリティの組み合わせ）によって変えることができます。優先順位は多重割り込みが発生した場合の実行優先順位です。優先順位の高い処理をしている最中は優先順位の低い割り込みは発生しません。



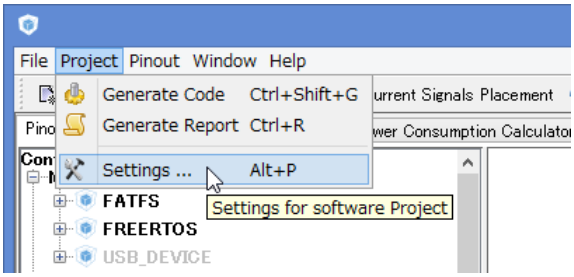
優先順位は全部で4ビット（0〜15）を設定できます。上の図では4ビット全てをプリエンプションに割り当てていますが、例えばプリエンプションに2ビット（0〜3）・サブプライオリティに2ビット（0〜3）を割り当てることもできます。振り分けたときはプリエンプションを同じ優先度にしてサブプライオリティで優先度を変えることもできます。プリエンプションはサブプライオリティよりも優

先されます。また0に近い方の優先度が高くなります。
優先度に同じ数値を割り当てると割り込みがうまく発生しないことがあります。必ずユニークな優先順位を指定してください。

プログラムの生成

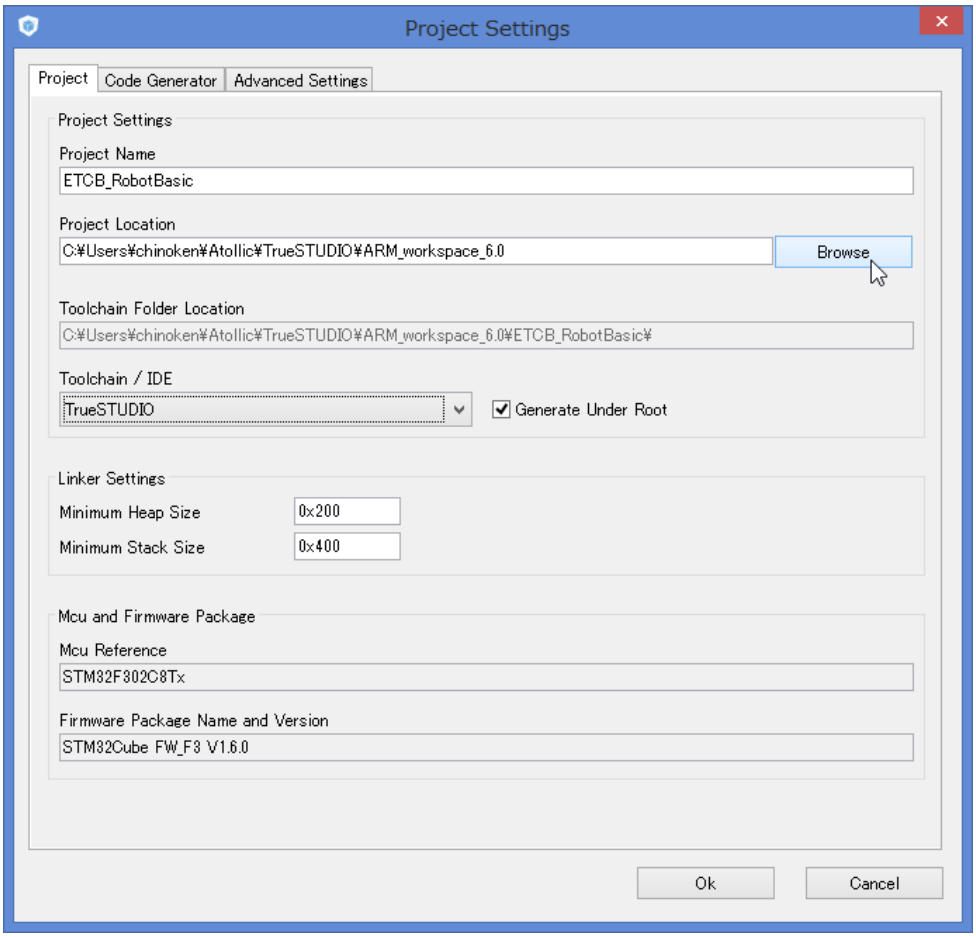
プロジェクトの保存

STM32CubeMXのFileメニューから「Save Project」を選択するか、Projectメニューから「Settings」を選択します。新規作成時は「Project Settings」ダイアログが表示されますので、下記のように設定してください。

