2022 度機械情報工学科 冬学期演習
ワンチップマイコン・組み込み開発

担当：中嶋 浩平
TA：情報理工学系研究科 知能機械情報学専攻 大学院生

■スケジュール　10 月 4 日・6 日

■演習時間　13:00〜16:40

■配布物の扱いについて　配布しているマイコンキットは注意深く取り扱うこと。ハードウェアが故障すると、ソフトウェアのバグとの区別や故障箇所の特定には非常に時間がかかる場合がある。マイコンボード（Arduino Nano）に触れる前には、机や窓などの金属部に触れて静電気を逃すなどすること。基板の部品やはんだ部分には触れないこと。ジャンパワイヤを使った配線時には短絡（ショート）に気をつけること。ショートを起こすと PC の USB ポートが壊れることがある。コネクタが壊れやすいので、USB ケーブルは真っ直ぐ抜き差しする。

■演習の成績評価基準

1. 演習時間中の演習への参加
2. TA による課題チェックおよび課題提出

■演習についての連絡先　`k-nakajima@isi.imi.i.u-tokyo.ac.jp`

目次

1 本演習の目的

マイコン（マイクロコントローラ、MCU: micro controller unit）は、家電などの電子機器の制御に使われているワンチップ化された小型の計算機である。パーソナルコンピュータやタブレットなどの汎用計算機と比較して、計算能力は小規模だが、小型・低価格である。デジタルカメラ、テレビ、自動車などのメカトロニクス機器それぞれの制御に特化されたコンピュータシステムは「組み込みシステム」と呼ばれ、そのためのソフトウェア開発を「組み込みソフトウェア開発」と呼ぶ。汎用計算機の上で動くソフトウェアと異なり、組み込みソフトウェアはハードウェアのふるまい・制約を強く意識した設計が求められる。組み込みシステムは小型・低価格であることが求められるため、マイコンが多用されている。一方で、組み込み機器の高度化に伴って、OS を持つ小型の汎用コンピュータがそのまま組み込みシステムに使われる例も増えている。IoT（Internet of Things）という言葉の流行に見られるように、あらゆる製品にコンピュータが内蔵され、製品のコンピュータネットワークへの接続が当たり前になっていく中で、組み込みシステムを理解することは重要である。

本演習の目的は組み込みソフトウェア開発の入門という位置づけで「ワンチップマイコン」の基本的な使い方を学ぶことである。ワンチップマイコンとは、CPU やメモリ、クロック発振器などのコンピュータとして必要な要素だけでなく、デジタル入出力、A/D 変換器、タイマ、通信インタフェースが 1 つのチップに収められた IC を指す。

本演習で扱うワンチップマイコンは Atmel 社の AVR である。AVR は広く流通しており、マイコンボード Arduino に採用されていることから関連情報が手に入りやすい安価なマイコンである。演習終了後も、興味をもった学生はぜひ自習してさらに活用してほしい。本演習では AVR マイコンを通じて以下のようなことを学ぶ。

- 「マイコン」で何ができるか理解し、基本的な使い方を習熟する。
- 組み込み開発の流れを体験し、クロスコンパイルの仕組みを理解する。
- レジスタ操作によるデジタル I/O やタイマー、割り込みの設定など基本機能の使い方を学ぶ。
- A/D 変換や PWM による波形生成など各種機能の使い方を学ぶ。
- ライブラリを利用してプログラムを簡素化する手法を理解する。

2 Atmel AVR 概要

Atmel AVR は，Atmel（アトメル）社が製造している RISC ベースの 8 ビットマイクロコントローラ製品シリーズである．2000 年初頭、8 ビットマイコンでは Microchip Technology（マイクロチップ・テクノロジー）社の PIC マイコンが電子工作に使うマイコンの主流だった。後発の AVR は充実したフリーのソフトウェア開発環境の提供によって徐々に認知されるようになり、マイコンボード Arduino への採用によって広く使われるようになった。2016 年、Atmel 社は Microchip Technology 社に買収された。

アセンブラを含んだ Atmel 社純正の統合開発環境（IDE: integrated development environment）が無償配布され，さらに Windows，Linux，Mac OS いずれの OS でも無料のコンパイラスイートである GCC が対応している．高級言語を使った組み込みソフトウェア開発で必要になる商用版のコンパイラやデバッガは一般的に高価であり、個人が手軽に購入できるものではないが、AVR マイコンではそれが無料で利用できる。電子工作愛好家、ホビイスト、あるいは Maker と呼ばれる人たちに人気があるのは、フリーの開発環境と、ウェ

ブ上の豊富な情報の蓄積による。

AVR には大きくわけて 2 つのカテゴリに分類される．mega シリーズと tiny シリーズである．前者は入出力ピン数やメモリ容量が大きいのに対し，後者は小型・低消費電力なシステムで有効と考えればよい．本演習では入手容易性の観点から ATmega328P を利用する．ATmega48, ATmega88, ATmega168 という型番と親戚関係にあるデバイスである．本資料はこれ以降 ATmega328P に特化した説明を行う．数ある AVR を網羅した説明は困難であるため，演習を取り組む上で重要と思われる情報のみを載せる。

2.1　Atmel AVR の内部構成と機能の概要

2.1.1　AVR CPU コアと AVR のメモリの構成

■CPU コア　AVR の CPU コアは図 1 のような構成になっている．算術演算回路（ALU: arithmetic logical unit），命令レジスタ（Instruction Register），命令デコーダ（Instruction Decoder），周辺回路とのやりとりをするためのデータバス（Data Bus, 8 ビット）から構成される．

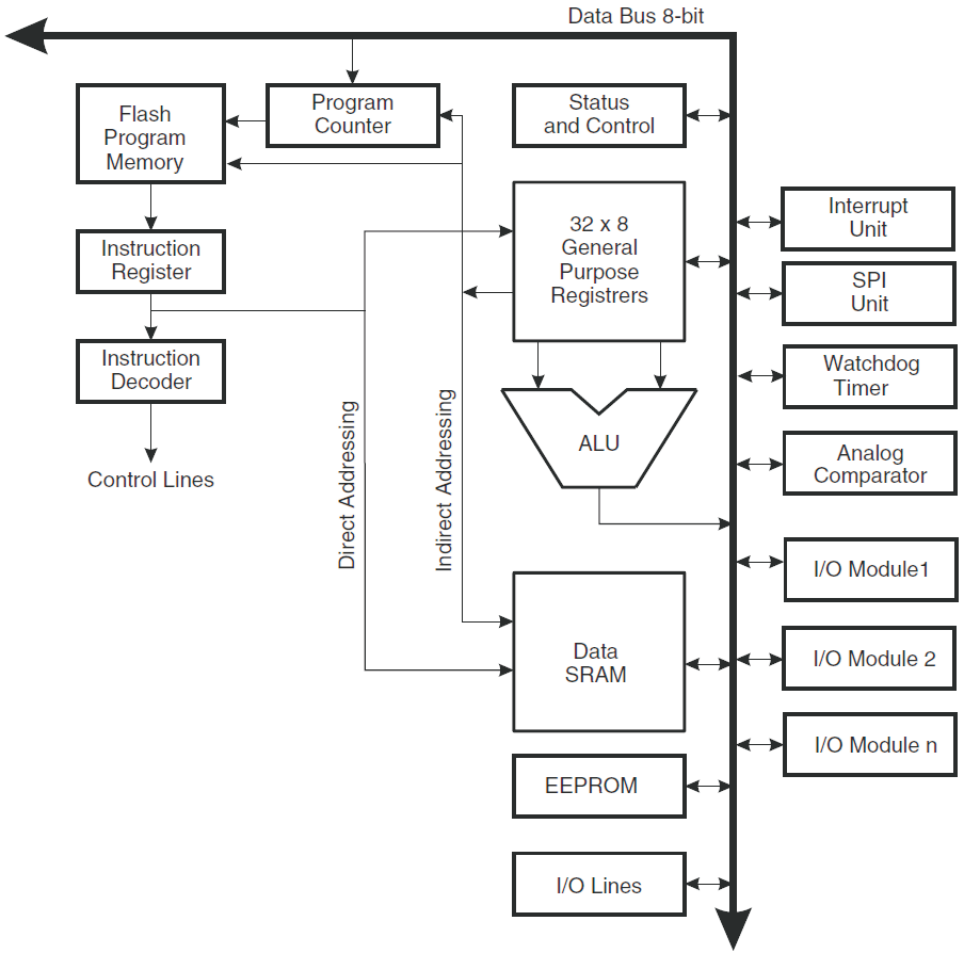


図 1　AVR CPU Core（Atmel 社データシート 8 ページより引用）

プログラムはフラッシュメモリ（Flash Program Memory）に格納されている．一方，プログラムの計算結

果を保持したりスタックとして使われたりするものが Data SRAM (static random access memory) である．このように命令用とデータ用の通信経路と記憶装置を物理的に分割し管理する構成法を「ハーバードアーキテクチャ」とよぶ．この構成によってメモリアクセスのボトルネックが解消され、高速化できる。

ALU では主として算術演算，論理演算，ビット操作の 3 種の計算が行われるが，基本的にそれらは汎用レジスタ（General Purpose Registers）を介して行われる．汎用レジスタは 32 個ありそれぞれ `r0`〜`r31` という名が付けられている．汎用レジスタは 8 ビットのデータが 32 個あると考えればよい．データバスが周辺回路で共有しているのに対し，汎用レジスタは ALU 部に直結して高速に動作することが特徴である．周辺回路を制御・監視するためのメモリは I/O レジスタとよばれる．Data SRAM と汎用レジスタは物理的に分断されているが，メモリ空間自体はリニアに結合されていることに注意する。

プログラムはプログラムカウンタが示すプログラムメモリのアドレスから命令を呼び出し，処理を実行する．CPU がプログラムメモリから命令を呼び出し，計算を実行するたびにプログラムカウンタはインクリメントされる．関数の呼び出しなどが起きるとプログラムカウンタは関数が記録された番地にジャンプし，関数が終了後関数を呼び出した時点でのプログラムカウンタに再設定される．

Status and Control とあるのは，割り込みの設定や関数の呼び出し元の情報や割り込みが起こった際，元の実行状態に復元するために保持しておくものである．代表的なレジスタはステータスレジスタ（SREG）である．

■メモリ構成 Atmel AVR ATmega328P にはプログラムの途中の計算結果や状態を保持する記憶空間とプログラム用の記憶空間があることを述べた．それぞれデータメモリとプログラムメモリとよぶ．前者は SRAM で実現されており，後者はプログラム可能なフラッシュメモリであり不揮発である．電源を落としてもプログラムが消えてしまうことはない.

