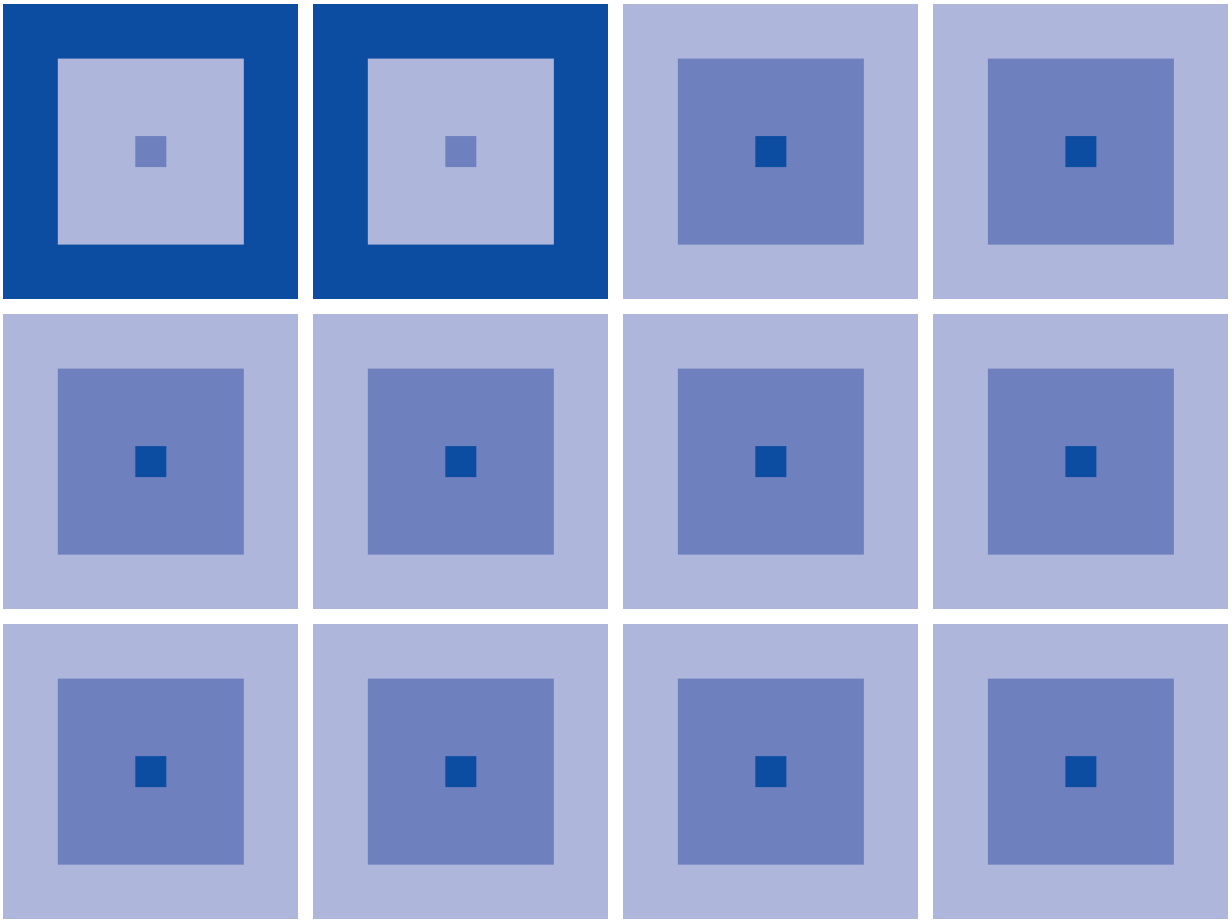
EPSON

CMOS 32-BIT SINGLE CHIP MICROCOMPUTER

S1C33 Family
スタンダードコア用
アプリケーションノート

(S5U1C33001Cツール対応)



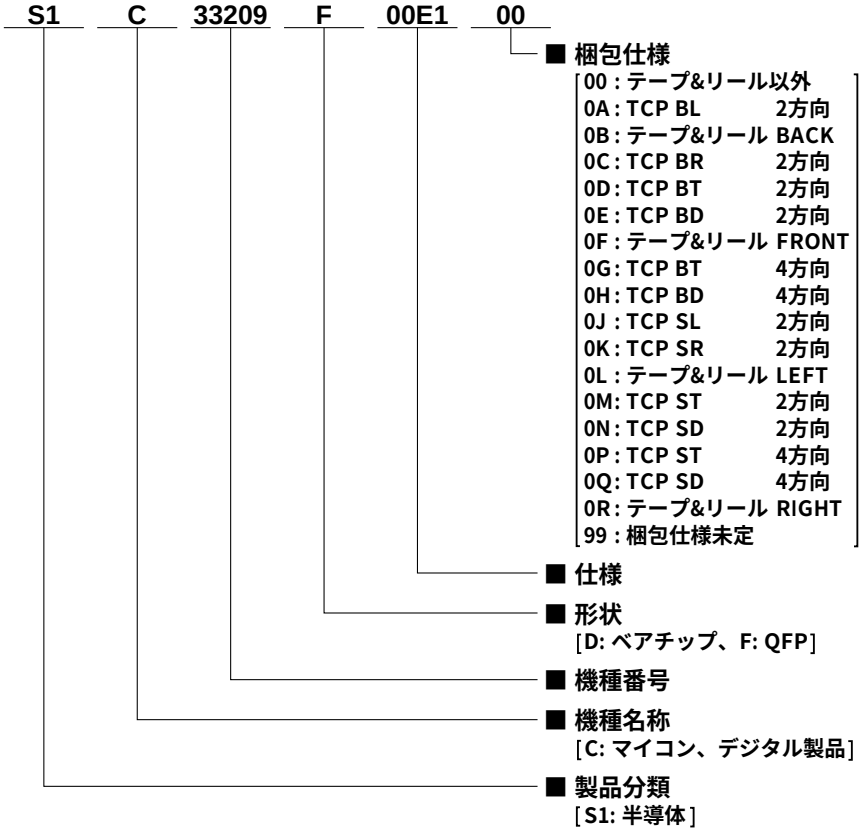
セイコーエプソン株式会社

本資料のご使用につきましては、次の点にご留意願います。

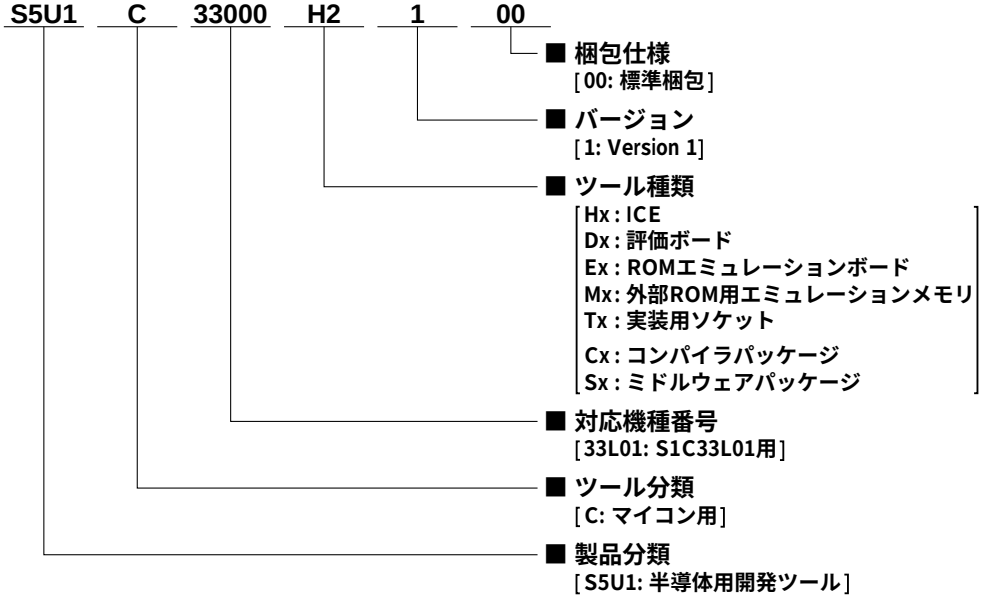
1. 本資料の内容については、予告なく変更することがあります。
2. 本資料の一部、または全部を弊社に無断で転載、または、複製など他の目的に使用することは堅くお断りします。
3. 本資料に掲載される応用回路、プログラム、使用方法等はあくまでも参考情報であり、これらに起因する第三者の権利(工業所有権を含む)侵害あるいは損害の発生に対し、弊社は如何なる保証を行うものではありません。また、本資料によって第三者または弊社の工業所有権の実施権の許諾を行うものではありません。
4. 特性表の数値の大小は、数直線上の大小関係で表しています。
5. 本資料に掲載されている製品のうち、「外国為替および外国貿易法」に定める戦略物資に該当するものについては、輸出する場合、同法に基づく輸出許可が必要です。
6. 本資料に掲載されている製品は、一般民生用です。生命維持装置その他、きわめて高い信頼性が要求される用途を前提としていません。よって、弊社は本(当該)製品をこれらの用途に用いた場合の如何なる責任についても負いかねます。

製品型番体系

●デバイス



●開発ツール



はじめに

本書はS1C33 Familyマイクロコンピュータを使用するアプリケーション開発者向けのマニュアルで、S1C33チップ(特にS1C33209)用のプログラミング、基本回路設計、音声出力方法などを説明します。
なお、本書に掲載のプログラム例は、S1C33 Family C Compiler Package(S5U1C33001C)のver. 1以降に含まれるサンプルから抜粋しています。

目次

1 S1C33000 CPUコアについて

S1C33000は、CMOS 32ビットシングルチップマイクロコンピュータS1C33 Family共通のCPUコアです。このコアを中心にROM、RAM、DMA、A/D変換器、タイマ等の周辺回路を加えS1C33 Familyのプロセッサ群を展開しています。

S1C33000の主な特長は次のとおりです。

●高コード効率を実現する命令セット

●高速動作、積和演算機能

●小さいCPUコアサイズ

●低消費電流

携帯機器からOA、FA機器、デジタル信号処理、制御機器まで組み込み用途に幅広く対応します。

1.1 概要

- 種類 セイコーエプソンオリジナル32ビットRISCコア
- 動作周波数 ..... DC～60MHz(S1C33xxxの機種により変わります)
- インストラクションセット 16ビット固定長
  直行性の良い105種の命令
  主要な命令は1サイクルで実行可能
- 積和演算機能 ..... MAC命令(16ビット×16ビット+64ビット ⇒ 64ビット)
  2サイクル/1積和で実行
- レジスタセット 32ビット汎用レジスタ 16本
  32ビット特殊レジスタ 5本
- メモリ空間 ..... 28ビット(256MB)空間
  命令、データ、I/O混在型リニア空間
  19エリアに分割し、コアよりセレクト信号出力
- 即値拡張 ..... EXT命令にて、命令の即値を32ビットまで拡張
- 割り込み ..... リセット、NMI、外部割り込み 216本
  ソフト例外 4本、命令実行例外 2種
  トラップテーブルよりベクタをフェッチし分岐
- リセット ..... コールドリセット(すべてリセット)
  ホットリセット(バスはリセットしない)
  ブート時、トラップテーブルは内蔵ROM、外部ROMより選択
  その後、再配置可能
- パワーダウンモード ..... HALT命令(コアのみ止まる)
  SLP命令(すべて止まる)
- その他 ..... リトルエンディアン(標準)/ビッグエンディアン
  ハーバードアーキテクチャ

1.2　メモリマップ

			サイズ
0xFFFFFFF	エリア18	外部メモリ	64MB
	エリア17	外部メモリ	64MB
	エリア16	外部メモリ	32MB
	エリア15	外部メモリ	32MB
	エリア14	外部メモリ	16MB
	エリア13	外部メモリ	16MB
	エリア12	外部メモリ	8MB
0x1000000	エリア11	外部メモリ	8MB
0x0C00000	エリア10	外部メモリ	4MB
	エリア9	外部メモリ	4MB
	エリア8	外部メモリ	2MB
	エリア7	外部メモリ	2MB
	エリア6	外部I/O	1MB
	エリア5	外部メモリ	1MB
0x0100000	エリア4	外部メモリ	1MB
0x0080000	エリア3	内蔵ROM	512KB
0x0060000	エリア2	ICE用に予約	128KB
0x0040000	エリア1	内蔵I/O	128KB
0x0000000	エリア0	内蔵RAM	256KB

1.3　トラップテーブル

	アドレス オフセット
予約	1023
マスク可能な外部割り込み215	929
:	
マスク可能な外部割り込み0	64
ソフトウェア例外3	60
:	
ソフトウェア例外0	48
予約	32~44
NMI	28
アドレス不整例外	24
予約	20
ゼロ除算	16
予約	4~12
リセット	0

トラップテーブル先頭アドレス
コールドリセット時は0x0C00000
リセット後、トラップテーブルはトラップテーブルベースレジスタ(TTBR、メモリにマップされたレジスタ)により再配置が可能
ブート、割り込み時はテーブルよりベクタフェッチ

割り込みのシーケンス
1）PCをスタックに退避
2）PSRをスタックに退避、IEをディセーブル
3）トラップテーブルよりベクタをフェッチ
4）ベクタアドレスにジャンプ

リセットのシーケンス
1）リセットベクタをフェッチ
2）ベクタアドレスにジャンプ

1.4　CPUレジスタ

PSR

31~12	11~8	7	6	5	4	3	2	1	0
Reserved	IL	MO	DS	–	IE	C	V	Z	N

IL:　**割り込みレベル**　**(0~15: 割り込み許可レベル)**
MO:　**MACオーバーフローフラグ**　**(1: オーバーフローあり, 0: なし)**
DS:　**被除数符号フラグ**　**(1: 負, 0: 正)**
IE:　**割り込み許可**　**(1: 許可, 0: 禁止)**
Z:　**ゼロフラグ**　**(1: ゼロ, 0: ゼロ以外)**
N:　**ネガティブフラグ**　**(1: 負, 0: 正)**
C:　**キャリーフラグ**　**(1: キャリー/ボローあり, 0: なし)**
V:　**オーバーフローフラグ**　**(1: オーバーフローあり, 0: なし)**

1.5　命令セットの特長

● 命令の分類

命令は以下の8種類の機能に分類されます。

- **8、16、32ビットデータ転送命令**
  LD.B, LD.UB, LD.H, LD.UH, LD.W
  レジスタ～メモリ間、2つのレジスタ間で8、16、32ビットのデータ転送を行います。
- **32ビット算術・論理演算命令**
  AND, OR, XOR, NOT, ADD, ADC, SUB, SBC, CMP, MLT.H, MLTU.H(16ビット), MLT.W, MLTU.W, DIV0S, DIV1S, DIV2S, DIV3S
  2つのレジスタの値、またはレジスタと即値で32ビットの演算を行います。
- **32ビットシフト&ローテート命令**
  SRL, SLL, SRA, SLA, RR, RL
  32ビットレジスタのデータを0～8ビットシフト、ローテートします。
- **ビット操作命令**
  BTST, BSET, BCLR, BNOT
  メモリ上のバイトデータに対して1ビットの処理を行います。
- **スタック操作命令**
  PUSHN, POPN
  R0からRnの内容を連続してスタックに保存、およびスタックから復帰します。
- **分岐命令**
  JRGT, JRGE, JRLT, JRLE, JRUGT, JRUGE, JRULT, JRULE, JREQ, JRNE, CALL, JP, RET, RETI, RETD, INT, BRK
  各種の条件付きジャンプ、コールとリターンなどを行います。
- **システム制御命令**
  HALT, SLP, NOP
  パワーダウン用の命令とノーオペレーション命令です。
- **その他**
  MAC, SCAN0, SCAN1, SWAP, MIRROR, EXT
  積和演算、データのスキャンや入れ替えなどを行います。

● アドレッシングモード

(1)基本アドレッシングモード

これは1命令で実現できるアドレッシングモードです。

- **6ビット即値データアドレッシング**

```
LD.W  %R1,sign6          6ビットデータを符号拡張してR1レジスタにロード
ADD   %R2,imm6           6ビットデータをR2レジスタに加算
```

6ビットの符号付き/符号なし即値データとレジスタの演算を行うモードです。

- **レジスタ直接アドレッシング**

```
LD.W  %R1,%R2            R2レジスタからR1レジスタへのデータ転送
JP    %R3                R3レジスタが保持するアドレスへのジャンプ
```

レジスタの値のみで演算などの処理を行うモードです。

- **レジスタ間接アドレッシング**

```
LD.B  %R2,[%R15]         R15で指定されるアドレスから符号付き8ビットデータをロード
LD.W  %R2,[%R15]+        R15で指定されるアドレスから32ビットデータをロード後、
                         R15レジスタをインクリメント
```

レジスタにメモリアドレスを設定し、そのアドレスのデータを扱うモードです。

- **ディスプレースメント付きSP間接アドレッシング**

```
LD.UB %R15,[%SP+imm6]    SP+imm6のアドレスから符号なし8ビットデータをロード
LD.W  %R15,[%SP+imm6]    SP+(imm6×4)のアドレスから32ビットデータをロード
```

スタックポインタからのオフセットアドレスを指定して、スタック内のデータを扱うモードです。

- **符号付き8ビットPC相対アドレッシング**

```
JP    sign8              現在のPCアドレスから+127～-128命令の位置にジャンプ
CALL  sign8              現在のPCアドレスから+127～-128命令の位置にあるサブルーチンをコール
```

PCからの相対アドレスを指定して分岐を行うモードです。

(2)拡張アドレッシングモード

EXT命令により基本アドレッシングモードを拡張することができます。

- **拡張即値データアドレッシング**

```
EXT  imm13 + ADD  %R1,imm6               → ADD %R1,imm19
EXT  imm13 + EXT  imm13 + ADD  %R1,imm6  → ADD %R1,imm32
```

EXT命令で即値サイズを19ビット、32ビットに拡張できます。

- **拡張レジスタ間接アドレッシング**

```
EXT  imm13 + LD.W  %R2,[%R15]+                → LD.W  %R2,[%R15+imm13]
EXT  imm13 + EXT  imm13 + LD.W  %R2,[%R15]+   → LD.W  %R2,[%R15+imm26]
```

EXT命令で13ビット、26ビットのオフセットアドレスを付加できます。

- **拡張ディスプレースメント付きSP間接アドレッシング**

```
EXT  imm13 + LD.B  %R15,[%SP+imm6]                  → LD.B  %R15,[%SP+imm19]
EXT  imm13 + EXT  imm13 + LD.B  %R15,[%SP+imm6]     → LD.B  %R15,[%SP+imm32]
```

EXT命令でオフセットを19ビット、32ビットに拡張できます。

- **拡張PC相対アドレッシング**

```
EXT  imm13 + CALL  sign8                → CALL  sign22
EXT  imm13 + EXT  imm13 + CALL  sign8   → CALL  sign32
```

EXT命令で分岐範囲を符号付き22ビット、32ビットに拡張できます。

● C言語に対する高いコード密度

S1C33 CPUコアでは2つの考えによりC言語に対して高いコード密度を実現しています。

1. Cでよく使用する出現頻度の高いパターンはできるだけ1命令で処理する
2. それ以外のパターンもEXT命令で最小の命令数に抑え、出現頻度の低いパターンについてもコード密度の悪化を防ぐ

(1)分岐パターン

・条件分岐

`JRNE　sign8`（+127～-128命令の分岐エリア）
　1命令(2バイト)で条件分岐の90%以上に対応
`EXT　imm13 + JRNE　sign8 → JRNE　sign22`（±2Mの分岐エリア）
　上記以外は2命令(4バイト)で対応

・サブルーチンコール

`EXT　imm13 + CALL sign8 → CALL sign22`（±2Mの分岐エリア）
　2命令(4バイト)でほとんどのケースに対応
`EXT　imm13 + EXT　imm13 + JRNE　sign9 → JRNE sign32`（すべてのエリアに分岐可能）
　上記以外は3命令(6バイト)で対応

(2)変数アクセスパターン

・オート変数アクセス

`LD.W　%R2,[%SP+imm6]`（intアクセス時SP+0～255のエリアをアクセス）
　1命令(2バイト)で80%以上のオート変数に対応
`EXT　imm13 + LD.W　%R2,[%SP+imm6] → LD.W　%R2,[%SP+imm19]`（512Kバイトエリアをアクセス）
　上記以外は2命令(4バイト)で対応

・ポインタ変数アクセス

`LD.B　%R2,[%R3]`
　1命令(2バイト)

・スタティック変数アクセス(グローバルポインタ方式)

`EXT　imm13 + LD.H　%R2,[%R15] → LD.H　%R2,[%R15+imm13]`（R15からの8Kバイトエリアをアクセス）
　2命令(4バイト)
`EXT　imm13 + EXT　imm13 + LD.H　%R2,[%R15] → LD.W　%R2,[%R15+imm26]`
　3命令(6バイト)

(3)演算パターン

・2オペランド、レジスタ - 即値

`ADD　%R2,imm6`（0～63をR2に加算）
　1命令(2バイト)
`EXT　imm13 + ADD %R2,imm6 → ADD　%R2,imm19`（0～512KをR2に加算）
　2命令(4バイト)
`EXT　imm13 + EXT　imm13 + ADD　%R2,imm6 → ADD　%R2,imm32`
　3命令(6バイト)

・2オペランド、レジスタ - レジスタ

`ADD　%R2,%R3`（R3をR2に加算）
　1命令(2バイト)

(4) その他

- **コール、リターン**

```
CALL  sign8              PCを自動的に待避
RET                      PCを自動的に復帰
```

それぞれ1命令削減

- **プッシュ、ポップ**

```
PUSHN  %Rn               R0～Rnをスタックに待避
POPN   %Rn               R0～Rnをスタックから復帰
```

サブルーチンごとに数命令を削減

- **データ変換**

```
LD.B  %R2,%R3            符号付き8ビットデータを32ビットに変換
LD.UB / LD.H / LD.UH     符号付き/符号なし8ビット、16ビットもサポート
```

Cのデータキャストに最適

- **ビット操作**

```
BSET  [%R5],2            [%R5](バイトサイズのメモリデータ)のビット2を1にセット
BCLR / BTST / BNOT       ビットのクリア、テスト、反転
```

3命令によるリード・モディファイ・ライトを1命令で実現

1.6 命令実行速度

以下に示す実行サイクル数はプログラムが内蔵ROM、データが内蔵RAMにありハーバードアーキテクチャで動作する場合です。外部メモリなどへのアクセスの場合、ウェイトなどのサイクルが追加されます。

- **レジスタ→レジスタ動作(算術・論理演算、システム、その他)**

AND, OR, XOR, NOT, ADD, ADC, SUB, SBC, CMP, MLT.H, MLTU.H, DIV0S, DIV1S, DIV2S, DIV3S, SRL, SLL, SRA, SLA, RR, RL, HALT, SLP, NOP, LD.B, LD.UB, LD.H, LD.UH, LD.W

1サイクル/命令

MLT.W, MLTU.W

5サイクル/命令

- **メモリ→レジスタ動作(ld.w, ld.b, ld.ub, ld.h, ld.uh)**

%RD, [%RB](インターロックなし), [%RB], %RS, %RD, [%SP+imm6], [%SP+imm6], %RS, [%RB]+, %RS

1サイクル/命令

%RD, [%RB]+, %RD, [%RB](インターロックあり)

2サイクル/命令

- **メモリ→メモリ動作**

BTST, BSET, BCLR, BNOT

3サイクル/命令

- **分岐動作**

JRGT, JRGE, JRLT, JRLE, JRUGT, JRUGE, JRULT, JRULE, JREQ, JRNE, JP

通常の分岐: 2サイクル/命令、遅延分岐(xxx.d): 1サイクル/命令

CALL, JP, RET, RETI, RETD, INT, BRK

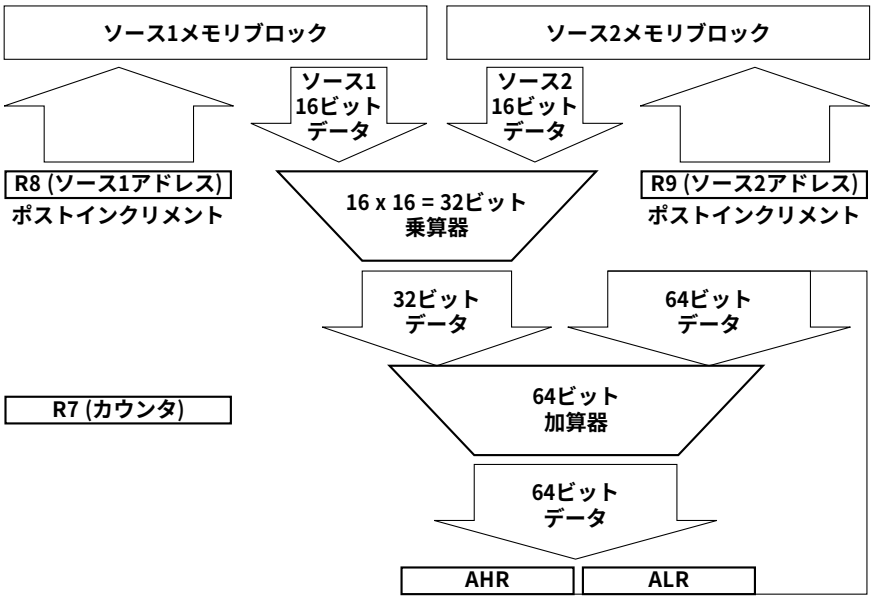
2～10サイクル/命令

- **その他の動作**

MAC	2×N+4サイクル
PUSHN, POPN	1×Nサイクル
SCAN0, SCAN1, SWAP, MIRROR	1サイクル/命令

1.7　積和演算機能

MAC命令は、1命令で16ビット × 16ビット + 64ビットの積和演算を2クロックごとに最大2G回実行できます。



例）
MAC　%R7
R7: 繰り返しカウンタ（最大4G）
R8: ソース1アドレス（ポストインクリメント）
R9: ソース2アドレス（ポストインクリメント）

ソース1とソース2の16ビットデータをそれぞれ読み出して乗算します。次に乗算結果の32ビットデータをAHR:ALRという64ビットレジスタに加算します。これを2クロックに1回ずつ繰り返します（ソース1、ソース2がともに内蔵RAMにある場合）。

1.8 命令セット一覧

● 命令形式と動作

（実行サイクル数は内蔵ROMに命令があり、内蔵RAMにアクセスする場合）

分類	命令	命令形式例	動作	実行サイクル数
相対分岐	jp, jrgt, jrge, jrlt, jrle, jrugt, jruge, jrult, jrule, jreq, jrne, call	jp sing8	PC + (sign8 × 2)に分岐	1,2(分岐時) callは3
相対遅延分岐	jp.d, jrgt.d, jrge.d, jrlt.d, jrle.d, jrugt.d, jruge.d, jrult.d, jrule.d, jreq.d, jrne.d, call.d	jp.d sing8	PC + (sign8 × 2)に分岐 分岐時に次の命令を実行	1 callは2
絶対分岐	call, jp, call.d, jp.d	call %rb	%rbの示すアドレスに分岐	1〜3
特殊分岐	ret, ret.d, int imm2, reti, brk, retd		リターン、割り込み等	3〜10
論理演算	and, or, xor, not	and %rd, %rs and %rd, sign6	%rd = %rd & %rs %rd = %rd & sign6	1

算術演算	add, sub	add %rd, %rs
		add %rd, imm6
		add %sp, imm12
			%rd = %rd + %rs
			%rd = %rd + imm6
			%sp = %sp + imm12
				1

比較演算	cmp	cmp %rd, %rs cmp %rd, sign6	%rd - %rs フラグのみ変化 %rd - sign6	1
キャリー演算	adc, sbc	adc %rd, %rs	%rd = %rd + %rs + carry flag	1
乗算	mlt.h, mlt.uh (16bit) mlt.w, mlt.uw (32bit)	mlt.h %rd, %rs	%alr = %rd × %rs (32 = 16 × 16) %ahr:%alr = %rd × %rs (64 = 32 × 32)	1 5
除算	div0s, div0u, div1, div2s, div3s		これらの組み合わせで除算実行	1
シフト	srl, sll (logical shift) sra, sla (arithmetical shift) rr, rl (rotate)	srl %rd, imm4 srl %rd, %rs	%rd = %rd >> imm4 %rd = %rd >> %rs シフト量は0〜8	1
メモリロード	ld.b (signed 8bit load) ld.ub (unsigned 8bit load) ld.h (signed 16bit load) ld.uh (unsigned 16bit load) ld.w (32bit load)	ld.w %rd, [%sp+imm6] ld.w [%sp+imm6], %rs ld.w %rd, [%rb] ld.w %rd, [%rb]+ ld.w [%rb], %rs ld.w [%rb]+, %rs	%rd = [%sp+imm6] スタック相対アクセス [%sp+imm6] = %rs %rd = [%rb] レジスタアドレスアクセス %rd = [%rb], %rb = %rb + 4 ポストINC. [%rb] = %rs [%rb] = %rs, %rb = %rb + 4	1〜2
レジスタロード	ld.w	ld.w %rd, %rs ld.w %rd, sign6 ld.w %rd, %ss ld.w %ss, %rs	レジスタ間コピー 即値代入 特殊レジスタよりコピー 特殊レジスタへコピー	1
変換	ld.b, ld.ub, ld.h, ld.uh	ld.b %rd, %rs	型変換を行う	1
ビット	btst, bset, bclr, bnot	btst [%rb], imm3	1bitのテスト、セット、クリア、反転	3
システム	nop, slp, hlt		ノーオペレーション、クロック停止	1
積和	mac		%ahr:%alr= [%r14] × [%r15] + %ahr:%alr を%r13回繰り返す	2 × N + 4
スタック	pushn, popn	pushn %rs	%r0から%rsまで連続push/pop	1 × N
スキャン	scan0, scan1	scan0 %rd, %rs	MSBから0や1の長さを計算、8bitまで	1
スワップ	swap, miror	swap %rd, %rs	8bit単位のビットスワップ、ミラー	1
拡張	ext	ext imm13	命令の即値を拡張	1

signX, immX: 即値、%XX: レジスタ

● EXT命令による即値拡張

拡張例）

命令単独	EXT命令を1つ付加	EXT命令を2つ付加
call sign8	ext imm13 call sign8 (= call sign22)	ext imm13 ext imm13 call sign8 (= call sign32)

分類	命令	1命令での形式例	EXT命令を1つ付加した動作例	EXT命令を2つ付加した動作例
相対分岐	jp, jrgt, jrge, jrlt, jrle, jrugt, jruge, jrult, jrule, jreq, jrne, callとその相対遅延分岐	jp sing8	jp sign22	jp sign32
演算	add, sub, and, or, xor, not, cmp, ld.w	add %rd, imm6 /sign6	add %rd, imm19/sign19	add %rd, imm32
スタックロード	ld.b, ld.ub, ld.h, ld.uh, ld.w	ld.w %rd, [%sp+imm6] ld.w [%sp+imm6], %rs	[%sp+imm19] オフセット値拡張	[%sp+imm32] オフセット値拡張
絶対ロード	ld.b, ld.ub, ld.h, ld.uh, ld.w	ld.w %rd, [%rb] ld.w %rd, [%rb]+ ld.w [%rb], %rs ld.w [%rb]+, %rs	[%rb+imm13] オフセット値付加	[%rb+imm26] オフセット値付加
ビット	btst, bset, bclr, bnot	btst [%rb], imm3	[%rb+imm13] オフセット値付加	[%rb+imm26] オフセット値付加

signX, immX: 即値、%XX: レジスタ

2　S1C33用プログラムの書き方

ここでは、S1C33 Family共通のプログラムの書き方について説明します。

2.1　ベクタテーブルとブートルーチン

S1C33用プログラムにはベクターテーブルとブートルーチンが最低限必要です。
S1C33チップはパワーオン時のコールドリセットにより、通常0xC00000番地からリセットベクタをフェッチして、そのアドレスから実行を開始します。そこで、最も簡単なアセンブラ記述例は次のようになります。

```
        .set SP_INI,0x0800    ; sp is in end of 2KB internal RAM
        .set DP_INI,0x0000    ; default data area pointer %r15 is 0x0

        .text
        .long BOOT            ; BOOT VECTOR
BOOT:
        xld.w  %r15,SP_INI
        ld.w   %sp,%r15       ; set SP
        ld.w   %r15,DP_INI    ; set default data area pointer
        xcall  main           ; goto main
        xjp    BOOT           ; infinity loop
```

実際のアプリケーションでは、他の例外や割り込みに対応したベクタテーブルとBCUの設定や周辺機能の初期化等を含むブートルーチンが必要になります。
以下に、それらを含む、アセンブラとCのそれぞれで記述した実用的な例を示します。

● アセンブラでの記述

ベクタテーブルの例です。

ベクタテーブル　vector.s

```
        .text

        .long  RESET          ; ベクタテーブル
        .long  RESERVED
        .long  RESERVED
        .long  RESERVED
        .long  ZERODIV
        .long  RESERVED
        .long  ADDRERR
        .long  NMI
        .long  RESERVED
        .long  RESERVED
        .long  RESERVED
        .long  RESERVED
        .long  SOFTINT0
        .long  SOFTINT1
        .long  SOFTINT2
        .long  SOFTINT3
        .long  INT0
        .long  INT1
        .long  INT2
        .long  INT3
        .long  INT4
        .long  INT5
                |
            (INT6–INT49)
                |
        .long  INT50
        .long  INT51
        .long  INT52
        .long  INT53
        .long  INT54
        .long  INT55

RESET:                        ; 未定義ベクタ用ダミーラベル
ZERODIV:
ADDRERR:
```

```
NMI:
RESERVED:
SOFTINT0:
SOFTINT1:
SOFTINT2:
SOFTINT3:
INT0:
INT1:
INT2:
INT3:
INT4:
INT5:
   |
(INT6–INT49)
   |
INT50:
INT51:
INT52:
INT53:
INT54:
INT55:
       .globalINT_LOOP
INT_LOOP:                       ; 未定義ベクタ用トラップルーチン
       nop
       jp      INT_LOOP
       reti
```

このファイルでは、ブートからハードウェア割り込みまで、

.long　　ラベル

という形でベクタテーブルを定義しています。このようにして、32ビットの飛び先アドレスが格納できます。
また、安全のため、特に定義されないアドレスは 一番下にあるINT_LOOPに飛ぶようにしています。
なお、このプログラムは実際に使用するベクタを別名で再定義することを前提としています(下の飛び先ラベルをほかの場所に移動して処理ルーチンを記述する方法もあります)。

不当な割り込みがかかるとINT_LOOPに飛びますので、デバッグ時は、ここにブレークポイントをセットしておくと便利です。
特にベクタテーブルの上から7番目のアドレス不正例外(ADDRERR)はデバッグ前のプログラムではよく発生します。上記例では他の例外や割り込みと分けていませんが、アドレス不正例外は別ルーチンに飛ぶようにしておくことを推奨します。
なお、アドレス不正例外は、16ビットのメモリリード/ライト時に奇数アドレスをアクセスした場合、32ビットのメモリリード/ライト時にワード境界(4倍数アドレス)以外をアクセスした場合に発生します。
S1C33コアでは、このようなメモリアクセスは禁止されています。

割り込みベクタの再定義　vector.h

```
;; Vector define
#define  RESET   BOOT
#define  INT12   int_16timer_u00
#define  INT15   int_16timer_c01
#define  INT18   int_16timer_u11
#define  INT23   int_16timer_c21
#define  INT27   int_16timer_c31
```

vector.sの中で実際に使用する例外/割り込みベクタラベルを別の名称で再定義し、適切なルーチンに飛ぶようにします。
上記例は、リセットベクタと16ビットタイマ割り込みベクタを実際の処理ルーチンのラベル名で再定義しています。

ブートルーチン　boot.s

```
;     file name : boot.s
;
;     Coptright (C) SEIKO EPSON CORP. 2002
;
;   BOOT:
;     boot program  set, SP, default data area pointer(%r15)
;     call _init_sys() and _init_lib().
;     And call main.

.set SP_INI,0x0800               ; sp is in end of 2KB internal RAM
.set DP_INI,0x0000               ; global pointer is 0x0

      .text
      .long BOOT                 ; BOOT VECTOR
BOOT:
      xld.w    %r15,SP_INI
      ld.w     %sp,%r15          ; set SP
      ld.w     %r15,DP_INI       ; set default data area pointer
      xcall    _init_bcu         ; Initialize BCU on boot time
      xcall    _init_sys         ; call _init_sys() in sys.c
      xcall    main              ; goto main
      xcall    _exit             ; in last, goto _exit

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; _init_bcu
;;   Type :    void
;;   Ret val : none
;;   Argument :void
;;   Function :Initialize BCU on boot time.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
      .global  _init_bcu
_init_bcu:
;; Set area 9-10 setting
;;    Area 9-10 setting ... Device size 16 bits, output disable delay 1.5, wait
control 2, burst ROM is not used in area 9-10, burst ROM burst read cycle wait con-
trol 0
      xld.w    %r11,BCU_A10_ADDR
      xld.w    %r10,BCU_BW_0|BCU_DRAH_NOT|BCU_DRAL_NOT|BCU_SZL_16|BCU_DFL_15|BCU_WTL_2
      ld.h     [%r11],%r10
```

このブートルーチン(BOOT)は、スタックポインタの初期化、デフォルトデータエリアポインタの初期化、BCUの初期化等を行ってからmainルーチンをコールします。

何らかの例外等が発生するとスタックを使用しますので、スタックポインタは他の処理の前にセットしてください。また、BCUもメモリ/デバイスをアクセスする前に必ず設定してください。

● Cでの記述

以下のプログラムは、gnu33\sample\drv33209\の中のサンプルです。

ベクタテーブル、ブートルーチン　drv33209\16timer\vector.c

```
/*****************************************************************************
 *                                                                           *
 *      Copyright (C) SEIKO EPSON CORP. 2002                                 *
 *                                                                           *
 *      File name: vector.c                                                  *
 *        This is vector and interrupt program with C.                       *
 *                                                                           *
 *****************************************************************************/

/* Prototype */
void boot(void);
void dummy(void);
extern void _init_bcu(void);
extern void _init_int(void);
extern void _init_sys(void);
extern void _exit(void);
extern void int_16timer_c0(void);
extern void int_16timer_u1(void);
extern void int_16timer_c2(void);
extern void int_16timer_u3(void);

/* vector table */
const unsigned long vector[] = {
       (unsigned long)boot,                 // 0   0
       0,                                   // 4   1
       0,                                   // 8   2
       0,                                   // 12  3
       (unsigned long)dummy,                // 16  4
       0,                                   // 20  5
       (unsigned long)dummy,                // 24  6
       (unsigned long)dummy,                // 28  7
       0,                                   // 32  8
       0,                                   // 36  9
       0,                                   // 40  10
       0,                                   // 44  11
       (unsigned long)dummy,                // 48  12
       (unsigned long)dummy,                // 52  13
                 |
          (56  14 – 120  30)
                 |
       (unsigned long)int_16timer_c0,       // 124 31
       (unsigned long)dummy,                // 128 32
       (unsigned long)dummy,                // 132 33
       (unsigned long)int_16timer_u1,       // 136 34
       (unsigned long)dummy,                // 140 35
       (unsigned long)dummy,                // 144 36
       (unsigned long)dummy,                // 148 37
       (unsigned long)dummy,                // 152 38
       (unsigned long)int_16timer_c2,       // 156 39
       (unsigned long)dummy,                // 160 40
       (unsigned long)dummy,                // 164 41
       (unsigned long)int_16timer_u3,       // 168 42
       (unsigned long)dummy,                // 172 43
                 |
          (176  44 – 268  67)
                 |
       (unsigned long)dummy,                // 272 68
       (unsigned long)dummy,                // 276 69
       (unsigned long)dummy,                // 280 70
       (unsigned long)dummy                 // 284 71
};
```

```
/************************************************************************
 * boot
 *   Type :     void
 *   Ret val :  none
 *   Argument : void
 *   Function : Boot program.
 ************************************************************************/
void boot(void)
{
       asm("xld.w %r15,0x2000");   // Set SP in end of 8KB internal RAM
       asm("ld.w  %sp,%r15");
       asm("ld.w  %r15,0b10000");  // Set PSR to interrupt enable
       asm("ld.w  %psr,%r15");
       asm("xld.w %r15,0x600000"); // Set DPR is 0x0
       _init_bcu();                // Initialize BCU on boot time
       _init_int();                // Initialize interrupt controller
       _init_sys();                // Initialize for sys.c
       main();                     // Call main
       _exit();                    // In last, go to exit in sys.c to use simulated I/O
}

/************************************************************************
 * dummy
 *   Type :     void
 *   Ret val :  none
 *   Argument : void
 *   Function : Dummy interrupt program.
 ************************************************************************/
void dummy(void)
{
INT_LOOP:
       goto   INT_LOOP;
       asm("reti");
}
```

このファイルにはベクタテーブルとブートルーチンが含まれています。

ベクタテーブルはconst型の32ビット配列として定義しています。このようにして、32ビットの飛び先アドレスがROM上に格納できます。
各ベクタのコメント(//x y)は、テーブル先頭からのオフセットアドレス(x)とベクタ番号(y)を示す10進数です。
この例では、外部参照定義した割り込み処理関数の先頭アドレスを直接記述しています。
使用しない割り込み等はダミールーチンに飛ぶようにしています。

ブートルーチンは、前述のアセンブラの例と同じ内容です。SPやPSRはasm( )命令で初期化します。

ダミーの例外/割り込み処理ルーチンのreti命令も、asm( )命令で記述しています。

2.2 割り込み処理ルーチン

● 割り込み処理関数

S5U1C33001Cでは、Cソース内で__attribute__ ((interrupt_handler))による関数のプロトタイプ宣言を行うことによって割り込み処理関数に設定できます。
また、割り込み関数内でasm("reti");を使用している部分も、asm文を削除して、__attribute__ ((interrupt_handler))を使用したプロトタイプ宣言に置き代えることができます。

● 割り込み処理関数の宣言

割り込み処理関数は次の形式でプロトタイプ宣言します。

<型><関数名> __attribute__ ((interrupt_handler));

割り込み処理例 gnu33\sample\int_c\int.c

```
// int.c  1998.1.7
// vector and interrupt program with C

extern volatile int int_num;

// functions that jump from vector table

extern void boot()__attribute__ ((interrupt_handler));
void div0() __attribute__ ((interrupt_handler));
void unalign() __attribute__ ((interrupt_handler));
void nmi() __attribute__ ((interrupt_handler));
void softint0() __attribute__ ((interrupt_handler));
void softint1() __attribute__ ((interrupt_handler));
void softint2() __attribute__ ((interrupt_handler));
void softint3() __attribute__ ((interrupt_handler));
void hardint0() __attribute__ ((interrupt_handler));
void hardint1() __attribute__ ((interrupt_handler));
void hardint2() __attribute__ ((interrupt_handler));
void hardint3() __attribute__ ((interrupt_handler));
void hardint4() __attribute__ ((interrupt_handler));
void hardint5() __attribute__ ((interrupt_handler));
void hardint6() __attribute__ ((interrupt_handler));
void hardint7() __attribute__ ((interrupt_handler));
void hardint8() __attribute__ ((interrupt_handler));
void hardint9() __attribute__ ((interrupt_handler));

// vector table

const unsigned long vector[] = {
       (unsigned long)boot,         // 0   0
       0,                           // 4   1
       0,                           // 8   2
       0,                           // 12  3
       (unsigned long)div0,         // 16  4
       0,                           // 20  5
       (unsigned long)unalign,      // 24  6
       (unsigned long)nmi,          // 28  7
       0,                           // 32  8
       0,                           // 36  9
       0,                           // 40  10
       0,                           // 44  11
       (unsigned long)softint0,     // 48  12
       (unsigned long)softint1,     // 52  13
       (unsigned long)softint2,     // 56  14
       (unsigned long)softint3,     // 60  15
       (unsigned long)hardint0,     // 64  16
       (unsigned long)hardint1,     // 68  17
       (unsigned long)hardint2,     // 72  18
       (unsigned long)hardint3,     // 76  19
       (unsigned long)hardint4,     // 80  20
       (unsigned long)hardint5,     // 84  21
       (unsigned long)hardint6,     // 88  22
       (unsigned long)hardint7,     // 92  23
```

```
        (unsigned long)hardint8,     // 96  24
        (unsigned long)hardint9      // 100 25
};

// interrupt routines

void div0()
    {
        int_num = 4;
    }

void unalign()
    {
        int_num = 6;
    }

void nmi()
    {
        int_num = 7;
    }

void softint0()
    {
        int_num = 12;
    }

void softint1()
    {
        int_num = 13;
    }

void softint2()
    {
        int_num = 14;
    }

void softint3()
    {
        int_num = 15;
    }

void hardint0()
    {
        int_num = 16;
    }

void hardint1()
    {
        int_num = 17;
    }

void hardint2()
    {
        int_num = 18;
    }

void hardint3()
    {
        int_num = 19;
    }

void hardint4()
    {
        int_num = 20;
    }

void hardint5()
    {
        int_num = 21;
    }

void hardint6()
```

```
    {
       int_num = 22;
    }

void hardint7()
    {
       int_num = 23;
    }

void hardint8()
    {
       int_num = 24;
    }

void hardint9()
    {
       int_num = 25;
    }
```

● ソフトウェア割り込み処理例

ソフトウェア割り込み処理例　gnu33\sample\int_c\main.c

```
// main.c 2002.2.14
// boot and main program with C

volatile int int_num;

void boot()
    {
LOOP:
       asm("xld.w     %r8,0x0800");         // set SP
       asm("ld.w      %sp,%r8");
       asm("xld.w     %r8,0b10000");        // set PSR to interrupt enable
       asm("ld.w      %psr,%r8");
       asm("ld.w      %r15,0x0000");        // set Default data area pointer
       main();                              // call main
       goto LOOP;
    }

int main()
    {
       int j,k;

       asm("int 1");                        // software interrupt 1

       for (j = 0 ; ; j++)
           {
              k = int_num;
           }
    }
```

asm("int 1");のルーチンは、割り込みベクタテーブルのソフトウェア割り込み1として定義しておきます。

```
(unsigned long)softint1,          // 52   13     ソフトウェア割り込み1の定義
```

この割り込みが発生するとプロトタイプ宣言した関数に飛び、割り込み処理を実行します。
上記例の場合、int_num = 13;を実行してmain();関数内に戻り、無限ループに入ります。

2.3　Cとアセンブラの混在法

引数、戻り値、レジスタ内容保護などのルールを守ることにより、Cとアセンブラルーチン間を自由に行き来することができます。

● Cからコールされるアセンブラルーチンの作成法

gnu33\UTILITY\lib_src\ansilib\string\src\strcpy.s

```
;************************************************
; strcpy
;        string copy from src to dest until 0 terminate
;
; arguments : %r6:dest addr, %r7:src addr (0 terminate string)
; return    : %r4:dest addr
;************************************************

         .section .text
         .align   1
         .global  strcpy
         .type    strcpy,@function

strcpy:
         ld.w     %r4, %r6          ; return dest add

strcpy_loop:
         ld.ub    %r10, [%r7]+      ; copy src 1 byte to dest
         ld.b     [%r6]+, %r10
         cmp      %r10, 0           ; continue until 0 terminate
         jrne     strcpy_loop
         ret
```

このルーチンは、Cルーチンから次のようにコールされます。
(gnu33\sample\ansilib\ansilib.cから抜粋)

```
                |
#include <string.h>
                |
int main()
    {
                |
       char *pchMem;         /* for malloc, strcpy */
                |
       strcpy(pchMem, "This is strcpy test");
    }
```

第1引数はR6レジスタ、第2引数はR7レジスタに入れて渡され、戻り値はR4レジスタに返されます。

このように
- 第1引数から第4引数はR6からR9レジスタに入れて渡されます。
- 特殊なケースや第5以降の引数はスタックに入れて渡されます(実際にコンパイルされたコードを参考にしてください)。
- 戻り値は、R4レジスタに入れて返されます。

というのが引数と戻り値に使用するレジスタのルールです。

また、Cからコールされるアセンブラルーチン内でのレジスタ使用に関する制限は次のとおりです。
- R0～R3を使用する場合は、使用する前に必ずpushn命令で内容を退避させ、リターン前にpopn命令で復帰させてください。
- R4とR5はリターン前に戻り値をセットするまでは自由に使用できます。
- R6～R9は引数を使い終わった後は自由に使用できます。リターン前に復帰させる必要はありません。
- R10～R15はシンボル参照用にアセンブラasおよびリンカldによって予約されています。できるだけ使用しないでください。
- 構造体を返す関数への引数渡し
  戻り値となる構造体データが8バイトに収まる場合、そのデータは戻り値格納用レジスタR4とR5に入れて返されます。この場合、通常は第1引数となる構造体へのポインタが関数に渡されることはありません。

たとえば、gnu33\UTILITY\lib_src\emulib\adddf3.sは倍精度浮動小数点の加算処理をします。このルーチンはレジスタをすべて使用するため、R0～R3レジスタの内容を保存し、リターン前に戻しています。

```
__adddf3:
        pushn   %r3             ; save register values
            |
        popn    %r3             ; restore register values
        ret
```

● Cの関数をコールするアセンブラルーチンの作成法

Cの関数は上記のルールに沿ってコンパイルされます。したがって、Cの関数をコールするアセンブラルーチンは以下の点に注意して作成してください。

引数、戻り値の受け渡しルール

- 第1引数から第4引数はR6からR9レジスタに入れて渡します。
- 戻り値は、R4レジスタで受け取ります。

リターン時のレジスタの状態

- R0からR3レジスタはコール時の内容を保持しています。
- R4からR15レジスタ、AHR、ALR、PSRは変更されている可能性があります。

2.4　アセンブル用ツールとファイル

ユーザが作成したアセンブリソースファイルは、次の3つのソフトウェアツールを使用してアセンブルされます。

1. Cコンパイラxgcc (-c -xassembler-with-cppオプションを指定)
2. プリプロセッサcpp
3. アセンブラas

* アセンブル自体はアセンブラのみでも実行可能ですが、プリプロセッサ 擬似命令を処理することはできません。

ユーザが作成したアセンブリソース例(.s)

```
; boot.s  2002.2.8
; boot program

#define SP_INI 0x0800         ; sp is in end of 2KB internal RAM          (1)
#define DP_INI 0x0000         ; default data area pointer %r15 is 0x0     (1)

       .text                                                              (2)
       .long  BOOT            ; BOOT VECTOR                               (2)
BOOT:
       xld.w  %r15,SP_INI                                                 (3)
       ld.w   %sp,%r15        ; set SP
       ld.w   %r15,DP_INI     ; set default data area pointer
       xcall  main            ; goto main                                 (3)
       xjp    BOOT            ; infinity loop                             (3)
```

(1)cppで処理される擬似命令です。
(2)asで処理される擬似命令です。
(3)asで処理される拡張命令です。

● プリプロセッサ命令

"#"で始まる命令はプリプロセッサの擬似命令で、マクロ命令や条件アセンブル命令、数値や文字列のシンボル定義など、アセンブラプログラムを分かりやすく作成するための付加機能を提供します。
これらの命令はcppで処理され、asでアセンブル可能な基本命令に展開されます。

プリプロセッサ擬似命令　gnu33\sample\asm\as_withcpp.s

```
/*********************************************************
       as instruction sample file with C preprocesser
       xgcc -c -xassembler-with-cpp as_withcpp.s
*********************************************************/

#include "test.h"

#define DATA1  0x1234                 // define DATA1
#define DATA2  (DATA1 + 2) << 1       // define DATA2

BAR:                                  /* start BAR function */
       xld.w  %r11,DATA1
       xld.w  %r10,DATA2
       sub    %r10,%r11
       xld.w  [BSS1],%r10
       ret

FOO:                                  // start FOO function
       add    %r6,%r7
       ld.w   %r4,%r6
       ret

       .section .bss
BSS1:
       .zero 4
```

● アセンブラ擬似命令

"."で始まるアセンブラ擬似命令は、主にセクションやROMに書き込むデータの定義に使用します。
アセンブラ擬似命令はasを通すまで処理されません。

アセンブラ擬似命令　gnu33\sample\asm\as_directive.s

```
; as_directive.s        2002.3.11
; sample source for as directives

      .set     RAM1,0x0          ; set absolute data

      .text                      ; start text section
      .global  BOOT              ; BOOT become global symbol
      .long    BOOT              ; 32bit data
BOOT:                            ; label in code section
      ld.w     %r15,0
      xld.w    %r1,[DATA1]
      xld.w    [%r15+RAM1],%r1
      xld.ub   %r2,[DATA1+8]
      xld.w    [BSS1],%r2
      jp       BOOT
      .word    0x0000            ; same with nop

      .section .data
      .align   2                 ; align to 4 byte boundary
DATA1:                           ; label in data section
      .long    0x12345678        ; 32bit data
      .word    0x1234,0x5678     ; 16bit data
      .byte    0x90              ; 8bit data
      .ascii   "abc"             ; string data
      .space   4,0               ; 4bytes 0

      .section .bss
      .align   2                 ; align to 4 byte boundary
      .global  BSS1
BSS1:
      .zero    4                 ; 4 byte global bss data area
LBSS1:
      .zero    4                 ; 4 byte local bss data area
```

● 主要なアセンブラ命令

アセンブラでプログラムする場合、以下の2つの命令の書き方について理解する必要があります。
CPUコア自体が持つ命令(基本命令)
asで展開されるマクロ命令(拡張命令)

それぞれの命令をすべて合わせるとかなりの数になります。
特にasの拡張命令がいろいろあります。

S1C33のプログラミングに慣れるまでは、以下に示す2種類の拡張命令とCPUコアの主要な基本命令を使用し、徐々に用途に応じた拡張命令を増やしていくことを推奨します。

2種類の拡張命令

```
xld.w   %r8,0x12345678  ; レジスタへの直値の代入
xcall   sub             ; ラベルへのcall
```

その他 よく使用する 基本命令

算術演算

```
add     %r1,%r2          ; sub, sbcも同様
add     %r3,3
adc     %r5,%r3

cmp     %r7,%r9
cmp     %r15,-1

mlt.h   %r9,%r8          ; unsignedのmltu.h, mltu.wもあり
mlt.w   %r1,%r2          ; divはサブルーチン形式にてサポート
```

論理演算

```
and     %r2,%r1          ; or, xorも同様
and     %r1,0b0111

not     %r2,%r1
not     %r1,-1
```

