EPSON
EXCEED YOUR VISION

CMOS 16-BIT SINGLE CHIP MICROCOMPUTER

S1C17 Family

スタートアップマニュアル
C言語版

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。

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工業所有権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. 本資料に掲載されている製品のうち「外国為替及び外国貿易法」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。

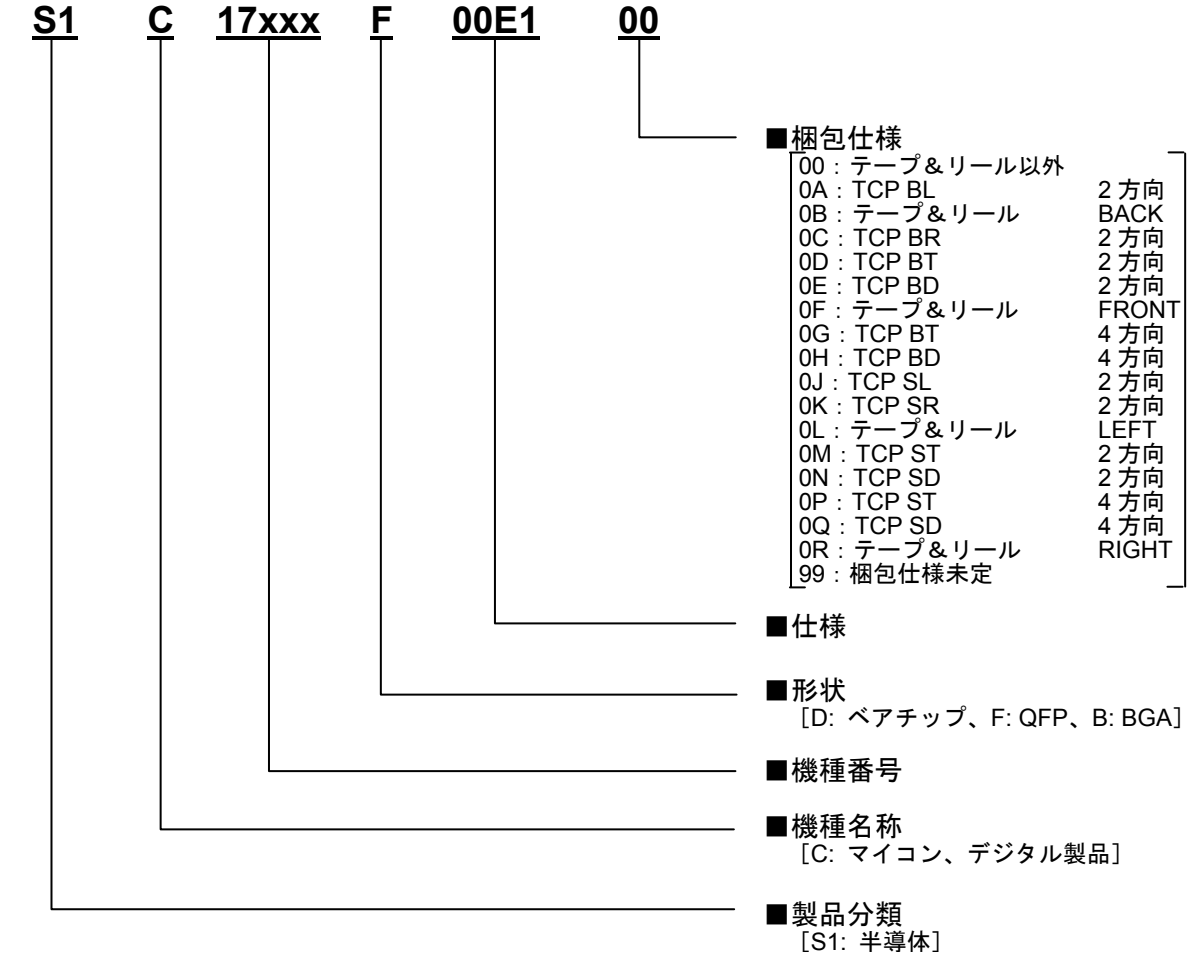
5. 本資料に掲載されている製品は、生命維持装置その他、きわめて高い信頼性が要求される用途を前提としていません。よって、弊社は本（当該）製品をこれらの用途に用いた場合のいかなる責任についても負いかねます。

