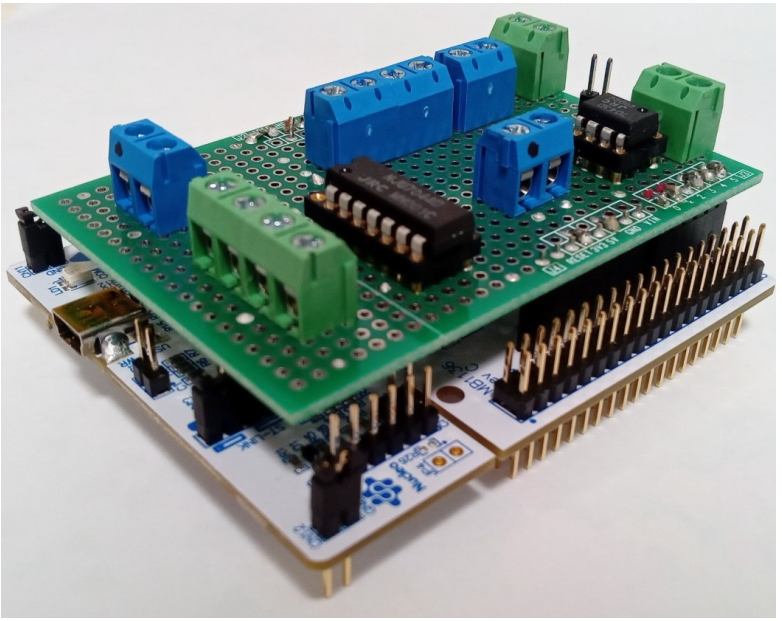
第1部第3章

Analog Discovery風 計測ステーションの製作

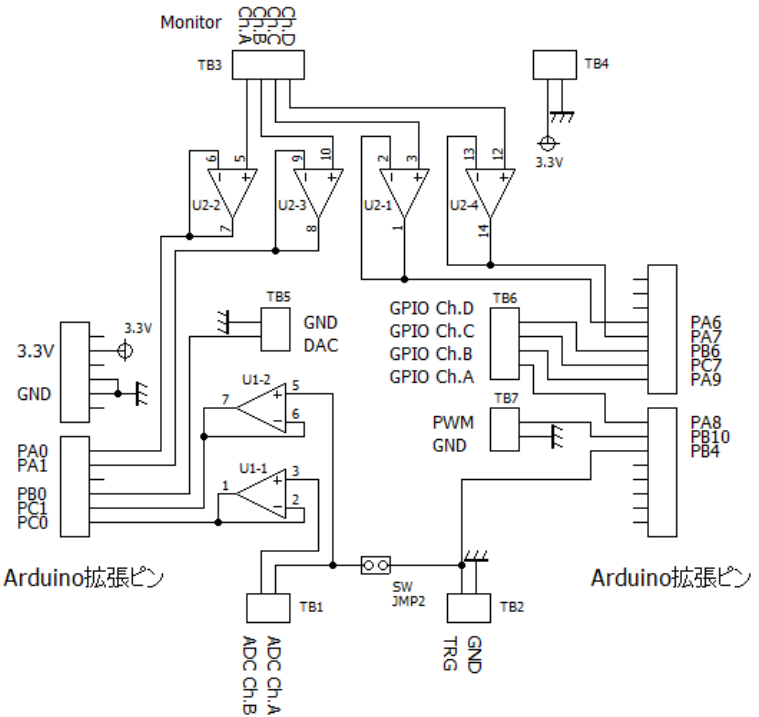


取り扱い説明書

【目次】

1. ハードウェア組立

拡張基板側の回路図になります。
AD変換のためのバッファアンプのみ外付けします。
拡張基板を無くして、Arduino 互換ピンから直接入出力することも可能です。その場合、計測対象物の出力インピーダンスが大きいと、正確なオシロスコープ計測ができません。

