AVR 8ビット マイクロ コントローラ

AVR42787 : AVRソフトウェア使用者の手引き

応用記述

序説

Atmel® AVR®コアは低メモリ使用量でCコード を効率的に走らせるように作成された強化されたRISC構造です。

この資料はtinyAVR®、megaAVR®、XMEGA®のMCUに適用します。この資料はAVRコード 開発で初めて及び中間のAVR開発者を手助けする、いくつかの頻繁に使用される関数、一般的な意味、よくある質問を記述します。

目次

1. AVR 8ﾋﾞｯﾄ構造

AVR構造はﾊｰﾊﾞｰﾄﾞ構造に基づきます。これはﾌﾟﾛｸﾞﾗﾑとﾃﾞｰﾀに対して独立したﾒﾓﾘとﾊﾞｽを持ちます。これはﾌﾟﾛｸﾞﾗﾑとﾃﾞｰﾀを同時に取得することを可能にします。それは単一ｸﾛｯｸ周期ｱｸｾｽ時間を持つ32個の8ﾋﾞｯﾄ高速ｱｸｾｽ汎用作業ﾚｼﾞｽﾀを持ちます。これらのﾚｼﾞｽﾀで算術や論理の命令が実行できるように、このﾚｼﾞｽﾀはALUに接続されます。1ｸﾛｯｸ周期でAVRは2つの任意のﾚｼﾞｽﾀからﾃﾞｰﾀをALUに送り込み、操作を実行して結果をﾚｼﾞｽﾀに書き戻すことができます。

ﾌﾟﾛｸﾞﾗﾑ ﾒﾓﾘ内の命令は単一ﾚﾍﾞﾙのﾊﾟｲﾌﾟﾗｲﾝで実行されます。1つの命令が実行さつつある間、次の命令がﾌﾟﾛｸﾞﾗﾑ ﾒﾓﾘから取得されます。この概念は毎ｸﾛｯｸ周期で実行されることを命令に許します。殆どのAVR命令は単一16ﾋﾞｯﾄ語形式を持ちます。全てのﾌﾟﾛｸﾞﾗﾑ ﾒﾓﾘが16または32ﾋﾞｯﾄ命令を含みます。

より多くの詳細については各々のﾃﾞﾊﾞｲｽのﾃﾞｰﾀｼｰﾄの「AVR CPUｺｱ」章を参照してください。

2. AVR GCCとツールチェーン

GCCはGNUｺﾝﾊﾟｲﾗ集合(GNU Compiler Collection)を表します。AVRに使用されるGCC版はAVR GCCと名付けられます。

より多くの詳細についてはGNUｺﾝﾊﾟｲﾗ集合使用者手引書を参照してください。

それはAVRﾏｲｸﾛ ｺﾝﾄﾛｰﾗ用に最終的に実行可能な応用を生成ために共に動く多くの他のﾂｰﾙを取り入れます。このﾂｰﾙ群がﾂｰﾙﾁｪｰﾝと呼ばれます。このAVRﾂｰﾙﾁｪｰﾝでacr_libcは通常の標準Cﾗｲﾌﾞﾗﾘで見つかるのと同じ関数の多くと、AVRに対して特有の多くの追加ﾗｲﾌﾞﾗﾘ関数を提供する重要なCﾗｲﾌﾞﾗﾘを務めます。

より多くの詳細についてはAVR-Libc使用者手引書を参照してください。

3. I/Oﾍｯﾀﾞ ﾌｧｲﾙ

I/Oﾍｯﾀﾞ ﾌｧｲﾙは特定ﾌﾟﾛｾｯｻ用の全てのﾚｼﾞｽﾀ名とﾋﾞｯﾄ名に対する識別子を含みます。それらはﾚｼﾞｽﾀがｺｰﾄﾞで使用されつつある時にｲﾝｸﾙｰﾄﾞされなければなりません。

AVR GCCは各ﾌﾟﾛｾｯｻに対して個別のI/Oﾍｯﾀﾞ ﾌｧｲﾙを持ちます。けれども、実際のﾌﾟﾛｾｯｻ型式は(-mmcu=ﾌﾟﾛｾｯｻ ﾌﾗｸﾞを使用して)ｺﾝﾊﾟｲﾗに対するｺﾏﾝﾄﾞ行ﾌﾗｸﾞとして指定されます。これは通常、makefileで行われます。これはどのﾌﾟﾛｾｯｻ型式に対しても単一のﾍｯﾀﾞ ﾌｧｲﾙだけを指定することを許します。

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

IAR™もどのﾌﾟﾛｾｯｻ型式に対しても単一のﾍｯﾀﾞ ﾌｧｲﾙだけを指定することを許します。

```
#include <ioavr.h>
```

GCCとIARのｺﾝﾊﾟｲﾗはﾌﾟﾛｾｯｻ型式を知り、上の単一ﾍｯﾀﾞ ﾌｧｲﾙを通して引き込んで正しい個別I/Oﾍｯﾀﾞ ﾌｧｲﾙをｲﾝｸﾙｰﾄﾞすることができます。これは1つの全般ﾍｯﾀﾞ ﾌｧｲﾙを指定することだけで、新しいI/Oﾍｯﾀﾞ ﾌｧｲﾙをｲﾝｸﾙｰﾄﾞするために全てのﾌｧｲﾙを変更することなしに応用を別のﾌﾟﾛｾｯｻへ容易に移植できる利点を持ちます。

注: IARはAVRのﾃﾞｰﾀｼｰﾄで使用されるのと同じﾚｼﾞｽﾀ名やﾋﾞｯﾄ名を常に使用する訳ではありません。AVR GCCのI/Oﾍｯﾀﾞ ﾌｧｲﾙとIARのI/Oﾍｯﾀﾞ ﾌｧｲﾙで見つかるﾚｼﾞｽﾀ名の間にはいくつかの不一致があるかもしれません。

4. ﾌﾗｯｼｭ変数

C言語は独立したﾒﾓﾘ空間を持つﾌﾟﾛｾｯｻ用に設計されませんでした。これはﾌﾟﾛｸﾞﾗﾑ(ﾌﾗｯｼｭ)ﾒﾓﾘに属すﾃﾞｰﾀの変数を定義するのに様々な非標準の方法があることを意味します。

AVR GCCはﾌﾟﾛｸﾞﾗﾑ ﾒﾓﾘ内の変数を宣言するのに変数属性を使用します。

```
int mydata[] __attribute__((__progmem__))
```

AVR-Libcも変数属性用に便利なﾏｸﾛを提供します。

```
#include <avr/pgmspace.h>
int mydata[] PROGMEM = ...
```