Windows2000 および WindowsXP は米国マイクロソフト社の登録商標です。
PC/AT および IBM は米国 International Business Machines 社の登録商標です。
その他のブランド名または製品名は、それらの所有者の商標もしくは登録商標です。

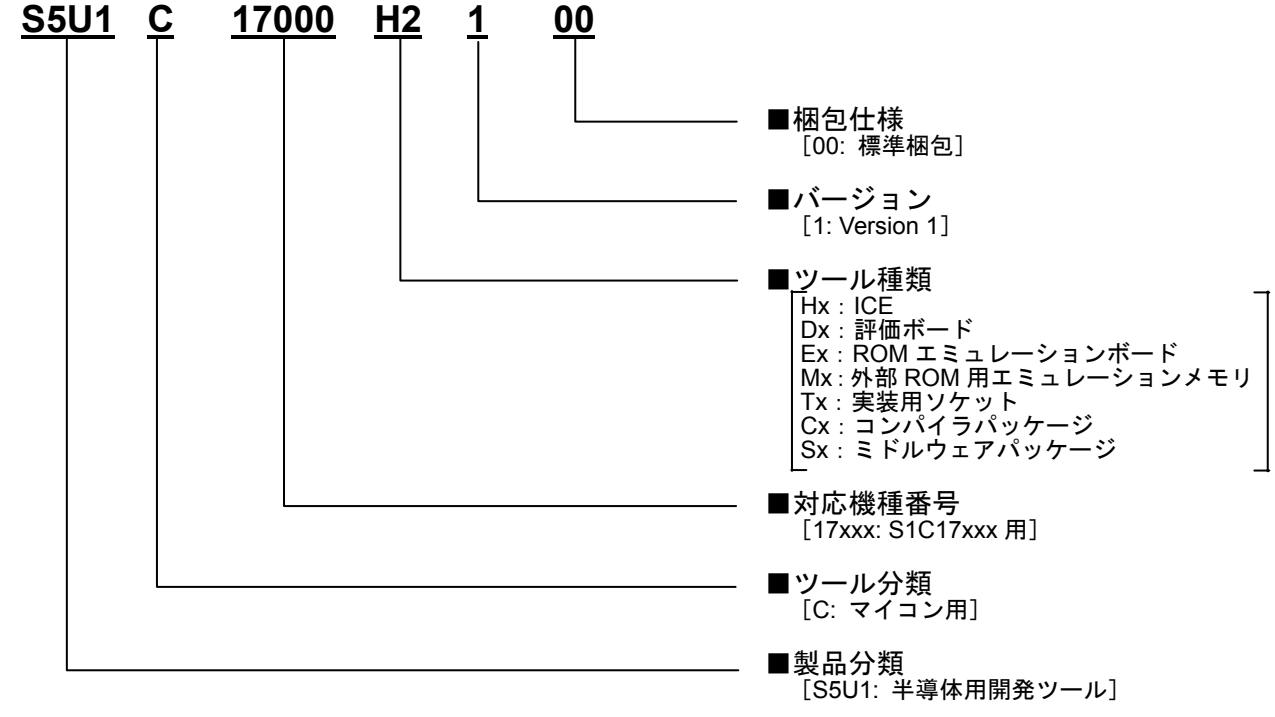
©SEIKO EPSON CORPORATION 2008, All rights reserved.

製品型番体系

●デバイス



●開発ツール



ーはじめにー

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。

- C 言語（ANSI C 準拠）に関する知識および C ソースプログラムの作成方法
- アセンブリ言語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、volatile 修飾子について、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、I/O レジスタへのアクセス方法について、サンプルプログラムを使用して説明しています。