シフト

```
srl     %r10,5           ; sll, sra, sla, rr, rlも同様
srl     %r9,%r5
```

レジスタコピー

```
ld.b    %r2,%r3          ; ld.ub, ld.h, ld,uh, ld,wも同様

ld.w    %r8,%alr         ; sp, ahr, alr, psrも同様
ld.w    %sp,%r9
```

メモリアクセス

```
ld.b    %r9,[%r9]        ; ld.ub, ld.h, ld,uh, ld,wも同様
ld.b    %r15,[%r0]+

ld.b    [%r3],%r2        ; ld.h, ld.wも同様
ld.b    [%r4]+,%r0

btst    [%r9],0x1        ; bset, bclr, bnotも同様
```

分岐

```
jrgt    SYM              ; jrXX, jp, jrXX.d, jp.dも同様
```

リターン

```
ret
ret.d
```

割り込み

```
reti
int     3
```

拡張命令

```
ext     0x123
```

その他

```
pushn   %r15
popn    %r0
mac     %r12
nop
halt
slp
```

● 基本命令

基本命令はS1C33000の命令セットのことで、asでマシンコードにアセンブルされます。
コアCPUのニーモニックコードをそのまま記述します。アドレスを即値指定するオペランドには、定義済みのラベルを単独に、あるいはディスプレースメントやシンボルマスク、擬似オペランドと組み合わせて記述できます。

```
例: jrgt  LABEL              ; ラベル指定

    call  LABEL+0x10         ;ラベル+ディスプレースメント指定

    ext   LABEL@h            ; ラベル+シンボルマスク指定
    ext   LABEL@m
    ld.w  %r9,LABEL@l        ; = ld.w %r9,LABEL

    ext   doff_hi(FOO)       ; 擬似オペランド指定
    ext   doff_lo(FOO)
    ld.w  %r0,[%r15]         ; = ld.w %r0,[%r15+(FOO address - __dp)]
```

以下に基本命令の一覧を示します。太字の命令は基本命令でのみ記述可能で、その他の命令は拡張命令で記述可能です。

基本命令一覧　gnu33\sample\asm\as_inst.s

```
; as_inst.s    2002.2.8
; sample source for as instructions

; arithmetic operations
        add     %r1,%r2
        add     %r3,3
        add     %sp,0x123
        adc     %r5,%r3
        sub     %r1,%r2
        sub     %r3,3
        sub     %sp,0x123
        sbc     %r5,%r3
        cmp     %r7,%r9
        cmp     %r15,-1
        mlt.h   %r9,%r8
        mltu.h  %r7,%r4
        mlt.w   %r1,%r2
        mltu.w  %r5,%r1
        div0s   %r1
        div0u   %r2
        div2s   %r3
        div3s

; logical operations
        and     %r2,%r1
        and     %r1,0b0111
        or      %r2,%r1
        or      %r1,11
        xor     %r2,%r1
        xor     %r1,0x11
        not     %r2,%r1
        not     %r1,-1

; shift & rotation operations
        srl     %r10,5
        srl     %r9,%r5
        sll     %r10,5
        sll     %r9,%r5
        sra     %r10,5
        sra     %r9,%r5
        sla     %r10,5
        sla     %r9,%r5
        rr      %r10,5
        rr      %r9,%r5
        rl      %r10,5
        rl      %r9,%r5

; etc
        pushn   %r15
        popn    %r0
        mac     %r13
        nop
        halt
        slp
        scan0   %r1,%r2
        scan1   %r3,%r4
        swap    %r5,%r6
        mirror  %r7,%r7

; bit operations
        btst    [%r9],0x1
        bset    [%r0],7
        bclr    [%r15],0b1
        bnot    [%r10],5

; ext operations
        ext     0x123
        ext     SYM@ah
        ext     SYM+0x56@ah
        ext     SYM@al
        ext     SYM+0x56@al
        ext     SYM@h
        ext     SYM+0x56@h
        ext     SYM@m
        ext     SYM+0x56@m
        ext     SYM@rh
        ext     SYM@rm

; load operations
        ld.b    %r2,%r3
        ld.b    %r9,[%r9]
        ld.b    %r15,[%r0]+
        ld.b    %r6,[%sp+8]
        ld.b    [%r3],%r2
        ld.b    [%r4]+,%r0
        ld.b    [%sp+0x10],%r11
        ld.ub   %r2,%r3
        ld.ub   %r9,[%r9]
        ld.ub   %r15,[%r0]+
        ld.ub   %r6,[%sp+8]
        ld.h    %r2,%r3
        ld.h    %r9,[%r9]
        ld.h    %r15,[%r0]+
```

```
        ld.h    %r6,[%sp+8]
        ld.h    [%r3],%r2
        ld.h    [%r4]+,%r0
        ld.h    [%sp+0x10],%r11
        ld.uh   %r2,%r3
        ld.uh   %r9,[%r9]
        ld.uh   %r15,[%r0]+
        ld.uh   %r6,[%sp+8]
        ld.w    %r2,%r3
        ld.w    %r8,%alr
        ld.w    %sp,%r9
        ld.w    %r9,[%r9]
        ld.w    %r15,[%r0]+
        ld.w    %r6,[%sp+8]
        ld.w    [%r3],%r2
        ld.w    [%r4]+,%r0
        ld.w    [%sp+0x10],%r11
        ld.w    %r9,SYM@l

; branch operations
        jrgt    -1
        jrgt    SYM
        jrgt    SYM@rl
        jrgt.d  2
        jrgt.d  SYM
        jrgt.d  SYM@rl
        jrge    -1
        jrge    SYM
        jrge    SYM@rl
        jrge.d  2
        jrge.d  SYM
        jrge.d  SYM@rl
        jrlt    -1
        jrlt    SYM
        jrlt    SYM@rl
        jrlt.d  2
        jrlt.d  SYM
        jrlt.d  SYM@rl
        jrle    -1
        jrle    SYM
        jrle    SYM@rl
        jrle.d  2
        jrle.d  SYM
        jrle.d  SYM@rl
        jrugt   -1
        jrugt   SYM
        jrugt   SYM@rl
        jrugt.d 2
        jrugt.d SYM
        jrugt.d SYM@rl
        jruge   -1
        jruge   SYM
        jruge   SYM@rl
        jruge.d 2
        jruge.d SYM
        jruge.d SYM@rl
        jrult   -1
        jrult   SYM
        jrult   SYM@rl
        jrult.d 2
        jrult.d SYM
        jrult.d SYM@rl
        jrule   -1
        jrule   SYM
        jrule   SYM@rl
        jrule.d 2
        jrule.d SYM
        jrule.d SYM@rl
        jreq    -1
        jreq    SYM
        jreq    SYM@rl
        jreq.d  2
        jreq.d  SYM
        jreq.d  SYM@rl
        jrne    -1
        jrne    SYM
        jrne    SYM@rl
        jrne.d  2
        jrne.d  SYM
        jrne.d  SYM@rl
        call    -1
        call    SYM
        call    SYM@rl
        call    %r5
        call.d  2
        call.d  SYM
        call.d  SYM@rl
        call.d  %r8
        jp      -1
        jp      SYM
        jp      SYM@rl
        jp      %r5
        jp.d    2
        jp.d    SYM
        jp.d    SYM@rl
        jp.d    %r8
        ret
        ret.d
        reti
        retd
        int     3
        brk
```

● アセンブリソースレベルデバッグ

通常のCソースファイルレベルでデバッグする場合、Cコンパイラxgccの-gstabsオプションを指定すると、出力されるelfファイルに.stab擬似命令によるデバッグ情報が付加され、ソースレベルデバッグが可能になります。

ここでは、アセンブリソースレベルのデバッグを行うための方法を説明します。

アセンブラasの--gstabsオプションを使用する方法

アセンブラasの起動時に--gstabsオプションを指定すると、出力オブジェクトファイルに行デバッグ情報が付加されます。付加された行デバッグ情報は、objdumpツールの-gオプションで確認することができます。

例:

```
(sample.s)
       1:     .text
       2:     add    %r0,1
       3:     ext    0x100
       4:     ld.w   %r1,0x10

>as -o sample.o sample.s --gstabs
>ld -o sample.elf sample.o
>objdump -g sample.elf

sample.elf:     file format elf32-c33

sample.s:
/* file sample.s line 2 addr 0xc00000 */
/* file sample.s line 3 addr 0xc00002 */
/* file sample.s line 4 addr 0xc00004 */
```

プリプロセッサcpp実行後、アセンブラasの--gstabsオプションを使用する方法

"#include <filename>"などのプリプロセッサ命令を使用したアセンブリソースは、アセンブラasの前にプリプロセッサcppを通す必要があります。

例: cpp展開例(sample.ps)

```
(inc1.h)
       1:     .set  DATA1, 0x01
       2:     .set  DATA2, 0x02
       3:     .set  DATA3, 0x03
       4:     .set  DATA4, 0x04
       5:     .set  DATA5, 0x05
       6:     .set  DATA6, 0x06

(sample.s)
       1:     #include  "inc1.h"
       2:             .text
       3:             add    %r0,1
       4:             ext    0x100
       5:             ld.w   %r1,0x10

>cpp sample.s > sample.ps

(sample.ps)
       1:     # 1 "sample.s"
       2:     # 1 "inc1.h" 1
       3:     .set  DATA1, 0x01
       4:     .set  DATA2, 0x02
       5:     .set  DATA3, 0x03
       6:     .set  DATA4, 0x04
       7:     .set  DATA5, 0x05
       8:     .set  DATA6, 0x06
       9:     # 1 "sample.s" 2
       10:
       11:    .text
       12:    add    %r0,1
       13:    ext    0x100
       14:    ld.w   %r1,0x10
```

```
>as -o sample.o sample.ps --gstabs
>ld -o sample.elf sample.o
>objdump -g sample.elf

sample.elf:     file format elf32-c33

sample.s:
/* file sample.s line 3 addr 0xc00000 */
/* file sample.s line 4 addr 0xc00002 */
/* file sample.s line 5 addr 0xc00004 */
```

xgccのコマンドラインで-xassembler-with-cppと--gstabsオプションを使用する方法

```
>xgcc -c -xassembler-with-cpp -Wa,--gstabs sample.s
>ld -o sample.elf sample.o
>objdump -g sample.elf

sample.elf:     file format elf32-c33

sample.s:
/* file sample.s line 3 addr 0xc00000 */
/* file sample.s line 4 addr 0xc00002 */
/* file sample.s line 5 addr 0xc00004 */
```

前記例のプリプロセッサcppとアセンブラasの処理が、これらのオプション指定のみで行えます。

● makeファイル

ワークベンチgwb33で作成される標準的なmakeファイルを以下に示します。
このmakeファイルを、gwb33の[Make file editor]または汎用エディタで修正して使用することができます。

makeファイル名:　sample.mak
アセンブリソース: boot.s
Cソース:　　main.c, sys.c
の場合、次のmakeファイルが生成されます。

makeファイル　sample.mak

```
# make file made by GWB33

# make file made by gnu make

# macro definitions for target file

TARGET= sample

# macro definitions for tools & dir

TOOL_DIR = C:/GNU33
CC= $(TOOL_DIR)/xgcc
AS= $(TOOL_DIR)/xgcc
LD= $(TOOL_DIR)/ld
RM= $(TOOL_DIR)/rm
LIB_DIR= $(TOOL_DIR)/lib
SRC_DIR= .

# macro definitions for tool flags

CFLAGS= -B$(TOOL_DIR)/ -c -gstabs -O -mgda=0 -mdp=1 -mlong-calls -I$(TOOL_DIR)/
        include -fno-builtin
ASFLAGS= -B$(TOOL_DIR)/ -c -xassembler-with-cpp -Wa,--gstabs
LDFLAGS= -T $(TARGET).lds -Map $(TARGET).map -N

# macro definitions for object files

OBJS= boot.o ¥
      main.o ¥
      sys.o  ¥

OBJLDS=

# macro definitions for library files

LIBS= $(LIB_DIR)/libc.a $(LIB_DIR)/libgcc.a

# dependency list start

### src difinition start
SRC1_DIR= .
### src difinition end
```

```
$(TARGET).elf : $(OBJS) $(TARGET).mak $(TARGET).lds
	$(LD) $(LDFLAGS) -o $@ $(OBJS) $(OBJLDS) $(LIBS)

## boot.s
boot.o : $(SRC1_DIR)/boot.s
	$(AS) $(ASFLAGS) -o boot.o $(SRC1_DIR)/boot.s

## main.c
main.o : $(SRC1_DIR)/main.c
	$(CC) $(CFLAGS) $(SRC1_DIR)/main.c

## sys.c
sys.o : $(SRC1_DIR)/sys.c
	$(CC) $(CFLAGS) $(SRC1_DIR)/sys.c

# dependency list end

# clean files except source

clean:
	$(RM) -f $(OBJS) $(TARGET).elf $(TARGET).map
```

●ヘッダファイルのアセンブラとCでの共有

S5U1C33001Cではヘッダファイル(*.h)をアセンブリソースとCソースで共有することが可能です。
アセンブラ、Cソース共に、プリプロセッサ定義を使用する個所よりも前でプリプロセッサ命令#include "filename.h"を使用することで参照できます。プリプロセッサの機能を使用するため、アセンブリソースをアセンブルするときは必ずxgccを通し、 -xassembler-with-cppオプションを指定するようにしてください。as単体のみでアセンブルしようとした場合、エラーが発生します。
アセンブラ固有の機能を#define命令で定義し、それをCソース内で使用することはできません。また、C言語固有の機能をアセンブリソース内の#defineで使用することはできません。
悪い例1:

(header1.h)
```
#define REF_SYMBOL  .extern symbol          ← アセンブラ固有の機能です。
```

(source.c)
```
    REF_SYMBOL;                             ← 構文エラーが発生します。
    int main( void ) {
    }
```

悪い例2:

(header1.h)
```
#define SUCCESS_FLAG(x)  ( x>= 0 )          ← C固有の機能です。
```

(source.s)
```
    cmp SUCCESS_FLAG(%r0)                   ← 構文エラーが発生します。
```

アセンブリソース内で#if、#ifdef、#ifndefといったプリプロセッサに依存した分岐を行わないでください。デバッガ上でソースと実際のアセンブラ行がずれて表示されます。また、アセンブリソース内で#includeによりヘッダファイルを参照する場合、参照するヘッダファイル中で#includeを使用しないでください。この場合もソースと実際のコードがずれて表示されます。
悪い例:

(header1.h)
```
#include "header2.h"
        |
```

(src.s)
```
#include "header1.h"
        |
```

ネストしてヘッダが参照されています。

良い例:

(header1.h)
```
/* #include "header2.h" */
        |
```

(src.s)
```
#include "header2.h"
#include "header1.h"
        |
```

以下にヘッダを共有した使用例を示します。\gnu33\sample\sharehにソースが入っています。ソースの詳細はそちらを参考してください。
この例ではタスク登録をする際に共通ヘッダで定義されているTASK_Xをアセンブリソース内で使用しています。ここではアセンブラとCソースで共に使用できる定数のマクロ置換機能を使用しています。

次のヘッダファイルは、アセンブラとCソースで共通に使用できます。
#defineプリプロセッサ命令を使用して、プリプロセッサが実行される時点で決定する値のみを定義しています。外部のファイル、もしくはリンクによって決定する値を#defineにより定義する場合は、Cもしくはアセンブラそれぞれの方法に従って記述する必要があります。

(shareh.h)

```
/* Macros in this header are used in both C and assembler sources. */

#define TASK_MASK               0x00ff
#define TASK_DISNABLE           0x0000
#define TASK_ENABLE             0x0001
#define TASK_DEFAULT            TASK_ENABLE

#define TASK_USINGMASK          0xff00
#define TASK_N( x )             ( 0x0100 << x )
#define TASK_0                  TASK_N( 0 )
#define TASK_1                  TASK_N( 1 )
#define TASK_2                  TASK_N( 2 )
#define TASK_3                  TASK_N( 3 )
#define TASK_4                  TASK_N( 4 )
#define TASK_5                  TASK_N( 5 )
#define TASK_6                  TASK_N( 6 )
#define TASK_7                  TASK_N( 7 )
#define MAX_TASK_NUM            8

#define TASK_0_7                ( TASK_0 + TASK_1 + TASK_2 + TASK_3 ¥
                                  TASK_4 + TASK_5 + TASK_6 + TASK_7 )
#define TASK_NOT0               TASK_0_7 - TASK_0
#define TASK_0_3                TASK_0 * 0xf

/*****************************************************************/
/* when defined, application go to start routine if exit.        */
/*****************************************************************/
#define GOSTART_IFAPPEND

/*****************************************************************/
/* if you change max task number, enable below line.             */
/*****************************************************************/
//#define OVERRIDE_MAXTASK    8
/* new MAX_TASK_NUM ( max task number is 8. ) */
#define OVERRIDE_MAX_TASK_NUM  8

/*****************************************************************/
/* if you don't use any tasks, enable below line.                */
/*****************************************************************/
//#define NO_TASK

/*****************************************************************/
/* psr default setting                                           */
/*****************************************************************/
/* This configuration psr is set at the top of program. */
/* if you don't permit interruption, comment here.*/
#define IE_BIT  0x10
/* set default interrput-level( In many case, below should be 0). */
#define DEFAULT_INTERRUPT_LEBEL      0

#define DEFAULT_PSR     ( IE_BIT | ( DEFAULT_INTERRUPT_LEBEL << 8 )   )

/* dummy definition */
#define MACRO_DAMMY
#undef MACRO_DAMMY
```

次のファイルではアセンブラでのみ参照可能な定義を行っています。アセンブラでプリプロセッサ定義を使用する場合、文字列や数字の定義による置換が有効です。

(onlys.s)

```
#define GETTASK_H            ext doff_hi( ulTaskManage )
#define GETTASK_L            ext doff_hi( ulTaskManage )
#define GETTASK( register )  ld.w register,[%r15]
```

```
#define PUTTASK_H            ext doff_hi( ulTaskManage )
#define PUTTASK_L            ext doff_hi( ulTaskManage )
#define PUTTASK( register )  ld.w [%r15],register
```

以下はアセンブリソース内でプリプロセッサ定義を使用した例です。

(asm.s)

```
/* Nesting header files is not supported in assembler sources. */
#include "shareh.h"
#include "onlys.h"
          |
        xld.w     %r0,DEFAULT_PSR              ← プリプロセッサ定義を使用
                            /* Set IE, IL. See shareh.h About DEFAULT_PSR. */

init_task:
        xld.w     %r6,TASK_0                   ← プリプロセッサ定義を使用
        xld.w     %r7,DO_INT
        call      add_task
        ret
          |
SOFT_INT:
        pushn     %r14
        /* delete old task0 */
        xld.w     %r6,TASK_0                   ← プリプロセッサ定義を使用
        call      delete_task
        /* add dummy task0 */
        xld.w     %r6,TASK_0                   ← プリプロセッサ定義を使用
        xld.w     %r7,vfnSetDummy
        call      add_task
        /* add exit task */
        xld.w     %r6,TASK_7                   ← プリプロセッサ定義を使用
        xld.w     %r7,EXIT_MAIN
        call      add_task
        popn      %r14
        reti
          |
/* exit all program */
/* --- no arguments --- */
EXIT_MAIN:
        /* read ulTaskManage */
        GETTASK_H                              ← アセンブラ固有の
        GETTASK_L                              ← 定義を使用
        GETTASK( %r4 )                         ←
        /* clear TASK_MASK part, and set TASK_DISABLE */
        xld.w     %r5,TASK_MASK
        not       %r5,%r5
        and       %r4,%r5
        /* write ulTaskManage */
        PUTTASK_H                              ← アセンブラ固有の
        PUTTASK_L                              ← 定義を使用
        PUTTASK( %r4 )                         ←
        ret
```

以下はCソースファイルでのみ使用可能なヘッダファイルの例です。ここではプリプロセッサ定義とdo { ... } while(1)というCの構文を使用することで、擬似的にローカルな変数を作成しています。また、ファイルの先頭でヘッダファイルが二重に参照されてもエラーが出ないように細工してあります。ヘッダファイルの相互関係が複雑になる場合非常に有用です。

(csrc.h)

```
#ifndef _ONLYC_H_                                    /* _ONLYC_H_ */

#define _ONLYC_H_

#include "shareh.h"

/* macro definition changing a task bit to a task number. */
/* This definition can be used in only C-source files. */
#define TASK_NUM(  task_bit, returned_tasknum )  ¥
```

```
            do  { ¥
                unsigned long ulBuf; ¥
                ¥
                ulBuf = task_bit; ¥
                returned_tasknum = 0; ¥
                while ( (ulBuf & 0x100) == 0x0 ) { ¥
                    returned_tasknum++; ¥
                    ulBuf >>= 1; ¥
                }; ¥
            } while ( 0 );

    /* #if, #ifdef or other branch prepro-instructions are
    *    used in only C sources. */
    #if defined( OVERRIDE_MAX_TASK_NUM )
    #undef MAX_TASK_NUM
    #define MAX_TASK_NUM       OVERRIDE_MAX_TASK_NUM
    #endif

    #ifdef NO_TASK
    /* disable all tasks. */
    #undef  TASK_DEFAULT
    #define TASK_DEFAULT    TASK_DISNABLE
    #endif

    #endif                                                  /* _ONLYC_H_ */
```

以下はプリプロセッサ定義を使用したCソースファイルの例です。

(csrc.c)

```
    #include "onlyc.h"
              |
    int main_loop( void )
        {

        while ( 1 ) {
            volatile unsigned long *ulpTask = &ulTaskManage;
            unsigned long          ulTaskBuf;
            unsigned int iTaskBit;
            int iTaskNum ;

            if ( ( *ulpTask & TASK_MASK ) == TASK_ENABLE ) {
                ulTaskBuf = *ulpTask;
                for ( iTaskNum = 0, iTaskBit = TASK_0 ;
                    iTaskNum < MAX_TASK_NUM ; iTaskNum++, iTaskBit <<= 1 ) {
                    if ( (ulTaskBuf & iTaskBit) != 0 ) {
                        /* Execute task if registered. */
                        if ( fnpTask[ iTaskNum ] != (void *)0x0 ) {
                            fnpTask[ iTaskNum ]();
                        }
                    }
                }
            } else {
                /* illeagal interruption occured */
                break;
            }
        }

    #if !defined( GOSTART_IFAPPEND )
    ENDLESS_LOOP:
        goto ENDLESS_LOOP
    #else
        return 0;
    #endif

    }

    void add_task( unsigned long ulTaskFlag, void *fpFunc )
    {

        int iNum = 0;

        TASK_NUM(  ulTaskFlag, iNum )

        fnpTask[ iNum ] = fpFunc;
        ulTaskManage = ulTaskManage | ulTaskFlag ;
    }
              |
```

2.5 データエリアとデータエリアポインタ

S5U1C33001Cには、データエリアと呼ばれる領域のベースアドレスを保持するデータエリアポインタ(CPUレジスタが割り当てられます)が複数用意されています。データエリア内に配置したグローバル変数、初期化付変数、定数データは、データエリアポインタを使用することで、メモリ全体をアクセスするのに比べ小さなオフセットアドレスで効率良く、高速にアクセスすることが可能となります。どのデータをどのデータエリアに配置するかについては、ソース内の定義とリンカスクリプトの設定によって指定することができます。
データエリアポインタを使用してデータエリアにアクセスするためには以下の操作が必要です。
1. プログラム先頭でデータエリアポインタを設定
2. プログラム内でデータエリアを指定して変数を配置
3. データエリアポインタに関するコンパイラオプションを設定
4. リンカスクリプト内でセクションを定義(必要な場合)
5. リンク時にデータエリアポインタ(データエリアのベースアドレス)をシンボルとして定義

2.5.1 データエリアの種類

S5U1C33001Cが対応しているデータエリアは以下の5つです。

・デフォルトデータエリア
・Gデータエリア
・Sデータエリア
・Tデータエリア
・Zデータエリア

各データエリアに配置されるセクション属性、データエリアポインタとして使用するCPUレジスタ、データポインタシンボルなどを表2.5.1.1に示します。

表2.5.1.1 データエリア

データエリア	配置セクションの属性			使用するレジスタ	アクセス用擬似命令	データエリアポインタシンボル	32ビットアドレス
	変数	初期化付変数	定数				
デフォルト	.bss	.data	.rodata	(STD) R15 (ADV) DP *1	(STD) doff_hi (STD) doff_lo (ADV) dpoff_h (ADV) dpoff_m (ADV) dpoff_l	__dp	**ADVに限り可**
G	.gbss	.gdata	.rodata *3	R15–R12 *2	goff_lo	__gdp	**不可**
Z	.zbss	.zdata	.rozdata	R14	zoff_hi zoff_lo	__zdp	**不可**
T	.tbss	.tdata	.rotdata	R13	toff_hi toff_lo	__tdp	**不可**
S	.sbss	.sdata	.rosdata	R12	soff_hi soff_lo	__sdp	**不可**

***1: コンパイル時に -mc33advオプション(アドバンスドマクロモードによるコンパイル)を使用すると デフォルトデータエリアポインタのレジスタ割り当てが変更されます。**
***2: Gデータエリアが使用するレジスタはコンパイラオプション-mgdp=XX(XX: dp, sdp, tdp, zdp)により変更可能です。**
***3: Gデータエリアの定数データはすべて.rodataセクションに配置されます。**

2.5.2 セクション

1つのデータエリアは複数のセクションにより構成されます。各セクションは独自の属性を持ちます。

・.text属性
プログラムが格納されます。この属性はデータエリアには属しません。
ROM上に実体を持ちます。RAMに転送して実行することも可能です。

・.bss, .gbss, .zbss, .tbss, .sbss属性
グローバル変数およびstatic宣言した変数が格納されます。RAM上に配置され、ROM上には実体を持ちません。
.bss(デフォルトデータエリアに配置)
.gbss(Gデータエリアに配置)
.zbss(Zデータエリアに配置)
.tbss(Tデータエリアに配置)
.sbss(Sデータエリアに配置)

・.data, .gdata, .zdata, .tdata, .sdata属性
初期化付変数が格納されます。ROM上に実体(初期値)を持ち、RAM上に転送することで初期化付変数として使用できます。配置されるデータエリアは.bss～.sbssと同様です。

・.rodata, .rozdata, .rotdata, .rosdata属性
定数が格納されます。ROM上に実体を持ちます。配置されるデータエリアは.bss～.sbssと同様です。ただし、Gデータエリアに配置指定した定数は、.rodataセクションに含まれ、デフォルトデータエリアに配置されます。

・その他の属性
任意の属性を設定することができます。

各セクションの属性は、リンカスクリプトファイル(.lds)内で指定します。
例: .bss属性を持つ.bssという名称のセクション定義

```
.bss 0x00000000 : {          ← この.bssは定義するセクションの名称
        __START_bss = . ;
        *(.bss) ;            ← .bss属性の指定
        __END_bss = . ;
}
```

2.5.3 データエリアポインタ

xgccコンパイラは、グローバルシンボル(変数、初期化付データ、定数)のアドレスを次のようにして、シンボルにアクセスするコードを生成します。

シンボルアドレス = データエリアポインタ値 + シンボルの値(オフセット値)

データエリアポインタを使用することにより、メモリにアクセスするたびに32ビットアドレスを指定する必要がなくなります。データエリアポインタからのオフセット値(26ビットもしくは13ビット)のみでアクセス可能なため、メモリアクセスの命令数が減り、アクセス時間を短縮できます。
たとえばCソースの

```
a = 1;
```

は以下のアセンブラコードとなります。

```
xld.w   %r4,1                (1)
ext     doff_hi( a )         (2-1)
ext     doff_lo( a )         (2-2)
ld.w    [ %r15 ], %r4        (2-3)
```

このアセンブラコードは次の操作を行っています。
(1)　　変数aに書き込む値をレジスタR4に作成します。
(2-1～2-3) 変数aのアドレス(R15 + シンボルオフセット値)に(1)でR4に設定した値を書き込みます。
ここではR15がデータエリアポインタの役割を担い(R15はデフォルトデータエリアポインタとして使用されます)、"シンボルのオフセット値"は(2-1)と(2-2)の2つの拡張命令でデータエリアポインタに加算されます。

リンカは(2-1)と(2-2)のシンボルを解決する際、(a - __dp)によりaのオフセットアドレスを取得します。__dpはデフォルトデータエリアポインタのシンボルで、通常その値(アドレス)はリンカスクリプトファイルで指定します(プログラムでR15にも同じ値を設定)。
(2-1)のコードには得られたオフセットアドレスのビット25～13が、(2-2)にはビット12～0が適用されます。

上の例では、シンボルのオフセット値を加算するために2つのext命令を使用していますが、データエリアポインタとシンボルの距離が近い場合(8Kバイト以下)、(2-1)は不要で命令数を1つ減らすことができます。

上記はデフォルトデータエリアポインタ__dp(R15)を使用した例ですが、他のデータエリアポインタのシンボルと使用するCPUレジスタは次のとおりです。

	シンボル	レジスタ	
デフォルトデータエリアポインタ:	**__dp**	**R15**	
Gデータエリアポインタ:	**__gdp**	**R15～R12**	**(他のデータエリアの1つと兼用)**
Zデータエリアポインタ:	**__zdp**	**R14**	
Tデータエリアポインタ:	**__tdp**	**R13**	
Sデータエリアポインタ:	**__sdp**	**R12**	

データエリアポインタはプログラムのブート時に上記のレジスタにセットしておく必要があります。これらのレジスタはコンパイラにより変更されることはありません(使用しないデータエリアポインタのレジスタは、コンパイラオプション-mdpにより、スクラッチレジスタとして使用することができます)。
例: スタンダードマクロモデルで、デフォルト、G、Z、Tデータエリアを使用する場合

```
/*    boot.s    */

        .global BOOT
        .align 2
        .text
BOOT:
        xld.w       %r4, 0x800
        ld.w        %sp, %r4

        xld.w       %r15, __dp        /* set default data area pointer */
        xld.w       %r14, __zdp       /* set default z area pointer    */
        xld.w       %r13, __tdp       /* set default t area pointer    */
          :
```

データエリアポインタシンボルの値は以下のいずれかの方法でリンカに設定します。

1. リンカのコマンドラインでデータエリアポインタを設定
   例: -defsymオプションで__dp、__gdp、__zdpを設定

```
DOS>ld.exe -Map test.map -o test.elf boot.o -defsym __dp=0x0 -defsym __gdp=0x0
    -defsym __zdp=0x10000 -N
```

2. リンカスクリプトファイル内でデータエリアポインタを指定
   例: リンカスクリプトファイル内で__dp、__gdp、__zdpを設定

```
DOS>ld.exe -T test.lds -Map test.map -o test.elf boot.o
```

(test.lds内のデータエリアポインタ指定部)

```
                    :
        SECTIONS
        {
            /* data pointer symbol By GWB33 */
            __dp = 0x00000000;
            __gdp = 0x00000000;
            __zdp = 0x00010000;

            /* section information By GWB33 */
            . = 0x0;
                    :
```

※ ワークベンチgwb33のリンカスクリプトエディタを使用することでもデータエリアポインタを設定可能です。

__dpは通常使用しますので、必ず定義してください。

2.5.4 コンパイラオプションの指定

データエリアを使用するためには、S5U1C33001C固有のコンパイラオプションを適切に設定する必要があります。
データエリアに関連するコンパイラオプションは以下のとおりです。

-mdp, -mgda, -mgdp, -mezda, -metda, -mesda

-mdpオプションは、使用するデータエリアを指定します。
-mdp=1　デフォルトデータエリアのみを使用
-mdp=2　デフォルトおよびZデータエリアを使用
-mdp=3　デフォルト、ZおよびTデータエリアを使用
-mdp=4　デフォルト、Z、TおよびSデータエリアを使用

-mdpで指定可能な組み合わせは上記の4種類のみです。Zデータエリアを使用せずにTとSデータエリアを使用するといった指定はできません。たとえば、Sデータエリアを使用する場合は、ZおよびTエリアを使用する必要があります。
-mdp=4を指定してデフォルトおよびZデータエリアのみを使用するといった使い方は可能です。この場合、TおよびSデータエリアにはプログラムでデータを配置しないようにします。ただし、途中のデータエリアを飛ばさずに、Z、T、Sの順に頭から使用するようにしてください。

-mgdaオプションはGデータエリアに配置するデータサイズを指定します。たとえば、4バイト以下の変数をGデータエリアに配置するには-mgda=4と指定します。-mgda=0を指定すると、Gデータエリアは使用されません。Gデータエリアは他のデータエリアの先頭部(8KB)に置かれます。-mgdpはそのデータエリアを指定するオプションです。たとえば、-mgdp=zdpと指定するとGデータエリアはZデータエリアの先頭に置かれ、データエリアポインタもZデータエリアと同じR14を使用します。デフォルトは-mgdp=dpでデフォルトデータエリアの先頭に置かれるようになっています。Gデータエリアについては、"2.5.7 Gデータエリア"を参照してください。

-mezda、-metda、-mesdaはそれぞれZ、T、Sデータエリアのサイズを64MBに拡張するためのオプションです。未指定時はそれぞれ8Kバイトで、2命令(ext+アクセス命令)でアクセスするようになっています。オプションを指定すると、データのアクセスは3命令(ext+ext+アクセス命令)に展開されます。8Kバイトを超えるメモリをデータポインタを使用してアクセスしたい場合に使用してください。なお、デフォルトデータエリアポインタは64MBまでのアクセスに固定されており、8KBに変更することはできません。

これらのコンパイラオプションは、使用するすべてのソースファイルで同じ指定を行う必要があります。指定が異なる場合、レジスタの破壊や誤ったメモリアクセスが発生する可能性やリンク時にエラーが発生する可能性があります。
特にライブラリを作成する場合は、できるだけデフォルトデータエリアのみを使用し、以下のオプションを指定することを推奨します。
-mdp=4, -mgda=0

R15～R12がすべてデータエリアポインタ用に確保され(コンパイラ用のスクラッチレジスタが減るため)、性能の観点からは好ましくありませんが、これらのレジスタが破壊されないため、他のデータエリアを使用するモジュールにもリンク可能なライブラリとなります。
S5U1C33001Cが提供するANSI libraryを使用する場合、使用環境に合わせた設定で再コンパイルすることでパフォーマンスの改善が見込めます。

-mdpオプションの補足説明

表2.5.4.1 -mdpオプションにより利用されるレジスタ

レジスタ	-mdp=1	-mdp=2	-mdp=3	-mdp=4	-mdp=5	-mdp=6
R15	__dp	__dp	__dp	__dp	__dp	__dp
R14	スクラッチレジスタ	__zdp	__zdp	__zdp	__zdp	__zdp
R13	スクラッチレジスタ	スクラッチレジスタ	__tdp	__tdp	__tdp	__tdp
R12	スクラッチレジスタ	スクラッチレジスタ	スクラッチレジスタ	__sdp	__sdp	__sdp
R11	スクラッチレジスタ	スクラッチレジスタ	スクラッチレジスタ	スクラッチレジスタ	保護レジスタ*	保護レジスタ*
R10	スクラッチレジスタ	スクラッチレジスタ	スクラッチレジスタ	スクラッチレジスタ	スクラッチレジスタ	保護レジスタ*

* xgccはこのレジスタの代わりにスタックを使用

-mdpオプションの有効数字が1～4となっていますが、実際には-mdp=5および-mdp=6としてもコンパイラは受け付け、エラーにはなりません。ただし、-mdp=5または-mdp=6を指定した場合は、指定内容がデータアエリアポインタ数の意味ではなくなります。データポインタとして使用するレジスタはR15からR12までの4本が最大のため、-mdp=4以上はデータポインタ本数は変わりません。-mdp=4の場合には、R11とR10はスクラッチレジスタとしてコンパイラが使用します。しかし、-mdp=5ではR11、-mdp=6ではR11とR10がコンパイラが使用しない、いわゆる保護レジスタになります。これらのレジスタはアセンブラからは値を保存する必要なく自由に使用できます。-mdp=5、6を指定する場合に注意すべき点として、通常R10、R11に割り当てられるレジスタ変数がスタックに割り当てられてしまい、その結果、実行速度が遅くなることがあります。このようなアセンブラ専用レジスタは、将来コンパイラのチューニングなどで効果を発揮することがあると思いますので、このまま残してあります。
通常は、1～4の範囲内で設定することをお勧めします。

2.5.5 データエリアへのデータの配置方法

特定のデータエリアに変数などを配置するには以下の方法でデータを定義します。

<データの型> <シンボル> __attribute__((*X*da));

(*X*da= zda, tda, sda)

例:

```
char c_default;                    // put to default
char c_z __attribute__((zda));     // put to Z
char c_t __attribute__((tda));     // put to T
char c_s __attribute__((sda));     // put to S
```

※この例では-mgda=0(Gデータエリアを使用しない)としています。

初期値を持たないグローバル変数は.*X*bss 、初期化付変数は.*X*data、定数は.ro*X*dataに配置されます。
データエリアの宣言なしに定義されたデータはデフォルトデータエリアに配置されます。

Gデータエリアに配置する場合、コンパイラオプション-mgdaで配置するデータサイズを指定できます。
-mgdaオプションで指定したサイズ以下のデータが Gデータエリアに配置されます。
例: -mgda=2を指定した場合

```
char c_data;                       // put to G
unsigned short us_data;            // put to G
long us_data;                      // put to default
```

※この例では-mgdp=dpとしています。

定数は-mgda、-mgdpオプションの指定にかかわらず、デフォルトデータエリア(.rodataセクション)に配置されます。
-mgdaオプションによりGデータエリアへの配置の対象となる変数が__attribute__((*X*da))宣言を伴っていた場合、*X*daエリアへ優先的に配置されます。

xgccコンパイラでは、extern宣言により他のソースの変数や定数などを参照する際には注意が必要です。
extern宣言が参照先の定義内容と同じ内容ではなかった場合、その変数は参照したい変数とは異なるデータエリアにある別の変数として扱われてしまいます。externにより変数を参照する場合、そのデータ型やデータエリアの宣言は元の変数の宣言と全く同じになるようにしてください。
例1: データエリア宣言の不備

file1.c:

```
char        c_sdata __attribute__((sda));
```

file2.c:

```
extern char c_sdata;
```

この場合、file2.cでは変数c_sdataがデフォルトもしくはGデータエリアの変数と見なされるため、file1.cで定義した変数にアクセスできません。file2.cでも__attribute__((sda))を宣言する必要があります。

例2: データサイズ指定の不備（コンパイラオプション-mgda=4指定時）

file1.c:

```
char        c_data[ 0x100 ];
```

file2.c:

```
extern char c_data[];
```

file1.cの変数c_dataはデフォルトデータエリアの変数ですが、file2.cでは4バイト以下でGデータエリアにある変数と見なされます。file2.cでも同じ変数宣言を行う必要があります。正しいextern宣言は次のようになります。

file1.c:

```
char        c_data[ 0x100 ];
```

file2.c:

```
extern char c_data[ 0x100 ];
```

正しいデータエリアにアクセスするためには、extern宣言の内容は必ず参照先の変数と同じにしてください。

2.5.6 データエリアポインタの設定方法

●基本設定

通常、データエリアポインタはデータエリア内の最下位アドレスに設定します。この場合、データエリア内の各セクションはデータエリアポインタ値から64Mバイト以内に収まる必要があります。ただし、Z、T、Sデータエリアを64Mバイトまで使用するためには、それぞれ-mezda、-metda、-mesdaコンパイラオプションによる指定が必要です。
アドバンスドマクロ対応のCPUを使用する場合に限り、32ビット空間すべてをデフォルトデータエリアとして使用することが可能です。

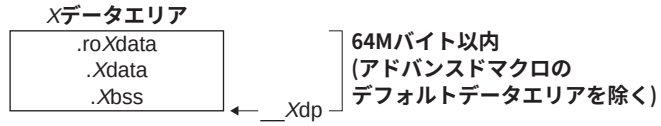


図2.5.6.1 データエリアの最大サイズ

以下に基本的な設定方法を説明します。.textセクションや.rodataセクションなど、読み込み専用で通常ROMに置く部分を「コード部」、.bssや.dataなど、通常RAMに置く部分を「データ部」とします。

「コード部」、「データ部」を共にデフォルトデータエリアポインタ（__dp）から64MB以内に配置

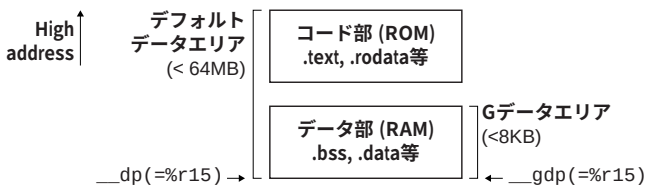


図2.5.6.2 配置例1

「コード部」よりも「データ部」の方が低いアドレスにある場合は、「データ部」の最下位アドレスを__dpに設定します。-mgdp=dpとしておくことで、Gデータエリアポインタを%r15で兼用できます。

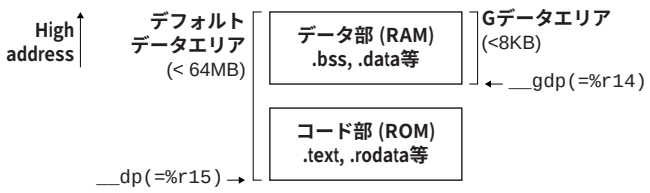
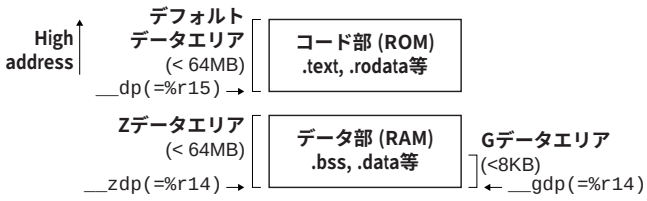


図2.5.6.3 配置例2

「コード部」の方が低いアドレスにある場合は、「コード部」の最下位アドレスを__dpに設定します。-mgdp=zdp指定によりGデータエリアポインタに%r14を割り当て、「データ部」の最下位アドレスを設定します。

「コード部」、「データ部」をデフォルトデータエリアポインタ(__dp)から64MB以内に配置できない場合
(たとえば、データ部がエリア17、コード部がエリア18に配置されるような場合です。)



「コード部」の最下位アドレスを__dpに設定します。-mgdp=zdp指定によりGデータエリアポインタに%r14を割り当て、「データ部」の最下位アドレスを設定します。

図2.5.6.4 配置例3

Gデータエリアは8KBのみのため、そこに入りきらない分については、アトリビュート指定(例: int __attribute__((sda)) foo;)により、S、T、Zデータエリアに配置します。
ただし、S、T、Zデータエリアがそれぞれ%r12、%r13、%r14を使用するため、Gデータエリアは使用しない設定にする必要があります(-mgda=0)。

.bss、.data、.rodataの3つのセクションはほとんどのプログラムで使用されます。できるだけ、これらのセクションがすべてデフォルトデータエリアポインタ__dpから 64Mバイト以内のデフォルトデータエリアに入るようにしてください。
通常よく使用される例を以下に示します。この例ではROMが0xc00000番地から、RAMが0x0番地から割り付けられているものとします。

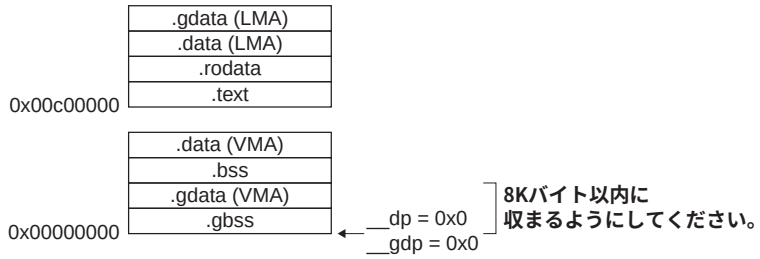


図2.5.6.5 セクション配置とデータエリアポインタの設定例1

ROMとRAMがそれぞれ1つずつのシステムでは、この例のような配置が標準的な構成です。高速にアクセスしたいデータがGデータエリアの8Kバイトに収まらない場合は、.dataセクション(VMA)や.gdata セクション(LMA)の後にZデータエリアを用意し、2命令展開によるアクセスを行うようにすることで対応します。高速処理には、RAM上でプログラムを実行することも有効です。

●データエリアポインタの追加

上記構成以外にメモリデバイスが存在する場合、S、T、またはZデータエリアのいずれかを設定します。
次の例は上記例の構成に加え、0x04000000番地から大容量 RAMが追加された場合を想定しています。

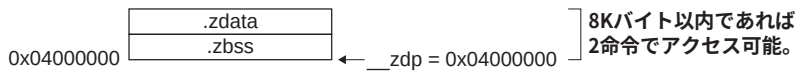
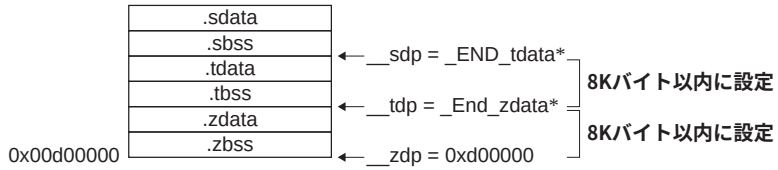


図2.5.6.6 セクション配置とデータエリアポインタの設定例2

この例ではZデータエリアを使用し、データエリアポインタ__zdpを0x04000000(デバイスの先頭アドレス)に設定します。ここで、__zdpを0x04000000よりも大きい値に設定すると、0x04000000から設定したアドレスまでが未使用領域として扱われます。データエリアポインタよりも低いアドレスにシンボルや変数を設定することも可能ですが、それらのシンボルをCソースからアクセスすることはできません。リンク時にワーニングが発生します。アセンブリソースからはアクセス可能です。

.zbss、.zdataセクションのサイズが 8Kバイト以上になる場合、コンパイル時に-mezdaオプションを指定することで領域サイズを64Mバイトに拡張できます。ただし、Zデータエリアへの個々のアクセスに3命令(8Kバイト時は2命令)使用されますので、コードサイズとアクセス速度からは望ましくありません。ここで、SまたはTデータエリアが使用可能な場合は、Zデータエリアに続けてそれらのデータエリアを設定することで、高速アクセス可能な領域を増やすことができます。



※ **gwb33ではシンボルを使用してデータエリアポインタの値を設定することはできません。gwb33を使用してリンカスクリプトファイルを作成している場合は、設定不可能なデータエリアポインタは設定せずにリンカスクリプトファイルを出力させた後、エディタ等でデータポインタの設定を追加してください。**

図2.5.6.7 セクション配置とデータエリアポインタの設定例3

この例の場合、ZおよびTデータエリアには2命令展開による高速アクセスが行えます。ただし、データエリアの使用数が増えるとコンパイラ用のスクラッチレジスタが減るため(この例ではCPUレジスタのR14～R12をスクラッチレジスタとして使用できません)、プログラム全体の速度低下を引き起こす可能性がありますので注意してください。

●定数のアクセス

以下の例のような定数はデフォルトデータエリアの.rodataセクションに入れられ、ROM上に置かれます。
例:

```
sprintf( szBuf, "Hello, World!!¥n" );
```

この文字配列(Hello, World!!\n)は.rodataセクションに配置されます。

特に定数を定義していない場合でも、このように.rodataセクションが存在する可能性があります。デフォルトデータエリアは3命令でアクセスされますので、頻繁に参照するような定数は他の8Kバイトのデータエリアに置くように定義することで、パフォーマンスの向上が見込めます。
例:

```
const char sz_s_msg1[16] __attribute__((sda)) = "Hello, World!!¥n";

sprintf( szBuf, sz_s_msg1 );
```

この例はSデータエリアが2命令展開によりアクセスできることを前提にしています(-mesdaを使用しない)。Sデータエリアを3命令でアクセスする場合、メモリのアクセスウェイト数が同じであれば速度の向上は見込めません。

2.5.7 G データエリア

Gデータエリアはデフォルト、S、T、およびZデータエリアとは異なる特殊なデータエリアです。サイズは8Kバイトに固定で、2命令による高速なアクセスが可能になっています。また、固有のデータエリアポインタを持たないため、他のデータエリアポインタを使用します。したがって、このエリアは他のデータエリアの一部を共有します。アドバンスドマクロモードを使用している場合は、Gデータエリアを他のデータエリアとは独立して設定可能です。
Gデータエリアにデータを置く場合は、S、T、およびZデータエリアのような宣言は必要ありません。デフォルトデータエリアと同様にデータエリアが明示されない変数の中で、コンパイラオプション-mgdaで指定されたサイズ以下の変数がGデータエリアに配置されます。
例: -mgda=2を指定した場合(例はすべてグローバル変数)

```
char   c_data;        ← Gデータエリアに配置
short  s_data;        ← Gデータエリアに配置
int    i_data;        ← デフォルトデータエリアに配置
```

-mgda=2を指定すると、2バイト以下の変数がGデータエリアに配置されます。-mgda=0を指定すると、Gデータエリアは使用されません。

通常、Gデータエリアのセクションは RAMの先頭に配置し、それに続けてデータエリアポインタを共有するデータエリアのセクションを配置します。ただし、Gデータエリアが RAMの先頭から8Kバイト以内に収まれば、RAMの先頭に配置する必要はありません。
次の例では Gデータエリアポインタとしてデフォルトデータエリアポインタを使用しています。

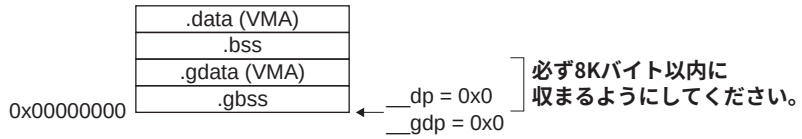


図2.5.7.1 Gデータエリア

Gデータエリアを使用する場合、コンパイル時に-mgdaおよび-mgdpオプションを指定すると共に、リンカスクリプトファイル内ではデータエリアポインタシンボル__gdpの値を指定のデータエリアポインタのシンボルと同じにしておく必要があります。

・コンパイラの-mgdpオプション
デフォルトデータエリアポインタ(R15)を使用: -mgdp=dp(デフォルト設定)
Zデータエリアポインタ(R14)を使用: -mgdp=zdp
Tデータエリアポインタ(R13)を使用: -mgdp=tdp
Sデータエリアポインタ(R12)を使用: -mgdp=sdp

・リンカスクリプトファイル内の設定
デフォルトデータエリアポインタを使用: __gdp=__dp※
Zデータエリアポインタを使用: __gdp=__zdp
Tデータエリアポインタを使用: __gdp=__tdp
Sデータエリアポインタを使用: __gdp=__sdp

必ず、使用するデータエリアポインタシンボルか、まったく同じ数値を指定します。

※ アドバンスドマクロ使用時は、__gdp=__dpである必要はありません。

Gデータエリアは初期値なしおよび初期値付き変数を格納するための、RAMタイプのデバイスにのみ有効なデータエリアです。ローカルなstatic変数もGデータエリアの配置対象になります。
Gデータエリアに定数データを配置することはできません。

アドバンスドマクロモード時のGデータエリアポインタ

アドバンスドマクロCPUにはDPレジスタが設けられており、これをデフォルトデータエリアポインタとして使用します。これにより、R15をGデータエリアポインタに割り当てることができますので、他のデータエリアポインタを使用しない、独立したデータエリアを設定できるようになっています。
Gデータエリアのサイズ(最大8Kバイト)や-mgdaオプションによるデータ配置条件は前記のスタンダードマクロ使用時と同じです。
この設定を使用するには、アドバンスドマクロCPU用のコンパイラオプション-mc33advを指定します。
-mgdpオプションはデフォルト設定(-mgdp=dp)で使用するため、省略可能です。リンカスクリプト内では、__gdpにR15と同じデータエリアの先頭アドレスを設定します。

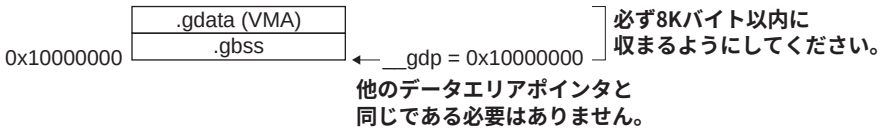


図2.5.7.2 アドバンスドマクロモード時のGデータエリア

2.6　Cとコードオプティマイズ

ここでは、gnu33\sample\ccode内のmain.cを例に、Cコンパイラで生成される命令コードのサイズやオプティマイズの手法について説明します。

元のソースは以下のとおりです。

アセンブリソースのブートルーチン　ccode\boot.s

```
; boot.s  2002.02.04
; boot program

          .text
          .align   2
          .global  boot
          .long    boot
boot:
          xld.w    %r15,0x0800
          ld.w     %sp,%r15          ; set SP
          xld.w    %r15,__dp         ; set default data area pointer
          xcall    main              ; goto main
          xjp      boot              ; infinity loop
```

Cソースのメインプログラム　ccode\main.c

```
/* main.c 1999.7.28 */
/* sample program for optimize*/

struct ST gst;
int a;
struct ST {
        int s1;
        int s2;
};

main()
    {
        int b;
        struct ST st;
        int ar[10];

        a = 1;
        b = 2;

        st.s1 = 3;
        ar[3] = 4;

        sub1(a, &b);
        sub2();

        gst.s2 = 5;
        sub3(&st, ar);
    }

sub1(a,b)
    int a;
    int *b;
    {
        *b = a;
    }

sub2()
    {
        volatile char *vp;

        vp = (volatile char *)0x40000;
        *vp = 2;

        *(volatile char *)(0x48000) |= 0x1;
    }

sub3(st, ar)
    struct ST *st;
    int ar[];
    {
        st->s2 = 4;
        ar[5]=5;
    }
```

これをデフォルトのmakeファイルでコンパイルした場合のコードは以下のとおりです。
```
>objdump -S sample.elf > dis_elf.txt
```