データメモリは図 2 のような構成をとっている．まず，CPU コアで述べた汎用レジスタは 0x00〜0x1F の 32 個分の番地を占有し，続いて代表的な周辺回路のアドレスとして 64 個のアドレスを，さらに拡張機能として 160 個のアドレスを用意している．0x は 16 進数を意味し，0x1F は 10 進数では 31 に相当する．拡張 I/O レジスタに続き 2048 個× 8 ビットの Internal SRAM（Data SRAM に相当）が用意される．C 言語プログラミングで変数を宣言すると，基本的にこの領域にデータが保存される．32 個の汎用レジスタ，64 個の I/O レジスタ，160 個の拡張 I/O レジスタ，Internal SRAM は物理的には分離したものであるが，論理的には連続的に配置されていることである．この方式のおかげで回路構成を意識せずに C 言語プログラムを記述することができるようになる．

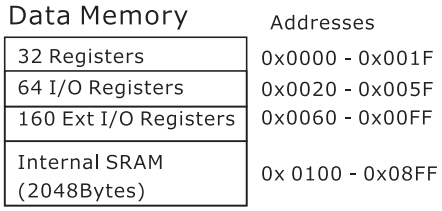


図 2 ATmega328P のデータメモリ

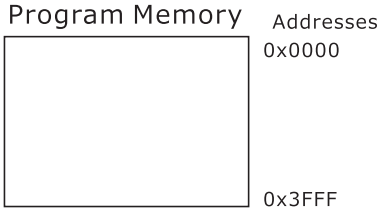


図 3 ATmega328P のプログラムメモリ

プログラムメモリは図 3 のような構成をとっている．ATmega328P の場合 0 番地から 0x3FFF = 16383 番地までメモリがある．データメモリは 1 番地あたり 8 ビットであったのに対し，こちらは 1 番地あたり 16 ビットである．つまり，プログラムメモリのサイズは 32768 バイトになる．16 ビットに演算子，計算対象の

メモリの番地，加算すべき値といったものが格納されている．AVR には 120 種程度の命令が存在し，大半は 1 命令あたり 16 ビットのデータで記述できる．

プログラムメモリとデータメモリほど重要ではないが, AVR のメモリの 3 つ目の構成要素として EEPROM がある．これは Electrically Erasable and ProgrammableRead Only Memory の略であり，電気的に書き換えることができる ROM の意味である．SRAM は電源を一旦落とすとデータが「揮発」するのに対し，電源を落としてもデータを保持することができる．ATmega328P の場合 EEPROM のサイズは 1024 バイトである．番地を 0x00 から 0x3FF = 1023 に変更しアクセスする．以上から，AVR ATmega328P のメモリのサイズは

- Flash：32K （32×2^{10}）バイト
- Internal SRAM：2K（2×2^{10}）バイト
- EEPROM：1K（1×2^{10}）バイト

とまとめられる．

2.1.2　AVR CPU コアを取り囲む周辺機能

CPU コアの図でも出てきているが，AVR は全体として図 4 のような構成になっている．

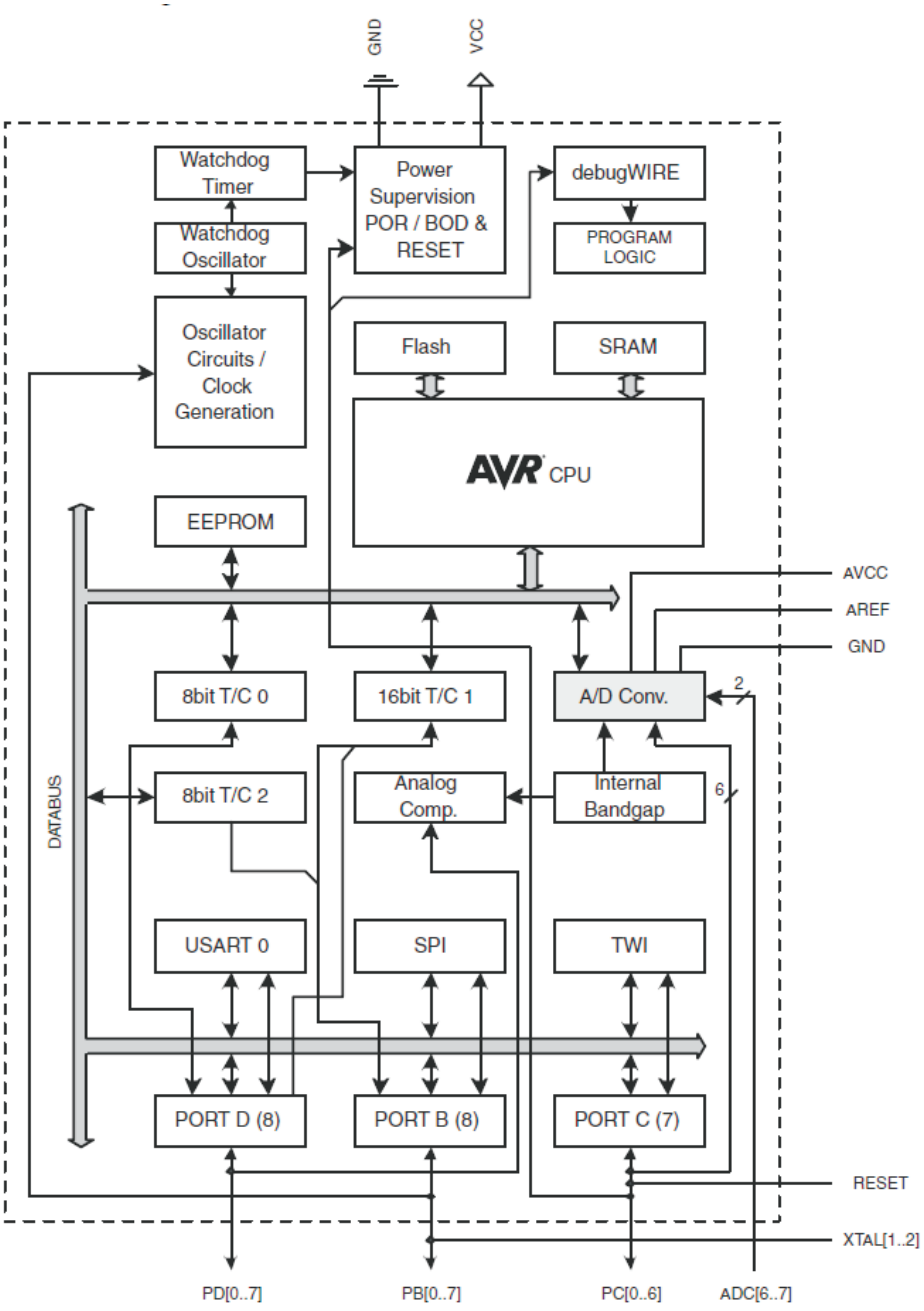


図 4　AVR ATmega328P のブロック図

様々な機能が 1 つのチップに収められているが，CPU コアの周りにそれぞれの機能を実現する回路がついていると考えればよい．AVR ATmega328P では入出力ポート，タイマ，A/D 変換，アナログコンパレータ，

SPI，TWI，USART，EEPROM，デバッグワイヤがついている．プログラムメモリのアドレスを指定し，データバスを経由し周辺回路を制御することになる．大まかに

- SPI, TWI, USART：通信機能に利用
- タイマ：一定周期の処理や時刻の計測に利用
- A/D 変換：（センサ等から出力される）アナログ信号をデジタル化
- アナログコンパレータ：A/D 変換と異なり基準電圧より大きい（小さい）かを判断
- EEPROM：不揮発性の（読み書き可能な）メモリ．データの記録に利用する．

演習で取り扱わない項目としてデバッグワイヤについて簡単に触れておく．これは文字通りデバッグの際に用いる機能である．一般的な PC プログラミングでは printf 文などが利用できるため，計算の途中結果や変数の値を確認することが容易であるし，専用のデバッグソフトウェアを利用することもできる．マイコンのように入出力に限りがある状況では，状態を把握するためのある種のハードウェア機構が不可欠である．CPU の内部状態にアクセスするための回路がデバッグワイヤである．

2.2　ATmega328P の DIP IC

ATmega328P はいくつもの IC 形状があるが，伝統的な DIP （Dual Inline Package）型のピン配置を紹介する．DIP 型とは長方形の形をした IC の両側面からピンが飛び出しているものを指す。ATmega328P のピン構成は Atmel 社のデータシートの 2 ページ目に記述されている．ここで記号の上にバーが表示されているものは負論理を示している．例えば $\overline{\text{RESET}}$ は LOW になった際に Active になる．

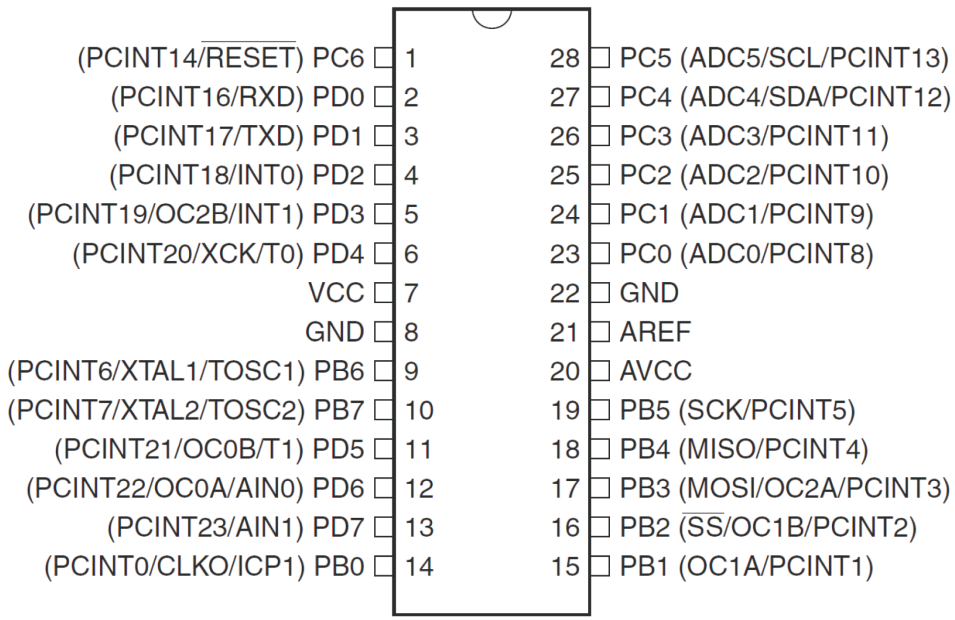


図 5　ATmega328P のピン構成（Atmel 社のデータシートより引用）

電源関連が 5，入出力用の多機能ピンが 23 という構成である．具体的には

- VCC：電源供給用
- GND：接地用（2 個ある）
- AREF：A/D 変換用の基準電圧
- AVCC：A/D 変換の供給電源
- PB0〜PB7，PC0〜PC6，PD0〜PD7：入出力多機能ピン

ここで多機能というのは，プログラムによりピンの役割を変えられるものをさす．例えば 25 番目のピンに相当する PC2 を A/D 変換用のピン ADC2 として利用することができる．上記の図に表れる言葉の意味を示す．

- PB0〜PB7，PC0〜PC6，PD0〜PD7：デジタル入出力として利用可能という意味である。それぞれのまとまりを、この演習ではポート B，ポート C，ポート D と呼ぶ。
- ADC0〜ADC5 の 6 ピンは A/D 変換に利用される．
- OC0A, OC0B, OC1A, OC1B, OC2A, OC2B は PWM 動作時に利用される出力ピンである．
- XTAL1，XTAL2 というのは外部の水晶発振器を接続するピンである．
- SCK，MISO，MOSI，SS，RXD，TXD，SCL，SDA は通信用である．
- PCINT0〜PCINT23 はピン変化割り込み用であり多数の入力を扱う場合便利である．
- INT0，INT1 は外部割込み用である．
- ICP1 はインプットキャプチャに用いられる．
- RESET はこれに最小パルス幅以上 L 入力を与えるとリセットがかかる．
- AIN0，AIN1 はアナログコンパレータ用のピンである．
- T0，T1 は外部クロックを活用したタイマを利用する際の入力である．
- XCK は USART を外部クロックで駆動する場合に用いるピンである．

ピンの数は限られており、デジタル出力に使っているピンを同時に A/D 変換に使うことはできない。ピンへの機能割り当てはシステム設計時に慎重に検討すべきである．

なお，このチップは 1.8V〜5.5V の電圧で駆動する．また，チップから供給できる電流の最大量は 200mA である．I/O ピン 1 つあたりの最大供給電流は 40mA である。CPU は最大 20MHz で駆動する．なお，演習では ATmega328P を 16MHz で駆動させる．電圧 5V をマイコンに印加する場合，5V（あたりの）電圧信号を H レベル，0V（あたりの）電圧を L レベルとよぶ．その他，マイコンの電気的特性に関しては Atmel 社データシートの 28 章 (Electrical Characteristics) を参照すること．

3 Arduino とは

ワンチップマイコンには様々な機能が内蔵されているとはいえ IC 部品であるため、利用には少なくとも基板上に実装し、プログラミングや入出力のためのコネクタやターミナルを足す必要がある。マイコンボードは、自分で IC を買ってきてはんだ付けしなくともすぐにマイコンが利用できる形式である。Arduino は非常にポピュラーなマイコンボードで、2005 年にイタリアで開発が始まった。PC と USB で接続してプログラミングができ、使いやすい開発環境が提供されている。さまざまな大きさの Arduino 互換マイコンボードが開発されている。Arduino は下記の特長がある。

- 初心者でも容易に開発ができるよう Arduino IDE というフリーのソフトウェア開発環境が用意されて

いる．これにより AVR 特有のアーキテクチャを意識せずにプログラミングが可能.

- ボードの回路図は一般に公開され，さらにその回路図の電子回路 CAD データも公開されているオープンソースハードウェアである。

すぐ使える、簡単に使えるという特長から、組み込み開発に不慣れなユーザーに人気を博し，その結果，ハードウェアのオープン性もあいまって、ニーズに応じて Arduino クローンと呼ばれる互換ボードが多数誕生することとなった．最も標準的なボードは Arduino UNO である．UNO の前身は Arduino Duemilanove である.

