EPSON
EXCEED YOUR VISION

CMOS 16-BIT SINGLE CHIP MICROCOMPUTER

S1C17 Family
スタートアップマニュアル
アセンブラ版

セイコーエプソン株式会社

Rev. 1.1

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。

本資料の内容については、予告無く変更することがあります。

1. 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いたします。

2. 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知的財産権およびその他の権利侵害あるいは損害の発生に対し、弊社はいか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知的財産権およびその他の権利の実施権の許諾を行うものではありません。

3. 特性値の数値の大小は、数直線上の大小関係で表しています。

4. 製品および弊社が提供する技術を輸出等するにあたっては「外国為替および外国貿易法」を遵守し、当該法令の定める手続きが必要です。大量破壊兵器の開発等およびその他の軍事用途に使用する目的をもって製品および弊社が提供する技術を費消、再販売または輸出等しないでください。

5. 本資料に掲載されている製品は、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いかなる責任についても負いかねます。

6. 本資料に掲載されている会社名、商品名は、各社の商標または登録商標です。

はじめに

S1C17 Family は、各種センサに対応可能な豊富なインタフェース、幅広い表示領域をカバーする LCD ドライバ/コントローラなど、多彩な周辺回路を内蔵した 16 ビット RISC プロセッサです。高速動作かつ低消費電力を実現し、携帯機器に適した製品を提供します。また、Flash Rom 内蔵製品を多数ラインナップ。充実した開発環境やオンチップ IC 機能により開発期間の短縮も可能にします。
本書では、S1C17 Family 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、S1C17 Family の基本的な組み込み用プログラミング方法などを説明します。

本書をお読みになるには、以下の内容が予備知識として必要です。
- アセンブリ言語に関する知識およびアセンブラソースプログラムの作成方法
- C 言語（ANSI C 準拠）に関する一般的な知識
- GNU に関する知識
- Windows2000 またはWindows XP の基本的な操作方法

なお、本書に掲載のプログラム例は S1C17FamilyC/C++コンパイラパッケージ（S5U1C17001C）の Ver.1.2.1 を使用して作成しています。

＜マニュアルの構成＞
本マニュアルは、以下に示す 4 つの章から構成されています。
1 章では、組み込み用ソフトウェアを作成するための基礎知識を掲載しています。
2 章では、S1C17 Family の基本的なプログラミングについて、サンプルプログラムを使用して説明しています。
3 章では、C とアセンブラの混在法についてサンプルプログラムを使用して説明しています。
4 章では、アセンブラでプログラミングする時の注意点を記載しています。

＜関連マニュアル＞
関連マニュアルは下記のとおりです。
- S1C17 コアマニュアル
- S5U1C17001C Manual（S1C17 Family C コンパイラパッケージ）
- S1C17 Family 各期種別テクニカルマニュアル

目　次

1. 組み込みの基礎知識

本章では、初めて組み込みソフトウェア開発をする方を対象に、最初に理解していただきたい、プログラムが動くための基本メカニズムやスタートアップルーチンによる初期化など、組み込みソフトウェア開発では極めて重要な考え方について説明します。

1.1 プログラムが動くための基本メカニズム

はじめに、S1C17 プロセッサ（以下 MCU）が動き始めるときの動作（基本メカニズム）について説明します。

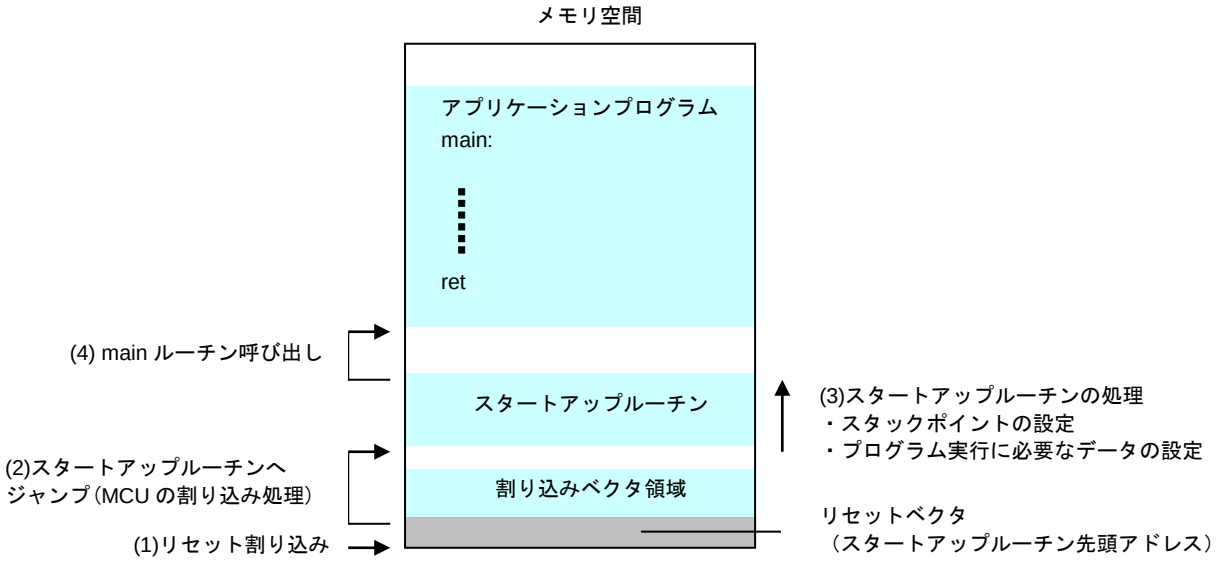


図 1.1　S1C17 プロセッサ起動時の基本メカニズム

(1) MCU に電源を投入するとリセット割り込みが発生し、MCU はベクタテーブル内の先頭のアドレスを読み込みます。
(2) MCU は(1)で読み出した内容（アドレス）にジャンプすることで、スタートアップ（初期化設定）ルーチンを呼び出します。
(3) スタートアップルーチンは、まずスタックの設定やプログラムの実行に必要な初期化処理を行います。
(4) 初期化処理を終了後、スタートアップルーチンは main ルーチンを呼び出します。