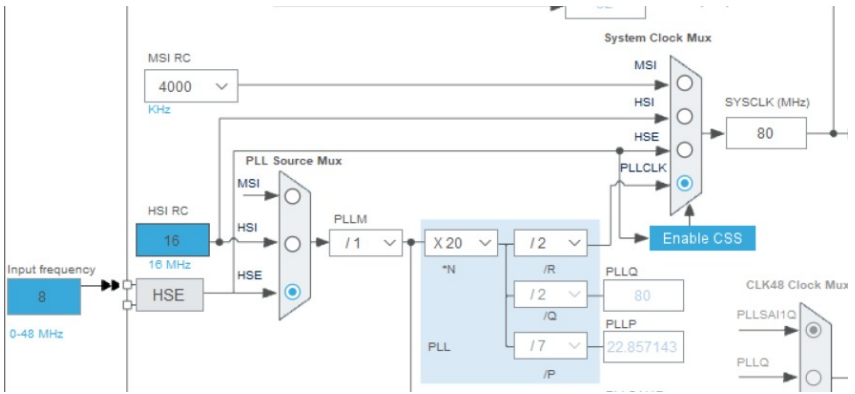


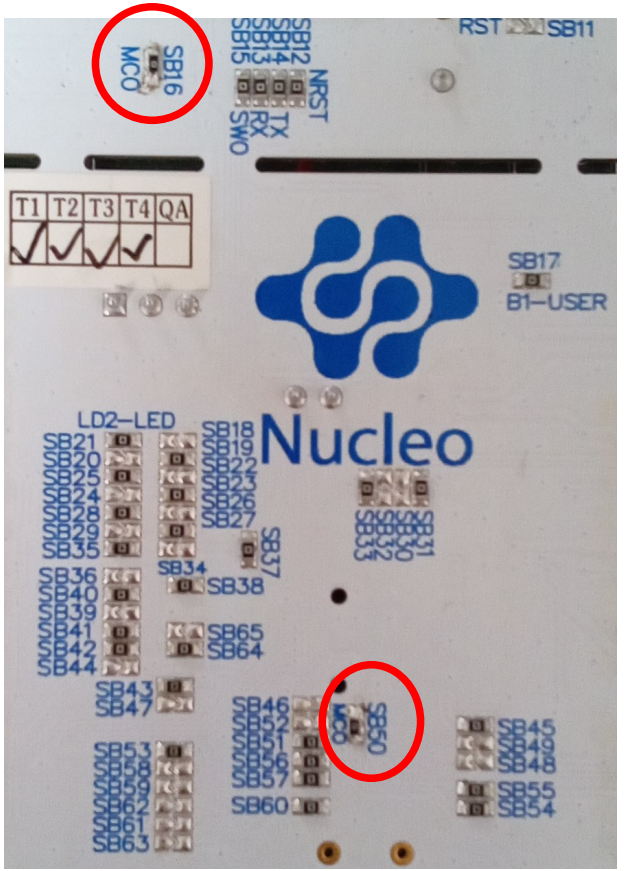
Pin名	機能名	ペリフェラル名
①2ch オシロスコープ入力		
PC1	ADC Ch. A	ADC1 IN2
PC0	ADC Ch. B	ADC3 IN1
②1chトリガ入力		
PB4	TRG	COMP2 INP
③4ch 電圧計測入力		
PA0	Monitor Ch. A	ADC2 IN5
PA1	Monitor Ch. B	ADC2 IN6
PA6	Monitor Ch. C	ADC2 IN11
PA7	Monitor Ch. D	ADC2 IN12
④1ch 任意信号発生器出力		
PB0	DAC Ch. A	DAC1 OUT2
⑤4ch GPIO出力		
PA8	GPIO Ch. A	GPIO
PA9	GPIO Ch. B	GPIO
PC7	GPIO Ch. C	GPIO
PB6	GPIO Ch. D	GPIO
⑥1ch PWM出力		
PB10	PWM	TIM2 CH3

Nucleoボードは、NUCLEO-L476RG を使用します。

基準クロック信号(8MHz) は、Nucleoデバッガボードから出力される、水晶振動子に基づく正確な基準クロックを利用します。Nucloe ボード裏の、SB16 と SB50部番に、0Ω抵抗(又はショート) を追加実装します。本改造を行いたくない方は、17ページのプログラム改造を行います。

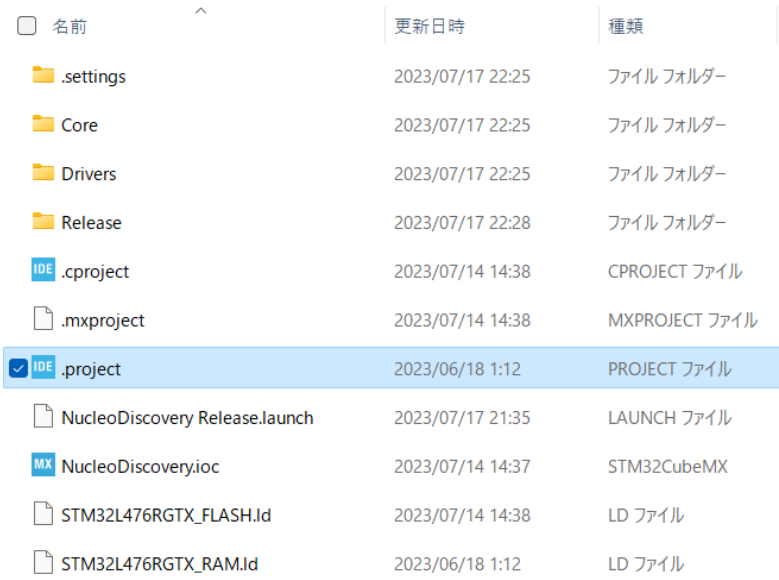
デバッガボードからの MCO信号



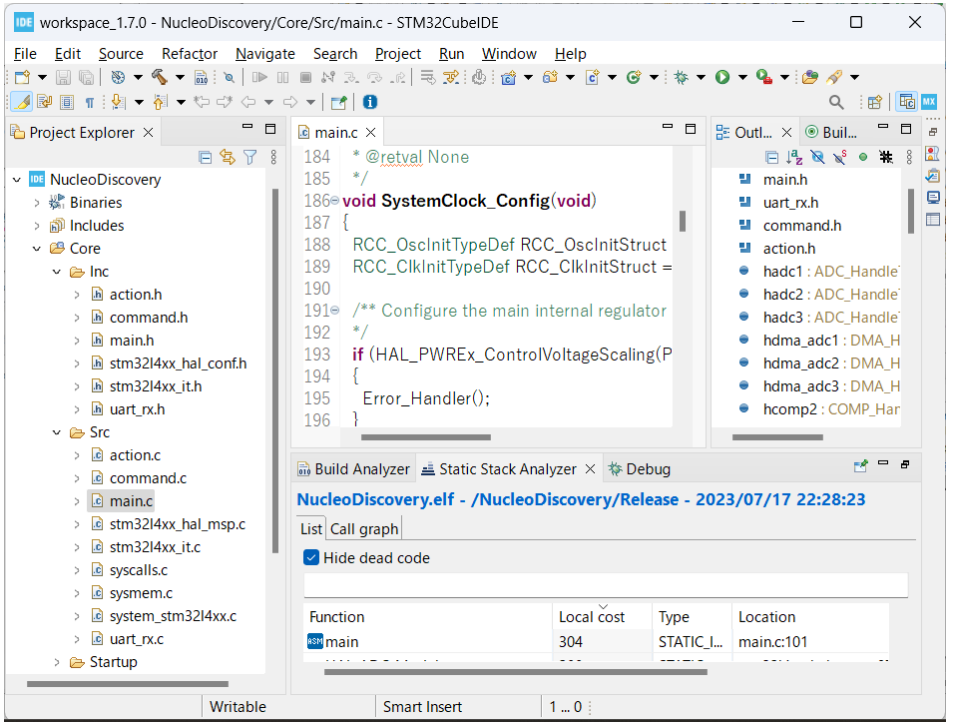


2. プロジェクトの立上げ

プロジェクトファイル一式が同梱されています。
.project ファイルをダブルクリックして、STM32CubeIDE を起動します。すぐにビルドできる状態になっています。
プログラムの修正が不要な方は、そのままビルドして、Nucleoボードに書込みます。