注: PROGMEMﾏｸﾛは<avr/pgmspace.h>のｲﾝｸﾙｰﾄﾞが必要です。これはﾌﾟﾛｸﾞﾗﾑ空間で変数を定義するための通常の方法です。

IARはﾌﾟﾛｸﾞﾗﾑ ﾒﾓﾘで変数を宣言するのに非標準のｷｰﾜｰﾄﾞを使用します。

```
__flash int mydata[] = ...
```

2つのｺﾝﾊﾟｲﾗ(AVR GCCとIAR)に共通してﾌﾟﾛｸﾞﾗﾑ ﾒﾓﾘで変数を定義する方式を作成する方法もあります。(以下の)これらの定義を持つﾍｯﾀﾞ ﾌｧｲﾙを作成してください。

```
#if (defined __GNUC__)
    #define FLASH_DECLARE(x) x __attribute__((__progmem__))
#elif (defined __ICCAVR__)
    #define FLASH_DECLARE(x) __flash x
#endif
```

このｺｰﾄﾞ断片は使用されつつあるｺﾝﾊﾟｲﾗがGCCかまたはIARかを調べ、使用されつつあるｺﾝﾊﾟｲﾗに基づいて適切な方法を用いてﾌﾟﾛｸﾞﾗﾑ ﾒﾓﾘで変数を宣言するFLASH_DECLARE(x)ﾏｸﾛを定義します。その後にそれを次の様に使用します。

```
FLASH_DECLARE(int mydata[] = ...);
```

AVR GCCではﾌﾗｯｼｭ ﾒﾓﾘのﾃﾞｰﾀを読み戻すのに<avr/pgmspace.h>で定義されるpgm_read_*()ﾏｸﾛを使用してください。全てのﾌﾟﾛｸﾞﾗﾑ ﾒﾓﾘを処理するﾏｸﾛがそこで定義されます。

IARではIARｺﾝﾊﾟｲﾗが自動的にLPM命令を生成するためﾌﾗｯｼｭ ﾒﾓﾘの変数を直接読むことができます。

2つのｺﾝﾊﾟｲﾗ(AVR GCCとIAR)に共通してﾌﾟﾛｸﾞﾗﾑ ﾒﾓﾘ内の変数を読む方式を作成する方法もあります。(以下の)これらの定義を持つﾍｯﾀﾞ ﾌｧｲﾙを作成してください。

```
#if (defined __GNUC__)
    #define PROGMEM_READ_BYTE(x) pgm_read_byte(x)
    #define PROGMEM_READ_WORD(x) pgm_read_word(x)
#elif (defined __ICCAVR__)
    #define PROGMEM_READ_BYTE(x) *(x)
    #define PROGMEM_READ_WORD(x) *(x)
#endif
```

5. 割り込み処理ﾙｰﾁﾝ

5.1. 割り込み処理ﾙｰﾁﾝの宣言と定義

C言語規格は割り込み処理ﾙｰﾁﾝ(ISR:Interrupt Service Routines)の宣言と定義に関する規格を指定しません。各種ｺﾝﾊﾟｲﾗはﾚｼﾞｽﾀを定義する各種の方法を持ち、其の内のいくつかは非標準言語構造を使用します。

AVR GCCはISRを定義するのにISRﾏｸﾛを使用します。このﾏｸﾛは<avr/interrupt.h>ﾍｯﾀﾞ ﾌｧｲﾙが必要です。AVR GCCでのISRは次の様に定義されます。

```
#include <avr/interrupt.h>
ISR(PCINT1_vect)
{
    // ｺｰﾄﾞ
}
```

IARでは、

```
#pragma vector=PCINT1_vect //C90
__interrupt void handler_PCINT1_vect()
{
    // ｺｰﾄﾞ
}
```

または

```
_Pragma("vector=PCINT1_vect") //C99
__interrupt void handler_PCINT1_vect()
{
    // ｺｰﾄﾞ
}
```

2つのｺﾝﾊﾟｲﾗ(AVR GCCとIAR)に共通してISRを定義する方式を作成する方法もあります。(以下の)これらの定義を持つﾍｯﾀﾞ ﾌｧｲﾙを作成してください。

```
#if defined(__GNUC__)
    #include <avr/interrupt.h>
#elif defined(__ICCAVR__)
    #define __ISR(x) _Pragma(#x)
    #define ISR(vect) __ISR(vector=vect) __interrupt void handler_##vect(void)
#endif
```

これはﾌﾟﾘｺﾝﾊﾟｲﾗによって読まれ、どちらのｺﾝﾊﾟｲﾗが使用されつつあるかに応じて正しいｺｰﾄﾞが使用されます。ISR定義はその後にIARとGCCで共通になり、以下の様に定義されます。

```
ISR(PCINT1_vect)
{
    // ｺｰﾄﾞ
}
```

5.2. 割り込み処理ルーチン内で更新される変数

ISRの内側で変更される変数はvolatile(揮発性)宣言されることが必要です。最適化機構使用時、以下のような繰り返しに於いて、

```
uint8_t flag;
...
ISR(SOME_vect) {
    flag = 1;
}
...
    while (flag == 0) {
        ...
    }
```

コンパイラはそれのコード経路分析がとにかく繰り返しの内側で"flag"の値を変更し得る物が何もないことを示すので、一般的に一度だけ"flag"をアクセスし、そして更なるアクセスを完全に別の場所に最適化します。この変数がそれのコード経路分析の可視範囲(例えば、割り込み処理ルーチン内)の外側で変更され得ることをコンパイラに知らせるために、変数は次の様な宣言が必要です。

```
volatile uint8_t flag;
```

変数が上のようにvolatile宣言されると、コンパイラは変数が更新されるまたは読まれる時に常にSRAMに変更を書き戻してSRAMから読むことを確かめます。

6. UARTボーレート計算

いくつかのAVRデータシートはボーレートを計算するために次式を与えます。

```
(F_CPU/(UART_BAUD_RATE*16UL)-1UL)
```

あいにく、除算演算子中の整数切り捨てのため、この式はクロック速度とボーレートの全ての組み合わせで動きません。

整数除算を行う時に通常は最低(切り捨て)よりもむしろ最近の整数への丸めがより良いことです。これを行うには除算をする前に分子に0.5(即ち分母の半分)を加えてください。そして使用されるべき式は次の通りです。

```
((F_CPU + UART_BAUD_RATE * 8UL) / (UART_BAUD_RATE * 16UL) - 1UL)
```

これは<util/setbaud.h>で実装される方法でもあります。

7. 電力管理と休止動作

SLEEP命令の使用は消費電力をかなり減らすことを応用に許します。AVRデバイスは各種休止動作に置くことができます。詳細についてはデバイスのデータシートを参照してください。