※ベクタテーブルは、各種割り込み処理ルーチンのアドレスを書き込んでおくテーブルで、割り込み発生時はここからアドレスが読み出され、対応する処理ルーチンにジャンプするようになっています。

組み込みアプリケーションのプログラムは main ルーチンからではなく、その前段階としてスタートアップルーチンというものが存在します。組み込みソフトウェアを開発するにあたって、プログラムを動かすためにはスタートアップルーチンを理解することが必要です。

1.2 スタートアップ（初期化設定）ルーチン

組み込みソフトウェアでは、main ルーチンを実行する前にスタートアップルーチンにて必要な初期化などを行います。一般に次のような処理を行います。
・スタックポインタの設定
・プログラムの実行に必要なデータの設定
  - 初期値を持たないメモリ領域のクリア（.bss セクションのクリア）
  - ROM 領域から RAM 領域への初期値データの転送（.data セクションのコピー）
・ハードウェアの初期化と割り込みの設定

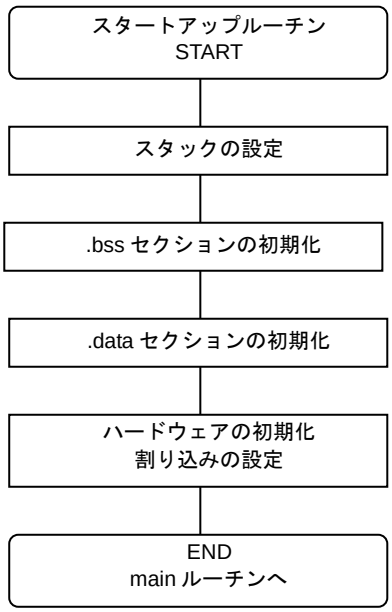


図 1.2　スタートアップルーチン

スタックはサブルーチンや関数を呼び出す際に、処理中のデータや戻りアドレスなどを一時的に退避するのに使う RAM 領域です。割り込みもスタックを使用するため、スタートアップルーチンでスタック領域を確保します。
プログラムの実行には初期値を持たないグローバル変数の初期化が必要になります。これらの設定はリセットによって不定になるため、初期化する必要があります（.bss セクションのクリア）。また初期値を持つグローバル変数は、初期値データを ROM から RAM にコピーする必要があります（.data セクションのコピー）。
そのほかに、ソフトウェアの実行に関わる初期化だけでなく、MCU やその他のハードウェアを使用するための初期化を実施します。また割り込みの設定として、マスク可能な外部割込みを許可します。

組み込みアプリケーションでは、このようにスタートアップルーチンを実行した後に main ルーチンが呼び出されます。

以上のことを念頭におき、組み込み機器のプログラムの開発を行ってください。

2. S1C17 用プログラムの作成方法

本章では、S1C17 Family 共通のプログラムの作成方法について説明します。
1 章で説明したように、組み込みアプリケーションでは main ルーチンを実行する前処理として、スタートアップルーチンを実行する必要があります。以下、スタートアップルーチンを含め、main ルーチンを呼び出すまでの処理の流れについて、サンプルプログラムを参考に説明します。

2.1 GNU17 を使用しての開発手順

GNU17 とは、C ソースプログラムのコンパイル、アセンブラソースプログラムのアセンブラからデバッグを行う一連のソフトウェアツールやユーティリティが含まれている統合開発環境です。
GNU17は、「EPSON　マイコンユーザーサイト」の「S1C17ファミリー」から「S1C17ソフトウェア統合開発環境GNU17」をダウンロードしてからインストールしてください。
GNU17を使用しての開発の流れは次のようになります。

- プロジェクトの作成
  GNU17を使用し、新規プロジェクトを作成します。
- ソースプログラムの作成
  GNU17のエディタまたは汎用のエディタを使用してソースファイルを作成し、プロジェクトに追加します。
- プログラムのビルド
  GNU17を使用し、Cコンパイラからリンカまでの起動オプションやリンカスクリプトを設定します。その後GNU17からビルドを実行すると、デバッグ可能なelf形式のオブジェクトファイルと、それをSレコード形式に変換したROMデータファイル（psaファイル）が生成されます。
- デバッグ
  リンカが生成したelf形式のオブジェクトファイルとSレコード形式のROMデータファイルを使用して、動作の確認とデバッグをデバッガで行います。デバッガ用の設定と起動はGNU17から行えます。

さらに詳しい説明については、「S5U1C17001C MANUAL」内の「ソフトウェア開発手順」の章を参照してください。「S5U1C17001C MANUAL」はGNU17をインストールした後、「EPSON¥GNU17¥doc」ディレクトリ内にあります。

2.2 ベクタテーブルの作成

S1C17用プログラムの実行には、ベクタテーブルとスタートアップルーチンが最低限必要になります。ここではベクタテーブルについて説明します。スタートアップルーチンについては2.4で説明します。

リスト 2.1　ベクタテーブルの例

```
;--------BOOT.s---------
.section .rodata

.global BOOT
.global DUMMY
.global NMI

;--- vector table ---
                    ; NO       BASE+
.long BOOT          ; 0         00
.long DUMMY         ; 1         04
.long NMI           ; 2         08
.long DUMMY         ; 3         0C
.long DUMMY         ; 4         10
     .
     .
     .
.long DUMMY         ; 29        74
.long DUMMY         ; 30        78
.long DUMMY         ; 31        7C


;--- Non Maskable Interrupt function ---
NMI:
jpr -1


;--- Dummy Interrupt function ---
DUMMY:
jpr -1
```

rodata セクション宣言

Global 宣言

①

アセンブラでのベクタテーブルの作成は.rodataセクションの宣言後、定数(①)を配置します。これによりベクタテーブルを作成することが出来ます。

詳細は S5U1C17001C Manual：セクションとリンク
S5U1C17001C Manual：データセクション定擬似命令(.rodata、.data)
S1C17701 テクニカルマニュアル：ベクタテーブル
を参照して下さい。