起動



3. 付属の python APIライブラリの使い方

Nucleoボードと通信して、計測制御するためのライブラリは「DiscoveryBase.py」です。
FFT演算に便利な補助ライブラリは「utility.py」です。FFT演算する場合にのみ使用します。

この2つのライブラリを使った、ユーザアプリケーションの事例が、「samples.py」です。様々な用例を例示しています。

【動作環境】

・Windows11

【必要な pythonライブラリ】

・pySerial
・numpy (utility.py で FFT する場合のみ必要)

【ライブラリの初期 / 終了処理】

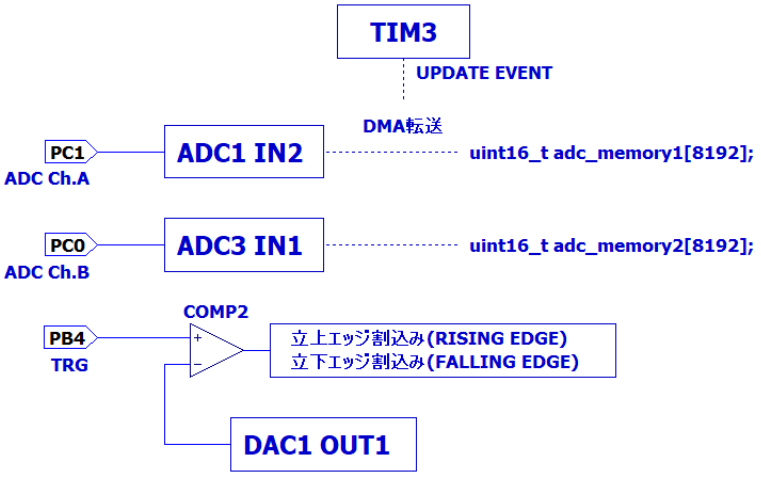
```
from DiscoveryBase import DiscoveryBase
import utility    # FFT演算しない場合は不要

discovery = DiscoveryBase( )
# openメソッドで、Nucleoボードと通信接続
discovery.open( )

# discovery オブジェクトを通じて、計測処理を実行
# 次ページ以降に、計測処理を行う DiscoveryBase クラスメソッドを紹介

# 計測終わったら、closeメソッドで Nucleoボードとの通信切断して終了
discovery.close( )
```

3.1. 2chオシロスコープ入力



オシロスコープは2ch入力とは別に、トリガ入力があります。
サンプリング開始は、「ソフトウェアトリガ」、「トリガ入力立上りエッジ」、「トリガ入力立下りエッジ」 の3種類から選択できます。

サンプリングデータは、マイコン内部のADC波形メモリに格納されます。各ch毎に、8192点の固定サンプル点になります。ADC波形メモリは、次にサンプリング開始されるまで、保持し続けます。サンプリング周波数は、最大 4 [MHz] とします。

以下のパラメータが調整できます。

・トリガ閾値電圧
・サンプリング間隔時間 (周波数)

※エッジトリガの場合、真のエッジからサンプリング開始まで、
22 [usec] の遅延があります。但し、遅延量は安定しているので、
同期という意味では、役割を果たします。今後の改善課題です。

●オシロスコープサンプリング条件設定1

```
def set_adc_sampling_time(sampling_time:float)->float:
```

オシロスコープのサンプリング間隔時間 [sec] を設定。
姉妹関数として、サンプリング周波数を設定する
set_adc_sampling_freqメソッドがある。

引数
sampling_time : サンプリング間隔時間 [sec] (0.25e-6以上)

戻値
実際に設定されたサンプリング間隔時間 [sec]
失敗時は 0

●オシロスコープサンプリング条件設定2

```
def set_adc_sampling_freq(sampling_freq:float)->float:
```

オシロスコープのサンプリング周波数 [Hz] を設定。

引数
sampling_freq : サンプリング周波数 [Hz] (4e6 以下)

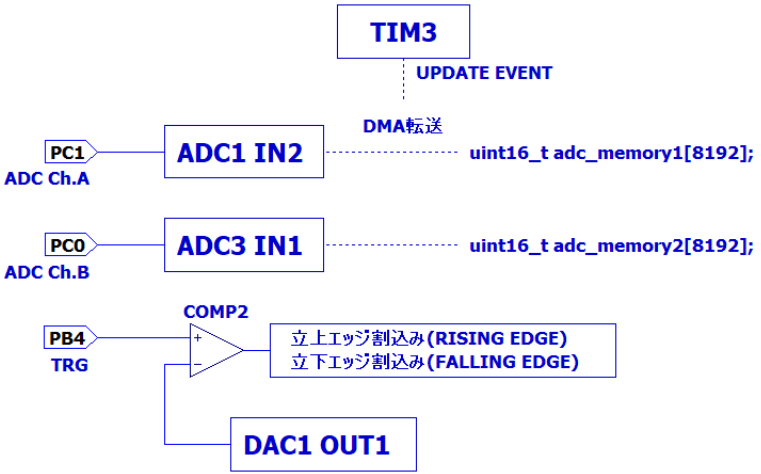
戻値
実際に設定されたサンプリング周波数 [Hz]
失敗時は 0

●トリガ閾値電圧設定 (DAC1 OUT1出力電圧設定)

```
def set_trigger(trigger_voltage:float)->bool:
```

引数
trigger_voltage : トリガ閾値の電圧[V] を設定 (0 - 3.3)

戻値
正常に処理されたら True / 失敗時は False



●オシロスコープサンプリング強制停止

def stop_adc( )->bool:

オシロスコープのサンプリングを強制停止。

戻値
正常に処理されたら True / 失敗時は False

●オシロスコープサンプリング状態取得

def get_adc_state( )->int:

現在のオシロスコープのサンプリング状態を取得

戻値
DiscoveryBase.ADC_STATE_STOP (=0) サンプリング停止状態
DiscoveryBase.ADC_STATE_RISE (=1) 立上りエッジトリガ待ち状態
DiscoveryBase.ADC_STATE_FALL (=2) 立下りエッジトリガ待ち状態
DiscoveryBase.ADC_STATE_SAMPLING (=3) サンプリング中
DiscoveryBase.ADC_STATE_COMPLETE (=4) サンプリング完了

