MCC を使ったイベント駆動型プログラミング

大全のアナログボードを使ってイベント駆動型プログラミングをしてみました。

PIC16F1778 と **MCC** を使いました。

MCC は一見どうでもいいような関数を出力します。どのように活用すれば開発者と同じ方向を向けるのでしょうか。

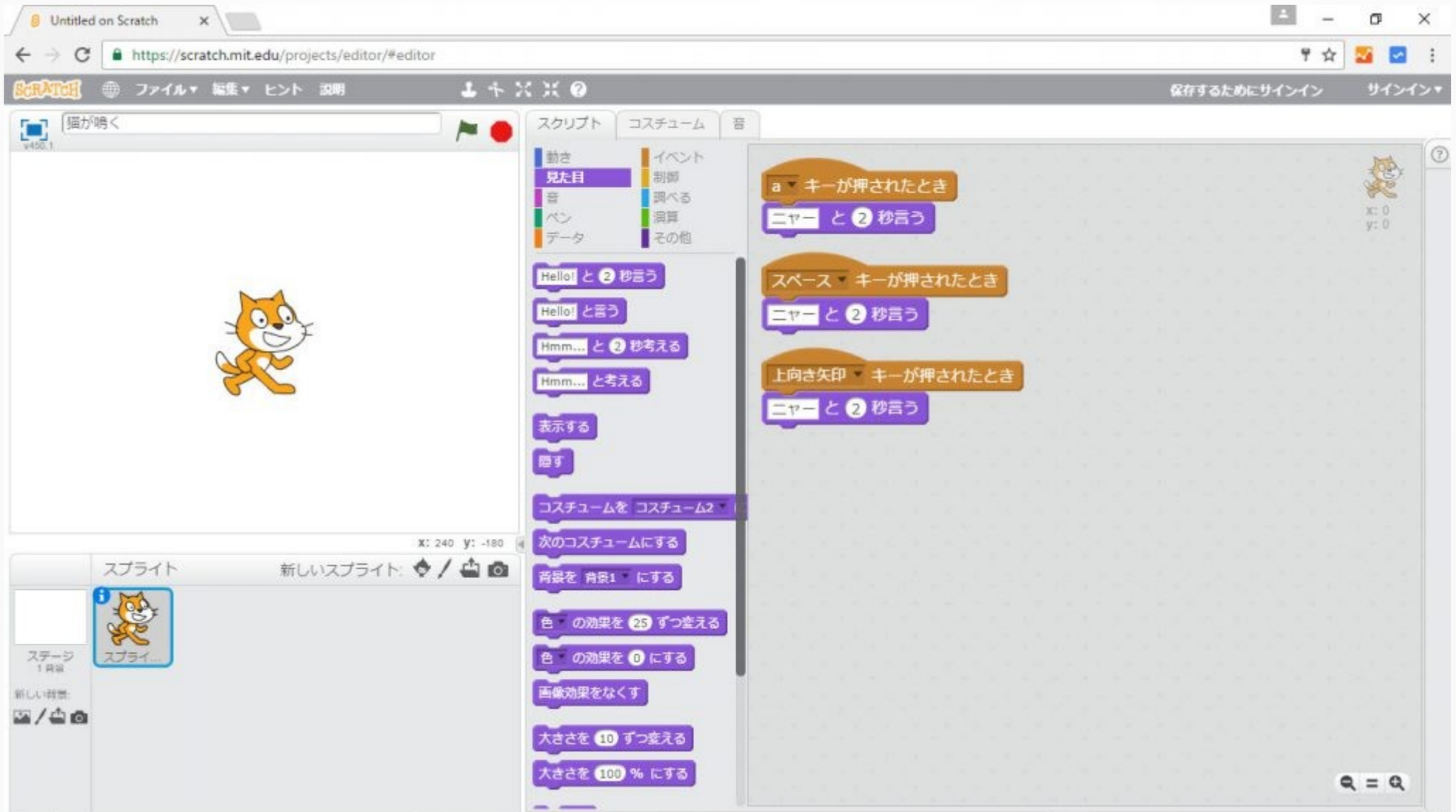
先日、**MCC** が吐き出す関数を眺めていましたら、これらがイベント駆動型のプログラミングに都合がよい構成になっているように見えてきましたので、
これらの関数のみを使用したプログラムを作りました。