簡単なプログラミング環境があるならその使い方だけ教えてくれればいいよ，と思うかもしれないが，機械情報工学科の学生にはこのようなお膳立てされた環境を提供する「作り手」になれる本質的な知識と技術力を身につけて欲しい．そのためにもレジスタ操作にまで踏み込んだマイコンのアーキテクチャの理解と、プログラミング方法を、演習で習熟してもらいたい.

4　組み込みソフトウェア開発の基本的な流れ

4.1　クロス開発とは

これまで計算機プログラミングといえば，Windows や MacOS，Linux 等が動く汎用計算機上でのプログラム作成・コンパイル・リンク，生成されたバイナリプログラムの実行であった．組み込みシステムは計算資源に限りある環境でプログラムを動作させる必要があるため，それ自体ではコンパイラやリンカなどを含む開発環境を持たないことが多い。したがって組み込みシステム開発では「クロス開発」という方法が使われる。

クロス開発を従来学んできたソフトウェア開発と対比して説明する．汎用計算機上のプログラミングは下記のような手順で行ってきた。

1. 処理内容を記述する C 言語ソースファイルを PC 上で作成
2. ソースファイルをオブジェクトファイルにコンパイル
3. 複数のオブジェクトファイルをリンクし，PC で動作する実行ファイル（バイナリファイル）を生成

一方，クロス開発とは，あるソフトウェアの開発をそのソフトウェアが動作するシステムとは異なるシステム上で開発することを指す．その際，プログラマが記述したソースコードは，ターゲットとなるシステム（本演習では AVR に該当）で動作するオブジェクトコードに変換する必要がある．こうした変換機能を持った特殊なコンパイラを，クロスコンパイラとよぶ．一般にクロス開発では下記のような流れを取る。

1. 処理内容を記述する C 言語ソースファイルを作成.
2. **クロスコンパイルにより**ソースを**ターゲット用**オブジェクトファイルにコンパイル
3. **ターゲット用**オブジェクトファイルをリンクし**ターゲット用**実行ファイルを生成
4. ターゲット用実行ファイルをプログラムメモリに書き込むためのデータ形式に変換
5. プログラムライタ（書き込み器）によりデータをメモリに書き込む.

ソースから変換される実行ファイルはターゲット用（AVR 用）であり、開発環境では動作しないこと、実行ファイルをさらに変換すること、変換したデータを組み込みシステム上のプログラムメモリに書き込む点が異なる。

4.2　AVR GCC でのクロス開発

標準的な C コンパイラとして GCC（GNU Compiler Collection）があり、フリーで利用できる。GCC はさまざまな CPU アーキテクチャをサポートするが、AVR マイコンもサポートされている。ここでは、クロス開発の具体的な流れを AVR GCC を使った例で説明する。Atmel 社純正の開発環境としては Windows で動作する Atmel Studio がある。

4.2.1　コマンドラインインタフェースからの AVR GCC の利用

ここでは，上述のクロス開発の手順のうち，手順 2 から 4 までの手順について説明する．

■クロス開発の手順 (2)，(3)．コンパイルとリンク　適当なプログラムを作成し，test.c として保存したとする．オブジェクトファイルの生成と標準関数を事前にコンパイルしオブジェクトファイル化した avr-libc と

リンクを行い，ターゲット用の実行ファイルを作成する．

```
$ avr-gcc test.c -mmcu=atmega328p  -o test.elf
```

実行ファイルが test.elf である．ELF というのは，Linux でよく使われる Executable and Linking Format 形式のファイルを意味する．`--mmcu` というのは数ある AVR の中から ATmega328P をターゲットとして利用しますよ，という注意書きのようなものである．

■クロス開発の手順 (4)．ファイル形式の変換　Linux における実行ファイル形式 ELF はバイナリファイルであるが，AVR のプログラムメモリのようなメモリへデータを書き込む際，直接バイナリデータを書き込まず，それに相当するアスキーデータを一旦用意してからフラッシュメモリへの書き込みを行うことが多い．avr-gcc では ELF 形式のファイルを Intel HEX 形式[*1]というアスキーフォーマットのファイルに変換を行う．コマンドラインでは，生成した test.elf を test.hex に変換する．

例えば test.c を

```
#include <avr/io.h>
int main(int argc, char* argv[]){
  int sum = 0;
  int i;
  for(i = 0; i < 13; i++){
    sum += i;
  }
  return 0;
}
```

とした場合，

```
$ avr-objcopy -O ihex test.elf test.hex
```

により

```
:100000000C9434000C9451000C9451000C94510049
:100010000C9451000C9451000C9451000C9451001C
:100020000C9451000C9451000C9451000C9451000C
:100030000C9451000C9451000C9451000C945100FC
:100040000C9451000C9451000C9451000C945100EC
:100050000C9451000C9451000C9451000C945100DC
:100060000C9451000C945100011241FBECFEFD8E026
:10007000DEBFCDBF11E0A0E0B1E0EAE0F1E002C0F8
:1000800005900D92A030B107D9F711E0A0E0B1E0E2
:1000900001C01D92A030B107E1F70E9453000C94FB
:1000A00083000C940000DF93CF93CDB7DEB7289781
:1000B0000FB6F894DEBF0FBECDBF9E838D837887C9
:1000C0006F831C821B821A8219820DC02B813C8196
:1000D00089819A81820F931F9C838B8389819A8166
:1000E00001869A83898389819A818D30910574F371
:1000F00080E090E028960FB6F894DEBF0FBECDBF2B
:0A010000CF91DF910895F894FFCF2E
:00000001FF
```

なる test.hex が生成される．この HEX 形式のファイルを AVR のフラッシュメモリに書き込む．なお，

```
$ avr-gcc test.c -mmcu=atmega328p -O0 -S -Wall
```

とするとアセンブラ形式のファイルが生成される[*2]．上記の例では

[*1] このフォーマットは AVR 特有のものではない．Intel HEX 形式と同様の目的の Motorola SREC 形式がある．

[*2] `-Wall` は警告を出力，`-O0` は最適化を行わない，の意味である．

```
.file "test.c"
__SREG__ = 0x3f
__SP_H__ = 0x3e
__SP_L__ = 0x3d
__CCP__  = 0x34
__tmp_reg__ = 0
__zero_reg__ = 1
.global __do_copy_data
.global __do_clear_bss
.text
.global main
.type main, @function
main:
push r29
push r28
in r28,__SP_L__
in r29,__SP_H__
sbiw r28,8
in __tmp_reg__,__SREG__
cli
out __SP_H__,r29
out __SREG__,__tmp_reg__
out __SP_L__,r28
/* prologue: function */
/* frame size = 8 */
std Y+6,r25
std Y+5,r24
std Y+8,r23
std Y+7,r22
        以下略...
```

なるコードが生成される.

5 組み込みソフトウェア開発 1

5.1 Hello World AVR!

5.1.1 Arduino IDE のインストールとサンプルプログラムの利用

各自の環境で Arduino IDE がインストールされているか確認する。

もし Arduino IDE がインストールされていない場合は、Arduino のウェブサイト `https://www.arduino.cc` のダウンロードページから各自の開発環境の OS に合わせた Arduino IDE のインストーラをダウンロードする。ダウンロードする際に寄付を促されるが、”Just Download”をクリックすれば無料である。

Arduino IDE をインストールしたら、起動する。下記の手順で Arduino にサンプルプログラムを書き込む。

機械工学総合演習第二「電子回路」のテキストも参照のこと。

- Arduino Nano と PC を USB ケーブルで接続する。
- Arduino IDE のメニュー「ツール」から接続したボードの設定
  - –「ボード」は「Arduino Nano」を選択
  - –「プロセッサ」は「ATmega328P(Old Bootloader)」を選択
  - –「シリアルポート」に適切なデバイスを指定する。Arduino を接続したときに新しく現れるポートである。MacOS なら「/dev/cu.usbserial-*nnnn*」、Windows の場合は COM*n* など。
- メニューの「ファイル Files」「スケッチ例 Examples] の中にサンプルプログラムがたくさん用意されている。
- ソースコードを開き、ウィンドウ上部の upload（マイコンボードに書き込む）ボタン（右矢印アイコン）を押す。コンパイルの後、書き込みが成功すると、「ボードへの書き込みが完了しました。」と表示される。

Windows 版の Arduino IDE で下記のようなエラーコードが出て書き込みに失敗する場合、USB で接続した Arduino がうまく認識されていない可能性がある。

```
stk500_recv(): programmer is not responding
```

スタートメニューを右クリックしたメニューの中のデバイスマネージャーを起動すると、認識されていないデバイスが黄色い「！」がついた「USB Serial」などの名前で確認できる。この時は、ドライバを追加でインストールする。ドライバを `http://www.wch.cn/download/ch341ser_exe.html` からダウンロードしてインストールする（Arduino Nano を PC に USB で接続した状態でドライバをインストールすること）。必要に応じて PC を再起動し、デバイスマネージャで Arduino Nano が COM*n* として認識されていることを確認する。

手動でドライバをインストールする方法もある:`https://www.arduino.cc/en/Guide/DriverInstallation`

課題 1

Arduino IDE に含まれるサンプルプログラム、「01.Basics」の中の「Blink」を Arduino Nano に書き込み、動作を確認せよ。

Arduino IDE では、ソースコードのことをスケッチと呼ぶ。拡張子は.ino である。

5.1.2　ISP

多くの Arduino シリーズは USB インタフェースを備えており、Arduino IDE からの書き込みに必要なのは USB ケーブルのみである。しかし、一般的にはマイコンボード上のマイコンチップへのプログラムの書き込みには、専用の書き込み器が必要になることが多い。通常 ISP という仕組みを利用する。ISP とは In-System Programming の略である．これは，マイクロコントローラにおいて，事前にプログラムを書き込んでからシステムに組み込むのではなく，組み込み済みの状態でプログラムを書き込むことをさす．この機能の利点は，システムの組み立て前に書き込みを完了する必要が無く、また書き込みのたびにマイコンチップを取り外す必要がないことである．一般的に ISP をサポートしたチップは，書き込みに必要なすべての電圧をシステムの通常の供給電圧から作り出す回路を内部に持っており，書き込み器（ライター）とはシリアル通信を行う。

なお，AVRISP では図 6 のような 6 ピンコネクタを採用しており，Arduino 基板から飛び出している ICSP などと書かれている 6 ピン（オス）は図 6 に示したピン名称に該当するピンに結線されている．

図 6　ISP 6 ピンコネクタ（穴側から見た場合）.