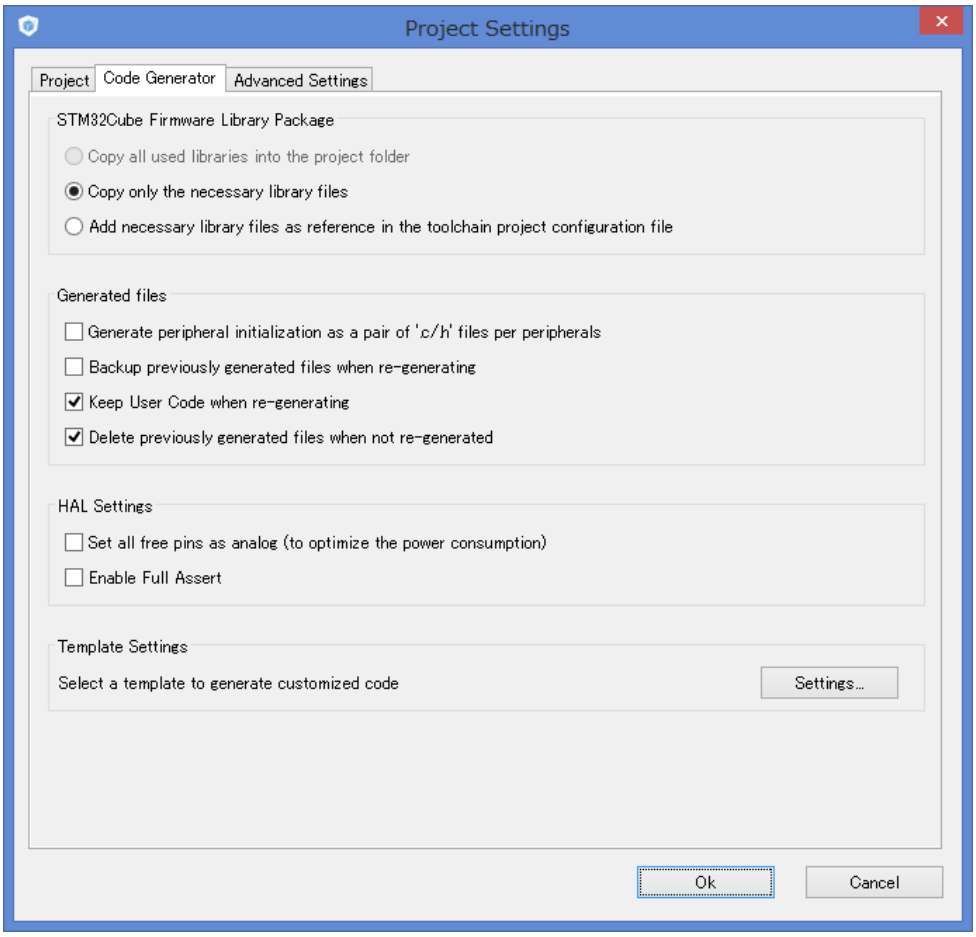


- Project Location　　TrueSTUDIOで使っているWorkspaceフォルダを指定
- Project Name　　任意の名前を設定。ここではETCB_RobotBasic

- Toolchain/IDE　　TrueSTUDIO



次にProject SettingsダイアログのCode Generatorタブを開きます。各項目のチェック状態を下図のようにします。



Generated files欄の「Generate periheral initialization as …」にチェックを入れると、AD変換、タイマー、DMA転送など機能ごとにC言語ソースファイルとヘッダファイルが生成されます。大規模プログラムを作成するときはプログラム構成や規模が確認しやすくなる反面、変数や関数の取り扱いがやや複雑になります。本例題ではチェックを入れないで、初期化関数は全てmain.cにまとめて保存されるように設定します。

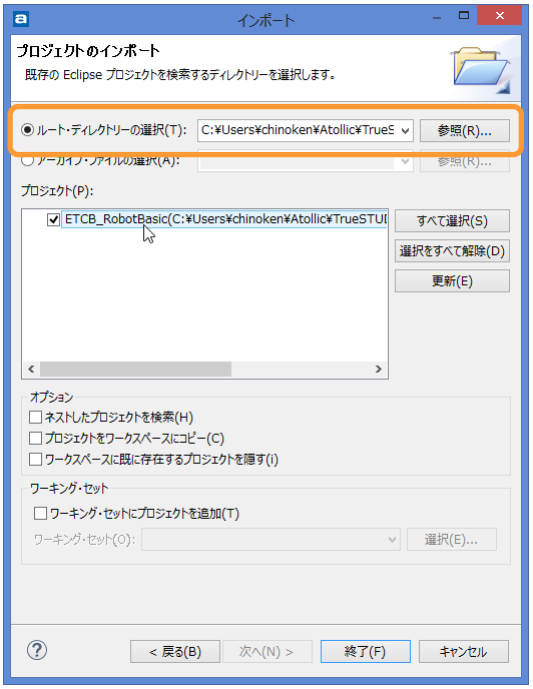
設定が終わったらOKボタンでダイアログを閉じます。

コード（プログラムひな形）の生成

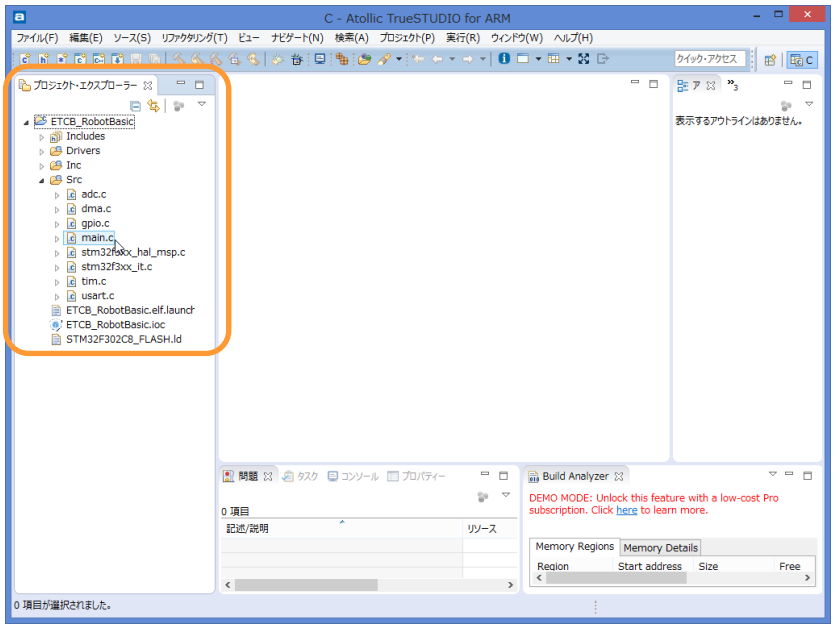
STM32CubeMXのメインメニューにある、Projectメニューから「Generate Code」を選択してください。Project Settingsで設定したフォルダにプログラムのひな形が保存されています。