＜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。

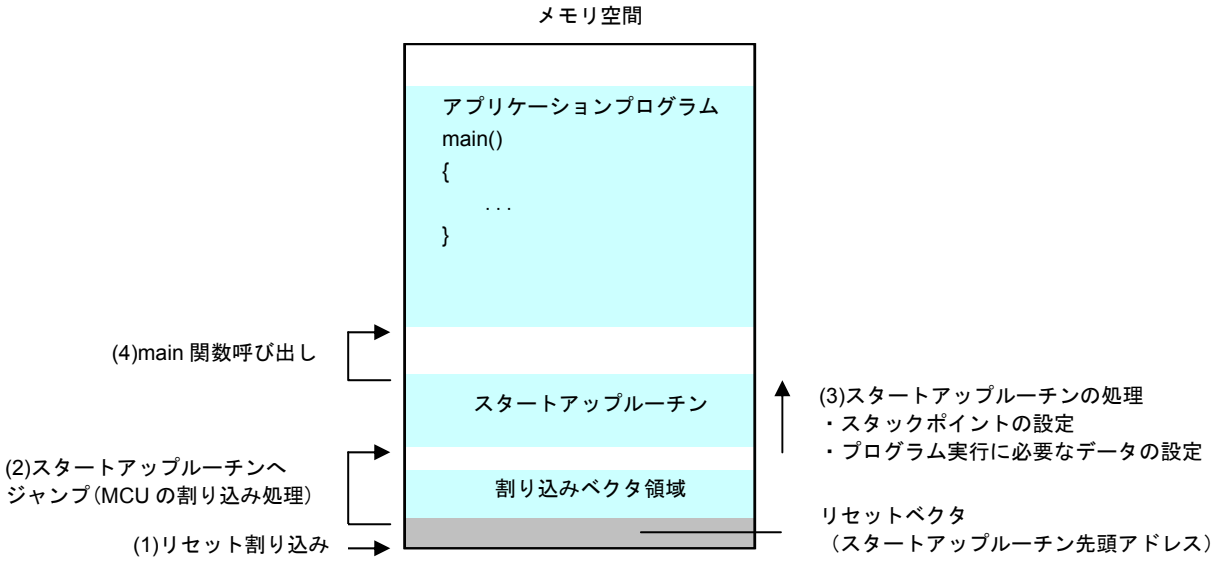


図 1.1　S1C17 プロセッサ起動時の基本メカニズム

(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 関数を呼び出します。

※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 関数からではなく、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

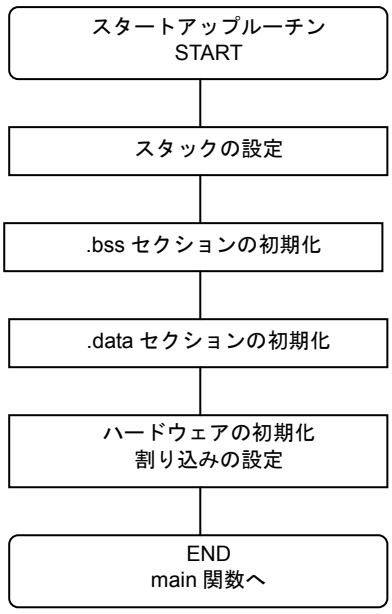


図 1.2　スタートアップルーチン

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 関数が呼び出されます。

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 関数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 関数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。

(1) 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。

(2) 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。

(3) 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。

(4) 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

```
/* Prototype */
void boot(void);
void dummy(void)__attribute__ ((interrupt_handler));
void nmi(void)__attribute__ ((interrupt_handler));

/* special definitions for vector table  */
typedef void func(void);
#ifdef __POINTER16
#define VECTOR(vec)  ((vec),0)
#else
#define VECTOR(vec)  (vec)
#endif

/* vector table */
func *const vector[] = {              // No     Base+
        VECTOR(boot),                 // 0      00
        VECTOR(dummy),                // 1      04
        VECTOR(nmi),                  // 2      08
        VECTOR(dummy),                // 3      0c
        VECTOR(dummy),                // 4      10
              .                         .      .
              .                         .      .
              .                         .      .
        VECTOR(dummy),                // 30     78
        VECTOR(dummy)                 // 31     7c
}

/* Dummy Interrupt function */
void dummy(void) {
        /* infinite loop */
        while(1) {
        }
}

/* Non Maskable Interrupt function */
void nmi(void) {
        /* Processing is undecided. */
}
```

① →

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。

2.2.2 ベクタテーブルとメモリモデルの関係について

GNU17 を使用してプロジェクトを作成する際にメモリモデルを選択してアドレス空間が設定されます。この設定によりポインタのサイズが異なるため、ベクタテーブルを作成する際は注意しなければなりません。

メモリモデルは REGULAR／MIDDLE／SMALL の 3 種類あります。
REGULAR はアドレスサイズが 24 ビットで 16M バイト空間を使用でき、C プログラムでのポインタのサイズは 32 ビットになります。
MIDDLE はアドレスサイズが 20 ビットで 1M バイト空間を使用でき、C プログラムでのポインタのサイズは 32 ビットになります。
SMALL はアドレスサイズが 16 ビットで 64K バイト空間を使用でき、C プログラムでのポインタのサイズは 16 ビットになります。
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。