ボリュームの開度で **LED** の点滅速度をコントロールします。
MCC の工程は **4** つで、メイン関数内に **7** 行書きます。

2019.04.27 福屋 浩幸

まず、イベント駆動型プログラミングに注目した理由

こいつがイベント駆動型で、小学生が使い始めるから…



MCC を使った LED の点滅とボリューム値の取り込み

MCC 出力関数不使用、点滅のみ

```
67      //INTERRUPT_PeripheralInterruptDisable();
68
69      LATBbits.LATB4 = 0;
70
71      while (1)
72      {
73          // Add your application code
74
75          LATBbits.LATB4 = 1;
76          __delay_ms(500);
77          LATBbits.LATB4 = 0;
78          __delay_ms(500);
79      }
80  }
81  /**
```

__delay_ms( ) を使用した従来の記述。
LATB4 が LED なのをコメントで追記必要。

遅延処理中にメインループが止まるのは、フロー駆動型プログラミングの限界。

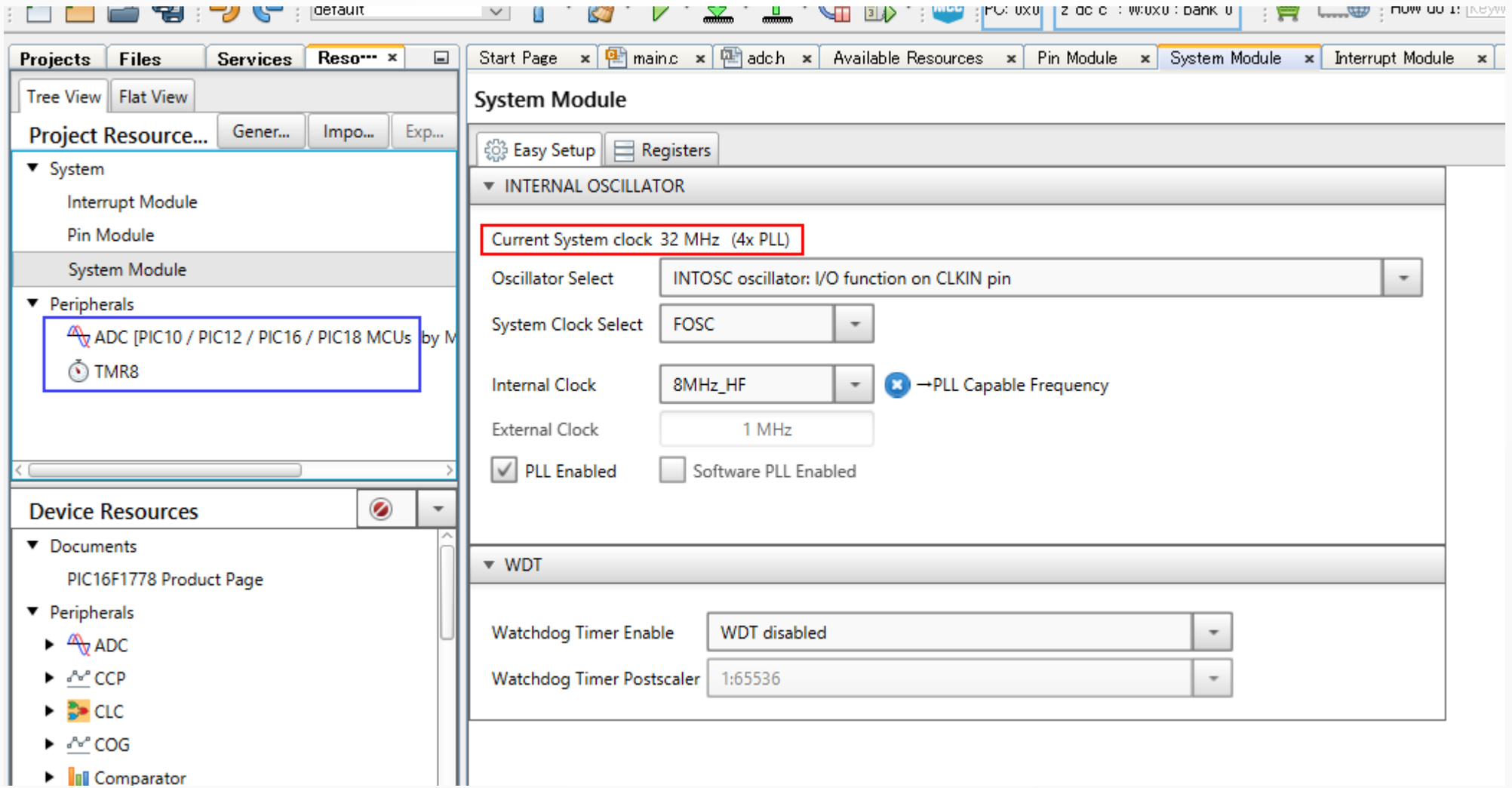
ボリューム値の読み込み付き

```
67      //INTERRUPT_PeripheralInterruptDisable();
68
69      LED1_SetLow();
70
71      while (1)
72      {
73          // Add your application code
74
75          if(TMR8_HasOverflowOccured()) {
76              LED1_Toggle();
77          }
78
79          if(ADC_IsConversionDone()) {
80              TMR8_Period8BitSet(ADRESH);
81          }
82      }
83  }
84  /**
```