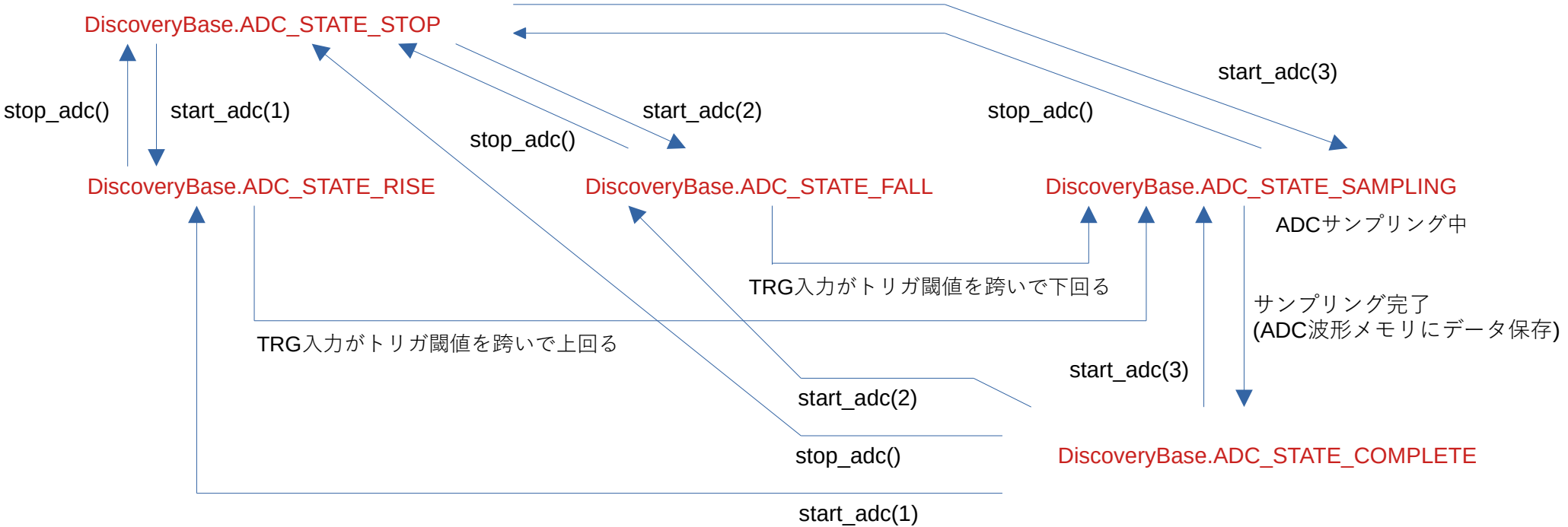
●オシロスコープサンプリング開始

def start_adc(trigger:int)->bool:

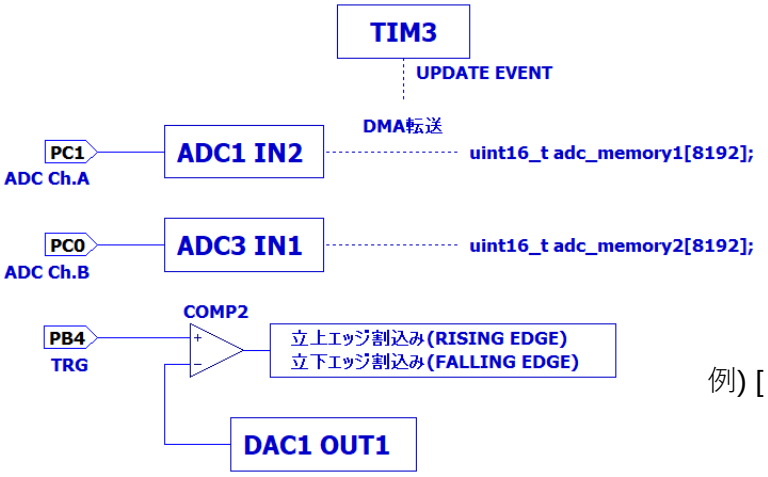
引数
trigger :
DiscoveryBase.ADC_TRIGGER_RISE (=1) TRG入力が、トリガ電圧を**跨いで**上回った時点でサンプリング開始する
DiscoveryBase.ADC_TRIGGER_FALL (=2) TRG入力が、トリガ電圧を**跨いで**下回った時点でサンプリング開始する
DiscoveryBase.ADC_TRIGGER_NONE (=3) すぐにサンプリング開始

戻値
正常に処理されたら True / 失敗時は False

オシロスコープの状態遷移 : get_adc_state( )



※ ADC波形メモリの中身は、DiscoveryBase.ADC_STATE_SAMPLING (=3) 状態になって、サンプリング開始されるまで、前回サンプリングされたデータが保持され続けています。
正しくデータサンプリングするには、start_adc() メソッドを呼び出してから、サンプリングが実施された後、状態がDiscoveryBase.ADC_STATE_COMPLETE (=4) 状態になるまで待ちます。



●オシロスコープサンプリングデータの取得

```
def get_adc_memory( )->list:
```

2次元リストで、Ch.A / B のサンプリングデータリストを取得する。
取得値は電圧(0-3.3[V]) に換算されている。
サンプリング点数は、各チャンネル 8192点固定。

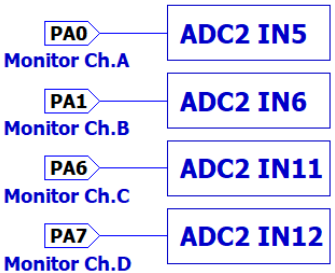
戻値
　2次元リスト。失敗すると空リスト。

例) [[1.235, 1.253, 1.321, 1.400, 2.513, 1.234 ...] , [2.165, 1.902, 1.034, 2.212, 1.212, 1.304 ...]]
　　<------- Ch.A の時系列データ8192点 -------> <------- Ch.B の時系列データ 8192点 ------->