表 2.2　メモリモデルとポインタのサイズ

メモリモデル	アドレスサイズ	アドレス空間	ポインタのサイズ
REGULAR	24 ビット	16M バイト	32 ビット
MIDDLE	20 ビット	1M バイト	32 ビット
SMALL	16 ビット	64K バイト	16 ビット

ハードウェアから見るとベクタテーブルは32ビット境界でアクセスされます。表2.2により、REGULAR・MIDDLEモデルではポインタサイズが32ビットなので問題ありませんが、SMALLモデルではポインタのサイズが16ビットなので注意しなければなりません。リスト2.1はメモリモデルがどのモデルを選択されていてもベクタテーブルが正しく実行できるような記述例になっています。

リスト2.1ベクタテーブルの例について説明をします。
①部分にある「__POINTER16」とは、メモリサイズがSMALLモデルの時にコンパイラが自動的に定義するマクロです。これによりSMALLモデルの場合は「#define VECTOR(vec) ((vec),0)」が、REGULAR・MIDDLEモデルの場合は「#define VECTOR(vec) (vec)」が反映される仕組みになっています。
次に、#defineによるマクロ定義によってベクタテーブルがどのように置換されるか説明します。実際にリスト2.1を「#define VECTOR(vec) ((vec),0)」と「#define VECTOR(vec) (vec)」での置換後を記述して説明します。

リスト 2.2　#define VECTOR(vec) ((vec), 0)の置換後

```
/* vector table */
func *const vector[] = {              // No     Base+
        ((boot),0),                   // 0      00
        ((dummy),0),                  // 1      04
        ((nmi),0),                    // 2      08
        ((dummy),0),                  // 3      0c
        ((dummy),0),                  // 4      10
             .                          .       .
             .                          .       .
             .                          .       .
        ((dummy),0),                  // 30     78
        ((dummy),0)                   // 31     7c
}
```

リスト 2.2 はベクタテーブルの各項目が 32 ビットになるように細工されています。16 ビットのポインタの後の 16 ビット領域に「0」をセットして 32 ビットを成立させています。ポインタを前半に記述しているのは、S1C17 シリーズの CPU はリトルエンディアン形式でアクセスされるからです。

リスト 2.3　#define VECTOR(vec) (vec)の置換後

```
/* vector table */
func *const vector[] = {              // No     Base+
        (boot),                       // 0      00
        (dummy),                      // 1      04
        (nmi),                        // 2      08
        (dummy),                      // 3      0c
        (dummy),                      // 4      10
            .                           .       .
            .                           .       .
            .                           .       .
        (dummy),                      // 30     78
        (dummy)                       // 31     7c
}
```

リスト 2.3 はポインタが 32 ビットになりますのでベクタテーブルは成立しています。

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 プロトタイプ宣言

割り込み処理ルーチンは次の形式でプロトタイプ宣言します。
＜型＞＜関数名＞__attribute__((interrupt_handler));

リスト2.4　プロトタイプ宣言の例

```
/* Prototype */
void dummy(void)__attribute__ ((interrupt_handler));
```

リスト 2.4 のようにプロトタイプ宣言することにより、割り込み処理終了時にハードウェアが退避してくれた PSR（プロセスステータスレジスタ）と PC（プログラムカウンタ）の内容を復帰する処理を自動的に組み込んでくれます。
また割り込み処理ルーチンを呼び出すには、ベクタテーブルの各ベクタ（分岐先アドレス）に各割り込み処理ルーチンを指定する必要があります。リスト 2.1 では、使用しない割り込みについては、「dummy 関数」にジャンプするように指定しています。

2.3.2 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。

2.3.3 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 2.3　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関数」にジャンプして自動復帰していますが、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 2.5　スタートアップルーチンの例

```
/* #include */
#include <string.h>

/* Prototype */
void boot(void);
void clearBss(void);
void copyLmaToVma(void);

/* extern */
extern int main(void);
extern unsigned char __START_bss;
extern unsigned char __END_bss;
extern unsigned char __START_data;
extern unsigned char __START_data_lma;
extern unsigned char __END_data;

/* Boot function  */
void boot(void) {
        asm("xld.a  %sp, 0x0fc0");     // Set SP in RAM
        clearBss();                    // clear the bss area
        copyLmaToVma();                // copy LMA data to VMA data
        asm("ei");                     // interrupt enable
        main();                        // Call main
}

/* ClearBss function  */
void clearBss(void) {
        memset(&__START_bss, 0, (size_t) (&__END_bss - &__START_bss));
}

/* copyLmaToVma function */
void copyLmaToVma(void) {
        memcpy(&__START_data, &__START_data_lma, (size_t) (&__END_data - &__START_data));
}
```