コンパイル後のコード　ccode\dis_elf.txt

```
sample.elf:     file format elf32-c33

Disassembly of section .text:

00c00000 <__START_text>:
  c00000: 0004  nop                     ***
  c00002: 00c0  pops    %psr            ***

00c00004 <boot>:
       .align 2
       .global boot
       .long  boot
boot:
       xld.w  %r15,0x0800
  c00004: c020  ext     0x20
  c00006: 6c0f  ld.w    %r15,0x0        xld.w     %r15,0x800
       ld.w   %sp,%r15       ; set SP
  c00008: a0f1  ld.w    %sp,%r15        ld.w     %sp,%r15
       xld.w  %r15,__dp      ; set default data area pointer
  c0000a: c000  ext     0x0
  c0000c: c000  ext     0x0
  c0000e: 6c0f  ld.w    %r15,0x0        xld.w     %r15,0x0
       xcall  main           ; goto main
  c00010: c000  ext     0x0
  c00012: c000  ext     0x0
  c00014: 1c04  call    0x4             xcall     0x8        (0x00C0001C)
       xjp    boot           ; infinity loop
  c00016: dff8  ext     0x1ff8
  c00018: dfff  ext     0x1fff
  c0001a: 1ef5  jp      0xf5            xjp       0xfffffffea (0x00C00004)

00c0001c <main>:
       int s2;
};

         |

sub3(st, ar)
    struct ST *st;
    int ar[];
    {
       st->s2 = 4;
  c00070: 6c44  ld.w    %r4,0x4         ld.w     %r4,0x4
  c00072: c004  ext     0x4
  c00074: 3c64  ld.w    [%r6],%r4       xld.w     [%r6+0x4],%r4

00c00076 <.LM23>:
       ar[5]=5;
  c00076: 6c54  ld.w    %r4,0x5         ld.w     %r4,0x5
  c00078: c014  ext     0x14
  c0007a: 3c74  ld.w    [%r7],%r4       xld.w     [%r7+0x14],%r4

00c0007c <.LM24>:
    }
  c0007c: 0640  ret                     ret
```

● 外部変数とauto変数について

まず、外部変数とauto変数のアクセス方法を説明します。

```
       a = 1;                                                    ← 外部変数へのアクセス
  c0001e: 6c16  ld.w    %r6,0x1         ld.w     %r6,0x1
  c00020: c000  ext     0x0
  c00022: c008  ext     0x8
  c00024: 3cf6  ld.w    [%r15],%r6      xld.w     [%r15+0x8],%r6

00c00026 <.LM4>:
       b = 2;                                                    ← auto変数へのアクセス
  c00026: 6c24  ld.w    %r4,0x2         ld.w     %r4,0x2
  c00028: 5ca4  ld.w    [%sp+0xa],%r4   ld.w     [%sp+0xa],%r4
```

aは外部変数(ここではRAM上の変数のほか、ROM上の定数値、static宣言された変数も含みます。絶対アドレスがあるもの)、bはauto変数(スタック上に置かれる変数)です。

通常、外部変数へのアクセスは
1）32ビット値(変数のアドレス)をR15に入れる
2）R15をベースにメモリアクセス
という手順で行われるため、4命令必要です。

auto変数は、
1）SP+offsetでアクセス
という手順のため、オフセットがバイトアクセスでは63バイト以下、ハーフワードアクセスでは126バイト以下、ワードアクセスでは252バイト以下のスタックエリアのauto変数なら1命令、それを越えても2命令でアクセスできます。
また、数が少ない場合は自動的にレジスタに割り振られ、さらに効率よく処理できる場合もあります。その場合、すでにレジスタに入っているため、0命令でのアクセスに相当します。

あるルーチンで一時的に使用する変数は、できるだけauto変数に割り振ることを推奨します。
これは、以下のような理由によります。
・ 上記のようにアクセスのための命令数が少なく、処理速度も速い
・ スタック上に一時的に置かれるため、RAMを常時占有せず、RAMを節約できる
・ レジスタへの割り付けや不要なアクセスの消去等、Cコンパイラのオプティマイズの恩恵を受けやすい

auto変数を多用することによるデメリットとしては、
・ stackサイズが大きくなり、かつ上限値が予想しにくくなる
ことがあげられます。

スタックサイズはデバッガを使用し次の手順で簡単にチェックできます。
1）スタックエリアを少し大き目に取る
2）スタックを5555等でフィル
3）アプリケーションを実行
4）最後にスタックエリアを表示し、スタックの最大使用範囲(5555が変更されている範囲)を確認

● volatile変数について

最新のCコンパイラでは、メモリへのロード/ストアをできるだけ減らし、レジスタに入っている値を可能な限り再利用して処理を行うことで、コードをコンパクトに、また処理速度を上げています。
逆に、Cでメモリアクセスの記述をしても、そこで必ずメモリがアクセスされるという保証はありません。I/Oレジスタへのアクセスを記述する場合、これでは困りますので、ANSI CではvolatileというI/O変数型を定義しています。I/Oレジスタ等はこれでアクセスするようにしてください。

```
sub2()
    {
       volatile char *vp;

       vp = (volatile char *)0x40000;
  c0005e: c000  ext     0x0
  c00060: d000  ext     0x1000
  c00062: 6c05  ld.w    %r5,0x0         xld.w     %r5,0x40000

00c00064 <.LM18>:
       *vp = 2;
  c00064: 6c24  ld.w    %r4,0x2         ld.w     %r4,0x2
  c00066: 3454  ld.b    [%r5],%r4       ld.b     [%r5],%r4       ← アクセス

00c00068 <.LM19>:

       *(volatile char *)(0x48000) |= 0x1;
  c00068: c200  ext     0x200
  c0006a: 6005  add     %r5,0x0         xadd      %r5,0x8000
  c0006c: b050  bset    [%r5],0x0       bset     [%r5],0x0       ← bsetアクセス

00c0006e <.LBE3>:
    }
  c0006e: 0640  ret                     ret
```

vpという変数をvolatile型として宣言し、0x40000というアドレスを設定して、そこに2を書き込んでいます。確実にメモリライトが発生します。

さらに、0x48000番地に0x1をOR書き込みをしています。ここでは、0x48000という直値をキャストしてアドレスポインタとして扱っています。また、volatileのバイト型でビットのセット、クリアを行う場合はbsetやbclr命令が生成され、通常3命令以上かかる処理が1命令で処理されます。

● ポインタ変数について

ポインタ変数によるポイント先へのアクセスは1命令で処理されます。

```
sub1(a,b)
    int a;
    int *b;
    {
       *b = a;
  c0005a: 3c76  ld.w    [%r7],%r6       ld.w     [%r7],%r6       ← 1命令アクセス

00c0005c <.LM14>:
    }
  c0005c: 0640  ret                     ret
```

● 構造体変数、配列について

構造体変数/配列は基本的に外部変数かauto変数で、前記の外部変数/auto変数と同じ方法でアクセスされます。

```
main()
    {
  c0001c: 840d  sub     %sp,0xd         sub      %sp,0xd
                :
                :
        st.s1 = 3;
  c0002a: 6c34  ld.w    %r4,0x3         ld.w     %r4,0x3
  c0002c: 5cb4  ld.w    [%sp+0xb],%r4   ld.w     [%sp+0xb],%r4   ← auto変数アクセス

00c0002e <.LM6>:
        ar[3] = 4;
  c0002e: 6c44  ld.w    %r4,0x4         ld.w     %r4,0x4
  c00030: 5c34  ld.w    [%sp+0x3],%r4   ld.w     [%sp+0x3],%r4   ← auto変数アクセス
                :
                :
        sub2();
  c0003c: c000  ext     0x0
  c0003e: c000  ext     0x0
  c00040: 1c0f  call    0xf             xcall     0x1e       (0x00C0005E)

00c00042 <.LM9>:

        gst.s2 = 5;
  c00042: 6c54  ld.w    %r4,0x5         ld.w     %r4,0x5
  c00044: c000  ext     0x0
  c00046: c004  ext     0x4
  c00048: 3cf4  ld.w    [%r15],%r4      xld.w     [%r15+0x4],%r4 ← 外部変数アクセス
```

Cコンパイラは、構造体や配列の各要素をauto変数の場合はSPからのオフセット、外部変数の場合は絶対アドレスに変換してアクセスしますので、通常のauto変数や外部変数とまったく同じ扱いとなります。

● ポインタ型構造体、配列について

```
sub3(st, ar)
    struct ST *st;
    int ar[];
    {
        st->s2 = 4;
  c00070: 6c44  ld.w    %r4,0x4         ld.w     %r4,0x4
  c00072: c004  ext     0x4                                      ← オフセットアクセス
  c00074: 3c64  ld.w    [%r6],%r4       xld.w     [%r6+0x4],%r4  ← 2命令

00c00076 <.LM23>:
        ar[5]=5;
  c00076: 6c54  ld.w    %r4,0x5         ld.w     %r4,0x5
  c00078: c014  ext     0x14                                     ← オフセットアクセス
  c0007a: 3c74  ld.w    [%r7],%r4       xld.w     [%r7+0x14],%r4 ← 2命令

00c0007c <.LM24>:
    }
  c0007c: 0640  ret                     ret
```

このように、外部変数の構造体や配列の先頭をポインタとして各要素にアクセスすると、2命令でアクセスできます。（オフセットは最大13ビットでアクセスエリアが4KBまで。 それ以上は3命令）
まとまった外部変数エリアのアクセスに有効な手法です。

● 同名のグローバルシンボル定義をしたときの注意点

1）1つのCソースファイル内に同名のグローバルシンボルを2つ以上定義しても、コンパイラではエラーおよびワーニングになりません。コンパイラは1つのシンボル定義と見なして処理します。
この際にワーニングを表示させたい場合は、"-Wredundant-decls"オプションを指定してコンパイルしてください。

2）異なるCソースファイルにextern宣言なしの同名のグローバルシンボルを定義した場合、リンカは重複エラーにしません。リンカはシンボルを同一アドレスと見なして処理します。
例:
(src1.c)
```
    int    iSym1;
    iSym1 = 1;
```
(src2.c)
```
    int    iSym1;
    iSym1 = 2;
```
実コードは正常にリンクされますが、シンボルiSym1は.bssセクション内に2つあると認識されて、2倍のサイズが.bssセクションに確保されてしまいます。
iSym1 = 4バイト、4バイト×2 = 合計8バイト

これにより、使用されない領域(この場合、4バイト)が発生します。
これを避けるため、外部シンボルを参照する場合は、必ずextern宣言を行うようにしてください。
例:
(src1.c)
```
    int    iSym1;
    iSym1 = 1;
```
(src2.c)
```
    extern int    iSym1;
    iSym1 = 2;
```

3）異なるCソースファイルに同名の定数シンボルを定義した場合にそのシンボルを参照すると、同一ソースファイル内に定義した値が参照されます。
例:
(src1.c)
```
    const sym1 = 1;
    int i;

    i = sym1;      /* iには1が代入されます。 */
```
(src2.c)
```
    const sym1 = 2;
    int k;

    k = sym1;      /* kには2が代入されます。 */
```
このようなコーディングは不具合や間違いの原因になりますので、同じシンボル名の定義は極力避けるようにしてください。

4）1つのファイルにも全く同じ名称のシンボルを定義することが可能です。たとえば、同じ名称のシンボルを2つ定義した場合、メモリ上には2つの変数領域が確保されます。ただし、一方の変数には正常にアクセスできますが、他方には全くアクセスできません。メモリエリアの浪費となりますので、同名のシンボルを定義することは避けてください。
例:
```
int  siData1;

int  siData1;

sub() {
      siData1 = 1;
};
```
一方のsiData1には正常にアクセスできますが、もう一方のsiData1にアクセスすることはできなくなり、.bssを無駄に消費してしまいます。

● まとめ

Cコードの記述やコードのオプティマイズに関して推奨する項目を、有効度の高い順に示します。

1）外部変数(絶対アドレスのあるもの)にする必要のないものは、auto変数(スタック上の変数)として使用します。
2）外部変数は構造体や配列にして、先頭のポインタからのオフセットでアクセスします。4KBの範囲まではかなり有効です。

なお、GCC33自体のオプティマイズスイッチには常に-Oを使用することを推奨します。-O2や-O3を指定しても特殊なオプティマイズ処理が行われるだけで、あまり効果は期待できません。

Cコンパイラは-O、-O2、-O3いずれか1つのスイッチの指定に従って最適化処理を行います。コード効率と速度(主にコード効率)を重視して最適化されたコードが出力されます。スイッチが指定されない場合は最適化は行われません。-Oの数値が上がるほどコード効率が高まりますが、デバッグ情報が一部出力されないなどの問題が発生することがありますので注意してください。正常に実行できないときは、最適化の数値を下げてください。通常は、-Oでコンパイルすることを推奨します。ワークベンチgwb33が生成する基本的なmakeファイルは、Cコンパイラxgccの起動コマンドに-Oオプションを使用しています。
最適化を指定すると、Cコンパイラxgccはメモリからレジスタにロードした値を再利用し、メモリのリード/ライトを減らすようにします。そのため、メモリへのアクセスの発生を保証できなくなります。これを避けるには、以下の方法で対応してください。

- 変数のvolatile宣言　　例) volatile char IO_port1;
- 最適化を行わない
- -fvolatile指定　　ポインタがvolatileとしてアクセスされます。
  -fvolatile-global指定　　外部変数がvolatileとしてアクセスされます。

2.7 リンカによるマップ

セクションはリンカスクリプトファイルに指定されたとおりメモリに配置されます。
リンカld にリンカスクリプトファイルを読み込むには、起動時に-Tオプションで指定します。
以下に示す例は、デフォルトデータエリアポインタだけを使用した、基本的なスクリプトファイルです(xgccオプション-mgdp=dpの場合)。

●メモリ配置

デフォルトデータエリアポインタを0x0番地に定義(__dp=0x0)

内部RAM

.gbssセクション: アドレス0x0から配置
.gdataセクション: .gbssの最終アドレス直後に配置(ロードエリアLMAは.rodataの後)
.bssセクション: .gdataの最終アドレス直後に配置
.dataセクション: .bssの終わりから配置(ロードエリアLMAは.gdataロードエリアの後)

外部ROM

test2.oとtest3.oの.textセクション: アドレス0x600000から配置

外部ROM

test2.oとtest3.o以外の.textセクション(test2.o、test3.o以外): 0xc00000から配置
.rodataセクション: .textの最終アドレス直後に配置
.gdataのロードエリア(LMA): .rodataの最終アドレス直後に配置
実行時は__start_gdataに初期値をコピー
.dataのロードエリア(LMA): .gdata(LMA)の最終アドレス直後に配置
実行時は__start_dataに初期値をコピー

ここでは、ブートベクタがboot.o内の.textセクション先頭にあるとします。

●リンカスクリプトファイル

上記のメモリ配置を指定するリンカスクリプトファイルは次のようになります。

リンカスクリプトファイル

```
OUTPUT_FORMAT("elf32-c33", "elf32-c33",
              "elf32-c33")
OUTPUT_ARCH(c33)
SEARCH_DIR(.);
SECTIONS
{
        __dp = 0x0;

        . = 0x0;
        .gbss   : { *(.gbss) }
        .gdata  :
                AT (ADDR (.text) + SIZEOF (.text) + SIZEOF (.rodata))
                        { __start_gdata = . ; *(.gdata); __end_gdata = . ; }
        .bss    : { *(.bss) }
        .data   :
                AT (LOADADDR (.gdata) + SIZEOF (.gdata))
                { __start_data = . ; *(.data); __end_data = . ; }

        . = 0x600000;
        outputa : {
                test3.o (.text)
                test2.o (.text)
        }

        . = 0xc00000;
        .text   : {
                boot.o(.text)                          ← (1)
                main.o(.text)
                C:/gnu33/lib/libc.a(.text)             ← (2)
                C:/gnu33/lib/libgcc.a(.text);
        }
        .rodata : { *(.rodata) }
}
```

(1)ブートベクタを持つboot.oをセクションの先頭に置くことを明示的に指定しています。

(2)セクションのファイル指定に"*"が指定できない場合、ライブラリは任意のセクションに強制的に追加されます。この追加されたライブラリが他のセクションと重複する可能性があるため、可能な限りライブラリの位置を明示的に指定してください。

注: セクションのファイル配置に"*"を指定せずにファイル名を一度でも指定した場合、それ以降に"*"を指定したセクションは、最後に指定したファイルがセクションの先頭に配置されるようになります。

●TEXTセクション、DATAセクション、BSSセクション

Cおよびアセンブリソースの記述内容は、最終的に3つのセクションに分類されます。

TEXTセクション　プログラム、ROMデータが入るセクションです。

DATAセクション　R/W可能な初期値付きデータが入るセクションです。

BSSセクション　R/W可能な初期値なしの変数が入るセクションです。

例)

```
int a;                  ← BSSセクションに入ります
int b=1;                ← DATAセクションに入ります
const int c=2;          ← TEXTセクションに入ります

main()                  ← プログラムはTEXTセクションに入ります
    {
       a=b=c;
    }
```

これをコンパイルすると以下のようになります。

```
        .global b
        .section .data
.stabs "b:G(0,1)",32,0,0,0
        .align 2
        .type   b,@object
        .size   b,4
b:
        .long 1                         ← bは、DATAセクションのデータ
        .global c
        .section .rodata
.stabs "c:G(0,1)",32,0,0,0
        .align 2
        .type   c,@object
        .size   c,4
c:
        .long 2                         ← cは、TEXTセクションのデータ
        .section .text
        .align 1
.stabs "main:F(0,1)",36,0,0,main
        .global main
        .type   main,@function
main:
.stabn 68,0,8,.LM1-main
.LM1:
.stabn 68,0,9,.LM2-main
.LM2:
        xld.w  %r4,2  ;0x2              ← 命令はすべてTEXTセクション
        ext doff_hi(b)
        ext doff_lo(b)
        ld.w   [%r15],%r4
        ext doff_hi(a)
        ext doff_lo(a)
        ld.w   [%r15],%r4
.stabn 68,0,10,.LM3-main
.LM3:
        ret
.Lfe1:
        .size   main,.Lfe1-main
.stabs "a:G(0,1)",32,0,0,0
        .global a
```

```
        .section .bss
        .align 2
        .type   a,@object
        .size   a,4
a:
        .zero  4                        ←aは、BSSセクション
        .text
        .stabs "",100,0,0,Letext
Letext:
        .ident "GCC: (GNU) 2.95.2 19991024 (release)"
```

なお、TEXT、DATA、BSSという分類や擬似命令は、UNIXの概念を取り入れています。
DATAセクション(初期値付き、R/W可能な変数)のサポートは、各社開発ツールによって異なります。サポートしていない開発ツールもありますので、ソースを新規に作成する場合はできるだけ使用しないないことを推奨します。
BSSセクションの変数として定義し、必要なものはプログラムで初期化するようにしておくと、移植性は高まります。
すでにPC上などで開発してあるCソースを使用する場合などは、記述されたDATAセクションをそのまま残すこともあります。組み込みシステムでDATAセクションをR/W可能なデータとして扱うには、データをROMに書き込んでおき、ブート時にRAMに展開する必要があります。具体的な例については後述します。

●DATAセクションの使用方法およびプログラムの内蔵RAMへのキャッシュ方法

DATAセクションとは、初期値を持つR/W可能な変数エリアのことです。(各セクションの意味については前述の"● TEXTセクション、DATAセクション、BSSセクション"を参照)
DATAセクションを使用するには、以下の3条件を満たす必要があります。
1) 変数の初期値をROMに書き込んでおくこと
2) ROMのデータをRAMに展開すること
3) RAMに展開されたデータをもとに、プログラムが動作すること

同様の方法で、プログラムを内蔵RAMにコピーして高速実行させることができます。外部ROMやFLASHメモリにプログラムがある場合は1〜2ウェイトのアクセスとなりますが、内蔵RAMにコピーすることにより0ウェイトで実行可能です。

この手順をgnu33\sample\sectionを例に説明します。

リンカスクリプトファイルの指定方法

例)section\section.ldsより抜粋

```
        .bss 0x00000000 :
          {
           __START_bss = . ;
           main.o(.bss) ;
           __END_bss = . ;
          }

        .data __END_bss : AT( __END_rodata )
          {
           __START_data = . ; ← RAM上のデータ展開エリアを示すセクションシンボル
           *(.data) ;
           __END_data = . ;
          }
        __START_data_lma = LOADADDR( .data ); ← ROM上の格納エリアを示すセクションシンボル

        .cache __END_data : AT( __START_data_lma + SIZEOF( .data ) )
          {
           __START_cache = . ; ← RAM上のプログラムキャッシュ領域を示すセクションシンボル
           cache.o(.text) ;
           __END_cache = . ;
          }
        __START_cache_lma = LOADADDR( .cache ); ← ROM上の格納エリアを示すセクションシンボル

        Share1 __END_cache :
          {
           __START_Share1 = . ;
           share1.o(.bss) ;
           __END_Share1 = . ;
          }
```

```
        Share2 __END_cache :
          {
           __START_Share2 = . ;
           share2.o(.bss) ;
           __END_Share2 = . ;
          }

        .text 0x00c00000 :
          {
           __START_text = . ;
           boot.o(.text)
           main.o(.text)
           c:/gnu33/lib/libgcc.a(.text) ;
           __END_text = . ;
          }

        .rodata __END_text :
          {
           __START_rodata = . ;
           *(.rodata) ;
           __END_rodata = . ;
          }
```

リンク後のマップの確認はobjdump -hで確認できます。

```
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .bss          00000008  00000000  00000000  00001000  2**2
                  ALLOC
  1 .data         00000004  00000008  00c000d4  00003008  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .cache        0000003e  0000000c  00c000d8  0000300c  2**1
                  CONTENTS, ALLOC, LOAD, CODE
  3 Share1        00000192  0000004a  0000004a  00001042  2**2
                  ALLOC
  4 Share2        00000192  0000004a  0000004a  00001eb0  2**2
                  ALLOC
  5 .text         000000ce  00c00000  00c00000  00002000  2**1
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  6 .rodata       00000006  00c000ce  00c000ce  000020ce  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  7 .stab         00000618  00c000d4  00c000d4  0000304c  2**2
                  CONTENTS, READONLY, DEBUGGING
  8 .comment      000000be  00c00a78  00c00a78  00003664  2**0
                  CONTENTS, READONLY
  9 .stabstr      0000038c  00c006ec  00c006ec  00003722  2**0
                  CONTENTS, READONLY, DEBUGGING
```

ブート時の転送

以下のように、リンカスクリプトファイル内に定義されたセクションシンボルを使用してデータをRAMに転送してから、プログラムの実行を開始してください。

例)section\boot.sより抜粋

```
        .text
        .long BOOT                          // BOOT VECTOR

BOOT:
        xld.w  %r15,0x800
        ld.w   %sp,%r15                     // set SP 0x800
        xld.w  %r15, __dp                   // set data pointer
        // if you use Advanced Macro CPU, please use below source.
/*      xld.w  %r4, __dp                    // set data pointer
        ld.w   %dp,%r4                      // set data pointer        */

        // ROM has data at end of rodata; copy it
        xld.w  %r6, __START_data_lma        // .data LMA
        xld.w  %r7, __START_data            // .data VMA
        xld.w  %r8, __END_data              // end of .data VMA
        call   HCOPY_LOOP

        // transfer function cache_exec's program code to RAM
        xld.w  %r6, __START_cache_lma       // .cash LMA
        xld.w  %r7, __START_cache           // .cash VMA
        xld.w  %r8, __END_cache             // .cash VMA
        call   HCOPY_LOOP
```

```
        // start main
        xcall   main                        // goto main
        jp      BOOT                        // infinity loop

        ; copy %r6 addr to %r7 addr until %r8 addr
HCOPY_LOOP:
        ld.b    %r4,[%r6]+                  // read byte from src addr
        ld.b    [%r7]+,%r4                  // write byte to dest addr
        cmp     %r7,%r8                     //
        jrult   HCOPY_LOOP
        ret
```

Cで記述する方法

Cで記述する場合は以下のようなコードとなります。

```
extern char __START_cache_lma;
extern char __START_cache;
extern char __END_cache;

char *src = &__START_cache_lma
char *dst = &__START_cache;

while(dst < &__END_cache)
        *dst++ = *src++;
```

注: リンカスクリプトで生成したシンボルをCコード内で参照すると、シンボル値によってはリンカでエラーとなる場合があります。
上記の"char *dst = &__START_cache;"をコンパイルすると、次のようなコードが生成されます。

```
ext     doff_hi(__START_cache)
ext     doff_lo(__START_cache)
add     %r4,%r15
```

シンボル(アドレス)からのオフセット値が26ビット長以内では問題ありませんが、その範囲を超えている場合はアドレスを正しく設定することができず、リンカでエラーとなります。この制約は、データエリアをサポートするS5U1C33001C特有のもので、GNU i386コンパイラ等では、この問題は発生しません。

●ライブラリの指定方法

S5U1C33001Cの提供するANSIライブラリ(lib/libc.a)またはエミュレーションライブラリ(lib/libgcc.a)をリンカスクリプトファイルで指定する場合、以下のように記述します。

```
          :
          :
.text 0xc01000 :
{
   boot.o(.text)
   main.o(.text)
   C:/gnu33/lib/libc.a(.text)
   C:/gnu33/lib/libgcc.a(.text)
}
```

ライブラリのドライブ名およびディレクトリは、makeファイルで指定した内容と全く同じに記述してください。大文字/小文字の区別も同じにする必要があります。

●リンカスクリプトファイルでのセクション名定義に関する注意事項

リンカスクリプトファイルでは、必ずセクション".text"と".data"を定義してください。リンク後のelfファイル内にこの2つのセクション名が存在しない場合、デバッガgdbでそのファイルをロードすることができません。

例)

```
TEST1 0xc00000 : { *.o (.text) }
TEST2 0xc01000 : { *.o (.data) }
```

のような定義では、elfファイルに".text"、".data"セクションが出力されません。

●リンク例

例1)デフォルトデータエリアのみを使用する場合

デフォルトデータエリアのみに対応した最小構成のリンカスクリプト例を以下に示します。

リンカスクリプトファイル例: sample\ldscript\default\default.lds

```
OUTPUT_FORMAT("elf32-c33", "elf32-c33",
              "elf32-c33")
OUTPUT_ARCH(c33)
SEARCH_DIR(.);
SECTIONS
{
       /* data pointer symbol By GWB33 */
       __dp = 0x0;                                                    ...1

       /* section information By GWB33 */
       . = 0x0;

       .bss 0x00000000 :                                              ...2
         {
          __START_bss = . ;
          *(.bss) ;
          __END_bss = . ;
         }

       .data __END_bss : AT( __END_rodata )                           ...3
         {
          __START_data = . ;
          *(.data) ;
          __END_data = . ;
         }
       __START_data_lma = LOADADDR( .data );

       .text 0x00c00000 :                                             ...4
         {
          __START_text = . ;
          *(.text) ;
          __END_text = . ;
         }

       .rodata __END_text :                                           ...5
         {
          __START_rodata = . ;
          *(.rodata) ;
          __END_rodata = . ;
         }

}
```

1. デフォルトデータエリアポインタ(__dp)を0x0に設定します。
2. 入力ファイルのすべての.bssセクションをアドレス0x0からはじまる.bss出力セクションに配置します。このセクションは実際のコードを持ちませんので、LMAを指定する必要はありません。
3. .bssセクションの直後に続くメモリ空間は、入力ファイルの.dataセクションに割り当てられます。初期値(実コード)は、.rodataセクションに続いて配置されます。このLMAはAT文で指定します。__END_rodataは.rodataコマンドで定義されたシンボルで、.rodataセクション直後のロケーションカウンタ値を示します。
   __START_dataと__END_dataは.dataセクション(VMA)の開始/終了アドレスを参照するために定義されたシンボルです。これらのシンボルは、初期化ルーチンでLMAからVMAに初期値をコピーするのに使用します。コピー方法については、"●DATAセクションの使用方法およびプログラムの内蔵RAMへのキャッシュ方法"の項に記載されている"ブート時の転送"を参照してください。
4. 入力ファイルのすべての.textセクションはアドレス0xc00000から配置されます。
5. 入力ファイルのすべての.rodataセクションは、.textセクションの直後に配置されます。.dataセクションの実コードをこの直後に配置するため、LMAを指定する__END_rodataを定義しています。

図2.7.1にこのスクリプト例で構成されるメモリマップを示します。

```
ld -o sample.elf file1.o file2.o -T default.lds
```

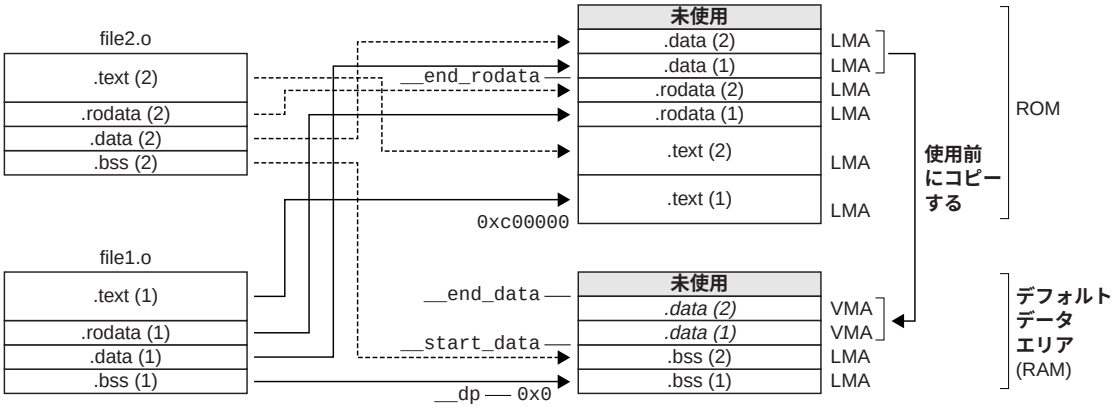


図2.7.1 メモリマップ(例1)

例2)すべてのデータエリアを使用する場合

すべてのデータエリアを使用する場合のリンカスクリプト例を以下に示します。

リンカスクリプトファイル例: sample\ldscript\all_area\allarea.lds

```
OUTPUT_FORMAT("elf32-c33", "elf32-c33",
              "elf32-c33")
OUTPUT_ARCH(c33)
SEARCH_DIR(.);
SECTIONS
{
    __dp  = 0x0;
    __gdp = 0x0;
    __sdp = 0x1000;
    __tdp = 0x2000;
    __zdp = 0x3000;

       . = 0x0;
       .gbss : { *(.gbss) }
       .gdata  :
        AT (__load_gdata)
        { __start_gdata = . ; *(.gdata); __end_gdata = . ; }

       .bss : { *(.bss) }
       .data  :
        AT (__load_data)
        { __start_data = . ; *(.data); __end_data = . ; }

       . = 0x1000;
       .sbss : { *(.sbss) }
       .sdata  :
        AT (__load_sdata)
        { __start_sdata = . ; *(.sdata); __end_sdata = . ; }

       . = 0x2000;
       .tbss : { *(.tbss) }
       .tdata  :
        AT (__load_tdata)
        { __start_tdata = . ; *(.tdata); __end_tdata = . ; }

       . = 0x3000;
       .zbss : { *(.zbss) }
       .zdata  :
        AT (__load_zdata)
        { __start_zdata = . ; *(.zdata); __end_zdata = . ; }

       . = 0xc00000;
       .text    : { *(.text)   }
       .rodata  : { *(rodata)  }
       .rosdata : { *(rosdata) }
       .rotdata : { *(rotdata) }
       .rozdata : { *(rozdata) }

       __load_data = . ;
       __load_gdata = . + SIZEOF(.data) ;
```

```
        __load_sdata = . + SIZEOF(.data) + SIZEOF(.gdata) ;
        __load_tdata = . + SIZEOF(.data) + SIZEOF(.gdata) + SIZEOF(.sdata);
        __load_zdata = . + SIZEOF(.data) + SIZEOF(.gdata) + SIZEOF(.sdata)
                         + SIZEOF(.tdata);
}
```

各データエリアの設定は、セクション名や定義するシンボルを除き例1のデフォルトデータエリアの場合と同様です。
図2.7.2にセクションの配置を示します。

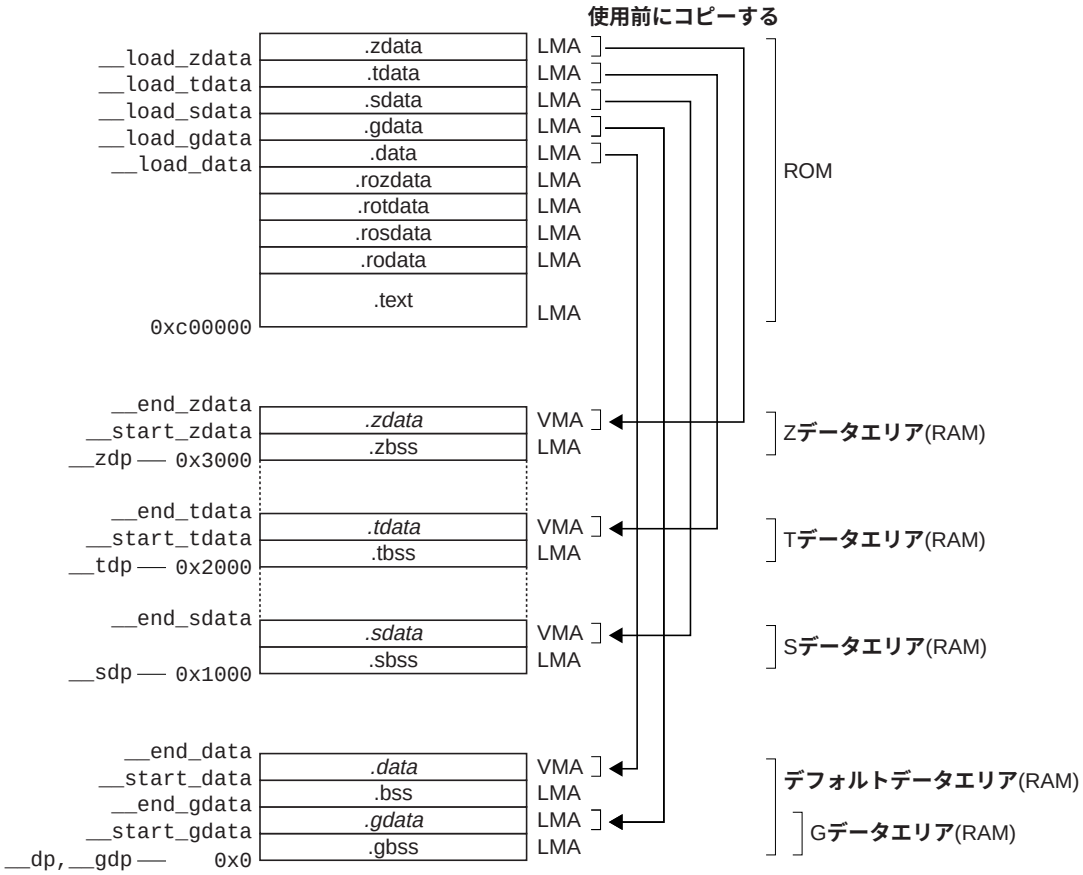


図2.7.2 メモリマップ(例2)

例3)仮想・共用セクションを使用する場合

仮想セクションおよび共用セクションを使用する場合のリンカスクリプト例を以下に示します。

仮想・共用セクションを使用するリンカスクリプト

```
OUTPUT_FORMAT("elf32-c33", "elf32-c33",
              "elf32-c33")
OUTPUT_ARCH(c33)
SEARCH_DIR(.);
SECTIONS
{
       /* data pointer symbol By GWB33 */
       __dp = 0x0;

       /* section information By GWB33 */
       . = 0x0;

       .bss 0x00000000 :
         {
          __START_bss = . ;
          *(.bss) ;
          __END_bss = . ;
         }

       .data __END_bss : AT( __END_rodata )
         {
          __START_data = . ;
          *(.data) ;
          __END_data = . ;
         }
       __START_data_lma = LOADADDR( .data );

       .text_foo1 __END_data : AT( __START_data_lma+SIZEOF( .data )
         {
          __START_text_foo1 = . ;
          foo1.o(.text) ;
          __END_text_foo1 = . ;
         }
       __START_text_foo1_lma = LOADADDR( .text_foo1 );

       .text_foo2 __END_data : AT( __START_text_foo1_lma+SIZEOF( .text_foo1 )
         {
          __START_text_foo2 = . ;
          foo2.o(.text) ;
          __END_text_foo2 = . ;
         }
       __START_text_foo2_lma = LOADADDR( .text_foo2 );

       .text_foo3 __END_data : AT( __START_text_foo2_lma+SIZEOF( .text_foo2 )
         {
          __START_text_foo3 = . ;
          foo3.o(.text) ;
          __END_text_foo3 = . ;
         }
       __START_text_foo3_lma = LOADADDR( .text_foo3 );

       .text 0x00600000 :
         {
          __START_text = . ;
          *(.text) ;
          __END_text = . ;
         }

       .rodata __END_text :
         {
          __START_rodata = . ;
          *(.rodata) ;
          __END_rodata = . ;
         }
}
```

図2.7.3にセクションの配置を示します。

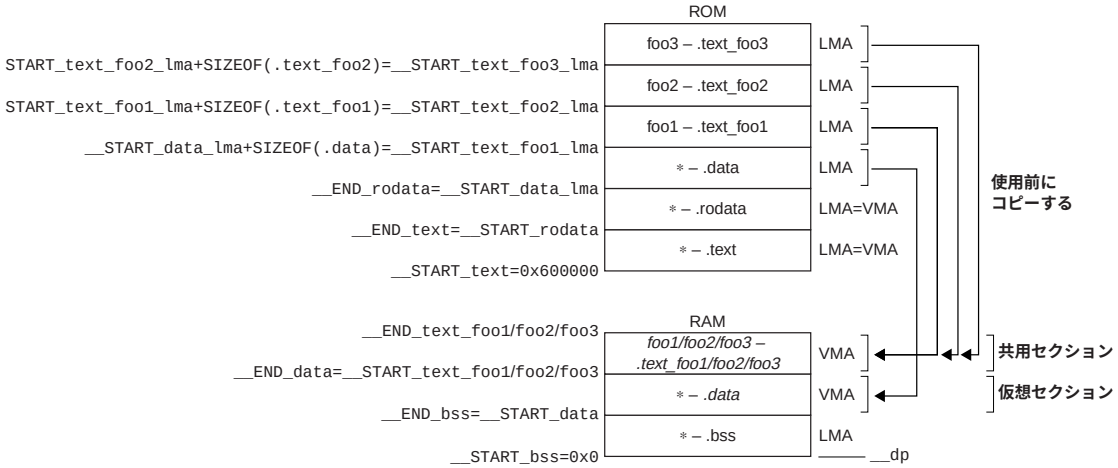


図2.7.3 メモリマップ(例3)

.dataセクションはROM上のLMAに実体を持ち、RAM上のVMA(.bssセクションの直後)にコピーして使用するように設定しています。RAM上の.dataセクション(VMA)はプログラムの実行開始時にはデータが存在しない仮想セクションと見ることができます。初期値を持つ変数などは、このように使用する必要があります。この例の場合、すべてのファイルの.dataセクションが1つにまとめられています。

.text_foo1はfoo1.oファイルの.textセクションで、ROM上のLMAに実コードを持ち、RAM上のVMAで実行するように設定しています。.text_foo2セクションと.text_foo3セクションも同様ですが、これら3つのセクションは同じVMAが設定されています。同じ実行アドレスを共用しているため、RAM上の.text_foo1/2/3セクションは共用セクションと見ることができます。プログラムを高速実行させるためのキャッシュのような使い方はこのように実現します。これら3つ以外のファイルの.textセクションは0x600000～の.textセクションに配置され、そのままROM上で実行されます。

2.8 ライブラリ

Cコンパイラパッケージには、以下に示す3種類のライブラリが含まれています。

libc.a: ANSIライブラリ
libgcc.a: エミュレーションライブラリ
libgccP.a: 高精度演算のエミュレーションライブラリ

ライブラリは、\gnu33\libディレクトリに入っています。
エミュレーションライブラリlibgcc.aおよびlibgccP.aは、Cコンパイラxgccが標準的に呼び出しますので、通常はリンクしてください。
ANSIライブラリlibc.aは、必要に応じてリンクしてください。

2.8.1 ANSIライブラリ(libc.a)

ANSIライブラリはレジスタやデータエリアに関して以下の規則で作成されています。ライブラリ関数を使用する場合は、この点を考慮してプログラミングしてください。

ライブラリで使用しているレジスタ

- R0からR11を使用しています。
- R12からR15は未使用です。
- R0からR3は関数が開始時にスタックにプッシュ、終了時にポップして保護されます。

ライブラリで使用しているデータエリア

- ライブラリ関数内で参照している変数等のデータは、デフォルトデータエリアに配置されます。
  ライブラリ関数は、デフォルトデータエリアポインタとしてR15を参照していますので、ライブラリ関数使用時はR15の初期化が必要となります。
- ライブラリはデフォルトデータエリアのみ使用しており、その他のG、S、T、Zデータエリアは使用しません。

2.8.2 エミュレーションライブラリ(libgcc.a, libgccP.a)

エミュレーションイブラリはレジスタやデータエリアに関して以下の規則で作成されています。ライブラリ関数を使用する場合は、この点を考慮してプログラミングしてください。

ライブラリで使用しているレジスタ

- R0からR14を使用しています。
- R15は未使用です。
- R0からR3は関数が開始時にスタックにプッシュ、終了時にポップして保護されます。また、R10～R14を使用しているライブラリは、これらのレジスタをスタックに退避させて保護しています。

ライブラリで使用しているデータエリア

エミュレーションライブラリは、データエリアを使用していません。

2.8.3 アドバンスドマクロでのライブラリ使用上の注意

-mc33advオプションを指定し、アドバンスドマクロ対応でコンパイルする場合、R15とDPは同じ値で初期化してください。
アドバンスドマクロに対応したアプリケーションが、通常デフォルトデータエリアポインタとしてDPを参照するのに対して、ANSIライブラリはDPではなくR15を参照します。これは、アドバンスドマクロでも同じライブラリ(libgcc.a、libc.a)を使用しているためです(アドバンスドマクロ専用ライブラリはありません)。したがって、整合性を保つためにはDPとR15を同じ値にする必要があります。

2.8.4 エミュレーションライブラリの割り込みマスクサイクル数

エミュレーションライブラリlibgcc.a、libgccP.aの中には、データエリア用のレジスタ(R12～R15)を使用している関数があります。これらの関数の実行中に割り込みが発生すると割り込みルーチンでデータエリアを正常にアクセスできなくなるため、関数の実行中は割り込みをマスクしています。
以下に割り込みマスクをしている関数と割り込みマスクサイクル数を示します。

関数名	マスクサイクル数
__mulsf3	163
__muldf3	184(180)
__floatsisf	76
__floatsidf	81
__extendsfdf2	63
__divsf3	158
__divdf3	208(220)
__adddf3	163(177)
__subdf3	163(177)

()内はlibgccP.aの関数のマスクサイクル数です。

エミュレーションライブラリで割り込みをマスクしているため、ライブラリ実行時に割り込みが発生するとプログラムの実行速度が低下することがあります。その場合の対策を以下に示します。

(1) /utility/lib_src/emulib内のソースから割り込みをマスクしている箇所を削除し、ライブラリを作成し直します。
(2) 割り込み処理ルーチンに、データポインタを再設定する処理を下記のように追加します。

```
関数の開始時:   R12～R15をスタックにプッシュ
                データエリアポインタ(R12～R15)を再設定
                        :
                データエリアのアクセス
                        :
関数の終了時:   R12～R15をポップ
```

ただし、割り込み処理の前後にコードが増えるため、割り込み処理自体の処理速度は落ちます。既存のライブラリを使用した場合とパフォーマンスを比較検討してください。

2.8.5 ライブラリ追加時の注意事項

プログラムのリンク時にANSIライブラリ、エミュレーションライブラリ以外のライブラリを追加する場合、リンカ ldに渡すファイルは以下の順序で指定してください。

ld.exe (プログラム.o) (追加ライブラリ) (ANSIライブラリ) (エミュレーションライブラリ)

オブジェクトファイル(もしくはライブラリ)は、リンカに渡すファイル順で自身よりも後方にあるライブラリのみ参照可能です。追加ライブラリを最後に指定した場合、追加ライブラリ内では外部ライブラリをいっさい使用することができません。floatやdouble演算、ANSIライブラリといった基本的な関数が使用できませんので、追加ライブラリは必ずエミュレーションライブラリ、ANSIライブラリよりも前に配置するようにしてください。

例: 1. NG

```
ld.exe -T withmylib.lds -o withmylib.elf boot.o libc.a libgcc.a mylib.a
```

mylib.aがエミュレーションライブラリ、ANSIライブラリを使用している場合、必ずリンク時にエラーが出ます。

2. OK

```
ld.exe -T withmylib.lds -o withmylib.elf boot.o mylib.a libc.a libgcc.a
```

リンク時にエラーは発生せず、mylib.aは正常にエミュレーションライブラリ、ANSIライブラリを使用できます。

追加したライブラリ同士で依存関係を持つ場合は、基本的なライブラリを最後に配置するようにしてください。

例: lib1.a　　エミュレーションライブラリ、ANSIライブラリのみを呼び出す
　　lib2.a　　エミュレーションライブラリ、ANSIライブラリに加え、 lib1.aを呼び出す
　　lib3.a　　エミュレーションライブラリ、ANSIライブラリに加え、 lib1.a、lib2.aを呼び出す

```
ld.exe -T withmylib.lds -o withmylib.elf boot.o lib3.a lib2.a lib1.a libc.a libgcc.a
```

ワークベンチgwb33が出力するmakeファイルには、ライブラリの指定を行うLIBSマクロがあります。このマクロを編集することでもライブラリの追加は可能です。

例: c:\myapp\lib\mylib.aを追加

```
LIBS= $(LIB_DIR)/libc.a $(LIB_DIR)/libgcc.a
                        ↓
LIBS= c:/myapp/lib/mylib.a $(LIB_DIR)/libc.a $(LIB_DIR)/libgcc.a
```

注: "\" は "/" に変換してください。LIBSマクロ内でワークベンチが認識、置換できるマクロ名はTOOL_DIRとLIB_DIRのみです。ワークベンチのリンカスクリプトを使用する場合、これ以外のマクロはLIBSマクロ内で使用しないでください。

LIBSマクロを変更した場合、makeファイルとリンカスクリプトファイル(*.lds)の同期が取れなくなりますので、必ずワークベンチgwb33上のリンカスクリプトエディタから更新操作を行ってください。更新するには、リンカスクリプトエディタを呼び出して[OK]ボタンをクリックします。これにより、関連するファイル(lds、ldt)が上書きされます。手動で変更したリンカスクリプトファイルの内容は失われますので注意してください。

2.9　S5U1C33001CとS5U1C33000Cとの違い

S5U1C33001CはGNUをベースとした共通インタフェースを持ち、従来の開発環境S5U1C33000Cに比べより柔軟かつ効率的な開発を行うことができます。
GNU共通のインタフェースを使用することにより、他の開発環境でのアセンブラやリンカ等の開発手法をそのまま使用することができます。Linux上での開発も可能です(Linux版は動作保証対象外となりますが、ツールのコンパイルにはWindows版と同じソースを使用しています)。
S5U1C33001Cは、アドバンスドマクロCPU専用の命令を扱うことができ、強力なアセンブラプログラミングが可能になっています。アドバンスドマクロCPUでは、事実上4Gバイトのメモリ空間をS5U1C33000Cで生成したコードと同じ命令数でアクセス可能となります。S5U1C33000Cではデータエリアポインタから64Mバイトまでのメモリエリアにしかアクセスできませんでしたが、S5U1C33001Cでは複数のデータエリアポインタを導入しているため、スタンダードマクロCPUでも最大で64×4＝256Mバイトの空間にアクセスできます。

●S5U1C33001Cの強化機能リスト

コンパイラ

- データエリアポインタの追加により、ユーザの多様な要求に対応
  - S5U1C33001Cでは-mdp=6(最大)で%r10～%r15まで、-mdp=4(推奨、デフォルト)で%r12～%r15まで使用可能(S5U1C33000Cでは-gpオプションにより%r8のみ使用可能)
  - データエリアポインタの拡張オプションにより、それぞれ64MBまでアクセス可能(__gdpを除く)
    最大で 64MB×4箇所、256MBまでのメモリ空間にアクセス可能(従来は64MB)
    アドバンスドマクロ対応モードでは 32ビット空間(4GB)すべてにアクセス可能
  - オプション指定によりS5U1C33000Cの2パスmakeに近いコード効率でコンパイル
- レジスタアサインの変更でROSの高速ディスパッチが可能
- 最適化ルーチンの強化(開発ベースgccバージョンのアップ)
- 多くの GNUフリーソースが流用可能
- Cで割り込みハンドラを記述可能。割り込み時のレジスタ退避ルーチンも自動生成

Binutils

- GNUベースのbinutilsが使用可能
  - アセンブラ、リンカ、make、objdump、objcopy等のGNU本来の多数のオプション機能が使用可能(S5U1C33000Cでは、Hex33(objcopy)、dis33(objdump)、dmp33(objdump)等専用ツールを使用)
- 簡潔な拡張命令を採用
  - コンパイラ出力コードが高速
  - アセンブラとソースの対応が理解しやすい
  - S5U1C33000Cのリンカコマンドver.3と同等の機能がS5U1C33001Cのリンカスクリプトで指定可能。また、多様な機能が使用可能

ワークベンチ

- ワークベンチにより視覚的にファイルの配置が可能

デバッガ

- S5U1C33000H(ICD ver2.0)に対応
  - db33でサポートしていたソフト/ハードブレーク、実行トレース、データブレークを完全サポート
  - c33 XXX形式のコマンドで従来のdb33コマンドに一部対応
- S5U1C33001H(ICD ver3.0)に完全対応
  - USB通信によりターゲットプログラムの高速ダウンロードが可能
  - エリアブレークが使用可能(アドバンスド対応CPUのみ)
  - バスブレーク、バストレースが使用可能(アドバンスド対応CPUのみ)
  - S5U1C33001H(ICD ver3.0)のフラッシュライタモードに対応(デバッガ単体でFlashライタとして使用可能)
  - トレースで扱える情報量が倍増
  - ターゲットがアドバンスドマクロの場合、エリアブレーク、バスブレーク、バストレース機能を実行可能

- スタンダード、アドバンスドマクロ両モード対応シミュレータを実装
- 式を利用したシンボルウォッチが可能
  - db33では不可能だった構造体メンバの内容表示、数式を利用したメモリシンボルの参照、ローカル変数のウォッチが可能
- デバッグ用ステップ系コマンドの充実
  - finishコマンドの実装により関数終了時点までのステップが可能
- elfフォーマット採用による容易なデバッグ情報抽出
  - ファイル形式にelfを採用しているためバイナリから情報の取得が容易(objdump.exeにより可能)
- GNU gdbと共通インタフェースを採用
  - S5U1C33001Cのツール知識が十分でなくてもデバッガの使用が可能
- コマンドファイル内で簡単なプログラムを作成可能
- テキストモードの実装(保証対象外)
  - --nwオプションによりCUIモードでデバッグが可能
- S5U1C33104H(ICE33)は非サポート
- 容易なソースレベルデバッグ
  - 内部RAMに転送されたプログラムも容易にデバッグ可能

ツール全般

- ソフトウェア(GNUツール)は、多くのプログラマにより常にアップデートやバグの修正が行われメンテナンスが用意
- 組み込み用途のソフトウェアの多くがGNUツールをベースに開発
  - Toppers、TINET、Linux、Husion TCP/IP等
- S5U1C33000Cツールの知識がなくても、GNUツールの知識である程度ツールが使用可能
  - WEB上のGNUツールのQ&Aを参考として利用可能
- 全ソースを提供。カスタマイズが可能(サポート、保証対象外)

表2.9.1 ツール対応表

機能	S1C33 Familyソフトウェアツール		S1C33 Familyハードウェアツール			
	S5U1C33000C	S5U1C33001C	S5U1C33000H ver.1	S5U1C33000H ver.2	S5U1C33000H ver.3	S5U1C330M1D1
スタンダードコア対応	○	○	○	○	○	○
アドバンスドコア対応	×	○	×	×	○	×
シリアル接続	○	○	○	○	×	○
パラレル接続	○	○	○	○	×	×
USB**接続**	×	○	×	×	○	×
CPU**クロック上限**			50MHz	60MHz	60MHz	60MHz
デバッガのトレース機能			○	○	○	×
デバッガのバスブレーク機能		○	×	×	○	×
デバッガのバストレース機能		○	×	×	○ *1	×
デバッガのエリアブレーク機能		○	×	×	○ *1	×
デバッガの強制ブレーク機能			○	○	○ *1	×
デバッグモニタ(S5U1C331M2S)			×	×	×	○ *2
ハードツールのリセット機能			×	×	○	○
S5U1C33000H(ICD33 ver.2)	○	○				
S5U1C33001H(ICD33 ver.3)	×	○				
S5U1C330M1D1	○	○				

○: 対応　×: 未対応
***1 アドバンスドコア対応チップのみ可能**
***2 S5U1C330M1D1は、専用のデバッグモニタ(S5U1C331M2S)が必要です。**

S5U1C33000Cと比べた場合、S5U1C33001Cを使用することにより、最大で25%程度パフォーマンスが向上します。
下記のソースを各ツールでコンパイルして実行した場合の実行時間は次のとおりです。
S5U1C33000C: 18926.80μs
S5U1C33001C: 14255.55μs

上記のとおり、S5U1C33001Cでは約3/4の実行時間となり、25%程度実行速度が上がっています。
また、コードサイズもS5U1C33001C の方が若干小さくなります。
S5U1C33000C: 274バイト（2パスmake後）
S5U1C33001C: 268バイト

条件

CPU クロック:　20MHz
外部RAMアクセス: 1ウェイト
コンパイル:　S5U1C33000Cは2パスmakeでコンパイル
S5U1C33001Cは"-O2 -mdp=1 -mgda=8192 -fno-builtin -mno-memopy"を指定