サンプル事例

```
import  time
# サンプリング希望周波数設定 23k [sps]
setting_freq = discovery.set_adc_sampling_freq (23000)
print( 'Ready to ADC: sampling frequency {} [Hz]\n'.format (setting_freq) )
# トリガ閾値電圧を 1 [V] に設定
discovery.set_trigger (1)
# トリガ入力に何かの信号を入れて、同電圧がトリガ閾値電圧を跨いで超過したら
# オシロスコープサンプリングを開始する。
discovery.start_adc (DiscoveryBase.ADC_TRIGGER_RISE)
# サンプリング完了するまで待つ
while discovery.get_adc_state ( ) != DiscoveryBase.ADC_STATE_COMPLETE:
    time.sleep(0.1)
# 各チャンネル 8192要素の電圧データを取得
chAdata, chBdata = discovery.get_adc_memory( )
```

3.2. 4ch電圧計測入力



●モニタ電圧取得

```
def get_monitor( )->list:
```

4chのモニタ電圧値[V] をリストで即時取得

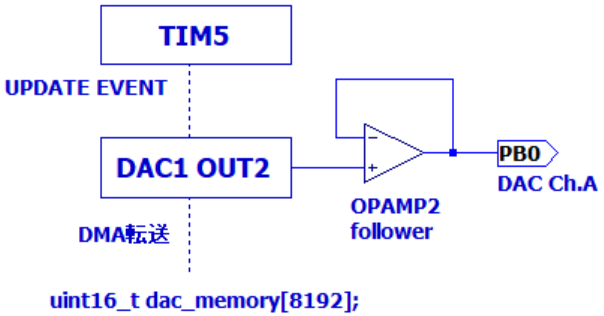
戻値

4要素の電圧値[V] リスト。Ch. A/B/C/D の順。
失敗時は空リスト

サンプル事例

```
sampling = discovery.get_monitor ( )
print( 'Ch.A {} [V]\n'.format (sampling[0]) )
print( 'Ch.B {} [V]\n'.format (sampling[1]) )
print( 'Ch.C {} [V]\n'.format (sampling[2]) )
print( 'Ch.D {} [V]\n'.format (sampling[3]) )
```

3.3. 1ch 任意信号発生器出力



任意信号発生器は、DAC波形メモリの電圧波形を、繰返しリピート出力します。DAC波形メモリの点数は 8192点です。波形の更新時間間隔は、最速 0.5 [usec] が実用限界です。

出力OFF時は、別途設定のOFF時電圧を保持します。OFF時電圧は、任意タイミングで変更できるので、非周期的な可変電圧出力用途に最適です。

以下のパラメータが調整できます。

・波形メモリの更新間隔時間 (周波数)
・波形メモリの開始 / 終了インデックス
・繰返し回数 (無限も可能)
・出力OFF時の静止電圧値
・出力のON / OFF

●任意信号発生器出力の正弦波波形設定

```
def set_dac_sinwave(refresh_freq:float, point:int, bias:float, amplitude:float,
                    phase:float=0, repeat_times:int=0 )->float:
```

正弦波を出力するための準備を行う。出力 ON / OFF は enable_dac( ), disable_dac( ) メソッドを別途用いる。
本メソッドは内部で、set_dac_memory( ) , set_dac( ) を自動設定している。

引数
refresh_freq : 出力したい正弦波の周波数 [Hz]
point : 正弦波1周期のサンプリング点数。大きいほど波形が滑らかになる。(2 - 8192)
bias : 正弦波のオフセット電圧 [V] (0 - 3.3)
amplitude : 正弦波の振幅電圧 [V] (0 - 3.3)
phase : 正弦波の初期位相 [度] (0 - 360)
※正弦波 = bias + amplitude * sin(refresh_freq[Hz] + phase[度]) [V]
repeat_times : 繰返し出力したい波数。0は無限繰返し。(0 - 65535)
※出力OFF 時の電圧は、波形メモリ初期点の電圧になる。

戻値
実際に設定された正弦波の周波数 [Hz]
失敗時は 0

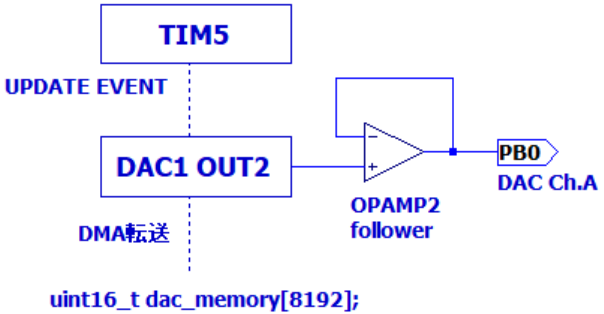
●任意信号発生器出力のDAC波形メモリ設定

```
def set_dac_memory(voltage_list:list)->bool:
```

DAC波形メモリへ、データ転送する。任意信号発生器は、DAC波形メモリの電圧を順次更新しながら、任意波形を出力する。最大メモリ点数は 8192点。

引数
voltage_list : 電圧値の波形リスト [V] (要素数8192以下の、0-3.3数値リスト)
要素数が8192未満の場合は、残りの波形メモリはゼロになる。

戻値
正常に処理されたら True / 失敗時は False



●任意信号発生器出力ON

```
def enable_dac( )->bool:
```

波形再生を ON にする。

戻値
　正常に処理されたら True / 失敗時は False

●任意信号発生器出力OFF

```
def disable_dac( )->bool:
```

波形再生を OFF にする。別途設定のOFF時電圧が出力される。

戻値
　正常に処理されたら True / 失敗時は False

●任意信号発生器出力の波形生成条件設定

```
def set_dac(refresh_time:float, start_index:int, stop_index:float,
            repeat_times:int=0, off_voltage:float=0 )->float:
```

DAC波形メモリの再生条件を設定する。

引数
　refresh_time : 波形メモリの更新間隔時間 [sec] 0.5e-6以上
　start_index : 波形メモリからの波形始点インデックス番号 (0 - 8191)
　stop_index : 波形メモリからの波形終点インデックス番号 (0 - 8191)
　repeat_times : 繰返し出力したい波数。0は無限繰返し。(0 - 65535)
　off_voltage : 出力OFF 時の静止電圧値 (0 - 3.3)