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。

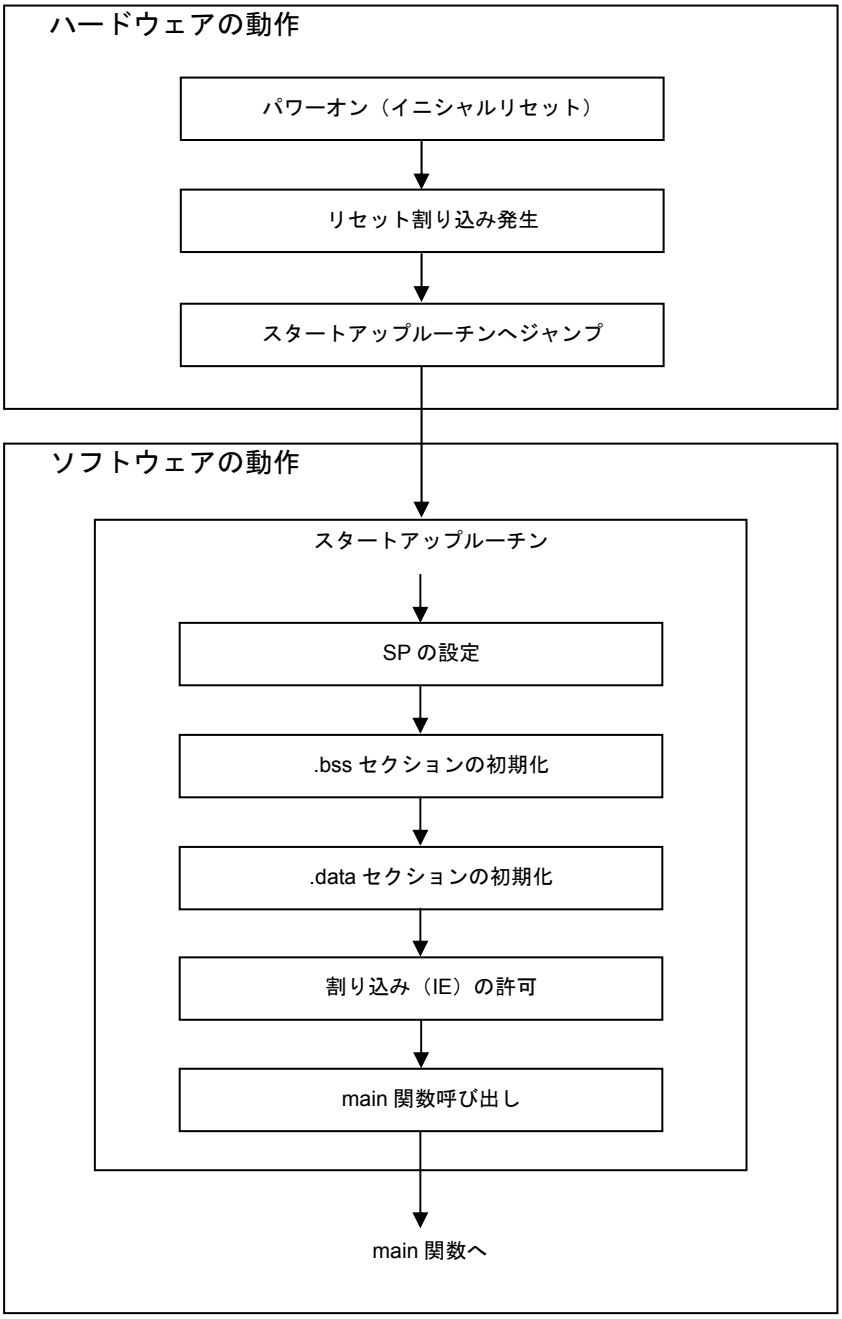


図 2.1　スタートアップルーチンの実行フロー

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 2.6　SP 設定の例

```
asm("xld.a  %sp, 0x0fc0");   // Set SP in RAM
```