評価に使用したソースファイル

```
#define MAT_ARG1         3
#define MAT_ARG2         3

#define MAT_ARRAYSIZE    50
struct stMatrix {
    long          lData[ MAT_ARG1 ][ MAT_ARG2 ];
};

// proto-types
void vfnInitMatrix( struct stMatrix *stClear );
void vfnMuliMatrix( struct stMatrix *stRet,
            struct stMatrix *stLeft,
            struct stMatrix *stRight );

struct stMatrix stTestData1[ MAT_ARRAYSIZE ], stTestData2[ MAT_ARRAYSIZE ];
struct stMatrix stRet[ MAT_ARRAYSIZE ];

int main()
    {
        int nCnt, nCnt2, nCnt3;

        // initialize matrix
        for( nCnt = 0 ; nCnt < MAT_ARRAYSIZE ; nCnt++ ) {
            vfnInitMatrix( &stTestData2[ nCnt ] );
        }

        // set each randam value
        for( nCnt = 0 ; nCnt < MAT_ARRAYSIZE ; nCnt++ ) {

            for( nCnt2 = 0 ; nCnt2< MAT_ARG1; nCnt2++ ) {
                for( nCnt3 = 0 ; nCnt3 < MAT_ARG2; nCnt3++ ) {
                    stTestData1[ nCnt ].lData[ nCnt2 ][ nCnt3 ] = nCnt + nCnt2 + nCnt3;
                }
            }
        }

        // mutiply
        for( nCnt = 0 ; nCnt < MAT_ARRAYSIZE ; nCnt++ ) {
            vfnMuliMatrix( &stRet[ nCnt ],
                     &stTestData1[ nCnt ], &stTestData2[ nCnt ]  );
        }

    return 0;
}

void vfnInitMatrix( struct stMatrix *stClear )
{
    int nCnt, nCnt2;
    long lBuf;

    for( nCnt = 0 ; nCnt< MAT_ARG1; nCnt++ ) {
        for( nCnt2 = 0 ; nCnt2 < MAT_ARG2; nCnt2++ ) {
            if ( nCnt == nCnt2 )
                lBuf = 1;
            else
```

```
                lBuf = 0;
            stClear->lData[ nCnt ][ nCnt2 ] = lBuf;
        }
    }
}
void vfnMuliMatrix( struct stMatrix *stRet,
            struct stMatrix *stLeft,
            struct stMatrix *stRight )
{
    int nCnt, nCnt2, nCnt3;
    long lBuf;

    // multiple each matrix
    for( nCnt = 0; nCnt < MAT_ARG1 ; nCnt++ ) {         //y
        for( nCnt2 = 0; nCnt2 < MAT_ARG2 ; nCnt2++ ) {      //x

            lBuf= 0;
            for ( nCnt3 = 0 ; nCnt3 < MAT_ARG1 ; nCnt3++ ) {
                lBuf +=  ( stLeft->lData[ nCnt ][ nCnt3 ] )
                        *
                         ( stLeft->lData[ nCnt3 ][ nCnt2 ] );
            }
            stRet->lData[ nCnt ][ nCnt2 ] = lBuf;
        }
    }
}
```

上記の内容は一例であり、すべてのケースにおいてS5U1C33000Cよりも大幅なパフォーマンスの向上が見込めるわけではありません。

2.10 S5U1C33000C資産の移植方法

ここではS5U1C33000Cの資産をS5U1C33001Cへ移植する方法を説明します。
S5U1C33000CパッケージとS5U1C33001Cパッケージとの相違点を表2.10.1に示します。

表2.10.1 S5U1C33000CパッケージとS5U1C33001Cパッケージとの相違点

ツール	S5U1C33000C	S5U1C33001C
make	**make.exe** (S1C33**オリジナル**)	**make.exe** (GNU) **全機能を使用するには環境パス設定が必要**
コンパイラ	**gcc33.exe** (GNU ANSI C)	**xgcc.exe** (GNU ANSI C) • S1C33**用擬似命令** • **データエリア対応** • **アドバンスドマクロ命令対応**
プリプロセッサ	**pp33.exe** (S1C33**オリジナル**)	**cpp.exe** (GNU)
アセンブラ	**as33.exe** (S1C33**オリジナル**) • **グローバルポインタ対応**(GP) • S1C33**オリジナル擬似命令**	**as.exe** (GNU) • **データエリア対応擬似オペランド** • S1C33**用擬似命令** • **アドバンスドマクロ命令対応** • **拡張命令の展開**
命令エクステンダ	**ext33.exe** (S1C33**オリジナル**) • **拡張命令の展開** • **2パスmake**	**拡張命令の見直しにより命令エクステンダを廃止(2パスmake機能も廃止)**
リンカ	**lk33.exe** (S1C33**オリジナル**)	**ld.exe** (GNU) • **データエリア対応**
リンカマッピング	**コマンドファイル** (cm)	**リンカスクリプトファイル** (lds)
ANSI**ライブラリ**	io.lib, lib.lib, math.lib, string.lib, ctype.lib	libc.a
エミュレーション ライブラリ	fp.lib, idiv.lib, fpp.lib	libgcc.a, libgccP.a
ライブラリアン	**lib33.exe** (S1C33**オリジナル**)	**ar.exe** (GNU)
デバッガ	**db33.exe** (S1C33**オリジナル**) SIM ICE33 ICD33 Ver. 2.x MON33 MEM33	**gdb.exe** (S1C33**オリジナル**) SIM ICD33 Ver. 2.x, Ver. 3**以降** MON33
HEXコンバータ	**hex33.exe** (S1C33**オリジナル**)	**objcopy.exe** (GNU) **objcopy.exeは多目的に使用可能**
逆アセンブル	**dis33.exe** (S1C33**オリジナル**)	**objdump.exe** (GNU) **objdump -S -dで可能**
ダンプツール	**dmp33.exe** (S1C33**オリジナル**)	**objdump.exe** (GNU) **objdumpのオプション指定により可能**
置き換えツール	**sed.exe**	**sed.exe**
ワークベンチ	**wb33.exe** (S1C33**オリジナル**)	**gwb33.exe** (S1C33**オリジナル**) **リンカ入力ファイルを編集可能**

移植時に変更する必要があるファイルは以下のとおりです。
- メイクファイル(*.mak)
- 初期処理
- Cソースファイル(*.c)
- アセンブリソースファイル(*.s)
- リンカコマンドファイル(*.cm)
- デバッガのパラメータファイル(*.par)

2.10.1 makeファイル(*.mak)の移植

S5U1C33001Cではほぼすべてのツールが変更になりますので、makeファイル内のツール関係の記述箇所を変更する必要があります。

- ディレクトリパス記述　　\ → /(スラッシュ)
- ツールディレクトリ　　C:\cc33 → C:/gnu33
- コンパイラ名　　gcc33 → xgcc
- アセンブラ名　　as33 → as
- リンカ名　　lk33 → ld
- ライブラリアン名　　lib33 → ar
- make　　2パスmakeをしていた場合、削除してください。
- pp33、ext33　　記述を削除してください。
- 各ツールのオプションフラグ　　S5U1C33001C仕様に変更してください。

●メイクファイルの変更例

S5U1C33000C用makeファイル

```
TOOL_DIR = C:¥cc33
GCC33 = $(TOOL_DIR)¥gcc33
PP33  = $(TOOL_DIR)¥pp33
EXT33 = $(TOOL_DIR)¥ext33
AS33  = $(TOOL_DIR)¥as33
LK33  = $(TOOL_DIR)¥lk33
LIB33 = $(TOOL_DIR)¥lib33
MAKE  = $(TOOL_DIR)¥make
SRC_DIR =

# macro definitions for tool flags

GCC33_FLAG = -B$(TOOL_DIR)¥ -S -g -O
PP33_FLAG  = -g
EXT33_FLAG =
AS33_FLAG  = -g
LK33_FLAG  = -g -s -m -c
EXT33_CMX_FLAG = -lk clock -c

# dependency list

test.srf : test.cm boot.o main.o
       $(LK33) $(LK33_FLAG) test.cm

boot.ms : $(SRC_DIR)boot.s
       $(PP33) $(PP33_FLAG) $(SRC_DIR)boot.s
       $(EXT33) $(EXT33_FLAG) boot.ps
boot.o : boot.ms
       $(AS33) $(AS33_FLAG) boot.ms

main.ms : $(SRC_DIR)main.c
       $(GCC33) $(GCC33_FLAG) $(SRC_DIR)main.c
       $(EXT33) $(EXT33_FLAG) main.ps
main.o : main.ms
       $(AS33) $(AS33_FLAG) main.ms

# clean files except source

clean:
       del *.srf
       del *.o
       del *.ms
       del *.ps
```

S5U1C33001C用makeファイル

```
TOOL_DIR = C:/gnu33
CC = $(TOOL_DIR)/xgcc
AS = $(TOOL_DIR)/as
LD = $(TOOL_DIR)/ld
RM = $(TOOL_DIR)/rm
SRC_DIR =

# macro definitions for tool flags

CFLAGS= -B$(TOOL_DIR)/ -c -gstabs -O -mgda=0 -mdp=1 -mlong-calls -I$(TOOL_DIR)/
        include -fno-builtin
ASFLAGS = -B$(TOOL_DIR)/ -xassembler-with-cpp -Wa,--gstabs
LDFLAGS = -Tsample.lds -N -Map test.map

# macro definitions for library files

LIBS = $(LIB_DIR)/libc.a $(LIB_DIR)/libgcc.a

# dependency list

test.elf : $(OBJS)
       $(LD) $(LFLAGS) -o test.elf $(OBJS) $(LIBS)

boot.o : $(SRC_DIR)boot.s
       $(AS) $(AFLAGS) -o boot.o boot.s

main.o : $(SRC_DIR)main.c
       $(CC) $(CFLAGS) main.c

# clean files except source

clean:
       $(RM) *.o
       $(RM) *.elf
```

2.10.2 初期処理

ブート処理でR8(グローバルポインタ)の設定を行っている命令は、データエリアポインタの設定に変更してください。
通常は、デフォルトデータエリアポインタのアドレスをR15に設定します。G、S、T、Zデータエリアを新たに使用する場合は、R12、R13、R14も設定します。これらのレジスタは、以下のようにデータエリアシンボル名を使用して初期化することが可能です。

```
xld.w  %r12,__sdp  ;set S data area pointer
xld.w  %r13,__tdp  ;set T data area pointer
xld.w  %r14,__zdp  ;set Z data area pointer
xld.w  %r15,__dp   ;set default data area pointer
```

2.10.3 Cソースファイル(.c)の移植*

gcc33(S5U1C33000C)、xgcc(S5U1C33001C)はともにANSI C準拠のCコンパイラですので、基本的な文法は同じです。ただし、xgccには起動オプションや予約語など、いくつかの項目が追加されています。

- 4バイト長以下のグローバル変数などは、デフォルトでGデータエリアへ配置されます。
  Gデータエリアは上限サイズが8KBに固定されていますが、エリア内のアクセスは2命令で済みます(他は3命令)。プログラムコードのサイズを小さくするため、できるだけ多くの変数がGデータエリアに入るよう、-mgdaオプションでGデータエリアに配置するデータのサイズを設定してください。
  例: int変数 barをGデータエリアへ配置します。

```
int bar;
- - - - - - - - -
>xgcc -S -O -mgda=4 sample.c
```

- データエリアを変更したい場合は、予約語__attribute__を使用してください。
  例: int変数 barをSデータエリアへ配置します。

```
int __attribute__((sda)) bar;
```

• 割り込み処理関数
1）S5U1C33000Cでは、インタラプトフィルタユーティリティ(c33\utility\filter_int)を使用して割り込み処理関数を作成します。
S5U1C33001Cでは、Cソース内で__attribute__ ((interrupt_handler))による関数のプロトタイプ宣言を行うことによって割り込み処理関数に設定できます。
例: 関数fooを割り込み関数とする場合は、次のプロトタイプ宣言を行います。

```
void foo(void) __attribute__ ((interrupt_handler));
```

2）割り込み関数内でasm("reti");を使用している部分も、asm文を削除して、__attribute__ ((interrupt_handler))を使用したプロトタイプ宣言に置き代えることができます。

2.10.4 アセンブリソースファイル(.s)の移植*

●レジスタ

S5U1C33000CとS5U1C33001Cによるレジスタ用途の違いを表2.10.4.1に示します。
アセンブリソースはその違いを修正する必要があります。ただし、pushn %rsやpopn %rsなどは、単純にレジスタ番号を変更しただけでは意味が異なってしまうため、注意してください。

表2.10.4.1 レジスタ用途の相違

レジスタ	S5U1C33000C	S5U1C33001C
R0 R1 R2 R3	**関数呼び出し時に値を保存する必要のあるレジスタ**	**関数呼び出し時に値を保存する必要のあるレジスタ**
R4 R5	**スクラッチレジスタ**	**戻り数格納用レジスタ** **戻り数格納用レジスタ**
R6 R7		**引数渡し用レジスタ(第1ワード)** **引数渡し用レジスタ(第2ワード)**
R8	**グローバルポインタ**	**引数渡し用レジスタ(第3ワード)**
R9	ext33**で使用**	**引数渡し用レジスタ(第4ワード)**
R10 R11	**戻り数格納用レジスタ** **戻り数格納用レジスタ**	**スクラッチレジスタ/未使用** **スクラッチレジスタ/未使用**
R12 R13 R14 R15	**引数渡し用レジスタ(第1ワード)** **引数渡し用レジスタ(第2ワード)** **引数渡し用レジスタ(第3ワード)** **引数渡し用レジスタ(第4ワード)**	**Sデータエリアポインタレジスタまたはスクラッチレジスタ** **Tデータエリアポインタレジスタまたはスクラッチレジスタ** **Zデータエリアポインタレジスタまたはスクラッチレジスタ** **デフォルトデータエリアポインタレジスタ**

●コメント

プリプロセッサcppを通すアセンブリソースファイル内のコメントは、";"を"//"または"/* ... */"に変更してください。";"で始まり引用符のみを含むコメントは、cppでエラーとなります。

S5U1C33000C用のソース

```
.code
.ascii "ABCD"              ← OK
ld.w   %r0,1  ;'           ← cppでエラー
ld.w   %r1,2  ;'A'         ← OK
ld.w   %r2,3  ;"ABC"       ← OK
ld.w   %r3,4  ;"           ← cppでエラー
```

S5U1C33001C用のソース

```
.text
.ascii "ABCD"              ← OK
ld.w   %r0,1  //'          ← OK
ld.w   %r1,2  //'A'        ← OK
ld.w   %r2,3  /*"ABC"*/    ← OK
ld.w   %r3,4  /*"    */    ← OK
```

●セクション定義擬似命令

アセンブラas33とasのセクション擬似命令の違いを表2.10.4.2に示します。

表2.10.4.2 セクション擬似命令

as33 (S5U1C33000C)	**as** (S5U1C33001C)
.code	.text
.data	.data
.comm	.global + .bss, .*x*bss (*x* = g, s, t, z)
.lcomm	.bss, .*x*bss (*x* = g, s, t, z)

1) .code

.textセクションに変更してください。

2) .data

そのまま使用できます。

3) .comm

.comm擬似命令で定義されている初期値を持たないグローバル変数は、.globalおよび.bss(.*X*bss)擬似命令による定義に変更してください。

例:

as33(S5U1C33000C)

```
      .comm    symbol 4
```

as(S5U1C33001C)

```
      .section .bss
      .global  symbol
      .align   2
  symbol:
      .skip    4
```

変更時は、シンボルのアライメントの違いに注意してください。

アセンブラas33では、シンボルはデータのサイズに従った境界アドレスにアライメントされます。

アセンブラasでは、シンボルのアライメントを上記例のように.align擬似命令で指定する必要があります。

asの擬似命令".comm symbol, サイズ"に変更しても移植は可能ですが、変更時は以下の点に注意してください。

シンボルサイズによるアライメントの違い

as33(S5U1C33000C)

サイズ	アラインメント(バイト数)
1	1
2	2
3以上	4

as(S5U1C33001C)

サイズ	アラインメント(バイト数)
1	1
2~3	2
4~7	4
8~15	8
16以上	16

リンク後のシンボルの配置順

as33の場合、アセンブリソースに記述した順序でメモリに配置されますが、asでは必ずしも記述順にはなりません。

例: .commセクションを0x10000番地から配置した場合

(アセンブリソース)

```
    .comm  data1 4
    .comm  data2 4
    .comm  data3 4
```

（リンク後のメモリ配置）

as33(S5U1C33000C)

```
0x10000: data1
0x10004: data2
0x10008: data3
```

as(S5U1C33001C)

```
0x10000: data3        ← ソースファイルの
0x10004: data1        ← 記述順では
0x10008: data2        ← ありません
```

シンボルのメモリ配置がソースの記述順であることを前提に作成されているプログラムの場合は、正確に移植するためにも、前述のとおり.bssセクションを宣言するように変更してください。

4) .lcomm

.lcomm擬似命令で定義されている初期値を持たないローカル変数は、.bss(.*X*bss)擬似命令による定義に変更してください。

例:

as33(S5U1C33000C)

```
    .lcomm    symbol 4
```

as(S5U1C33001C)

```
    .section .bss
    .align   2
symbol:
    .skip    4
```

.comm擬似命令の変更と同様に、シンボルのアライメントに注意してください。

●データ定義擬似命令

データ定義擬似命令は、アセンブラas33とasでサイズの異なるものがありますので変更してください。表2.10.4.3にデータ定義擬似命令のサイズの比較を示します。

表2.10.4.3 データ定義擬似命令

as33 (S5U1C33000C)	**as** (S5U1C33001C)	サイズ
.byte	.byte	1**バイト**
.half	.short	2**バイト**
	.hword	2**バイト**
	.word	2**バイト**
.word	.int	4**バイト**
	.long	4**バイト**

変更例:

as33(S5U1C33000C)

```
        .code
        .word  boot          ;vector address
boot:
        xld.w  %r15,0x0800
        ld.w   %sp,%r15
```

as(S5U1C33001C)

```
        .text
        .long  boot          ;vector address
boot:
        xld.w  %r15,0x0800
        ld.w   %sp,%r15
```

アセンブラas33の.word擬似命令で定義されるデータは4バイト長ですが、asでは2バイト長です。.word擬似命令は、.intまたは.long擬似命令に変更してください。

●デバッグ擬似命令

アセンブラas33のデバッグ擬似命令.endfile、.loc、.defは、asでは無効ですので削除してください。デバッグ情報を付加する場合は、アセンブラasの--gstabsオプションを指定し、再度アセンブルしてください。

●拡張命令

アセンブラasは、命令エクステンダext33(S5U1C33000C)のすべての拡張命令をサポートしている訳ではありません。使用できない拡張命令のコード部分は、基本命令と使用可能な拡張命令を組み合わせて作成し直す必要があります。

表2.10.4.4　ext33/as拡張命令差分表(スタンダードマクロ)

ext33 (S5U1C33000C)		**as** (S5U1C33001C)	
xadd	%rd,%rd,imm32	xadd	%rd,imm32
xsub	%rd,%rd,imm32	xsub	%rd,imm32
xadd	%sp,%sp,imm32	–	
xsub	%sp,%sp,imm32	–	
xadd	%rd,%rs,imm32	–	
xsub	%rd,%rs,imm32	–	
xadd	%rd,%sp,imm32	–	
xsub	%rd,%sp,imm32	–	
xadd	%rd,%rd,%sp	–	
xsub	%rd,%rd,%sp	–	
xadd	%sp,%sp,%rs	–	
xsub	%sp,%sp,%rs	–	
xcmp	%rd,sign32	xcmp	%rd,sign32
xcmp	%rd,%sp	–	
xcmp	%sp,%rs	–	
xand	%rd,%rd,sign32	xand	%rd,sign32
xoor	%rd,%rd,sign32	xoor	%rd,sign32
xxor	%rd,%rd,sign32	xxor	%rd,sign32
xand	%rd,%rs,sign32	–	
xoor	%rd,%rs,sign32	–	
xxor	%rd,%rs,sign32	–	
xnot	%rd,sign32	xnot	%rd,sign32
xsrl	%rd,%rs	–	
xsll	%rd,%rs	–	
xsra	%rd,%rs	–	
xsla	%rd,%rs	–	
xrr	%rd,%rs	–	
xrl	%rd,%rs	–	
xsrl	%rd,imm5	xsrl	%rd,imm5
xsll	%rd,imm5	xsll	%rd,imm5
xsra	%rd,imm5	xsra	%rd,imm5
xsla	%rd,imm5	xsla	%rd,imm5
xrr	%rd,imm5	xrr	%rd,imm5
xrl	%rd,imm5	xrl	%rd,imm5
xld.b	%rd,[%sp+imm32]	xld.b	%rd,[%sp+imm32]
xld.ub	%rd,[%sp+imm32]	xld.ub	%rd,[%sp+imm32]
xld.h	%rd,[%sp+imm32]	xld.h	%rd,[%sp+imm32]
xld.uh	%rd,[%sp+imm32]	xld.uh	%rd,[%sp+imm32]
xld.w	%rd,[%sp+imm32]	xld.w	%rd,[%sp+imm32]
xld.b	[%sp+imm32],%rs	xld.b	[%sp+imm32],%rs
xld.h	[%sp+imm32],%rs	xld.h	[%sp+imm32],%rs
xld.w	[%sp+imm32],%rs	xld.w	[%sp+imm32],%rs
xld.b	%rd,[symbol±imm32]	xld.b	%rd,[symbol+imm26]
xld.ub	%rd,[symbol±imm32]	xld.ub	%rd,[symbol+imm26]
xld.h	%rd,[symbol±imm32]	xld.h	%rd,[symbol+imm26]
xld.uh	%rd,[symbol±imm32]	xld.uh	%rd,[symbol+imm26]
xld.w	%rd,[symbol±imm32]	xld.w	%rd,[symbol+imm26]
xld.b	[symbol±imm32],%rs	xld.b	[symbol+imm26],%rs
xld.h	[symbol±imm32],%rs	xld.h	[symbol+imm26],%rs
xld.w	[symbol±imm32],%rs	xld.w	[symbol+imm26],%rs
xld.w	[symbol±imm32],%sp	–	

ext33 (S5U1C33000C)		**as** (S5U1C33001C)	
xld.b	%rd,[imm32]	–	
xld.ub	%rd,[imm32]	–	
xld.h	%rd,[imm32]	–	
xld.uh	%rd,[imm32]	–	
xld.w	%rd,[imm32]	–	
xld.b	[imm32],%rs	–	
xld.h	[imm32],%rs	–	
xld.w	[imm32],%rs	–	
xld.w	[imm32],%sp	–	
xld.b	%rd,[%rb+symbol±imm32]	–	
xld.ub	%rd,[%rb+symbol±imm32]	–	
xld.h	%rd,[%rb+symbol±imm32]	–	
xld.uh	%rd,[%rb+symbol±imm32]	–	
xld.w	%rd,[%rb+symbol±imm32]	–	
xld.b	[%rb+symbol±imm32],%rs	–	
xld.h	[%rb+symbol±imm32],%rs	–	
xld.w	[%rb+symbol±imm32],%rs	–	
xld.w	[%rb+symbol±imm32],%sp	–	
xld.b	%rd,[%rb+imm32]	xld.b	%rd,[%rb+imm26]
xld.ub	%rd,[%rb+imm32]	xld.ub	%rd,[%rb+imm26]
xld.h	%rd,[%rb+imm32]	xld.h	%rd,[%rb+imm26]
xld.uh	%rd,[%rb+imm32]	xld.uh	%rd,[%rb+imm26]
xld.w	%rd,[%rb+imm32]	xld.w	%rd,[%rb+imm26]
xld.b	[%rb+imm32],%rs	xld.b	[%rb+imm26],%rs
xld.h	[%rb+imm32],%rs	xld.h	[%rb+imm26],%rs
xld.w	[%rb+imm32],%rs	xld.w	[%rb+imm26],%rs
xld.w	[%rb+imm32],%sp	–	
xld.w	%rd,symbol±imm32	xld.w	%rd,symbol±imm32
xld.w	%rd,sign32	xld.w	%rd,sign32
xbtst	[symbol±imm32],imm3	xbtst	[symbol+imm26],imm3
xbclr	[symbol±imm32],imm3	xbclr	[symbol+imm26],imm3
xbset	[symbol±imm32],imm3	xbset	[symbol+imm26],imm3
xbnot	[symbol±imm32],imm3	xbnot	[symbol+imm26],imm3
xbtst	[imm32],imm3	–	
xbclr	[imm32],imm3	–	
xbset	[imm32],imm3	–	
xbnot	[imm32],imm3	–	
xbtst	[%rb+symbol±imm32],imm3	–	
xbclr	[%rb+symbol±imm32],imm3	–	
xbset	[%rb+symbol±imm32],imm3	–	
xbnot	[%rb+symbol±imm32],imm3	–	
xbtst	[%rb+imm32],imm3	xbtst	[%rb+imm26],imm3
xbclr	[%rb+imm32],imm3	xbclr	[%rb+imm26],imm3
xbset	[%rb+imm32],imm3	xbset	[%rb+imm26],imm3
xbnot	[%rb+imm32],imm3	xbnot	[%rb+imm26],imm3
xbtst	[%sp+imm32],imm3	–	
xbclr	[%sp+imm32],imm3	–	
xbset	[%sp+imm32],imm3	–	
xbnot	[%sp+imm32],imm3	–	
–		scall	label+imm22
–		scall.d	label+imm22
–		sjp	label+imm22
–		sjp.d	label+imm22
–		sjreq	label+imm22
–		sjreq.d	label+imm22
–		sjrne	label+imm22
–		sjrne.d	label+imm22
–		sjrgt	label+imm22
–		sjrgt.d	label+imm22
–		sjrge	label+imm22
–		sjrge.d	label+imm22
–		sjrlt	label+imm22
–		sjrlt.d	label+imm22

ext33 (S5U1C33000C)	**as** (S5U1C33001C)
–	sjrle label+imm22
–	sjrle.d label+imm22
–	sjrugt label+imm22
–	sjrugt.d label+imm22
–	sjruge label+imm22
–	sjruge.d label+imm22
–	sjrult label+imm22
–	sjrult.d label+imm22
–	sjrule label+imm22
–	sjrule.d label+imm22
–	scall sign22
–	scall.d sign22
–	sjp sign22
–	sjp.d sign22
–	sjreq sign22
–	sjreq.d sign22
–	sjrne sign22
–	sjrne.d sign22
–	sjrgt sign22
–	sjrgt.d sign22
–	sjrge sign22
–	sjrge.d sign22
–	sjrlt sign22
–	sjrlt.d sign22
–	sjrle sign22
–	sjrle.d sign22
–	sjrugt sign22
–	sjrugt.d sign22
–	sjruge sign22
–	sjruge.d sign22
–	sjrult sign22
–	sjrult.d sign22
–	sjrule sign22
–	sjrule.d sign22
xcall label+imm32	xcall label+imm32
xcall.d label+imm32	xcall.d label+imm32
xjp label+imm32	xjp label+imm32
xjp.d label+imm32	xjp.d label+imm32
xjreq label+imm32	xjreq label+imm32
xjreq.d label+imm32	xjreq.d label+imm32
xjrne label+imm32	xjrne label+imm32
xjrne.d label+imm32	xjrne.d label+imm32
xjrgt label+imm32	xjrgt label+imm32
xjrgt.d label+imm32	xjrgt.d label+imm32
xjrge label+imm32	xjrge label+imm32
xjrge.d label+imm32	xjrge.d label+imm32
xjrlt label+imm32	xjrlt label+imm32
xjrlt.d label+imm32	xjrlt.d label+imm32
xjrle label+imm32	xjrle label+imm32
xjrle.d label+imm32	xjrle.d label+imm32
xjrugt label+imm32	xjrugt label+imm32
xjrugt.d label+imm32	xjrugt.d label+imm32
xjruge label+imm32	xjruge label+imm32
xjruge.d label+imm32	xjruge.d label+imm32
xjrult label+imm32	xjrult label+imm32
xjrult.d label+imm32	xjrult.d label+imm32
xjrule label+imm32	xjrule label+imm32
xjrule.d label+imm32	xjrule.d label+imm32
xcall sign32	xcall sign32
xcall.d sign32	xcall.d sign32
xjp sign32	xjp sign32
xjp.d sign32	xjp.d sign32
xjreq sign32	xjreq sign32

ext33 (S5U1C33000C)	**as** (S5U1C33001C)
xjreq.d sign32	xjreq.d sign32
xjrne sign32	xjrne sign32
xjrne.d sign32	xjrne.d sign32
xjrgt sign32	xjrgt sign32
xjrgt.d sign32	xjrgt.d sign32
xjrge sign32	xjrge sign32
xjrge.d sign32	xjrge.d sign32
xjrlt sign32	xjrlt sign32
xjrlt.d sign32	xjrlt.d sign32
xjrle sign32	xjrle sign32
xjrle.d sign32	xjrle.d sign32
xjrugt sign32	xjrugt sign32
xjrugt.d sign32	xjrugt.d sign32
xjruge sign32	xjruge sign32
xjruge.d sign32	xjruge.d sign32
xjrult sign32	xjrult sign32
xjrult.d sign32	xjrult.d sign32
xjrule sign32	xjrule sign32
xjrule.d sign32	xjrule.d sign32

表2.10.4.5　ext33/as拡張命令差分表（アドバンスドマクロ）

ext33 (S5U1C33000C)	**as** (S5U1C33001C)
–	ald.b %rd,[symbol+imm19]
–	ald.ub %rd,[symbol+imm19]
–	ald.h %rd,[symbol+imm19]
–	ald.uh %rd,[symbol+imm19]
–	ald.w %rd,[symbol+imm19]
–	ald.b [symbol+imm19],%rs
–	ald.h [symbol+imm19],%rs
–	ald.w [symbol+imm19],%rs
–	xld.b %rd,[%dp+imm32]
–	xld.ub %rd,[%dp+imm32]
–	xld.h %rd,[%dp+imm32]
–	xld.uh %rd,[%dp+imm32]
–	xld.w %rd,[%dp+imm32]
–	xld.b [%dp+imm32],%rs
–	xld.h [%dp+imm32],%rs
–	xld.w [%dp+imm32],%rs

2.10.5 リンカコマンドファイル(.cm)の移植*

作成したオブジェクトのセクションを実メモリへ配置するために、S5U1C33001Cではリンカスクリプトファイルを作成して、リンカldの実行時に-Tオプションで指定します。
S5U1C33000C用のリンカコマンドファイル(*.cm)では、セクションの配置アドレスとリンクファイルの両方を指定しましたが、S5U1C33001Cの場合は配置アドレスをリンカスクリプトファイルで指定し、リンクファイルはリンカldのコマンドラインで指定します。
また、ライブラリも新しいファイルに置き代える必要があります。

表2.10.5.1 ライブラリ

as33 (S5U1C33000C)	**as** (S5U1C33001C)
io.lib, lib.lib, math.lib, string.lib, ctype.lib	libc.a
fp.lib, idiv.lib	libgcc.a
fpp.lib, idiv.lib	libgccP.a

●変更例

lk33用リンカコマンドファイルsample.cm(S5U1C33000C)

```
;Map set
-code 0x0601000           ; set relative code section start address
-bss  0x0000400           ; set relative bss  section start address

-code 0x0600000 {boot.o}  ; set code sections to absolute address

;Library path
-l ..¥..¥lib

;Executable file
-o ansilib.srf

;Object files
boot.o
main.o
sys.o
lib.o

;Library files
io.lib
lib.lib
math.lib
string.lib
ctype.lib
fp.lib
idiv.lib
```

ld用リンカスクリプトファイルsample.lds(S5U1C33001C)

```
OUTPUT_FORMAT("elf32-c33", "elf32-c33",
              "elf32-c33")
OUTPUT_ARCH(c33)
SEARCH_DIR(.);
SECTIONS
{
   __dp = 0x400;

   . = 0x400;
   .bss      : { *(.bss) }
   .data     : { *(.data) }

   . = 0x600000;
   .bootsec  : { boot.o (.text) }

   . = 0x0601000;
   .text     : { *(.text) }

}
```

リンカldのコマンドライン

```
ld -o sample.elf boot.o main.o sys.o lib.o ../lib/libc.a ../lib/libgcc.a -Tsample.lds
```

●lk33 Ver 3.0用リンカコマンドファイルの変更例

lk33用リンカコマンドファイルsample.cm（S5U1C33000C）

```
-v3

-defaddr IN_RAM=0x0
-defaddr EXT_ROM=0xc00000

-codeblock OBJ1
{
       sample2.o
       sample3.o
}

-ucodeblock OBJ2      ...共有ブロックの定義
{                       sample1.oとsample4.oのCODEセクションは同一アドレス
       sample1.o
       sample4.o
}

-addr IN_RAM
{
       DEFAULT_BSS
       @DEFAULT_DATA
       @OBJ1          ...OBJ1を仮想ブロックとして配置
       @OBJ2          ...OBJ2を仮想共有ブロックとして配置
}

-addr EXT_ROM
{
       DEFAULT_CODE
       DEFAULT_DATA
       OBJ1
       OBJ2
}
sample1.o
sample2.o
sample3.o
sample4.o
sample5.o
sample.lib
```

ld用リンカスクリプトファイルsample.lds（S5U1C33001C）

```
OUTPUT_FORMAT("elf32-c33", "elf32-c33",
              "elf32-c33")
OUTPUT_ARCH(c33)
SEARCH_DIR(.);
SECTIONS
{
       /* data pointer symbol By GWB33 */
       __dp = 0x0;

       /* section information By GWB33 */
       . = 0x0;

       .bss 0x00000000 :
         {
          __START_bss = . ;
          *(.bss) ;
          __END_bss = . ;
         }

       .text 0x00c00000 :
         {
          __START_text = . ;
          sample5.o(.text) ;
          C:/GNU33/lib/sample.a(.text);
          __END_text = . ;
         }

       .rodata __END_text :
         {
          __START_rodata = . ;
          *(.rodata) ;
```

```
    __END_rodata = . ;
  }

.data __END_bss : AT( __END_rodata )
  {
   __START_data = . ;
   *(.data) ;
   __END_data = . ;
  }
__START_data_lma = LOADADDR( .data );

.OBJ1 __END_data : AT( __START_data_lma+SIZEOF( .data ))
  {
   __START_OBJ1 = . ;
   sample2.o(.text)
   sample3.o(.text) ;
   __END_OBJ1 = . ;
  }
__START_OBJ1_lma = LOADADDR( .OBJ1 );

.OBJ2_1 __END_OBJ1 : AT( __START_OBJ1_lma+SIZEOF( .OBJ1 ))
  {
   __START_OBJ2_1 = . ;
   sample1.o(.text) ;
   __END_OBJ2_1 = . ;
  }
__START_OBJ2_1_lma = LOADADDR( .OBJ2_1 );

.OBJ2_2 __END_OBJ1 : AT( __START_OBJ2_1_lma+SIZEOF( .OBJ2_1 ))
  {
   __START_OBJ2_2 = . ;
   sample4.o(.text) ;
   __END_OBJ2_2 = . ;
  }
__START_OBJ2_2_lma = LOADADDR( .OBJ2_2 );
}
```

OBJ1の定義のように複数のオブジェクトファイルを1つのセクションに指定することで、lk33のブロック定義と同様になります。

リンカldによるsample.ldsのリンク結果（objdump -h sample.elf）

```
block.elf:     file format elf32-c33

Sections:
Idx Name       Size      VMA       LMA       File off  Algn
  0 .bss       00000014  00000000  00000000  00001000  2**2
               ALLOC
  1 .text      00000006  00c00000  00c00000  00001000  2**1
               CONTENTS, ALLOC, LOAD, READONLY, CODE
  2 .rodata    00000005  00c00006  00c00006  00001006  2**0
               CONTENTS, ALLOC, LOAD, DATA
  3 .data      00000014  00000014  00c0000b  00001014  2**2
               CONTENTS, ALLOC, LOAD, DATA
  4 .OBJ1      0000000c  00000028  00c0001f  00001028  2**1
               CONTENTS, ALLOC, LOAD, CODE
  5 .OBJ2_1    00000006  00000034  00c0002b  00001034  2**1
               CONTENTS, ALLOC, LOAD, CODE
  6 .OBJ2_2    00000006  00000034  00c00031  00002034  2**1
               CONTENTS, ALLOC, LOAD, CODE
  7 .stab      00000714  0000003c  0000003c  0000203c  2**2
               CONTENTS, READONLY, DEBUGGING
  8 .comment   000000be  00000b66  00000b66  00002750  2**0
               CONTENTS, READONLY
  9 .stabstr   00000416  00000750  00000750  0000280e  2**0
               CONTENTS, READONLY, DEBUGGING
```

OBJ2_1とOBJ2_2のVMAが同一アドレス（0x00000034）になっています。

2.10.6 デバッガのパラメータファイル(*.par)の移植

デバッガgdb(S5U1C33001C)もdb33(S5U1C33000C)と同じように、起動には基本的にパラメータファイルが必要です。ワークベンチgwb33(S5U1C33001C)もwb33(S5U1C33000C)と同様のパラメータファイルジェネレータを備えていますので、この機能を使って*.parファイルを作成し直してください。
gwb33の[PAR edit]ボタンでパラメータファイル作成ダイアログが表示されますので、必要な指定を行いパラメータファイルを作成してください。

2.10.7 srf33オブジェクトファイル(S5U1C33000C)と elfオブジェクトファイル(S5U1C33001C)の構造の違い

S5U1C33000Cパッケージのアセンブラas33およびリンカlk33により生成されるsrf33形式ファイルと、S5U1C33001Cパッケージのアセンブラasおよびリンカldにより生成されるelf形式ファイルの構造の違いを以下に示します。

●srf33形式ファイルの構造

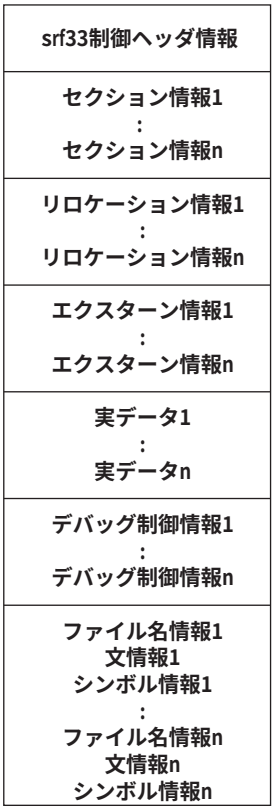


図2.10.7.1 srfファイル内の配置

セクション情報は、それぞれリロケーション情報、エクスターン情報、実データとリンクしています。
srf33オブジェクトファイルの詳細については、"S5U1C33000C Manual"のAppendixを参照してください。

●elf形式ファイルの構造

図2.10.7.2　elfファイル内の配置

elfヘッダ情報には、アドレスのサイズ、アーキテクチャの種類、ファイルバージョン、エントリポイントなどの情報が記述されています。
アセンブラ、リンカは、セクションヘッダテーブルが記述する論理セクションの集合としてelfファイルを扱います。これに対し、システムローダは、プログラムヘッダテーブルが記述するセグメントの集合としてelfファイルを扱います。

elfオブジェクトファイルの仕様の詳細については、Webサイトおよび関連書籍を参照してください。

上記に示したように、ヘッダ情報の内容および構造に互換性はありません。

2.11 S5U1C33001Cツール使用上の注意

●Cygwinの改行文字

Cygwinがインストールされているパーソナルコンピュータ上でS5U1C33001Cを使用する場合、改行文字の設定によってはツールが正常に動作しません。S5U1C33001Cが推奨する改行文字の設定はCR+LF（Windowsと同様）です。これ以外に設定されている場合はCygwinの再セットアップにより改行文字を変更してください。

●アセンブラ拡張命令の一部オフセット表示について

S5U1C33001Cのアセンブラasでは、拡張命令のオペランドの中に"SYMBOL+imm26"の表記があります。このimm26は、実際には32ビット値としてアセンブル可能となっています。ただし、リンク時にセットされる実際のオフセット値は次の条件式を満たしている必要があります。

imm26 + [SYMBOLのオフセット値] < 0x4000000

（SYMBOLのオフセット値はデータエリアポインタからSYMBOLまでの距離です。）

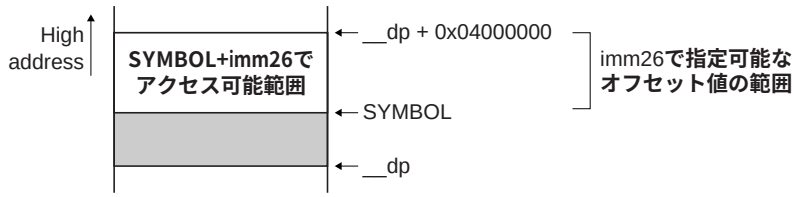
例:

```
xld.b  [SYMBOL+imm26],rd
```

この命令は以下の形式に展開されます。

```
ext    (SYMBOL+imm26)@ah     ...A = ext ((SYMBOL+imm26) - __dp)[25:13]
ext    (SYMBOL+imm26)@al     ...B = ext ((SYMBOL+imm26) - __dp)[12:0]
xld.b  [%r15], %rd
```

__dp < SYMBOLの場合



__dp > SYMBOLの場合 (Cからはアクセス不可)

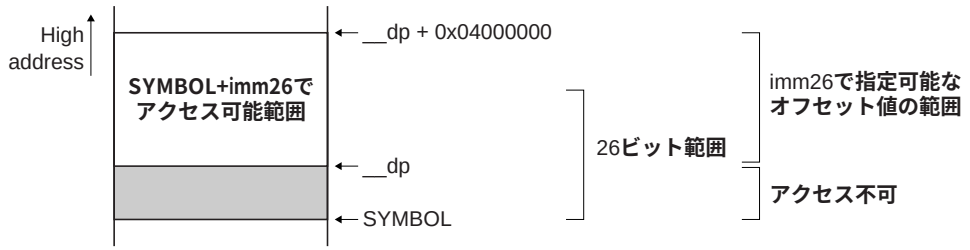


図2.11.1 SYMBOL+imm26指定可能範囲

図に示すとおり、上記命令のアクセス可能範囲の上限は__dp + 0x04000000です。
したがって、通常の__dp < SYMBOLの場合、imm26には"0"から"(__dp + 0x04000000) - SYMBOL"までの値が指定可能です。
__dp > SYMBOLの場合、26ビットの範囲を超えるオフセット値も指定可能となります。ただし、CソースからSYMBOLを利用したアドレスをアクセスすることはできません。アセンブリソースからは、SYMBOL+imm26が__dpから(__dp + 0x04000000)の範囲内であればアクセス可能です。

●GUIデバッグ用SHAREファイルについて

\gnu33\utility\shareフォルダ内にあるファイル群はTCL/TK言語で記述されたファイルで、デバッガgdbのGUIモードで使用されます。
これらはテキストファイルなので、エディタ等で変更も可能です。ただし、変更内容によってはgdbが動作不能になる場合がありますので注意が必要です。変更が必要な場合は、詳細な仕様を充分理解した上で行ってください。

これらのファイルの中には、セイコーエプソンがinsight gdbからS5U1C33001C用にカスタマイズしたファイルがいくつかあります。
\gnu33\utility\share\modifyフォルダ内にカスタマイズされたファイル名と変更内容を示したファイルがあります。

3 S1C33周辺機能のプログラミング

ここでは、S1C33チップの基本的な周辺機能のプログラムについて説明します。

注: 以下の周辺機能、プログラムは特に指定したものを除きS1C33209の例です。機種によっては、機能や制御レジスタアドレス等が異なる場合があります。

3.1 BCUの設定

次のサンプルプログラムでSRAM(ROM、FLASHも同様)とDRAMの設定方法について説明します。
これは、S1C33209のコアおよびバスが共に25MHz動作で、エリア10にSRAM、エリア13にDRAMが接続されている場合のBCU設定例です。

BCU設定例

```
void setbcu()
    {
       volatile short *ps0;
       volatile char *pc0;

// set bcu

       ps0 = (short *)0x48126;      // area 9-10 1 wait
       *ps0 = 0x01;

       ps0 = (short *)0x48122;      // area 13    dram                          (1)
       *ps0 = 0x82;                 // area 14    2 wait

       pc0 = (char *)0x4014d;       // pre-scaler fpr 8bit TM0                  (2)
       *pc0 = 0x09;                 //      1/4
       pc0 = (char *)0x40161;       // 8bit TM0 reload
       *pc0 = 0x7e;                 //      20us in 25MHz
       pc0 = (char *)0x40160;       // 8bit TM0
       *pc0 = 0x3;                  //      start

       ps0 = (short *)0x4812e;                                               (3)
       *ps0 = 0x06e0;               //  fast page, col=9bit, refresh enable, CBR,
       ps0 = (short *)0x48130;
       *ps0 = 0x208;                //  ras1/cas2, precharge1, cefunc=01
    }
```

● SRAM、ROM、FLASH用の設定

SRAM、ROM、FLASH用の設定は、アドレス 0x48120～0x4812BのBCUレジスタで以下のエリアごとに行えます。

設定エリア

18–17、16–15、14–13、12–11、10–9、8–7、6、5–4

設定内容

a)デバイスサイズ : 8または16ビット
(エリア6は、アドレスにより8/16ビットが切り換わります。)
b)ウェイト数: 0～7サイクル
(ライト時は、0に設定しても1以上となります。)
c)出力ディセーブル遅延時間: 0.5～3.5サイクル
(異なるエリア間のアクセス時に挿入されるウェイトサイクルです。)

この例ではエリア9–10を、デバイスサイズ16ビット、1ウェイト、出力ディセーブル遅延時間0.5に設定しています。

```
ps0 = (short *)0x48126;             // area 9-10 1 wait
*ps0 = 0x01;
```

なお、x8タイプのSRAM 2個で16ビット幅にしている場合はこれでかまいませんが、x16タイプのSRAMを使用している場合は外部インタフェース方式設定(0x4812EのD3)を 1の#BSLにする必要があります。2つのタイプの混在はできません。詳細は、"4 S1C33チップ用ボードの基本回路"内の"x16タイプSRAMの接続"で説明します。

● DRAM用の設定

エリア14、13、8、7をDRAMに設定可能です。

(1)DRAMの選択

DRAMを使用するエリアのDRAM選択ビットを1にします。この例では、エリア13を16ビット幅のDRAMとして設定しています。
エリア14は2ウェイト、16ビット幅のSRAM等のエリアとして使用できます。

```
ps0 = (short *)0x48122;             // area 13    dram
*ps0 = 0x82;                        // area 14    2 wait
```

(2)8ビットタイマ0によるDRAMリフレッシュの設定

この例では、まず8ビットタイマ0のクロック入力用プリスケーラを1/4モードにします。これで、25MHzの1/4分周のクロックが8ビットタイマ0に入力されます。
さらに、タイマリロード値として0x7eをセットします。これでタイマ入力クロックが125(0x7e+1)分周されますので、リフレッシュサイクルは元の動作クロック(25MHz)を500分周した20μsとなります。

```
pc0 = (char *)0x4014d;              // pre-scaler fpr 8bit TM0
*pc0 = 0x09;                        //     1/4
pc0 = (char *)0x40161;              // 8bit TM0 reload
*pc0 = 0x7e;                        //     20us in 25MHz
pc0 = (char *)0x40160;              // 8bit TM0
*pc0 = 0x3;                         //     start
```

(3)DRAMパラメータの設定

最後に、DRAMの詳細設定を行います。
なお、以下の設定は複数のエリアにDRAMが接続されている場合でも、接続されたDRAMすべてに反映されます。
アドレス0x4812Eで
1. EDO/高速ページモード
2. カラムサイズ(8〜11ビット)
3. リフレッシュイネーブル/ディセーブル
4. セルフ/CBRリフレッシュ
5. リフレッシュRPCディレイ(1、2)
6. リフレッシュRASパルス幅(2〜5)

が選択可能です。
さらに、アドレス0x48130で
7. 連続RASモード
8. RASプリチャージ数
9. CASサイクル数
10. RASサイクル数

を選択します。

この例では、高速ページモード、CBRリフレッシュ、RAS=1サイクル、CAS=2サイクル、プリチャージ=1サイクルを設定しています。

```
ps0 = (short *)0x4812e;
*ps0 = 0x06e0;              //  fast page, col=9bit, refresh enable, CBR,
ps0 = (short *)0x48130;
*ps0 = 0x208;               //  ras1/cas2, precharge1, cefunc=01
```

また、DRAMの場合は電源投入後に使用可能になるまでの時間や、ダミーサイクルが必要な場合があります。上記例のほかに、これらに対応したプログラムが必要です。

● BCLK、CEFUNC

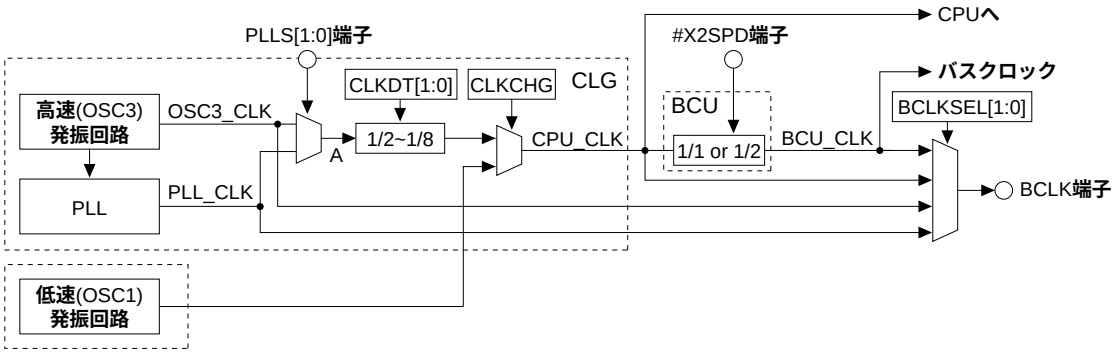
アドレス0x4812Eと0x48130には、BCUの特別な設定を行う制御ビットがあります。
この中でよく使用する2つの制御ビットを以下に説明します。

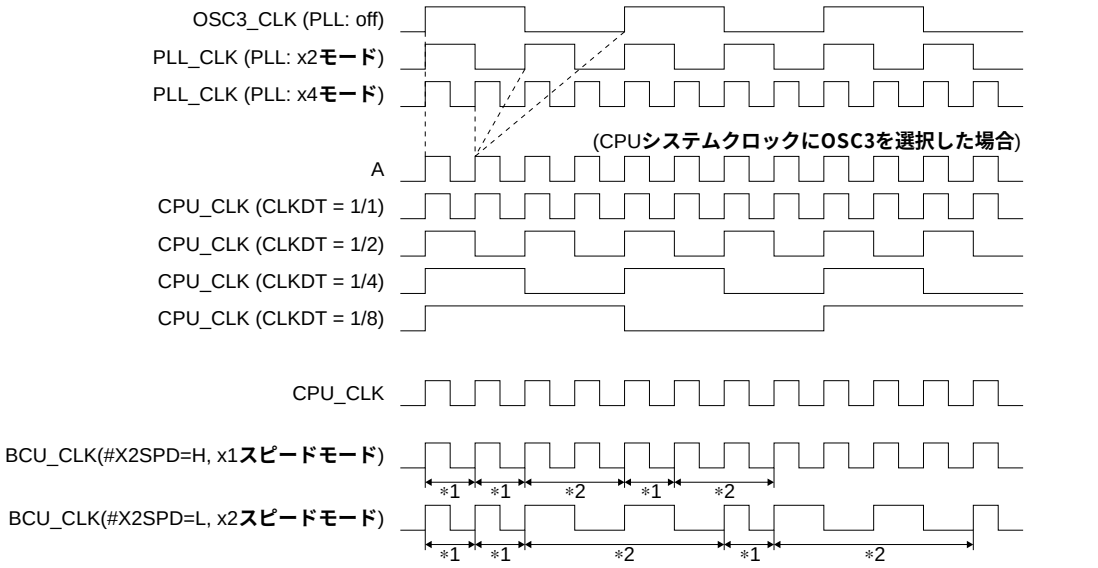
BCLK(0x4812E・DF): BCLK出力イネーブル

BCLK端子からのクロック出力を制御します。デフォルトは、出力(0)です。
ただし、この出力により数mAの電流を消費しますので、必要がなければH固定(1)としてください。

出力する場合、BCLK出力クロックはBCLKSEL[1:0](0x4813A・D[1:0])でPLL出力クロック、OSC3クロック、BCUクロック、CPUクロックから選択できます(デフォルトはCPUクロック)。

BCLKSEL1	BCLKSEL0	出力クロック
1	1	PLL_CLK (PLL**出力クロック**)
1	0	OSC3_CLK (OSC3**発振クロック**)
0	1	BCU_CLK (BCU**動作クロック**)
0	0	CPU_CLK (CPU**動作クロック**)





∗1 **内蔵RAMアクセス、またはA1X1MD = 1での内蔵周辺回路アクセス**
∗2 **外部アクセス、またはA1X1MD = 0での内蔵周辺回路アクセス**
(A1X1MD(0x4813A, D3)により、x2スピードモードでの内蔵周辺回路アクセスをCPUクロックの2サイクル、4サイクルから選択できます。)

クロック系(S1C33209)

CEFUNC[1:0](0x48130･D[A:9]): #CE端子機能選択

S1C33209には#CE端子が7本しかないため、全アドレス空間を同時に使用することはできません。
その代わりに、このCEFUNCの設定によって使用するエリアを選択可能です。

端子	CEFUNC = "00"	CEFUNC = "01"	CEFUNC = "1x"
#CE4	#CE4	#CE11	#CE11+#CE12
#CE5	#CE5	#CE15	#CE15+#CE16
#CE6	#CE6	#CE6	#CE7+#CE8
#CE7/#RAS0	#CE7/#RAS0	#CE13/#RAS2	#CE13/#RAS2
#CE8/#RAS1	#CE8/#RAS1	#CE14/#RAS3	#CE14/#RAS3
#CE9	#CE9	#CE17	#CE17+#CE18
#CE10EX	#CE10EX	#CE10EX	#CE9+#CE10EX

(**デフォルト**: CEFUNC = "00")

エリア	アドレス	
エリア9 SRAM**タイプ** **バースト**ROM**タイプ** 8 or 16**ビット**	0x0BFFFFF 0x0800000	**外部メモリ**(4MB)
エリア8 SRAM**タイプ** DRAM**タイプ** 8 or 16**ビット**	0x07FFFFF 0x0600000	**外部メモリ**(2MB)
エリア7 SRAM**タイプ** DRAM**タイプ** 8 or 16**ビット**	0x05FFFFF 0x0400000	**外部メモリ**(2MB)
エリア6 SRAM**タイプ**	0x03FFFFF 0x0380000	**外部**I/O (16**ビットデバイス**)
	0x037FFFF 0x0300000	**外部**I/O (8**ビットデバイス**)
エリア5 SRAM**タイプ** 8 or 16**ビット**	0x02FFFFF 0x0200000	**外部メモリ**(1MB)
エリア4 SRAM**タイプ** 8 or 16**ビット**	0x01FFFFF 0x0100000	**外部メモリ**(1MB)
エリア3 16**ビット** 1**サイクル固定**	0x00FFFFF 0x0080000	(Reserved) **ミドルウェア用**
エリア2 16**ビット** 3**サイクル固定**	0x007FFFF 0x0060000	(Reserved) CPU**コア/デバッグモード用**
エリア1 8, 16**ビット** 2 or 4**サイクル**	0x005FFFF 0x0050000 0x004FFFF 0x0040000 0x003FFFF 0x0030000	(**内蔵**I/O**メモリのミラー**) **内蔵**I/O**メモリ** (**内蔵**I/O**メモリのミラー**)
エリア0 32**ビット** 1**サイクル固定**	0x002FFFF 0x0000000	**内蔵**RAM