戻値
　実際に設定された波形メモリの更新間隔時間 [sec]
　失敗時は 0

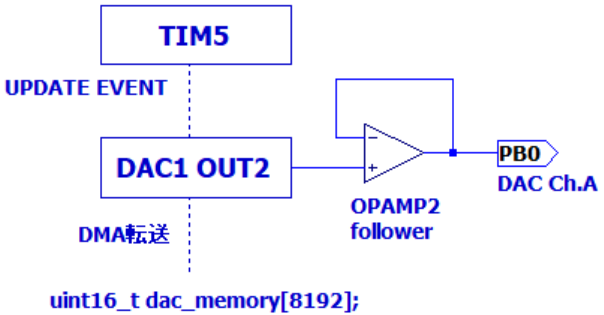
●任意信号発生器出力のOFF時の静止電圧設定

```
def set_dac_off_voltage(off_voltage:float=0 )->bool:
```

出力OFF時の静止電圧値を設定する。出力OFF時ならば、即時反映。

引数
　off_voltage : 出力OFF 時の静止電圧値 (0 - 3.3)

戻値
　正常に処理されたら True / 失敗時は False

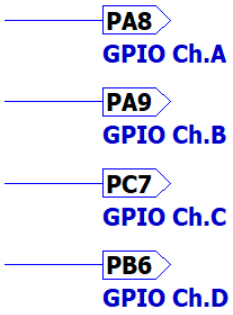


サンプル事例

```
import time

desired_freq = 1.36e3   # 1.36kHz の正弦波を所望
# 1.36kHz, 正弦波1周期64点表現, 1.2Vバイアス, 0.73V振幅, 初期位相180度, 無限繰返し
sine_freq = discovery.set_dac_sinwave(desired_freq, 64, 1.2, 0.73, 180, 0)
print( 'Ready to output {} [Hz] sine wave.\n'.format(sine_freq) )
# 任意信号波形を5秒間出力 ON
discovery.enable_dac ( )
time.sleep(5)
# 任意信号波形出力 OFF ( OFF時は 1.2V で静止する = 波形メモリのインデックス0点 )
discovery.disable_dac ( )
time.sleep(2)
# 2秒後に 3V出力
discovery.set_dac_off_voltage(3.0)
```

3.4. 4ch GPIO出力



● GPIO出力論理値を設定

```
def set_gpio(chA:int, chB:int, chC:int, chD:int)->bool:
```

引数

chA , chB , chC , chD 各チャンネル設定論理値

0 : Low

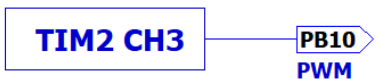
1 : High

それ以外 : 現在論理値を保持

戻値

正常に処理されたら True / 失敗時は False

3.5. 1ch PWM出力



● PWM出力条件設定

```
def set_pwm(freq:float, duty:float)->tuple:
```

引数

freq : PWM周波数 [Hz]

duty : Duty比 (0.0 - 1.0)

戻値

2要素のタプル

1要素目は、実際に設定された PWM 周波数 [Hz]