5.2　デジタル出力

プログラムを書きながら AVR の諸機能を学んでいこう。まずはデジタル出力を取り上げる。AVR のピンの状態をプログラムにより論理値 H，論理値 L に設定するものである。

デジタル出力を行うことで Arduino 上の LED を点灯させてみよう。Arduino Nano の基板上には 4 つの LED が載っている。そのうち、TX と RX は通信状況を表し、PWR は電源 ON を表す。残りの 1 つがプログラム可能な LED である。なお、LED の色は互換機では変更されている場合がある。

課題 2

(1) Arduino Nano の回路図を開け。
(2) デジタル出力によってプログラム可能な LED を回路図から見つけ出し、ATmega328p のどのポートに接続されているか把握せよ。
(3) 回路図での接続状況から、LED を点灯・消灯させるためのポートの状態（high/low）を考察せよ。

マイコンのデジタル I/O ポートの状態を制御するためには、専用のレジスタの値を操作する必要がある。ポート B（PB0〜PB7）のそれぞれのピンの信号方向（direction）、つまり入力か出力かを決めるレジスタが DDRB、ピンの状態（電圧）に対応したレジスタが PORTB である。同様にポート C（PC0〜PC6）には DDRC，PORTC，ポート D（PD0〜PD7）には DDRD，PORTD が割り当てられている。Atmel 社のデータシート（`doc8271.pdf` 度々利用するので開いておくこと）では総称して PORTx，DDRx と書かれる（例えば 93 ページ目を参照）．x には B，C，D が入る．DDRx は Data Direction Register を意味する．ポートが出力に設定された場合，PORTB の値を 1 に設定するとピンは H を出力，0 に設定すると L を出力す

る．PORTx，DDRx はそれぞれ 1 バイトのレジスタである．それぞれのポート Pxn の設定・制御・状態は DDRx の n ビット目と PORTx の n ビット目に対応している．

例えば，下記の設定は PB7，PB6，PB5，PB4 を出力に設定し PB3，PB2，PB1，PB0 を入力として用い、PB7 と PB6 を H 出力に設定し PB5 と PB4 を L 出力に設定することを意味する．

```
  DDRB = B11110000;
  PORTB = B11001111;
```

Arduino のスケッチ（拡張子.ino）では次のような記述になる。B は 2 進数を表記するための接頭語である。

```
void setup() {
  DDRB = B........;
}

void loop() {
  PORTB = B........;
}
```

ちなみに、これを avr-gcc のための C 言語で書き直すと次のようになるだろう。

```
#include <avr/io.h>

int main(int argc, char* argv[])
{
  DDRB = 0b........;

  while(1){
    PORTB = 0b.......;
  }
  return 0;
}
```

Arduino IDE での標準的な実装は下記のようになる。

```
void setup() {
  pinMode(..., OUTPUT);
}

void loop() {
  digitalWrite(..., HIGH);
}
```

これは、Arduino ライブラリが AVR-GCC の記述や関数を使いやすくしたラッパーにすぎないことを示している。Arduino ライブラリを使った実装は簡便だが、オーバーヘッドが大きく、実行時間は余計にかかる。この演習では、極力 AVR-GCC のサポートするネイティブの実装方法を用い、Arduino ライブラリの使用を原則禁止とする。

レジスタ操作によって LED を点灯・消灯させるプログラムを書いてみよう。

課題 3

(1) Atmel 社の 8 ビットマイコンのデータシートを開け。ファイル名は doc8271.pdf である。
(2) データシートの 93 ページを開き、I/O ポート設定用のレジスタ群とその名前を確認せよ.
(3) レジスタ DDRB と PORTB の設定によって Arduino 上の LED を点灯するスケッチ blink1.ino を作成し、Arduino に書き込んで動作を確認せよ。なお、使わないポートの DDRB の設定は任意で良い。
(4) LED を消灯するように blink1.ino を書き換えて動作を確認せよ。

5.2.1　遅延

タイミングを制御する目的で正確な遅延を作り出したい場合がある。for 文を無駄に回すことなどで遅延を作ることもできるが、コンパイルの最適化によって挙動が変わるなど，時間が一定に保たれる保証はない．また、計算資源の無駄である。

この問題に対しユーティリティ関数として遅延処理関数 delay(ms) が用意されている。チップの種類や動作周波数が適切に設定されていれば、一般的な記述で正確な遅延を作り出せる。なお、さらに短い遅延のためには delayMicroseconds(us) がある。

課題 4

(1) delay() 関数を用いて LED が 2 秒間 ON、1 秒間 OFF を繰り返すようなスケッチ blink4.ino を作成せよ

5.3　レジスタ操作の実装

レジスタに値を設定するとはどういうことなのか，レジスタ操作がライブラリによって隠蔽されている Arduino IDE ではなく、AVR GCC による C 言語プログラミングでの PORTB や DDRB の操作を例に理解しておこう．以下のようなレジスタへの値に設定した時、実際には何が起こっているだろうか。

```
PORTB = 0b11001111;
```

デジタル入出力の設定には io.h を読み込む必要がある。この io.h の 221 行目に下記の記述がある。

```
#elif defined (__AVR_ATmega328P__) || defined (__AVR_ATmega328__)
#  include <avr/iom328p.h>
```

Atmel Studio のプロジェクト設定で 328p という設定を行うと，avr/iom328p.h がインクルードされるということになる．この iom328p.h を開いてみると，74 行目に下記の記述がある。

```
#define PORTB _SFR_IO8(0x05)
```

これは, 0x05 番地目を意味するものと考えられる．さて, _SFR_IO8 とは何かを探ってみよう．ATmega328p の場合，_SFR_IO8 とは sfr_defs.h の 179 行目

```
#define _SFR_IO8(io_addr) _MMIO_BYTE((io_addr) + __SFR_OFFSET)
```

に該当し_MMIO_BYTE とは

```
#define _MMIO_BYTE(mem_addr) (*(volatile uint8_t *)(mem_addr))
```

となる．つまりは，I/O レジスタの 0x05 番地（汎用レジスタ 32 個分：__SFR_OFFSET ずれた番地）のポインタの中身を意味する．実際のところ

```
PORTB = 0b11001111;
```

というのは

```
volatile uint8_t* portb = (volatile uint8_t*) (0x05 + 0x20);
*portb = 0b11001111;
```

と同じである．つまり 0x25 番地のメモリの中身を直接操作しているということになる．Atmel 社のデータシートの 93 ページ，13.4.2 を開いてみよう．PORTB は I/O レジスタの開始番地から相対 0x05 番地であることがわかる．なお，一般的な PC でこのようなメモリの直接操作を行った場合，セグメンテーションフォルトが発生する。

オプション課題 1

(1) インターネットで英単語 volatile の意味と C 言語の予約語である volatile の意味を調べよ．

5.4　ビット操作

マイコンプログラミングの要はレジスタのビット操作である。

5.4.1　シフト演算マクロ

ところで、デジタル入出力の操作で使った下記の表現はいささか読みづらい．

```
PORTB = B11001111;
```

そもそも 1 と 0 を 8 個も入力するのは間違いが生じやすく、見分けにくい。そこで次のような書き方がよく用いられる。

```
PORTB = 0xCF;
```

これは 16 進数表記で 8 ビット分を 2 桁の 16 進数で表現したものであり，この 2 つの命令は全く同じ意味である．効率性を保ったまま可読性が上昇したが，まだわかりやすいとは言えない。コメントに処理の内容を書くというのも一案であるが，コードの書き換えと同期したコメントの修正が求められ保守性が低くなる．そこで次のようにシフト演算子を用いた方法もよく行われる．

```
PORTB = (1 << 7) | (1 << 6) | (1 << 3) | (1 << 2) | (1 << 1) | (1 << 0);
```

1 << 7 は 8 ビットのうち 0 ビット目を 1 に設定した状態で 7 ビット分左にずらす，即ち PORTB レジスタの 7 ビット目を 1 に設定することを意味する．設定したいビットが複数ある場合，上記のように論理和を用いる．これにより 1 を設定するビット番号が直接現れることになる．このように指定したビットに 1 を設定することをセット（set），0 に設定することをクリア（clear）とよぶ．ただ，1 <<という表現がまどろっこしいということで AVR GCC では`#define _BV(bit) ( 1<<(bit) )`のように定義されるマクロ関数が用意さ

れており、次のように利用できる。

```
PORTB = _BV(7) | _BV(6) |  _BV(3) | _BV(2) | _BV(1) | _BV(0);
```

このコードは PORTB = 0xCF; と全く等価である．さらにわかりやすくするために下記のように書くこともできる。

```
PORTB = _BV(PORTB7)|_BV(PORTB6)| _BV(PORTB3)|_BV(PORTB2)|_BV(PORTB1)|_BV(PORTB0);
```

PORTB7 のように PORTB の 7 ビット目をあらわすマクロは iom328p.h で定義されている．これにより PORTB のなかの特定のビット操作の把握が容易になる．元々 PORTx はビットとピンとの対応が直接的であるため無意味にみえるかもしれない．このような書き方は、ある機能が何番目のビットに割り振られているかわかりにくい時に役立つ。

利点を確認できる例をみてみよう．ATmega328P では，A/D 変換を実行する際，その開始時に ADCSRA というレジスタ（マニュアル 264 ページ目参照）の 6 ビット目にある ADSC をセットする．セット直後に A/D 変換が開始する．変換中はセットされたままであり，A/D 変換が終了するとこの 6 ビット目がクリアされる．6 ビット目がクリアされるまで待機し，その後 A/D 変換した値を読み取れば AVR に接続されたアナログ信号を読み取ることができる．この手順を記したものが

```
ADCSRA |= _BV(ADSC);
while(ADCSRA & _BV(ADSC)){
//do nothing
}
sensor_value = ADC;
```

```
ADCSRA |= 0b01000000;
while(ADCSRA & 0b01000000){
//do nothing
}
sensor_value = ADC;
```

である．2 進数で表記した場合と比較してコードの可読性と保守性が大幅に上がることがわかる．なお，ADCSRA |=_BV(ADSC) というのは，ADSC 以外の情報はそのままに，ADSC をセットすることを意味する．

5.4.2　ビット監視

前節での例に出てくる条件分岐は，ADCSRA レジスタの ADSC ビットが 1 である限り待機するという意味である．このとき AVR GCC では bit_is_set という便利なマクロが用意されており

```
while(bit_is_set(ADCSRA, ADSC)){
}
```

のように簡略化できる． bit_is_set は対象とするビットがセットされているとき非零を返し，クリアされているときに 0 を返すものである．一方ビットがクリアされているときに非零を返し，セットされているときに 0 を返す関数は bit_is_clear である．ビットの状態が変化するまで待機する，さらに簡単なマクロとして

```
loop_until_bit_is_set(ADCSRA, ADSC);
```

や loop_until_bit_is_clear というコードも可能である．こちらは do-while 文を使っているが本質的に同一の処理を行う．

5.4.3　応用

以後、積極的にマクロ定義を利用し、2 進数表記を用いることや、ピン番号を整数で直接指定することは避けること。

課題 5

先の delay() 関数を用いた LED の点滅のプログラムを_BV および PORTxn の表現を用いて書き直したスケッチ blink4macro.ino を作成せよ。

5.5　デジタル入力

デジタル出力に引き続きデジタル入力についても学んでいこう．

5.5.1　周辺回路の準備

Arduino にはスイッチが一つあるがリセット用のものであるため，ジャンパワイヤとブレッドボードを使ってピンに H/L を入力してみよう。

ブレッドボードとは電子回路の実験を行う際，半田付けすることなく部品を抜き差しして回路を組むことができるものである．ブレッドボードは半田付けが不要なため，部品を再利用でき，回路のデバッキングができる，等のメリットがある．しかし部品が抜ける、接触が不安定になるといったデメリットもある．ブレッドボードの構造は図 7 のようになっている．縦方向の穴は内部で接続されている．つまり同じブロックにある縦方向の 5 個の穴のどれかに部品の足やワイヤをさすと同じブロックのほかの穴と自動的に接続される．これに対し，横方向は接続していないので，自動的には接続されない．ブレッドボードに沿った赤黒の線は電源を接続するもので，これらは横方向につながっている．通常赤い線はプラス電圧，黒い線は GND に設定する．

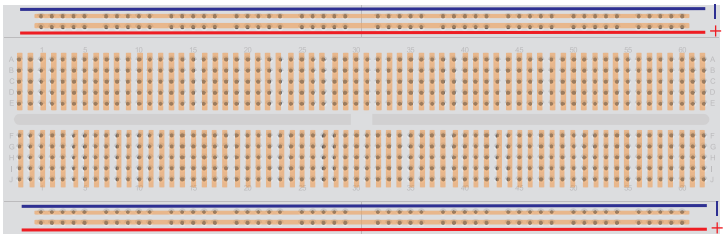


図 7　ブレッドボードの構造

短絡を防ぐために、**Arduino に USB ケーブルをさしたまま配線作業しないこと．また、5V ラインが GND に直接つながっていないかよく確認すること。**USB ケーブルを挿した時に LED がつかないなど異常がある場合はすぐに USB ケーブルを抜くこと。

5.5.2　プルダウン・プルアップ

回路を組み立てる前にデジタル回路に関する重要な基礎知識として，プルアップ抵抗・プルダウン抵抗というものについて説明をしておこう．これらは入出力ピンが「オープン」となって状態が不定とならないために用いられる．

AVR などの CPU 等が回路からの入力を判断する基準は電圧である．しかしながら CPU に入力される回路にオープンな状態が存在する場合（グラウンドもしくは Vcc などの基準電圧との接続が存在しない場合），その回路の電圧レベルは H（5V 周辺の電圧）か L（0V 近辺の電圧）か安定しない．例えば図 8 を見て欲しい．スイッチが OFF の場合，CPU への入力は安定せずそれゆえにスイッチの状態の判別が困難になる．