エリア	アドレス	
エリア18 SRAM**タイプ** 8 or 16**ビット**	0xFFFFFFF 0xD000000 0xCFFFFFF 0xC000000	**外部メモリ**(16MB)
エリア17 SRAM**タイプ** 8 or 16**ビット**	0xBFFFFFF 0x9000000 0x8FFFFFF 0x8000000	**外部メモリ**(16MB)
エリア16 SRAM**タイプ** 8 or 16**ビット**	0x7FFFFFF 0x7000000 0x6FFFFFF 0x6000000	**外部メモリ**(16MB)
エリア15 SRAM**タイプ** 8 or 16**ビット**	0x5FFFFFF 0x5000000 0x4FFFFFF 0x4000000	**外部メモリ**(16MB)
エリア14 SRAM**タイプ** DRAM**タイプ** 8 or 16**ビット**	0x3FFFFFF 0x3000000	**外部メモリ**(16MB)
エリア13 SRAM**タイプ** DRAM**タイプ** 8 or 16**ビット**	0x2FFFFFF 0x2000000	**外部メモリ**(16MB)
エリア12 SRAM**タイプ** 8 or 16**ビット**	0x1FFFFFF 0x1800000	**外部メモリ**(8MB)
エリア11 SRAM**タイプ** 8 or 16**ビット**	0x17FFFFF 0x1000000	**外部メモリ**(8MB)
エリア10 SRAM**タイプ** **バースト**ROM**タイプ** 8 or 16**ビット**	0x0FFFFFF 0x0C00000	**外部メモリ**(4MB)

S1C33のアドレス空間

エリア	アドレス	
エリア10(#CE10) SRAMタイプ バーストROMタイプ 8 or 16ビット	0x0FFFFFF 0x0C00000	外部メモリ6(4MB)
エリア9(#CE9) SRAMタイプ バーストROMタイプ 8 or 16ビット	0x0BFFFFF 0x0800000	外部メモリ5(4MB)
エリア8(#CE8/#RAS1) SRAMタイプ DRAMタイプ 8 or 16ビット	0x07FFFFF 0x0600000	外部メモリ4(2MB)
エリア7(#CE7/#RAS0) SRAMタイプ DRAMタイプ 8 or 16ビット	0x05FFFFF 0x0400000	外部メモリ3(2MB)
エリア6(#CE6) SRAMタイプ	0x03FFFFF 0x0380000 0x037FFFF 0x0300000	外部I/O (16ビットデバイス) 外部I/O (8ビットデバイス)
エリア5(#CE5) SRAMタイプ 8 or 16ビット	0x02FFFFF 0x0200000	外部メモリ2(1MB)
エリア4(#CE4) SRAMタイプ 8 or 16ビット	0x01FFFFF 0x0100000	外部メモリ1(1MB)

CEFUNC = "00"

エリア	アドレス	
エリア17(#CE17) SRAMタイプ 8 or 16ビット	0xBFFFFFF 0x9000000 0x8FFFFFF 0x8000000	(外部メモリ6のミラー) 外部メモリ6(16MB)
エリア15(#CE15) SRAMタイプ 8 or 16ビット	0x5FFFFFF 0x5000000 0x4FFFFFF 0x4000000	(外部メモリ5のミラー) 外部メモリ5(16MB)
エリア14(#CE14/#RAS3) SRAMタイプ DRAMタイプ 8 or 16ビット	0x3FFFFFF 0x3000000	外部メモリ4(16MB)
エリア13(#CE13/#RAS2) SRAMタイプ DRAMタイプ 8 or 16ビット	0x2FFFFFF 0x2000000	外部メモリ3(16MB)
エリア11(#CE11) SRAMタイプ 8 or 16ビット	0x17FFFFF 0x1000000	外部メモリ2(8MB)
エリア10(#CE10) SRAMタイプ バーストROMタイプ 8 or 16ビット	0x0FFFFFF 0x0C00000	外部メモリ1(4MB)
エリア6(#CE6) SRAMタイプ	0x03FFFFF 0x0380000 0x037FFFF 0x0300000	外部I/O (16ビットデバイス) 外部I/O (8ビットデバイス)

CEFUNC = "01"

エリア	アドレス	
エリア17+18 (#CE17+18) SRAM タイプ 8 or 16 ビット	0xFFFFFFF 0xD000000 0xCFFFFFF 0xC000000 0xBFFFFFF 0x9000000 0x8FFFFFF 0x8000000	(外部メモリ7'のミラー) 外部メモリ7'(16MB) (外部メモリ7のミラー) 外部メモリ7(16MB)
エリア15–16 (#CE15+16) SRAMタイプ 8 or 16ビット	0x7FFFFFF 0x7000000 0x6FFFFFF 0x6000000 0x5FFFFFF 0x5000000 0x4FFFFFF 0x4000000	(外部メモリ6'のミラー) 外部メモリ6'(16MB) (外部メモリ6のミラー) 外部メモリ6(16MB)
エリア14 (#CE14/#RAS3) SRAMタイプ DRAMタイプ 8 or 16ビット	0x3FFFFFF 0x3000000	外部メモリ5(16MB)
エリア13 (#CE13/#RAS2) SRAMタイプ DRAMタイプ 8 or 16ビット	0x2FFFFFF 0x2000000	外部メモリ4(16MB)
エリア11–12 (#CE11+12) SRAMタイプ 8 or 16 ビット	0x1FFFFFF 0x1000000	外部メモリ3(16MB)
エリア9–10 (#CE9+10EX) SRAMタイプ バーストROMタイプ 8 or 16ビット	0x0FFFFFF 0x0800000	外部メモリ2(8MB)
エリア7–8 (#CE7+8) SRAM タイプ 8 or 16ビット	0x07FFFFF 0x0400000	外部メモリ1(4MB)

CEFUNC = "10"または"11"

外部メモリ領域の選択

3.2　8ビットタイマの設定

周辺機能を使用するには大きく分けて4つの設定が必要です。

1. プリスケーラの設定
   各周辺機能の動作クロックは必ずプリスケーラで分周して供給されます。

2. 周辺機能自体の設定
   各周辺機能は動作モードを決定したり、スタート/ストップを制御するレジスタを持っています。

3. 割り込みコントローラの設定(割り込み使用時)
   各周辺機能で発生した割り込み要求は、必ず割り込みコントローラを経由してCPUコアに入ります。

4. 外部端子の設定(外部端子使用時)
   外部端子はデフォルトで汎用の入出力兼用ポート、入力ポートになっていますので、周辺機能用に使用する場合はレジスタを設定して切り換える必要があります。

まず、8ビットタイマを使用した簡単な割り込み制御プログラムを、S5U1C33001Cのsample\icdtrc\内のサンプルを使用して説明します。

● S5U1C33000Hトレース補助割り込みプログラム

このサンプルは、S5U1C33000Hのトレース機能を補強するためのプログラム例です。
S5U1C33000Hのトレース機能は、プログラム実行開始時などの起点となるPC値とデバッガが持つ逆アセンブル情報からプログラムフロー解析を行ってPC値を表示します。しかし、トレースをオーバーライトするモードでは起点のPC値が分かるとは限りません。そこで、このプログラムでは8ビットタイマで定期的に割り込みを発生させてPC値を確定し(reti実行時にPCの絶対値が出力されるため)、そこを起点としてそれ以降のS5U1C33000HによるPC解析を可能にしています。

ベクタ部

```
        .text
        .long BOOT              ; boot,rest VECTOR
        .long RESERVED          ; reserved 4
        .long RESERVED          ; reserved 8
        .long RESERVED          ; reserved 12
        .long EXP_DIV0          ; divided by 0 exception
        .long RESERVED          ; reserved 20
        .long EXP_UNADDR        ; address un-aligned exception
        .long NMI               ; nmi
        .long RESERVED          ; reserved 32
        .long RESERVED          ; reserved 36
        .long RESERVED          ; reserved 40
        .long RESERVED          ; reserved 44

        .long SOFT_INT          ; software interrupt 0
        .long SOFT_INT          ; software interrupt 1
        .long SOFT_INT          ; software interrupt 2
        .long SOFT_INT          ; software interrupt 3

        .long HARD_INT          ; hardware interrupt  0
        .long HARD_INT          ; hardware interrupt  1
                                        |
        .long HARD_INT          ; hardware interrupt  35
        .long TIME_INT          ; hardware interrupt  36                    (1)
                                      ;set 8 bit timer ch0 interrupt vector
        .long HARD_INT          ; hardware interrupt  37
        .long HARD_INT          ; hardware interrupt  38
```

(1)　ベクタテーブルの設定
割り込みルーチンをベクタテーブルに登録します。

```
        .long HARD_INT          ; hardware interrupt  35
        .long TIME_INT          ; hardware interrupt  36
                                      ;set 8 bit timer ch0 interrupt vector
        .long HARD_INT          ; hardware interrupt  37
```

初期化と割り込みルーチン例

```
.global INIT_8TIMER
INIT_8TIMER:

;interrupt disable
        ld.w    %r4,0x00                                                (1)
        ld.w    %psr,%r4        ;IE disnable
        xld.w   %r4,0x40160                                             (2)
        ld.w    %r5,0x00
        ld.b    [%r4],%r5       ;8timer0 disable in timer-reg
        xld.w   %r4,0x4014d                                             (3)
        ld.w    %r5,0x7
        ld.b    [%r4],%r5       ;8timer0 disable in pri-scaler

;8bit timer set
        xld.w   %r4,0x40161                                             (4)
        xld.w   %r5,0x26        ;interrupt every 10000 clocks
        ld.b    [%r4],%r5       ;8timer0 interval

        xld.w   %r4,0x40146                                             (5)
        ld.w    %r5,0x00
        ld.b    [%r4],%r5       ;8timer0 clock is divided clock

        xld.w   %r4,0x40269                                             (6)
        xld.w   %r5,0x03
        ld.b    [%r4],%r5       ;8timer0 interrupt priority level 3

        xld.w   %r4,0x40275                                             (7)
        xld.w   %r5,0x01
        ld.b    [%r4],%r5       ;8timer0 interrupt enable

        xld.w   %r4,0x40285                                             (8)
        xld.w   %r5,0x00
        ld.b    [%r4],%r5       ;8timer0 interrupt flag clear

        xld.w   %r4,0x4014d                                             (9)
        ld.w    %r5,0x0f
        ld.b    [%r4],%r5       ;8timer0 enable in pri-scaler

;8bit timer start
        ld.w    %r4,0x10                                                (10)
        ld.w    %psr,%r4        ;IE enable

        xld.w   %r4,0x40160                                             (11)
        xld.w   %r5,0x3
        ld.b    [%r4],%r5       ;clock out on,preset,start
; exit function
        ret

.global TIME_INT
TIME_INT:
        pushn   %r1
        xld.w   %r1,0x40285
        xld.w   %r0,0x01
        ld.b    [%r1],%r0       ;8timer5 interrupt flag reset
        popn    %r1
        reti
```

(1)割り込みを禁止

IEフラグをディセーブル(割り込み禁止)にします。割り込み許可状態で割り込みの設定を変更することは誤動作の原因になります。

```
        ld.w    %r4,0x00
        ld.w    %psr,%r4        ;IE disnable
```

(2)8ビットタイマのカウンタを停止

タイマを再設定するために一度タイマのカウントダウンを停止します。

```
        xld.w   %r4,0x40160
        ld.w    %r5,0x00
        ld.b    [%r4],%r5       ;8timer0 disable in timer-reg
```

(3)8ビットタイマへのプリスケーラのクロック供給を停止
タイマの設定を変更する前に、安全のためにプリスケーラからのクロック供給を停止します。

```
xld.w   %r4,0x4014d
ld.w    %r5,0x7
ld.b    [%r4],%r5       ;8timer0 disable in pri-scaler
```

(4)8ビットタイマの設定
リロードデータ0x26は、プリスケーラ入力クロック(デフォルトはOSC3またはPLL出力)の10,069クロックごとの割り込みを発生させる設定値です。

(0x26 + 1) × 256 = 10,059クロック

CPUコアが20MHz動作の場合、約0.5msごとに割り込みが発生します。

```
xld.w   %r4,0x40161
xld.w   %r5,0x26        ;interrupt every 10000 clocks
ld.b    [%r4],%r5       ;8timer0 interval
```

(5)8ビットタイマに使用する入力クロックを選択
8ビットタイマの入力クロックとして分周クロックを選択します。これはプリスケーラに関する設定です。

```
xld.w   %r4,0x40146
ld.w    %r5,0x00
ld.b    [%r4],%r5       ;8timer0 clock is divided clock
```

(6)8ビットタイマの割り込みプライオリティレベルの設定
割り込みレベルを3に設定します。

```
xld.w   %r4,0x40269
xld.w   %r5,0x03
ld.b    [%r4],%r5       ;8timer0 interrupt priority level 3
```

(7)8ビットタイマ割り込みイネーブルレジスタの設定を変更
8ビットタイマ割り込みを許可します。

```
xld.w   %r4,0x40275
xld.w   %r5,0x01
ld.b    [%r4],%r5       ;8timer0 interrupt enable
```

(8)8ビットタイマ割り込み要因のクリア
8ビットタイマ割り込み要因ビットをクリアします。

```
xld.w   %r4,0x40285
xld.w   %r5,0x00
ld.b    [%r4],%r5       ;8timer0 interrupt flag clear
```

(9)プリスケーラからのクロック供給を開始
プリスケーラの設定により、8ビットタイマへのクロック供給を開始します。
この時点ではタイマのカウントダウンは停止しています。

```
xld.w   %r4,0x4014d
ld.w    %r5,0x0f
ld.b    [%r4],%r5       ;8timer0 enable in pri-scaler
```

(10)割り込みを許可
PSRのIEビットを1にして割り込み許可にします。これ以後はあらゆる割り込みを受け付けるようになります。

```
ld.w    %r4,0x10
ld.w    %psr,%r4        ;IE enable
```

(11)8ビットタイマのカウンタのリセットおよびカウントダウン開始
最後に8ビットタイマのカウンタのリセットとカウントダウンの開始を同時に行います。ここでは2つを同時に行っていますが、リセット、開始の順に行っても問題ありません。

```
xld.w   %r4,0x40160
xld.w   %r5,0x3
ld.b    [%r4],%r5       ;clock out on,preset,start
```

3.3 16ビットタイマの設定

ここではS5U1C331M2Sミドルウェアのソースを例に、16ビットタイマの割り込み、およびPWM出力の制御方法について説明します。S1C33104の16ビットタイマは機能がS1C33209と大きく異なっていますので、注意してください。

● 割り込み用の設定

16ビットタイマ4のコンペアB割り込みについて説明します。

ベクタ部

```
#define INT30  mdyInt         // mdy interrupt routine                  (1)

       .text

       .long  RESET
       .long  RESERVED
       .long  RESERVED
       .long  RESERVED
       .long  ZERODIV
       .long  RESERVED
       .long  ADDRERR
       .long  NMI
       .long  RESERVED
       .long  RESERVED
       .long  RESERVED
       .long  RESERVED
       .long  SOFTINT0
       .long  SOFTINT1
       .long  SOFTINT2
       .long  SOFTINT3
       .long  INT0
       .long  INT1
              |
       .long  INT29
       .long  INT30                                                    (1)
       .long  INT31
              |
```

(1)割り込みベクタの設定
INT30(16ビットタイマ4のコンペアB割り込み)のベクタとして割り込みルーチンmdyIntを登録します。

割り込み禁止とPSRの退避、復帰ルーチン

```
       .section .bss
       .align   2
MDY_PSR :
       .zero    4

       .global mdyInt
mdyIntOff:                                                             (1)
       xld.w  %r10,MDY_PSR
       ld.w   %r11,%psr      // save %psr and IE disable
       ld.w   [%r10],%r11
       ld.w   %r10,0
       ld.w   %psr,%r10
       ret
       .global mdyIntOn
mdyIntOn:                                                              (2)
       xld.w  %r10,MDY_PSR
       ld.w   %r11,[%r10]
       ld.w   %psr,%r11      // restore %psr
       ret
```

(1)割り込み禁止
PSRの内容を保存し、IEビットを0にして割り込みを禁止しています。

(2)割り込み許可
(1)で保存したPSRの内容を戻しています。

これらはCから呼び出されます。

16ビットタイマ設定部

```
//*****************************************************************
// void mdyTmOpen(unsigned short freq)
//      start timer 4, underflow interrupt with freq count
//      prescaler is 1/1024
//*****************************************************************

void mdyTmOpen(unsigned short freq)
    {
        unsigned char ucTmp;

// interrupt disable
        mdyIntOff();                                                          (1)

// set TM4 prescaler to 1/1024, 0b00001110

        *(volatile unsigned char *)(0x4014b) = 0xe;                           (2)

// set TM4 reload and compare data

        *(volatile unsigned short *)(0x481a2) = freq;      //set compare b    (3)
        *(volatile unsigned short *)(0x481a4) = 0x0;       //dummy data for up counter

// set TM4 control register
// fine mode off,compare buf off,reverse off,internal clock,clock out off,preset,stop
//      0x401a6 0010,

        *(volatile unsigned char *)(0x481a6) = 0x0;

// set TM4 match compare b come to cpu interrupt

        *(volatile unsigned char *)(0x40291) &= 0xBf;      //set timer 4 enable (4)

// set TM4,interrupt priority level 3

        ucTmp = *(volatile unsigned char *)(0x40268);
        ucTmp = ucTmp & 0xf0;
        ucTmp = ucTmp | 0x3;
        *(volatile unsigned char *)(0x40268) = ucTmp;

// clear TM4 interrupt factor flags (write 1, and reset)

        *(volatile unsigned char *)(0x40284) &= 0x0C;

// set TM4 underflow interrupt enable

        *(volatile unsigned char *)(0x40274) |= 0x04;      //set timer 4 enable

// start TM4 counter

        *(volatile unsigned char *)(0x481a6) |= 0x01;                         (5)

// interrupt enable
        mdyIntOn();                                                           (6)
    }
```

(1)割り込み禁止
安全のため、割り込みを禁止します。

```
// interrupt disable
    mdyIntOff();
```

(2)プリスケーラの設定
プリスケーラで1024分周されたクロックがタイマ4の入力クロックとなります。

```
// set TM4 prescaler to 1/1024, 0b00001110
    *(volatile unsigned char *)(0x4014b) = 0xe;
```

(3)タイマ4の周期(コンペアB)、コンペアA、その他の設定

次の設定でタイマ4のコンペアB割り込み周期は、クロック数で(freq + 1) × 1024となります。

```
// set TM4 reload and compare data
    *(volatile unsigned short *)(0x481a2) = freq;      //set compare b
    *(volatile unsigned short *)(0x481a4) = 0x0;       //dummy data for up counter
```

タイマ4のその他の設定を行います。

```
// set TM4 control register
// fine mode off,compare buf off,reverse off,internal clock,clock out off, ...
//  0x401a6 0010,
    *(volatile unsigned char *)(0x481a6) = 0x0;
```

(4)割り込みコントローラの設定

タイマ4のコンペアB割り込みが、IDMAの起動ではなくCPUへの即時割り込みになるように設定します。

```
// set TM4 match compare b come to cpu interrupt
    *(volatile unsigned char *)(0x40291) &= 0xBf;      //set timer 4 enable
```

割り込みプライオリティは3とします。

```
// set TM4,interrupt priority level 3
    ucTmp = *(volatile unsigned char *)(0x40268);
    ucTmp = ucTmp & 0xf0;
    ucTmp = ucTmp | 0x3;
    *(volatile unsigned char *)(0x40268) = ucTmp;
```

割り込み要因フラグを安全のためクリアしておきます。

```
// clear TM4 interrupt factor flags (write 1, and reset)
    *(volatile unsigned char *)(0x40284) &= 0x0C;
```

コンペアB割り込みを有効とします。

```
// set TM4 underflow interrupt enable
    *(volatile unsigned char *)(0x40274) |= 0x04;      //set timer 4 enable
```

(5)タイマスタート

タイマ4のカウントを開始させます。

```
// start TM4 counter
    *(volatile unsigned char *)(0x481a6) |= 0x01;
```

(6)割り込み許可

最後に、割り込み許可に戻します。

```
// interrupt enable
    mdyIntOn();
```

なお、上記の他に割り込みルーチン(mdyInt)の記述が別途必要です。
割り込みルーチンでは、割り込み要因フラグを必ずクリアしてください。
クリア例)

```
// clear TM4 interrupt factor flags (write H and reset)
    *(volatile unsigned char *)(0x40284) &= 0x0C;
```

クリアしないと、割り込みが許可された時点で再度同じ割り込みが発生してしまいます。

● PWM用の設定

以下PWM処理についてS5U1C331M2Sミドルウェアのソースを例に説明します。

PWM初期設定

```
//*****************************************************************
// void mdyTm0Set (unsigned short count, unsigned short compare)
//*****************************************************************

static void mdyTm0Set (unsigned short count, unsigned short compare, int reverse)
    {
// interrupt disable
       mdyIntOff();                                                          (1)

// set P22 port to TM0

       *(volatile char *)(0x402d8) |= 0x04;                                  (2)

// set TM0 prescaler to 1/16, 0b0001011

       *(volatile unsigned char *)(0x40147) = 0x0b;                          (3)

// set TM0 reload and compare data

       *(volatile unsigned short *)(0x48182) = count;           //compare B  (4)
       *(volatile unsigned short *)(0x48180) = compare;         //compare A

// set TM0 control register
// fine mode off, compare buf, reverse, internal clock, clock out on, preset, stop
// internal clock, clock out on, preset, stop
//     0x4018e 0b00010100 or 0b00110100

       if (reverse==1){
              *(volatile unsigned char *)(0x48186) = 0x34;
       }
       else{
              *(volatile unsigned char *)(0x48186) = 0x24;
       }

// reset TM0 counter

       *(volatile unsigned char *)(0x48186) |= 0x02;                         (5)

// interrupt enable
       mdyIntOn();                                                           (6)
    }
```

(1)割り込みを禁止

安全のため、割り込みを禁止します。

```
// interrupt disable
    mdyIntOff();
```

(2)ポート機能の切り換え

PWM(16ビットタイマ)出力に使用するポートはデフォルトで汎用入出力ポートとなりますので、PWM出力に切り換えます。

```
// set P22 port to TM0
    *(volatile char *)(0x402d8) |= 0x04;
```

(3)プリスケーラの設定

プリスケーラで16分周されたクロックがタイマ0の入力クロックとなります。

```
// set TM0 prescaler to 1/16, 0b0001011
    *(volatile unsigned char *)(0x40147) = 0x0b;
```

(4)タイマ0の設定

まず、コンペアAとコンペアBレジスタを設定します。

コンペアB+1のカウントで1周期となります。出力は通常モードでは0からスタート、反転モードでは1からスタートします。コンペアA+1のカウントで出力の0と1が切り換わります。

たとえば、通常モードでコンペアBが5、コンペアAが0の場合、最初の1クロックは0、残りの4クロックは1となり、これが繰り返されます。

```
// set TM0 reload and compare data
    *(volatile unsigned short *)(0x48182) = count;           //compare B
    *(volatile unsigned short *)(0x48180) = compare;         //compare A
```

タイマ0のその他の設定を行います。

```
// set TM0 control register
// fine mode off, compare buf, reverse, internal clock, clock out on, preset, stop
// internal clock, clock out on, preset, stop
//  0x4018e 0b00010100 or 0b00110100
    if (reverse==1){
           *(volatile unsigned char *)(0x48186) = 0x34;
    }
    else{
           *(volatile unsigned char *)(0x48186) = 0x24;
    }
```

(5)タイマ0のカウンタリセット

タイマ0のカウンタを0にリセットします。

```
// reset TM0 counter
    *(volatile unsigned char *)(0x48186) |= 0x02;
```

(6)割り込みを許可

最後に、割り込み許可に戻します。

```
// interrupt enable
    mdyIntOn();
```

PWMスタート部

```
//*********************************************************************
// void mdyTm0Start ()
//*********************************************************************

static void mdyTm0Start ()
    {
// start TM0 counter

       *(volatile unsigned char *)(0x48186) |= 0x03;
    }
```

この関数でPWMをスタートします。

PWM変更部

```
//*********************************************************************
// void mdyTm0Change (unsigned short count, unsigned short compare)
//*********************************************************************

static void mdyTm0Change (unsigned short count, unsigned short compare)
    {
// set TM0 reload and compare data
       *(volatile unsigned short *)(0x48182) = count;    // compare B
       *(volatile unsigned short *)(0x48180) = compare;  // compare A
    }
```

この関数でPWM波形の周期、デューティを変更します。
mdyTm0Set()関数の(4)の設定で、コンペアバッファを許可(0x48186・D5 = 1)しました。
それにより、コンペアA/Bデータをカウンタとは非同期にバッファに書き込むことができます。いったんバッファに入ったデータは、コンペアBマッチでカウンタが0に戻った時点でコンペアA/Bレジスタに設定されます。コンペアバッファを使用しない場合、コンペアA/Bデータは書き込みと同時に有効になりますので、同期がとれていないとコンペアAマッチが1回無効となる(発生しない)場合があります。

3.4 シリアルインタフェースの設定

ここではS5U1C331M2Sミドルウェアのソースを例に、シリアルインタフェースを使用した非同期通信の制御方法について説明します。

● 外部クロック使用の非同期通信

以下の例は、mon33g\src\m3s_sci.sから抜粋したアセンブリソースです。この例では割り込みを使用せずに、ポーリングにより通信を制御します。

初期化ルーチン

```
#define        MON_VER  0x10              ;moinitor firm-ware version

#define        STDR     0x000401e0        ;transmit data register(ch0)
#define        SRDR     0x000401e1        ;receive data register(ch0)
#define        SSR      0x000401e2        ;serial status register(ch0)
#define        SCR      0x000401e3        ;serial control register(ch0)
#define        SIR      0x000401e4        ;IrDA control register(ch0)
#define        PIO_SET  0x07              ;port function register

#define        SIR_SET  0x0               ;SIR set(1/16 mode)
#define        SCR_SET  0x7               ;SCR set(#SCLK input 1.843MHz 115200bps)
#define        SCR_EN   0xc0              ;SCR enable
#define        PIO      0x000402d0        ;IO port (P port) register

      .text
;*****************************************************************************
;
;     void m_io_init()
;               serial port initial function
;
;*****************************************************************************
      .global  m_io_init
m_io_init:
      xld.w    %r1,SIR                                                    (1)
      ld.w     %r0,SIR_SET               ;1/16 mode
      ld.b     [%r1],%r0                 ;SIR set

      xld.w    %r1,SCR                                                    (2)
      ld.w     %r0,SCR_SET
      ld.b     [%r1],%r0                 ;SCR set(#SCLK input 1.843MHz)

      xld.w    %r1,PIO                                                    (3)
      xld.w    %r0,PIO_SET
      ld.b     [%r1],%r0                 ;IO port set

      xld.w    %r1,SCR                                                    (4)
      xld.w    %r0,SCR_EN
      xoor     %r0,SCR_SET
      ld.b     [%r1],%r0                 ;SCR set
      ret
```

(1)クロック分周比選択

サンプリングクロックの分周比を1/16に設定します。

```
xld.w    %r1,SIR
ld.w     %r0,SIR_SET       ;1/16 mode
ld.b     [%r1],%r0         ;SIR set
```

(2)転送モードの設定

調歩同期式8ビット、ストップビットを1ビット、パリティなし、SCLKを外部クロックに設定します。S5U1C331M2Sの通信にはS5U1C330M1D1から1.843MHzの外部クロックを入力します(115,200bpsのボーレート)。

```
xld.w    %r1,SCR
ld.w     %r0,SCR_SET
ld.b     [%r1],%r0         ;SCR set(#SCLK input 1.843MHz)
```

(3)入出力端子の切り換え

入出力ポートと兼用の端子をシリアルインタフェース用に切り換えます。

```
xld.w    %r1,PIO
xld.w    %r0,PIO_SET
ld.b     [%r1],%r0                ;IO port set
```

(4)送受信を許可

送受信をイネーブルにします。

```
xld.w    %r1,SCR
xld.w    %r0,SCR_EN
xoor     %r0,SCR_SET
ld.b     [%r1],%r0                ;SCR set
```

送信ルーチン

```
;*****************************************************************************
;
;     void m_snd_1byte( sdata )
;               1 byte send function
;                 IN : uchar sdata (R6)  send data
;
;*****************************************************************************
      .global  m_snd_1byte
m_snd_1byte:
      pushn    %r3                 ;save r3-r0
snd000:
      xld.w    %r0,SSR             ;Address set
      xld.w    %r1,STDR

      xbtst    [%r0],0x1           ;TDBE1(bit1) == 0(full) ?            (1)
      jreq     snd000              ;if full, jp snd000
      xld.b    [%r1],%r6           ;write data                          (2)
      popn     %r3                 ;restore r3-r0
      ret
```

(1)送信バッファのチェック

シリアルI/Fステータスレジスタのビットをチェックし、送信バッファが空になるまで待ちます。

```
xbtst    [%r0],0x1           ;TDBE1(bit1) == 0(full) ?
jreq     snd000              ;if full, jp snd000
```

(2)1バイトのデータを送信

送信バッファが空になると、R6にある1バイトのデータを送信します。

```
xld.b    [%r1],%r6           ;write data
```

受信ルーチン

```
;*****************************************************************************
;
;     uchar m_rcv_1byte()
;               1 byte receive function
;                 OUT : 0 receive OK
;                       1 receive ERROR (framing  err)
;                       2               (parity   err)
;                       3               (over run err)
;
;*****************************************************************************
      .global  m_rcv_1byte
m_rcv_1byte:
      pushn    %r3                 ;save r3-r0
      xld.w    %r1,SSR             ;Address set
      xld.w    %r2,SRDR
rcv000:
      xbtst    [%r1],0x0           ;RDBF1(bit0) == 0(empty) ?           (1)
      jreq     rcv000              ;if empty, jp rcv000

      ld.w     %r4,0x0                                                  (2)
      xbtst    [%r1],0x4           ;FER1(bit4) == 0 ?
      jreq     rcv010
      xbclr    [%r1],0x4           ;FER1(bit4) 0 clear
      ld.w     %r4,0x1             ;return 1
```

```
rcv010:
      xbtst    [%r1],0x3            ;PER1(bit3) == 0 ?
      jreq     rcv020
      xbclr    [%r1],0x3            ;PER1(bit3) 0 clear
      ld.w     %r4,0x2              ;return 2
rcv020:
      xbtst    [%r1],0x2            ;OER1(bit2) == 0 ?
      jreq     rcv030
      xbclr    [%r1],0x2            ;OER1(bit2) 0 clear
      ld.w     %r4,0x3              ;return 3
rcv030:
      xld.b    %r0,[%r2]            ;read data                              (3)
      xld.w    %r1,m_rcv_data       ;read data set
      ld.b     [%r1],%r0
      popn     %r3                  ;restore r3-r0
      ret
```

(1)受信待ち

受信データがバッファに入るまで待ちます。

```
rcv000:
     xbtst  [%r1],0x0              ;RDBF1(bit0) == 0(empty) ?
     jreq   rcv000                 ;if empty, jp rcv000
```

(2)受信エラーのチェック

フレーミングエラー、パリティエラー、オーバーランエラーがないかチェックします。

```
     ld.w   %r4,0x0
     xbtst  [%r1],0x4              ;FER1(bit4) == 0 ?
     jreq   rcv010
     xbclr  [%r1],0x4              ;FER1(bit4) 0 clear
     ld.w   %r4,0x1                ;return 1
rcv010:
     xbtst  [%r1],0x3              ;PER1(bit3) == 0 ?
     jreq   rcv020
     xbclr  [%r1],0x3              ;PER1(bit3) 0 clear
     ld.w   %r4,0x2                ;return 2
rcv020:
     xbtst  [%r1],0x2              ;OER1(bit2) == 0 ?
     jreq   rcv030
     xbclr  [%r1],0x2              ;OER1(bit2) 0 clear
     ld.w   %r4,0x3                ;return 3
```

(3)受信データの読み出し

エラーがなければ、1バイトの受信データをバッファから読み出してRAM上に保存します。

```
rcv030:
     xld.b  %r0,[%r2]              ;read data
     xld.w  %r1,m_rcv_data         ;read data set
     ld.b   [%r1],%r0
```

● 内部クロック使用の非同期通信

送信および受信部は、外部クロック使用時と同じで、初期化ルーチンのみが異なります。
なお、これはS1C33104用のソースです。S1C33209でも、プルアップ処理が不要な以外は同一です。

初期化ルーチン

```
#define     MON_VER   0x11          ;moinitor firm-ware version
#define     P8TS3     0x0004014e    ;8bit timer3 clock rate register
#define     PT3       0x0004016c    ;8bit timer3 control register
#define     RLD3      0x0004016d    ;8bit timer3 reload data register
#define     STDR1     0x000401e5    ;transmit data register
#define     SRDR1     0x000401e6    ;receive data register
#define     SSR1      0x000401e7    ;serial status register
#define     SCR1      0x000401e8    ;serial control register
#define     SIR1      0x000401e9    ;IrDA control register
#define     PIO       0x000402d0    ;IO port (P port) register
#define     IOU       0x000402d3    ;IO port (P port) pull up register

       .text
;***************************************************************************
;
```

```
;     void m_io_init()
;               serial port initial function
;
;*****************************************************************************
      .global  m_io_init
m_io_init:
      xld.w    %r1,SIR1                                                  (1)
      ld.w     %r0,0x00
      ld.b     [%r1],%r0     ;SIR1 set
      xld.w    %r1,SCR1                                                  (2)
      ld.w     %r0,0x03
      ld.b     [%r1],%r0     ;SCR1 set
      xld.w    %r1,PIO                                                   (3)
      xld.w    %r0,0x30
      ld.b     [%r1],%r0     ;IO port set
      xld.w    %r1,IOU                                                   (4)
      xld.w    %r0,0xff
      ld.b     [%r1],%r0     ;pull up set
      xld.w    %r1,P8TS3                                                 (5)
      xld.w    %r0,0x80
      ld.b     [%r1],%r0     ;P8TS3 set
      xld.w    %r1,RLD3                                                  (6)
;     xld.w    %r0,0x20      ;9600bps
      xld.w    %r0,0x7       ;38400bps
;     xld.w    %r0,0x3       ;38400bps(clock:10MHz)for debug
;     xld.w    %r0,0xf       ;19200bps
      ld.b     [%r1],%r0     ;RLD3 set
      xld.w    %r1,PT3                                                   (7)
      ld.w     %r0,0x07
      ld.b     [%r1],%r0     ;PT3 set
      xld.w    %r1,SCR1                                                  (8)
      xld.w    %r0,0xc3
      ld.b     [%r1],%r0     ;SCR1 set
      ret
```

(1)クロック分周比選択

サンプリングクロックの分周比を1/16に設定します。

```
xld.w    %r1,SIR1
ld.w     %r0,0x00
ld.b     [%r1],%r0     ;SIR1 set
```

(2)転送モードの設定

調歩同期式8ビット、ストップビットを1ビット、パリティなし、内部クロック(8ビットタイマ3)に設定します。

```
xld.w    %r1,SCR1
ld.w     %r0,0x03
ld.b     [%r1],%r0     ;SCR1 set
```

(3)入出力端子の切り換え

入出力ポートと兼用の端子をシリアルインタフェース用に切り換えます。

```
xld.w    %r1,PIO
xld.w    %r0,0x30
ld.b     [%r1],%r0     ;IO port set
```

(4)プルアップの設定(S1C33104)

シリアルインタフェース用入力端子のプルアップをイネーブルに設定します。この処理はS1C33104用で、S1C33209では不要です。

実際のアプリケーションでは、どちらの場合も外部にプルアップ抵抗を接続することを推奨します。

```
xld.w    %r1,IOU
xld.w    %r0,0xff
ld.b     [%r1],%r0     ;pull up set
```

(5)プリスケーラの設定

8ビットタイマ3用のプリスケーラの分周比を、内部クロックの1/2とします。

なお、S1C33209では、1/1を選択(0x40146・D3 = 1)することもできます。

```
xld.w    %r1,P8TS3
xld.w    %r0,0x80
ld.b     [%r1],%r0     ;P8TS3 set
```

(6)8ビットタイマの設定

8ビットタイマに7(7 + 1 = 8分周)をプリセットします。
エプソン内部のプロトボード(動作クロック = 20MHz)の場合、
20MHz→プリスケーラで2分周→タイマで8分周→シリアルI/Fで2分周→サンプリング用に16分周
= 20,000,000 / 2 / 8 / 2 / 16 = 39,062bsp
となります。
38,400bpsに対して誤差が+1.7%となります。この程度の誤差であれば、相手側に大きな誤差がなく、かつ通常の使用条件であれば問題なく動作します。

```
xld.w    %r1,RLD3
xld.w    %r0,0x7       ;38400bps
ld.b     [%r1],%r0     ;RLD3 set
```

(7)8ビットタイマをスタート

8ビットタイマをスタートします。

```
xld.w    %r1,PT3
ld.w     %r0,0x07
ld.b     [%r1],%r0     ;PT3 set
```

(8)送受信を許可

送受信をイネーブルにします。

```
xld.w    %r1,SCR1
xld.w    %r0,0xc3
ld.b     [%r1],%r0     ;SCR1 set
```

3.5 A/D変換器の設定

ソフトウェアトリガによるA/D変換ルーチンを、gnu33\sample\drv33209\demo_ad2\内のサンプルから抜粋して説明します。

ベクタテーブル vector.c

```
extern void int_ad(void);                                                    (1)

/* vector table */
const unsigned long vector[] = {
       (unsigned long)boot,         // 0   0
                      |
       (unsigned long)dummy,        // 248 62
       (unsigned long)dummy,        // 252 63
       (unsigned long)int_ad,       // 256 64                                (1)
       (unsigned long)dummy,        // 260 65
       (unsigned long)dummy,        // 264 66
       (unsigned long)dummy,        // 268 67
       (unsigned long)dummy,        // 272 68
       (unsigned long)dummy,        // 276 69
       (unsigned long)dummy,        // 280 70
       (unsigned long)dummy         // 284 71
};
```

(1)ベクタテーブルの設定

このサンプルでは、A/D変換終了時に割り込みを発生させ、割り込みルーチンでA/D変換データを取得します。その割り込みルーチンの先頭アドレスをベクタテーブル(ベクタテーブル先頭アドレス+0x100の位置)に登録します。

A/D変換器の初期設定 drv_ad2.c

```
#include "..¥include¥ad.h"
#include "..¥include¥common.h"
#include "..¥include¥int.h"
#include "..¥include¥io.h"
#include "..¥include¥presc.h"

/* Prototype */
void init_ad(void);
unsigned short read_ad_data(void);
void int_ad(void);
extern void save_psr(void);
extern void restore_psr(void);

/************************************************************************
 * init_ad
 *   Type :     void
 *   Ret val :  none
 *   Argument : void
 *   Function : Initialize A/D converter.
 ************************************************************************/
void init_ad(void)
{
       /* Save PSR and disable all interrupt */
       save_psr();

       /* Set A/D converter port setting */
       *(volatile unsigned char *)IN_CFK6_ADDR = IN_CFK60_AD0;   // A/D ch.0 port (1)

       /* SPT = A/D converter sampling time
            OSC3 = OSC3 clock (40MHz)
            PDR = Prescaler clock division (1/32)
            ST = A/D converter sampling time (9clock)
            TADC = A/D converter sampling and  convert time (10us)
               SPT = ST / (OSC3 x PDR)
                   = 9 / (40 x 1000000 x 1/32)
                   = 7.2us
               Must be SPT > TADC / 2 */
```

```
        /* Set A/D converter prescaler setting (CLK/32) */
        *(volatile unsigned char *)PRESC_PSAD_ADDR                              (2)
               = PRESC_PTONL_ON | PRESC_CLKDIVL_SEL4;
               // Set A/D converter prescaler (CLK/32)

        /* Set A/D converter status register */
        *(volatile unsigned char *)AD_CH_ADDR = AD_MS_NOR | AD_TS_SOFT;         (3)
               // A/D converter software trigger and normal mode
        *(volatile unsigned char *)AD_CS_ADDR = AD_CS_0 | AD_CE_0;              (3)
               // A/D converter start channel AD0 and A/D end channel AD0
        *(volatile unsigned char *)AD_OWE_ADDR                                  (3)
               =  AD_ADE_ENA | AD_ADST_STOP | AD_OWE_NOERR;
               // A/D converter enable, A/D converter stop,
               // A/D converter over write error clear
        *(volatile unsigned char *)AD_ST_ADDR = AD_ST_9;                        (2)
               // A/D converter sampling 9 clocks

        /* Set A/D converter interrupt CPU request on interrupt controller */
        *(volatile unsigned char *)INT_RS1_RADE_RP4_ADDR = INT_RIDMA_DIS;
               // IDMA request disable and CPU request enable                  (4)

        /* Set A/D converter interrupt priority level 3 on interrupt controller */
        *(volatile unsigned char *)INT_PSIO1_PAD_ADDR = INT_PRIH_LVL3;          (4)

        /* Reset A/D converter interrupt factor flag on interrupt controller */
        *(volatile unsigned char *)INT_FADE_FCTM_FP4_ADDR = INT_FADE;           (4)
               // Reset A/D converter interrupt factor flag

        /* Set A/D converter interrupt enable on interrupt controller */
        *(volatile unsigned char *)INT_EADE_ECTM_EP4_ADDR = INT_EADE;           (4)
               // Set A/D converter interrupt enable

        /* Restore PSR */
        restore_psr();
}
```

最初のインクルードファイル群は、gnu33\sample\drv33209\includeにありますので、定義内容については各ファイルを参照してください。

(1)アナログ入力端子の設定

K60汎用入力ポートと兼用されているA/D変換器ch0の入力端子を、アナログ入力用に切り換えます（デフォルトはK60汎用入力端子）。

```
/* Set A/D converter port setting */
*(volatile unsigned char *)IN_CFK6_ADDR = IN_CFK60_AD0;   // A/D ch.0 port
```

(2)プリスケーラとサンプリング時間の設定

```
/* SPT = A/D converter sampling time
     OSC3 = OSC3 clock (40MHz)
     PDR = Prescaler clock division (1/32)
     ST = A/D converter sampling time (9clock)
     TADC = A/D converter sampling and  convert time (10us)
              SPT = ST / (OSC3 x PDR)
                  = 9 / (40 x 1000000 x 1/32)
                  = 7.2us
        Must be SPT > TADC / 2 */
```

このコメントは、A/D変換器の入力クロックの計算法を示しています。

まず、システムクロックからA/D変換器の動作クロックを生成するプリスケーラの分周比を設定します。1/2から1/256まで、2の倍数で選択できます。ここでは40MHzのシステムクロックを想定し、プリスケーラで1/32分周します。

次に入力サンプリング時間を9 A/D変換クロックとします。これはサンプル&ホールドの時間です。この時間は、A/D変換時間t_{ADC}(min. 10µs)の1/2以上(5µs以上)としてください。この例では、7.2µsとなります。もし、プリスケーラで1/16を選択すると倍の3.6µsとなってしまいます。なお、5µs以下でも動作はしますが、サンプル時間の不足により誤差が大きくなります。

A/D変換器はサンプル&ホールド後、約10クロックで逐次比較を行い、10ビットのA/D変換結果を出します。

A/D変換クロック用のプリスケーラを1/32分周に、A/D変換のサンプリングクロック数を9に設定します。

```
/* Set A/D converter prescaler setting (CLK/32) */
*(volatile unsigned char *)PRESC_PSAD_ADDR = PRESC_PTONL_ON | PRESC_CLKDIVL_SEL4;
       // Set A/D converter prescaler (CLK/32)
                                |
*(volatile unsigned char *)AD_ST_ADDR = AD_ST_9;
       // A/D converter sampling 9 clocks
```

(3) A/D変換器の設定

変換モード(連続、通常)を通常モード、トリガ(外部/K52、8ビットタイマ0、16ビットタイマ0、ソフトトリガ)をソフトトリガに設定します。

```
/* Set A/D converter status register */
*(volatile unsigned char *)AD_CH_ADDR = AD_MS_NOR | AD_TS_SOFT;
       // A/D converter software trigger and normal mode
```

変換チャネルをch0に設定します。

```
*(volatile unsigned char *)AD_CS_ADDR = AD_CS_0 | AD_CE_0;
       // A/D converter start channel AD0 and A/D end channel AD0
```

A/D変換を許可します。

```
*(volatile unsigned char *)AD_OWE_ADDR
        =  AD_ADE_ENA | AD_ADST_STOP | AD_OWE_NOERR;
       // A/D converter enable, A/D converter stop, A/D .. over write error clear
```

(4) 割り込みの設定

割り込みコントローラで、A/D変換割り込みをCPUへの割り込み要求とします。

```
/* Set A/D converter interrupt CPU request on interrupt controller */
*(volatile unsigned char *)INT_RS1_RADE_RP4_ADDR = INT_RIDMA_DIS;
       // IDMA request disable and CPU request enable
```

割り込みレベルを3に設定します。

```
/* Set A/D converter interrupt priority level 3 on interrupt controller */
*(volatile unsigned char *)INT_PSIO1_PAD_ADDR = INT_PRIH_LVL3;
```

割り込み要因フラグをクリアします。

```
/* Reset A/D converter interrupt factor flag on interrupt controller */
*(volatile unsigned char *)INT_FADE_FCTM_FP4_ADDR = INT_FADE;
       // Reset A/D converter interrupt factor flag
```

割り込みを許可します。

```
/* Set A/D converter interrupt enable on interrupt controller */
*(volatile unsigned char *)INT_EADE_ECTM_EP4_ADDR = INT_EADE;
       // Set A/D converter interrupt enable
```

割り込み処理 drv_ad2.c

```
/************************************************************************
 * read_ad_data
 *   Type :     unsigned short
 *   Ret val :  A/D converter data
 *   Argument : void
 *   Function : Read A/D converter data.
 ************************************************************************/
unsigned short read_ad_data(void)
{
       return(*(volatile unsigned short *)AD_ADD_ADDR);  // A/D converter data   (2)
}

/************************************************************************
 * int_ad
 *   Type :     void
 *   Ret val :  none
 *   Argument : void
 *   Function : A/D converter interrupt function.
 *                Read A/D converter status and A/D convert data.
 ************************************************************************/
void int_ad(void)
{
       extern volatile unsigned short ad_data;    // A/D data
       extern volatile int ad_int;                // A/D converter interrupt flag

       INT_BEGIN;                                                              (1)
       ad_data = read_ad_data();    // Read A/D converter data                 (2)
       ad_int = TRUE;               // A/D converter interrupt flag on
       *(volatile unsigned char *)INT_FADE_FCTM_FP4_ADDR = INT_FADE;
                                    // Reset A/D converter interrupt factor flag (3)
       INT_END;                                                                (1)
}
```

A/D変換が終了するとA/D割り込みが発生し、int_ad( )が呼び出されます。

(1)レジスタの待避/復帰

処理ルーチンの最初と最後で行うレジスタの待避/復帰には、common.h内に定義されているINT_BEGINとINT_ENDを使用しています。

```
#define INT_BEGIN     asm("pushn     %r15")
#define INT_END       asm("popn      %r15¥n reti")
```

(2)変換結果の読み出し

read_ad_data( )を呼び出してA/Dの変換結果を変数に入れ、読み出しが完了したことを示すフラグをセットします。

```
ad_data = read_ad_data();    // Read A/D converter data
ad_int = TRUE;               // A/D converter interrupt flag on
```

(3)割り込み要因フラグのリセット

割り込み要因フラグをクリアします。

```
*(volatile unsigned char *)INT_FADE_FCTM_FP4_ADDR = INT_FADE;
       // Reset A/D converter interrupt factor flag
```

アプリケーション部　demo_ad2.c

```
    |
unsigned short ad_data;
volatile int ad_int;         // A/D converter interrupt flag
    |
init_ad();                                                                    (1)
    |
for (i = 0; i < DATA_SIZE; i++) {
       ad_int = FALSE;
       /* A/D converter start by software trigger */
       *(volatile unsigned long*)AD_OWE_ADDR |= 0x02;                         (2)
                            // Set A/D converter run bit (ADST[D1] = 1)

       for (;;) {
              if (ad_int == TRUE) {                                           (3)
                     write_str("   A/D AD0 data ... ");
                     write_hex(ad_data);
                     break;
              }
       }
}
```

これは、実際にA/D変換を行う制御プログラムです。

(1)初期設定

前述の init_ad()を呼び出して、A/D変換器と割り込みの初期設定を行います。

(2)A/D変換スタート

ソフトトリガによりA/D変換をスタートします。

```
/* A/D converter start by software trigger */
*(volatile unsigned long*)AD_OWE_ADDR |= 0x02;
                      // Set A/D converter run bit (ADST[D1] = 1)
```

(3)A/D変換結果の取得

A/D変換が終了すると前述の割り込み処理ルーチンint_ad( )が呼ばれ、処理が終了するとフラグad_intがセットされます。そのフラグを確認し、1ならば変数ad_dataから変換結果を読み出し表示します。

```
for (;;) {
       if (ad_int == TRUE) {
              write_str("   A/D AD0 data ... ");
              write_hex(ad_data);
              break;
       }
}
```

3.6　IDMAの設定

ソフトウェアトリガでIDMAを起動してデータを転送する例を、gnu33\sample\drv33209\idma内のサンプルから抜粋して説明します。

IDMAテーブル用ヘッダファイル　\sample\include\idma.h

```
       |
#define  IDMA_DCHN_ADDR    0x48204         // Address for IDMA channel number, IDMA
                                              start register
#define  IDMA_DMAEN_ADDR   0x48205         // Address for IDMA enable register
       |
/* IDMA control word bit field definition ... 1st word */
#define  IDMA_LNKEN_ENA    0x80000000      // Link enable
       |
/* IDMA control word bit field definition ... 2nd word */
#define  IDMA_DINTEN_ENA   0x80000000      // IDMA terminate interrupt enable
#define  IDMA_DINTEN_DIS   0x00000000      // IDMA terminate interrupt disable
#define  IDMA_DATSIZ_HW    0x40000000      // Data size .. half word
#define  IDMA_DATSIZ_BYTE  0x00000000      // Data size .. byte
#define  IDMA_SRINC_INC    0x30000000      // Increase address (return initial value
                                              when block transfer mode)
       |
/* IDMA control word bit field definition ... 3rd word */
#define  IDMA_DMOD_BLOCK   0x80000000      // Block transfer mode
       |
#define  IDMA_DSINC_INC    0x30000000      // Increase address (return initial value
                                              when block transfer mode)
       |
```

IDMAの設定と起動　demo_idma.c

```
       |
#include "..¥include¥common.h"
#include "..¥include¥idma.h"

/* Prototype */
int main(void);
void fill_mem1(unsigned long);
void fill_mem2(unsigned long);
void check_data(unsigned long , unsigned long);
extern void write_str(char *);
extern void init_idma(unsigned long, unsigned char);
extern void write_idma_info(unsigned long *, unsigned long, unsigned long, unsigned
long);

/* Global variable define */
volatile int idmaint_flg;              // IDMA interrupt flag

/* IDMA source and destination address */
#define  IDMA_SRC_START0   0x6c0000    // IDMA ch.0 source start address
#define  IDMA_SRC_START1   0x6d0000    // IDMA ch.1 source start address
#define  IDMA_DST_START0   0x6e0000    // IDMA ch.0 destination start address
#define  IDMA_DST_START1   0x6f0000    // IDMA ch.1 destination start address

/* IDMA start channel, link channel, transfer block size, transfer counter */
#define  IDMA_ST_CH0       0x0         // IDMA start ch.0
#define  IDMA_LINK         0x1000000   // IDMA transfer link ch.1 (1st word:D30-D24)
#define  IDMA_CNT          0x100       // IDMA transfer count (1st word:D23-D8)
#define  IDMA_BSIZE        0x80        // IDMA block size (1st word:D7-D0)

struct {
         unsigned long data[3];
} dma_control[128];                                                          (1)

/*****************************************************************************
 * main
 *   Type :     void
 *   Ret val :  none
 *   Argument : void
 *   Function : IDMA demonstration program.
 *****************************************************************************/
```

```
int main(void)
{
   unsigned long first_wd, second_wd, third_wd;
   unsigned char start_ch;

   write_str("*** IDMA demonstration ***¥n");
   write_str("¥n");

   /* Fill memory */
   write_str("*** Filling memory area(address 0x6c0000 - 0x6c00ff) ***¥n");
   write_str("    Pattern  ... 0x00 0x01 .... 0xfe 0xff¥n");
   fill_mem1(IDMA_SRC_START0);
   write_str("*** Filling memory area(address 0x6e0000 - 0x6e00ff) ***¥n");
   write_str("    Pattern  ... 0xff 0xfe .... 0x01 0x00¥n");
   fill_mem2(IDMA_SRC_START1);

   /* Disable IDMA transfer */
   *(volatile unsigned char*)IDMA_DMAEN_ADDR &= 0xfe;

   /* Initialize IDMA */
   write_str("*** Initialize IDMA following setting ***¥n");
   write_str("¥n");
   write_str("  IDMA ch.0 setting¥n");
   write_str("       LINK enable ... Next channel is 1¥n");
   write_str("       Transfer 1 time¥n");
   write_str("       128 half word transfer¥n");
   write_str("       Interrupt enable¥n");
   write_str("       Half word data size¥n");
   write_str("       Source and destination address increment¥n");
   write_str("       Block transfer mode¥n");
   /* Set IDMA ch.0 */                                                    (2)
   first_wd = IDMA_LNKEN_ENA | IDMA_LINK | IDMA_CNT | IDMA_BSIZE;
   second_wd = IDMA_DINTEN_ENA | IDMA_DATSIZ_HW| IDMA_SRINC_INC | IDMA_SRC_START0;
   third_wd = IDMA_DMOD_BLOCK | IDMA_DSINC_INC | IDMA_DST_START0;
   write_idma_info((unsigned long *)(&dma_control[0]), first_wd, second_wd, third_wd);

   write_str("¥n");
   write_str("  IDMA ch.1 setting¥n");
   write_str("       LINK disable¥n");
   write_str("       Transfer 1 time¥n");
   write_str("       128 half word transfer¥n");
   write_str("       Interrupt enable¥n");
   write_str("       Half word data size¥n");
   write_str("       Source and destination address increment¥n");
   write_str("       Block transfer mode¥n");
   /* Set IDMA ch.1 */                                                    (3)
   first_wd = IDMA_LNKEN_DIS | IDMA_CNT | IDMA_BSIZE;
   second_wd = IDMA_DINTEN_ENA |IDMA_DATSIZ_HW | IDMA_SRINC_INC | IDMA_SRC_START1;
   third_wd = IDMA_DMOD_BLOCK | IDMA_DSINC_INC | IDMA_DST_START1;
   write_idma_info((unsigned long *)(&dma_control[1]), first_wd, second_wd, third_wd);

   /* Intialize IDMA control information and start channel */
   start_ch = IDMA_ST_CH0;
   init_idma((unsigned long)dma_control, start_ch);                       (1)
                                     // Set IDMA control information and start ch.0

   /* Initialize IDMA interrupt flag */
   idmaint_flg = FALSE;

   /* Enable IDMA transfer */
   *(volatile unsigned char*)IDMA_DMAEN_ADDR |= 0x01;

   /* Start IDMA transfer */
   write_str("¥n");
   write_str("*** IDMA transfer starts by software trigger ***¥n");
   *(volatile unsigned char *)IDMA_DCHN_ADDR |= 0x80;                      (4)

   while (1) {
            if (idmaint_flg == TRUE) {
                     break;
            }
   }
```

```
   /* Disable IDMA transfer */
   *(volatile unsigned char *)IDMA_DMAEN_ADDR &= 0xfe;    // Disable IDMA transfer.

   /* Checking IDMA data */
   write_str("¥n");
   check_data(IDMA_SRC_START0, IDMA_DST_START0);
   write_str("¥n");
   check_data(IDMA_SRC_START1, IDMA_DST_START1);

   write_str("¥n");
   write_str("*** IDMA demonstration finish ***¥n");
}
       |
```