2要素目は、実際に設定された PWM Duty比

失敗時は空タプル

●PWM出力ON

```
def enable_pwm( )->bool:
```

戻値

正常に処理されたら True / 失敗時は False

●PWM出力OFF

```
def disable_pwm( )->bool:
```

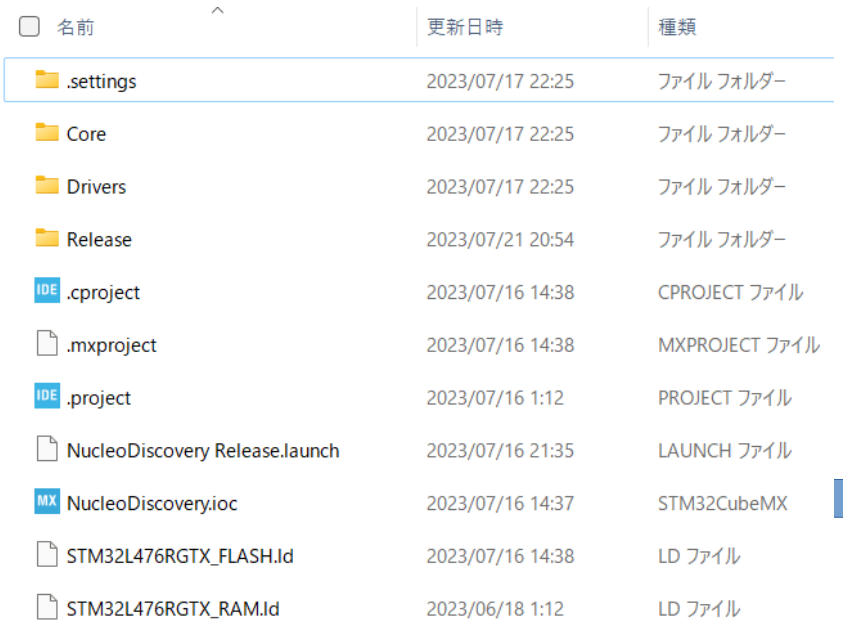
戻値

正常に処理されたら True / 失敗時は False

4. プログラム改造手順

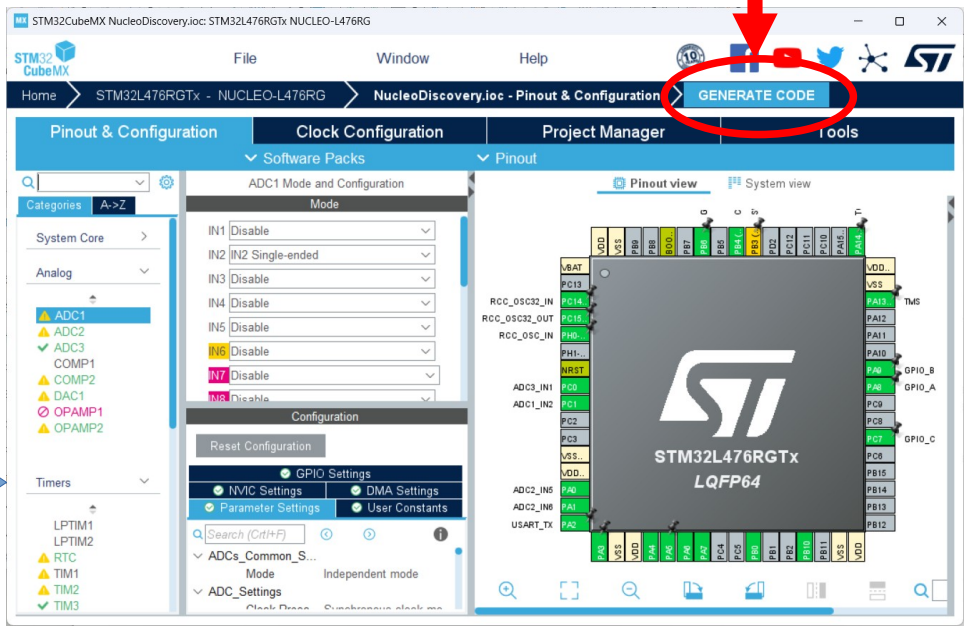
まずは、必要なペリフェラルの選択、パラメータ設定、クロックツリー設定を行います。ペリフェラルの機能追加、クロックツリー修正の必要がない方は、本ページは飛ばします。

NucleoDiscovery.ioc をダブルクリックして、STM32CubeMX を起動します。必要な修正が終わりましたら、 GENERATE CODE ボタンを押して、コード自動生成します。



起動

コード自動生成

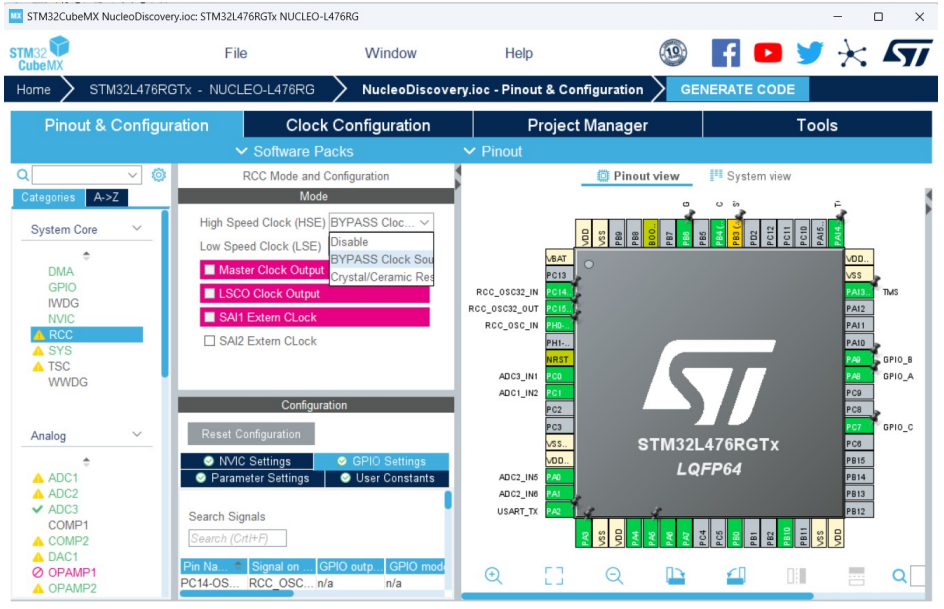


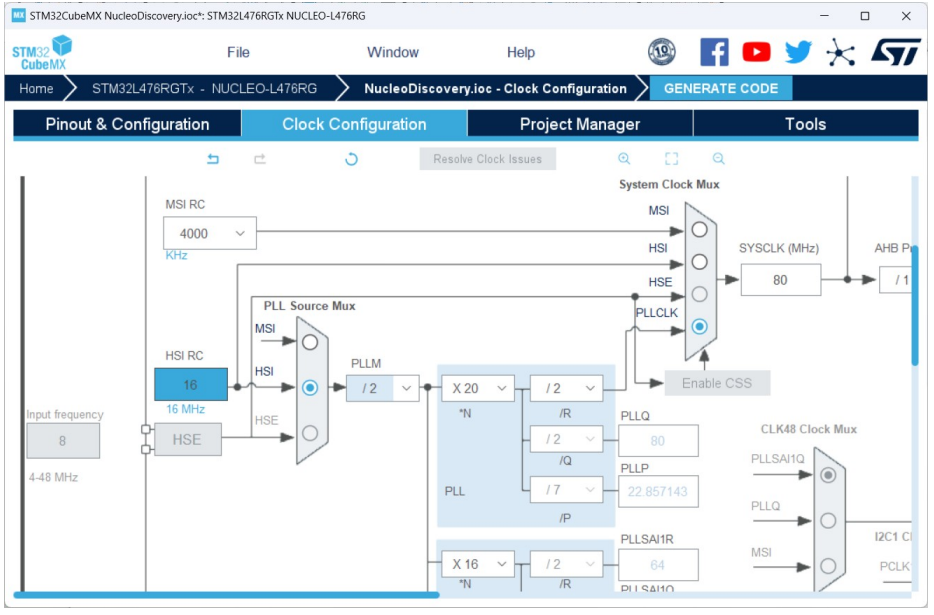
<備考>

High Speed Clock をマイコン内蔵クロックに切り替えたい場合も、STM32CubeMX による設定変更が必要です。 4ページにある Nucleo ボードの改造を行わない場合は、マイコン内蔵クロックを使用します。

Pinout & Configuration タブの、ペリフェラルの RCC を選択し、High Speed Clock (HSE) を Disable にします。

Clock Configuration タブの、PLL Source Mux を HSI にして、PLLM を /2 にします。最後に、GENARATE CODEボタンで、ソースコード自動生成してください。