入力ポートに抵抗をつけて VCC に接続し，OFF の時は VCC に，ON のときは GND となるようにした回路の構成をとることをプルアップするという．プルアップのための抵抗をプルアップ抵抗と呼ぶ（図 9）．AVR などでは入力ポートに流れ込む電流の大きさを考え，比較的大きな抵抗を用いるのが一般的である．

プルダウンとは，プルアップと逆の役割で OFF のとき 0V，ON のとき 5V となるものである（図 10）．本演習で組んだテスト回路ではプログラミングのわかりやすさからプルダウン構成となっているが，マイコン自体の消費電力を抑えられるという理由でプルアップ抵抗が歴史的に広く用いられている．

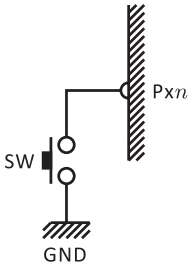


図 8　オープンな回路

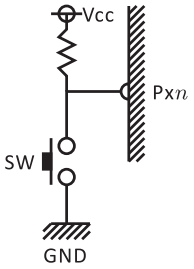


図 9　プルアップ

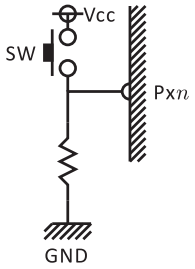


図 10　プルダウン

なお，消費電力の観点[*3]で LED の接続も図 11 よりも図 12 が一般に用いられるが，プログラム時のわかりやすさから図 11 のような構成を取っている．

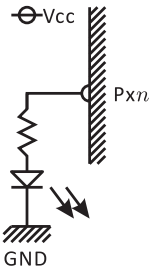


図 11　LED の接続方法 1

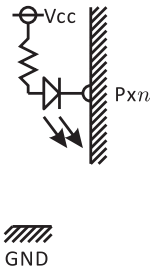
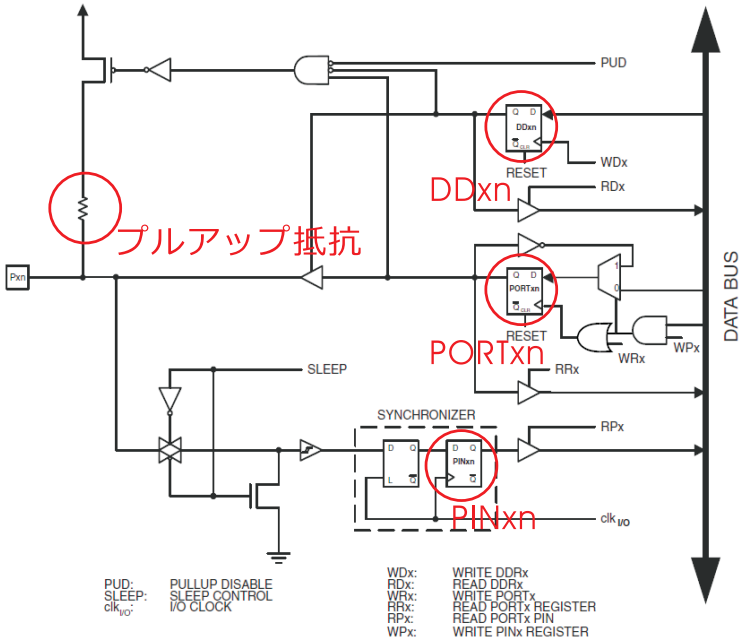


図 12　LED の接続方法 2

5.5.3　基本的なデジタル入力のプログラミング

デジタル入力のプログラミングでは基本的に DDRx により入力か出力を決定し，PINxn で電圧状態を読み取ればよい．デジタル出力ではピンの状態を設定するために使われたレジスタ PORTxn は、デジタル入力では内部プルアップ抵抗の ON/OFF に使われる。あるピンに対応する PORTxn のビットをセットした場合，入力は 20〜100KΩ でプルアップされる．具体的には図 13 のような回路構成と等価となる．セットするかクリアするかは外部に接続する回路に応じて設定する．

[*3] 一般的なマイコンでは吐き出す電流よりも，吸い込む電流のほうが大きいのが通常であるが，AVR はその限りではない．



Note: 1. WRx, WPx, WDx, RRx, RPx, and RDx are common to all pins within the same port. $clk_{I/O}$, SLEEP, and PUD are common to all ports.

図 13　Pxn の等価回路．Atmel 社のデータシート 77 ページ参照

具体的な Arduino スケッチの例を示す。プログラム中の「...」は目的に応じて補完・変更する部分である。

```
void setup() {
  DDRB = ........;
  DDRD = ........;
  PORTD = ........;
}

void loop() {
  if (bit_is_set(..., ...)) {
    PORTB |= ...;
  }
  else {
    PORTB &= ...;
  }
}
```

なお、C 言語では以下のようになるだろう。

```
#include <avr/io.h>

int main(int argc, char* argv[])
{

  DDRB = ........;
  DDRD = ........;
  PORTD = ........;

  while(1){
    if(bit_is_set(PIND, ...)){
      PORTB |= ...;
    } else{
      PORTB &= ...;
    }
    ....
  }
  return 0;
}
```

上記のコードを参考に、下記の課題に取り組め。Arduino に USB ケーブルを接続し直した時に全ての LED がつかない、あるいは PC 側で USB ポートの過電流が検出されるなどしたら直ちに USB ケーブルを抜いて回路を修正すること。

課題 6

(1) ピン D2 (PD2) の状態が low になる（GND に接続される）と Arduino 基板上の LED が消灯するスケッチ input1.ino を実装せよ。入力ピンに対応する `PORTxn` をセットすることでマイコン内部のプルアップ抵抗を利用すること。

(2) USB ケーブルを Arduino から抜き、ブレッドボードに Arduino Nano を差し込んで回路を組もう。ショート防止のため、ON 状態で回路を組み換えないこと。

(3) タクトスイッチ等を使って回路図を参考にプルアップ型のスイッチ回路を作成して、D2 ピンに H/L を入力できるようにしよう。内部プルアップ抵抗をセットすることで、外部のプルアップ抵抗を省くことができる。もしタクトスイッチなどがない場合は、簡易的にジャンパワイヤを用いることができる。入力ピンにジャンパワイヤの片側を接続した状態で、同じジャンパワイヤのもう片側をどこにも差し込んでいないときは内部のプルアップ抵抗によって H 入力、GND に差し込めば L 入力となる。抵抗を挟まずに +5V につなぐことは避ける。

5.5.4　状態遷移とチャタリング対策

電灯のような対象を制御する場合、ボタンスイッチを押下している間だけでなく、ボタンスイッチを押すごとに点灯・消灯を切り替えるような動作がさせたくなる。これには状態の記憶が必要である。

スイッチが押された時に、プルアップされて H になっていたピンの状態が L に変化することを検知して LED の状態を切り替えるプログラム例が下記である。

```
boolean state = false;

void setup() {
  DDRB |= _BV(DDB5);
  DDRD &= ~_BV(DDD2);
  PORTD |= _BV(PORTD2);
}

void loop() {
  if ( bit_is_clear(PIND, PIND2) ) {
    state = !state;
  }

  if ( state ) {
    PORTB |= _BV(PORTB5);
  }
  else {
    PORTB &= ~_BV(PORTB5);
  }
}
```

同じプログラムを Arduino 標準ライブラリを用いて書き直すと次にようになる。

```
boolean state = false;
int pinLED = 13;
int pinSW = 2;

void setup() {
  pinMode(pinLED, OUTPUT);
  pinMode(pinSW, INPUT_PULLUP);
}

void loop() {
  if ( !digitalRead(pinSW) ) {
    state = !state;
  }

  if ( state ) {
    digitalWrite(pinLED, HIGH);
  }
  else {
    digitalWrite(pinLED, LOW);
  }
}
```

上記のプログラムに不備はないように思われるが、実際にはスイッチを押しても LED の状態が切り替わらないことが起こる。これは、実際の電圧信号がきれいに H → L というステップ信号にならず、スイッチの可動接点の機械的振動などが原因で HLLHLHLLL というようにばたつくからである。この現象をチャタリングと呼ぶ。チャタリングはジャンパワイヤをブレッドボードに差し込んだ時も起こる。この対策を考えよう。

課題 7

(1) スケッチ stmc2.ino を開き、コンパイルと書きこみを行え。
(2) 入力ピンの状態変化に対して、必ずしも所望の動作をしないことを確認せよ。
(3) 動作しない理由を説明せよ。

チャタリング対策の一つは、遅延を利用して定常状態になるまで待つことである。

課題 8

(1) スケッチ stmc3.ino を開き、コンパイルと書きこみを行え。
(2) チャタリング防止はどのように実装されているか把握せよ。
(3) 入力ピンの状態変化による LED の点灯・消灯が安定して行えるように、適切な遅延時間を設定せよ。
ただし、全く誤りが起こらないようにすることは難しい。

6　組み込みソフトウェア開発 2

入出力ポートに続き，AVR に組み込まれた諸機能を学んでいこう．

6.1　割り込み

6.1.1　基本的な考え方

「割り込み」とは，プログラム実行中に何らかのイベントを検知した時にプログラムを一旦中断し，そのイベントに対応する処理を先に実行した後に改めて作業途中のプログラムを再開する仕組みを指す．イベントの発生の有無を定期的に確認する監視処理をプログラムに加えておく方式も考えられる。このような監視をポーリングとよぶ。いつ起こるかわからないイベントを高頻度で監視するのは処理のムダであり、一方で低頻度で監視を行うとイベント発生の検知が遅れる可能性がある。近年のマイコンは計算資源が潤沢であるため，時間要求が厳しくない条件ではポーリングでも対応できてしまうが，割り込みを利用する方が合理的な処理は多い．代表的な割り込みの利用例を下記に示す。

- ボタンが押されたことなどの即時検知
- 内部タイマーを使った正確な周期での処理実行
- データ受信のタイミングのわからない通信

AVR における割り込みの動作の仕組みについて説明する．AVR では電源が投入されプログラムが起動すると（プログラム 0 番地から実行が開始され,）main 関数が実行されることになっている．main 関数に記述した処理を実行中，割り込みが発生した場合，その割り込み発生源に対応した処理を行う．その際，プログラムメモリの特定の番地に飛んで処理を実行する．この番地情報を記したものが「割り込みベクタ」である．割り込み処理の終了後，割り込み発生直前に実行していたプログラムの番地に戻り処理を再開する（図 14）.

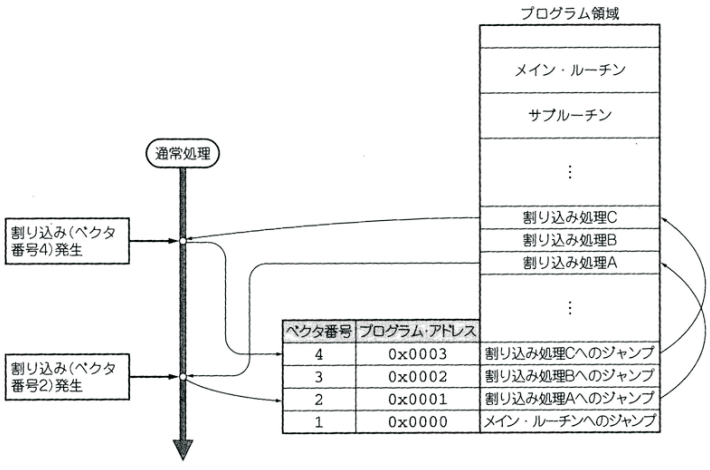


図 14　AVR における割り込み処理の実現

6.1.2 ATmega328P 割り込み機能の役割

ATmega328P の割り込みベクタを表 1 に示す.