プログラム編集

まず19ページの「*TrueSTUDIOを使ったソースコード編集・デバッグ*」と同様に、作成したひな形をインポートします。インポート時に「ルート・ディレクトリの選択」をチェックして、ひな形（フォルダ）があるディレクトリ（フォルダ）を選択します。そうすると「プロジェクト」欄にインポート可能なプロジェクトが表示されますので、チェックマークを入れて終了ボタンを押します。



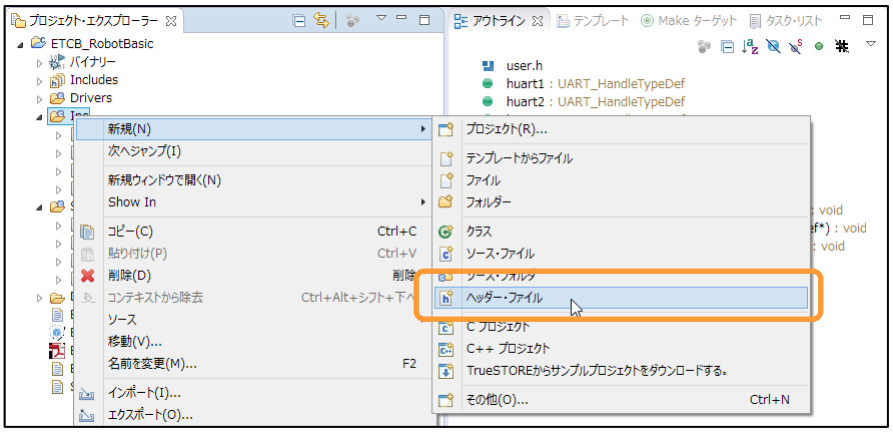
インポートが正常に実行されると、次のようなプログラム編集画面が表示されます。

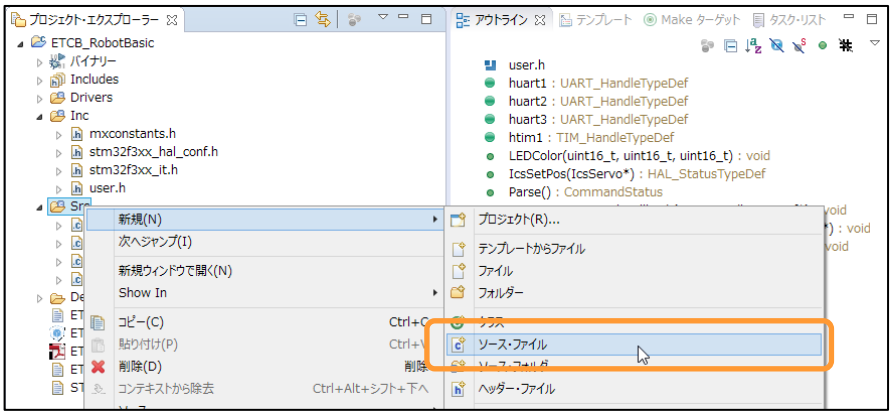


プロジェクト構成の確認とソースコード・ヘッダファイルの追加

TrueSTUDIOのプロジェクトエクスプローラーウィンドウにいくつかファイルとディレクトリが表示されますが、ユーザーがソースコード（.c）を追加する場合は、「Src」ディレクトリに追加してください。関数や変数の定義をするヘッダファイルは「Inc」ディレクトリに追加してください。

今回はヘッダファイル「user.h」とソースコードファイル「user.c」をそれぞれIncディレクトリとSrcディレクトリに追加します。追加するときは追加したいディレクトリのアイコンを右クリックして、新規(N)メニューからソース・ファイルまたはヘッダー・ファイルを選んでください。





例題プログラム概要（内容の確認）

最初にプログラム上の各機能の実装について説明します。

1. 初期設定について

   DMA転送設定やUART通信設定など各機能設定は、STM32CubeMXによって初期化関数がmain.cに自動生成されています。自動生成された関数名は最初に「MX_」、最後に「_Init」とついています。また、これらの初期化関数はすでにmain関数の冒頭で呼び出されていますので、ユーザーは初期化（STM32CubeMXで行った設定）について改めてプログラムを書く必要はありません。

   初期化が終わった後で、ユーザーはタイマーや割り込みを開始するコードを書きます。

2. main.cでのプログラム作成上の注意
   STM32CubeMXで生成したプログラムのmain.cでは、ユーザーが関数を記載して良い場所が決まっています。「/* USER CODE BEGIN 0 */」から「/* USER CODE END 0 */」などです。同様にinclude定義をする場所や、ローカル変数を記載する場所も決まっています。これらはSTM32CubeMXでPinoutを修正してコードを再度出力するときに、ユーザーが書いたコードを削除してしまわないための目印となっています。自由に関数を作成したいときは、main.c以外のCソースファイルを作成してください。

3. PWM信号でLEDを点灯（カラー調整）する
   PWM信号は元となるタイマー（TIM1）を`HAL_StatusTypeDef HAL_TIM_Base_Start(TIM_HandleTypeDef *htim)`関数で起動します。その後、`HAL_StatusTypeDef HAL_TIM_PWM_Start(TIM_HandleTypeDef *htim, uint32_t Channel)`関数でPWMを3チャンネル（RGB各カラーの明るさを調節する）起動します。
   PWMデューティは各カラーを`void LEDColor(uint16_t, uint16_t, uint16_t)`関数で調整します（100段階）。