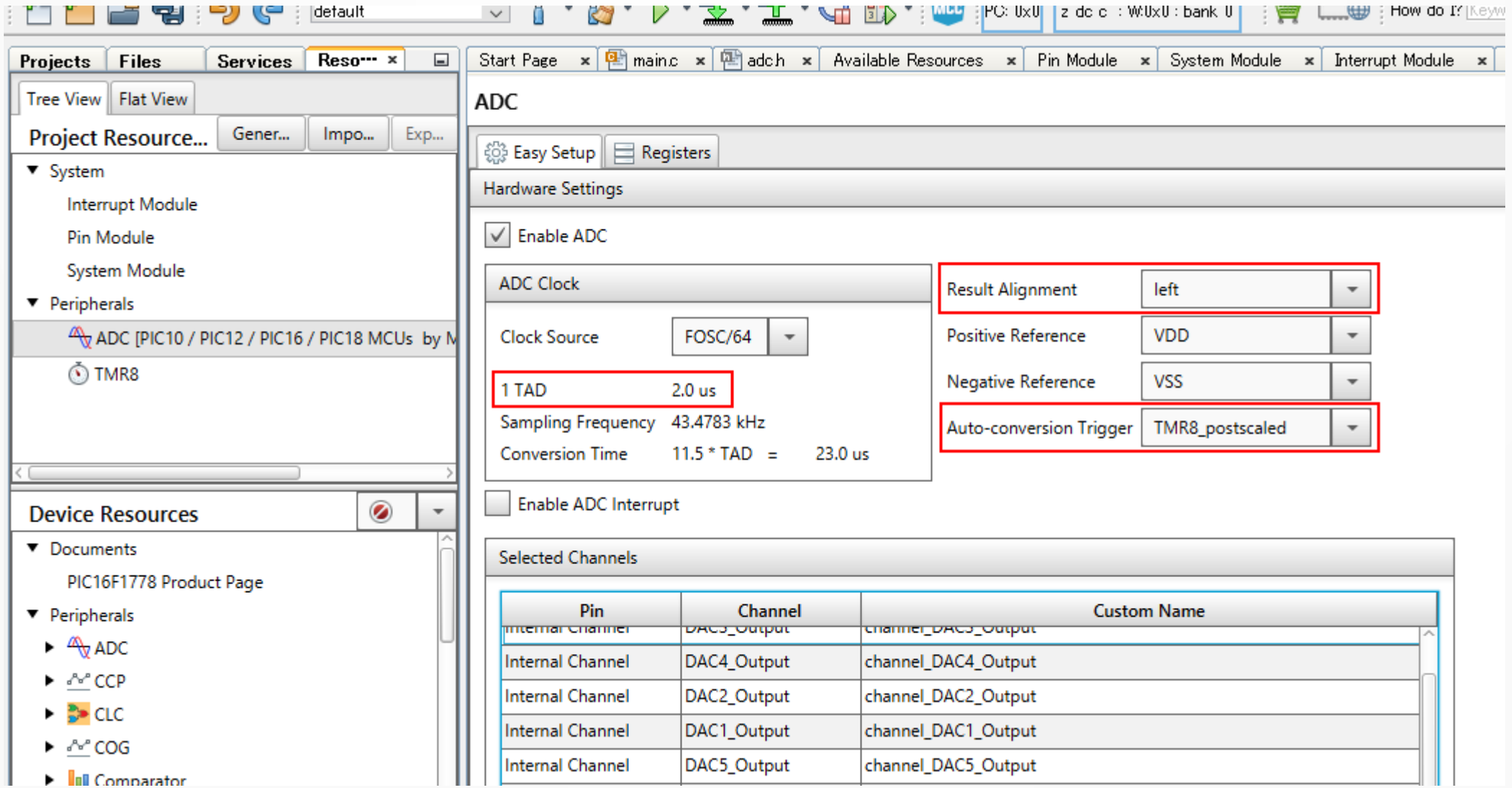
緑囲み部は MCC が出力する関数、よって自作関数は使っていませんし、作ってもいません。