実際にデバイスを休止に置くためにこのヘッダ ファイルで提供される様々なマクロがあります。最も簡単な方法はset_sleep_mode()を用いて望む休止動作を設定してその後にsleep_mode()を呼ぶことです。このマクロは自動的に休止許可ビットを設定して休止へ行き、そして休止許可ビットを解除します。

例:
```
#include <avr/sleep.h>
...
    set_sleep_mode(<mode>);
    sleep_mode();
```

注: 目的が(ハードウェア リセットまで)CPUを完全に固定化することでない限り、休止へ行く前に割り込みが許可されることが必要です。

度々割り込み処理ルーチン(ISR)は検査されるソフトウェア フラグまたは変数を設定し、主繰り返し(main)で設定ならば処理されます。主繰り返しで休止命令が使用される場合、競合状態を起こす可能性があります。以下のコードでは発行されつつある休止命令とISRで設定されつつあるフラグの間で競合状態があります。

```
#include <avr/interrupt.h>
#include <avr/sleep.h>
...
volatile bool flag = false;
...
ISR(PCINT1_vect){
    flag = true;
}
int main(){
    ...
    while(1){
        if(flag){
           flag = false;
           ...
        }
        sleep_cpu();
        ...
    }
}
```

このコードでの問題はISRが事実上時間内のどの点でも起こり得ることの事実から来ます。if文が評価された直後で割り込みが起こる場合、デバイスはif文で必要とされる何かを行うことなく休止へ行くでしょう。これの実際の結果は応用依存です。このような競合状態を避けるための方法はcli()命令を用いて切り替えフラグを調べる前に全体割り込みを禁止することです。

例:

```
#include <avr/interrupt.h>
#include <avr/sleep.h>
...
volatile bool flag = false;
...
ISR(PCINT1_vect){
    flag = true;
}
int main(){
    ...
    sleep_enable();
    while(1){
        cli();
        if(flag){
            flag = false;
            ...
            sei();
            sleep_cpu();
        }
        sei();
        ...
    }
}
```

この手順は禁止されつつある割り込みと共にflagの非分断検査を保証します。条件が一致した場合、休止動作が準備され、SEI命令の直後にSLEEP命令が予定されます。SEI直後の命令は割り込みが起動され得る前に実行されることを保証されるため、デバイスが休止に置かれることを保証します。全体割り込みが禁止されている時に割り込みが保留中の場合、デバイスはその後ISRへ飛んでSEIと休止(SLEEP)の命令後の実行を続けます。プログラムの流れはif文に達して、休止で新しい割り込みを座して待たないでしょう。

いくつかのAVRデータシートが休止命令直前に休止を許可して起き上がった直後に休止を禁止することを推奨していることに注意してください。この推奨はプログラム作成者が間違ったポインタを作成してしまった場合に(意に)反する休止移行を防ぐためです。これはフラッシュ メモリの不適切な部分からコードの実行を開始する場合に、与えられたどの応用に対しても何が最良の動きになるか議論の余地があります。保護の最良の形はデバイスをリセットするのにウォッチドッグ タイマ(WDT)を使用するようなことです。

いくつかのﾃﾞﾊﾞｲｽは休止へ行く前に低電圧検出器(BOD:Brown Out Detector)を禁止する能力を持ちます。これは休止中の間にも電力を減らします。指定したAVRﾃﾞﾊﾞｲｽがこの能力を持つ場合、追加のsleep_bod_disable()ﾏｸﾛが定義されます。このﾏｸﾛは休止する前にBODを禁止するための時間制限手順を正しく実装するｲﾝﾗｲﾝ ｱｾﾝﾌﾞﾘ ｺｰﾄﾞを生成します。けれども、BODが禁止されてしまった後にﾃﾞﾊﾞｲｽを休止動作に置くことができる制限された周期数があり、さもなければBODは本当に禁止されません。推奨される慣習は以下のように、BODを禁止(sleep_bod_disable())し、割り込みを設定(sei())し、その後にﾃﾞﾊﾞｲｽを休止に置く(sleep_cpu())事です。

```
cli();
if (some_condition){
    sleep_bod_disable();
    sei();
    sleep_cpu();
}
sei();
```

7.1. 関数

7.1.1. void sleep_enable

```
void sleep_enable(void)
```

ﾃﾞﾊﾞｲｽの休止動作を可能にします。ﾃﾞﾊﾞｲｽがどの休止動作にされるかはset_sleep_mode()関数で選択された指定動作に依存します。より多くの詳細についてはﾃﾞﾊﾞｲｽに対するﾃﾞｰﾀｼｰﾄをご覧ください。

休止許可(SE)ﾋﾞｯﾄを設定(1)します。

7.1.2. void sleep_disable

```
void sleep_disable(void)
```

休止許可(SE)ﾋﾞｯﾄを解除(0)します。

7.1.3. void sleep_cpu

```
void sleep_cpu(void)
```

ﾃﾞﾊﾞｲｽを休止動作に置きます。その前にSEﾋﾞｯﾄが設定(1)されなければならず、後でこれを解除(0)することが推奨されます。

7.1.4. void sleep_mode

```
void sleep_mode(void)
```

ﾃﾞﾊﾞｲｽを休止動作に置きます。その前にSEﾋﾞｯﾄが設定(1)され、後でこれを解除(0)することに注意してください。

7.1.5. void sleep_bod_disable

```
void sleep_bod_disable(void)
```

休止へ行く前にBODを禁止します。全てのﾃﾞﾊﾞｲｽで利用可能な訳ではありません。

8. 遅延ルーチン

ここで記述され、delay.hﾍｯﾀﾞ ﾌｧｲﾙで見つかるような関数は<util/delay_basic.h>からの基本的な多忙待機関数を囲む被せ物(ﾗｯﾊﾟｰ)です。それらは待機のための周期数よりもむしろ実際の時間値を指定できる便利関数として意図されます。背景にある考え方はF_CPUﾏｸﾛによって渡されるCPU周波数に基づき、必要とされる遅延周期数を計算するのに浮動小数点式を使用するように、ｺﾝﾊﾟｲﾙ時の定数式がｺﾝﾊﾟｲﾗの最適化によって省略されることです。

意図されるようにこれらの関数が動くにためには、ｺﾝﾊﾟｲﾗの最適化が許可され、遅延時間がｺﾝﾊﾟｲﾙ時に既知の定数である式でなければなりません。これら必要条件が合わなければ、結果の遅延はもっとより長く(そして基本的に予測できなく)なり、その他として浮動小数点計算を使わない応用は応用にﾘﾝｸされる浮動小数点ﾗｲﾌﾞﾗﾘ ﾙｰﾁﾝによって厳しいｺｰﾄﾞ肥大を経験するでしょう。