表 1 ATmega328P の割り込みベクタ一覧（Atmel 社データシート 66 ページより引用）

VectorNo.	Program Address	Source	Interrupt Definition
1	0x0000	RESET	External Pin, Power-on Reset, Brown-out Reset and Watchdog System Reset
2	0x0002	INT0	External Interrupt Request 0
3	0x0004	INT1	External Interrupt Request 1
4	0x0006	PCINT0	Pin Change Interrupt Request 0
5	0x0008	PCINT1	Pin Change Interrupt Request 1
6	0x000A	PCINT2	Pin Change Interrupt Request 2
7	0x000C	WDT	Watchdog Time-out Interrupt
8	0x000E	TIMER2 COMPA	Timer/Counter2 Compare Match A
9	0x0010	TIMER2 COMPB	Timer/Counter2 Compare Match B
10	0x0012	TIMER2 OVF	Timer/Counter2 Overflow
11	0x0014	TIMER1 CAPT	Timer/Counter1 Capture Event
12	0x0016	TIMER1 COMPA	Timer/Counter1 Compare Match A
13	0x0018	TIMER1 COMPB	Timer/Coutner1 Compare Match B
14	0x001A	TIMER1 OVF	Timer/Counter1 Overflow
15	0x001C	TIMER0 COMPA	Timer/Counter0 Compare Match A
16	0x001E	TIMER0 COMPB	Timer/Counter0 Compare Match B
17	0x0020	TIMER0 OVF	Timer/Counter0 Overflow
18	0x0022	SPI, STC	SPI Serial Transfer Complete
19	0x0024	USART, RX	USART, Rx Complete
20	0x0026	USART, UDRE	USART, Data Register Empty
21	0x0028	USART, TX	USART, Tx Complete
22	0x002A	ADC	ADC Conversion Complete
23	0x002C	EE EADY	REEPROM Ready
24	0x002E	ANALOG COMP	Analog Comparator
25	0x0030	TWI	2-wire Serial Interface
26	0x0032	SPM READY	Store Program Memory Ready

この表に示した割り込みのうち，本演習で取り扱う割り込みの概要と利用例を示す.

- INT0, INT1：外部変化割り込み
  INT0，INT1 に入力される電圧が低い，もしくは H から L，L から H へ変化したときを検知
- TIMER0/1/2 OVF：タイマカウンタオーバーフロー割り込み
  （例えば）一定間隔の処理を実現するときに使う
- TIMER0/1/2 COMPA/COMPB：タイマカウンタ比較一致割り込み
  （例えば）一定間隔の処理を実現するときに使う
- TIMER1 CAPT：タイマカウンタキャプチャイベント
  外部ピンの電圧変化が発生したことを検知し，さらにその発生時刻の記録に役立つ
- USART, RX：USART 読み込み完了割り込み
  外部デバイスとの通信手段の 1 つである USART で外部からの通信が終了したことを検知
- ADC：A/D 変換終了割り込み
  A/D 変換が終了することを検知[*4].

[*4] A/D 変換はその仕組み上，13us〜260us 変換時間を必要とする．この場合，（Arduino でいえば 200〜4000 ステップ時間）の待機を意味するため，計算資源の効率化に有効といえば有効である.

演習では扱わないが，プログラマが意識しないで利用している重要な割り込みに，リセット割り込みがある．Arduino で言えばリセットボタンをスイッチを押すとリセットピンには 0V が入力される，この状態が続くとプログラムは 0 番地までに戻って処理を再開する．Arduino の場合リセットボタンを押すと main 関数が再開されることになる．また，AVR の ISP 書き込みは，書きこみの開始時に ISP コネクタのリセットピンにかける電圧を 0 にすることでリセットボタンを押した状態にする．AVR がリセット状態にある間にフラッシュメモリへの書きこみを実行する．

6.1.3　割り込み要求フラグ，特定機能割り込み許可，全割り込み許可

A/D 変換は，それに伴う処理時間が気にならないこともある．そのときには「割り込み要求フラグ」とよばれる，特定のレジスタの完了状態を示すビット値を監視することで完了を確認することができる．この役割を担うビットを「割り込み要求フラグ」と呼ぶ．

「割り込み要求フラグ」は「機能別割り込み許可」や「全割り込み許可」をしていなくても特定の処理が終わり次第，セットされる．例えば A/D 変換終了の割り込みでは A/D 変換が完了したタイミングで「A/D 変換完了割り込み要求フラグ」がセットされる．その際，「A/D 変換完了割り込み許可」と「全割り込み許可」がセットされていれば，実際に割り込みが発生する．全割り込み許可を設定するには SREG というステータスレジスタの I（アイ）ビットと呼ばれるビットをセットすればよい（Atmel 社のデータシート 534 ページ参照）．以下に示す `sei(): set interrupts flag` という関数や `cli(): clear interrupts flag` という関数は I ビットをセットする，クリアする，という意味である．

6.1.4　AVR GCC における割り込み処理の設定の流れ

AVR GCC で割り込み処理を実現する場合の流れを考えてみる．

1. プログラムの仕様にあった割り込みを選ぶ
2. 割り込み要求を有効にする（機能別割り込み許可ビットをセットする）
3. 全割り込み許可をセットする．
4. 割り込みが発生時の割り込み処理を記述

外部割込みの利用例を示す．

```
#include <avr/io.h>
#include <avr/interrupt.h> //割り込み処理する場合インクルードする必要アリ

volatile int counter = 0;

//INT0 割り込みが発生したときに処理する関数．
ISR(INT0_vect)
{
  counter++;
}

int main(int argc, char* argv[])
{
  ...
  EIMSK = _BV(INT0); //INT0 割り込みを許可する．
  sei();//以降の処理で全割り込みを許可する．
  while(1);
  return 0;
}
```

大まかには

- 機能別割り込み許可：特定の割り込み処理が実行されるよう，関連するレジスタを設定
- 全機能割り込み許可：割り込み処理が行われるように割り込み許可 `sei`[*5]を実行
- `ISR` と書かれた関数で割り込み発生時の処理を記述

を行えばよい．なお，表 1 には INT0 とあるが，一方 C 言語プログラムには `INT0_vect` とある．INT0 割り込み用の割り込みベクタのマクロであるが，実際には 328P 用のヘッダである iom328p.h に

```
#define INT0_vect          _VECTOR(1)   /* External Interrupt Request 0 */
```

とあり，`_VECTOR(1)` は avr/sfr_def.h において，

```
__vector_1
```

として展開される．`__vector_1` は割り込みベクタ番号 1（プログラミングは 0 始まりなので表 1 における VectorNo. 2 の割り込み）を指す．上記の例では「フラッシュメモリ上の 0x0002 番地から {} で囲われた関数を呼び出す」なるコードを書いていると思えばよい．

6.1.5 AVR GCC における ATmega328P の割り込みベクタのマクロ

ATmega328P の割り込みベクタを表すマクロを示す．引用元は ATmega328P のマクロが書かれたヘッダファイル iom328p.h である

```
#define INT0_vect          _VECTOR(1)   /* External Interrupt Request 0 */
#define INT1_vect          _VECTOR(2)   /* External Interrupt Request 1 */
#define PCINT0_vect        _VECTOR(3)   /* Pin Change Interrupt Request 0 */
#define PCINT1_vect        _VECTOR(4)   /* Pin Change Interrupt Request 0 */
#define PCINT2_vect        _VECTOR(5)   /* Pin Change Interrupt Request 1 */
#define WDT_vect           _VECTOR(6)   /* Watchdog Time-out Interrupt */
#define TIMER2_COMPA_vect  _VECTOR(7)   /* Timer/Counter2 Compare Match A */
#define TIMER2_COMPB_vect  _VECTOR(8)   /* Timer/Counter2 Compare Match A */
#define TIMER2_OVF_vect    _VECTOR(9)   /* Timer/Counter2 Overflow */
#define TIMER1_CAPT_vect   _VECTOR(10)  /* Timer/Counter1 Capture Event */
#define TIMER1_COMPA_vect  _VECTOR(11)  /* Timer/Counter1 Compare Match A */
#define TIMER1_COMPB_vect  _VECTOR(12)  /* Timer/Counter1 Compare Match B */
#define TIMER1_OVF_vect    _VECTOR(13)  /* Timer/Counter1 Overflow */
#define TIMER0_COMPA_vect  _VECTOR(14)  /* TimerCounter0 Compare Match A */
#define TIMER0_COMPB_vect  _VECTOR(15)  /* TimerCounter0 Compare Match B */
#define TIMER0_OVF_vect    _VECTOR(16)  /* Timer/Couner0 Overflow */
#define SPI_STC_vect       _VECTOR(17)  /* SPI Serial Transfer Complete */
#define USART_RX_vect      _VECTOR(18)  /* USART Rx Complete */
#define USART_UDRE_vect    _VECTOR(19)  /* USART, Data Register Empty */
#define USART_TX_vect      _VECTOR(20)  /* USART Tx Complete */
#define ADC_vect           _VECTOR(21)  /* ADC Conversion Complete */
#define EE_READY_vect      _VECTOR(22)  /* EEPROM Ready */
#define ANALOG_COMP_vect   _VECTOR(23)  /* Analog Comparator */
#define TWI_vect           _VECTOR(24)  /* Two-wire Serial Interface */
#define SPM_READY_vect     _VECTOR(25)  /* Store Program Memory Read */
```

[*5] 全割り込み禁止の場合，`cli` と書く．

6.1.6 外部割り込み

割り込みの簡単な例題として INT0, INT1 に相当する「外部割込み」の機能を実際に体験してみよう．外部割込みは INT0, INT1 に接続されたピンの電圧状態に応じて割り込みを発生させるものであり，これまでに行ってきたデジタル入力の効率化を図る．演習ではスイッチの入力を検出する例を扱う．実際にはこの例はポーリングで置き換えても問題ない．一方以下の場合，外部割込みは有効である．ATM 等のシステムにおいてサービス利用者が不在の場合，システムの消費電力を抑えスリープ状態にしておく．サービス利用者が画面をタッチすると，スリープ状態から復帰しサービスを再開するという流れである．

外部割り込みの一例として，スイッチを押されると LED が点灯する例を見てみよう．

オプション課題 2

(1) プロジェクト ext1 を開け
(2) ビルド・書き込みを行え
(3) 動作を確認せよ．

以下ではタイマと A/D 変換の使い方を学びつつ、あわせて割り込みの使用法を学んでいこう。

6.2 タイマ/カウンタ

6.2.1 利用

タイマ/カウンタは CPU のクロックが 1 ステップ進むごとに自動的にカウントアップされる．その値をレジスタを介して読み書きすることで，一定周期で処理を行う場合に利用できる。

一定時間毎に処理を行うために遅延を使って待機する方法も考えられる。この場合，プログラムは逐次的であり直感的である一方，遅延を発生させるためだけに AVR の計算資源が消費されてしまう。また、遅延時間をどれだけ設ければ所望の周期を実現できるかは、遅延以外の計算処理時間に左右される。さらに，異なる周期で複数の処理を実行したい場合，遅延処理では実装が困難になる．このような状況から，時間を強く意識する必要がある組み込み系ではタイマの活用は重要である．

タイマの役割は一定間隔毎の処理だけではない．代表的なタイマの応用として考えられるのが，波形生成である。その中でも重要なのがパルス幅制御（Pulse Width Moduration: PWM）である．AVR では入出力ポートの一部のピン（ATmega328P では計 6 ピン）が PWM 出力に利用可能である．

6.2.2 ATmega328P のタイマの概要

■ノーマルモードと CTC モード カウンタは CPU で計算が実行されるたびに値が増えていく．時計の秒針と同様，一定時間が過ぎれば 0 にリセットされ、これを繰り返す．具体的にはカウンタが 8 ビットでその値の上限が 255 であるとすれば，255 の次のステップで値は 0 になり再び 1，2，3 へと値が増加していく．0 にリセットされるタイミングで発生する割り込みを「オーバーフロー割り込み」とよぶ．オーバーフロー割り込みに対し処理を記述すれば，その処理は一定周期で行われることになる．カウンタを格納するレジスタのビット数で「8 ビットタイマ」といったり「16 ビットタイマ」とよぶ．このように，カウンタが 0 からレジスタがとりえる上限値（8 ビットタイマの場合 0xFF，16 ビットタイマの場合 0xFFFF）まで値が増え，0 に戻る状況をノーマルモードと呼ぶ（図 15）．

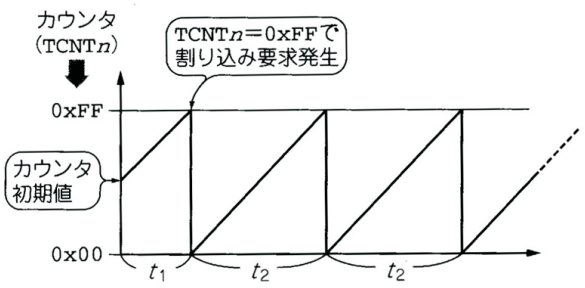


図 15　タイマカウンタの振る舞い（ノーマルモード）

ところで、CPU のクロックは高速なので、1 ステップごとにカウンタを増やしてしまうと，高い頻度でオーバーフローが起きてしまう．アプリケーションによっては頻度を落とすべき場合がある．このときに便利な機能がプリスケーラ（分周比）である（図 16）．プリスケーラを 1024 にした場合，CPU が 1024 ステップ進むごとにカウンタが 1 つ増加するものであり，つまり，カウンタが 0 から最大値 0xFF を経由しオーバーフロー割り込みが発生するまでに 1024 × 256 ステップの CPU の実行時間が必要になる．CPU クロック周波数を 16MHz とすると