リスト 2.6 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 2.6 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]

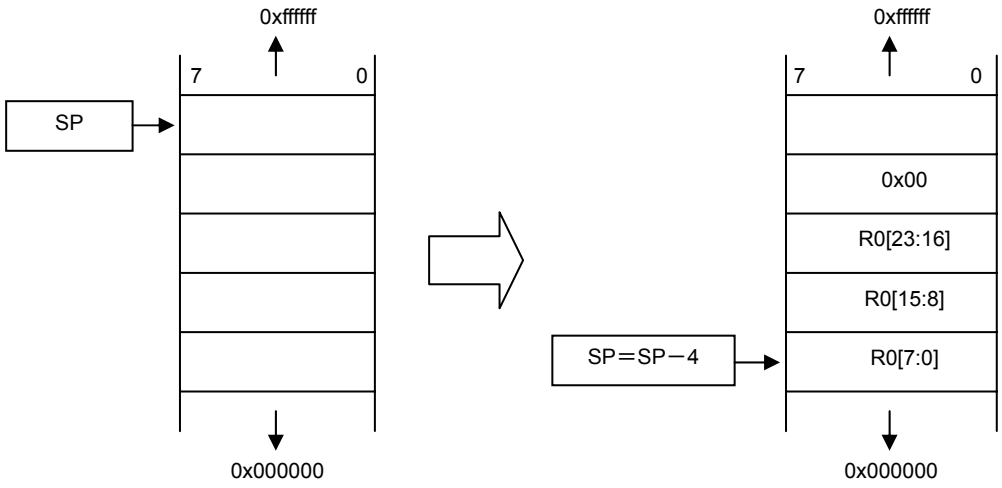


図 2.2　SP とスタック

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 2.4 に S1C17701 のメモリ構成を表記します。

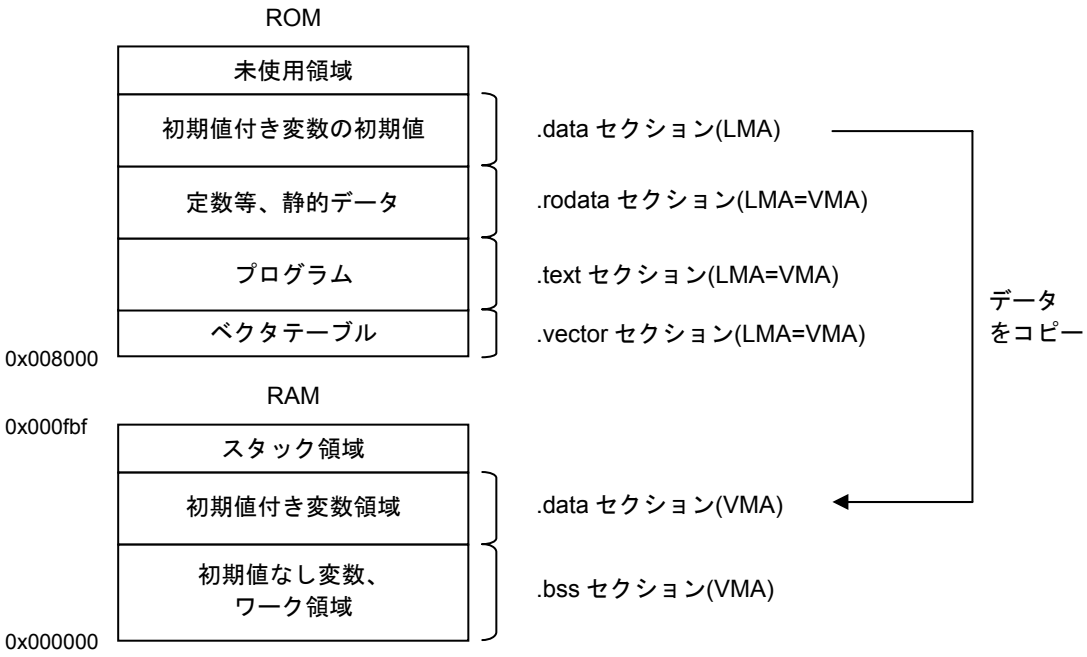


図 2.2　メモリ構成例（S1C17701）

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2.4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。