8.1. F_CPU

F_CPUﾏｸﾛは遅延ﾏｸﾛによって考慮されるCPU周波数を指定します。このﾏｸﾛは通常、環境(例えば、ﾌﾟﾛｼﾞｪｸﾄ ﾍｯﾀﾞで、またはﾌﾟﾛｼﾞｪｸﾄのmakefileから)によって供給されます。使用者が提供するこのような定義を見つけることができない場合に”普通の”代替として<util/delay.h>での1MHzの値が提供されるだけです。

遅延関数の条件に於いて、CPU周波数は浮動小数点定数(例えば、3.6864MHzに対しては3.6864E6)として与えることができます。けれども、<util/setbaud.h>内のﾏｸﾛは整数値であることが必要です。

8.2. void_delay_ms

```
void _delay_ms(double __ms)
```

最高分解能での可能な最大遅延は'262.14ms/MHzでのF_CPU'です。使用者が最大可能値を超える遅延を要求すると、_delay_m s ()は機能的に減らされた分解能を提供します。この動作での_delay_ms()は1/10msの分解能で動き、(CPU周波数と無関係に)6.5535秒までを提供します。使用者は減らされた分解能について通知されません。

avr-gccツールチェーンが_builtin_avr_delay_cycles()支援を持つ場合、可能な最大遅延は'4294967.295ms/MHzでのF_CPU'です。可能な最大遅延よりも大きな値については、溢れで遅延なし、即ち0msに終わります。

_msのクロック周期(数)への変換は常に整数に帰着しないかもしれません。既定により、クロック周期は次の整数へ丸め上げられます。これは使用者が少なくても_msの遅延を得ることを保証します。

代わりに、このヘッダ ファイルをインクルードする前に、_DELAY_ROUND_DOWN_または_DELAY_ROUND_CLOSEST_のマクロを定義することによって、算法は各々、切り捨て丸め、または最も近い整数への丸めにすることができます。

注: _builtin_avr_delay_cycles()に基づく_delay_ms()の実装は旧実装との後方互換がありません。前の版との機能的な後方互換を得るためには、このヘッダ ファイルをインクルードする前に"_DELAY_BACKWARD_COMPATIBLE_"が定義されなければなりません。また、丸めを必要とする数学機能がコンパイラに対して利用できないため、自立環境(GCCの-ffreestanding任意選択)でコンパイルされる場合にも後方互換算法が選択されます。

8.3. void_delay_us

```
void _delay_us(double __us)
```

F_CPUマクロは(Hzでの)CPUクロック周波数を定義する定数を定義することになっています。

最大可能遅延は'768μs/MHzでのF_CPU'です。

使用者が最大可能値よりも大きな遅延を要求した場合、_delay_us()は代わりに自動的に_delay_ms()を呼びます。使用者はこの場合について通知されません。

avr-gccツールチェーンが_builtin_avr_delay_cycles()支援を持つ場合、可能な最大遅延は'4294967.295μs/MHzでのF_CPU'です。可能な最大遅延よりも大きな値については、溢れで遅延なし、即ち0μsに終わります。

_usのクロック周期(数)への変換は常に整数に帰着しないかもしれません。既定により、クロック周期は次の整数へ丸め上げられます。これは使用者が少なくても_usの遅延を得ることを保証します。

代わりに、このヘッダ ファイルをインクルードする前に、_DELAY_ROUND_DOWN_または_DELAY_ROUND_CLOSEST_のマクロを定義することによって、算法は各々、切り捨て丸め、または最も近い整数への丸めにすることができます。

注: _builtin_avr_delay_cycles()に基づく_delay_us()の実装は旧実装との後方互換がありません。前の版との機能的な後方互換を得るためには、このヘッダ ファイルをインクルードする前に"_DELAY_BACKWARD_COMPATIBLE_"が定義されなければなりません。また、丸めを必要とする数学機能がコンパイラに対して利用できないため、自立環境(GCCの-ffreestanding任意選択)でコンパイルされる場合にも後方互換算法が選択されます。

9. コード量を減らすための助言と策略

例のコードと本章での試験結果は以下の条件に基づきます。

1. AVR GCC 8ビット ツールチェーン版 : AVR_8_bit_GNU_Toolchain_3.2.1_292 (GCC version 4.5.1)
2. 目的対象デバイス : ATmega88PA

9.1. コード量を減らすための助言と策略

この項ではコードの大きさを減らす方法についていくつかの助言を一覧にします。各助言に対して説明と見本コードが与えられます。

9.1.1. 助言1 データ型と大きさ

可能な最小の適切なデータ型を使用してください。コードと特にデータ型を評価してください。レジスタからの8ビット(1バイト)値読み込みは2バイト変数ではなく、単一バイト変数が必要なだけで、従ってコード空間とデータ空間を節約します。

AVR 8ビット マイクロ コントローラでのデータ型の大きさは<stdint.h>ヘッダ ファイルで見つけることができ、下表で要約されます。

表9-1. <stdint.h>でのAVR 8ビット マイクロ コントローラのデータ型

データ型		大きさ
signed char / usigned char	int8_t / uint8_t	8ビット
signed int / usigned int	int16_t / uint16_t	16ビット
signed long / usigned long	int32_t / uint32_t	32ビット
signed long long / usigned long long	int64_t / uint64_t	64ビット

コンパイラの或る-スイッチがこれを変えることができることに注意してください(avr-gcc -mint8は整数データ型を8ビット整数に変えます)。下表は異なるデータ型と大きさの効果を示します。avr_sizeユーティリティからの出力は応用が-Os(大きさ最適化)で構築される時に使用されるコード空間を示します。

表9-2. 異なるデータ型と大きさの例

項目	符号なし整数 unsigned int (16ビット)	符号なし文字 unsigned char (8ビット)

Cソース コード	include <avr/io.h>
	unsigned int readADC() {
	return ADCH;
	};
	int main(void)
	{
	unsigned int mAdc = readADC();
	}
		#include <avr/io.h>
		unsigned char readADC() {
		return ADCH;
		};
		int main(void)
		{
		unsigned char mAdc = readADC();
		}

AVRメモリ使用量	プログラム : 92バイト (1.1%使用)	プログラム : 90バイト (1.1%使用)
コンパイラ最適化レベル	-Os (大きさ最適化)	-Os (大きさ最適化)

左の例ではreadADC()関数からの戻り値としてと、readADC()関数からの戻り値を格納するのに使用される一時変数でint(2バイト)データ型を使用します。

右の例では代わりにcahr(1バイト)を使用しています。ADCHレジスタからの読み出しは8ビットだけで、これはcharで充分なことを意味します。readADC()関数の戻り値とmainでの一時変数がint(2バイト)からcahr(1バイト)に変更されたために2バイトが節約されます。