2.2.1 ベクタテーブルとは

ベクタテーブルは、プログラム実行中に割り込みが発生した場合に実行する割り込み処理ルーチンへのベクタ（分岐先アドレス）を配列として格納したテーブルです。

表 2.1　ベクタテーブルの構成

ベクタ No.／ソフトウェア割り込み No.	割り込み	ベクタアドレス
0（0x00）	リセット	TTBR＋0x00
1（0x01）	アドレス不整割り込み	TTBR＋0x04
2（0x02）	NMI	TTBR＋0x08
3（0x03）	マスク可能な割り込み 3	TTBR＋0x0c
:	:	:
31（0x1f）	マスク可能な割り込み 31	TTBR＋0x7c

表 2.1 のベクタアドレス欄に示した「TTBR（トラップテーブルベースレジスタ）」はベクタテーブルの先頭のアドレスを表しています。
※TTBR 値については機種により異なってきますので、各機種のテクニカルマニュアルを参照してください。

2.3 割り込みについて

S1C17 コアでは 32 種類の割り込みを受けることができます。(最初の 3 つはリセット、アドレス不整、NMI に使用されています。)
割り込み処理ルーチンは、それぞれの割り込み要因によって受け付けて呼ばれるルーチンです。それぞれに適した処理を書いてください。割り込み要因と設定については機種により異なってきますので、各機種のテクニカルマニュアルを参照してください。

2.3.1 リセット

パワーオン時にリセット割り込みが発生します。リセット処理ではベクタテーブルの先頭からリセットベクタが呼び出され、PCにセットされます。これにより、リセットベクタのスタートアップルーチンに分岐してプログラムが実行されます。

2.3.2 reti 命令

reti命令は、割り込み処理ルーチン用のリターン命令です。割り込み処理ではリターンアドレスと共にPSRもスタックにセーブされますのでreti命令によってPSRの内容を復帰させる必要があります。reti命令ではPC,PSRの順にスタックから読み出されます。
割り込み処理ルーチンではプログラムの最後に必ず reti 命令を実行してください。それにより PC を割り込みが発生した位置に戻すと同時に、PSR の値もスタックから戻し中断していた命令列に処理を戻せます。

2.3.3 アドレス不整

メモリやI/O領域をアクセスするロード命令は、命令により転送するデータサイズが決まってきます。そのアドレスはデータサイズごとの境界線でなければなりません。

表 2.2　ロード命令とアドレス境界

命令	転送データサイズ	アドレス
ld.b / ld.ub	バイト(8 ビット)	バイト境界（全アドレスが対象）
ld	16 ビット	16 ビット境界（アドレスの最下位ビットが 0）
ld.a	32 ビット	32 ビット境界（アドレスの下位 2 ビットが 00）

ロード命令の指定アドレスがこの条件を満たしていない場合、プロセッサはアドレス不整割り込みとして割り込み処理に移行します。
リスト2.1では、アドレス不整割り込みが発生したら「dummy関数」にジャンプして永久ループ処理をしていますが、適意に変更して使用してください。

2.3.4 NMI

割り込みにはマスク可能な割り込みとマスク不可能な割り込みがあります。
マスク不可能な割り込みをNMI (Non-Maskable Interrupt) といいます。NMIは他の割り込みに優先して、無条件にCPUに受け付けられます。
リスト 2.1 では、NMI 割り込みが発生したら「NMI」にジャンプして無限ループしますが、適意に変更して使用してください。

2.4 スタートアップルーチンの作成

リスト 2.2　スタートアップルーチンの例

```
;--- Boot function  ---
.text
.align 1
BOOT:
xld.a    %sp,0x0fc0                              ;SPの設定

xcall    clearBss                                ;clearBss呼び出し
xcall    copyLmaToVma                            ;copyLmaToVma呼び出し

ei                                               ;割り込み許可
xcall    main                                    ;mainルーチン呼び出し
xcall    end                                     ;endルーチン呼び出し

ret

;--- ClearBss function  ---
clearBss:
  *clearBssのプログラム内容は2.4.3  .bss/.dataセクションを参照して下さい。
        ret

;--- copyLmaToVma function ---
copyLmaToVma:
  *copyLmaToVmaのプログラム内容は2.4.3  .bss/.dataセクションを参照して下さい。
        ret
```

2.4.1 スタートアップルーチンとは

スタートアップルーチンは、パワーオン（イニシャルリセット）時にリセット割り込みが発生し、その割り込みに対応するベクタテーブルから呼び出された関数のことを示します。