リスト 2.7　.bss セクションの初期化

```
/* #include */
#include <string.h>

/* extern */
extern unsigned char __START_bss;
extern unsigned char __END_bss;


/* ClearBss function  */
void clearBss (void) {
        memset(&__START_bss, 0 , (size_t)(&__END_bss - &__START_bss));
}
```

リスト 2.7 は、memset 関数を使用して「__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。

外部参照している「__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

※memsetを使用する際は「string.h」をインクルードします。

「.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）

リスト 2.8　.data セクションの初期化

```
/* #include */
#include <string.h>

/* extern */
extern unsigned char __START_data;
extern unsigned char __START_data_lma;
extern unsigned char __END_data;

/* copyLmaToVma function */
void copyLmaToVma(void) {
        memcpy(&__START_data, &__START_data_lma, (size_t) (&__END_data - &__START_data));
}
```

リスト 2.7 は、memcpy 関数を使用して「__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。

外部参照している「__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

※memcpy を使用する際は「string.h」をインクルードします。

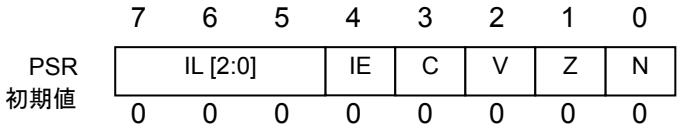
2.4.4 割り込み（IE）の許可

アセンブラの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 2.9　割り込み（IE）許可

```
asm("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。



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 2.3　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2.10のように記述します。

リスト 2.10　割り込み（IE）禁止

```
asm("di");          // interrupt disable
```

3. volatile 修飾

コンパイラとプログラムによる変数へのアクセスに影響を及ぼす修飾子として、volatile があります。一般的に C コンパイラはより高速で効率的なコードを作成するため、メモリへのアクセス回数を減らし、レジスタのロードした値をできる限り再利用して処理を行うことで最適化を図っています。ただし、この最適化によって、プログラムに記述されているメモリアクセスが実際には省略されてしまう可能性があります。この場合、ハードウェアによって変更された制御レジスタ値や、割り込み処理で変更された変数は、変更された値が反映されずに処理されてしまいます。volatile 修飾子は、変数が変更されている可能性があるものとして、変数の参照のたびにその内容を取得し直すことをコンパイラに指定します。つまり、volatile 宣言付きで定義した変数は、参照時に必ずメモリアクセスが発生し、最新の内容に更新された上で処理されます。
以下に、volatile 宣言の有無によるアセンブラコードを比較した例をリスト 3.1 で示します。

リスト 3.1　volatile 宣言の有無を比較するための例

```
/* prototype */
int main(void);
void sample_Int(void)__attribute__ ((interrupt_handler));

/* Global variable define*/
int Normal_flg;             // volatile no declaration
volatile int Volatile_flg;  // volatile declaration

int main(void){
     Normal_flg = 0;
    /* (1)volatile no declaration  */
    while(Normal_flg == 1 ){


      }


    Volatile_flg = 0;
    /* (2)volatile declaration  */
    while(Volatile_flg == 2 ){


      }

    return 0;
}

/* sample interrupt handler */
void sample_Int(void){
    Normal_flg = 1;
    Volatile_flg = 2;
    return;
}
```

（1）volatile宣言なし部分をアセンブラコードへの変換後の説明

```
ld    %r2,[0x0] ←  r2にアドレス0x0(Normal_flg)の値を転送
cmp   %r2,0x1   ←  r2と0x1を比較
jrne  0x7e      ←  上記の比較結果が等しくないと矢印の場所へ分岐する
```

割り込み処理ルーチン（sample_Int関数）で変数の値が変更されても実際のメモリを参照しないため、レジスタの値は更新されないままwhile文から抜ける事ができない

（2）volatile宣言あり部分をアセンブラコードへの変換後の説明

```
ld    %r2,[0x2] ←  r2にアドレス0x2(Volatile_flg)の値を代入
cmp   %r2,0x2   ←  r2と0x2を比較
jrne  0x7d      ←  上記の比較結果が等しくないと矢印の場所へ分岐する
```

割り込み処理ルーチン（sample_Int関数）で変数の値が変更されると、参照時に必ずメモリの値を確実に反映させるのでwhile文から抜ける事ができる

※リスト 3.1 は、最適化オプション「 - O1」を使用してコンパイルしています。

4. I/O レジスタへのアクセス方法

4.1 I/O レジスタへのリード/ライト

I/O レジスタから読み出し（リード）する場合は、リスト 3.2 のように記述します。

リスト 3.2　IO レジスタへのリード