大きさでの違いは変数がこの例で行われるのよりも多く操作される場合に増加します。一般的に16ビット変数の算術と論理の両操作は8ビット変数よりももっと多くの周期数と空間を占領します。

注: main()から走行する前に始動コードがあります。それは単純Cコードが約90バイトを占領する理由です。

9.1.2. 助言2 全域変数と局所変数

殆どの場合で、全域変数の使用は推奨されません。。可能な時は必ず局所変数を使用してください。変数が関数でだけ使用されるなら、局所変数として関数の内側で宣言されるべきです。

理論的に、全域かまたは局所の変数としてどちらの変数を宣言するかを選ぶのはそれがどう使われるかによって決めされるべきです。

全域変数が宣言された場合、プログラム リンク時にこの変数にSRAM内の固有のアドレスが割り当てられます。全域変数のアクセスはそれのアドレスを得るために代表的に追加のバイト(通常、16ビット アドレスに対して2バイト)が必要です。

局所変数はそれらが宣言された時に支援されていれば、できればレジスタに割り当てられるか、またはスタックに割り当てられます。関数が活性になると、関数の局所変数もまた活性になります。一旦関数を抜け出すと、その関数の局所変数は取り去ることができます。

下表は全域と局所の変数の効果を示します。

表9-3. 全域変数と局所変数の例

項目	全域変数	局所変数

Cソース コード	#include <avr/io.h>
	uint8_t global_1;
	int main(void)
	{
	global_1 = 0xAA;
	PORTB = global_1;
	}
		#include <avr/io.h>
		int main(void)
		{
		uint8_t local_1;
		local_1 = 0xAA;
		PORTB = local_1;
		}

AVRメモリ使用量	プログラム : 104バイト (1.3%使用) (.text + .dat + .bootloader) データ : 1バイト (0.1%使用) (.dat + .bss + .noinit)	プログラム : 84バイト (1.0%使用) (.text + .dat + .bootloader) データ : 0バイト (0.0%使用) (.dat + .bss + .noinit)
コンパイラ最適化レベル	-Os (大きさ最適化)	-Os (大きさ最適化)

左の例ではバイトの大きさの全域変数が宣言されています。avr_sizeユーティリティからの出力は最適化レベル-Os(大きさ最適化)でコード空間の104バイトとデータ空間の1バイトを使用することを示します。

右の例では局所関数としてmain()関数の内側で変数を宣言した後で、コード空間が84バイトに減らされてSRAMは全く使用されません。

9.1.3. 助言3 繰り返し指標

繰り返し(ループ)は8ビットAVRコードで広く使用されます。”while ( ) { }”繰り返し、”for ( )”繰り返し、”do { } while ( )”繰り返しがあります。-Os最適化が許可されるなら、コンパイラは繰り返しを同じコードの大きさを持つように自動的に最適化します。

けれども、未だコードの大きさを僅かに減らすことができます。”do { } while ( )”繰り返しを使用する場合、増加進行と減少後退の繰り返し指標は違うコード量を生成します。通常、我々は繰り返し計数を0から最大値へ書きます(増加進行)が、繰り返しを最大値から0へ計数する(減少後退)ことがもっと効率的です。

それは増加進行繰り返しのでは繰り返し指標が最大値に達したかを調べるために毎回の繰り返しで繰り返し指標と最大値を比較するのに比較命令が必要とされるためです。

減少後退繰り返しを使用すると、減少された繰り返し指標の結果が0に達した場合にSREGでZ(ゼロ)フラグが設定(1)されるので、この比較命令がもはや必要とされません。

下表は増加進行と減少後退の繰り返し指標を持つ”do { } while ( )”繰り返しの効果を示します。

表9-4. 増加進行と減少後退の繰り返し指標を持つdo { } while ( )繰り返しの例

項目	増加進行繰り返し指標を持つdo { } while ( )	減少後退繰り返し指標を持つdo { } while ( )

Cソース コード	`#include <avr/io.h>`
	`int main(void)`
	`{`
	`  uint8_t local_1 = 0;`
	`  do {`
	`  PORTB ^= 0x01;`
	`  local_1++;`
	`  } while (local_1<100);`
	`}`
		`#include <avr/io.h>`
		`int main(void)`
		`{`
		`  uint8_t local_1 = 100;`
		`  do {`
		`  PORTB ^= 0x01;`
		`  local_1--; (注)`
		`  } while (local_1);`
		`}`

AVRメモリ使用量	プログラム : 96バイト (1.2%使用) (.text + .dat + .bootloader) データ : 0バイト (0.0%使用) (.dat + .bss + .noinit)	プログラム : 94バイト (1.1%使用) (.text + .dat + .bootloader) データ : 0バイト (0.0%使用) (.dat + .bss + .noinit)
コンパイラ最適化レベル	-Os (大きさ最適化)	-Os (大きさ最適化)

注: Cコード行での明瞭な比較をするため、この例では通常C本で使用される”do { } while (--count);”のようではなく、”do {count-- ;} while (count);”のように書かれます。2つの形式は同じコードを生成します。

9.1.4. 助言4 繰り返しの詰め込み

ここでの繰り返し(ループ)妨害は文と操作を各種繰り返しからより少ない繰り返しまたは1つの繰り返しに統合することを参照し、従ってコードでの繰り返し数を減らします。

いくつかの場合で、様々な繰り返しが1つ毎に実装されます。これは繰り返しの長い一覧の結果に繋がるかもしれません。この場合、繰り返しの詰め込みは実際に1つに組み合わされた繰り返しを持つことによってコード効率を増すことを手助けするかもしれません。

繰り返しの詰め込みはコード量を減らし、繰り返しの周回での間接負荷を取り去ることによって更にコードをより速く走らせます。次表は繰り返しの詰め込みの効果を示します。

表9-5. 繰り返しの詰め込みの例

項目	分離した繰り返し	繰り返しの詰め込み

Cソース コード	#include <avr/io.h>
	int main(void)
	{
	uint8_t i, total = 0;
	uint8_t tmp[10] = {0};
	for (i=0; i<10; i++) {
	tmp [i] = ADCH;
	}
	for (i=0; i<10; i++) {
	total += tmp[i];
	}
	UDR0 = total;
	}
		#include <avr/io.h>
		int main(void)
		{
		uint8_t i, total = 0;
		uint8_t tmp[10] = {0};
		for (i=0; i<10; i++){
		tmp [i] = ADCH;
		
		total += tmp[i];
		}
		UDR0 = total;
		}

AVRメモリ使用量	プログラム : 164バイト (2.0%使用) (.text + .dat + .bootloader) データ : 0バイト (0.0%使用) (.dat + .bss + .noinit)	プログラム : 98バイト (1.2%使用) (.text + .dat + .bootloader) データ : 0バイト (0.0%使用) (.dat + .bss + .noinit)
コンパイラ最適化レベル	-Os (大きさ最適化)	-Os (大きさ最適化)