イベント駆動型プログラミングではメインループが停止しない前提で設計されてタスクの追加が可能になります。

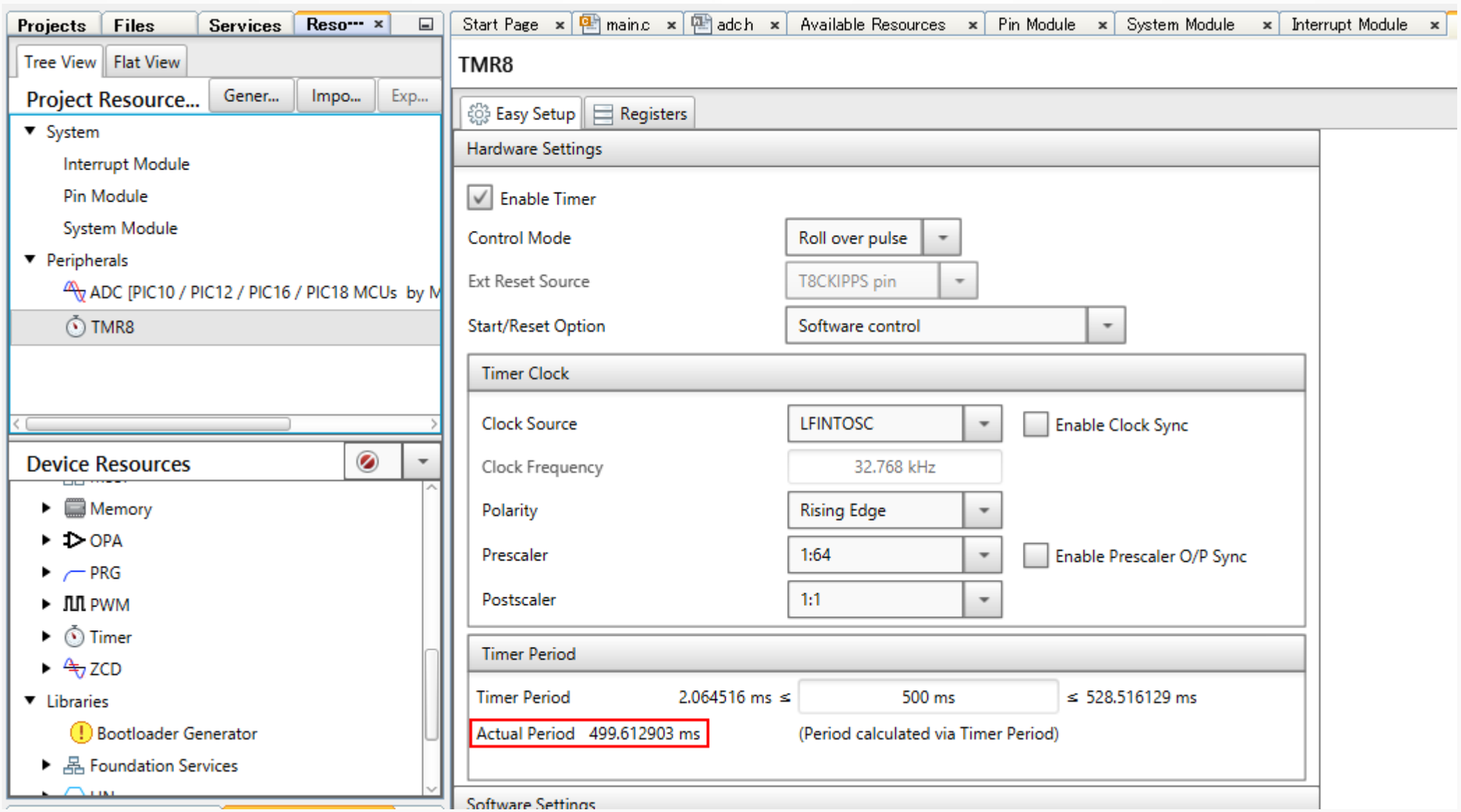
MCC 実際の操作 1


Projects
Files
Services
Reso···
Tree View
Flat View
Project Resource...
Gener...
Impo...
Exp...
System
Interrupt Module
Pin Module
System Module
Peripherals
ADC [PIC10 / PIC12 / PIC16 / PIC18 MCUs
TMR8
Device Resources
Documents
PIC16F1778 Product Page
Peripherals
ADC
CCP
CLC
COG
Comparator
Start Page
main.c
adc.h
Available Resources
Pin Module
System Module
Interrupt Module
System Module
Easy Setup
Registers
INTERNAL OSCILLATOR
Current System clock 32 MHz (4x PLL)
Oscillator Select
INTOSC oscillator: I/O function on CLKIN pin
System Clock Select
FOSC
Internal Clock
8MHz_HF
→PLL Capable Frequency
External Clock
1 MHz
PLL Enabled
Software PLL Enabled
WDT
Watchdog Timer Enable
WDT disabled
Watchdog Timer Postscaler
1:65536