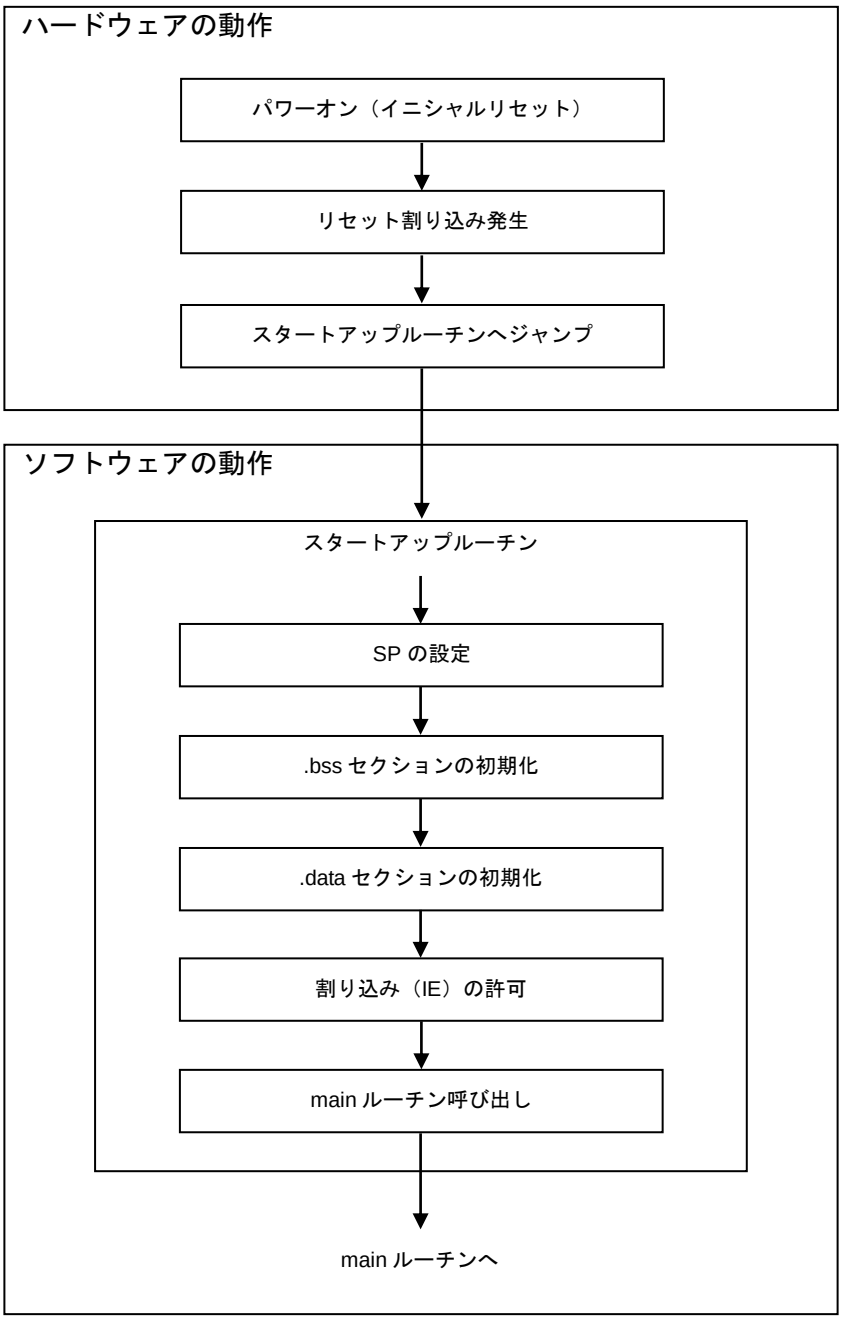


図 2.1　スタートアップルーチンの実行フロー

2.4.2 SP の設定

最初にアセンブラの ld 命令を使用して、SP（スタックポインタ）にスタックの先頭アドレスを設定します。

リスト 2.3　SP 設定の例

```
xld.a    %sp, 0x0fc0          ; Set SP in RAM
```

リスト 2.3 では 0x0fc0 を設定していますが任意の RAM 領域のアドレスを設定しても構いません。スタックの領域と RAM データが占領する領域が重複しないように注意して設定してください。
リスト 2.3 でアドレス 0x0fc0 を設定している説明をします。
S1C17 シリーズの CPU は、若いアドレス方向にスタックを積んでいきます。S1C17701 では RAM 領域が 0x0000～0x1000 なので、SP に設定可能な最大値は 0x1000 ですが、0x0fc0～0x0fff はオンチップデバッガ用に予約されています。従って、領域が重複しないように 0x0fc0 を設定しています。
※RAM 領域、およびオンチップデバッガ用領域のアドレスについては、各機種のテクニカルマニュアルを参照してください。

参考に、スタックへレジスタを退避する場合と SP の移行について下記の図に示しました
例：ld.a　　- [%SP], %r0
命令文の説明：スタックポインタを 4 バイト分ディクリメントした後、R0 レジスタの 24 ビットデータをそのアドレスに転送します。メモリには上位 8 ビットを 0 とした 32 ビットデータを書き込まれます。
(1) SP＝SP－4
(2) R0→[SP]

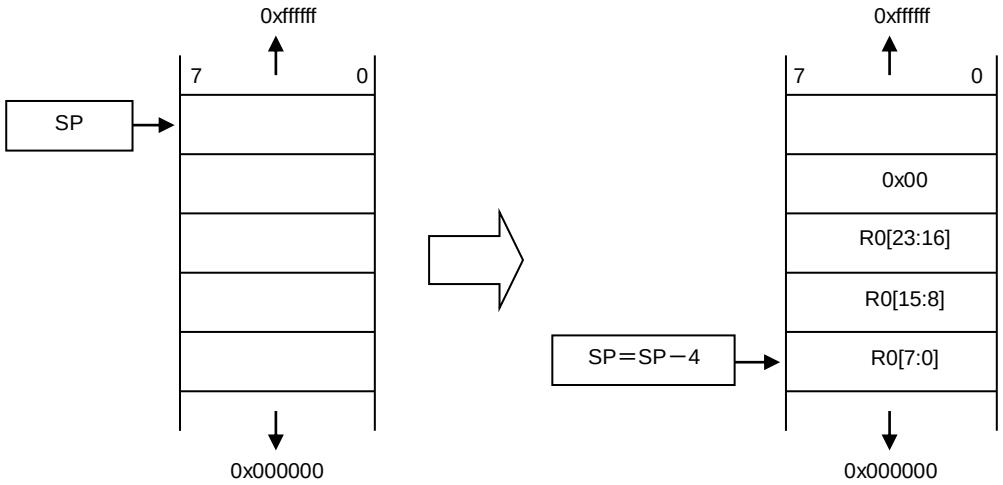


図 2.2　SP とスタック

2.4.3 .bss/.data セクションの初期化

まず「.bss/.data セクション」の初期化を説明する前に、GNU17 で作成したプロジェクトのメモリ構成を説明します。図 2.3 に S1C17701 のメモリ構成を表記します。