(1)コントロール情報領域の確保

次の構造体により、128チャネルのコントロール情報領域をRAM上に確保します。

```
struct {
      unsigned long data[3];
} dma_control[128];
                :
init_idma((unsigned long)dma_control, start_ch); // Set IDMA control information
```

この構造体のアドレスをdrv_idma.c内のinit_idma()でIDMAベースアドレスレジスタ(0x48200、0x48202)に設定しています。

```
void init_idma(unsigned long addr, unsigned char ch)
{
                :
      /* Set IDMA control information address */
      *(volatile unsigned long *)IDMA_DBASEL_ADDR = addr;
                :
```

このサンプルではch0とch1の2つのチャネルを以下のように設定しています。

(2)IDMA ch0の設定

ch0は0x6c0000～0x6c00ffのデータを0x6e0000～0x6e00ffに転送します。データサイズを16ビット、転送回数を128回に設定し、1回の転送で上記のアドレスのデータをコピーします。また、ch0の転送終了後、ch1が続けて起動するようにリンクを設定しています。

```
/* Set IDMA ch.0 */
first_wd = IDMA_LNKEN_ENA | IDMA_LINK | IDMA_CNT | IDMA_BSIZE;
second_wd = IDMA_DINTEN_ENA | IDMA_DATSIZ_HW| IDMA_SRINC_INC | IDMA_SRC_START0;
third_wd = IDMA_DMOD_BLOCK | IDMA_DSINC_INC | IDMA_DST_START0;
write_idma_info((unsigned long *)(&dma_control[0]), first_wd, second_wd, third_wd);
```

write_idma_info()はdrv_idma.c内にあります。

```
void write_idma_info(unsigned long *addrptr, unsigned long word1,
         unsigned long word2, unsigned long word3)
{
         /* Set IDMA control information */
         *addrptr = word1;              // Write 1st word
         *(addrptr + 1) = word2;        // Write 2nd word
         *(addrptr + 2) = word3;        // Write 3rd word
}
```

(3)IDMA ch1の設定

ch1は0x6d0000～0x6d00ffのデータを0x6f0000～0x6f00ffに転送します。データサイズを16ビット、転送回数を128回に設定し、1回の転送で上記のアドレスのデータをコピーします。ch1の転送終了後のリンクチャネルは設定していません。

```
/* Set IDMA ch.1 */
first_wd = IDMA_LNKEN_DIS | IDMA_CNT | IDMA_BSIZE;
second_wd = IDMA_DINTEN_ENA |IDMA_DATSIZ_HW | IDMA_SRINC_INC | IDMA_SRC_START1;
third_wd = IDMA_DMOD_BLOCK | IDMA_DSINC_INC | IDMA_DST_START1;
write_idma_info((unsigned long *)(&dma_control[1]), first_wd, second_wd, third_wd);
```

(4)転送の実行

IDMAスタートレジスタ(0x48204)に0x80を書き込み、ch0をソフトウェアトリガにより起動します。

```
*(volatile unsigned char *)IDMA_DCHN_ADDR |= 0x80;
```

ch0が設定したコントロール情報のとおりデータ転送を行い、その終了後、リンクによりch1が起動して同様にデータを転送します。

3.7 HSDMAの設定

ソフトウェアトリガでHSDMA ch1を起動してデータを転送する例を、gnu33\sample\drv33209\hsdma内のサンプルから抜粋して説明します。

HSDMA用ヘッダファイル　\sample\include\hsdma.h

```
                |
#define  HSDMA_HSDMA_ADDR   0x4029a    // Address for HSDMA software trigger register
                |
#define  HSDMA_HSD1_SOFT    0x0000     // HSDMA ch.1 software trigger
                |
#define  HSDMA_HST1         0x02       // HSDMA ch.1 software trigger
                |
#define  HSDMA_DUAL_DUAL    0x80000000 // HSDMA dual mode
                |
#define  HSDMA_DINTEN_ENA   0x80000000 // HSDMA interrupt enable
                |
#define  HSDMA_DATSIZE_HALF 0x40000000 // HSDMA half-word
                |
#define  HSDMA_DMOD_BLK     0x80000000 // HSDMA transfer mode
                |
#define  HSDMA_INC_INIT     0x20000000 // HSDMA address control Inc.(init)
                |
```

HSDMAの設定と起動　demo_hsdma.c

```
                |
#include "..¥include¥common.h"
#include "..¥include¥hsdma.h"

/* Prototype */
int main(void);
void fill_mem(unsigned long);
void check_data(unsigned long , unsigned long);
extern void write_str(char *);
extern void init_hsdma(unsigned char, unsigned char, unsigned long, unsigned long,
unsigned long);

/* Global variable define */
volatile int hdmaint_flg;              // High-speed DMA interrupt flag

/* HSDMA source and destination address */
#define     HSDMA_SRC_START   0x6c0000 // HSDMA ch.1 source start address
#define     HSDMA_DST_START   0x6d0000 // HSDMA ch.1 destination start address

/* HSDMA channel */
#define     HSDMA_CH1         1        // HSDMA ch.1

/************************************************************************
 * main
 *   Type :     void
 *   Ret val :  none
 *   Argument : void
 *   Function : High-speed DMA demonstration program.
 ************************************************************************/
int main(void)
{
   unsigned char trig;
   unsigned long mode, src, dst;

   write_str("*** High-speed DMA demonstration ***¥n");
   write_str("¥n");

   /* Fill memory */
   write_str("*** Filling memory area(address 0x6c0000 - 0x6c00ff) ***¥n");
   write_str("    Pattern  ... 0x00 0x01 .... 0xfe 0xff¥n");
   fill_mem(HSDMA_SRC_START);

   /* Initialize HSDMA */
   write_str("*** Initialize HSDMA following setting ***¥n");
   write_str("  HSDMA ch.1 setting¥n");
```

```
   write_str("       Transfer 1 time¥n");
   write_str("       128 half word transfer¥n");
   write_str("       Interrupt enable¥n");
   write_str("       Half word data size¥n");
   write_str("       Source and destination address increment¥n");
   write_str("       Block transfer mode¥n");

   /* Disable HSDMA transfer */                                          (1)
   *(volatile unsigned char *)HSDMA_HS1EN_ADDR &= 0xfe;   // Disable HSDMA transfer.

   /* HSDMA trigger mode */                                              (1)
   trig = HSDMA_HSD1_SOFT;                          // HSDMA ch.1 software trigger
   /* HSDMA mode and transfer count */                                   (1)
   mode = HSDMA_DUAL_DUAL | (1 << 8) | 0x80;      // HSDMA dual mode and transfer
count 1, block size 0x80
   /* HSDMA source address */                                            (1)
   src = HSDMA_DINTEN_ENA | HSDMA_DATSIZE_HALF | HSDMA_INC_INIT | HSDMA_SRC_START;
            // HSDMA interrupt enable, half word size, source address increment
(init.), source address 0x6c0000
   /* HSDMA destination address */
   dst = HSDMA_DMOD_BLK | HSDMA_INC_INIT | HSDMA_DST_START;              (1)
            // Destination address increment (init.), destination address 0x6d0000
   init_hsdma(HSDMA_CH1, trig, mode, src, dst);                          (1)

   /* Initialize HSDMA interrupt flag */
   hdmaint_flg = FALSE;

   /* Enable IDMA transfer */                                            (2)
   *(volatile unsigned char *)HSDMA_HS1EN_ADDR |= 0x01;   // Enable HSDMA transfer.

   /* Start HSDMA transfer */
   write_str("¥n");
   write_str("*** HSDMA transfer starts by software trigger ***¥n");
   *(volatile unsigned char *)HSDMA_HSDMA_ADDR |= HSDMA_HST1;            (2)

   while (1) {
            if (hdmaint_flg == TRUE) {
                     break;
            }
   }

   /* Disable HSDMA transfer */
   *(volatile unsigned char *)HSDMA_HS1EN_ADDR &= 0xfe;   // Disable HSDMA transfer.

   /* Checking HSDMA data */
   write_str("¥n");
   check_data(HSDMA_SRC_START, HSDMA_DST_START);

   write_str("¥n");
   write_str("*** HSDMA demonstration finish ***¥n");
}
             |
```

(1) HSDMAの設定

HSDMAの設定の前に、必ずHSDMAをディセーブルにしてください。
動作中は、レジスタが正しく読み書きできない場合があります。

```
*(volatile unsigned char *)HSDMA_HS1EN_ADDR &= 0xfe;   // Disable HSDMA transfer.
```

HSDMA ch1を以下のとおり設定します。

- トリガ　ソフトウェアトリガ
- アドレスモード　デュアルアドレスモード
- 転送モード　ブロック転送モード
- データサイズ　ハーフワード
- 転送元/転送先アドレス制御　インクリメント(初期化付き)
- 転送カウンタ　128
- 転送元アドレス　0x6c0000
- 転送先アドレス　0x6d0000

```
   /* HSDMA trigger mode */
   trig = HSDMA_HSD1_SOFT;                          // HSDMA ch.1 software trigger
   /* HSDMA mode and transfer count */
   mode = HSDMA_DUAL_DUAL | (1 << 8) | 0x80;       // HSDMA dual mode and transfer
count 1, block size 0x80
   /* HSDMA source address */
   src = HSDMA_DINTEN_ENA | HSDMA_DATSIZE_HALF | HSDMA_INC_INIT | HSDMA_SRC_START;
            // HSDMA interrupt enable, half word size, source address increment
(init.), source address 0x6c0000
   /* HSDMA destination address */
   dst = HSDMA_DMOD_BLK | HSDMA_INC_INIT | HSDMA_DST_START;
            // Destination address increment (init.), destination address 0x6d0000
   init_hsdma(HSDMA_CH1, trig, mode, src, dst);
```

(2)転送の実行

設定終了後、HSDMAをイネーブルにします。

```
   /* Enable IDMA transfer */
   *(volatile unsigned char *)HSDMA_HS1EN_ADDR |= 0x01;   // Enable HSDMA transfer.
```

ソフトウェアトリガレジスタ(0x4029a)のch1ソフトウェアトリガビット(D1)を"1"に設定してHSDMA ch1を起動します。

```
   /* Start HSDMA transfer */
   write_str("¥n");
   write_str("*** HSDMA transfer starts by software trigger ***¥n");
   *(volatile unsigned char *)HSDMA_HSDMA_ADDR |= HSDMA_HST1;
```

3.8　クロックの設定

S1C33209には、1/128秒単位で最大64K日までカウントできる計時タイマが内蔵されています。
ここでは、計時タイマで1分後にアラーム割り込みを起こす方法について説明します。
サンプルとして、gnu33\sample\drv33209\ct内のプログラム例を使用します。

● 1分後のアラーム割り込み

ベクタテーブル　vector.c

```
/* vector table */
const unsigned long vector[] = {
       (unsigned long)boot,         // 0   0
              |
              |
       (unsigned long)dummy,        // 252 63
       (unsigned long)dummy,        // 256 64
       (unsigned long)int_ct,       // 260 65                                  (1)
       (unsigned long)dummy,        // 264 66
              |
};
```

(1)ベクタテーブルの設定
　計時タイマの割り込みベクタとして、割り込み処理ルーチン int_cを登録します。

初期設定　drv_ct.c

```
#include "..¥include¥common.h"
#include "..¥include¥ct.h"
#include "..¥include¥int.h"

/* Prototype */
void init_ct(void);
void int_ct(void);
extern void save_psr(void);
extern void restore_psr(void);

/*****************************************************************************
 * init_ct
 *   Type :          void
 *   Ret val :       none
 *   Argument :      void
 *   Function :      Initialize clock timer to use real time clock.
 *****************************************************************************/
void init_ct(void)
{
       /* Save PSR and disable all interrupt */
       save_psr();                                                             (1)

       /* Set clock timer interrupt disable on interrupt controller */
       *(volatile unsigned char *)INT_EADE_ECTM_EP4_ADDR = INT_ENABLE_DIS;
              // Set clock timer interrupt disable

       /* Stop clock timer */
       *(volatile unsigned char *)CT_TCRUN_ADDR &= 0xfe;                       (2)

       /* Reset clock timer */
       *(volatile unsigned char *)CT_TCRUN_ADDR |= CT_TCRST_RST;

       /* Set clock timer data (1999.01.01 21:05) */
       *(volatile unsigned char *)CT_TCHD_ADDR = 0x05;                         (3)
              // Minute data (5 minutes)
       *(volatile unsigned char *)CT_TCDD_ADDR = 0x15;
              // Hour data (21 hours)
       *(volatile unsigned char *)CT_TCNDL_ADDR = 0xd7;
              // Year-month-day low byte data (3287 days)
       *(volatile unsigned char *)CT_TCNDH_ADDR = 0x0c;
              // Year-month-day high byte data (3287 days)
```

```
        /* Set clock timer comparison data */
        *(volatile unsigned char *)CT_TCCH_ADDR = 0x06;                          (4)
                // Minute comparison data (6 minutes)
        *(volatile unsigned char *)CT_TCCD_ADDR = 0x0;
                // Hour comparison data (0 hour)
        *(volatile unsigned char *)CT_TCCN_ADDR = 0x0;
                // Day comparison data (0 day)

        /* Set clock timer interrupt factor control flag */
        *(volatile unsigned char *)CT_TCAF_ADDR                                  (5)
                = CT_TCISE_NONE | CT_TCASE_M | CT_TCIF_RST | CT_TCAF_RST;

        /* Set clock timer interrupt priority level 3 on interrupt controller */
        *(volatile unsigned char *)INT_PCTM_ADDR = INT_PRIL_LVL3;                (6)

        /* Reset clock timer interrupt factor flag on interrupt controller */
        *(volatile unsigned char *)INT_FADE_FCTM_FP4_ADDR = INT_FCTM;
                // Reset clock timer interrupt factor flag

        /* Set clock timer interrupt enable on interrupt controller */
        *(volatile unsigned char *)INT_EADE_ECTM_EP4_ADDR = INT_ECTM;
                // Set clock timer interrupt enable

        /* Restore PSR */
        restore_psr();                                                           (7)
}
```

(1)割り込みを禁止

PSRを待避して、IEで割り込みをマスクします。

```
/* Save PSR and disable all interrupt */
save_psr();
```

割り込みコントローラで計時タイマ割り込みを禁止します。

```
/* Set clock timer interrupt disable on interrupt controller */
*(volatile unsigned char *)INT_EADE_ECTM_EP4_ADDR = INT_ENABLE_DIS;
        // Set clock timer interrupt disable
```

(2)計時タイマをリセット

計時タイマを停止後、カウンタをリセットします。

```
/* Stop clock timer */
*(volatile unsigned char *)CT_TCRUN_ADDR &= 0xfe;

/* Reset clock timer */
*(volatile unsigned char *)CT_TCRUN_ADDR |= CT_TCRST_RST;
```

(3)日付と時刻の設定

1999年1月1日 21時5分に設定します。日カウンタに設定する3287日は、1990年1月1日を起点として計算した日数です。

```
/* Set clock timer data (1999.01.01 21:05) */
*(volatile unsigned char *)CT_TCHD_ADDR = 0x05;
        // Minute data (5 minutes)
*(volatile unsigned char *)CT_TCDD_ADDR = 0x15;
        // Hour data (21 hours)
*(volatile unsigned char *)CT_TCNDL_ADDR = 0xd7;
        // Year-month-day low byte data (3287 days)
*(volatile unsigned char *)CT_TCNDH_ADDR = 0x0c;
        // Year-month-day high byte data (3287 days)
```

(4)アラームの設定

ここでは、6分を比較データとして設定し、1分後にアラーム割り込みが発生するようにします。

```
/* Set clock timer comparison data */
*(volatile unsigned char *)CT_TCCH_ADDR = 0x06;
        // Minute comparison data (6 minutes)
*(volatile unsigned char *)CT_TCCD_ADDR = 0x0;
        // Hour comparison data (0 hour)
*(volatile unsigned char *)CT_TCCN_ADDR = 0x0;
        // Day comparison data (0 day)
```

(5)アラーム割り込み用の設定

分のアラーム割り込みのみを有効にします。また、割り込み要因発生フラグとアラーム要因発生フラグをクリアしておきます。

```
/* Set clock timer interrupt factor control flag */
*(volatile unsigned char *)CT_TCAF_ADDR
       = CT_TCISE_NONE | CT_TCASE_M | CT_TCIF_RST | CT_TCAF_RST;
```

これは計時タイマ内の機能で、割り込みコントローラの設定ではありません。
この制御レジスタは初期値が保証されませんので、必ずリセットしてください。

(6)割り込みコントローラの設定

割り込みレベルを3に設定します。

```
/* Set clock timer interrupt priority level 3 on interrupt controller */
*(volatile unsigned char *)INT_PCTM_ADDR = INT_PRIL_LVL3;
```

計時タイマの割り込み要因フラグをクリアします。

```
/* Reset clock timer interrupt factor flag on interrupt controller */
*(volatile unsigned char *)INT_FADE_FCTM_FP4_ADDR = INT_FCTM;
       // Reset clock timer interrupt factor flag
```

計時タイマ割り込みを許可します。

```
/* Set clock timer interrupt enable on interrupt controller */
*(volatile unsigned char *)INT_EADE_ECTM_EP4_ADDR = INT_ECTM;
       // Set clock timer interrupt enable
```

なお、計時タイマの割り込みにはIDMAを要求するフラグはなく、CPUへの割り込み機能のみです。

(7)リターン処理

PSRを復帰して、割り込みを許可します。

```
/* Restore PSR */
restore_psr();
```

割り込み処理　drv_ct.c

```
/*****************************************************************************
 * int_ct
 *   Type :       void
 *   Ret val :    none
 *   Argument :   void
 *   Function :   Clock timer interrupt function.
 *****************************************************************************/
void int_ct(void)
{
       extern volatile int ctint_flg;

       INT_BEGIN;                                                          (1)
       ctint_flg = TRUE;     // Clock timer interrupt flag on              (2)
       *(volatile unsigned char *)INT_FADE_FCTM_FP4_ADDR = INT_FCTM;       (3)
              // Reset clock timer interrupt factor flag
       INT_END;                                                            (1)
}
```

(1)レジスタの待避/復帰

処理ルーチンの最初と最後で行うレジスタの待避/復帰には、common.h内に定義されているINT_BEGINとINT_ENDを使用しています。

```
#define INT_BEGIN     asm("pushn     %r15")
#define INT_END       asm("popn      %r15¥n reti")
```

(2)割り込み発生確認用フラグのセット

割り込みが発生したことを上位ルーチンに知らせるフラグをセットします。

```
ctint_flg = TRUE;                    // Clock timer interrupt flag on
```

(3)割り込み要因フラグのリセット

割り込み要因フラグをクリアします。

```
*(volatile unsigned char *)INT_FADE_FCTM_FP4_ADDR = INT_FCTM;
       // Reset clock timer interrupt factor flag
```

アプリケーション部　demo_ct.c

```
        |
/* Initialize clock timer */
write_str("*** Initialize clock timer and start ***¥n");
write_str("    Today date and time (1999.01.01 21:05)¥n");
write_str("    Set minute alarm interrupt enable (6 minutes)¥n");
init_ct();                                                          (1)

/* Run clock timer */
write_str("*** Run clock timer ***¥n");
*(volatile unsigned char *)CT_TCRUN_ADDR |= 0x01;                   (2)

/* Initialize clock timer interrupt flag */
ctint_flg = FALSE;

write_str("*** Wait 1 minute ***¥n");
write_str("¥n");

while (1) {                                                         (3)
       if (ctint_flg == TRUE) {
             break;
       }
}

/* Stop clock timer */
write_str("*** Stop clock timer ***¥n");
*(volatile unsigned char *)CT_TCRUN_ADDR &= 0xfe;                   (4)
        |
```

(1)初期設定
前述の init_ct()を呼び出して、計時タイマと割り込みの初期設定を行います。

(2)計時タイマをスタート
計時タイマをスタートし、割り込み発生確認用フラグをクリアします。

```
*(volatile unsigned char *)CT_TCRUN_ADDR |= 0x01;

/* Initialize clock timer interrupt flag */
ctint_flg = FALSE;
```

(3)アラーム割り込み待ち
アラーム時間になると前述の割り込み処理ルーチンint_ct( )が呼ばれ、処理が終了するとフラグctint_flgがセットされます。そのフラグが1になるまでループします。アラーム割り込みは計時タイマスタートから1分後に発生します。

```
while (1) {
       if (ctint_flg == TRUE) {
             break;
       }
}
```

(4)計時タイマを停止
割り込み発生後、計時タイマを停止します。

```
*(volatile unsigned char *)CT_TCRUN_ADDR &= 0xfe;
```

3.9 SLEEP

ここでは、SLEEPモード移行前の処理とアラーム機能を使用したSLEEP解除の方法を、S5U1C33000C ver.3以降のgnu33\sample\drv33209\osc内のサンプルを使用して説明します。

メインルーチン　demo_osc.c

```
int main(void)
{
        unsigned char  pwr;    /* Power control register data */
        unsigned char  clk;    /* Clock control register data */

        write_str("*** OSC demonstration ***¥n");
        write_str("¥n");

        /* OSC3 high-speed mode */
        write_str("*** OSC3 high-speed mode ***¥n");
        write_str("     System clock select 1/1, Prescaler output ON, CPU clock OSC3,
              OSC3 ON, OSC1 ON¥n");
        write_str("     HALT clock option OFF, OSC3-stabilize waiting function ON¥n");
        write_str("     OSC1 external output control OFF¥n");
        pwr = OSC_CLKDT_11 | OSC_PSCON_ON | OSC_CLKCHG_OSC3 | OSC_SOSC3_ON |      (1)
              OSC_SOSC1_ON;
        clk = OSC_HALT2OP_OFF | OSC_8T1ON_ON | OSC_PF1ON_OFF;
        set_OSC(pwr, clk);

        /* If you use sleep mode, you set OSC3-stabilize waiting function on
              and run 8-bit timer 1 */
        /* Initialize 8-bit timer */
        write_str("*** Initialize 8-bit timer ***¥n");
        write_str("     8-bit timer 1 ... CLK/4096¥n");
        write_str("     8-bit timer 1 reload data
              ... 0x62 (10ms on OSC3 clock 40MHz)¥n");
        init_8timer1();                                                          (2)

        /* Initialize clock timer */
        write_str("*** Initialize clock timer and start ***¥n");
        write_str("     Today data and time (1999.01.01 21:05)¥n");
        write_str("     Set minute alarm interrupt enable (6 minutes)¥n");
        init_ct();                                                               (3)

        /* Run clock timer */
        write_str("*** Run clock timer ***¥n");
        *(volatile unsigned char *)CT_TCRUN_ADDR |= 0x01;                        (4)

        write_str("*** Wait 1 minute ***¥n");

        write_str("*** Sleep mode ***¥n");
        write_str("¥n");

        /*Run 8-bit timer 1 */
        run_8timer(T8P_PTRUN1_ADDR);                                             (5)

        /* Sleep */
        asm("slp");                                                              (6)

        /* Stop 8-bit timer 1 */
        write_str("*** Return to OSC3 high-speed mode from sleep mode ***¥n");

        write_str("¥n");
        write_str("*** OSC demonstration finish ***¥n");
}
```

(1)発振回路の設定

set_osc()を呼び出し、以下の設定を行います。

```
pwr = OSC_CLKDT_11 | OSC_PSCON_ON | OSC_CLKCHG_OSC3 | OSC_SOSC3_ON | OSC_SOSC1_ON;
clk = OSC_HALT2OP_OFF | OSC_8T1ON_ON | OSC_PF1ON_OFF;
set_OSC(pwr, clk);
```

OSC_CLKDT_11	システムクロック分周比 = 1/8
OSC_PSCON_ON	プリスケーラON
OSC_CLKCHG_OSC3	CPU動作クロック = OSC3
OSC_SOSC3_ON	高速(OSC3)発振ON
OSC_SOSC1_ON	低速(OSC1)発振ON
OSC_HALT2OP_OFF	HALT2モードOFF
OSC_8T1ON_ON	SLEEP解除後高速(OSC3)発振待ち機能ON
OSC_PF1ON_OFF	OSC1クロック外部出力OFF *

* OSC1クロックの外部出力は、OSC1ブロックからFOSC端子の内部配線のクロックもOFFにして、SLEEPモード時の消費電流を少しでも低減させるために停止しています。必要に応じてSLEEP時もOSC1クロックを出力させることができます。その場合は、P14/DCLK/FOSC1端子をOSC1クロック出力(FOSC1)に設定しておく必要があります。

(2)8ビットタイマ1の初期化

SLEEP解除後に高速(OSC3)発振待ち機能を使用しますので、8ビットタイマ1にその待ち時間を設定します。この処理はinit_8timer1()で行います。

(3)計時タイマの設定

init_ct()を呼び出して、計時タイマをスタートから1分後にアラーム割り込みが発生するように設定します。init_ct()の処理内容については、"3.8 クロックの設定"を参照してください。

(4)計時タイマをスタート

計時タイマをスタートさせます。

```
*(volatile unsigned char *)CT_TCRUN_ADDR |= 0x01;
```

(5)8ビットタイマ1をスタート

run_8timer()を呼び出して8ビットタイマ1をスタートさせます。

(drv_8timer.c)

```
void run_8timer(unsigned long reg)
{
        *(volatile unsigned char *)reg |= 0x01;
}
```

(6)SLEEP

SLP命令を実行し、SLEEPモードにします。高速(OSC3)発振回路は停止します。

```
asm("slp");
```

(7)SLEEP解除

SLEEPモードでも低速(OSC1)発振回路を計時タイマは動作し、設定したアラーム割り込みの発生によりSLEEPモードは解除されます。SLEEP解除により高速(OSC3)発振回路が動作を開始しますが、プログラムの実行は、8ビットタイマ1に設定した発振安定待ち時間(10ms)が経過後に再開します。

このプログラムをS5U1C33208D3で実行して電流を測定した結果は、通常動作時が65mA、SLEEP時が35mAと、約30mAダウンしました。

発振回路の設定 drv_osc.c

```
void set_osc(unsigned char pwr, unsigned char clk)
{
        /* Before power control register write access,
               set power control register protect flag write enable */
        *(volatile unsigned char *)OSC_CLGP_ADDR = OSC_CLGP_ENA;            (1)

        /* Set power control register */
        *(volatile unsigned char *)OSC_SOSC_ADDR = pwr;

        /* Before clock control register write access,
               set power control register protect flag write enable */
        *(volatile unsigned char *)OSC_CLGP_ADDR = OSC_CLGP_ENA;            (2)

        /* Set clock control register */
        *(volatile unsigned char *)OSC_PF1ON_ADDR = clk;
}
```

(1)パワーコントロールレジスタの設定

パワーコントロールレジスタ(0x40180)の書き込み保護を解除し、main()から渡された設定値を書き込みます。

```
/* Before power control register write access,
       set power control register protect flag write enable */
*(volatile unsigned char *)OSC_CLGP_ADDR = OSC_CLGP_ENA;

/* Set power control register */
*(volatile unsigned char *)OSC_SOSC_ADDR = pwr;
```

(2)クロックオプションレジスタの設定

クロックオプションレジスタ(0x40190)の書き込み保護を解除し、main()から渡された設定値を書き込みます。

```
/* Before clock control register write access,
       set power control register protect flag write enable */
*(volatile unsigned char *)OSC_CLGP_ADDR = OSC_CLGP_ENA;

/* Set clock control register */
*(volatile unsigned char *)OSC_PF1ON_ADDR = clk;
```

8ビットタイマ1の設定 drv_8timer.c

```
void init_8timer1(void)
{
        /* Save PSR and disable all interrupt */
        save_psr();                                                         (1)

        /* Set 8bit timer1 prescaler */
        *(volatile unsigned char *)PRESC_P8TS0_P8TS1_ADDR                   (2)
               |= (PRESC_PTONH_ON | PRESC_CLKDIVH_SEL7);
               // Set 8bit timer1 prescaler (CLK/4096)

        /* Set 8bit timer1 reload data */
        *(volatile unsigned char *)T8P_RLD1_ADDR = 0x62;                    (3)
               // Set reload data (0x62 ... 10ms on OSC3 clock 40MHz)

        /* Set 8bit timer1 clock output off, preset and timer stop */
        *(volatile unsigned char *)T8P_PTRUN1_ADDR
               = T8P_PTOUT_OFF | T8P_PSET_ON | T8P_PTRUN_STOP;

        /* Set 8bit timer1 interrupt CPU request on interrupt controller */
        *(volatile unsigned char *)INT_R16T5_R8TU_RS0_ADDR = INT_RIDMA_DIS; (4)
               // IDMA request disable and CPU request enable

        /* Set 8bit timer1 interrupt priority level 3 on interrupt controller */
        *(volatile unsigned char *)INT_P8TM_PSIO0_ADDR = INT_PRIL_LVL3;

        /* Reset 8bit timer1 interrupt factor flag on interrupt controller */
        *(volatile unsigned char *)INT_F8TU_ADDR = INT_F8TU1;
               // Reset 8bit timer1 underflow interrupt factor flag
```

```
        /* Set 8bit timer1 interrupt disable on interrupt controller */
        *(volatile unsigned char *)INT_E8TU_ADDR &=~INT_E8TU1;
                // Set 8bit timer1 underflow interrupt disable

        /* Restore PSR */
        restore_psr();                                                    (5)
}
```

(1)割り込みを禁止

PSRを待避して、IEで割り込みをマスクします。

```
/* Save PSR and disable all interrupt */
save_psr();
```

(2)プリスケーラの設定

プリスケーラの分周比を1/4096に設定します。

```
/* Set 8bit timer1 prescaler */
*(volatile unsigned char *)PRESC_P8TS0_P8TS1_ADDR
        |= (PRESC_PTONH_ON | PRESC_CLKDIVH_SEL7);
        // Set 8bit timer1 prescaler (CLK/4096)
```

(3)8ビットタイマの設定

リロードデータとして0x62を設定します。この値は、40MHzの動作クロックの場合に約10msのOSC3発振安定待ち時間を生成します。

25μs (=1/40MHz) × 4096 × (0x62 + 1) = 約10ms

```
/* Set 8bit timer1 reload data */
*(volatile unsigned char *)T8P_RLD1_ADDR = 0x62;
        // Set reload data (0x62 ... 10ms on OSC3 clock 40MHz)
```

上記のリロードデータをカウンタにプリセットします。ここではまだタイマをスタートさせません。

```
/* Set 8bit timer1 clock output off, preset and timer stop */
*(volatile unsigned char *)T8P_PTRUN1_ADDR
        = T8P_PTOUT_OFF | T8P_PSET_ON | T8P_PTRUN_STOP;
```

(4)割り込みコントローラの設定

8ビットタイマ1の割り込みによるIDMAの起動を禁止します。

```
/* Set 8bit timer1 interrupt CPU request on interrupt controller */
*(volatile unsigned char *)INT_R16T5_R8TU_RS0_ADDR = INT_RIDMA_DIS;
        // IDMA request disable and CPU request enable
```

8ビットタイマの割り込みプライオリティレベルを3に設定します。

```
/* Set 8bit timer1 interrupt priority level 3 on interrupt controller */
*(volatile unsigned char *)INT_P8TM_PSIO0_ADDR = INT_PRIL_LVL3;
```

8ビットタイマ1の割り込み要因フラグをリセットします。

```
/* Reset 8bit timer1 interrupt factor flag on interrupt controller */
*(volatile unsigned char *)INT_F8TU_ADDR = INT_F8TU1;
        // Reset 8bit timer1 underflow interrupt factor flag
```

8ビットタイマ1の割り込みはディセーブルに設定しておきます。

```
/* Set 8bit timer1 interrupt disable on interrupt controller */
*(volatile unsigned char *)INT_E8TU_ADDR &=~INT_E8TU1;
        // Set 8bit timer1 underflow interrupt disable
```

(5)リターン処理

PSRを復帰して、割り込みを許可します。

```
/* Restore PSR */
restore_psr();
```

3.10 SDRAMコントローラ

現在SDRAMコントローラを内蔵している機種は以下のとおりです。
S1C33L03, S1C33205

以下に、SDRAMを実際に使用する場合の初期設定プログラム例を示します。

4Mワード×16ビット×4バンク(32MB) SDRAM使用時の初期設定例

```
;*****************************************************************************
;
;   Copyright (C) SEIKO EPSON CORP. 2002
;   All rights Reserved
;
;   File name : SDRAM_led.s
;
;    Revision  :
;       2002/10/01  M.Toki start
;       2003/04/18  CH.Yoon Port to GNU
;*****************************************************************************
    .text
    .long   START

START:

INIT_SDRAM_32MB:
;;;----------------------- SDRAM access configuration ---------------------------------
;;;*****************************************************************
;;;**************** C33 macro setting part ***********************
;;;*****************************************************************
;;; set CEFUNC to use #CE13/14 (upper area)
    xld.w %r0,0x48131
    bset [%r0],0x1
;;; set area 6,13,14 to internal access
    xld.w %r0,0x48132
    xld.w %r1,0x2200
    ld.h [%r0],%r1
;;; area 6 -> output disable 0.5, wait 2
    xld.w %r0,0x4812A
    xld.w %r1,0x0237
    ld.h [%r0],%r1
;;; available #WAIT
    xld.w %r0,0x04812E
    bset [%r0],0x0
;;; area 13,14 -> 8bit device, output disable 2.5, wait 0
    xld.w %r0,0x048122
    xld.w %r1,0x30
    ld.h [%r0],%r1

;;;*****************************************************************
;;;************** SDRAM Controller REG setting part ****************
;;;*****************************************************************
;;;-------------------------------------------------
;;;  area13 0x2000000 - 0x2FFFFFF(16MB)
;;;  area14 0x3000000 - 0x3FFFFFF(16MB)
;;;-------------------------------------------------
;/////////////////////////////////////////
;;; SDRAM area configuration register
    xld.w %r0,0x39FFC0 ;
    xld.w %r1,0xc8     ; set area13&14 to SDRAM area, #SDCE0(#CE13) available
    ld.b [%r0],%r1     ; (32MB area available)
;/////////////////////////////////////////
;;; SDRAM control register
;;; xld.w %r0,0x39FFC1 ;
;;; xld.w %r1,0xff     ; SDRAM self-refresh -> disable, initial sequence ->PRE REF MRS
;;; ld.b [%r0],%r1     ; Little endian
;/////////////////////////////////////////
;;; SDRAM address configuration register
    xld.w %r0,0x39FFC2 ;
    xld.w %r1,0x2a     ; col 512 / row 8K / bank 4 -> 256Mb[32MB] available
    ld.b [%r0],%r1     ;
;/////////////////////////////////////////
```

```
;;; SDRAM mode set-up register
    xld.w %r0,0x39FFC3 ;
    xld.w %r1,0x40     ; 2 CAS Latency ,burst length = 1
    ld.b [%r0],%r1     ;
;////////////////////////////////////////
;;; SDRAM timing set-up register 1
    xld.w %r0,0x39FFC4 ;
    xld.w %r1,0x4A     ; Tras=2,Trp=1,Trc=2        ... x1スピードモード, 25MHz時における推奨設定です
    ld.b [%r0],%r1     ;
;////////////////////////////////////////
;;; SDRAM timing set-up register 2
    xld.w %r0,0x39FFC5 ;
    xld.w %r1,0x48     ; Trcd=1,Trsc=2,Trrd=1
    ld.b [%r0],%r1     ;
;////////////////////////////////////////
;;; SDRAM auto refresh count low-order register
;;; xld.w %r0,0x39FFC6 ;
;;; xld.w %r1,0xff     ;
;;; ld.b [%r0],%r1     ;
;////////////////////////////////////////
;;; SDRAM auto refresh count high-order register
    xld.w %r0,0x39FFC7 ;
    xld.w %r1,0x00     ;
    ld.b [%r0],%r1     ;
;////////////////////////////////////////
;;; SDRAM self refresh count register
;;; xld.w %r0,0x39FFC8 ;
;;; xld.w %r1,0x0f     ;
;;; ld.b [%r0],%r1     ;
;////////////////////////////////////////
;;; SDRAM advanced control register
    xld.w %r0,0x39FFC9 ;
    xld.w %r1,0x20     ; data width -> 16bit, bank interleave -> on
    ld.b [%r0],%r1     ;

;;;****************************************************************
;;;**************** SDRAM controller power up *********************
;;;****************************************************************
    xld.w %r0,0x39FFC1 ; SDRAM control register
    xld.w %r1,0x39FFCA ; SDRAM status register
    xld.w %r2,0x0
    xld.w %r3,0x10

;;; enable SDRAM signal
    bset [%r0],0x7     ; set SDRENA[D7/0x39FFC1]

SDRAM_SIGNAL_EN:
    add %r2,0x1        ; SDRAM signal enable waiting loop
    cmp %r2,%r3
    xjrne SDRAM_SIGNAL_EN

;;; SDRAM power up
    bset [%r0],0x6     ; set SDRINI[D6/0x39FFC1]

POWER_UP:
    btst [%r1],0x7     ; SDRAM power-up waiting loop
    xjrne POWER_UP

;;;---------------------- end of SDRAM access configuration -------------------------

;;;*******************************************************************************
;;;************************** Program translating ********************************
;;;*******************************************************************************
; Transfer data from 0x00c00400 to 0x00001000
    xld.w  %r9,0x2000000    ; r9 = 0x2000000 : destination address
    xld.w  %r8,0x0001000    ; r8 = 0x0001000 : source      address
    xld.w  %r7,0x0000000    ; reference data ( "nop,nop" of end -> 0x00000000)

DATA_TRANSFER:
    ld.w   %r6,[%r8]+       ; data transfer FROM iRAM to SDRAM
    ld.w  [%r9]+,%r6        ;
```

```
    cmp    %r6,%r7             ; compare the instruction code with 0x00000000(nop,nop)
    xjrne  DATA_TRANSFER       ; if instruction code = "nop,nop", exit this loop
;;;*****************************************************************************
    xld.w   %r9,0x2000000      ; r9 = 0x2000000 :
    jp      %r9                ; jp to "LED on/off loop" of SDRAM
;;;------------------------- end of program translating ---------------------------


;*****************************************************************************
;
;    Target source program    - LED ON/OFF - program
;
;*****************************************************************************
main:
    xld.w   %r10,0x402d2
    xld.w   %r12,0x08
    ld.b    [%r10],%r12        ; P7 I/O port INPUT mode
    xld.w   %r10,0x402d1
    xld.w   %r12,0x00
    ld.b    [%r10],%r12        ; LED ON

    xld.w   %r13,0x100000      ; wait counter
loop:
    sub     r13,0x1
    xjrgt   loop

    xld.w   %r12,0x08
    ld.b    [%r10],%r12        ; LED OFF

    xld.w   %r13,0x100000      ; wait counter
loop2:                         ; loop for wait
    sub     %r13,0x1
    xjrgt   loop2

    xld.w   %r12,0x00
    ld.b    [%r10],%r12        ; LED ON

    xld.w   %r13,0x100000      ; wait counter
loop3:                         ; loop for wait
    sub     %r13,0x1
    xjrgt   loop3

    xjp     loop               ; jp to "loop"
;*****************************************************************************
    nop                        ; program end indicator
    nop                        ;
    nop                        ;
    nop                        ;
```

2Mワード × 16ビット × 4バンク(16MB) SDRAM使用時の初期設定例

```
INIT_SDRAM_16MB:
;;;----------------------- SDRAM access configuration ---------------------------------
;;;*****************************************************************
;;;***************** C33 macro setting part ************************
;;;*****************************************************************

;;; set CEFUNC to use #CE13/14 (upper area)
        xld.w   %r0,0x48131
        bset    [%r0],0x1

;;; set area 6,13,14 to internal access
        xld.w   %r0,0x48132
        xld.w   %r1,0x2200
        ld.h    [%r0],%r1

;;; area 6 -> output disable 0.5, wait 2
        xld.w   %r0,0x4812A
        xld.w   %r1,0x0237
        ld.h    [%r0],%r1

;;; available #WAIT
        xld.w   %r0,0x04812E
        bset    [%r0],0x0

;;; area 13,14 -> 16bit device, output disable 2.5, wait 0
        xld.w   %r0,0x048122
        xld.w   %r1,0x30
        ld.h    [%r0],%r1

;;;*****************************************************************
;;;************** SDRAM Controller REG setting part ****************
;;;*****************************************************************
;;;-------------------------------------------------
;;;area13       0x2000000 - 0x2FFFFFF(16MB)
;;;area14       0x3000000 - 0x3FFFFFF(16MB)
;;;-------------------------------------------------
;/////////////////////////////////////////
;;; SDRAM area configuration register
        xld.w   %r0,0x39FFC0     ;
        xld.w   %r1,0x88         ; set area13 to SDRAM area, #SDCE0(#CE13) available
        ld.b    [%r0],%r1        ; (16MB area available)
;/////////////////////////////////////////
;;; SDRAM control register
;;;     xld.w   %r0,0x39FFC1     ;
;;;     xld.w   %r1,0xff         ; SDRAM self-refresh -> disable, initial sequence ->PRE REF MRS
;;;     ld.b    [%r0],%r1        ; Little endian
;/////////////////////////////////////////
;;; SDRAM address configuration register
        xld.w   %r0,0x39FFC2     ;
        xld.w   %r1,0x26         ; col 512 / row 4K / bank 4 -> 128Mb[16MB] available
        ld.b    [%r0],%r1        ;
;/////////////////////////////////////////
;;; SDRAM mode set-up register
        xld.w   %r0,0x39FFC3     ;
        xld.w   %r1,0x40         ; 2 CAS Latency ,burst length = 1
        ld.b    [%r0],%r1        ;
;/////////////////////////////////////////
;;; SDRAM timing set-up register 1
        xld.w   %r0,0x39FFC4     ;
        xld.w   %r1,0x4A         ; Tras=2,Trp=1,Trc=2    ... x1スピードモード, 25MHz時における推奨設定です
        ld.b    [%r0],%r1        ;
;/////////////////////////////////////////
;;; SDRAM timing set-up register 2
        xld.w   %r0,0x39FFC5     ;
        xld.w   %r1,0x48         ; Trcd=1,Trsc=2,Trrd=1
        ld.b    [%r0],%r1        ;
;/////////////////////////////////////////
;;; SDRAM auto refresh count low-order  register
;;;     xld.w   %r0,0x39FFC6     ;
;;;     xld.w   %r1,0xff         ;
```

```
;;;     ld.b    [%r0],%r1        ;
;/////////////////////////////////////////
;;; SDRAM auto refresh count high-order register
        xld.w   %r0,0x39FFC7     ;
        xld.w   %r1,0x00         ;
        ld.b    [%r0],%r1        ;
;/////////////////////////////////////////
;;; SDRAM self refresh count register
;;;     xld.w   %r0,0x39FFC8     ;
;;;     xld.w   %r1,0x0f         ;
;;;     ld.b    [%r0],%r1        ;
;/////////////////////////////////////////
;;; SDRAM advanced control register
        xld.w   %r0,0x39FFC9     ;
        xld.w   %r1,0x20         ; data width -> 16bit, bank interleave -> on
        ld.b    [%r0],%r1        ;

;;;*************************************************************
;;;***************** SDRAM controller power up *****************
;;;*************************************************************
        xld.w   %r0,0x39FFC1     ; SDRAM control register
        xld.w   %r1,0x39FFCA     ; SDRAM status  register
        xld.w   %r2,0x0
        xld.w   %r3,0x10

;;; enable SDRAM signal
        bset    [%r0],0x7        ; set SDRENA[D7/0x39FFC1]
SDRAM_SIGNAL_EN:
        add     %r2,0x1          ; SDRAM signal enable waiting loop
        cmp     %r2,%r3
        jrne    SDRAM_SIGNAL_EN

;;; SDRAM power up
        bset    [%r0],0x6        ; set SDRINI[D6/0x39FFC1]
POWER_UP:
        btst    [%r1],0x7        ; SDRAM power-up waiting loop
        jrne    POWER_UP

;;;------------------------ end of SDRAM access configuration ---------------------------
ret
```

以上で、SDRAM にアクセスすることが可能になります。

●注意事項

(1) SDRAMコントローラが割り付けられているエリア6は、アクセス時に2ウェイトサイクルが挿入されるように設定してください。これ以外のウェイト数では、制御レジスタへの正常な書き込みができない場合があります。

(2) SDRAMに使用するエリアは内部アクセス(A8IO(0x48132・DA) = "1"またはA14IO(0x48132・DD) = "1")に設定してください。

(3) HALT2モードとSLEEPモードではSDRAMのオートリフレッシュができなくなりますので、それぞれのモードに入る前に、SDRAMをセルフリフレッシュモードに設定しておく必要があります。この場合、SDRSRM(0x39FFCA・D6)が"0"(SDRAMがセルフリフレッシュ中)になったことを確認した後でhaltまたはslp命令を実行します。
なお、セルフリフレッシュ中にSDRAMへのアクセスが発生するとセルフリフレッシュモードが解除されますので、SDRSRMのチェックとhalt/slp命令は必ずSDRAM以外のデバイスで実行してください。

(4) ユーザプログラムでCPUを制御できなくなりますので、0x39FFCB～0x39FFCDはアクセスしないでください。

(5) アドレス設定レジスタ(0x39FFC2)で設定したアドレス範囲を越える領域をアクセスすると、意図しない領域がアクセスされ、元のデータを書き換えてしまうことがあります。設定した領域以外をアクセスしないでください。

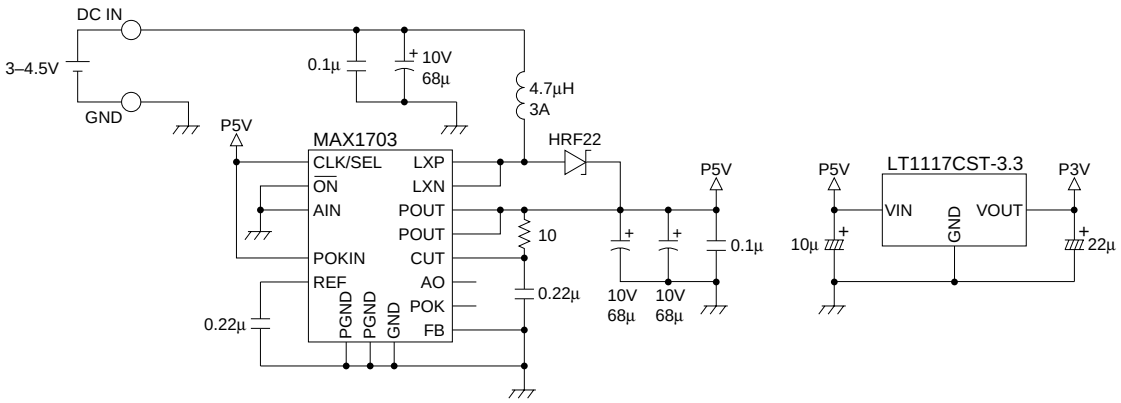
4 S1C33チップ用ボードの基本回路

ここでは、S1C33209を使用する基本的な回路設計について説明します。

4.1 電源

ここではS5U1C33209D1の回路を例に、DC-DCコンバータを使用した電源について説明します。

● DC-DCコンバータ

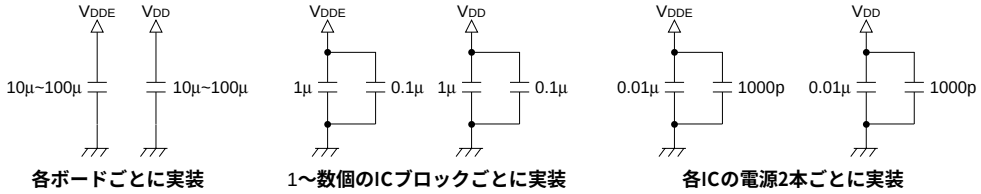


この電源回路は、3～4.5Vの入力電圧をスイッチングレギュレータで昇圧し、外部I/Oやメモリ系電源、アナログ系電源となる5Vを作ります。
さらに、その5Vをリニアレギュレータで降圧し、CPUコア電源用の3.3Vを作ります。
S1C33209のCPUコアは3.3V動作のため、外部インタフェースが5Vの場合はこのように2電源必要になります。

S5U1C33209D1のように、スイッチング電源を使う場合、コンデンサの選定には注意してください。
電池とコイルの間に、デカップリング用の68µFコンデンサを設けてあります。ここは大きなラッシュ電流が流れるため、ESR(等価内部抵抗)が大きいと電力消費が起こり、電池寿命が短くなります。
たとえば、S5U1C33209D1で使っているOSコンデンサと通常の電解コンデンサでは、使用状況により電池寿命に1.5倍以上の開きがありました。
コイルの後のコンデンサは電池寿命にはほとんど影響しません。ただし、ESRが小さいコンデンサの方がノイズ面で有利です。このコンデンサはコイルの後の電圧をできるだけ一定にするためのもので、その変化電圧(リップル)はESRに比例して大きくなります。S5U1C33209D1の場合、これに電解コンデンサを使用すると音声出力にノイズが交じり実用レベルにはなりませんでした。OSコンデンサでは1/10程度の問題ないレベルまで改善されました。
デジタル回路ではコンデンサの違いによりノイズマージンが若干低下する程度ですが、アナログ回路では大きな差となります。
また可能であれば、アナログ回路にはスイッチング系の電源を使わない方が無難です。

● デカップリングコンデンサ

電源とGNDラインのノイズ対策としては、以下の4つの方法が有効です。
1)4層以上の基板を使用し、全面GND、全面VDDの層を設ける
2)基板1枚につき100µFの電解コンデンサ、小さい基板には10µFのタンタルコンデンサを取り付ける
3)CPUとメモリのブロックには、1µF + 0.1µFの積層セラミックコンデンサを取り付ける
4)各ICの電源端子のできるだけ近くに、0.01µF + 1000pFのチップ積層セラミックコンデンサを取り付ける



2)〜4)のコンデンサで、以下の周波数範囲をカバーします。
100µF:　数100kHz以下のAC成分吸収
1µF:　　数100kHzから数MHzのAC成分吸収
0.1µF:　数MHzから20MHz程度のAC成分吸収
0.01µF:　10MHzから50MHz程度のAC成分吸収
1000pF:　数10MHzから100MHz程度のAC成分吸収

いずれかのコンデンサを省略すると、その領域のノイズが吸収しにくくなります。
たとえば、従来は一般的であった各ICに0.1µFのみでは、10MHz前後の吸収性能は高い反面、数10MHz以上は吸収できません。
また、40MHz超で動作するS1C33209の基板は100MHz付近やそれ以上のノイズも含み、これは1000pF程度のコンデンサを使用しないと吸収できません。
さらに、1000pFのコンデンサは、配線が長くなるとインダクタンス成分により吸収範囲の上限が下がりますので、後述するPLLのコンデンサの次の優先度で端子からの最短位置に配置してください。この配慮を怠ると、吸収範囲の上限は100MHzに届きません。

両面基板を使用する場合、少なくともGNDはできるだけ強化し、GND電位が各部で異なることのないようにしてください。電源ラインは各ブロックごとにデカップリングコンデンサで強化し、電圧がふらつかないようにしてください。

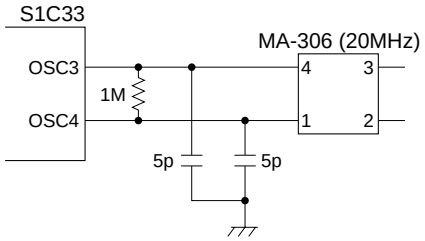
4.2 発振回路

S5U1C33209D1、S5U1C33208E1を例に、発振回路について説明します。

● 20MHz発振子

水晶発振子、抵抗、コンデンサで高速(OSC3)発振回路を構成するS5U1C33209D1の例です。

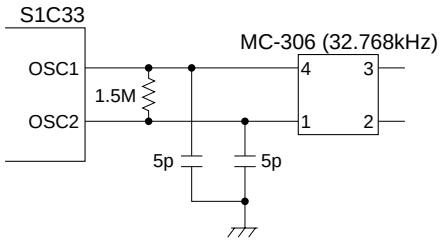
抵抗とコンデンサの値については水晶発振子の資料を参照してください。



● 32kHz発振子

水晶発振子と、抵抗、コンデンサで32kHzの低速(OSC1)発振回路を構成するS5U1C33209D1の例です。

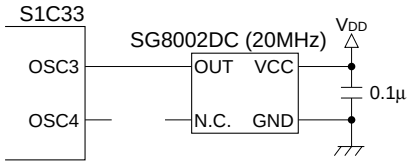
抵抗とコンデンサの値については水晶発振子の資料を参照してください。



● 20MHz発振器

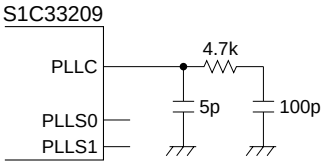
発振器の出力クロックをOSC3端子に供給するS5U1C33208E1の例です。

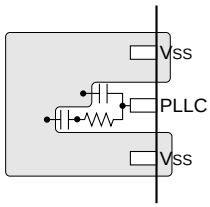
入力するクロックの電圧レベルは、CPUコアの動作電圧(V_{DD})と同じにしてください。(例: 3.3V)
これは、OSC1端子に外部クロックを入力する場合も同様です。



