AVR4027 :8ビットAVRマイクロ コントローラ用 Cコード最適化のための助言と秘訣

要点

- ATMEL® AVR® コアとATMEL AVR GCC紹介
- コード量を減らすための助言と秘訣
- 実行時間を減らすための助言と秘訣
- 応用例

1.序説

AVRコアはCコード用に調整されて進化したRISC構造です。それは低費用でのより大きな機能を持つ良好な製品の開発を確実にします。

最適化について語る時に、通常、コード量とコード速度の2つの方向を参照します。今日、Cコンパイラは量と速度のどちらかでの効率的なコードを得るために開発者を手助けする各種最適化任意選択を持ちます。

けれども、良いCコードの書き方は望まれるようにコードを最適化するためのコンパイラに対してより多くの機会を与えます。そしていくつかの場合には、2つの方向の1つの最適化は他方に影響を及ぼす、または退化さえさせ、故に開発者はそれらの特定の要求に従って2つを平衡しなければなりません。8ビットAVRに関するCの書き方についてのいくつかの助言と秘訣の理解は、コード効率改善に於いて注目する場所を知るための開発者を助けます。

この応用記述で、助言はAVR-GCC (Cコンパイラ)に基づきます。けれども、それらの助言は他のコンパイラまたは同様のコンパイラ任意選択で実行することが可能で、またその逆もです。

2.ATMEL AVRコアとATMEL AVR GCCの知識

組み込み系ソフトウェアの最適化前に、AVRコアがどう構築され、このプロセッサに対してAVR GCCが効率的なコードを生成するために何の戦略を使用するかの充分な理解を持つことが必要です。ここではAVRコアとAVR GCCの特徴の短い紹介があります。

2.1.ATMEL AVR 8ビット構造

AVRはプログラムとデータに対して独立したメモリとバスを持つハーバード構造を使用します。それは単一クロック周期アクセス時間を持つ32× 8ビットの汎用作業レジスタの高速アクセスレジスタファイルを持ちます。32個の作業レジスタは効率的なCの書き方の鍵の1つです。これらのレジスタはそれらが32個あることを除き、従来の累積器と同じ機能を持ちます。AVRの算術と論理の命令はこれらのレジスタで動き、故にそれらはより少ない命令空間を取ります。1クロック周期で、AVRはレジスタから2つの任意のレジスタをALUへ供給して、操作を実行し、そして結果をレジスタファイルに書き戻します。

プログラムメモリの命令は単一段のパイプラインで実行されます。1つの命令が実行されつつあるのと同時に、次の命令がプログラムメモリから予め取得されます。この概念は毎クロック周期で実行されるべき命令を許します。殆どのAVR命令は単一16ビット語形式を持ちます。全てのプログラムメモリアドレスは16または32ビットの命令を含みます。

より多くの詳細についてはデバイスのデータシートで「AVR CPUコア」章を参照してください。

2.2.AVR GCC

GCCはGNUコンパイラ集合 (GNU Compiler Collection)を表します。GCCがAVR目的対象に使用されると、それは一般的にAVR GCCとして知られます。実際のプログラム"gcc"は "avr-"が前置、換言すると、"avr-gcc"になります。

AVR GCCは多数の最適化段階を提供します。それらは -O0,-O1,-O2,-O3, そして -Osです。各段階に於いて、最適化なしを意味する -O0を除いて、許された各種最適化任意選択があります。最適化段階で許された任意選択の他にも、特定の最適化を得るために独立した最適化任意選択を許可することができます。

最適化の任意選択と段階の完全な一覧については、下の通りにGNUコンパイラ集合手引書を参照してください。

http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html#Optimize-Options

8ビットATMEL
マイクロ コントローラ

応用記述

本書は一般の方々の便宜のため有志により作成されたもので、ATMEL社とは無関係であることを御承知ください。しおりのはじめにでの内容にご注意ください。

Rev.8453A-11/11,8453AJ0-12/11

"avr-gcc"の他に、AVRマイクロ コントローラ用の最終実行可能応用を生成するために共に働く他の多くのツールを利用します。このツール群はツールチェーンと呼ばれます。このAVRツールチェーンでは、avr-libcが通常の標準Cライブラリで見つかる同じ関数の多くとAVR特有の多くの追加ライブラリ関数を提供する重要なCライブラリとして扱います。