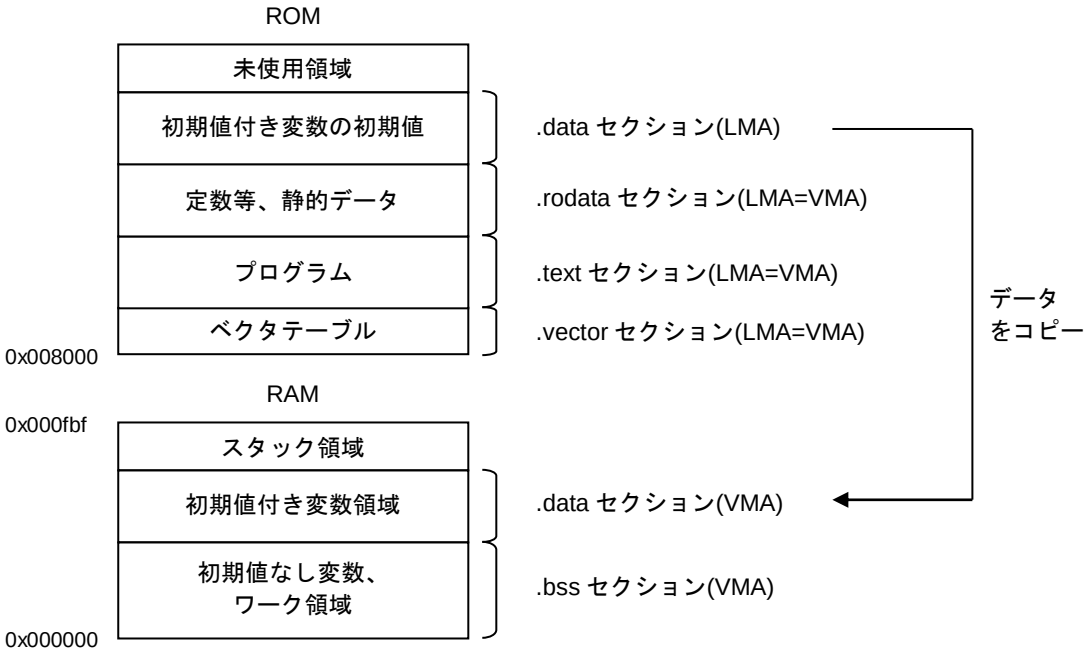


図 2.3　メモリ構成例（S1C17701）

アドレス0x8000から配置されているROMに、プログラムとデータを図2.3のように配置します。プログラムはROM上の格納アドレス (LMA) でそのまま実行するものとし、静的データもROMから直接読み出して使用するものとします。
RAMには初期値を持たない変数領域をアドレス0x0から配置し、その後初期値を持つ変数領域として使用します。変数の初期値はROMに格納しておき、アプリケーションプログラムがRAMにコピーします。
セクションについての詳しい説明は「S5U1C17001C MANUAL」を参照してください。

「.bss セクション」の初期化について説明します。
「.bss セクション」は、初期値を持たない変数が配置されるセクションです。
「__START_bss」から「__END_bss」までの領域に「0」をセットしてデータをクリアしています。

リスト 2.4　.bss セクションの初期化

```
;--- ClearBss function  ---

clearBss:

xld.a    %r0,__START_bss
xld.a    %r1,__END_bss
cmp.a    %r0,%r1
jreq     clearBss_End

ld       %r2,0x0

clearBss_10:
ld.b     [%r0]+,%r2
cmp.a    %r1,%r0
jrne     clearBss_10

clearBss_End:
ret
```

START_bss, END_bss を各レジスタに格納。
r0,r1 を比較し START_bss, END_bss が一致した場合 clear_End にジャンプする

START_bss～ END_bss 領域に 0 をセットしデータクリアする

「__START_bss」と「__END_bss」は「Linker script file(file.lds)」で定義されています。
__START_bss　・・・・　.bssセクションの先頭のアドレス
__END_bss　・・・・・　.bssセクションの終了のアドレス

「.dataセクション」の初期化について説明します。
「.data セクション」は、初期値を持つ変数が配置されるセクションです。RAM 内 (LMA) 「__START_data_lma」からのデータを、ROM 内 (VMA) 「__START_data」から「__END_data」までの領域にコピーしています。

リスト 2.5　.data セクションの初期化

```
;--- copyLmaToVma function ---
copyLmaToVma:

xld.a     %r0,__START_data
xld.a     %r1,__START_data_lma
xld.a     %r2,__END_data
sub.a     %r2,%r0
jreq      copyLmaToVma_End

copyLmaToVma_10:
ld.b      %r3,[%r1]+
ld.b      [%r0]+,%r3
sub       %r2,0x01
jrne      copyLmaToVma_10

copyLmaToVma_End:
ret
```

START_data START_data_lma END_data をレジスタに格納し START_data END_data を減算することで初期値もち変数が無かった場合 copyLmaToVma_End にジャンプする。

START_data_lma のデータを START_data ～ END_data までの領域にコピーしています。
sub %r2,0x01 で 0 フラグが立った時ループから抜けます。

「__START_data」と「__START_data_lma」と「__END_data」は「Linker script file(file.lds)」で定義されています。

__START_data　・・・・・　.data セクションの先頭のアドレス
__START_data_lma　・・・　.data セクション LMA 部の先頭のアドレス
__END_data　・・・・・・　.data セクションの終了のアドレス