```
(unsigned char ) i  = *(volatile unsigned char *) (  0x4020  )          ←8ビットデバイスの場合

(unsigned short ) j  = *(volatile unsigned short *) (  0x4200  )        ←16ビットデバイスの場合
```

リスト 3.2 では、変数「i」にアドレス 0x4020 の内容を 8 ビット転送し、「j」にアドレス 0x4200 の内容を 16 ビット転送しています。

I/O レジスタへ書き込み（ライト）する場合は、リスト 3.3 のように記述します。

リスト 3.3　IO レジスタへのライト

```
*(volatile unsigned char *) (  0x4020  )          = (unsigned char ) 0x01         ←8ビットデバイスの場合

*(volatile unsigned short *) (  0x4200  ) = (unsigned short ) 0x0001   ←16ビットデバイスの場合
```

リスト 3.3 では、アドレス 0x4020 に「0x01」を書き込み、アドレス 0x4200 に「0x0001」を書き込んでいます。

※I/O レジスタにアクセスする際は、周辺回路のデバイスサイズに注意してください。周辺回路のデバイスサイズは、各機種のテクニカルマニュアルを参照してください。

4.2 ビットフィールド

I/O レジスタはビットフィールドを使用することもできます。
I/O レジスタ（デバイスサイズ 8）にビットフィールドを使用してアクセスする例をリスト 3.4 に記述します。リスト 3.4 は、アドレス 0x4020 の 0 ビット目に「1」を書き込んでいます。

リスト 3.4　ビットフィールド使用例

```
/* Bit Field For Byte Data. */
struct BbitF {
        unsigned int b0     : 1;
        unsigned int b1     : 1;
        unsigned int b2     : 1;
        unsigned int b3     : 1;
        unsigned int b4     : 1;
        unsigned int b5     : 1;
        unsigned int b6     : 1;
        unsigned int b7     : 1;
};

/* Sample Register */
#define BF_SAMPLE           (*(volatile struct BbitF *)( 0x4020))

/* Sample Register writing */

BF_SAMPLE.b0 = 1;
```

⇨

```
アセンブラコードへの変換後の説明
xld.a    %r3,0x4020          ← r3に0x4020を転送
ld.b     %r2,[%r3]           ← r3（アドレス0x4020）の値をr2に転送
or       %r2,0x1             ← r2と0x1（0ビット目）の論理和をr2に転送
ld.b     [%r3],%r2           ← r3（アドレス0x4020）にr2に転送
```

ビットフィールドを使用して I/O レジスタにアクセスすると、バイトサイズでアクセスされますので注意してください。詳しい説明については、「S5U1C17001C MANUAL」内の「コンパイル出力」の「データ表現」の章を参照してください。

ビットフィールドを使用して「1」を書き込むことで「0」にリセットされる I/O レジスタに書き込みをしないでください。このような I/O レジスタについては、4.1 で説明した方法でアクセスしてください。
リスト 3.5 は、「1」を書き込むことで「0」にリセットされる I/O レジスタにビットフィールドを使用して書き込みをした具体例を記述します。

リスト 3.5　具体例

```
/* アドレス0x4020（「1」を書き込むことで「0」にリセットされるI/Oレジスタ）が0xaaの場合*/
xld.a    %r3,0x4020
ld.b     %r2,[%r3]           ← r2の値（01010101）
or       %r2,0x1             ← r2の値（01010101）
ld.b     [%r3],%r2           ← r3の値（01010101）⇒ r3 の値（00000000）
```

リスト 3.5 では、アドレス 0x4020 の 0 ビット目のみをリセットするつもりが、結果的に「1」がセットされているビットは、すべて「0」にリセットされてしまいます。

改定履歴

Date	revision	modification	note
08/08/18	1.0	新規作成	
08/09/18	1.1	表紙修正	

セイコーエプソン株式会社
半導体事業部　IC営業部

〈IC国内営業グループ〉
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　エプソン大阪ビル15F
TEL(06)6120-6000(代表)　FAX(06)6120-6100

インターネットによる電子デバイスのご紹介　http://www.epson.jp/device/semicon/

ドキュメントコード：411551101
2008年8月作成
2008年9月改訂