AVR Libc一括はATMEL AVR 8ビットRISCマイクロ コントローラ用の標準Cライブラリの下位集合を提供します。加えて、ライブラリは殆どの応用で必要とされる基本的な始動コードを提供します。

avr-libcの手引きについては下のリンクを調べてください。

http://www.nongnu.org/avr-libc/user-manual/

2.3.開発基盤

この資料に於けるコード例と結果検査は以下の環境とデバイスに基づきます。

1.統合開発環境 (IDE:Integrated Development Environment) :ATMEL AVR Studio 5 (5.0.1119版)
2.AVR GCC 8ビットツールチェーン版 :AVR_8_bit_GNU_Toolchain_3.2.1_292 (gcc 4.5.1版)
3.目的対象デバイス :ATMEL ATmega88PA

3.コード量を減らすための助言と秘訣

本章ではコード量を減らす方法についてのいくつかの助言を一覧にします。

3.1.助言1 - データ型と大きさ

可能な限り適用可能な最小のデータ型を使用してください。あなたのコードを、特にデータ型を評価してください。レジスタから8ビット(バイト)値を読むにはダブル バイト変数ではなく、バイトの大きさの変数だけが必要で、従ってコード空間を節約します。

8ビットAVRでのデータ型の大きさは<stdint.h>ヘッダファイルで得られ、表3-1で要約されます。

表3-1.<stdint.h>内の8ビットAVRでのデータ型

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

或るコンパイラ スイッチがこれを変更し得ることに注意してください (avr-gcc -mint8は整数データ型を8ビット整数に切り換えます)。

表3-2内の2つのコード例は異なるデータ型と大きさの影響を示します。avr-sizeユーティリティからの出力は、この応用が -Os(大きさ最適化)で構築された時に使用されるコード空間を示します。

表3-2.異なるデータ型と大きさの例

	unsigned int (16ビット)	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 (全1.1%)	プログラム:90バイト (全1.1%)
コンパイラ最適化段階	-Os (大きさに対して最適化)	-Os (大きさに対して最適化)

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char(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。mainで関数readADC()の戻り値と一時変数がint(2バイト)からchar(1バイト)に変更されるため、2バイトが節約されます。

注:main()から走行する前に始動コードがあります。それ故、単純なCコードは約90バイトを使用します。

3.2 助言 2 - 全域変数と局所変数

殆どの場合、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。変数が関数内で使用される場合、局所変数として関数の内側で宣言されるべきです。

理屈上、変数が全域または局所のどちらの変数として宣言されるかの選択はそれがどう使用されるかによって宣言されるべきです。

全域変数が宣言される場合、リンク時にプログラムに於いてこの変数に SRAM 内の独自のアドレスが割り当てられます。また、全域変数へのアクセスは代表的にそれのアドレスを得るのに追加バイト(通常、16ビット長アドレスに対して 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なるべくレジスタに割り当てられるか、または支援されていればスタックに配置されます。関数が活性になる時に、その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を取り去ることができます。

表 3-3には全域変数と局所変数の影響を示す 2つの例があります。

表 3-3.全域変数と局所変数の例

	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 (全 1.3%) (.text+.data+.bootbader) データ : 1バイト (全 0.1%) (.data+.bss+.noinit)	プログラム : 84バイト (全 1.0%) (.text+.data+.bootbader) データ : 0バイト (全 0.0%) (.data+.bss+.noinit)
コンパイラ最適化段階	-Os (大きさに対して最適化)	-Os (大きさに対して最適化)

左の例では、バイトの大きさの全域変数を宣言されています。avr-sizeユーティリティからの出力は最適化段階 -Os(量最適化)で 104バイトのコード空間と1バイトのデータ空間の使用を示します。

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

3.3 助言 3 - 繰り返し指標

繰り返しは 8ビットAVRコアで広範囲に使用されます。"while() {}"繰り返し、"for()"繰り返し、"do {}while()"繰り返しがあります。-Os(最適化任意選択が許可されている場合、コンパイラは同じコード量を持つように繰り返しを自動的に最適化します。