2.4.4 割り込み (IE) の許可

ei 命令を使用して、PSR（プロセッサステータスレジスタ）の IE ビット（割り込み許可）を「1」にセットし、マスク可能な外部割込みを許可しています。

リスト 2.6　割り込み (IE) 許可

```
ei      ; interrupt enable
```

PSR は CPU の状態を保持する 8 ビットレジスタで、実行した命令の結果によって変化します。PSR の内容は、IE ビット以外をプログラムで直接変更することはできません。

	7	6	5	4	3	2	1	0
PSR	IL [2:0]			IE	C	V	Z	N
初期値	0	0	0	0	0	0	0	0

IL：　割り込みレベル　　(0〜7：割り込み)
IE：　割り込み許可　　(1：許可、0：禁止)
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)
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)

図 2.4　PSR

参考に、マスク可能な外部割込みを禁止する場合は、アセンブラのdi命令を使用してリスト2.7のように記述します。

リスト 2.7　割り込み (IE) 禁止

```
di      ; interrupt disable
```

3. C、アセンブラのインタフェース

本章ではアセンブラ、C 言語を混在させたプログラムの書き方について説明します。

3.1 アセンブラ、C の混在法

引数、戻り値、レジスタ内容保護などルールを守ることにより C、アセンブラルーチン間を自由に行き来出来ます。
GNU17 が C ソースをコンパイルするとき、各レジスタは下記の用途で使用されます。引数、戻り値が何処に格納されるか説明しています。

表 3.1　汎用レジスタの使用法

レジスタ	使用法
R0	引数渡し用レジスタ(第 1 ワード)　スクラッチレジスタ 戻り値格納用レジスタ(8/16 ビットデータ、ポインタ、32 ビットデータの下位 16 ビット)
R1	引数渡し用レジスタ(第 2 ワード) スクラッチレジスタ 戻り値格納用レジスタ(32 ビットデータの上位 16 ビット)
R2	引数渡し用レジスタ(第 3 ワード) スクラッチレジスタ
R3	引数渡し用レジスタ(第 4 ワード) スクラッチレジスタ
R4	関数呼び出し前後で値が保障されるレジスタ
R5	
R6	
R7	

汎用レジスタの使用は表 3.1 に則って使用して下さい。

※スクラッチレジスタ
スクラッチレジスタは関数呼び出し前後で値が保障されないレジスタです。

詳細は S5U1C17001C Manual：レジスタ使用法を参照してください。

3.1.1 C 言語→アセンブラ時の関数呼び出し

C 言語で記述されたプログラムにアセンブラで作成したルーチンを呼び出す方法について説明します。

リスト 3.1　C プログラム　strcpy ルーチン呼び出し

```
/* #include */
#include <string.h>
int main(void){
    char pchMem[15];
    strcpy(pchMem,"strcpy test");      //strcpyルーチン呼び出し
    return 0;
}
```

リスト 3.1 のプログラム中で strcpy ルーチンを呼び出しています。strcpy ルーチンは文字型配列に文字列をコピーします。呼び出された strcpy ルーチンは第一引数“pchMem”、第二引数“strcpy test”それぞれのポインタを汎用レジスタ R0,R1 にそれぞれ格納します。戻り値は pchMem の先頭アドレスで R0 レジスタに格納されます。リスト 3.1 のプログラムからリスト 3.2 のプログラムを呼び出しています。

リスト 3.2　アセンブラプログラム　strcpy ルーチン

```
;--- strcpy program  ---
.section .text
.align 1
.global strcpy

strcpy:
ld.a     %r3,%r0
strcpy_loop:
ld.b     %r2,[%r1]+
ld.b     [%r3]+,%r2
cmp      %r2,0
jrne     strcpy_loop

ret
```

第一引数は%r0、第二引数は%r1 に格納され、第二引数が NULL 文字までインクリメントされるまで文字列を文字型配列にコピーしています。

上記の様なプログラムで C プログラム中でもアセンブラルーチンを呼び出せます。
R4～R7 レジスタを使用する必要がある際は必ずスタックに退避させてから使用して下さい。

詳細は S5U1C17001C Manual：レジスタ使用法を参照して下さい。

3.1.2 アセンブラ→C 言語時の関数呼び出し

アセンブラで記述されたプログラム中に C 言語で作成した関数を呼び出す方法について説明します。3.1 アセンブラ、C の混在法のルールに従いプログラムを作成して下さい。

●アセンブラ→C 言語時の関数呼び出し（引数、戻り値をスタックで渡さない場合）

アセンブラプログラム中に C 関数を呼び出すには「xcall」を使用します。
呼び出された関数は R0,R1,R2,R3 レジスタに格納されているデータを引数とし処理を行います。C 関数を呼び出す前に必要な引数は各レジスタに格納しておいて下さい。このとき引数を格納するレジスタが足りない場合はスタックに格納しますので注意してください。C 関数の呼び出し処理が終わると戻り値は R0,R1 レジスタに格納されます。各レジスタの格納データ容量は 16 ビットです。それを超えるデータ容量の場合はスタックに格納しますので注意して下さい。
下記は簡単なプログラムを例に関数を呼び出すまでの流れを説明しています。

リスト 3.3　C プログラム　addi 関数

```
;--- ter program---
short
addi (unsigned short a, unsigned short b){
   return short (a + b) ;
   }
```

リスト 3.3 を呼び出すにはリスト 3.4 のようにプログラムします。

リスト 3.4　アセンブラプログラム　addi 関数呼び出し

```
;---text section---
.section .text
.align 1
.global main
main:
ld      %r0, 0x1                        ;%r0(第一引数)に0x1格納
ld      %r1, 0x2                        ;%r1(第二引数)に0x2格納
xcall   addi                            ;C関数呼び出し addi関数
main_loop:
cmp     %r0, 0x3                        ;戻り値(%r0)と0x3を比較
jrne    main_loop                       ;差異があればmain_loopへ
ret
```