9.1.5. 助言5 プログラム空間での定数

多くの応用はフラッシュ メモリを使い果たす前にデータを格納するSRAMを使い果たします。決して変更されない一定の全域変数、表、配列は通常、読み込み専用領域(8ビットAVRでのフラッシュ メモリやEEPROM)に割り当てられるべきです。この方法によって貴重なSRAM空間を節約することができます。

この例ではCキーワードの"const"を使いません。"const"オブジェクトの宣言はそれの値が変更されないことを告知します。"const"はコンパイラにデータが"読み出し専用"であり、最適化の機会を増すことを告げるのに使用されます。それはデータが格納されるべき場所を識別しません。

プログラム空間(読み込み専用)内にデータを割り当ててプログラム空間からそれらを取得するために、AVR-Libcは簡単な"PROGMEM"マクロと"pgm_read_byte"マクロを提供します。"PROGMEM"マクロと"pgm_read_byte"マクロは<avr/pgmspace.h>システム ヘッダ ファイルで定義されます。

下表は全域文字列をプログラム空間へ移動することによってSRAMを節約する方法を示します。

表9-6. プログラム空間での定数の例

項目	データ空間での定数	プログラム空間での定数

Cソース コード	#include <avr/io.h>
	uint8_t string[12]
	= {"hello world!"};
	int main(void)
	{
	UDR0 = string[10];
	}
		#include <avr/io.h>
		#include <avr/pgmspace.h>
		uint8_t string[12] PROGMEM
		= {"hello world!"};
		int main(void)
		{
		UDR0 = pgm_read_byte(&string[10]);
		}

AVRメモリ使用量	プログラム : 122バイト (1.5%使用) (.text + .dat + .bootloader) データ : 12バイト (1.2%使用) (.dat + .bss + .noinit)	プログラム : 102バイト (1.2%使用) (.text + .dat + .bootloader) データ : 0バイト (0.0%使用) (.dat + .bss + .noinit)
コンパイラ最適化レベル	-Os (大きさ最適化)	-Os (大きさ最適化)

定数をプログラム空間内に割り当て後、プログラム空間とデータ空間の両方が減らされたことが見れます。けれども、関数の実行はSRAMから直接データを読むよりも遅いため、データを読み戻す時に僅かな付随負荷があります。

フラッシュ メモリに格納されたデータがコードに於いて複数回使用される場合、数回直接"pgm_read_byte"マクロを使用する代わりに一時変数を使用することによって量が減らされます。

プログラム空間で各種データ型を格納と取得を行うために<avr/pgmspace.h>システム ヘッダ ファイルにより多くのマクロと関数があります。より多くの詳細についてはAVR-Libc使用者手引書を参照してください。

9.1.6. 助言6 アクセス型:Static

全域データに関して、可能な時は必ず"static"キーワードを使用してください。"static"キーワードを持つ全域変数が検出された場合、それらはそれらが定義されたファイルでだけアクセスすることができます。これは他のファイル内のコードによって(外部変数としての)変数の無計画な使用を防ぎます。

一方、関数内の局所変数は殆どの場合で"static"を宣言されるべきではありません。静的(static)局所変数の値は関数を呼んで変数がプログラム全体を通して持続する間に保存されることが必要です。従って定常的なデータ空間(SRAM)記憶域とそれをアクセスするための余分なコードが必要です。それは可視範囲がそれが定義された関数内であることを除いて全域変数と同じです。

静的関数はそれが宣言されたファイルの外側でその名前が不可視で、それが他のどのファイルからも呼ばれないので、コンパイラに関して最適化がより容易です。

静的関数が最適化(-O1,-O2,-P3,-Os)許可でファイル内で一度だけ呼ばれた場合、その関数はインライン関数としてコンパイラによって自動的に最適化され、飛び込みと抜け出しのためのアセンブリ コードが全く作成されません。下表は静的関数の効果を示します。

表9-7. 型:静的(static)関数アクセスの例

項目	(一度だけ呼ばれる)全域関数	(一度だけ呼ばれる)静的関数

Cソース コード	`#include <avr/io.h>`
	`uint8_t string[12] = {"hello world!"};`
	`void USART_TX(uint8_t data);`
	`int main(void)`
	`{`
	`  uint8_t i = 0;`
	`  while (i<12) {`
	`  USART_TX(string[i++]);`
	`  }`
	`}`
	
	`void USART_TX(uint8_t data)`
	`{`
	`  while(!(UCSR0A&(1<<UDRE0)));`
	`  UDR0 = data;`
	`}`
		`#include <avr/io.h>`
		`uint8_t string[12] = {"hello world!"};`
		`static void USART_TX(uint8_t data);`
		`int main(void)`
		`{`
		`  uint8_t i = 0;`
		`  while (i<12) {`
		`  USART_TX(string[i++]);`
		`  }`
		`}`
		
		`void USART_TX(uint8_t data)`
		`{`
		`  while(!(UCSR0A&(1<<UDRE0)));`
		`  UDR0 = data;`
		`}`

AVRメモリ使用量	プログラム : 152バイト (1.9%使用) (.text + .dat + .bootloader) データ : 12バイト (1.2%使用) (.dat + .bss + .noinit)	プログラム : 140バイト (1.7%使用) (.text + .dat + .bootloader) データ : 12バイト (1.2%使用) (.dat + .bss + .noinit)
コンパイラ最適化レベル	-Os (大きさ最適化)	-Os (大きさ最適化)

注: 関数が複数回呼ばれる場合、これが直接関数呼び出しよりもより多くのコードを生成するため、インライン関数へ最適化されません。

9.1.7. 助言7 低位アセンブリ命令

上手くコード化されたアセンブリ命令は常に最良の最適化されたコードです。アセンブリ コードの1つの欠点は可搬性のない構文規則で、故に殆どの場合でプログラムの書き手に対して推奨されません。

けれども、アセンブリ マクロの使用はアセンブリ コードと関連した痛みを度々減らし、可読性を改善します。2,3行のアセンブリ コードよりも少なく生成する作業用の関数の代わりにマクロを使用してください。次表は関数使用と比較したアセンブリ マクロ使用コードの例を示します。