けれども未だコード量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(インクリメント)または減少(デクリメント)の繰り返し指標が異なるコード量を生じます。通常、0から最大値までの繰り返し計数(増加)を書きますが、最大値から0までの繰り返しの計数(減少)がもっと効率的です。

それは増加繰り返しに於いて、繰り返し指標が最大値に達したかを調べるために繰り返し毎に繰り返し指標を最大値で比較するための比較命令が必要なためです。

減少繰り返し使用時、この比較はそれが 0に達した場合に、減少された繰り返し指標の結果が SREG内の Z(0)フラグを設定(1)するためにそれ以上何も必要とされません。

表 3-4には増加と減少の繰り返し指標で "do {}while()"繰り返しによって生成されるコードを示す 2つの例があります。ここでは最適化段階 -Os(量最適化)が使用されます。

表 3-4.増加と減少の繰り返し指標での do {}while()繰り返しの例

	増加繰り返し指標での do{}while()	減少繰り返し指標での 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--;
		} while (local_1);
		}

AVRメモリ使用量	プログラム :96バイト (全 1.2%) (.text+.data+.bootbader) データ :0バイト (全 0.0%) (.data+.bss+.noinit)	プログラム :94バイト (全 1.1%) (.text+.data+.bootbader) データ :0バイト (全 0.0%) (.data+.bss+.noinit)
コンパイラ最適化段階	-Os (大きさに対して最適化)	-Os (大きさに対して最適化)

Cコード行に於いて明確な比較を持つように、この例は通常、C本で使用される "do{}while(--count);"のようではなく、"do{count--;}while(count);"のように書かれています。この 2形式は同じコードを生成します。

3.4.助言 4 - 詰め込み繰り返し

ここでの詰め込み繰り返しは各種繰り返しからより少ない繰り返しまたは 1つの繰り返しへ宣言と操作を統合することを参照し、故に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。そしてこれは長い繰り返しの一覧を導くかもしれません。この場合、実際に 1つに結合された繰り返しを持つことによって、詰め込み繰り返しはコード効率を増すのに役立つかもしれません。

詰め込み繰り返しはコード量を減らし、繰り返し周回の付随処理を無くすことによって、その上 コードをより速く走らせます。表 3-5での例から、詰め込み繰り返しがどう動くかを見ることができます。

表 3-5.詰め込み繰り返しの例

	分離繰り返し	詰め込み繰り返し

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 (全 2.0%) (.text+.data+.bootbader) データ :0バイト (全 0.0%) (.data+.bss+.noinit)	プログラム :98バイト (全 1.2%) (.text+.data+.bootbader) データ :0バイト (全 0.0%) (.data+.bss+.noinit)
コンパイラ最適化段階	-Os (大きさに対して最適化)	-Os (大きさに対して最適化)

3.5. 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体変数、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またはEEPROM）に配置されるべきです。このようにして貴重な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"を使用しません。オブジェクト"const"の宣言はその値が変更されないことを告げます。"const"はデータが"読み込み専用"であるべきで、最適化に対する機会を増すことをコンパイラに告げます。それはデータが格納されるべき場所と同一であると見做しません。

プログラム空間（読み込み専用）にデータを割り当ててプログラム空間からそれらを受け取るために、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関数は<avr\pgmspace.h>システムヘッダファイルで定義されます。

表 3-6での以下の例は全域文字列をプログラム空間内に移動することによってSRAMをどう節約するかを示します。

表 3-6. プログラム空間内の定数の例

	データ空間内の定数	プログラム空間内の定数

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（全1.5%） (.text+.data+.bootbader) データ　：12バイト（全1.2%） (.data+.bss+.noinit)	プログラム：102バイト（全1.2%） (.text+.data+.bootbader) データ　：0バイト（全0.0%） (.data+.bss+.noinit)
コンパイラ最適化段階	-Os（大きさに対して最適化）	-Os（大きさに対して最適化）