●アセンブラ→C 言語時の関数呼び出し（引数、戻り値をスタックで渡す場合）

アセンブラルーチンから C 関数を呼び出す時、引数、戻り値をスタックで渡す場合の一例を説明します。
リスト 3.5 の関数は第一引数 64 ビット、第二引数 16 ビット、戻り値 64 ビットです。

リスト 3.5　C プログラム　addi 関数

```
;--- addi program---
long long
addi (unsigned long long a, unsigned short b){
    return (long long) (a + b) ;
}
```

呼び出し元アセンブラルーチンはリスト 3.6 です。

リスト 3.6　アセンブラプログラム　addi 関数呼び出し　引数、戻り値 64 ビット

```
;---text section---
.section .text
.align 1
.global main
main:
ld.a    [%sp]-,%r4
xld     %r0,0xffff
xld     %r1,0xffff
xld     %r2,0xffff
xld     %r3,0xffff
ld      %r4,0x1
sub.a   %sp,0xc
ld      [%sp+0x8],%r4
ld      [%sp+0x6],%r3
ld      [%sp+0x4],%r2
ld      [%sp+0x2],%r1
ld      [%sp+0x0],%r0
ld.a    %r0,%sp
xcall   addi
ld      %r0,[%sp+0x0]
ld      %r1,[%sp+0x2]
ld      %r2,[%sp+0x4]
ld      %r3,[%sp+0x6]
add.a   %sp,0xc
ld.a    %r4,[%sp]+
ret
```

r0～r3 レジスタに第一引数用のデータを、r4 レジスタに第二引数用のデータを格納します。

sub.a でスタックに必要データ分の領域を確保します。
確保したスタック領域に引数を格納していきます。
戻り値は 32 ビットデータを超える為スタックに格納されます。格納先アドレスを r0 レジスタに格納し関数に渡します。

戻り値は r0 で指定したアドレスに格納されていますので、そこから戻り値を取得します。
最後に確保したスタック領域を開放します。

関数呼び出し時に引数が 64 ビット以上の場合、スタックに引数を格納してから受け渡します。引数の 10 バイト分のスタック領域を sub.a **%sp**,0xc で確保しています。SP で指定できるアドレスは 32 ビット境界となりますので注意して下さい。
引数の受け渡しの詳細については S5U1C17001C Manual：レジスタ使用法を参照して下さい。
戻り値が 32 ビットを超える時は R0 レジスタ に戻り値の格納先アドレスを指定してください。
確保したスタック領域は使用が終わったら必ず開放して下さい。

4. アセンブラプログラム作成時の注意点

この章ではアセンブラプログラムを作成する時の注意点を記述します。

4.1 ext の使用法

16 ビット固定長の命令コードで指定できる即値は、命令によって異なりますが 7 ビットまたは 10 ビットのビットフィールドで指定します。ext 命令はこの即値のサイズを拡張するために使用します。
ext 命令はデータ転送命令、演算命令、分岐命令と組み合わせて使用し、即値の拡張を行いたい命令の直前に置きます。命令は「ext imm13」(immX は符号なし X ビット即値です)の形式で記述します。1 個の ext 命令で拡張可能な即値サイズは 13 ビットで、更に即値拡張するために 2 個まで ext 命令を続けて記述することが出来ます。
ext 命令が有効となるのは直後に記述された即値拡張が可能な命令に対してのみで、その他の命令に対しては無効です。3 個以上の ext 命令が連続的に記述された場合は最後の 2 個が有効となり、それ以外は無視されます。
ext 命令の直後の命令が ext による拡張命令に対応していない場合、その ext 命令は nop 命令として実行されます。
次ページは即値アドレッシングの拡張の一例です。

●即値を 16 ビット、20 ビット、24 ビット即値に拡張します

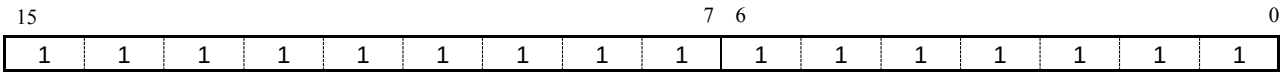
add は 16 ビット加算命令です。7 ビット即値をゼロ拡張して Rd レジスタに加えます。
即値は符号なし 7 ビットです。

・16 ビット即値への拡張
即値を 16 ビットに拡張するには、1 個の ext 命令を対象命令の前に置きます。

リスト 4.1　16 ビット拡張例

```
ext     0x1ff
add     %rd,0x7f                    ;=xadd    %rd, 0xffff
```

拡張された即値

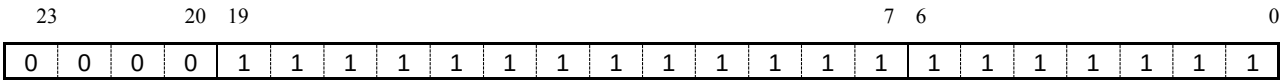


・20 ビット即値への拡張
即値を 20 ビットに拡張するには、1 個の ext 命令を対象命令の前に置きます。

リスト 4.2　20 ビット拡張例

```
ext     0x1fff
add.a   %rd,0x7f                    ;=xadd.a  %rd, 0xfffff
```

拡張された即値



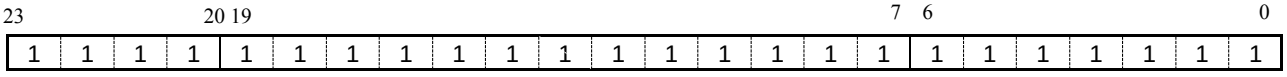
ビット 23～20 までは 0 で埋まります。(ゼロ拡張)