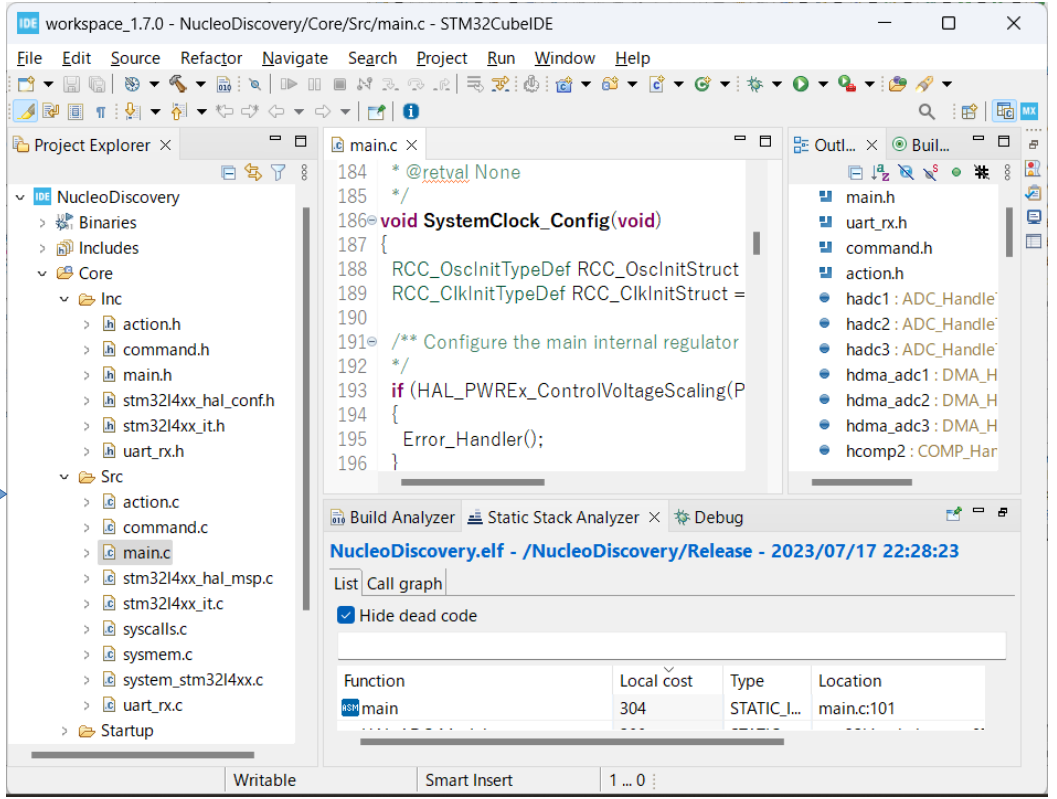




ソースコードの変更は、STM32CubeIDE で行います。

名前	更新日時	種類
.settings	2023/07/17 22:25	ファイル フォルダー
Core	2023/07/17 22:25	ファイル フォルダー
Drivers	2023/07/17 22:25	ファイル フォルダー
Release	2023/07/21 20:54	ファイル フォルダー
.cproject	2023/07/16 14:38	CPROJECT ファイル
.mxproject	2023/07/16 14:38	MXPROJECT ファイル
.project	2023/07/16 1:12	PROJECT ファイル
NucleoDiscovery Release.launch	2023/07/16 21:35	LAUNCH ファイル
NucleoDiscovery.ioc	2023/07/16 14:37	STM32CubeMX
STM32L476RGTX_FLASH.ld	2023/07/16 14:38	LD ファイル
STM32L476RGTX_RAM.ld	2023/06/18 1:12	LD ファイル

起動



5. プログラムの構造

5.1. main.c

現在の main.c 構成です。外部モジュール 「uart_rx.c」「command.c」「action.c」を参照しています。
uart_rx.c は UART通信の受信をバックグラウンド処理しているため、変更する必要はないでしょう。
機能追加する場合には、「command.c」「action.c」の修正が必要になります。

```
// UARTのコマンド受け付けはバックグラウンド(割込み)処理、uart_rx.c 参照
 uart_rx_init(&huart2);
 // ペリフェラルの操作を  action.c 内に委譲する、action.c 参照
 action_init(&hadc1, &hadc3, &hadc2, &hdma_adc1, &hdma_adc3, &hdma_adc2, &hcomp2,
                 &hdac1, &hdma_dac_ch2, &hopamp2, &htim2, &htim3, &htim4, &htim5, &huart2);

while (1)  {
        // バックグラウンドの  UART受信処理の結果確認、uart_rx.c 参照
        command_size = uart_rx_get_command(command_str, UART_RX_BUFFER_SIZE);
         // コマンド文字確定している場合
        if(command_size != 0){
                // コマンド文字列から、コマンド構造体に変換、command.c 参照
                command_analysis(command_str, command_size, &command);
                // コマンドを実行し、結果をUARTに返信する、action.c 参照
                action_command(&command);
                // 再びUART受信を開始する、uart_rx.c 参照
                uart_rx_set_empty();
        }
        // action.c 内のポーリング処理のために、main ループ内で一回は呼び出すこと
        action_polling();
}
```

5.2. command.c

UART受信された文字列から、コマンドを解析して、内部表現のコマンド構造体に格納する機能を持ちます。
新しいコマンドを追加した場合は、コマンド解析(パラメータ範囲チェック) して、コマンド構造体に格納します。修正すべき関数は、command_analysis 関数になります。

```
// コマンド構造体
typedef struct {
        uint8_t effective; // 0: 不当なコマンド, 1: 正当なコマンド
        uint8_t command_header;   // +, -
        uint8_t command_type;     // A-Z, a-z
        uint32_t params[COMMAND_PARAMETERS];
} CommandTypeDef;
```

コマンドの任意引数(最大32bit値) を格納
COMMAND_PARAMETERS はヘッダファイルで10と定義

5.3. action.c

ペリフェラルの操作を一手に引き受けます。
新しいペリフェラルを追加された場合は、action_init 関数を修正して、追加されたペリフェラルのハンドラを受け取ります。
新しいコマンドを追加した場合は、action_command 関数を修正して、コマンドに対する処理を追加します。引数のコマンド構造体を受けて、コマンドに応じた処理を行い、UART で戻値を返信します。