4. AD変換とDMA転送
   DMA転送開始と転送先変数を登録する関数は`HAL_StatusTypeDef HAL_ADC_Start_DMA(ADC_HandleTypeDef* hadc, uint32_t* pData, uint32_t Length)`関数で、pDataにあらかじめ宣言した変数名を入れます。
   次にAD変換がDMA転送完了すると、割り込み関数`void HAL_ADC_ConvCpltCallback (ADC_HandleTypeDef *hadc)`が自動的に読み出されます。この関数の中でAD値をfloat型の電圧や温度に変換します。HALライブラリでは割り込み関数はライブラリ内で「__weak」定義[8]していますので、ユーザーは改めて関数宣言する必要はありません。関数だけ実装してください。

5. MCU温度計算
   MCUの温度は下記の式で計算します。V_{25} やAvg_Slopeはデータシートから代表的な数値を取り出しています。
   Temperature [℃] = { (V_{25} - V_{TS}) / Avg_Slope } + 25

   A) V_{25} = 1.43 [V], Avg_Slope = 4.3 [mV/℃]

   B) V_{TS} = x * 3.3 / 4096 (xは計測したAD値)

[8] weak関数属性があると、ユーザーが関数を重複して定義・実装した場合に、関数がオーバーロードされ、ユーザー定義関数が優先実行されます。HALライブラリでは割り込み関数は全てweak定義されています。

C) Avg_Slopeの単位はmV/℃なので、単位を合わせるためAvg_Slope＝0.0043で計算する

6. Raspberry Piからコマンドを送信、ETCBで受信する

UART1の受信割り込みで位置命令を受け取ります。割り込みを処理は`void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)`の中で実装します。この関数もweak関数属性がついていますので、関数を宣言する必要はありません。関数の実装だけしてください。

UART割り込み処理の実行は、プログラム内で`HAL_StatusTypeDef HAL_UART_Receive_IT(UART_HandleTypeDef *huart, uint8_t *pData, uint16_t Size)`を実行しておくと、データ受信待ち状態となります。データが来ると受信割り込み関数`HAL_UART_RxCpltCallback`が実行されますので、その関数内で再度`HAL_UART_Receive_IT`関数を実行し、次の受信待ち状態にすることで、繰り返し割り込み受信します。`pData`に受信バッファ用の配列変数を指定し、`Size`には1を指定します。`Size`を例えば3などと指定すると3文字受信するまで割り込みが発生しなくなります。

Raspberry Piからのコマンドは、データ受信間隔が5ms（タイムアウト）程度開いたら、一連のコマンドの区切り（コマンド受信完了）と判断します。`HAL_UART_RxCpltCallback`関数内でタイムアウト変数（u1timeout）を5にセットします。後述のSYSTICKタイマーイベントでu1timeoutを1ずつ減らし、0になったらコマンド受信完了と判断し、それまで受け取ったデータを`Parse`関数で解析します。`Parse`関数では受信したコマンドをID番号と目標位置へ分解して、サーボモーターのデータ保存用変数（`IcsServo`構造体）にコピーします。

Raspberry Piから送信する命令は次のような構造にしてください。

A) ICS番号は下記の式で決めます。プログラム内ではサーボ構造体変数配列のインデックス番号として使用しています。

ICS＝ID番号×2＋Port　（Port=0（S1~S6）、Port=1（S7~S12））

B) 近藤科学製のサーボモーターはポジション（角度）を3500（-135°）～11500（135°）まで指定できます。POS_Lはポジションの下位1バイト、POS_Hは上位1バイトとなります。例えば7500（0x1D4C）を指定する場合は、POS_L＝0x4C（76）、POS_H＝0x1D（29）です。

C) SUMはチェックサムです。SUM以外のデータを全て足し合わせた数の下位1バイトのデータを指定してください。

SUM＝ICS1＋POS1_L＋POS1_H＋･･･＋ICSn＋POSn_L＋POSn_H

D) 受け取ったデータは`Parse`関数で内容を確認し、正しいコマンドと判断されたら`IcsServo`構造体配列の`dpos`変数に受け取ったポジションデータを保存します。ポジションデータは下位バイトを先に指定してください。

バイト数	1	2	3	4	5	6	…
内容	ICS1	POS1_L	POS1_H	ICS2	POS2_L	POS2_H	…
	…	3n-2	3n-1	3n	3n+1		
	…	ICSn	POSn_L	POSn_H	SUM		

7. 20ms周期でサーボにコマンドを送る
   TIM2を使って20ms周期で割り込みを発生させ、割り込み関数でサーボにコマンドを送ります。TIM2の周期割り込み（タイマー割り込み）関数は`void HAL_TIM_PeriodElapsedCallback (TIM_HandleTypeDef *htim)`です。割り込み関数内でサーボモーターにポジションコマンドを送ります。ポジションは`IcsServo`構造体から読み出します。

8. STM32CubeMXのSYSメニューで指定したSYSTICK（36ページ「デバッガ設定」を参照）では、1ms間隔で割り込みが発生します（`void SystemClock_Config(void)`内で定義）。割り込み関数は`void HAL_SYSTICK_Callback ()`です。この関数と受信割り込み関数を使って、下記のような手順でコマンドの区切りを探し、コマンドの実行を行います。

   A) UART受信割り込みが発生するたびに`u1timeout`が5にリセットされる

   B) 受信割り込みが発生しない（コマンドが来ない）と、`HAL_SYSTICK_Callback`関数で`u1timeout`が1msあたり1ずつカウントダウンされる

   C) `u1timeout`が0になったら、それまで受信したデータをひとまとめのコマンドとして実行する（`Parse`関数）

   D) カウントダウンの最中にUART受信割り込みが発生したら、`u1timeout`が再度5にリセットされ、コマンドが続いている物と判断する

   SYSTICK割り込みはあまり正確ではないので注意してください。

9. 今回は設定のみで実際は使用していませんが、UARTからDMA転送でデータを出力するときは`HAL_StatusTypeDef HAL_UART_Transmit_DMA(UART_HandleTypeDef *huart, uint8_t *pData, uint16_t Size)`関数を使います。返事をpDataと同じ型の変数に代入し、`HAL_UART_Transmit_DMA`関数を実行してください。