・24 ビット即値への拡張
即値を 24 ビットに拡張するには、2 個の ext 命令を対象命令の前に置きます。

リスト 4.3　24 ビット拡張例

```
ext     0xf
ext     0x1fff
add.a   %rd,0x7f                    ;=xadd.a  %rd, 0xffffff
```

拡張された即値



詳細は S1C17 コアマニュアル：ext 付きアドレッシングモード を参照して下さい。

4.2 拡張命令

アセンブラ as は以下に説明する拡張命令に対応しています。
拡張命令は、通常 ext 命令を含む複数の命令で記述する内容を 1 つの命令をして記述出来るようにしたもので、命令機能及びオペランドの即値サイズによって必要最小限の基本命令に展開されます。
下記でスタック～レジスタ間データ転送命令を使用して説明します。

・説明で使用するシンボル
immX　　符号なし X ビット即値
(x,y)　　ビット X からビット Y のビットフィールド

表 4.1　拡張命令の種類と機能

拡張命令	機能	展開形式
sld.b %rd,[%sp+imm20]	%rb ← B[%sp+imm20](符号拡張)	(1)
sld.ub %rd,[%sp+imm20]	%rb ← B[%sp+imm20](ゼロ拡張)	(1)
sld %rd,[%sp+imm20]	%rb ← W[%sp+imm20]	(1)
sld.a %rd,[%sp+imm20]	%rb ← B[%sp+imm20](23:0),無視←A[%sp+imm20](31:24)	(1)
sld.b [%sp+imm20],%rs	B[%sp+imm20]←%rs(7:0)	(1)
sld [%sp+imm20],%rs	W[%sp+imm20]←%rs(15:0)	(1)
sld.a [%sp+imm24],%rs	A[%sp+imm20](23:0)←%rs(23:0),A[%sp+imm20](31:24)←0	(1)
xld.b %rd,[%sp+imm24]	%rb ← B[%sp+imm24](符号拡張)	(2)
xld.ub %rd,[%sp+imm24]	%rb ← B[%sp+imm24](ゼロ拡張)	(2)
xld %rd,[%sp+imm24]	%rb ← W[%sp+imm24]	(2)
xld.a %rd,[%sp+imm24]	%rb ← B[%sp+imm24](23:0),無視←A[%sp+imm24](31:24)	(2)
xld.b [%sp+imm24],%rs	B[%sp+imm24]←%rs(7:0)	(2)
xld [%sp+imm24],%rs	W[%sp+imm24]←%rs(15:0)	(2)
xld.a [%sp+imm24],%rs	A[%sp+imm24](23:0)←%rs(23:0),A[%sp+imm24](31:24)←0	(2)

※上記表のアルファベットは以下に対応します。
B（バイト）→ 8 ビット
W（ワード）→ 16 ビット
A（アドレスデータ）→ 32 ビット（ただし上位 8 ビットは 0 が書き込まれます）

アドレスデータについては S5U1C17001C Manual：データ形式を参照して下さい。

・展開後の基本命令

sld.b	xld.b	ld.b 命令に展開
sld.ub	xld.ub	ld.ub 命令に展開
sld	xld	ld 命令に展開
sld.a	xld.a	ld.a 命令に展開

・展開形式

imm20,imm24 を省略した場合、[%sp+0x0]を指定したものとして展開されます。

(1) sld.a %rd, [%sp+imm20]
sld.a [%sp+imm20], %rs

例：sld.a %rd,[%sp+imm20]

imm20≦0x7f	**0x7f<imm20**
ld.a %rd,[%sp+imm20(6:0)]	ext imm20(19:7) ld.a %rd,[%sp+imm20(6:0)]

(2) xld.a %rd, [%sp+imm24]
xld.a [%sp+imm24], %rs

例：xld.a %rd,[%sp+imm24]

imm24≦0x7f	**0x7f<imm24≦0xfffff**	**0xfffff<imm24**
ld.a %rd,[%sp+imm24(6:0)]	ext imm24(19:7) ld.a %rd,[%sp+imm24(6:0)]	ext imm24(23:20) ext imm24(19:7) ld.a %rd,[%sp+imm24(6:0)]

詳細は S5U1C17001C Manual：拡張命令を参照して下さい。

4.3 メモリモデルについて

開発するアプリケーションシステムのプロセッサの種類及びメモリ空間のサイズにより、ツールの起動コマンドオプションやリンクするライブラリを切り替えます。そのために適切なメモリモデルを選択してください。
メモリモデルは新規プロジェクト作成時に設定します。後からの変更も可能です。

メモリモデルは REGULAR／MIDDLE／SMALL の 3 種類あります。
SMALL→MIDDLE→REGULAR になるにつれてアドレス空間は大きくなりますが、コード効率が悪くなっていきますので適切なメモリアドレスを選択して下さい。

表 4.2　メモリモデルとアドレスのサイズ

メモリモデル	アドレスサイズ	アドレス空間
REGULAR	24 ビット	16M バイト
MIDDLE	20 ビット	1M バイト
SMALL	16 ビット	64K バイト

改定履歴表

付－1

Rev. No.	日付	ページ	種別	改訂内容（旧内容を含む） および改訂理由
Rev. 1.0	2008/09/16	全ページ	新規	新規制定
Rev. 1.1	2017/12/13	P11, 12	誤記 修正	リスト 2.4 と 2.5 内の jpeq を ireq に修正

セイコーエプソン株式会社

営業本部　デバイス営業部

東京　〒191-8501　東京都日野市日野 421-8
TEL（042）587-5313（直通）　FAX（042）587-5116

大阪　〒541-0059　大阪市中央区博労町 3-5-1　御堂筋グランタワー15F
TEL（06）6120-6000（代表）　FAX（06）6120-6100

ドキュメントコード：411574501
2008 年　9 月 作成
2017 年 12 月 改訂 Ⓛ