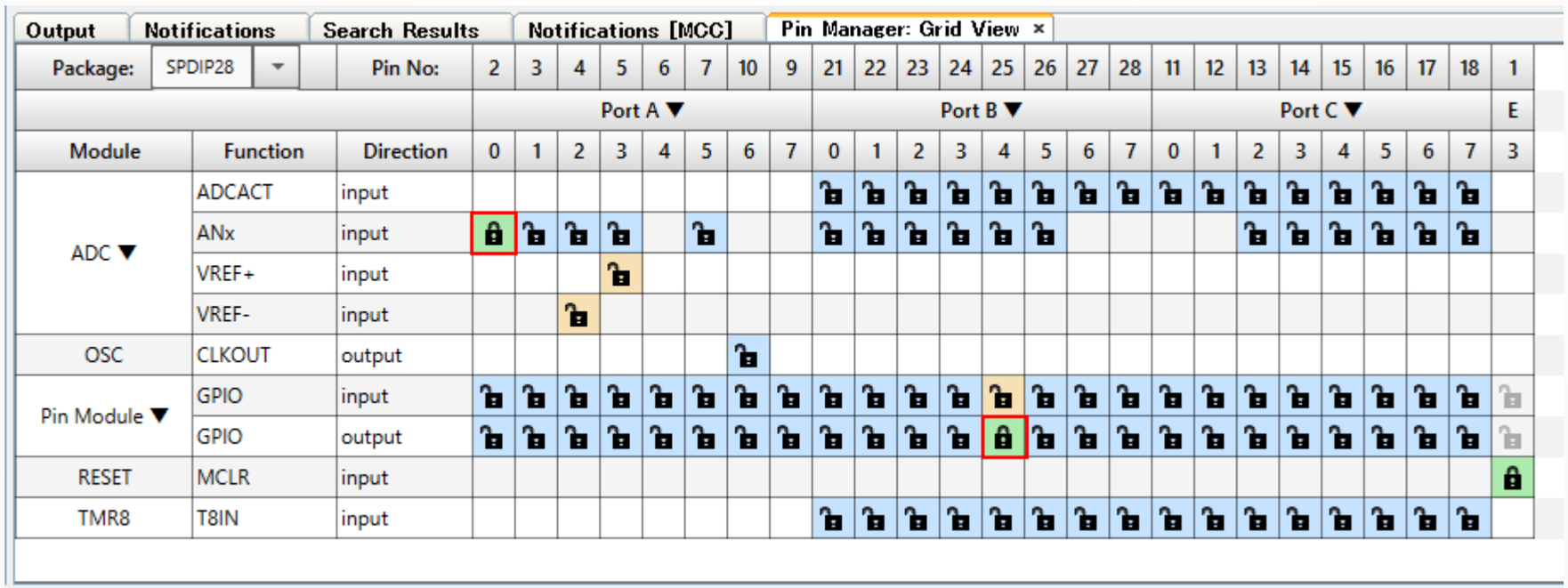
MCC 実際の操作 2

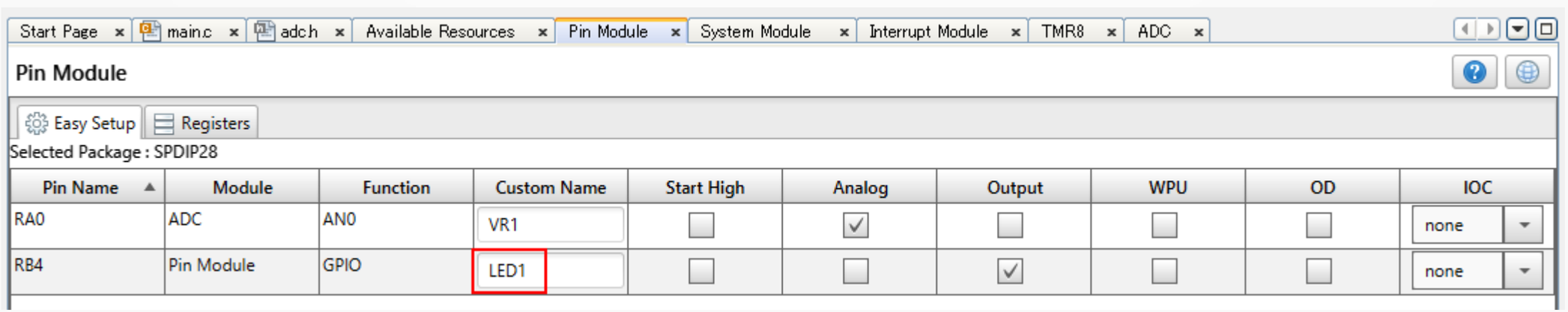

default
Projects
Files
Services
Reso···
Tree View
Flat View
Project Resource...
Gener...
Impo...
Exp...
System
Interrupt Module
Pin Module
System Module
Peripherals
ADC [PIC10 / PIC12 / PIC16 / PIC18 MCUs by M
TMR8
Device Resources
Documents
PIC16F1778 Product Page
Peripherals
ADC
CCP
CLC
COG
Comparator
Start Page
main.c
adc.h
Available Resources
Pin Module
System Module
Interrupt Module
ADC
Easy Setup
Registers
Hardware Settings
Enable ADC
ADC Clock
Clock Source
FOSC/64
1 TAD
2.0 us
Sampling Frequency
43.4783 kHz
Conversion Time
11.5 * TAD
=
23.0 us
Result Alignment
left
Positive Reference
VDD
Negative Reference
VSS
Auto-conversion Trigger
TMR8_postscaled
Enable ADC Interrupt
Selected Channels
Pin
Channel
Custom Name
Internal Channel
DAC4_Output
channel_DAC4_Output
Internal Channel
DAC2_Output
channel_DAC2_Output
Internal Channel
DAC1_Output
channel_DAC1_Output
Internal Channel
DAC5_Output
channel_DAC5_Output