● PLL、コアクロック、バスクロック

関係する端子として、PLLCとPLLS[1:0]があります。





PLLCは、他のすべての配線に優先して最短のパターンとしてください。また、他の信号線からクロストークノイズ等が入らないように、できるだけ大きなGNDのパターンで囲んでください。
このラインのノイズ特性が悪化すると、ジッタが増大したり、クロックのデューティに悪影響がでます。

PLLS0とPLLS1端子の処理により、S1C33209の高速動作クロックを次の3種類から選択できます。
PLLS1 = 0, PLLS0 = 0: OSC3クロックをそのまま使用(PLLを使用しないため、消費電流が少し下がる)
PLLS1 = 1, PLLS0 = 1: OCS3の2倍のクロック(OSC3は10～20MHz入力)
PLLS1 = 0, PLLS0 = 1: OSC3の4倍のクロック

このクロックがチップ内のCPUコアに入ります。
さらに、#X2SPD端子でバスの動作クロックを決めます。
#X2SPD = 1: コアと同じクロックでバスが動く
#X2SPD = 0: コアのクロックを1/2分周してバスが動く

組み合わせ例:

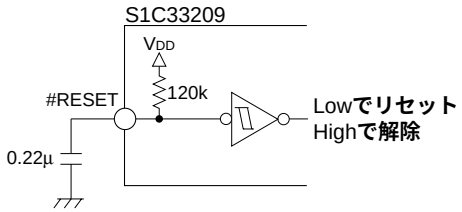
OSC3	PLLS1	PLLS0	#X2SPD	コアクロック	バスクロック
20MHz	0	0	1	20MHz	20MHz
20MHz	1	1	0	40MHz	20MHz
15MHz	0	1	0	60MHz	30MHz

4.3　リセット回路

ここでは、単純なCRのリセット回路、および電源電圧検出機能を持つリセットICを使った回路について説明します。

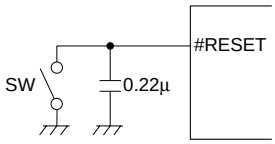
● CRによるリセット

S1C33209のリセット端子は、120kΩ程度のプルアップ抵抗付きのシュミットトリガ入力です。
外部に0.22μF程度のコンデンサを接続することで、簡単なリセット回路が構成できます。 0.22μFのコンデンサは積層セラミックでも電解コンデンサでもかまいません。



これで、電源立ち上がりからVDD/2まで15～20ms程度の時定数となります。
この回路は最も簡単に構成できる反面、リセット入力が 120kΩのプルアップのみで、しかも立ち上がりエッジで動くため、ノイズに弱いという欠点があります。
リセット端子からコンデンサまでは、数cm以内でできるだけ短く配線してください。

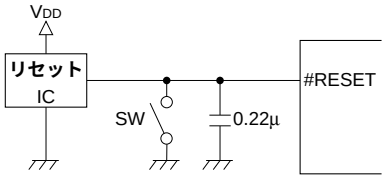
また、S5U1C33000Hでデバッグする場合は、リセットスイッチをつけておくことを推奨します。
S5U1C33000Hとターゲットがもしうまくつながらない場合、リセットスイッチを押しながらS5U1C33000HをONしてリセットスイッチをはなすと確実につながるからです。
開発用にのみ使用する場合は、コンデンサとVSSの間にスイッチをつけるだけで済みます。



● リセットICを使用した回路

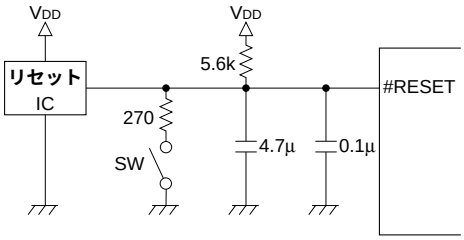
S5U1C33T01D1で使用しているリセットIC（ミツミ, PST572）はVDDとGNDを接続する3端子タイプで、VDDが所定のレベル以下の場合はVOUTをLowで駆動、所定レベル以上になるとハイインピーダンスにします。

上記のリセット回路にこのICを追加すると次のようになります。（S5U1C33T01D1回路より）

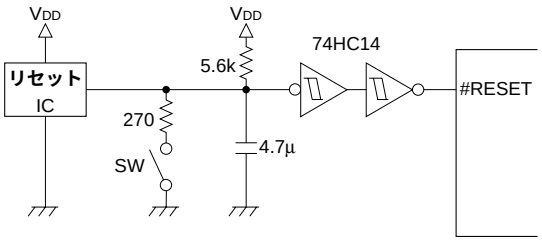


● リセットの対ノイズ強化

上記リセット回路をICから離して線を引き回すと、クロストークノイズ等の影響を受けやすくなります。その場合は次のような対策を施してください。
1)プルアップ抵抗を小さくする。
2)端子の側にデカップリングコンデンサを付加する。
3)クロストークの影響をさけるため、GNDパターンで囲む。
4)ロジックで、High/Lowを低インピーダンスで駆動する。



これは、外部の5.6kΩでプルアップする例です。また、SWには270Ωを直列に接続し、スイッチに流れ込む電流を制限しています。また、リセット端子の側に0.1µFのデカップリングコンデンサを入れて、クロストーク等で乗りやすい高周波ノイズを低減させてます。



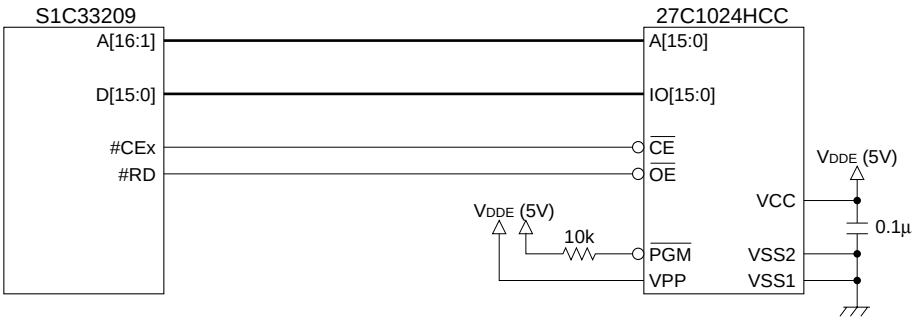
これは、74HC14(シュミットタイプインバータ)を入れて、ロジックで駆動する例です。
これにより、ノイズに強い回路になります。

なお、リセットに限らずNMIや入力割り込み等、エッジで動作するポートはノイズによる誤動作に注意してください。
特にプルアップ/プルダウンでHigh/Lowのレベルを確定する入力については、配線をできるだけ短くし、上記と同様の対策を施してください。
特に100kΩ程度のプルアップ/プルダウンの場合、端子と最短で結線してください。10kΩ以下のプルアップ/プルダウンでも配線はあまり伸ばさず、クロストーク等ノイズが入らないか、オシロスコープで確認することを推奨します。

4.4　ROMの接続

S5U1C33209D1とS5U1C33000Hを例に、ROMの接続図を示します。

● x16 ROMの接続



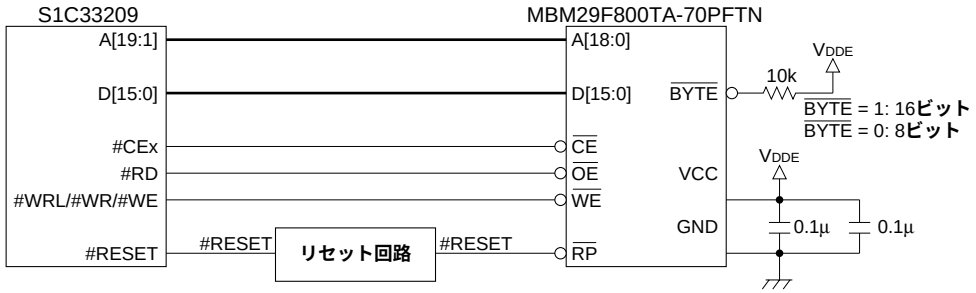
S5U1C33209D1には、PLCC44ピンの1MビットEPROMが搭載されています。
S1C33209 I/Oも、このROMも5Vで動作します。
バスクロックが20MHzの場合、ROMのアクセス時間要件は次のとおりです。
1ウェイト、2サイクルリード(バスサイクル = 100ns): アクセス時間80ns(3.3Vでは75ns)以上のROM
2ウェイト、3サイクルリード(バスサイクル = 150ns): アクセス時間130ns(3.3Vでは125ns)以上のROM

4.5　FLASHメモリの接続

S5U1C33209D1を例に、x16タイプのFLASHメモリの接続図を示します。

● x16 FLASHメモリの接続

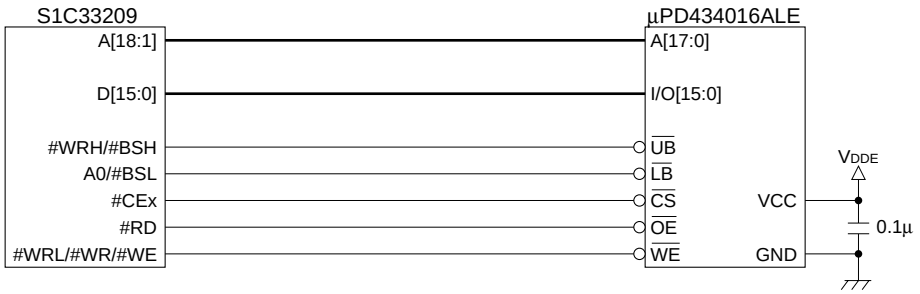
TSOP 48ピンタイプの8MビットFLASHをダイレクトに接続しています。29F800というx8/x16両用タイプを、x16で使用しています。



4.6 SRAMの接続

● x16 SRAMの接続

これは、4Mビット、x16のSRAMを1個接続する例です。

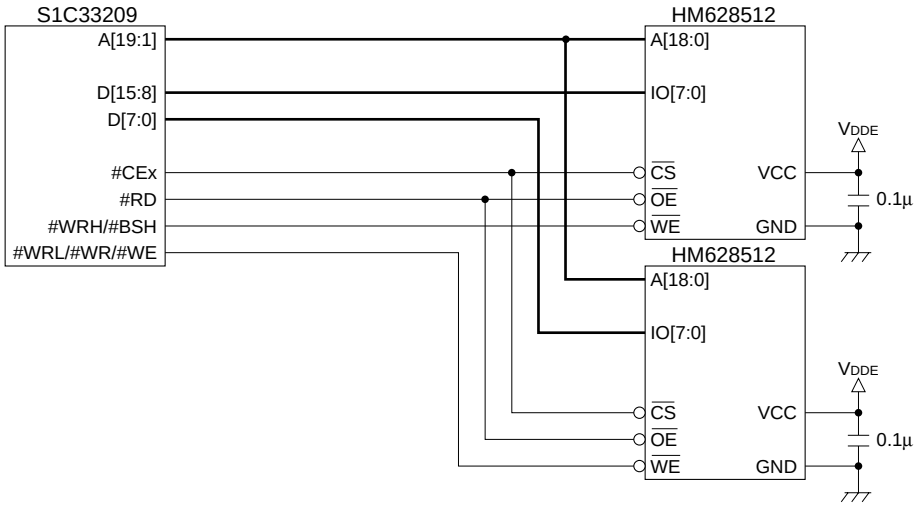


BCUがデフォルト設定のままでは、このタイプのRAMはアクセスできません。
BCUをBSL方式に変更すると、上記結線で動作するようになります。
0x4812EのD3を1にするとBSL方式となり、0では通常のA0方式です。

注: BSL方式は、S1C33209およびS1C33L01ではS5U1C33000Hのデバッグと組み合わせて使えないという制限があります。S5U1C331M2Sを利用してデバッグをお願いします。

● x8 SRAMを2個接続

SRAMが1個では足りなくて2個必要な場合、x8タイプを2個使用すると外部ロジックも不要で簡単に接続できます。以下の図は4Mビット、x8のSRAMを2個接続するS5U1C33209D1の例です。



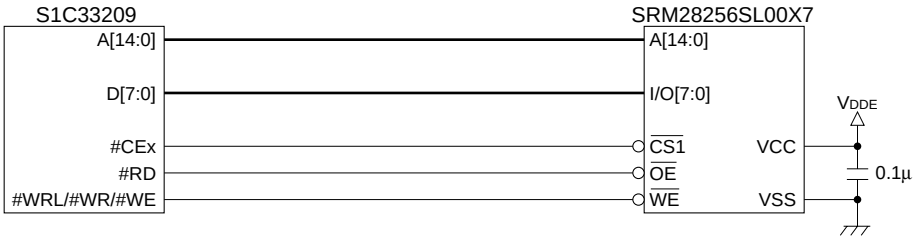
アドレス、#CE、#RD出力は2デバイスの接続により負荷容量が増えますが、ダイレクトに接続可能です。

バスクロックが20MHzの場合、RAMのアクセス時間要件は次のとおりです。
1ウェイト、2サイクル(バスサイクル = 100ns): アクセス時間80ns(3.3Vでは75ns)以上のRAM
2ウェイト、3サイクル(バスサイクル = 150ns): アクセス時間130ns(3.3Vでは125ns)以上のRAM

S5U1C33209D1(20MHz動作)のSRAMは55nsのため、1ウェイトでアクセスできます。

● x8 SRAMを1個接続

これは、256Kビット、x8のSRAMを1個接続する例です。

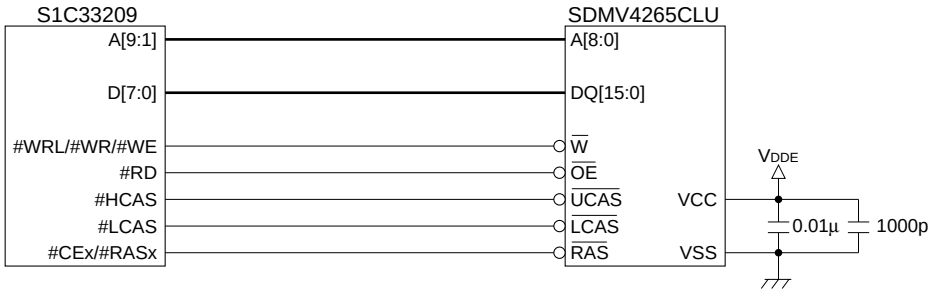


BCUのデフォルト設定では、16ビットサイズになっていますので、8ビットに設定し直してください。
各エリアの設定レジスタのD6もしくはDEビットを1にします。

4.7　DRAMの接続

S5U1C33L01D1を例に、DRAMの接続図を示します。なお、S5U1C33L01D1にDRAMのパターンは作成されていますが、DRAMは未実装です。

● 4Mビット、x16 DRAMの接続

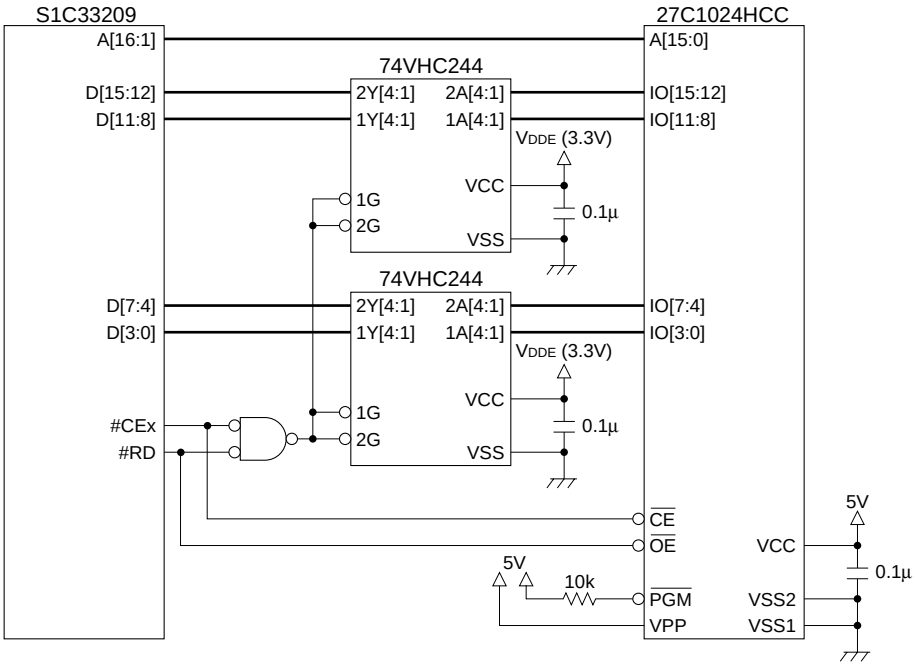


BCUの設定については、"3.1 BCUの設定"を参照してください。

4.8 5V ROMと3.3Vバスの接続

● 3.3Vのバスに5VのROMを接続する方法

S1C33209のバスは、5Vトレラントにはなっていません。また、他の3.3Vメモリデバイスが接続されていると、そちらにも電流が流れ込んでしまいます。このため、3.3V I/OのS1C33チップに5Vデバイスを接続する場合、電位差を吸収するバッファが必要になります。



この例では74VHC244を2個使用し、ROMのリード時に5VのROM出力データを3.3Vに変換します。
バッファはROM選択時のみ駆動されます。この例はS5U1C33000H内部で使われています(ただし、S1C33104使用)。
VHCタイプのICは5V入力トレラントがありますので、電源電圧3.3Vでも5V信号を受けることができます。低電圧系CMOS ICにはトレラント機能をもつものが多くあります。
ROMに入力するアドレス、#RD、#CE信号は3.3Vですが、ROMがTTL(Highは2.0V以上、Lowは0.8V以下)レベルに対応していれば、直接入力可能です。
バッファICを244ではなく16244にすると1個で済みます。
バッファのG端子には、#CEと#RDをANDした信号を接続しています。ROMからのリード時のみYよりデータが出てきます。
バッファを245や16245にしてG端子に#CE、DIR端子に#RDを接続すると双方向になり、上記のANDロジックも不要です。また、MEM33DIP42など、ROMエミュレーションメモリも使えるようになりますので、双方向にしておくと便利です。

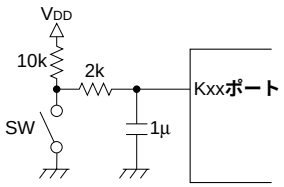
4.9　ポート関係

● 未使用入出力兼用(P)ポートの処理

入出力兼用ポートは、デフォルトで入力ポートとなります。
使用しないポートは、VDDまたはVSSに接続するか、ブート直後、出力に切り換えてください。
なお、VDDまたはVSSに接続したポートは出力にしないよう注意してください。

● 入力(K, P)ポートのチャタリング除去

K60～K67以外のKポートとPポートはシュミット入力になっています。
スイッチの2値入力で単純に数msのチャタリングを除去するには、次のような回路を構成します。



この例では内部プルアップを使用していません。
0→VDDは10ms程度の立ち上がり時間となり、数msのチャタリングが除去できます。VDD→0は、2ms程度の立ち下がり時間となります。
時定数を大きくすると(C×Rを大きくすると)より大きなチャタリングが除去できます。
また、Rを大きくするとスイッチON時の電流を低減できます。ただし、ノイズに弱くなりますので、線の引き回しには注意が必要です。

シュミット入力になっていない端子については、チャタリング除去には74HC14等を利用してください。
なお、シュミット入力かどうかの詳細については、各ICのマニュアルで確認してください。(S1C33209テクニカルマニュアルでは、"Appendix B 端子特性"参照)

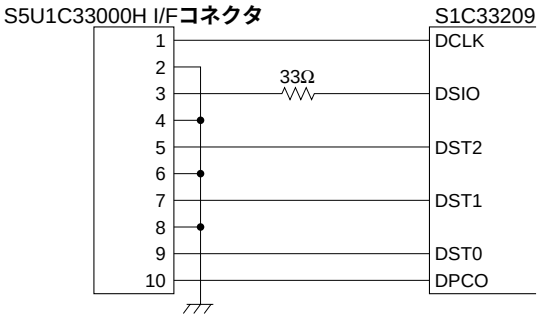
4.10 デバッグ用の接続

ここでは、S5U1C33000Hとの接続とS5U1C331M2S用S5U1C330M1D1の接続についてS5U1C33209D1を例に説明します。

● S5U1C33000Hの接続

S1C33209からは、デバッグ専用端子が6本出ています。
DCLK、DSIO、DST2、DST1、DST0、DPCO

DSIOに33Ωの抵抗を直列に入れ、GNDを4本加えた計10本でS5U1C33000Hに接続します。



基板上に上記のパターンを配置できない場合は、33Ωの抵抗を入れない空中配線で接続しても動作します。

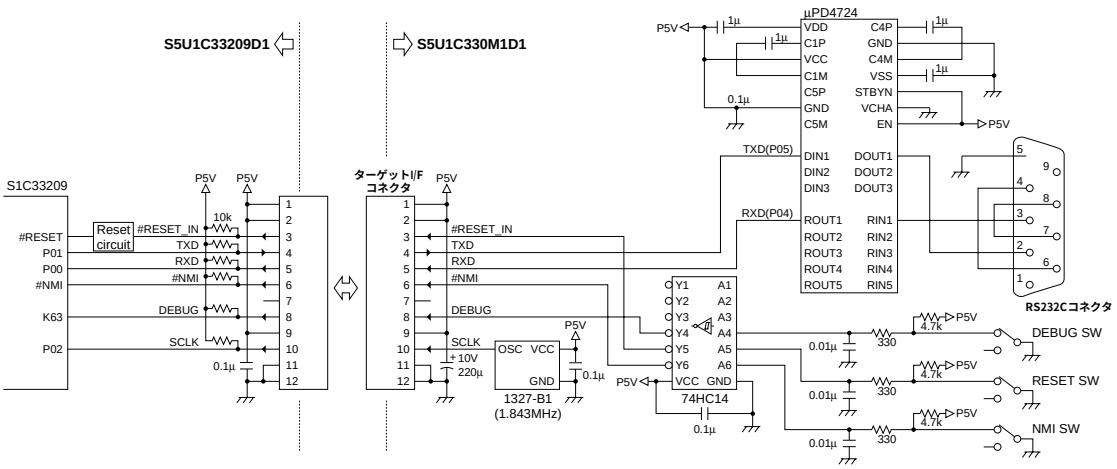
また、PCトレース機能をS5U1C33000Hでディセーブルにして(右端のDIP SWを下にセット)、DCLK、DSIO、DST2とGND 1本の計4本を接続するだけでも、PCトレース以外のICDモードのデバッグ機能はすべて使用できます。

なお、配線長は5cm程度以下としてください。
また、DSIOの33Ωは33チップにできるだけ近く配置してください。DSIOは唯一の入力ピンで、内部の120kΩでプルアップされ、Lowパルスが入るとデバッグモードになります。誤動作を避けるため、非デバッグ時は33Ωを未実装としてDSIOのパターン長を最小とするか、もしくは3.3V(コアのV_{DD}電圧)にプルアップしてノイズが入らないようにしてください。

● S5U1C330M1D1との接続

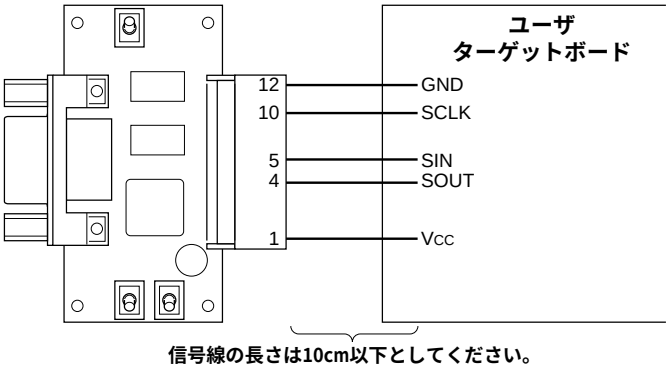
S5U1C331M2Sは、以下のリソースを使用します。
ROM 10Kバイト、RAM 4Kバイト、シリアルインタフェース1ch

次の図は、S5U1C330M1D1の回路図とターゲット上のインタフェース部です。



S5U1C33209D1は、S5U1C330M1D1の全機能が使用できるように結線されています。
この中で、RESET入力、NMI入力、入力ポート接続用デバッグスイッチの3本は利便的なもので、必須ではありません。

次のように必須ピンのみを接続することも可能です。



必須ピンはSCLK、SIN、SOUT、GND、VDDの5本です。
VDDは、S5U1C330M1D1は5V、S5U1C330M1D2は3.3Vです。

5 ファインPWMによるスピーカ出力と外部アナログ回路

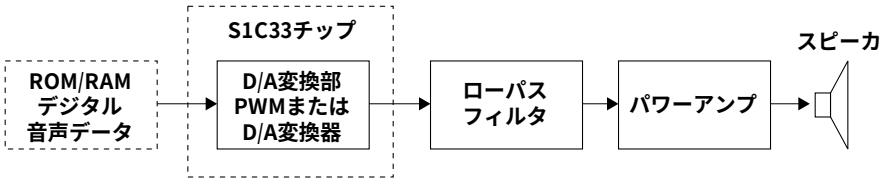
5.1 マイコンによる一般的な音声出力回路

マイコンにより音声(音楽)をスピーカに出力するには、大きく分けて3つのブロックが必要です。

1)D/A変換部
デジタルの音データをアナログに変換する部分

2)ローパスフィルタ部
D/A変換部で生成したアナログ音から量子化ノイズを取り除き、連続的なアナログ波形にスムージング処理する部分

3)パワーアンプとスピーカ部
ローパスフィルタ部で処理されたアナログ波形を増幅し、スピーカを駆動する部分



ここでは、単一電源の比較的ローコストな音声出力システムを構築する一般的な方法や各ブロックの構成、サンプリング周波数、出力精度と音質について説明します。

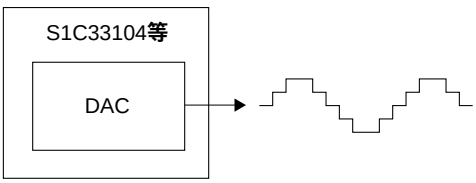
5.1.1 D/A変換部

デジタルの音データをアナログに変換する方法としては、次の3つが一般的です。
1)DACによる方法
2)ラダー抵抗による方法
3)PWMによる方法

以下、それぞれの方法を説明します。

● DACによる方法

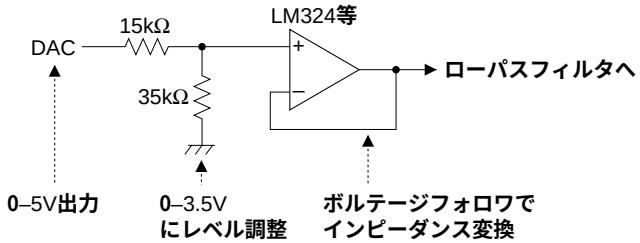
マイコンに内蔵されたDACより出力する方法です。



マイコン内蔵型のDACは、8ビットから10ビット精度のR-2Rラダー抵抗方式のものが一般的です。
それ以上の精度が必要な場合は、外部に専用のDACを用意します。音声出力には12ビットR-2R方式のDAC、オーディオ用には14～20ビットデルタシグマ方式のDACがよく使われます。

DACを使用する場合、出力インピーダンスに注意が必要です。
R-2Rの後段のOPアンプにより低インピーダンスで出力できる場合(5～10mAの出力が可能な場合)、次のローパスフィルタ部で直接受けることができます。

高インピーダンスで出力される場合は、ボルテージフォロワによってインピーダンスを下げる必要があります。



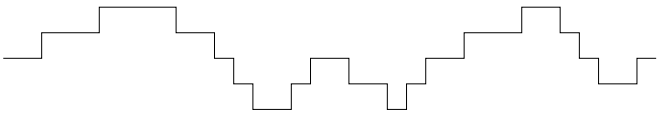
使用するOPアンプにより入力電圧が制限されます。この例ではCMOS型の安価なOPアンプ(LM324等)を使用するために、入力電圧を抵抗分割によって0～3.5Vの範囲内に調整しています。この場合、15kΩ + 35kΩの抵抗を通してGNDに電流が流れますので、それがDACの出力電流の規定を超えないように注意してください。

電源電圧に対するOPアンプの入力電圧範囲の目安は、次のとおりです。
1)　通常のバイポーラ型、FET型(+と-の2電源で動作するRC4558など)の場合
　+電源電圧 - 1～1.5V　から　-電源電圧 + 1～1.5V程度

2) CMOS型(単一電源のLM324など)の場合
　+電源電圧 - 1～1.5V　から　GND + 数mV～数10mV(ほぼGND)

これは、入力だけではなく、出力電圧にも適用されます。
電源に対してフルスイングできるレイルツーレイル型もあります。出力レイルツーレイルは比較的安価ですが、入出力のレイルツーレイルは高価なため、低コストなシステムでは実用的ではありません。

DACからの出力は、次のような量子化ノイズの乗ったアナログ波形となります。

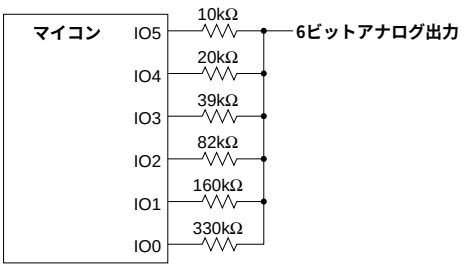


たとえば、8kHzのサンプリング周波数で出力する場合、1/8000秒ごとにデジタルデータをソフトウェア等でDACに書き込みます。次のデータ書き込みまで出力は変わらないため、出力は不連続な階段状の波形となります。これが量子化ノイズで、サンプリングと同じ周波数を中心に、音質を劣化させます。これをカットするために次段のローパスフィルタが必要で、また、ローパスフィルタの性能が音質を大きく左右します。

S1C33104チップでは、内蔵の8ビットDACを音声出力に使用することができます。ただし、8ビットでは音質的に十分とは言えません。
S1C33209では16ビットタイマを使用し、後述のファインPWM方式で10ビットから最高15ビットの精度を実現しています。

● ラダー抵抗による方法

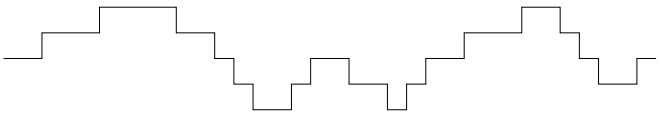
これは、マイコンのI/Oポートに外部抵抗を接続し、簡易的にDACを構成する手法です。
特にDACを持たないマイコン用ですが、ほとんどのマイコンで使用可能です。



抵抗値はE24系列から選択　　10kΩ、20kΩは1%誤差、他は5%精度

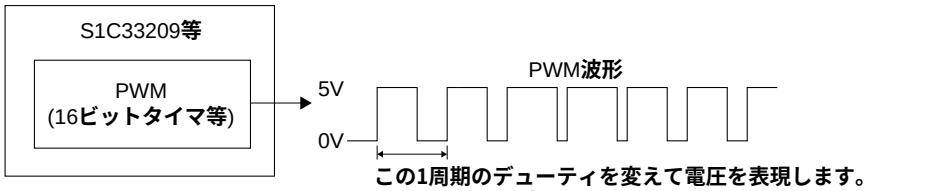
使用する抵抗は、上位ビットほど高い精度が必要です。1%精度レベルではこの例のように6ビット、0.5%精度でも7ビット程度が限界です。
ただし、相対的な精度が重要なため、R-2R抵抗(R-2R方式で抵抗を1つの部品に集約し、抵抗精度をあげた抵抗アレイ、米国BIテクノロジ社製品など)を使用して8ビット程度の構成も可能です。
12ビット以上のR-2R方式のD/Aでは、多くの場合内部でトリミングをして精度を出しています。
なお、この方式は出力インピーダンスが高いため、ボルテージフォロワを使用して低インピーダンスに変換後、ローパスフィルタ部に入れる必要があります。

波形自体は上記DACと同様、量子化ノイズを含んだ階段状の波形となります。



● PWMによる方法

これは、アナログ電圧を出力するのではなく、デジタル波形のデューティ比(1と0のパルス幅の比率)を変化させて電圧を表現する方式です。PWM出力はDAC出力波形とはまったく異なりますが、出力をローパスフィルタで平滑化すると、DAC等と同様の量子化ノイズを含んだ階段状のアナログ波形を得ることができます。また、音声部分のスペクトルがDAC出力と同様なため、人間の耳にはどちらも同じ音として聞こえます。



この場合、PWMの周期(搬送波の周期)は、D/A変換の周期(再生したい帯域)より大きくします。たとえば、音声再生では搬送波を80kHz以上にします。そのうえで、20kHz以上をカットするローパスフィルタを通すと、上記DACなどと同じ階段状のアナログ波形を得ることができます。

PWM波形には搬送波の周波数、たとえば80kHzを中心に大きなノイズスペクトルがあります。
この周波数域は可聴周波数を超えており、このPWM波形を直接スピーカ出力したとしても、聞こえる音自体には影響しません。スピーカへの影響を考慮すると、ローパスフィルタで連続的なアナログ波形にしておくほうが無難ですが、次段の量子化ノイズをカットするためのローパスフィルタが兼用できますので、このD/A変換部では不要です。
なお、80kHzにすべてのノイズが集中しているわけではなく、実際には可聴帯域にもPWMノイズが若干混じります。搬送波を160～320kHzにするとそれらのノイズを低減できますが、10ビットのD/A変換では、80kHzで問題ありません。
また、出力インピーダンスもPWM用I/Oパッドで決まるため低インピーダンスとなり、ボルテージフォロワ等によるインピーダンス変換は不要です。

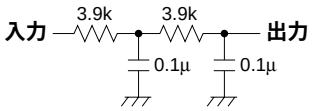
PWM方式の精度はパルス幅の細かさで決まります。
8ビット精度を実現するためには1周期が256×80kHz となり、20MHzの基準クロックが必要です。
S1C33209用の音声出力ライブラリではPWMで10ビット精度を実現し、10ビットDACと同等な音質を確保しています。通常、10ビット精度には1024×80kHz = 80MHzのクロックが必要ですが、S1C33209ではファインPWM技術により40MHzクロックで実現しています。

PWM方式は、以前より微分精度の高いD/A変換方式として知られていました。ただし、高周波数の基本クロックが必要なため、音声帯域ではあまり実用的ではありませんでした。逆に、PWMをPDM(パルス密度モジュレーション)にし、S/N比を改善するために時間軸方向にデジタル信号処理を加えたものが音声でよく用いられるデルタシグマ型DACです。
ファインPWMは、パルス幅を半クロック単位で制御するセイコーエプソン独自の技術です。40MHz以上で動作可能なS1C33チップとこの技術により、音声出力にはあまり実用的ではなかったPWMもすでにDACを超えるものに進化したといえます。

5.1.2 ローパスフィルタ部

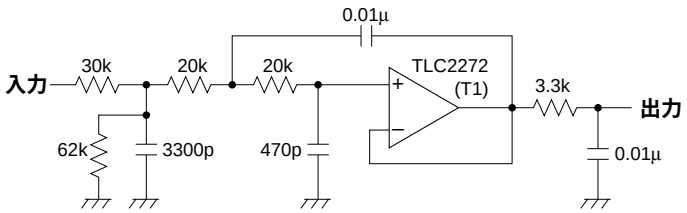
D/A変換で生じる量子化ノイズは音質を劣化させます。聴感上は音のざらつきや高音部が強調されるといった形で現れます。これを低減するには、ローパスフィルタ部をしっかり設計し量子化ノイズを除去することが重要です。
ただし、コストにも配慮する必要があります。
ローコストなシステムでは、2次から4次程度のローパスフィルタを推奨します。
カットオフ周波数は、サンプリング周波数の1/2.5(4次フィルタ)、1/3(3次フィルタ)、1/3.5～1/4(2次フィルタ)程度とし、それ以上を減衰するように設計します。カットオフ周波数をこれより上げていくと、サンプリング周波数を中心に大き目のパワーをもつ量子化ノイズが目立ち始め、音質が劣化します。
1次のローパスフィルタは減衰率が低いため、使用しないほうが無難です。
なお、使用条件によっては(騒音下でもよく聞こえるように擬似的に高音を強調したい場合など)、わざと量子化ノイズを出すケースもあります。

● 2次フィルタ例



CRのローパスフィルタを重ねて2次フィルタにします。この例は8kHz出力に対応し、カットオフは2kHz弱です。
抵抗2個とコンデンサが2個で、安価なフィルタが構成できます。
ただし、カットオフ周波数付近の減衰はなだらかで、音質は多少劣化し、キンキンした音が聞こえます。

● 4次フィルタ例



これは、OPアンプ1個で3次アクティブフィルタを構成し、その後にCRの1次ローパスフィルタを加えて4次フィルタとしています。減衰特性も良好で、ローコストなシステムでは十分な音質が得られます。
この例は8kHz出力に対応し、カットオフは3kHz程度です。
また、入力の30kΩと62kΩにより入力電圧の範囲を0.67倍にしぼって、OPアンプ入力が飽和しないようにしています。

● オーバーサンプリング

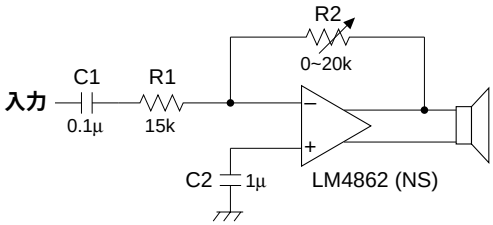
セイコーエプソンの音声、音楽ミドルウェア(S5U1C330V1S、S5U1C330T1S、S5U1C330S1S等)では、音出力に2倍オーバーサンプリング技術を用いて、ソフト的にも量子化ノイズを大きく低減させ、ローパスフィルタ部の負荷を減らしています。その効果により、4次フィルタ以上にしても聴感上の差異はほとんどありません。
オーバーサンプリングしないと、上記の4次フィルタでも不充分です。
通常、5次以上のチェビシュフフィルタがよく使われますが、複雑で高コストになりがちです。

5.4項の"音声出力アナログ回路例"も参照してください。

5.1.3 パワーアンプとスピーカ部

ここでは2種類の例について説明します。
専用の差動型パワーアンプを使い大きな音量を出す例と、トランジスタを使い中音量をローコストで出す例です。

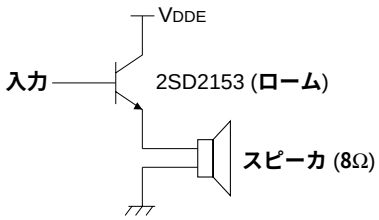
● パワーアンプ例



まず、入力部のコンデンサC1で1次ハイパスフィルタを構成し、直流成分をカットします。C1とR1で決まるカットオフ周波数は、通常50Hz程度とします。これを低くしていくと、音声スタート時のポップ音がめだつ場合があります。
また、この部分の入力インピーダンスは、ローパスフィルタ部の出力インピーダンスより数倍以上高くなっている必要があります。両者のインピーダンスが同じかこの逆になっていると、ローパスフィルタとハイパスフィルタの相互干渉により特性が変わってしまいます。

次の差動アンプでスピーカを駆動します。音量はR2/R1で決まります。

● トランジスタによるスピーカの駆動



ここでは、エミッタフォロアでスピーカを駆動しています。
この場合のトランジスタには、hfeの大きいもの(500以上、ダーリントンは電圧範囲が狭いので不可)を使用してください。また、電流増幅のため、D/A部とローパスフィルタ部のインピーダンスもこれにあわせておく必要があります。

5.4項の"音声出力アナログ回路例"も参照してください。

5.2　サンプリング周波数、ビット数精度と音質について

● サンプリング周波数

サンプリング周波数が高くなるほど、より高音がでるようになり、自然な音質になります。
最低でも、2kHz以上のサンプリング周波数が必要です。これより低いと音声が不明慮になり、人間の声の場合は話している内容が聞き取れなくなります。
サンプリング周波数を4kHz、8kHz、16kHzと上げると、それに比例して音質も上がります。この中でも電話に8kHzが採用されていることもあり、多くの機器で8kHzが使用されます。
16kHzの上は22kHz、32kHzで、それぞれ10kHz、15kHz付近まで再生できるようになります。ただし、人間の聴力が落ちてくるのとともに、ローコストシステムでは再生能力の問題もあり、音質の向上は頭打ちになります。
これ以上では、44.1kHzのCDや48kHzのDATクラスとなります。

サンプリング周波数を上げるとデータ量もそれに比例して増えてしまいます。
ローコストシステムの一般的な目安としては、以下のサンプリング周波数を推奨します。

人間の音声: 8kHz（データ量重視の場合）
　　16kHz（音質重視の場合）
音楽:　　22.05kHz
　　32kHz（ハイエンドの音質）

● ビット数精度

D/A変換部のビット数により、S/N比が大きく変わります。大まかな値としては、1ビット増えるとS/N比が6dB上がります。ビット数と音質の目安を以下に示します。

(1) 8kHzサンプリング
　1〜3ビット:　音声がノイズに隠れて、話している内容が聞き取りにくい（ただ人間の声として認識できる）
　4〜5ビット:　内容を聞き取れるようにはなるが、ノイズは依然大きい
　6〜7ビット:　音声は明瞭になるが、ノイズは耳につく
　8〜9ビット:　若干うるさい環境ではノイズが気にならない実用レベル
　10ビット以上: 静かな環境で聞いてもノイズは分からない
　最低でも8ビット、できれば10ビットを検討してください。

(2) 22kHz以上のサンプリング
　サンプリング周波数を上げると、量子化ノイズが耳につきやすくなります。
　8〜9ビット:　若干うるさい部屋でもノイズが耳につく
　10〜11ビット: 通常の状態ではノイズは分からない
　12ビット以上: 静かな環境で聞いてもノイズは分からない
　最低でも10ビット、できれば12ビット程度を検討してください。

(3) 16kHzのサンプリング
　ほぼ8kHzと22kHz以上の中間の音質です。
　最低でも9ビット、できれば11ビット程度を検討してください。

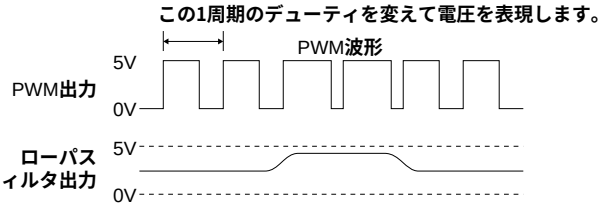
S1C33104単体では、内蔵の8ビットDACにより8ビット出力のみに対応します。S1C33209では、PWMによって8〜32kHz、10〜15ビット出力という余裕を持ったスペックを実現できます。

5.3　PWMによる10ビットD/A変換

S1C33209ではファインPWM方式により10ビットから最大15ビット精度の音声出力を実現しています。
以下、まずファインPWMによる10ビット出力について説明します。15ビット出力については、後述します("5.6 PWMによる15ビットD/A変換")。

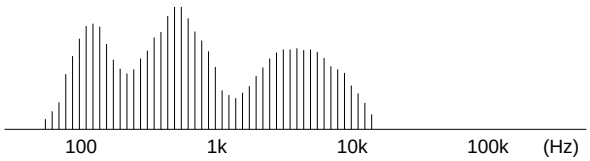
● PWMとDACの違い

前述のとおり、PWMはデューティ比により電圧を表現する方式で、見かけ上の波形はまったく違います。ところが、PWM成分をローパスフィルタでカットすると、DAC出力と同様の波形となります。

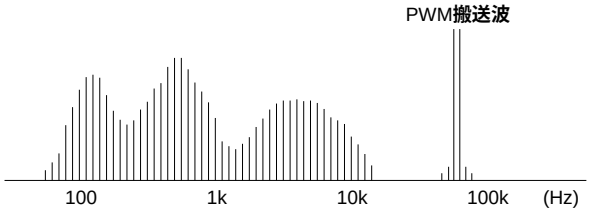


実際、人間の耳は波形を聞いている訳ではなく、周波数スペクトラムを音としてとらえています。

DAC出力のスペクトラム



PWM出力のスペクトラム

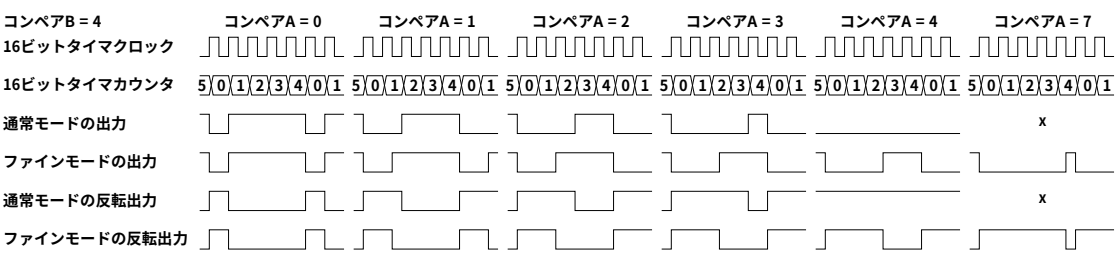


PWM出力では、搬送波の部分に大きなパワーがありますが、可聴帯域にもDAC出力と同じスペクトラムを持ちます。そのため、両者の出力波形はまったく異なりますが、人間の耳にはPWM出力もDAC出力も同じ音に聞こえます。また、量子化ノイズを除去するローパスフィルタ部でPWM搬送波のノイズは消えますので、最終的なスペクトラムは一致します。

● ファインPWMモードについて

PWMの出力精度は、出力波形のデューティ比をいかに細かく制御できるかにかかっています。
周期を80kHz一定として8ビット精度をもたせるには、80kHz×256クロック＝20MHzのクロックが必要で、0.05μs単位にパルス幅を制御する必要があります。
S1C33209用の音声出力ミドルウェア等で提供しているPWMは10ビット精度のため、1クロック幅の制御方式では80kHz×1024＝80MHzのクロックが必要になります。S1C33209では40MHzクロックでPWM用16ビットタイマを動作させ、さらに半クロック単位で出力パルス幅を制御することによって80MHz相当のPWM出力を実現しています。

ファインモードのPWM出力



● ファインモードを使用するPWMプログラミング

ここでは、gnu33\sample\drv33208\pwmを例に、ファインモードを使用するPWM出力方法を説明します。

ファインPWM制御　drv_pwm.cより

```
void init_16timer1(unsigned short compareA, unsigned short compareB)
{
      /* Save PSR and disable all interrupt */
      save_psr();

      /* Set 16bit timer1 prescaler */
      *(volatile unsigned char *)PRESC_P16TS1_ADDR                          (1)
              = PRESC_PTONL_ON | PRESC_CLKDIVL_SEL0;
              // Set 16bit timer1 prescaler (CLK/1)

      /* Set 16bit timer1 TM1 port enable */
      *(volatile unsigned char *)IO_CFP2_ADDR |= IO_CFP23_TM1;              (2)

      /* Set 16bit timer1 comparison match A data */
      *(volatile unsigned short *)T16P_CR1A_ADDR = compareA;                (3)

      /* Set 16bit timer1 comparison match B data */
      *(volatile unsigned short *)T16P_CR1B_ADDR = compareB;                (3)

      /* Set 16bit timer1 mode, fine mode, comparison buffer enable, output normal */
      *(volatile unsigned char *)T16P_PRUN1_ADDR = T16P_SELFM_FM | T16P_SELCRB_ENA
              | T16P_OUTINV_NOR | T16P_CKSL_INT | T16P_PTM_ON | T16P_PSET_OFF
              | T16P_PRUN_RUN;                                              (4)

      /* Restore PSR */
      restore_psr();
}

void  set_16timer1(unsigned short compareA)
{
      /* Set 16bit timer1 comparison match A data */
      *(volatile unsigned short *)T16P_CR1A_ADDR = compareA;
}
```

PWMタイマの初期化(16ビットタイマch1)

(1)プリスケーラの設定

プリスケーラではクロックを分周せずに16ビットタイマ1に供給します。

```
/* Set 16bit timer1 prescaler */
*(volatile unsigned char *)PRESC_P16TS1_ADDR
       = PRESC_PTONL_ON | PRESC_CLKDIVL_SEL0;
       // Set 16bit timer1 prescaler (CLK/1)
```

(2)ポート機能の切り換え

入出力ポートと兼用の端子をPWM出力に切り換えます。

```
/* Set 16bit timer1 TM1 port enable */
*(volatile unsigned char *)IO_CFP2_ADDR |= IO_CFP23_TM1;
```

(3)コンペアデータのセット

16ビットタイマ1のコンペアAデータ(パルス立ち上がりタイミング)をセットします。

```
/* Set 16bit timer1 comparison match A data */
*(volatile unsigned short *)T16P_CR1A_ADDR = compareA;
```

16ビットタイマ1のコンペアBデータ(周期)をセットします。

```
/* Set 16bit timer1 comparison match B data */
*(volatile unsigned short *)T16P_CR1B_ADDR = compareB;
```

(4)16ビットタイマ1のモード設定とスタート

タイマの動作モードを設定し、PWM出力を開始させます。

```
/* Set 16bit timer1 mode, fine mode, comparison buffer enable, output normal */
*(volatile unsigned char *)T16P_PRUN1_ADDR = T16P_SELFM_FM | T16P_SELCRB_ENA
| T16P_OUTINV_NOR | T16P_CKSL_INT | T16P_PTM_ON | T16P_PSET_OFF | T16P_PRUN_RUN;
```

ここで設定している内容は以下のとおりです。
・ ファインモードを選択(ファインPWM出力を行うため)
・ コンペアデータバッファをイネーブル(非同期にデューティ変更データをセットするため)
・ 非反転出力を選択(各周期は0から始まる)
・ 内部クロックを選択(プリスケーラ出力クロック)
・ タイマ出力をON(PWM波形を出力)

タイマをスタートすると出力波形は0から始まり、カウンタがコンペアAに一致したところで1に立ち上がり、コンペアBに一致すると0に立ち下がります。ここまでがコンペアBの設定値で決まる1周期です。この時点でコンペアAレジスタが変更されていなければ、次の周期も同じ波形を出力します。

デューティ比の変更

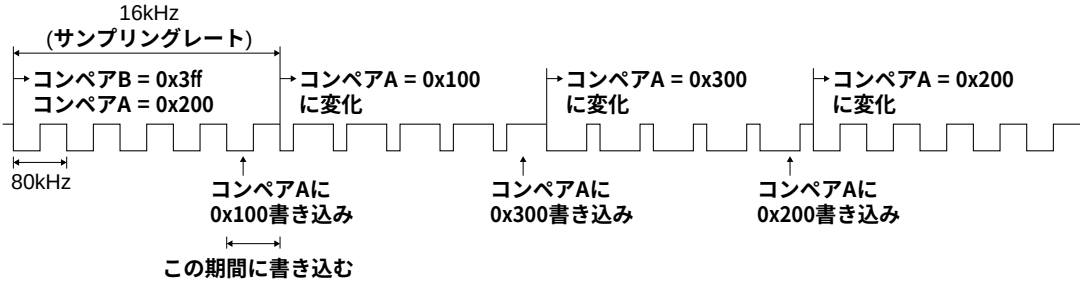
(4)でコンペアデータバッファをイネーブルにしてありますので、カウント動作とは非同期にコンペアAデータを書き込むことができます。

```
void    set_16timer1(unsigned short compareA)
{
        /* Set 16bit timer1 comparison match A data */
        *(volatile unsigned short *)T16P_CR1A_ADDR = compareA;
}
```

この関数でコンペアAを書き換えると、その次の周期から新しいデューティに変わります。書き込み時点の出力波形には影響を与えないため、波形をスムーズに変更できます。

コンペアデータ

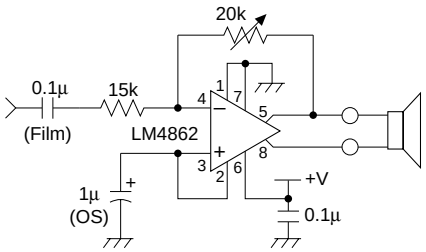
音声出力では、コンペアBを80kHz以上となるように設定し、各サンプリング周期(8～32kHz)ごとにD/A変換するデータをそのままコンペアAデータバッファに非同期に書き込みます。



5.4　音声出力アナログ回路例

● パワーアンプ

下図はS5U1C330A1D1に実装されているパワーアンプ回路です。



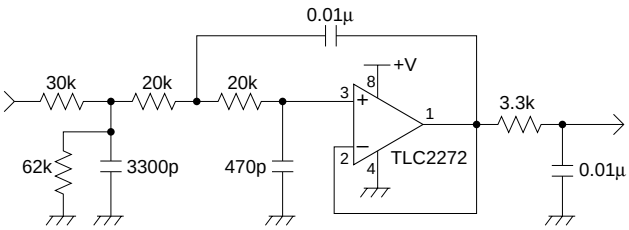
信号を受けるコンデンサとしては、セラミックコンデンサよりフィルムコンデンサが適しています。安価なポリエチレンフィルムコンデンサで問題ありません。セラミックコンデンサには、わずかにヒステリシスがありますので、直接信号が通過する部分に使用すると信号が歪むことがあります。
GNDから分離し、ACカップリングを行う1µFにはOSコンデンサが最適です。電解コンデンサも使用可能ですが、音質にわずかですが影響が出る場合があります。
抵抗は炭素皮膜の5%精度でかまいません。

スピーカはオーディオ用では4Ωや8Ωが一般的です。通常の携帯機器などには8Ωのものがよく使われます。さらに小さいものでは、24Ωなど、8Ω以上となります。

● OPアンプによるローパスフィルタ

OPアンプ1個で構成する8、16、22.05kHzサンプリング用のローパスフィルタ回路の例を以下に示します。すべて、S5U1C330A1D1やS5U1C330A2D1に実装されている音質重視の4次のローパスフィルタ回路です。

8kHzサンプリング用4次ローパスフィルタ（S5U1C330A1D1）



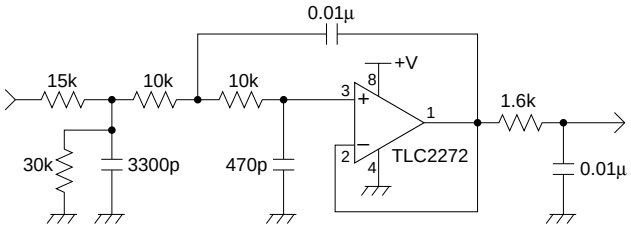
8kHzサンプリングの量子化ノイズを除去するには、カットオフ周波数を3.5kHzから2.7kHz程度にします。このローパスフィルタでは3.0kHzに設定されています。
カットオフ周波数を上げると、3.5kHzあたりから量子化ノイズが聞こえ始めます(2倍オーバーサンプリング使用時)。
最初の分割抵抗は、5V入力を3V強まで下げて、OPアンプの入力規定(0～3.5V程度)に適合させます。
OPアンプは3次フィルタ、その後のCRは1次フィルタで、合計4次のローパスフィルタを構成しています。