表9-8. 低位ｱｾﾝﾌﾞﾘ命令の例

項目	関数	ｱｾﾝﾌﾞﾘ ﾏｸﾛ

Cｿｰｽ ｺｰﾄﾞ	#include <avr/io.h>
	void enable_usart_rx(void)
	{
	UCSR0B |= 0x80;
	};
	int main(void)
	{
	enable_usart_rx();
	while (1){
	}
	}
		#include <avr/io.h>
		#define enable_usart_rx() ¥
		__asm__ __volatile__ (¥
		"lds r24,0x00C1" "¥n¥t" ¥
		"ori r24, 0x80" "¥n¥t" ¥
		"sts 0x00C1, r24" ¥
		::)
		int main(void)
		{
		enable_usart_rx();
		while (1){
		}
		}

AVRﾒﾓﾘ使用量	ﾌﾟﾛｸﾞﾗﾑ : 90ﾊﾞｲﾄ (1.1%使用) (.text + .dat + .bootloader) ﾃﾞｰﾀ : 0ﾊﾞｲﾄ (0.0%使用) (.dat + .bss + .noinit)	ﾌﾟﾛｸﾞﾗﾑ : 86ﾊﾞｲﾄ (1.0%使用) (.text + .dat + .bootloader) ﾃﾞｰﾀ : 0ﾊﾞｲﾄ (0.0%使用) (.dat + .bss + .noinit)
ｺﾝﾊﾟｲﾗ最適化ﾚﾍﾞﾙ	-Os (大きさ最適化)	-Os (大きさ最適化)

より多くの詳細についてはAVR-Libc使用者手引書を参照してください。

9.2. 実行時間を減らすための助言と策略

この項では実行時間を減らす方法についていくつかの助言を一覧にします。各助言に対して短い説明と見本ｺｰﾄﾞが与えられます。

9.2.1. 助言8 ﾃﾞｰﾀ型と大きさ

ｺｰﾄﾞの大きさの低減に加え、正しいﾃﾞｰﾀ型と大きさの選択はまた実行時間も減らします。AVR 8ﾋﾞｯﾄに関して、8ﾋﾞｯﾄ(1ﾊﾞｲﾄ)値ｱｸｾｽは常に最も効果的な方法です。

下表は8ﾋﾞｯﾄと16ﾋﾞｯﾄの変数間の違いを示します。

表9-9. ﾃﾞｰﾀ型と大きさの例

項目	16ﾋﾞｯﾄ変数	8ﾋﾞｯﾄ変数

Cｿｰｽ ｺｰﾄﾞ	#include <avr/io.h>
	int main(void)
	{
	uint16_t local_1 = 10;
	do {
	PORTB ^= 0x80;
	} while (--local_1);
	}
		#include <avr/io.h>
		int main(void)
		{
		uint8_t local_1 = 10;
		do {
		PORTB ^= 0x80;
		} while (--local_1);
		}

AVRﾒﾓﾘ使用量	ﾌﾟﾛｸﾞﾗﾑ : 94ﾊﾞｲﾄ (1.1%使用) (.text + .dat + .bootloader) ﾃﾞｰﾀ : 0ﾊﾞｲﾄ (0.0%使用) (.dat + .bss + .noinit)	ﾌﾟﾛｸﾞﾗﾑ : 92ﾊﾞｲﾄ (1.1%使用) (.text + .dat + .bootloader) ﾃﾞｰﾀ : 0ﾊﾞｲﾄ (0.0%使用) (.dat + .bss + .noinit)
実行周期数	90	79
ｺﾝﾊﾟｲﾗ最適化ﾚﾍﾞﾙ	-O2	-O2

注: 繰り返しは-O3任意選択で自動的にｺﾝﾊﾟｲﾗによって展開されます。そして繰り返しは繰り返し指標によって示される繰り返し動作に展開され、故にこの例については-O3任意選択許可で違いは全くありません。

9.2.2. 助言9 条件文

通常、標準的なｺｰﾄﾞ行での事前減少と事後減少(または事前増加と事後増加)は全く違いを生じません。例えば、"i--"と"--i"は単純に同じｺｰﾄﾞを生成します。けれども、これらの演算子を繰り返し指標として条件文での使用は違う生成ｺｰﾄﾞになります。

助言3で述べられるように、減少する繰り返し指標の使用はより小さなｺｰﾄﾞ量に帰着します。これは条件文でより速い実行を得るのにも役立ちます。

更に、事前減少と事後減少も異なる結果を持ちます。次表から、事前減少条件文でより速いｺｰﾄﾞが生成されることを見ることができます。ここでの周期計数値は最長繰り返しの実行時間を表します。

表9-10. 条件文の例

項目	条件文での事後減少	条件文での事前減少

Cソース コード	#include <avr/io.h>
	int main(void)
	{
	uint8_t loop_cnt = 9;
	do {
	if (loop_cnt--) {
	PORTC ^= 0x01;
	} else {
	PORTB ^= 0x01;
	loop_cnt = 9;
	}
	} while (1);
	}
		#include <avr/io.h>
		int main(void)
		{
		uint8_t loop_cnt = 10;
		do {
		if (--loop_cnt) {
		PORTC ^= 0x01;
		} else {
		PORTB ^= 0x01;
		loop_cnt = 10;
		}
		} while (1);
		}

AVRメモリ使用量	プログラム : 104バイト (1.3%使用) (.text + .dat + .bootloader) データ : 0バイト (0.0%使用) (.dat + .bss + .noinit)	プログラム : 102バイト (1.2%使用) (.text + .dat + .bootloader) データ : 0バイト (0.0%使用) (.dat + .bss + .noinit)
実行周期数	75	61
コンパイラ最適化レベル	-O3	-O3

この例が同じように動くことを確実にするため、"loop_cnt"は2つの例で違う値が割り当てられます。全周でポートCのPC0は9回切り替えられ、一方ポートBのPB0は1回切り替えられます。

9.2.3. 助言10 繰り返しを展開

いくつかの場合で、コード実行を速度向上するために繰り返しを展開することができます。これは特に短い繰り返しに対して効果的です。繰り返しが展開された後、試験すべき繰り返し指標は全くなく、繰り返しに於いて各一回りでより少ない分岐が実行されます。

下表は繰り返しの展開の効果を示します。

表9-11. 繰り返しの展開の例

項目	繰り返し	繰り返しを展開

Cソース コード	#include <avr/io.h>
	int main(void)
	{
	uint8_t loop_cnt = 10;
	do {
	PORTB ^= 0x01;
	} while (--loop_cnt);
	}
		#include <avr/io.h>
		int main(void)
		{
		PORTB ^= 0x01;
		PORTB ^= 0x01;
		PORTB ^= 0x01;
		PORTB ^= 0x01;
		PORTB ^= 0x01;
		PORTB ^= 0x01;
		PORTB ^= 0x01;
		PORTB ^= 0x01;
		PORTB ^= 0x01;
		PORTB ^= 0x01;
		}