定数をプログラム空間に割り当てた後、プログラム空間とデータ空間の両方が減らされることを知ります。けれども、関数実行が直接SRAMから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処理（遅れ）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多数回使用される場合、直接多数回"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より少ない量を得ます。

データの各種の型をプログラム空間とで格納と取得を行うため、<avr\pgmspace.h>システムヘッダファイル内に、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使用者の手引きを調べてください。

3.6. 助言 6 - アクセス型 :static

全域データについて、可能な時は必ずstaticキーワードを使用してください。全域変数がstaticキーワードで宣言された場合、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でコードによって（外部変数として）変数の意図せぬ使用を防ぎます。

一方、関数内側の局所変数はstaticとしての宣言を避けられるべきです。"static"局所変数の値は関数呼び出しの間中保護が必要で、その変数はプログラム全体を通して存続します。従って、常駐のデータ空間（SRAM）記憶とそれをアクセスするための追加コードが必要です。これはその有効範囲がそれが定義された関数内であることを除き、全域変数と同様です。

static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、他のどのファイルからも呼ばれることがないため、最適化がより容易です。

static関数が最適化許可（-O1,-O2,-O3,-Os）でファイル内で一度だけ呼ばれる場合、関数はコンパイラによってインライン関数として自動的に最適化され、この関数に対してアセンブリ言語コードは全く出力されません。効果については表 3-7での例を調べてください。

表 3-7.アクセス型 static関数の例

	global関数（一度呼び出し）	static関数（一度呼び出し）

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（全1.9%） (.text+.data+.bootloader) データ　：12バイト（全1.2%） (.data+.bss+.noinit)	プログラム：140バイト（全1.7%） (.text+.data+.bootloader) データ　：12バイト（全1.2%） (.data+.bss+.noinit)
コンパイラ最適化段階	-Os（大きさに対して最適化）	-Os（大きさに対して最適化）

注：関数が複数回呼ばれる場合、それが直接関数呼び出しよりも大きなコードを生成するため、インライン関数に最適化されません。

3.7 助言 7 - 低位アセンブリ命令

良く書かれたアセンブリ命令は常に最良に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作成者に推奨されません。

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の連携で度々苦労を減らし、可読性と可搬性を改善します。2,3行のアセンブリコードよりも少なく生成する作業に対して、関数の代わりにマクロを使用してください。表 3-8での例は関数使用と比べたアセンブリマクロのコード用法を示します。

表 3-8.低位アセンブリ命令の例

	関数	アセンブリ マクロ

Cソース コード	`#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	プログラム：90バイト（全1.1%） (.text+.data+.bootloader) データ　：0バイト（全0.0%） (.data+.bss+.noinit)	プログラム：86バイト（全1.0%） (.text+.data+.bootloader) データ　：0バイト（全0.0%） (.data+.bss+.noinit)
コンパイラ最適化段階	-Os（大きさに対して最適化）	-Os（大きさに対して最適化）

8ビットAVRのCでのアセンブリ言語使用についてのより多くの詳細に関しては、avr-libc使用者の手引き内の「インライン アセンブリ料理本」章を参照してください。

4.実行時間を減らす助言と秘訣

本章では実行時間をどう減らすかについてのいくつかの助言を一覧にします。各助言に関して、いくつかの説明と試供コードが与えられます。

4.1.助言 8 - データ型と大きさ

コード量の削減に加えて、正しいデータ型と大きさの選択はその上実行時間も減らします。8ビットAVRについて、8ビット(バイト)値のアクセスは常に最も効率的な方法です。

8ビットと16ビットの変数の違いに関して表 4-1.での例を調べてください。

表 4-1.データ型と大きさの例

	16ビット変数	8ビット変数

Cソース コード	`#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	プログラム : 94バイト (全 1.1%) (.text+.data+.bootloader) データ : 0バイト (全 0.0%) (.data+.bss+.noinit)	プログラム : 92バイト (全 1.1%) (.text+.data+.bootloader) データ : 0バイト (全 0.0%) (.data+.bss+.noinit)
周期計数器	90	79
コンパイラ最適化段階	-O2	-O2