MCC 実際の操作 3


Projects
Files
Services
Reso···
Tree View
Flat View
Project Resource...
Gener...
Impo...
Exp...
System
Interrupt Module
Pin Module
System Module
Peripherals
ADC [PIC10 / PIC12 / PIC16 / PIC18 MCUs by M
TMR8
Device Resources
Memory
OPA
PRG
PWM
Timer
ZCD
Libraries
Bootloader Generator
Foundation Services
Start Page
main.c
adc.h
Available Resources
Pin Module
System Module
Interrupt Module
TMR8
Easy Setup
Registers
Hardware Settings
Enable Timer
Control Mode
Roll over pulse
Ext Reset Source
T8CKIPPS pin
Start/Reset Option
Software control
Timer Clock
Clock Source
LFINTOSC
Enable Clock Sync
Clock Frequency
32.768 kHz
Polarity
Rising Edge
Prescaler
1:64
Enable Prescaler O/P Sync
Postscaler
1:1
Timer Period
Timer Period
2.064516 ms ≤
500 ms
≤ 528.516129 ms
Actual Period 499.612903 ms
(Period calculated via Timer Period)
Software Settings

MCC 実際の操作４


Output
Notifications
Search Results
Notifications [MCC]
Pin Manager: Grid View
Package: SPDIP28
Pin No: 2 3 4 5 6 7 10 9 21 22 23 24 25 26 27 28 11 12 13 14 15 16 17 18 1
Port A ▼
Port B ▼
Port C ▼
E
Module Function Direction 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 3
ADC ▼ ADCACT input
ANx input
VREF+ input
VREF- input
OSC CLKOUT output
Pin Module ▼ GPIO input
GPIO output
RESET MCLR input
TMR8 T8IN input


Start Page
main.c
adc.h
Available Resources
Pin Module
System Module
Interrupt Module
TMR8
ADC
Pin Module
Easy Setup
Registers
Selected Package : SPDIP28
Pin Name Module Function Custom Name Start High Analog Output WPU OD IOC
RA0 ADC AN0 VR1 none
RB4 Pin Module GPIO LED1 none

MCC が出力した関数を見に行く 1

Start Page | pin_manager.h

Source | History

```
87
88  // get/set LED1 aliases
89  #define LED1_TRIS               TRISBbits.TRISB4
90  #define LED1_LAT                LATBbits.LATB4
91  #define LED1_PORT               PORTBbits.RB4
92  #define LED1_WPU                WPUBbits.WPUB4
93  #define LED1_OD                 ODCONBbits.ODB4
94  #define LED1_ANS                ANSELBbits.ANSB4
95  #define LED1_SetHigh()          do { LATBbits.LATB4 = 1; } while(0)
96  #define LED1_SetLow()           do { LATBbits.LATB4 = 0; } while(0)
97  #define LED1_Toggle()           do { LATBbits.LATB4 = ~LATBbits.LATB4; } while(0)
98  #define LED1_GetValue()         PORTBbits.RB4
99  #define LED1_SetDigitalInput()  do { TRISBbits.TRISB4 = 1; } while(0)
100 #define LED1_SetDigitalOutput() do { TRISBbits.TRISB4 = 0; } while(0)
101 #define LED1_SetPullup()        do { WPUBbits.WPUB4 = 1; } while(0)
102 #define LED1_ResetPullup()      do { WPUBbits.WPUB4 = 0; } while(0)
103 #define LED1_SetPushPull()      do { ODCONBbits.ODB4 = 0; } while(0)
104 #define LED1_SetOpenDrain()     do { ODCONBbits.ODB4 = 1; } while(0)
105 #define LED1_SetAnalogMode()    do { ANSELBbits.ANSB4 = 1; } while(0)
106 #define LED1_SetDigitalMode()   do { ANSELBbits.ANSB4 = 0; } while(0)
107
```

ポートアクセス用の関数の実体はマクロなので、ヘッダファイル pin_manager.h に出力されます。 MCC で付けたポート名で関数が作られます。

自作の関数を減らすことがデバッグ時間の短縮に直結しますから、自作のスキルがあったとしても純正品を使うべきです。呼び出しコストが問題になるようなら、マイコンのグレードを上げて対処したほうがよい結果になる場合が多いです。