ここからは分かりやすくコメントを入れたソースコードを掲載します。

user.h

user.hはユーザー定義関数の定義や、変数定義などを行っています。詳細はプログラムのコメントを参照してください。

```
/*
 * user.h
 *
 *  Created on: 2016/09/18
 *      Author: chinoken
 */

#ifndef USER_H_
#define USER_H_
```

```
#include "stm32f3xx_hal.h" // HALライブラリを使うためにインクルードしておくこと

/*
 * 共通設定
 */
typedef enum
{
  FALSE = 0, // Bool変数は定義されていないので、ユーザーが定義する
  TRUE = 1
} Boolean;

/*
 * 定義
 */
#define UART1_RXBUF_SIZE    32 // 最大受信バッファサイズ
#define UART1_RX_TIMEOUT    5  // 受信タイムアウト(5=5ms)

/*
 * サーボポート指定(ETCBのS1～S12までのポートを指定するときに使用する)
 */
typedef enum
{
  S1 = 0,
  S2 = 0,
  S3 = 0,
  S4 = 0,
  S5 = 0,
  S6 = 0,
  S7 = 1,
  S8 = 1,
  S9 = 1,
  S10 = 1,
  S11 = 1,
  S12 = 1
} SIO_Port;

/*
 * コマンドエラー
 */
typedef enum
{
  CMD_ERROR_EXCEPTION = -1, // use for HAL_ERROR
  CMD_ERROR_SUCCESS = 0, // succeed to execute command
  CMD_ERROR_TRANSMIT = 1, // fail to transmit data
  CMD_ERROR_RECEIVE = 2, // fail to receive data (enough data length and so on)
  CMD_ERROR_LENGTH = 3, // invalid command length
```

受信バッファサイズを変更する場合はこの数値を変える

サーボモーターの取付け位置を以下の式で数値化しておく
ics = id * 2 + port
ics番号をサーボ構造体配列インデックスに使うと、ポートごとに交互にポジション命令が出せるようになる

```
  CMD_ERROR_INVALID_COMMAND = 4, // invalid command (header)
  CMD_ERROR_CHECKSUM = 5 // invalid checksum
}
CommandStatus;

/*
 * 近藤科学ICSサーボ用のコマンド
 */
#define ICS_SERVO_NEUTRAL      7500
#define ICS_SERVO_LOWER_LIMIT  3500
#define ICS_SERVO_UPPER_LIMIT  11500
#define ICS_SET_POS            0x80 // 0b101xxxxx, xはIDが入る
#define MAX_SERVO_COUNT        8    // サーボの数を最大8個にする(変更時はTIM2タイマー間隔を調整すること)

/*
 * サーボモーターを管理しやすくするための共用体
 * ics番号は配列の添え字で使用し、ics = id * 2 + portの関係になる
 */
typedef union
{
  struct
  {
    char port:1;  // SIOポート番号(S1~S12)
    char id:5;    // ID番号
    char mode:1;  // normal=0, rotate=1;
    char enable:1;// サーボ使用bit(使用する=1,使用しない=0)
  } id_bits;
  struct
  {
    char ics:6;   // ICS番号:ics = id * 5 + portの計算が成立する
    char b:2;     // 使用しない
  } ics_bits;
  uint8_t value;
} IcsConfig;

/*
 * サーボモーターの位置などを保持するための構造体
 */
typedef struct
{
  IcsConfig config;  // サーボID情報
  uint16_t dpos;     // ポジション命令を出すときの目標位置
  uint16_t cpos;     // サーボから取得した現在位置
} IcsServo;
```

```
IcsServo ics_servo[MAX_SERVO_COUNT]; // サーボデータ保存用構造体変数を宣言する

/*
 * 簡単な変数の範囲制限
 */
#define TRIM(a,min,max)     ((a)<(min)?(a)=(min):(a)>(max)?(a)=(max):(a))
/*
 * AD変換
 */
uint16_t adc_raw_data[5]; // AD1~4と温度データを保存する
float adc_value[5]; // AD返還後のデータを保存する
/*
 * UART1受信割り込みバッファ
 */
uint8_t u1buff[1]; // 1文字受信用のバッファ
uint8_t u1cmd[UART1_RXBUF_SIZE];
uint8_t u1index;
uint16_t u1timeout;

/*
 * 関数定義(オリジナル関数のみ)
 */
void LEDColor(uint16_t, uint16_t, uint16_t);
HAL_StatusTypeDef IcsSetPos (IcsServo *);
CommandStatus Parse();

#endif /* USER_H_ */
```