注 :繰り返しは -O3任意選択でのコンパイラによって自動的に展開されます。その後にこの繰り返しは繰り返し指標によって示される繰り返し操作に広げられ、故にこの例は -O3任意選択許可で違いは全くありません。

4.2.助言 9 - 条件命令文

通常、正常なコード行での事前減少と事後減少 (または事前増加と事後増加)は違いを生じません。例えば、"i--"と"--i"は単に同じコードを生成します。けれども、繰り返し指標としてと条件命令文でそれらの演算子の使用は違うコードを生成します。

助言 3 - 繰り返し指標で述べられたように、減少繰り返し指標の使用はより小さなコード量に帰着します。これは条件命令文でより速いコードを得るのにも役立ちます。

その上、事前減少と事後減少も異なる結果を持ちます。表 4-2.での例から、より速いコードが事前減少条件命令文で生成されるのを見ることができます。ここでの周期計数器値は最長繰り返しの実行時間を表します。

表 4-2.条件命令文の例

	条件命令文内の事後減少	条件命令文内の事前減少

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 (全 1.3%) (.text+.data+.bootbader) データ : 0バイト (全 0.0%) (.data+.bss+.noinit)	プログラム : 102バイト (全 1.2%) (.text+.data+.bootbader) データ : 0バイト (全 0.0%) (.data+.bss+.noinit)
周期計数器	75	61
コンパイラ最適化段階	-O3	-O3

例の動きが同じ(各回転でポートBのB0が1回切り換わると同時にポートCのPC0が9回切り換わる)ことを保証するために、"loop_cnt"は表4-2での2つの例に於いて異なる値で割り当てられます。

4.3.助言 10 - 繰り返しの展開

いくつかの場合で、コード実行の速度向上の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効果的です。繰り返しが展開された後、検査されるべき繰り返し指標が全くなく、繰り返しで各周回でより少ない分岐が実行されます。

表 4-3での例は1つのポートピンを10回切り換えます。

表 4-3.繰り返しの展開の例

	繰り返し	繰り返しの展開

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 (全 1.1%) (.text+.data+.bootbader) データ : 0バイト (全 0.0%) (.data+.bss+.noinit)	プログラム : 142バイト (全 1.7%) (.text+.data+.bootbader) データ : 0バイト (全 0.0%) (.data+.bss+.noinit)
周期計数器	80	50
コンパイラ最適化段階	-O2	-O2

do{}while()繰り返しの展開により、コード実行を80クロック周期から50クロック周期へ著しく速度向上します。

繰り返しの展開後にコード量が94バイトから142バイトに増やされていることに注意してください。これは速度と量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、ｺﾝﾊﾟｲﾗはこの繰り返しを自動的に展開して手動で繰り返しを展開したのと同じｺｰﾄﾞを生成します。

4.4.助言 11 - 流れ制御 ：if-elseとswitch-case

"if-else"と"switch-case"はCｺｰﾄﾞで広範囲に使用され、正しい分岐構造は実行時間を減らすことができます。

"if-else"については、最初の場所で常に最もありそうな条件を配置してください。その後に後続する条件は実行がより少なそうです。従って殆どの場合に時間が節約されます。

"switch-case"の使用は"switch-case"に対して通常ｺﾝﾊﾟｲﾗが指標と共に参照表を生成して正しい位置へ直接飛ぶため、"if-else"の欠点を消し去るかもしれません。

"switch-case"の使用が難しいなら、"if-else"分岐をより小さな副分岐に分割することができます。この方法は最悪条件に対する実行を減らします。下の例では、ADCからの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 4-4. if-else副分岐の例

	if-else分岐	if-else副分岐

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;`
	`  }`
	`  send(output);`
	`}`
		`#include <avr/io.h>`
		
		`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;`
		`  }`
		`  else{`
		`  output = 0x48;`
		`  }`
		`  }`
		`  else{`
		`  if (ad_result <= 210){`
		`  output = 0x57;`
		`  }`
		`  else{`
		`  output = 0x45;`
		`  }`
		`  }`
		`  }`
		`  send(output);`
		`}`