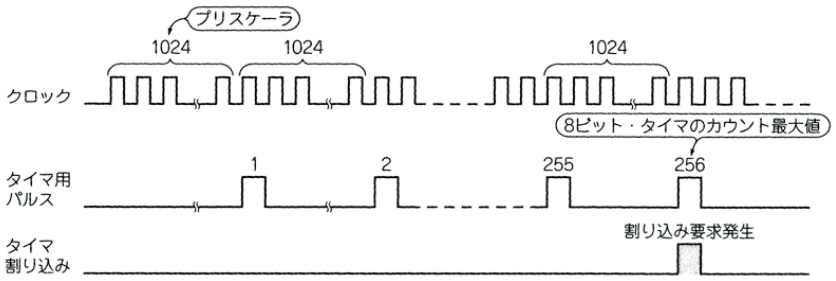


図 16　タイマカウンタのプリスケーラ（1024 の場合）

タイマの利用により一定時間間隔に処理を行うことが簡単になったが，所望の間隔で処理を行うことはできない（例えば 10ms）．次の 2 つの方法でこの問題は解決する．

- ノーマルモードにおいて，オーバーフロー割り込みが発生した際，カウンタの初期値を 0 から適当な値に置き換え，再度オーバーフローが発生するタイミングを調整する方法（図 17）
- **CTC（Clear Timer on Compare）モード**とよばれる比較レジスタ（Output Compare Register）を利用する方法
  カウンタと比較レジスタとの大小関係をチェックし，一致した直後，カウンタが増分するタイミングで 0 に戻すというものである（図 18）．なお，0 に戻すタイミングでタイマカウンタ比較一致割り込みが発生する．

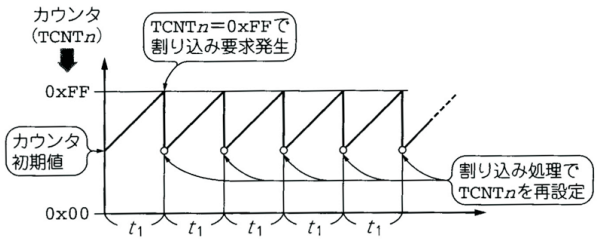


図 17　タイマカウンタの振る舞い（ノーマルモード）

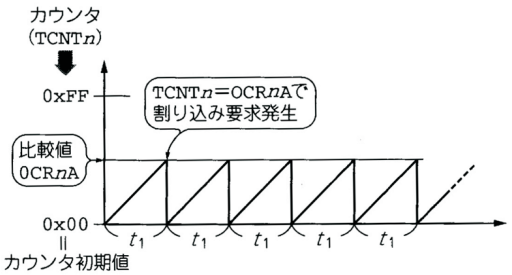


図 18　タイマカウンタの振る舞い（CTC モード）

データシートでは，カウンタの最小値（一般に 0）を BOTTOM とよび，8 ビットカウンタなら 0xFF，16 ビットの場合は 0xFFFF を MAX，ノーマルモードにおける 0xFF，CTC における比較レジスタの値を TOP と呼んでいる．TOP 値は言い換えれば BOTTOM になる直前の値を指す．

■PWM　タイマでは，比較レジスタを利用し，デジタル波形を生成することもできる．適当なピンの電圧を，例えば，

- カウンタが比較レジスタの値以下なら H 出力
- カウンタが比較レジスタの値以上なら L 出力

というルールに則り制御することができれば，比較レジスタの値を変更するだけでパルス幅制御（PWM）が実現する．カウンタの TOP を MAX（8 ビットタイマの場合 0xFF）にした場合，H もしくは L のパルス幅を制御するに留まるが，カウンタの TOP を CTC モードにより調整すれば任意の周波数の PWM が生成できる．ATmega328P ではタイマ 0，タイマ 1，タイマ 2 のそれぞれに関連付けされた出力ピン（合計 6 つ）が用意されている（Output Compare outputs：OCnx とよばれる）．様々な動作モードが存在するので本資料では網羅が難しいが，代表的なモードとして，高速 PWM と位相基準 PWM の概要のみ示す．それぞれ図 19 と図 20 のような挙動をする．

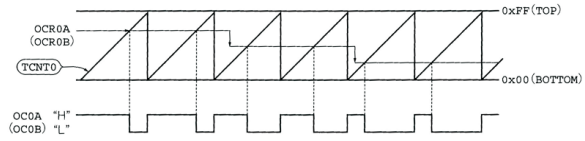


図 19　高速 PWM

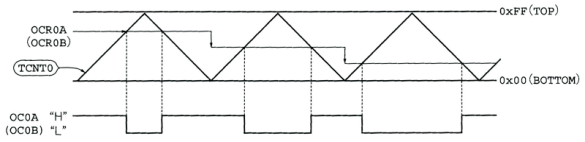


図 20　位相基準 PWM

高速 PWM とは今までどおりカウンタが TOP を超えると 0 に戻るのに対し，位相基準 PWM では TOP の次のステップでカウンタは TOP-1 になり，0 になるまで 1 つずつデクリメントするものになっている．

6.2.3　タイマ 0 / タイマ 2 のプログラミング

タイマ 0 ならびにタイマ 2 は 8 ビットタイマである．CPU 周波数 16MHz で使えば 1 周 16us の時計として利用できる．以下の機能・要素に着目し利用する．

1. 割り込みフラグの監視
2. オーバーフロー割り込みの設定

3. プリスケーラの設定：1，8，64，256，1024
4. 比較一致機能
5. 波形生成の設定

タイマ 2 は基本的にはタイマ 0 と同様の機能を持つと思えば良い．以下，タイマ 0 を前提とした説明を行う．

実際のプログラミングでは以下の 5 種（正確には 7 個）の I/O レジスタを制御・監視すればよい．

- Timer/Counter Control Register：TCCR0A, TCCR0B
- Timer/Counter：TCNT0
- Output Compare Register：OCR0A, OCR0B
- Timer Interrupt Flag Register: TIFR0
- Timer Interrupt Mask Register: TIMSK0

TCCR0 はタイマのモードを設定するために，TCNT0 は文字通り時計の針の情報を，OCR は比較一致機能を利用する際，一致タイミングを調整するため，TIFR0 は時計が一周すると特定のビットがセットされるため，一定周期時間が経過したことを確認するために用いられ，一方，TIMSK0 はオーバーフローもしくは比較一致時に割り込みを発生させたいときに制御するためのレジスタである．それらの設定についての情報は，Atmel 社のデータシート 107 ページ（14.9 節）が参考になる．タイマについては多くのモード・使用法が存在し，演習で全てをカーバすることは困難である．従って，各モードに関する解説は避け，基本的な使用法を例題から学んでもらうことにする．

■タイマ 0 の例 1：ノーマルモード（一定周期毎の処理，ポーリングの利用） 一定時間毎にカウントアップを行わせる処理を書いてみよう．タイマ 0 のカウンタである TCNT0 が 0 から 255 まで変化し，0 になるごとにプログラムのステータスを 1 つずつ増やし，16 段階の表示を行わせるものを考える．TCNT0 が 0 から 255 まで増分することを繰り返す状態をノーマルモードとよぶ．

CPU クロック周波数 16MHz で駆動しているとき，プリスケーラを 1024 とするとステータスが 1 増分するには_____秒要する．16 回オーバーフローするごとに 1 カウントを増加させるものと考えると，16 段階の LED 表示の周期は_____秒である．プリスケーラを 1024 とするときにはデータシート 111 ページ Table14-9 を参考にすると TCCR0B レジスタの Clock Select ビット CS02，CS01，CS00 をそれぞれ 1，0，1 にすればよい．

次の処理を行うことで所望のプログラムを実現してみよう．

1. ポートの設定
2. タイマの初期化
   - TCCR0A は今のところ何も設定する必要ない．
   - TCCR0B のプリスケール CS02:CS00 を設定．
   - TCNT0 の初期値を 0 にする．
3. 無限ループ内で LED 状態を制御．
   - TCNT0 がオーバーフローする (255 から 0 になる瞬間) まで待機．
     オーバーフロー時に TIFR0 レジスタの_____というビットがセットされることを利用．
   - オーバーフローの回数に応じて LED の点灯状態を制御

- ____をセットする.
  そのタイミングでそのビットがクリアされるので，オーバーフローまで待機となる[*6].

この振る舞いを実現するために途中まで実装した timer1 を完成させてみよう.

オプション課題 3

(1) ファイル timer1.ino を開け.
(2) timer1.ino を完成させる前に TIFR0 レジスタについて調べよ.[a]
(3) 適切なコードを足して完成させ、コンパイル・書き込みを行い動作を確認せよ．Arduino IDE のシリアルモニタ起動（画面右上のアイコン）して通信速度を 57600 に設定すると、割り込み毎に出力が送られくるのがわかる。
(4) 分周比[b]を変更し，1 秒周期の表示を作成してみよ．ただし，誤差は ±0.1 秒まで認める.
(5) この方法は厳密に正確といえない．その理由を考えよ.
(6) 時間の厳密性の他，この方法にはどのような問題があるか考えよ.

[a] データシート 112 ページの 14.9.7 が参考になる
[b] タイマ 0 の場合データシート 111 ページ Table14.9 が参考になる

■タイマ 0 の例 2：ノーマルモード（一定周期毎の処理，割り込み利用） 上述の timer1.ino ではタイマのオーバーフローを検知するために loop_until_bit_is_set を行い待機していた．遅延処理と比べ，待機時間以外の計算コストに依存せず正確に一定周期の処理が実現できるようになったが，タイマのそもそもの役割でも述べた計算リソースの効率化は実現していない．そこで割り込みを利用することを考える．上記と同様のカウントアップを繰り返すプログラムを割り込みにより実現してみよう.

1. ポートの設定
2. タイマの初期化
   - プロジェクト timer1 と同様の初期化.
   - TCNT0 がオーバーフローした際に割り込みが可能なように TIMSK0 レジスタの_____をセットする.
3. タイマ 0 のオーバーフロー割り込み他，全ての割り込みを許可するために sei(); という関数を呼び出す.（割り込みを禁止したいときは cli(); という関数を呼び出す.）
4. 無限ループ内でステータスを制御.
   - プロジェクト timer1 とは異なりオーバーフロー検知を行わない.
5. 割り込み関数
   ステータスを更新し，カウントを表示.