サーボに出力可能な範囲に数値を制限する

u1buffは１文字受信割り込み用バッファに使う。受信したデータはu1cmdに保存し、u1indexで受信したデータ数を数えておく

user.c

user.cではuser.hで定義したユーザー関数の実装と割り込み関数を実装しています。

```
/*
 * user.c
 *
 *  Created on: 2016/09/18
 *      Author: chinoken
 */

#include "user.h" // 定義や宣言を読み込むため、user.hをインクルードする

/*
 * main.cで定義されている変数をuser.cで使うためextern宣言する
 */
```

```
extern UART_HandleTypeDef huart1;
extern UART_HandleTypeDef huart2;
extern UART_HandleTypeDef huart3;
extern TIM_HandleTypeDef htim1;

/*
 * PWM出力(LEDのカラー)を変更
 */
void LEDColor (uint16_t r, uint16_t g, uint16_t b)
{
  htim1.Instance->CCR1 = (uint32_t)TRIM(r,0,100); // 最大値は100=10ms
  htim1.Instance->CCR2 = (uint32_t)TRIM(g,0,100);
  htim1.Instance->CCR3 = (uint32_t)TRIM(b,0,100);
}

/*
 * ICSサーボにポジションコマンドを送る
 */
HAL_StatusTypeDef IcsSetPos (IcsServo *servo)
{
  HAL_StatusTypeDef hStat; // HAL Library関数返値の受け取り
  uint8_t cmd[3], rx[3];
  uint16_t pos = 0;
  UART_HandleTypeDef *huart;

  cmd[0] = ICS_SET_POS | servo->config.id_bits.id;
  cmd[1] = (servo->dpos >> 7) & 0x7F; // 上位7bit
  cmd[2] = (servo->dpos) & 0x7F; // 下位7bit

  // ポートによってUSARTを切り替える
  huart = (servo->config.id_bits.port == 0 ? &huart3 : &huart2);

  // コマンドを送信する(受信タイムアウトは2ms)
  // 半二重通信なので、送信と受信の切り替えが必要です
  HAL_HalfDuplex_EnableTransmitter(huart); // 送信を有効にする
  hStat = HAL_UART_Transmit(huart, cmd, 3, 2); // cmd変数3byteを2msのタイムアウトで送信
  if (hStat == HAL_OK)
  {
    // 正常に送信できたのでサーボからの返事を受信する
    HAL_HalfDuplex_EnableReceiver(huart); // 受信を有効にする
    hStat = HAL_UART_Receive(huart, rx, 3, 2); // rx変数に3byteを2msのタイムアウトで受信
    if (hStat == HAL_OK) // 受信成功の場合
    {
      if ((rx[0] & 0x3F) == servo->config.id_bits.id) // 受信データのidを確認
      {
        pos = (rx[1] << 7); // 上位7bit
```

近藤科学ICSサーボコマンドに準拠した処理

```
        pos += (rx[2] & 0x7F); // 下位7bit分を追加
        servo->cpos = TRIM(pos, 3500, 11500); // 返ってきた値を範囲制限して保存する
      }
    }
    else
    {
      return HAL_TIMEOUT;
    }
  }
  else
  {
    return HAL_TIMEOUT;
  }

  return HAL_OK;
}

/*
 * Rasberry Piから受信したコマンドを解析する
 */
CommandStatus Parse()
{
  /*
   * コマンドの内容(各1バイト)
   * ICS1 POS1_L POS1_H ICS2 POS2_L POS2_H ,..., ICSn POSn_L POSn_H SUM
   * ICS = ID x2 + Port (Port = 0(S1~S6), Port = 1(S7~S12))
   * SUMはそれ以外のデータの総和の下位1バイト
   */
  if (u1index % 3 != 1) // コマンドの数は3の倍数+1
  {
    return CMD_ERROR_LENGTH;
  }

  // チェックサム
  uint8_t sum = 0;
  uint16_t i;

  for (i = 0; i < u1index - 1; i++)
  {
    sum += u1cmd[i];
  }

  if (sum != u1cmd[u1index - 1])
  {
    return CMD_ERROR_CHECKSUM;
  }
```

```
  // 解析
  uint16_t pos;
  uint8_t ics;

  for (i = 0; i < u1index - 1; i+=3) // 0~23
  {
    ics = u1cmd[i] & 0x3F; // 下位6bit
    pos = ((uint16_t)u1cmd[i + 2]) << 8; // 上位8bit
    pos += u1cmd[i + 1]; // 下位8bit

    pos = TRIM(pos,ICS_SERVO_LOWER_LIMIT,ICS_SERVO_UPPER_LIMIT);

    ics_servo[ics].dpos = pos; // dposに受け取った目標位置を保存する
  }

  return CMD_ERROR_SUCCESS;
}

/*
 * 以下は割り込みの処理
 */
/*
 * HAL_ADC_ConvCpltCallback関数はDMA転送完了後に自動的に実行される
 */
void HAL_ADC_ConvCpltCallback (ADC_HandleTypeDef *hadc)
{
  // AD値を元の単位へ変換する
  adc_value[0] = ((float)adc_raw_data[0])*3.3/4096.0;
  adc_value[1] = ((float)adc_raw_data[1])*3.3/4096.0;
  adc_value[2] = ((float)adc_raw_data[2])*3.3/4096.0;
  adc_value[3] = ((float)adc_raw_data[3])*3.3/4096.0;
  // 温度計算
  float Vts = ((float)adc_raw_data[4])*3.3/4096.0; // AD値を電圧に変換
  adc_value[4] = ((1.43 - Vts) / 0.0043) + 25; // 温度
}

/*
 * タイマー割り込み
 */
void HAL_TIM_PeriodElapsedCallback (TIM_HandleTypeDef *htim)
{
  if (htim->Instance == TIM2) // 割り込みの種類がTIM2だったとき
  {
    HAL_StatusTypeDef hStat;
    uint16_t i, j;
```

TIM2割り込み関数でdpos変数を使ってポジションコマンドを実行する

割り込み発生周期はAD変換周期とほぼ同じ

タイマー割り込みは割り込みの発生元を確認すること
割り込みタイマーが複数の場合はifで分岐する

```
    // サーボに命令を送る
    for (i = 0; i < MAX_SERVO_COUNT; i++)
    {
      if (ics_servo[i].config.id_bits.enable == TRUE)
      {
        // サーボにポジションコマンドを送信する
        hStat = IcsSetPos(&ics_servo[i]);

        if (hStat != HAL_OK)
        {
          // エラー処理をする場合
        }
      }
      // ちょっと間を空ける
      for (j = 0; j < 100; j++) {}
    }
  }
}

/*
 * UART1受信完了割り込み
 */
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
{
  if (huart->Instance == USART1) // 割り込み条件がUART1だった場合
  {
    u1cmd[u1index] = u1buff[0]; // 受信した1文字を配列に追加
    HAL_UART_Receive_IT(&huart1, u1buff, 1); // 1文字受信待ちを再開

    u1index ++; // カウンタ(配列のインデックス)を進める

    if (u1index > UART1_RXBUF_SIZE)
    {
      u1index = 0; // 受信バッファが変数の最大サイズを超えたらインデックスを最初に戻す
    }

    u1timeout = UART1_RX_TIMEOUT; // 受信
  }
}

/*
 * SYSTICK割り込み(システムクロックに
 */
void HAL_SYSTICK_Callback ()
{
  // 受信タイムアウトが発生したらそれまで受け取った命令を解析する
```