AVRﾒﾓﾘｰ使用量	ﾌﾟﾛｸﾞﾗﾑ：198ﾊﾞｲﾄ (全2.4%) (.text+.data+.bootbader) ﾃﾞｰﾀ　：1ﾊﾞｲﾄ (全0.1%) (.data+.bss+.noinit)	ﾌﾟﾛｸﾞﾗﾑ：226ﾊﾞｲﾄ (全2.8%) (.text+.data+.bootbader) ﾃﾞｰﾀ　：1ﾊﾞｲﾄ (全0.1%) (.data+.bss+.noinit)
周期計数器	58 (最悪の場合に対して)	48 (最悪の場合に対して)
ｺﾝﾊﾟｲﾗ最適化段階	-O3	-O3

最悪の場合で分岐に達するのに、より少ない時間が必要なのを見ることができます。故に量と速度での特定の必要条件に従って平衡にすべきです。

5.例応用と検査結果

例応用は先に言及された助言と秘訣を示すのに使用されます。この例では量最適化 -s任意選択が許可されます。

全てではなく多数の助言と秘訣がこの例応用の最適化に使用されます。

この例では、入力を採取するのに1つのADCﾁｬﾈﾙが使用され、5秒毎に結果がUSARTを通して送り出されます。ADCの結果が範囲外の場合、応用が異常状態で固定化される前に30秒間警報が送り出されます。main繰り返しの残りでは、ﾃﾞﾊﾞｲｽがﾊﾟﾜｰｾｰﾌﾞ動作形態に置かれます。

最適化前と最適化後の試供応用の速度と量の最適化の結果が表5-1で一覧にされます。

表5-1.例応用の速度と量の最適化の結果

検査項目	最適化前	最適化後	検査結果
ｺｰﾄﾞ量	1444ﾊﾞｲﾄ	630ﾊﾞｲﾄ	-56.5%
ﾃﾞｰﾀ量	25ﾊﾞｲﾄ	0ﾊﾞｲﾄ	-100%
実行速度 (注)	3.88ms	2.6ms	-33.0%

注 : 5回のADC採取と1回のUSART送信を含む1周回

6.結び

この資料に於いて、量と速度でのCｺｰﾄﾞの効率性について一覧にされたいくつかの助言と秘訣があります。最新のCｺﾝﾊﾟｲﾗのおかげで、それらは異なる状態で異なる最適化任意選択を自動的に起動することに於いて賢いのです。けれども、ｺﾝﾊﾟｲﾗが開発者よりもｺｰﾄﾞを良く知ることはなく、故に良い書き方が常に重要です。

例で示されたように、1方向の最適化は他での影響を持ちます。特定の要求に基づいてｺｰﾄﾞ量と速度間の平衡が必要です。

例えCｺｰﾄﾞ最適化に関するそれらの助言と秘訣を持っていても、それらのより良い使い方に関して、作業でのﾃﾞﾊﾞｲｽとｺﾝﾊﾟｲﾗの充分な理解が絶対的に必要です。そして確かに、各種の応用の場合に於いてｺｰﾄﾞを効率的に最適化するための他の技術と方法があります。

7.目次

Atmel Corporation
2325 Orchard Parkway
San Jose, CA 95131
USA
TEL (+1)(408) 441-0311
FAX (+1)(408) 487-2600
www.atmel.com

Atmel Asia Limited
Unit 01-5 & 16, 19F
BEA Tower, Millennium City 5
418 Kwun Tong Road
Kwun Tong, Kowloon
HONG KONG
TEL (+852) 2245-6100
FAX (+852) 2722-1369

Atmel Munich GmbH
Business Campus
Parking 4
D-85748 Garching b. Munich
GERMANY
TEL (+49) 89-31970-0
FAX (+49) 89-3194621

Atmel Japan
104-0032東京都品川区
大崎 1-6-4
新大崎勧業ﾋﾞﾙ 16F
ｱﾄﾒﾙ ｼﾞｬﾊﾟﾝ株式会社
TEL (+81)(3)-6417-0300
FAX (+81)(3)-6417-0370

本応用記述はATMELのAVR4027応用記述（doc8453.pdf Rev.8453A-11/11）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。