抵抗には、5%精度以下の炭素皮膜抵抗が使用可能です。金属皮膜抵抗が最適ですが、微少信号ではないため結果はほとんど変わりません。

コンデンサには注意が必要です。積層セラミックコンデンサを使用する場合、できるだけ動作温度範囲内で±10%以下の精度を保証するB特性のものを使用してください(最低でも±20%以下)。+80% -40%のZ精度のコンデンサは絶対に使用しないでください。特に0.01µFの安価なものはZ精度が多いため注意してください。精度が下がると、ローパスフィルタの特性が変わってしまいます。
アナログ回路にはフィルムコンデンサの方が適していますが、ローコストな音声出力では絶対に必要とはいえません。

OPアンプには入力電圧範囲が0～3.5V程度の単電源CMOSタイプを使用します。安価なOPアンプで問題ありません。
これは、PWM出力だけでなくDAC出力の場合も同様です。

16kHzサンプリング用4次ローパスフィルタ（S5U1C330A2D1）

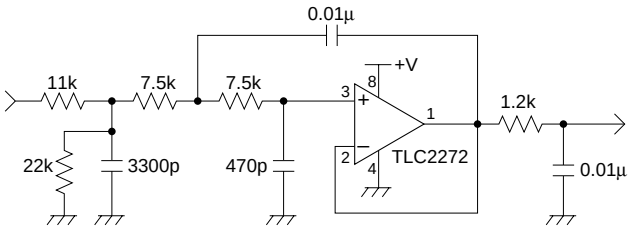


構成は8kHzサンプリングの回路と同じで、カットオフ周波数は6.1kHzに設定されています。
コンデンサの値を変えずに抵抗値をすべて1/2にすると、特性カーブは同じでカットオフ周波数が2倍になります。
抵抗値を変えずにコンデンサをすべて1/2にしても同様です。しかし、E6系列が主のコンデンサは細かく容量を選択できませんので、E24系列の抵抗の方を変えています。

E6系列: 10、15、22、33、47、68の6種類（1.5倍おき）
E24系列: 10、11、12、13、15、16、18、20、22、24、27、30、33、36、39、43、47、51、56、62、68、75、82、91の24種類（1.1倍おき）

さらに細かい値の部品もありますが、入手が容易な上記の値で設計しておいた方が無難です。

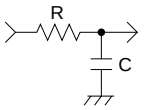
22.05kHzサンプリング用4次ローパスフィルタ（S5U1C330A2D1）



この回路も上記と同様で、8kHzサンプリング用の抵抗値を8/22の値に置き換えて実現しています。カットオフ周波数は8.3kHzとなります。

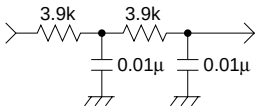
● CRによるローパスフィルタ

CRの1次ローパスフィルタは下図のように構成され、カットオフ周波数=1/(2π×R×C)で計算できます。



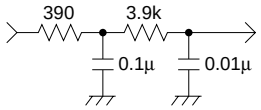
ただし、減衰率は6dB/oct で、周波数が2倍になると波形は1/2になります。このため、量子化ノイズはそれほど除去できません。そこで、これを2段重ねて使用します。コスト重視のシステムには有効なローパスフィルタとなります。
なお、CRローパスフィルタに使用する抵抗とコンデンサにも、OPアンプによる4次フィルタと同じ注意が必要です。特にコンデンサはZ精度を使わないでください。

8kHzサンプリング用CR2次ローパスフィルタ



この構成で、2kHzをカットオフ周波数とするローパスフィルタになります。ただし、前後のCRが同じインピーダンスのため、干渉によってカットオフ周波数付近の下がり方がなだらかになります。

次の図は、この部分を改善した例です。(S5U1C330A1D1で使用)



前段に対し後段のインピーダンスが10倍違うため、カットオフ周波数付近の減衰特性がシャープになります。
ただし前段の抵抗が小さいため、S1C33209チップから2mA程度の電流が流れます(5V動作時)。これが3.9kΩの場合は約0.2mAです。
また、PWMの出力特性がややおわん型になりますが、この形は前段の抵抗値で決まります。390Ωでは約40mV、3.9kΩでは約4mVおわん型になります。これはひずみ率に若干影響します。

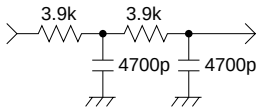
S1C33104のDACに接続する場合、DACの出力部に約250Ωの内部抵抗が直列に入っていますので、初段の390Ωを150Ωに変更します。

後段のインピーダンスは、パワーアンプのハイパスフィルタより低くしておく必要があります。インピーダンスの干渉を防止するためには、少なくとも1/4以下、できれば1/10以下に設定します。
3.9kΩの抵抗値は、パワーアンプの入力インピーダンスが15kΩ以上あることを想定した値です。
パワーアンプの1/4以上のインピーダンスではそれぞれの特性に影響を与えますので、パワーアンプの設計も含め総合的に判断してください。

上記2つの回路では前者(3.9kΩ+0.01µFを2段重ね)の例を推奨します。この例で高域をもう少し強調したい場合は、0.01µFを6800pFに替えてみてください。ただし、量子化ノイズも大きくなります。

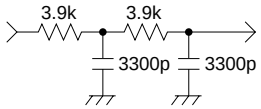
なお、CRを2段重ねで使う場合、絶対に後段のインピーダンスを前段より低くしないでください。特性が悪化し実用にはなりません。

16kHzサンプリング用CRローパスフィルタ



8kHzサンプリング用のコンデンサ0.01µFを約1/2の4700pFに変えています。カットオフ周波数は約4kHzです。さらに高音を出したい場合、4700pFを3300pFに替えてください。ただし、量子化ノイズも大きくなります。

22.05kHzサンプリング用CRローパスフィルタ

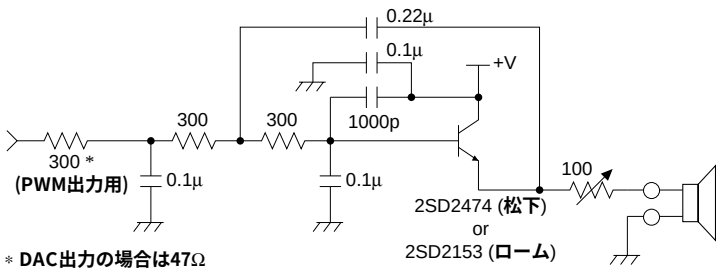


8kHzサンプリング用のコンデンサ0.01µFを約1/3の3300pFに変えています。カットオフ周波数は約6kHzです。さらに高音を出したい場合、3300pFを2200pFに替えてください。ただし、量子化ノイズも大きくなります。

● トランジスタによるスピーカ駆動

トランジスタを使用する場合、ローパスフィルタとパワーアンプ部をまとめて設計します。
ここでは、3次のローパスフィルタを採用しています。

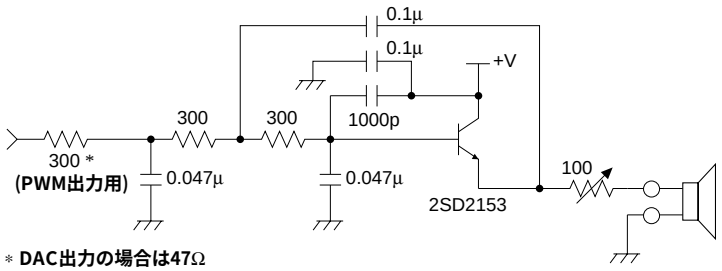
8kHzサンプリング用トランジスタアンプ回路



トランジスタは、hfe(電流増幅率)が500以上のものを選びます。
電流増幅となりますので、ローパスフィルタ部は低インピーダンスにします。D/A変換部からトランジスタまでを1kΩ程度にするとバランスは良好です。それ以上にすると、音量がどんどん小さくなります。逆にそれ以下にするには、コンデンサ容量や電流値も含め、設計が難しくなります。また、音量もわずかしか上がりません。
上記例の、カットオフ周波数は約2.5kHzです。

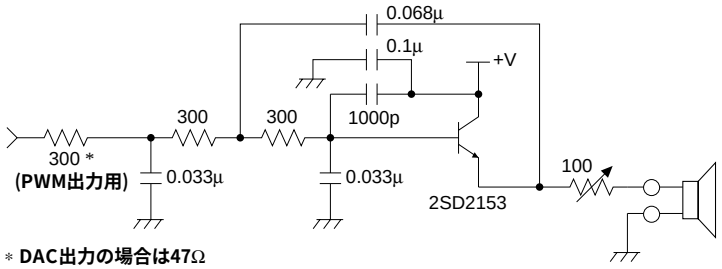
S1C33104のDACから入力する場合、初段の300Ω抵抗はDACの内部直列抵抗の約250Ωを引いた47Ωに替えてください。
また、+Vにつながっている0.1µFは電源のデカップリング用、1000pFは発振防止用です。これらを入れないと、トランジスタ出力が発振する場合があります。
スピーカの前の100Ω可変抵抗は音量調整用です。

16kHzサンプリング用トランジスタアンプ回路



ローパスフィルタのカットオフ周波数は約5kHzです。

22.05kHzサンプリング用トランジスタアンプ回路



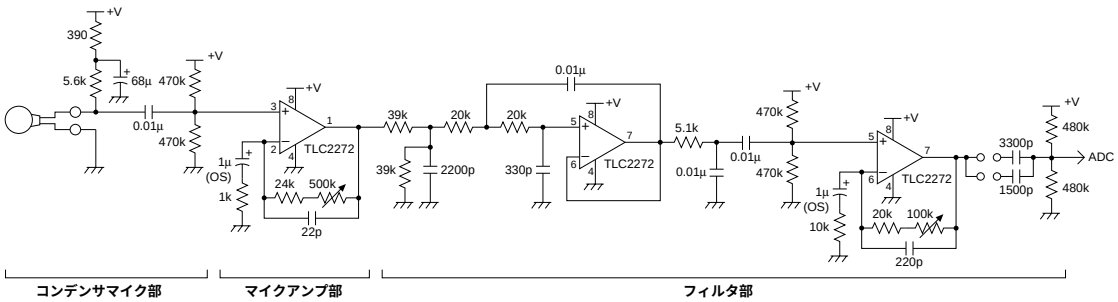
ローパスフィルタのカットオフ周波数は約8kHzです。

ここで用いたローパスフィルタ回路は、S1C33209のPWMやS1C33104のDACと組み合わせて使用することができます。

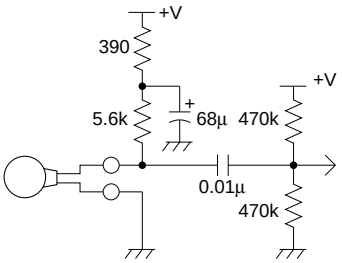
5.5　音声入力アナログ回路例

ここでは、A/D変換器を使用して音を入力する方法について説明します。

音声入力回路の構成は入力ソース等によっても異なりますが、ここでは上記のように考えます。
（S5U1C330A1D1回路に準拠）



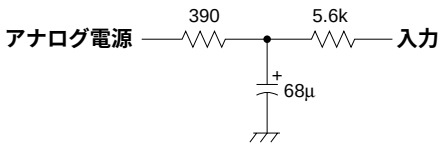
● コンデンサマイク部



コンデンサマイクとACカップリング

コンデンサマイクに電源を供給するラインの5.6kΩ抵抗は、本来は1.5kΩがメーカーの推奨値です。これは、ここの電位差が入力信号レベルとなるためで、後段のマイクアンプの負荷を減らすため、抵抗値を大きくして3.7倍のゲインを得ています。また、消費電流を低減する効果もあります。ただし、あまり大きくすると電流が小さくなりすぎてコンデンサマイク自体が不安定になりますので、4倍程度が限界です。ここではmV以下の微少信号を扱いますので、金属皮膜抵抗を使用します。

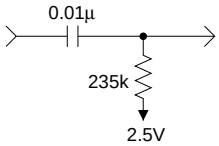
電源ノイズも含め、ここに現れるノイズは後段も含めたゲイン分そのまま増幅されますので、最も低ノイズが要求されます。このために、以下のようにアナログ電源には、390Ωと68µFのカットオフ周波数5Hzの1次ローパスフィルタを設けて音声帯域のノイズを広くカットしています。



68µFは電解コンデンサで問題ありません。

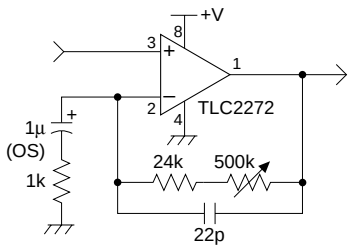
次に、0.01µFのコンデンサとGND、電源への470kΩの抵抗で1/2電源電圧にACカップリングし、かつ1次のハイパスフィルタとしてDC成分をカットします。カットオフ周波数は約70Hzで、それ以下を減衰させます。

ハイパス部等価回路



ACカップリング用の抵抗は1%以下の精度ものを使用します。ここで正確に中点を設定しておかないと、たとえば5%精度ではマイクアンプの大きな増幅によって信号がV_{DD}–GNDの範囲を越え、クリッピングされてしまう可能性がでてきます。増幅率によっては0.5%精度を使用する必要もあります。
ハイパス用コンデンサは微少信号が通過しますので、フィルムコンデンサ(ポリエステル)を使用します。セラミックなどでは音質が劣化する場合があります。

● マイクアンプ部



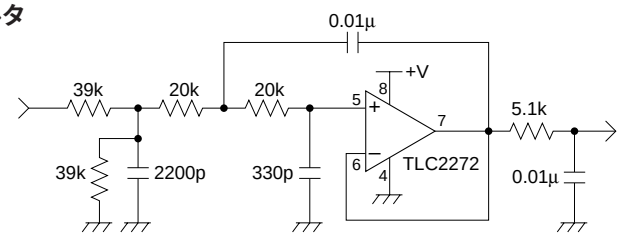
このAC増幅器は、ゲインを24倍から524倍まで可変抵抗器で調整できます。コンデンサマイク部の3.7倍と併せると、90倍から2000倍となります。ただし、524倍は実験用のため、実際の製品ではアンプを2段構成にするなど、考慮が必要です。なお、同じゲインの場合、2段より1段の増幅の方がノイズは減ります。

ゲインは500kΩの可変抵抗によって24k/1k = 24倍から(24k+500k)/1k = 524倍の範囲で調整します。
可変抵抗は、1、2、5の値から選択しておくと入手が容易です。
24kΩ+500kΩに並列に接続する22pFは、広域でゲインを下げるローパスフィルタです。ただし、OPアンプの発振防止用のためカットオフ周波数は高く、また可変抵抗の値により変わります。
このようなフィードバック系のローパスフィルタは、発振防止効果はやや低めです。本来は入力部でCRのローパスフィルタでゲインを下げた方が、カットオフ周波数も固定となり発振防止効果も高いのですが、すでにACカップリングしていますので、CRのローパスフィルタは見送りました。

1kΩと1µFは、150Hzをカットオフとする1次のハイパスフィルタを構成します。
本書で対象としているローコストシステムでは、50、60Hzを含むハムノイズや低周波はいろいろな障害の原因となります。前段のACカップリングとともに、このフィルタでこれらのノイズを最低限除去します。残ったノイズは後段のフィルタ部でさらに除去します。

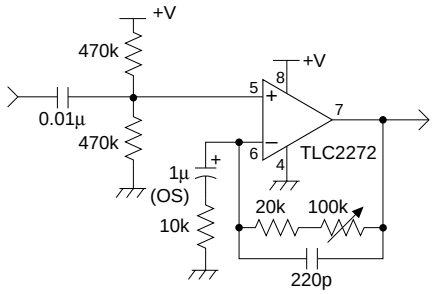
● フィルタ部

4次ローパスフィルタ



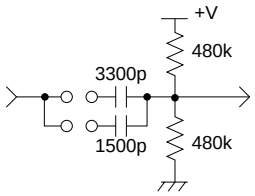
S5U1C330A1D1ボードには、この図のとおりカットオフ3.5kHzのマイク用ローパスフィルタが実装されています。これにより不要な高域成分がカットされ、音に落ち着きが出ます。ただし、効果はそれほど大きくありませんので、省略することもできます。また、OPアンプに合わせ、振幅を抵抗分割で1/2にしています。これは、次のAC増幅器のゲインを考慮した値です。

AC増幅器



この回路は、2倍から20倍のAC増幅器です。
0.01µFと470kΩはカットオフ70Hzの1次ハイパスフィルタ、10kΩと1µFは15Hzの1次ハイパスフィルタ、20kΩ+100kΩ(20～120kΩ)と220pFは50kHz～10kHzの1次ローパスフィルタを構成します。
高い周波数まで増幅したいときは、220pFを小さくします。この容量に反比例してカットオフ周波数が大きくなります。

ハイパスフィルタ

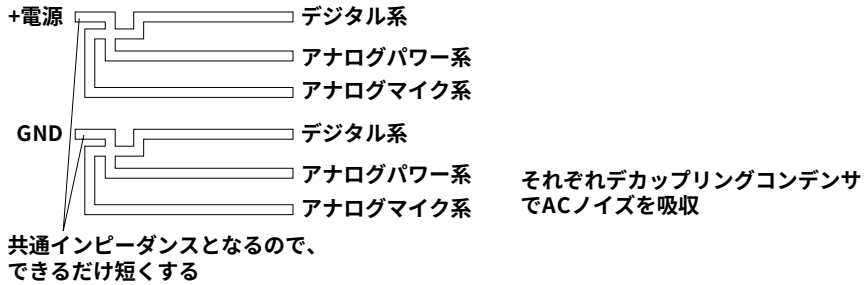


ここでは、電源電圧の1/2にACカップリングし、かつ音声圧縮に悪影響を及ぼす低域をハイパスフィルタでカットします。
コンデンサの容量とカットオフ周波数の対応は次のとおりです。
4800pF: 250Hzカットオフ
3300pF: 300Hzカットオフ
1500pF: 500Hzカットオフ

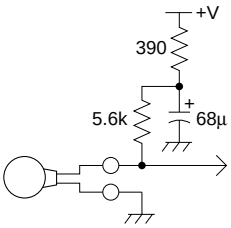
S5U1C330A1D1のデフォルトは4800pFですが、使用環境により他の容量も試してみてください。たとえば、音声圧縮・伸長ミドルウェアS5U1C330V1Sに入っているVSX音声圧縮はDC系ノイズに弱いので、500Hzがよい場合もあります。

● アナログ電源について

アナログ系にデジタル系と同じ電源を使用すると、ノイズ等の問題が起きやすくなります。できれば、アナログ系に専用の電池やリニアレギュレータを使用し、デジタル系と分離するようにしてください。さらに、スピーカなどの重負荷系とマイクなどの微少電圧系を分けるとさらに効果的です。
レギュレータを複数用意するのが最良ですが簡易的には、1点アース(電源中心にGNDを1点で結ぶ)により共通インピーダンスを持たせないようにすることも効果はあります。



また、マイコンではプログラムにより周期的に負荷変動する場合があります。それが電源変動としてマイク入力に影響すると問題になります。この変動を吸収するには、レギュレータを分離したり、S5U1C330A1D1のように数～10Hzカットオフのローパスフィルタをコンデンサマイクの電源に入れる方法が効果的です。



リニアレギュレータでも、低ドロップタイプなどはノイズがマイク入力に影響します。安全のため、マイク入力回路には、無条件にこのローパスフィルタをつけることを推奨します。

S5U1C33209D1のようなスイッチング電源の場合は、出力コンデンサに低ESRのOSコンデンサやSPキャップを使用し、リプルを最低限度に押さえてください。電解コンデンサはノイズが大きくなるため絶対に使用しないでください。
S5U1C33209D1+S5U1C330A1D1では各種の特性を計測し、マイク電源のローパスフィルタのみでノイズを押さえています。実際にはACカップリング部、OPアンプの電源など、まだ不完全と思われる部分もあります。
基本的にはスイッチングレギュレータを使用せず、問題の少ないリニアレギュレータを使用することを推奨します。スイッチングレギュレータを使用する場合は、実機検証と各種ノイズ対策が 不可欠です。

5.6　PWMによる15ビットD/A変換

S1C33209はセイコーエプソン独自のハイブリッドPWM技術により、最大15ビット精度、8kHzから48kHzのサンプリング周波数をサポートし、CDに近い高音質をきわめて低いコストで実現できます。

ハイブリッドPWM技術は以下の3つの手法を組み合わせて構成されます。

(1) ファインPWM
5.3項で説明したとおり、半クロック単位でPWM出力をコントロールすることにより、1ch単独で最大10ビット精度の音声/音楽出力ができます。

(2) デュアルPWM
上記ファインPWMを2チャネル合成することにより、最大15ビットの高精度で、音声/音楽出力ができます。

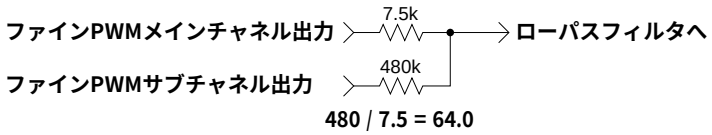
(3) ソフトアジャストPWM
PWM出力時、ソフトウェアの補正処理により直線性誤差が0.01%という高精度の出力を実現します。

ここでは、デュアルPWMとソフトアジャストPWMについて説明します。

● デュアルPWM

基本原理

デュアルPWMは同じ出力データを2つのチャネルからファインPWMで出力し、外部抵抗で合成することによってビット精度を拡張する手法です。推奨としてはメインチャネルとサブチャネルを1対64の比率で合成します。合成はローパスフィルタを通す前のPWM波形のまま行います。

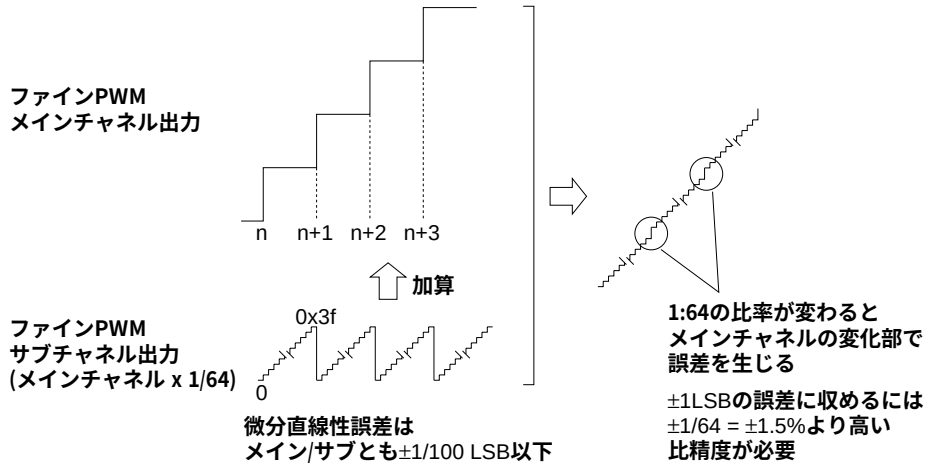


ファインPWMはきわめて微分精度が高く、実測レベルで1/100 LSB以下という誤差です。（なお、PLLはx2以上で使用してください。x1のOSC3そのままではデューティ比が崩れ、このような微分精度にはなりません。1chのファインPWMには、x1でも問題ありません。）

メインのPWMに、正確に1/64したサブのPWMを加えることにより、メインチャネルのみのビット数に6ビットの精度を加えることができます。メインチャネルはノイズ低減のため、搬送波は160kHz以上とします(320kHzが上限、それ以上はあげないこと)。これにより、メインチャネルの精度は9ビットとなります(40MHz以上で動作時)。サブチャネルの6ビットを加えると合計で15ビットとなります。

抵抗精度

1対64を構成する抵抗の精度は、D/A変換の精度に影響を与えます。
正確に480.0kΩと7.5kΩであれば問題ありませんが、実際の量産に使用可能な抵抗は、コスト面から±1%誤差か、±0.5%誤差です。また、480kΩはE24系列にないため、手に入らない場合は470kΩ + 10kΩの2抵抗で実現します。
この誤差が大きく影響する箇所は、メインチャネルの変化部です。サブチャネルが正確にメインチャネルの1/64になっている場合、メインチャネルの変化部でサブチャネルが0x3fから0x0に変わります。合成抵抗の誤差は、この位置関係にずれを生じさせます。
ここの部分のみの微分誤差は次のようになります。
0.1%誤差の抵抗:　15ビット±1LSB以下
0.5%誤差の抵抗:　14ビット±1LSB以下
1%誤差の抵抗:　13ビット±0.7LSB以下

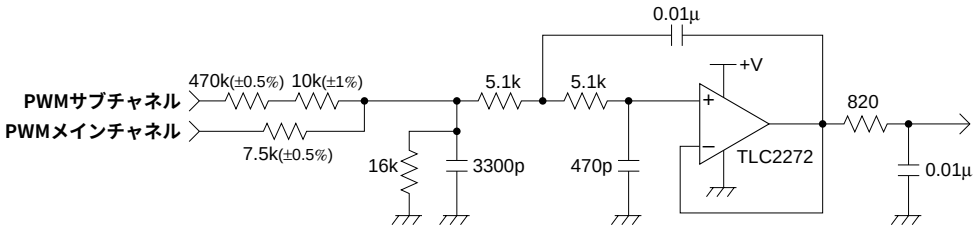


ただし、サブチャネルがその他の値に変化する63/64のパターンには15ビット±0.5LSB程度の微分精度が適用されますので、音質が上記の誤差で大きく低下するわけではありません。それでも、できれば0.5%精度の誤差の少ない抵抗を使用することを推奨します。最低でも1%誤差のものを使用してください。5%誤差の抵抗は使用しないでください。
高精度を必要とする抵抗は抵抗合成する2〜3個のみで、後段のローパスフィルタには5%誤差のものも使用可能です。

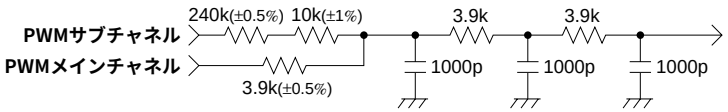
● 回路例（S5U1C330A3D1）

32kHz以上のサンプリング用ローパスフィルタ

4次のOPアンプ方式



3次のCR方式



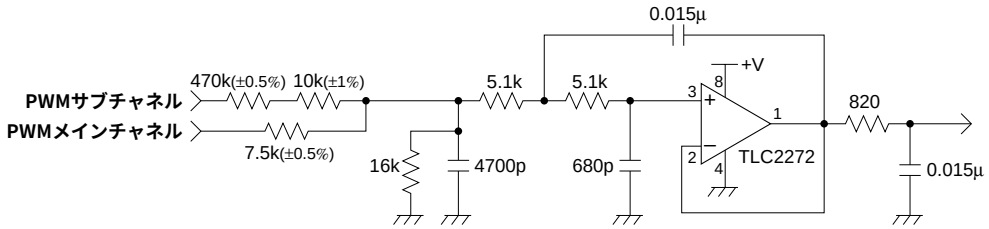
通常のローパスフィルタの前に、前期の合成用抵抗を設け2チャネルのPWM出力を接続します。合成用抵抗の比率はできるだけ64.0倍に近づけてください（計算で±0.2%以内の誤差、63.87〜64.13倍程度）。
入手の容易なE24系列の抵抗値を使用し、手に入りにくい抵抗値は2個使用して解決します。
合成用抵抗には高精度（0.5%〜1%）のものを使用します。
上記例の抵抗値は、次のように±0.2%以内に収まっています。
480k/7.5k = 64.0　（480k = 470k + 10k）
250k/3.9k = 64.10　（250k = 240k + 10k）

減衰率を重視し、CRフィルタは3段としています。32kHzサンプリングでは差はわずかですが、OPアンプを使用した4次フィルタの方が効果はあります。

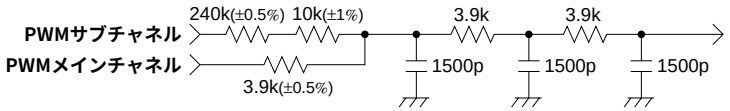
以下の回路は、コンデンサの値を変え、カットオフ周波数を変えて22.05kHzサンプリングと16kHzサンプリングに対応させた例です。いずれも、初段の合成用抵抗の比率は1対64にしてあります。

22.05kHzサンプリング用ローパスフィルタ

4次のOPアンプ方式

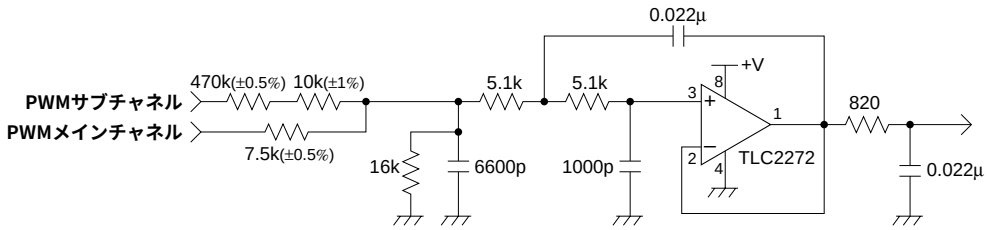


3次のCR方式

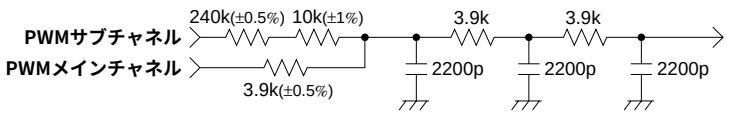


16kHzサンプリング用ローパスフィルタ

4次のOPアンプ方式

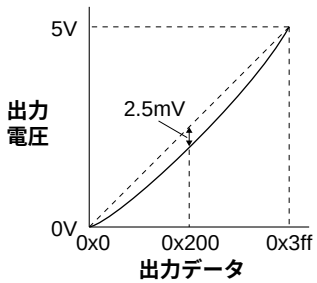


3次のCR方式



● ソフトウェアによる直線性補正

ファインPWM技術における微分精度は、1/100 LSB以下(実測値)という究極の精度を実現しています。直線性誤差はそれほど小さくはありません。中心部が下がったお椀型の特性になります。
これはPWMのHigh出力とLow出力でインピーダンスが微妙に異なるためで、ローパスフィルタ初段の抵抗とS1Cチップ内部の等価抵抗の差が分かれば、ずれ方は理論的に計算できます。
たとえばPWMの出力電圧が0.0Vと5.0Vの場合、初段の抵抗が3.9kΩでは中心部が2.5mV下にたわみます。 7.5kΩでは1.3mV、 390Ωでは25mVたわみます。



これを、以下のようなテーブル(3.9kΩに対応)を使用して補正します。

テーブル例）

```
const unsigned char ucAdj18 [] = {  // PWM adjust for 3.9Kohm with 18bit precision
        0x4,    // 0
        0x8,    // 1
        0xc,    // 2
        0x10,   // 3
        0x14,   // 4
        0x17,   // 5
        0x1b,   // 6
        0x1f,   // 7
        0x22,   // 8
        0x26,   // 9
        0x29,   // a
        0x2d,   // b
        0x30,   // c
        0x33,   // d
        0x36,   // e
        0x39,   // f
        0x3c,   // 10
        0x3f,   // 11
        0x42,   // 12
        0x45,   // 13
        0x48,   // 14
        0x4b,   // 15
        0x4d,   // 16
        0x50,   // 17
        0x52,   // 18
        0x55,   // 19
        0x57,   // 1a
        0x5a,   // 1b
        0x5c,   // 1c
        0x5e,   // 1d
        0x60,   // 1e
        0x62,   // 1f
        0x64,   // 20
        0x66,   // 21
        0x68,   // 22
        0x6a,   // 23
        0x6c,   // 24
        0x6d,   // 25
        0x6f,   // 26
        0x71,   // 27
        0x72,   // 28
        0x74,   // 29
        0x75,   // 2a
        0x76,   // 2b
        0x77,   // 2c
        0x79,   // 2d
        0x7a,   // 2e
        0x7b,   // 2f
        0x7c,   // 30
        0x7d,   // 31
        0x7e,   // 32
        0x7e,   // 33
        0x7f,   // 34
        0x80,   // 35
        0x80,   // 36
        0x81,   // 37
        0x81,   // 38
        0x82,   // 39
        0x82,   // 3a
        0x83,   // 3b
        0x83,   // 3c
        0x83,   // 3d
        0x83,   // 3e
        0x83,   // 3f
        0x83,   // 40
        0x83,   // 41
        0x83,   // 42
        0x83,   // 43
        0x82,   // 44
```

```
        0x82,  // 45
        0x82,  // 46
        0x81,  // 47
        0x80,  // 48
        0x80,  // 49
        0x7f,  // 4a
        0x7e,  // 4b
        0x7e,  // 4c
        0x7d,  // 4d
        0x7c,  // 4e
        0x7b,  // 4f
        0x7a,  // 50
        0x79,  // 51
        0x78,  // 52
        0x76,  // 53
        0x75,  // 54
        0x74,  // 55
        0x72,  // 56
        0x71,  // 57
        0x6f,  // 58
        0x6d,  // 59
        0x6c,  // 5a
        0x6a,  // 5b
        0x68,  // 5c
        0x66,  // 5d
        0x64,  // 5e
        0x62,  // 5f
        0x60,  // 60
        0x5e,  // 61
        0x5c,  // 62
        0x5a,  // 63
        0x57,  // 64
        0x55,  // 65
        0x52,  // 66
        0x50,  // 67
        0x4d,  // 68
        0x4b,  // 69
        0x48,  // 6a
        0x45,  // 6b
        0x42,  // 6c
        0x3f,  // 6d
        0x3c,  // 6e
        0x39,  // 6f
        0x36,  // 70
        0x33,  // 71
        0x30,  // 72
        0x2d,  // 73
        0x29,  // 74
        0x26,  // 75
        0x22,  // 76
        0x1f,  // 77
        0x1b,  // 78
        0x17,  // 79
        0x14,  // 7a
        0x10,  // 7b
        0xc,   // 7c
        0x8,   // 7d
        0x4,   // 7e
        0x0,   // 7f
};
```

このテーブルの値は上位7ビットにより補正値を引いてきて、18ビット精度でデータを作成してありますので、最終的には3ビット右シフトして足し合わせて補正をかけます。
この補正により、直線性誤差を平均±0.2mV程度に、大きなところでも±1mV程度に押さえることができます。±1mVの誤差は、5Vに対して12ビット±1LSB程度です。

補正しないとひずみという形で波形に現れますが、2.5mV程度では聴感上ほとんど差は出ませんので、補正は必須ではありません。音声ミドルウェアではソフトウェアの負荷を減らすため、処理をはずしています。

5.7　圧電ブザーでのメロディ出力

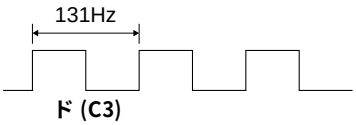
ここでは、PWMによるメロディ出力と圧電ブザーの接続方法について説明します。

● PWMとメロディ

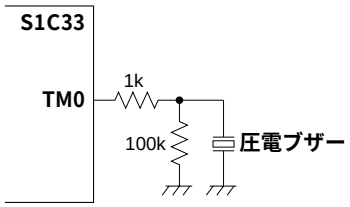
人間の耳は音の周波数で音階を聞き分けます。たとえば、131Hzの音はド(C3)と聞こえます。262Hzは1オクターブ高いド(C4)、65.5Hzは1オクターブ低いド(C2)となります。
1オクターブ(2倍の周波数まで)を12等分して 約6%ずつ増加するように周波数を上げていくと、音程が半音ずつ上がって聞こえます。これでド、レ、ミ...という音階を表現できます。

S5U1C331M2Sミドルウェアや一般的なメロディICは、PWM(矩形波)の波形によりこれらを表現します。なお、完全な50%デューティの波形の場合、基本の周波数に3倍、9倍というように、3倍単位の高調波がのり、実際の音階のほかにかなりの高音成分が含まれます。

● 1ch出力

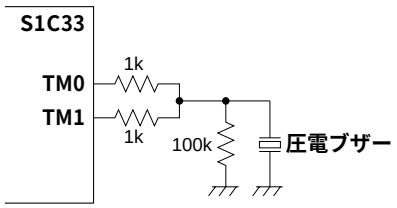


周波数が131Hzの出力波形はド(C)の音になります。



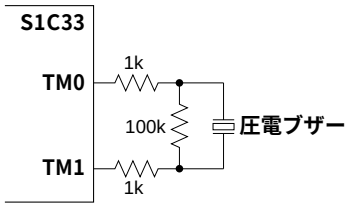
1chの出力では、このように圧電ブザーを駆動します。

● 2ch合成



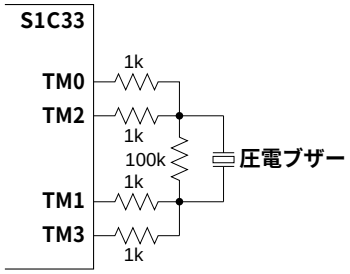
このように2ch以上を合成することもできます。

● 差動出力



PWMを1ch反転した差動出力を行い、音量を大きくすることもできます。

● 差動出力、2ch合成

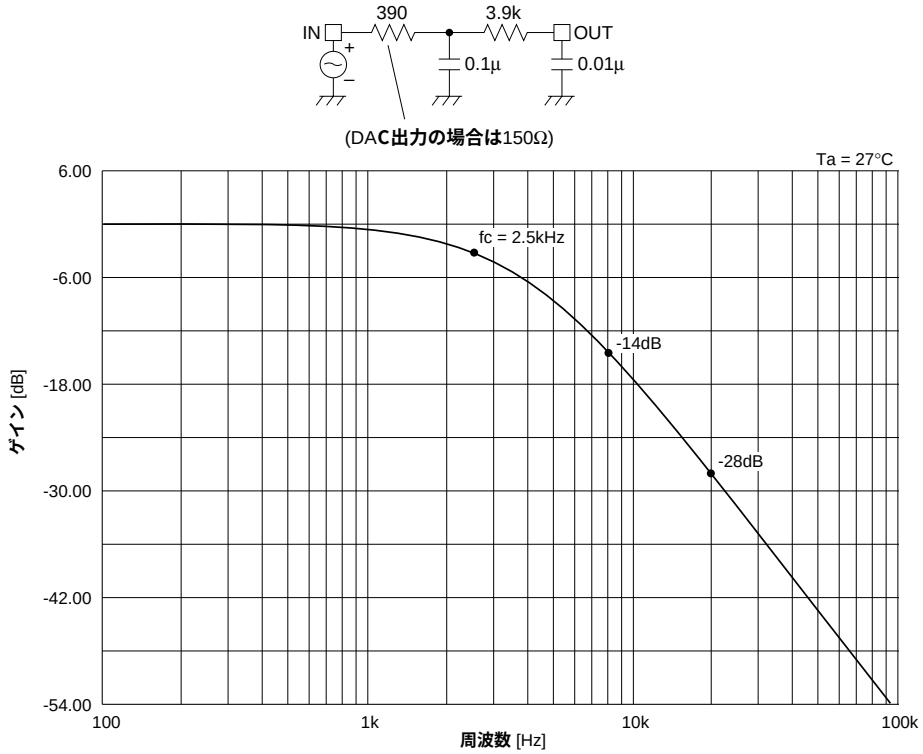


差動出力を2つ使用して、2ch合成と差動出力の組み合わせもできます。

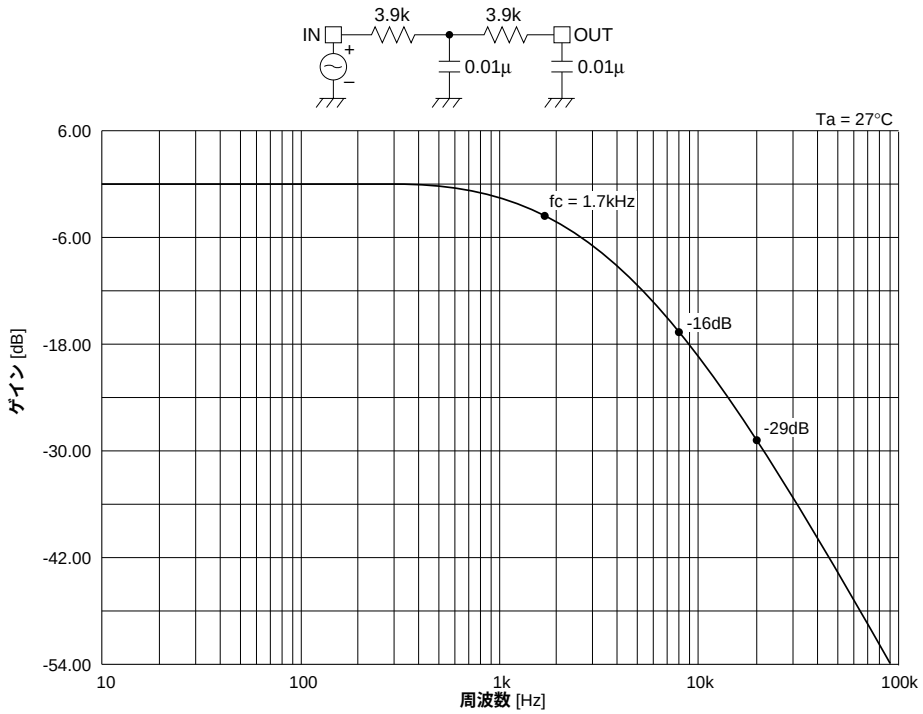
5.8　<参考資料> 特性グラフ

● CR 2次ローパスフィルタ周波数特性（8kHzサンプリング用）

(1) fc = 2.5kHz

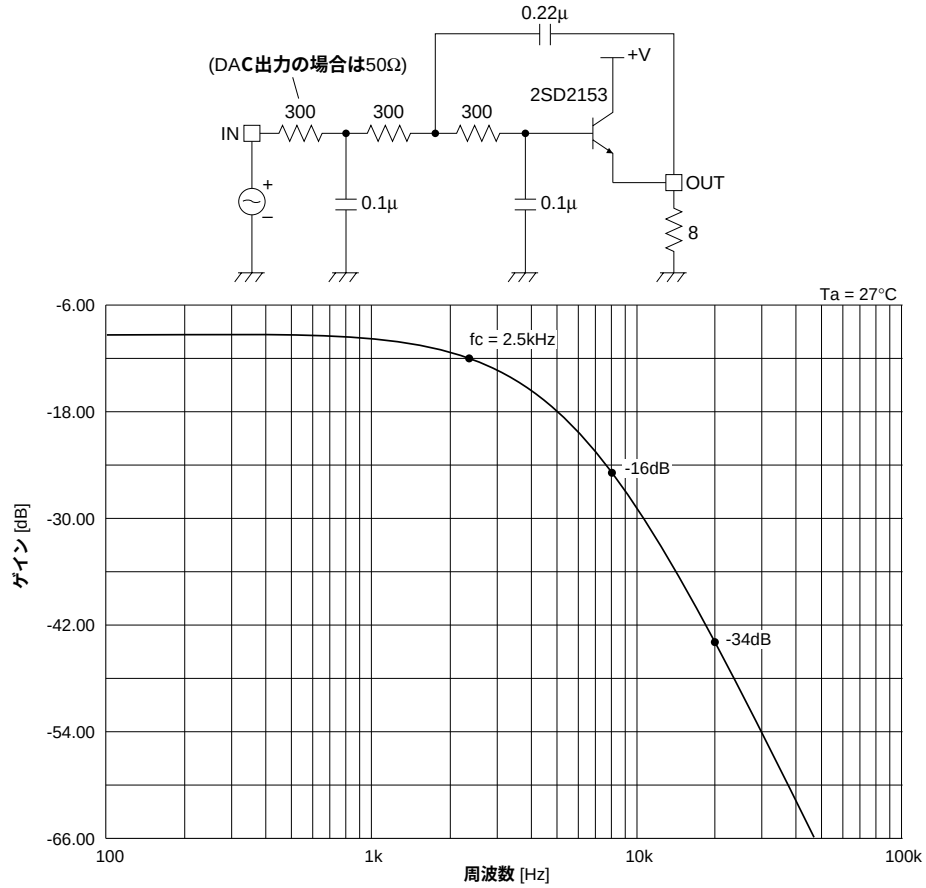


(2) fc = 1.7kHz



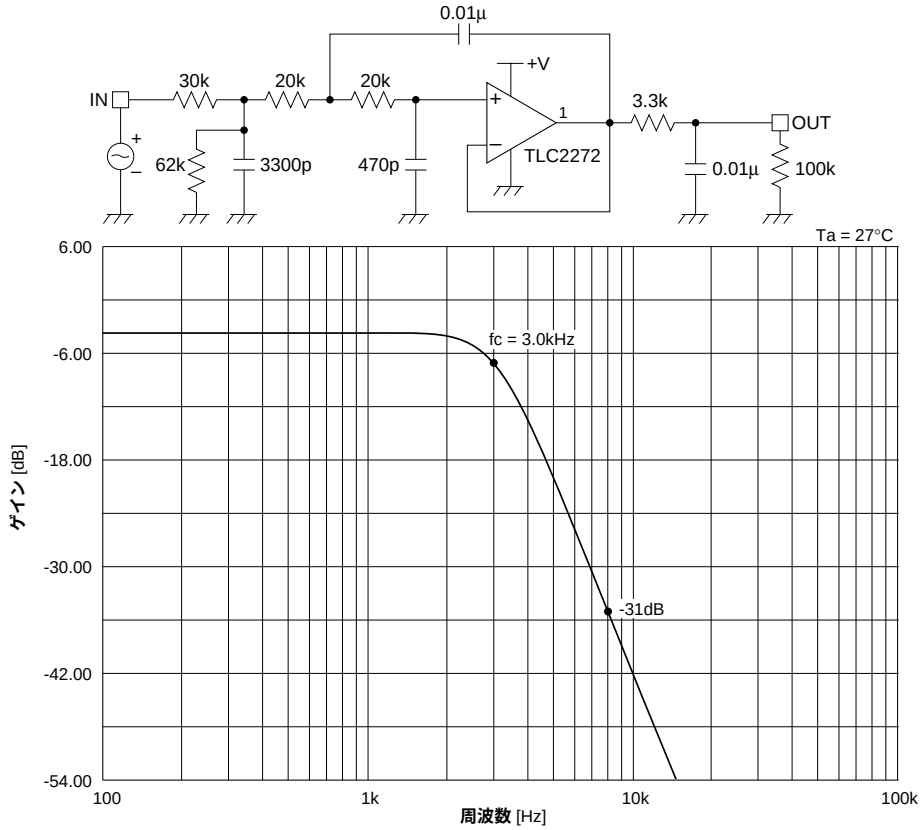
● トランジスタ3次ローパスフィルタ周波数特性（8kHzサンプリング用）

fc = 2.5kHz



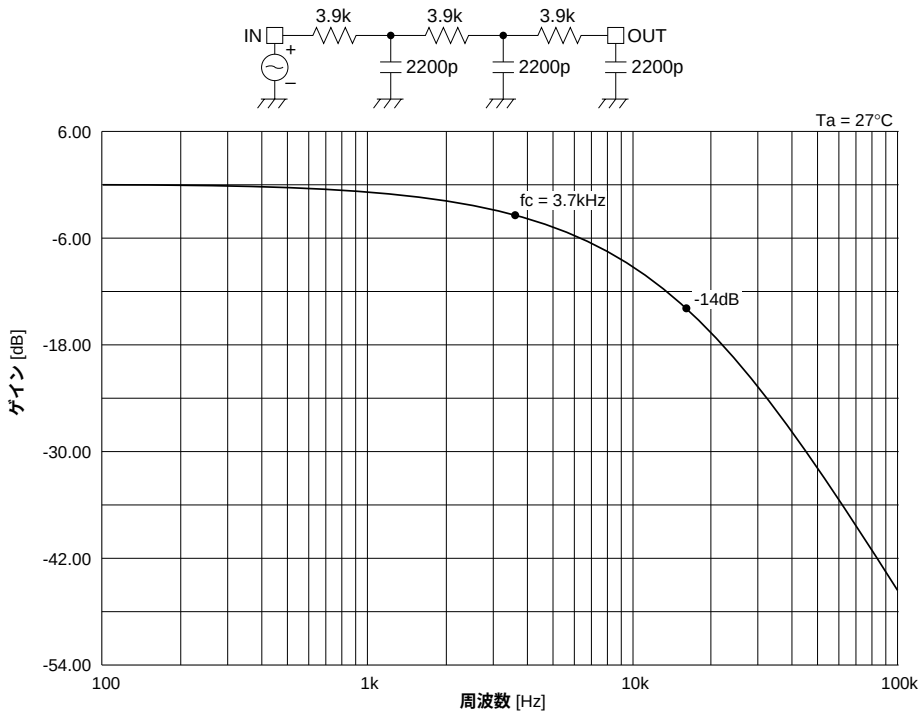
● オペアンプ4次ローパスフィルタ周波数特性（8kHzサンプリング用）

fc = 3kHz

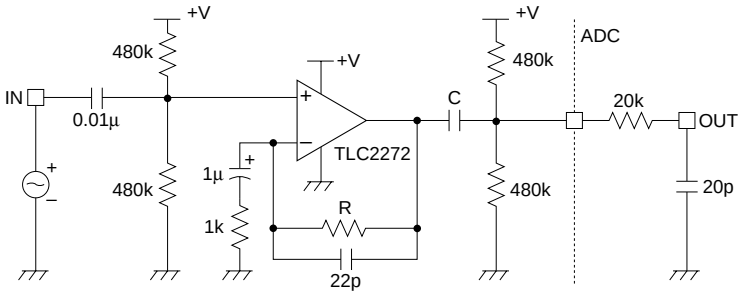


● CR 3次ローパスフィルタ周波数特性（16kHzサンプリング用）

fc = 3.7kHz

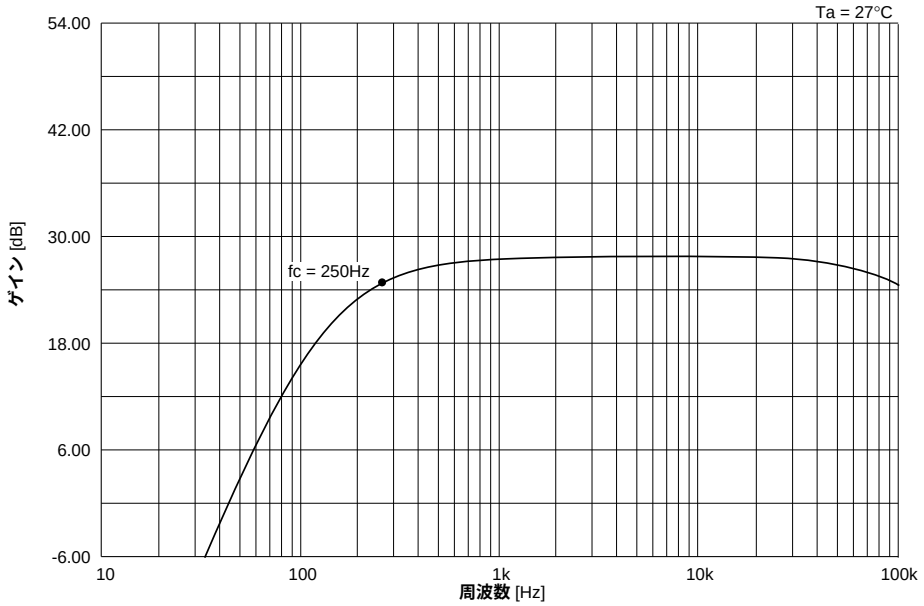


● ACアンプ ハイパスフィルタ周波数特性(8kHzサンプリング用)

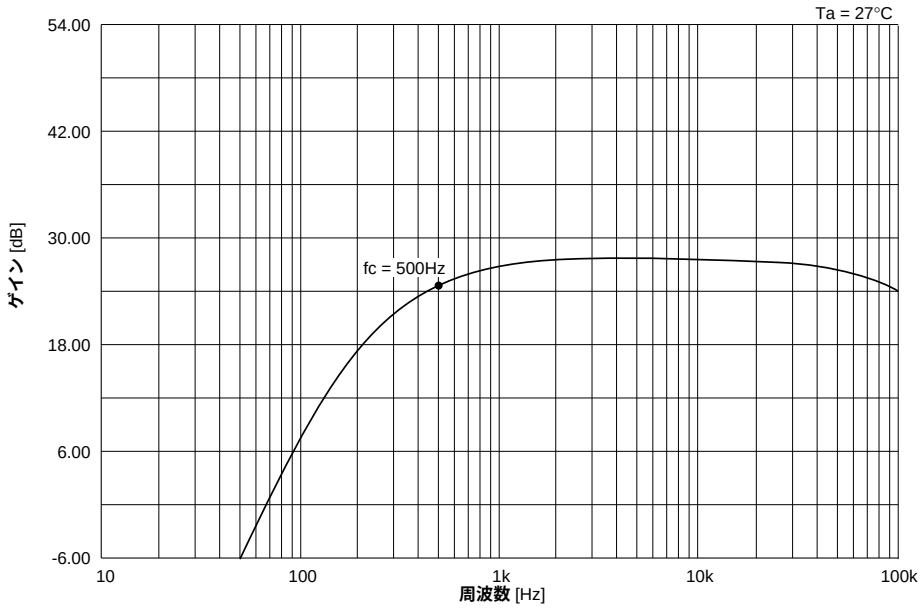


(1) R = 24kΩ, C = 0.0048µF (fc = 250Hz)
(2) R = 24kΩ, C = 0.0015µF (fc = 500Hz)

(1) fc = 250Hz



(2) fc = 500Hz



S1C33 Family
スタンダードコア用アプリケーションノート
(S5U1C33001Cツール対応)

セイコーエプソン株式会社
電子デバイス営業本部

〈ED東日本営業部〉
東京　〒191-8501　東京都日野市日野421-8
TEL (042)587-5313(直通)　FAX (042)587-5116

仙台　〒980-0013　宮城県仙台市青葉区花京院1-1-20　花京院スクエア19F
TEL (022)263-7975(代表)　FAX (022)263-7990

〈ED西日本営業部〉
大阪　〒541-0059　大阪市中央区博労町3-5-1　エプソン大阪ビル15F
TEL (06)6120-6000(代表)　FAX (06)6120-6100

名古屋　〒461-0005　名古屋市東区東桜1-10-24　栄大野ビル4F
TEL (052)953-8031(代表)　FAX (052)953-8041

インターネットによる電子デバイスのご紹介　http://www.epsondevice.com/domcfg.nsf

ドキュメントコード：404861900

2004年2月作成 ㊊A