MCC が出力した関数を見に行く 2

Start Page | tmr8.c

Source | History

```
146
147   void TMR8_Period8BitSet(uint8_t periodVal)
148   {
149       PR8 = periodVal;
150   }
151
152   void TMR8_LoadPeriodRegister(uint8_t periodVal)
153   {
154       TMR8_Period8BitSet(periodVal);
155   }
156
157   bool TMR8_HasOverflowOccured(void)
158   {
159       // check if  overflow has occurred by checking the TMRIF bit
160       bool status = PIR4bits.TMR8IF;
161       if(status)
162       {
163           // Clearing IF flag.
164           PIR4bits.TMR8IF = 0;
165       }
166       return status;
167   }
```

動作を把握しやすい関数名を付けることはプログラミングの効率に響くので、すぐ下の別名関数 TMR8_LoadPeriodRegister( ) も出力しています。

割り込みを使わなくてもフラグは立つので、イベント駆動型プログラミングに TMR8_HasOverflowOccered を使うことは有効です。
フラグの状態を戻り値で返すと同時にフラグクリアも行うので、ユーザー側でクリアする手間が省けます。

ここで注目したいのは、「MCC は軽くてイベント駆動型プログラミングに使いやすい関数を出力する」ということです。

MCC が出力した関数を見に行く 3

```
93
94  void ADC_StartConversion()
95  {
96      // Start the conversion
97      ADCON0bits.GO = 1;
98  }
99
100
101 bool ADC_IsConversionDone()
102 {
103     // Start the conversion
104     return ((bool)(!ADCON0bits.GO));
105 }
106
```

AD 変換が終了すれば ADCON0bits.GO がゼロになるので、これをイベントとしてチェックします。

AD 変換開始のトリガは Timer8 のオーバーフロー時に自動的にかかるように MCC で設定したので、 ADC_StartConversion 関数は使う必要がなくなりました。

103 行目のコメントはおそらく 96 行目からコピーしてきて内容を書き換えるのを忘れたのでしょう。（MCC のコメントはしっかりチェックされていないことが多い）

一通り関数を見てきたので、最初のメインループに戻ってみます。

イベント駆動型プログラミングをする

```
67      //INTERRUPT_PeripheralInterruptDisable();
68
69      LED1_SetLow();
70
71      while (1)
72      {
73          // Add your application code
74
75          if(TMR8_HasOverflowOccured()) {
76              LED1_Toggle();
77          }
78
79          if(ADC_IsConversionDone()) {
80              TMR8_Period8BitSet(ADRESH);
81          }
82      }
83  }
84  /**
```

Timer8 がオーバーフローしたら
LED をスイッチする

AD 変換が終了したら
タイマー周期を変更する

Scratch のプログラミングパラダイム（基本的な枠組み）はイベント駆動型です。
今回の C プログラムを日本語で表現して色分けすると、こちらもイベント駆動型になっていることがわかります。
数年以内に小学校で始まるプログラミング教育ではおそらく Scratch が教材になるでしょう。
イベント駆動型のプログラミング教育がフロー駆動型よりも先になるのです。
C 言語はどのパラダイムでも記述のしかたで対応できるので、先々使ってもらえるプログラムを目指すならイベント駆動型で書くのがよいでしょう。

イベント駆動型プログラミングをするときの注意点

メインループを極力スムーズに回し、ループ時間のばらつきをなくすことが重要です。

長い遅延処理は書いてはいけません。

メインループで毎回走るようなタスクはなるべく書かず、タイマーで周期を作るようにします。

イベントを拾ったらタスクをスタートするフラグを立て、タスクはステートマシン化して 1 ループ 1 ステートの実行にします。タスク実行にタイムアウトを付加するのならタイマーを使います。

メインの 1 ループ時間が待てない処理には割り込みを使いますが、割り込み処理内ではレジスタ操作や値の取り込みだけを行い、時間を要する処理はメインループに回します。

良く知られて枯れた RTOS が存在しない 16F1 でイベント駆動型プログラムを作るのはそれなりに面倒ですが、 MCC をうまく使えばかなり楽ができると思います。