ics_servo配列のインデックスはics番号なので、インクリメントするだけでIcsSetPos命令をUART2,3ポートに交互に出せる

UARTも割り込み発生元を確認すること

SYSTICK割り込み周期はmain.cのSystemClock_Config関数で
HAL_SYSTICK_Config(HAL_RCC_GetHCLKFreq()/1000);
と定義している。クロックカウント＝1秒なので、1000で割ることで1msの割り込みを設定している。1000を10000とすると0.1msの割り込みになる

```
  if (u1timeout != 0)
  {
    if (--u1timeout == 0) // 受信タイムアウトイベントが発生
    {
      CommandStatus c_stat = Parse(); //コマンド(受信データ)を実行

      if (c_stat != CMD_ERROR_SUCCESS)
      {
        // エラーがあった場合の処理
      }

      u1index = 0;   // インデックスをリセット
    }
  }
}
```

main.c

本例題では、位置命令の受信やサーボへの命令の送受信、AD変換など全て処理は割り込み関数の中で行っているので、main.cでは特にプログラム処理命令はありません。変数初期化やETCBの登録のみ行っています。

```
/*
 * user.c
 *
 *  Created on: 2016/09/18
 *      Author: chinoken
 */
/* Includes ------------------------------------------------------------------*/
#include "stm32f3xx_hal.h"

/* USER CODE BEGIN Includes */
#include "user.h"
/* USER CODE END Includes */

/* Private variables ---------------------------------------------------------*/
ADC_HandleTypeDef hadc1;
DMA_HandleTypeDef hdma_adc1;

TIM_HandleTypeDef htim1;
TIM_HandleTypeDef htim2;

UART_HandleTypeDef huart1;
UART_HandleTypeDef huart2;
```

ユーザーのインクルード定義は
ここの間に記入する

```
UART_HandleTypeDef huart3;
DMA_HandleTypeDef hdma_usart1_tx;

/* USER CODE BEGIN PV */
/* Private variables ---------------------------------------------------------*/

/* USER CODE END PV */

/* Private function prototypes -----------------------------------------------*/
void SystemClock_Config(void);
void Error_Handler(void);
static void MX_GPIO_Init(void);
static void MX_DMA_Init(void);
static void MX_ADC1_Init(void);
static void MX_TIM1_Init(void);
static void MX_TIM2_Init(void);
static void MX_USART1_UART_Init(void);
static void MX_USART2_UART_Init(void);
static void MX_USART3_UART_Init(void);

void HAL_TIM_MspPostInit(TIM_HandleTypeDef *htim);


/* USER CODE BEGIN PFP */
/* Private function prototypes -----------------------------------------------*/

/* USER CODE END PFP */

/* USER CODE BEGIN 0 */

/* USER CODE END 0 */

int main(void)
{

  /* USER CODE BEGIN 1 */
  uint16_t i;
  /* USER CODE END 1 */

  /* MCU Configuration--------------------------------------------------------*/

  /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
  HAL_Init();

  /* Configure the system clock */
  SystemClock_Config();
```

ユーザーのグローバル変数はここの間に記入する

ユーザーの関数プロトタイプ宣言はここの間に記入する

ユーザー関数はここの間に記入する（今回は別ファイルを作った）

ここにローカル変数の定義を記載する

クロック設定

```
  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_DMA_Init();
  MX_ADC1_Init();
  MX_TIM1_Init();
  MX_TIM2_Init();
  MX_USART1_UART_Init();
  MX_USART2_UART_Init();
  MX_USART3_UART_Init();

  /* USER CODE BEGIN 2 */
  /*
   * PWM出力(LED点灯)
   */
  HAL_TIM_Base_Start(&htim1); // TIM1を開始
  HAL_TIM_PWM_Start(&htim1, TIM_CHANNEL_1); // PWMタイマーを開始
  HAL_TIM_PWM_Start(&htim1, TIM_CHANNEL_2);
  HAL_TIM_PWM_Start(&htim1, TIM_CHANNEL_3);
  LEDColor(100,0,0); // 赤を点灯
  /*
   * AD変換を開始する
   */
  HAL_ADC_Start_DMA(&hadc1, (uint32_t *)adc_raw_data, 5); // AD返還後にデータを6個DMA転送する
  /*
   * UART受信割り込み
   */
  // 受信文字数カウンタをリセット
  u1index = 0;
  // UARTの受信割り込みは1文字受信待ちを実行すると開始となる
  HAL_UART_Receive_IT(&huart1, (uint8_t *)u1buff, 1);
  // 受信タイムアウト時間をリセット
  u1timeout = 0;
  /*
   * サーボモーターのデータを初期化する
   */
  for (i = 0; i < MAX_SERVO_COUNT; i++)
  {
    ics_servo[i].config.value = 0; // Port=0,ID=0,Enable=FALSE(0)
    ics_servo[i].dpos = ICS_SERVO_NEUTRAL;
    ics_servo[i].cpos = ICS_SERVO_NEUTRAL;
  }

  /*
   * ここから使用するサーボだけID(例として4個分)を登録する
```