AVRメモリ使用量	プログラム : 94バイト (1.5%使用) (.text + .dat + .bootloader) データ : 0バイト (0.0%使用) (.dat + .bss + .noinit)	プログラム : 142バイト (1.7%使用) (.text + .dat + .bootloader) データ : 0バイト (0.0%使用) (.dat + .bss + .noinit)
実行周期数	80	50
コンパイラ最適化レベル	-O2	-O2

"do { } while ( )"繰り返しの展開によって、80クロック周期から50クロック周期へコード実行をかなり速度向上します。

繰り返しの展開後にコード量が94バイトから142バイトに増加されることに注意してください。これは速度と大きさの最適化間の二律背反を示す例でもあります。

注: この例で-O3任意選択が許可された場合、コンパイラは繰り返しを自動的に展開し、手動での繰り返しの展開と同じコードを生成します。

9.2.4. 助言11 制御の流れ:If-ElseとSwitch-Case

”if-else”と”switch-case”はCｺｰﾄﾞで広く使われ、分岐の正しい構成は実行時間を減らすことができます。

”if-else”については、最初の場所に最も起こりそうな条件を置いてください。そして後続する条件はあまり実行されそうにありません。従って殆どの場合で時間が節約されます。

”switch-case”については通常ｺﾝﾊﾟｲﾗが指標と共に参照表を作成して正しい場所へ直接飛ぶため、”switch-case”の使用は”if-else”の欠点を無くすかもしれません。

”switch-case”を使うことが難しい場合、”if-else”分岐をより小さな補助分岐に分割することができます。この方法は最悪の場合の条件に対する実行時間を減らします。下表ではA/D変換器(ADC)からﾃﾞｰﾀを得てその後にUSARTを通してﾃﾞｰﾀを送ります。”ad_result <= 240”が最悪の場合です。

表9-12. if-else補助分岐の例

項目	if-else分岐	if-else補助分岐

Cソース ｺｰﾄﾞ	#include <avr/io.h>
	uint8_t ad_result;
	uint8_t readADC() {
	return ADCH;
	};
	
	void send(uint8_t data){
	UDR0 = data;
	};
	
	int main(void)
	uint8_t output;
	ad_result = readADC();
	if(ad_result <= 30){
	output = 0x6C;
	}else if(ad_result <= 60){
	output = 0x6E;
	}else if(ad_result <= 90){
	output = 0x68;
	}else if(ad_result <= 120){
	output = 0x4C;
	}else if(ad_result <= 150){
	output = 0x4E;
	}else if(ad_result <= 180){
	output = 0x48;
	}else if(ad_result <= 210){
	output = 0x57;
	}else if(ad_result <= 240){
	output = 0x45;
	}
		int main(void)
		{
		uint8_t output;
		ad_result = readADC();
		if (ad_result <= 120){
		if (ad_result <= 60){
		if (ad_result <= 30){
		output = 0x6C;
		}else{
		output = 0x6E;
		}
		}else{
		if (ad_result <= 90){
		output = 0x68;
		}else{
		output = 0x4C;
		}
		}
		}else{
		if (ad_result <= 180){
		if (ad_result <= 150){
		output = 0x4E;
		}
		}

AVRﾒﾓﾘ使用量	ﾌﾟﾛｸﾞﾗﾑ : ???ﾊﾞｲﾄ (?.?%使用) (.text + .dat + .bootloader) ﾃﾞｰﾀ : ?ﾊﾞｲﾄ (?.?%使用) (.dat + .bss + .noinit)	ﾌﾟﾛｸﾞﾗﾑ : ???ﾊﾞｲﾄ (?.?%使用) (.text + .dat + .bootloader) ﾃﾞｰﾀ : ?ﾊﾞｲﾄ (?.?%使用) (.dat + .bss + .noinit)
実行周期数	58 (最悪の場合)	48 (最悪の場合)
ｺﾝﾊﾟｲﾗ最適化ﾚﾍﾞﾙ	-O3	-O3

最悪の場合で分岐に至るのに少ない時間が必要なことを見ることができます。ｺｰﾄﾞ量が増やされることにも注意することができます。従って大きさまたは速度での指定必要条件に従った結果に釣り合わせるべきです。

9.3. 結び

この章では大きさと速度でのCｺｰﾄﾞ効率性についていくつかの助言と策略を一覧にします。現代のCｺﾝﾊﾟｲﾗに有難う、これらは異なる場合で各種最適化任意選択を自動的に実施することに於いて賢明です。けれども、ｺﾝﾊﾟｲﾗは開発者よりもより良いｺｰﾄﾞを分からず、故に良いｺｰﾄﾞ書きが常に重要です。

例で示されるように、或る一面の最適化は他での影響を持つかもしれません。特定の求めに基づいてｺｰﾄﾞの大きさと速度の間の調和が必要です。

Cｺｰﾄﾞ最適化のためにこれらの助言と策略を持つとは言え、それらのより良い使い方のために、取り組んでいるﾃﾞﾊﾞｲｽとｺﾝﾊﾟｲﾗの良好な理解がかなり必要です。そして異なる応用の場合でｺｰﾄﾞ効率を最適化するには他の技能と方法も確実にあります。

10. 参照

- GNUｺﾝﾊﾟｲﾗ集合手引書 (https://gcc.gnu.org/onlinedocs/gcc/)
- AVR-Libc使用者の手引き (http://www.nongnu.org/avr-libc/user-manual/)
- Atmel Studio (http://www.atmel.com/tools/atmelstudio.aspx?tab=overview)
- Atmel START (http://start.atmel.com)
- IAR AVR用C/C++ｺﾝﾊﾟｲﾗ使用者の手引き (http://ftp.iar.se/WWWfiles/AVR/webic/doc/EWAVR_CompilerReference.pdf)

11. 改訂履歴

資料改訂	日付	注釈
42787A	2016年10月	初版資料公開

Atmel | Enabling Unlimited Possibilities®

Atmel Corporation 1600 Technology Drive, San Jose, CA 95110 USA TEL:(+1)(408) 441-0311 FAX: (+1)(408) 436-4200 | www.atmel.com

本応用記述はAtmelのAVR42787応用記述(Rev.42787A-10/2016)の翻訳日本語版です。日本語では不自然となる重複する形容表現は省略されている場合があります。日本語では難解となる表現は大幅に意訳されている部分もあります。必要に応じて一部加筆されています。頁割の変更により、原本より頁数が少なくなっています。

必要と思われる部分には( )内に英語表記や略称などを残す形で表記しています。

青字の部分はﾘﾝｸとなっています。一般的に赤字の0,1は論理0,1を表します。その他の赤字は重要な部分を表します。