以下のような割り込み関数と sei(); による割り込み許可が必要である.

```
ISR(TIMER0_OVF_vect)
{
...
```

[*6] データシート 112 ページ，113 ページが参考になる．特に 113 ページ目の Alternatively, 以下の文は重要である

```
}
```

一方 main 関数で実行される制御構造は簡素になることに注目して欲しい．この振る舞いを実現するために途中まで実装した timer2 を完成させてみよう．

課題 9

(1) ファイル timer2.ino を開け．
(2) timer2.ino を完成させる前に TIMSK0 レジスタについて調べよ．[a]
(3) 適切なコードを足して完成させ、コンパイル・書き込みを行い動作を確認せよ。割り込みが起こるごとに数字がシリアル通信で USB ポートを通じて PC に送られるので，Arduino IDE のシリアルモニタ（画面右上の虫眼鏡のようなアイコン）を使って受信する．プログラム内で通信速度を 57600bps に設定しているので，シリアルモニタの速度も同じ値にすること。通信速度が一致しないと文字化けが起こる．新しいバージョンの Arduino IDE では、シリアルモニタ左下「タイムスタンプを表示（Show timestamp）」にチェックを入れると便利。

[a] データシート 112 ページの 14.9.6 が参考になる

■タイマ 0 の例 3：カウンタ値の調整による任意周期の時計の作成 正確に 1 秒周期の動作を行いたい場合どのようにすればよいか．そのときは割り込みが発生したときに TCNT0 の値を適切な値に変更すればよい．この場合，割り込み発生時に TCNT0 の値を 0 から_____にすればよい．それを実現するプロジェクトが timer3 である．

課題 10

(1) ファイル timer3.ino を開け．
(2) カウント周期（シリアルポートの表示が 16 カウントする周期）が 1 秒になるように timer3.ino を完成させよ。割り込み周期と、値をシリアル送信する周期が一致していないことに注意する。ここでは誤差 ±0.01 秒以内で実現せよ．
(3) 適切なコードを足して完成させ、コンパイル・書き込みを行い動作を確認せよ。Arduino IDE のシリアルモニタを使う。

■タイマ 0 の例 4：CTC モード（任意周期の時計の作成） 例 3 では TCNT0 の値を直接制御することで任意周期を作成しようとしていたが，今度は CTC 機能を利用してみよう．timer3.ino との違いは

- TCCR0A を CTC モードに設定する．そのために_____の 3 ビット分のうち[*7]，TCCR0A に関連する 2 ビット分を適宜設定する必要がある．
- 比較一致させるための上限値（TCNT0 をいくつを超えたら 0 に戻すのかの設定値）を決める必要がある．そのレジスタは_____である[*8]．
- 比較一致が起きたときに割り込みを発生させる必要がある．TIMSK0 の____というビットを設定する．
- 割り込み発生時のハンドラ関数を TIMER0_OVF_vect から TIMER0_COMPA_vect に変更する必要がある．

[*7] データシート Table14-8 参照
[*8] データシート 14.9.4 参照

である．

課題 11

(1) ファイル timer4.ino を開け．
(2) CTC 動作を実現するためにレジスタを適宜設定せよ．
(3) 1 秒周期のカウントが実現するように TCNT0 の上限値を適宜設定せよ．
(4) 適切なコードを足して完成させ、コンパイル・書き込みを行い動作を確認せよ。Arduino IDE のシリアルモニタを使う。

例 3 に比べ，TCNT0 を 0 にするタイミングにズレがなく，より正確に所望の周期のタイマを作れることになる．

6.2.4 タイマ 1 のプログラミング

タイマ 1 については解説のみにとどめる。タイマ 1 は 16 ビットタイマである．タイマ 0 と比較しより長周期の時計を作成することができる．CPU 周波数 16MHz で使えば 1 周 4096us ≒ 4ms の時計として利用できる．以下の機能・要素に着目し利用する．タイマ 0 とほぼ同様の機能である．タイマ 1 のカウンタである TCNT1 が 16 ビット分占有することが違いといえる．

1. 割り込みフラグの監視
2. オーバーフロー割り込みの設定
3. プリスケーラの設定：1，8，64，256，1024
4. 比較一致機能
5. 波形生成の設定
6. 入力捕獲機能

最後の入力捕獲機能がタイマ 0 ではできない，独自の機能である．実際のプログラミングでは以下の 6 種（正確には 9 個）の I/O レジスタを制御・監視すればよい．

- Timer/Counter Control Register：TCCR1A, TCCR1B, TCCR1C
- Timer/Counter：TCNT1
- Output Compare Register：OCR1A, OCR1B
- Timer Interrupt Flag Register: TIFR1
- Timer Interrupt Mask Register: TIMSK1
- Input Capture Register: ICR1

殆どが 0 から 1 に置き換わったものである．実際には TCNT1 は 16 ビットレジスタであり，詳細には TCNT1H と TCNT1L の 2 つの 8 ビットレジスタをまとめて読み書きできるようにしたマクロと思えば良い．同様に OCR1A や OCR1B，ICR1 も 16 ビットレジスタである．これらは Atmel 社のデータシート 533 ページ目を見ると，0x85 と 0x84 番地が TCNT1H と TCNT1L であり，一方 ATmega328P に関するレジスタのマクロ情報が記載された iom328p.h を見てみると，525 行目に

```
#define TCNT1 _SFR_MEM16(0x84)
#define TCNT1L _SFR_MEM8(0x84)
...
```

```
#define TCNT1H _SFR_MEM8(0x85)
```

とある．つまり 0x84 番地から 16 ビット分の変数とみなしたポインタの中身とみるか，8 ビット分のポインタと見るかで TCNT1 と TCNT1L が異なるということになる．

タイマ 1 の設定についての情報は，Atmel 社のデータシート 135 ページ（15.11 節）が参考になる．

6.3　A/D 変換

6.3.1　ATmega328P の A/D 変換の概要

ATmega328P に内蔵された A/D コンバータ（以後 ADC）は 10 ビット分解能である．この ADC は逐次比較型動作で動作している．変換時間は 13us から 260us となっている．A/D 変換器そのものは 1 つであるため，複数のアナログ信号を扱えるようにするために ADC の前にアナログマルチプレクサが置かれている．このマルチプレクサはポート C の一部に接続されている（PC0〜PC5 の 6 ピン）．これにより 6 つのアナログ信号を読み取ることができるようになるが，一度に変換できる信号はこれら 6 つのうち 1 つに限られる．

精度良いアナログ信号の計測にはノイズ低減が必須である．具体的なノイズ低減の方法については本演習のスコープから外れるため説明は省略させてもらうが，Atmel 社のマニュアルの 23.6 節（257 ページ）が参考になるだろう．

ADC 関連のレジスタは以下の通りである．

- ADC Control & Status Register：`ADCSRA, ADCSRB`
- ADC Data Register：`ADC(ADCH, ADCL)`
- ADC Multiplexer Select：`ADMUX`

`ADCSR` は AD 変換のモードを設定したり AD 変換の状態を監視するためのレジスタであり，`ADMUX` は ADC の際の基準電圧やチャンネルを選択するためのレジスタであり変換後の値は `ADC` に入る．タイマと同じく基本的な使用法を例題から学んでもらうことにする．

■ADC の例 1：ポーリングによるサンプリング　今回は PC0（A0）に入力されたアナログ電圧の値によって LED の輝度を変えるプログラムを書いてみよう．

プロジェクト adc1 では次の流れでプログラムを書いている．

1. ADC の設定．この場合 `ADCSRA` と `ADMUX` を設定する．
2. `ADCSRA` の＿＿＿ビットをセットすると変換が開始
3. `ADCSRA` の＿＿＿ビットがクリアもしくは＿＿＿＿＿ビットがセットされるまで待機．
4. 変換データを取り出す．変換データは `ADC` に入っている．
5. 以上を繰り返す．

可変電圧を作るために、ここでは可変抵抗器を使った分圧回路を使う（図 21）。

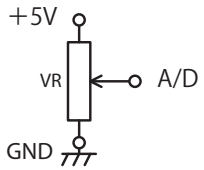


図 21　可変抵抗を使った分圧回路

この流れを実現するものが adc1 というプロジェクトである．

課題 12

(1) Arduino を USB ポートから抜き、ブレッドボード上で半固定抵抗器（トリマー）を使った分圧回路を構成して、可変電圧を A/D 変換ポートに接続せよ。半固定抵抗器のピン配置はデータシートを参照すること。
(2) プロジェクト adc1 を開け．
(3) adc1.ino を開き，関連するレジスタに適切な値を設定せよ[a]．
(4) ビルド・書きこみを行え．動作を確認せよ．シリアルモニタで AD 変換の結果が確認できる。アナログ電圧に応じて LED の輝度が変わることを確認せよ。+5V を入力すると LED の輝度最大、GND を入力すると輝度最小になる。

[a] データシートの ADMUX，ADCSRA（263 ページ 23.9 参照）

■ADC の例 2：割り込みによるサンプリング　効率よく A/D 変換を行うために，割り込みを使った方法をプロジェクト adc2 で紹介する．まずはフリーランニングモードとよばれる連続変換を行う状況で，変換終了後にデータを読み込む方式を採用する．adc1 との違いは

- ADCSRA で割り込み許可を設定する．オートトリガーモードにする．オートトリガーモードにするには ADCSRA レジスタの＿＿＿ビットをセットすればよい．
- オートトリガーモードのソースを選択する．
- 割り込み関数を記述する．
- main 関数では割り込み許可を行う．

オプション課題 4

(1) プロジェクト adc2 を開け．
(2) adc2.ino を開き，関連するレジスタに適切な値を設定せよ[a]．
(3) ビルド・書きこみを行え．動作を確認せよ．

[a] データシートの ADMUX，ADCSRA（263 ページ 23.9 参照）

■ADC の例 3：タイマ駆動サンプリング　上記の割り込みはかなり高頻度になることが予想される．むしろ 1KHz，100Hz といった一定の周期でのサンプリングが組み込み系で求められることが大半である．オートトリガモードのトリガソースを置き換えることで一定周期の AD 変換を実現してみよう．adc2 との違いは

- タイマ 0 を起動.
- オートトリガーのソースをタイマ 0 のオーバーフロー割り込みに設定.
- タイマ 0 のオーバーフロー発生時にタイマカウンタを適宜設定し，所望の周期のサンプリングが可能なように設定する.

オプション課題 5

(1) プロジェクト adc3 を開け.
(2) adc3.ino の一部の設定を書き換え，（約）100Hz のサンプリングを実現 (確認) せよ.
(3) ビルド・書きこみを行い動作を確認せよ.

7　組み込みソフトウェア開発 3

これまで学んできたタイマ，A/D 変換，割り込み等のプログラミングの知識だけでも，様々なセンサ・アクチュエータを利用することができる．本節では，演習の初期段階で一度体験しておくとよいと思われる例題として，サーボモータ制御を選定した．

7.1　サーボモータの回転制御

サーボモータは，位置・速度を制御するためのフィードバック制御機構を有するモータである．本演習では，ホビー用途のラジコン用サーボモータを利用する。

最も一般的な RC サーボは PWM (pulse width modulation) によるものである。この場合、RC サーボの接続は GND と電源、信号線の 3 本である。0.8〜2ms 程度のパルス幅の H 信号を 20ms 周期（50Hz）を作って信号線に加える（図 22)。このときパルス幅に応じて RC サーボの出力軸の角度が変わる。なお、PWM 制御方式以外に、シリアル通信方式の RC サーボもあるので実際の購入や利用では注意すること。

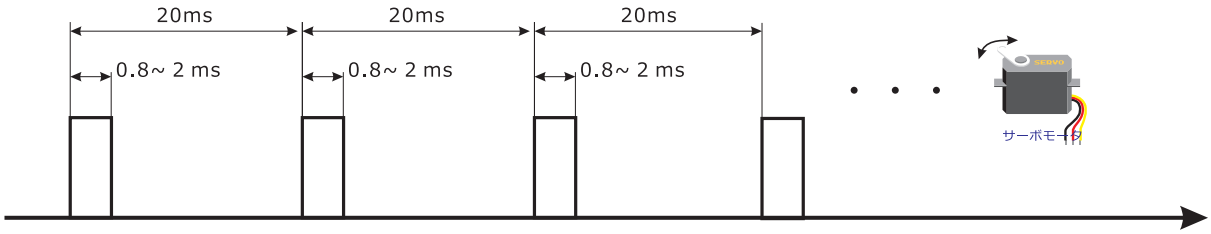


図 22　サーボ制御のための PWM 信号

小型の RC サーボであれば Arduino に接続した USB バスパワーで駆動できるが，電流が十分でないとサーボホーンが振動したり，Arduino が頻繁にリセットしたりすることがある．一般には RC サーボ用の電源を別途用意する．

マイコンの機能を使って実際に 20ms 周期の PWM をどのように作ればよいか．一番簡単な方法として、8bit のタイマ 0 ではなく 16bit のタイマ 1 を Mode14（データシート Table 15-4 参照）にして，ICR1 レジスタを適切に設定すれば 20ms 周期の PWM が作れる．H レベルのパルスの幅は OCR1A レジスタを書き換えればよい．

課題 13

(1) サーボのケーブルをブレッドボードに正しく接続せよ．信号線 1 本と電源 2 本の合計 3 本である．ケーブルの色との対応は，各自調べること．
(2) プロジェクト servo を開け．servo.ino の処理内容を理解せよ．
(3) タイマ 1 の挙動についてデータシートを参考に適切に値を設定せよ．タイマ 1 は 8 ビットでないことに注意せよ．
(4) 周期関数とアナログ入力に従ってサーボモータが動作するように servo.ino を完成させよ．
(5) 動作を確認せよ。アナログピン A0 に GND や +5V を入力した時の動作を考察しよう。何も接続していない時はノイズが入力される。