自動生成されたGPIO、ADなど諸設定関数のプロトタイプ宣言中身はmain関数の後に記載している

サーボのenableビットをいったん0にする

```
 * S1~S6ポートにID=0,1のサーボを2個、S7~S12ポートにID=0,1のサーボを2個、合計4個登録する
 * index[0]: id=0,port=S1
 * index[1]: id=0,port=S7
 * index[2]: id=1,port=S2
 * index[3]: id=1,port=S8
 */
ics_servo[0].config.id_bits.port = S1; // S1ポートにID=0のサーボを取り付ける
ics_servo[0].config.id_bits.id = 0;
ics_servo[0].config.id_bits.enable = TRUE; // サーボを有効にする

ics_servo[1].config.id_bits.port = S7; // S7ポートにID=0のサーボを取り付ける
ics_servo[1].config.id_bits.id = 0;
ics_servo[1].config.id_bits.enable = TRUE; // サーボを有効にする

ics_servo[2].config.id_bits.port = S2; // S2ポートにID=1のサーボを取り付ける
ics_servo[2].config.id_bits.id = 1;
ics_servo[2].config.id_bits.enable = TRUE; // サーボを有効にする

ics_servo[3].config.id_bits.port = S8; // S8ポートにID=1のサーボを取り付ける
ics_servo[3].config.id_bits.id = 1;
ics_servo[3].config.id_bits.enable = TRUE; // サーボを有効にする

/*
 * TIM2タイマー割り込み開始
 */
HAL_TIM_Base_Start_IT(&htim2);

/*
 * 準備完了したのでLEDを緑にする
 */
LEDColor(0,100,0); // 緑を点灯

/* USER CODE END 2 */

/* Infinite loop */
/* USER CODE BEGIN WHILE */
while (1)
{
/* USER CODE END WHILE */

/* USER CODE BEGIN 3 */
  HAL_Delay(50);

  LEDColor (0,i,0); // 適当に

  i = (i + 1) % 100;
```

これ以降はSTM32CubeMXが自動生成した関数（MX_）なので省略

```
  }
  /* USER CODE END 3 */
}
```

コマンドを送信する

53ページの「5(エ)」で説明した命令を、USB1ポートから送る手順について説明します。

1. CoolTermを起動します。
2. ツールバーのOptionsボタンを押すか、メインメニューからConnection＞Optionsを選択して、Connection Optionダイアログを表示します。
3. Connection Optionダイアログで、下記のように設定します。

   (ア) Port： ETCBのUSB仮想COMポートを指定します

   (イ) Baudrate： 115200

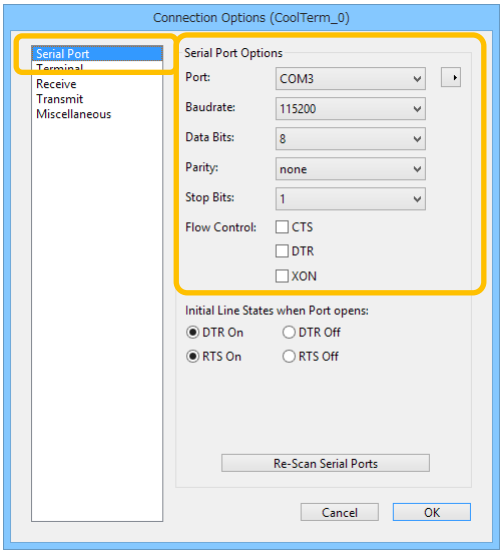
   (ウ) Data Bits： 8

   (エ) Parity： none

   (オ) Stop Bits： 1

   (カ) Flow Controlは全てチェックオフ

   (キ) OKボタンを押して閉じます。



4. ツールバーのConnectボタンを押すか、メインメニューからConnection＞Connectを選択して、仮想COMポートを開きます。

5. ツールバーのView Hexボタンを押すか、メインメニューからView＞View Hexを選択して、画面を16進数表示に変更します。

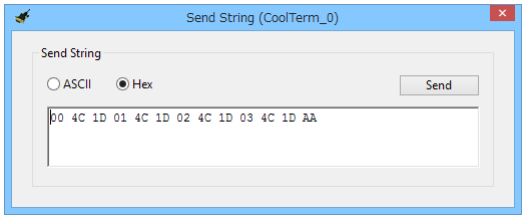
6. メインメニューからConnection＞Send Stringを選択して、データ入力用のSend Stringダイアログを開きます。

7. サーボモーターへの命令を入力します。下記の(ア)はプログラムで登録した全てのサーボモーター（ID=0,Port=S1/ID=1,Port=S2/ID=0,Port=S6/ID=1,Port=S7）がニュートラル位置（7500=0x1D4C）へ移動するコマンドです。(イ)は全てのサーボが6000（=0x1770）、(ウ)は全てのサーボが9000（=0x2328）に移動するコマンドになります。リトルエンディアンで受け取るので、数値は下位のバイトを先にします。青い文字はICS番号で赤い文字はCHECKSUMになります。
下図は(ア)を入力した、SendString画面です。SendString画面で文字列の最後を改行してしまうと、改行コードも送信されてETCBでエラーが出るので、改行を入れないでください。

   (ア) 00 4C 1D 01 4C 1D 02 4C 1D 03 4C 1D AA

   (イ) 00 70 17 01 70 17 02 70 17 03 70 17 22

   (ウ) 00 28 23 01 28 23 02 28 23 03 28 23 32



8. 上記の(ア)～(ウ)を入力してから、「Send」ボタンを押してコマンドを送信します。正しくサーボモーターが動作するか確認します。

9. 数値を変えてみて動作検証もしてみてください。もしうまく動かない場合は（1）ポートを開いていない、（2）HEXで入力していない、（3）CHECKSUMの計算が間違っているなどの問題がないか再確認してください。


chinoken