秋月通商製（AKI_H8）シリーズ用
共通基本 I/O処理（IOCS）仕様書

目　　次

[1] 概要：

AKI_H8において各種アプリ（ファームウェア）を開発するにあたり、いくつかの共通部分がある。　各種ファーム開発において都度、共通に出来る基本部分を、作り直すのでは効率が悪い。

それと、AKI_H8シリーズにおいてはいくつかの製品があり仕様が少しずつ異なる部分がある。　違いを整理すると

(1)　　ＣＰＵの違い
(2)　　ＣＰＵクロックの違い
(3)　　マザーボードの違い

となる。　これらの違いも吸収する形で、共通基本I/O処理（IOCS）の仕様を定義する。そして、IOCSを用いる事により、CPUの違い、クロックの違い、マザーボードの違いに極力影響を受けずにプログラム開発が行えるよう、開発効率の改善を図る事を IOCSの目的とする。

[2] AKI_H8シリーズの種類：

CPUボードは

(1)　　H8/3048F（ CPUクロック 16 [MHz] ）

(2)　　H8/3052F（ CPUクロック 25 [MHz] ）

(3)　　H8/3069F-LAN（ CPUクロック 20 [MHz] ）
（ ON Board LANなので基板形状が異なる。）

CPUの種類の違い

(1)　内部実装メモリ容量の違い
H8/3048F （ ROM : 128K , RAM : 4K ）
H8/3052F （ ROM : 512K , RAM : 8K ）
H8/3069F （ ROM : 512K , RAM : 16K ）

(2)　フラッシュROM 書き込み方法の違い
（ プログラムには影響しない。）

(3)　CPUクロックが異なるとシリアル通信のボーレイトに影響する。

マザーボードは

(1)　　ベーシックな マザーボード
DIPスイッチ、押しボタンスイッチ、発光ダイオード、液晶表示器の I/Oが付いたもの。（ CPUモード７で動作させる事を想定 ）

(2)　　AKI-H8-USB基板
DIPスイッチ、押しボタンスイッチ、発光ダイオード、液晶表示器に ＵＳＢのインタフェースが実装されたもの。
USBの周辺回路を接続するため、アドレスバス、データバスを出力させているため、ベーシックなI/Oの点数、及び I/O端子のピンアサインが、マザーボードとは異なる。

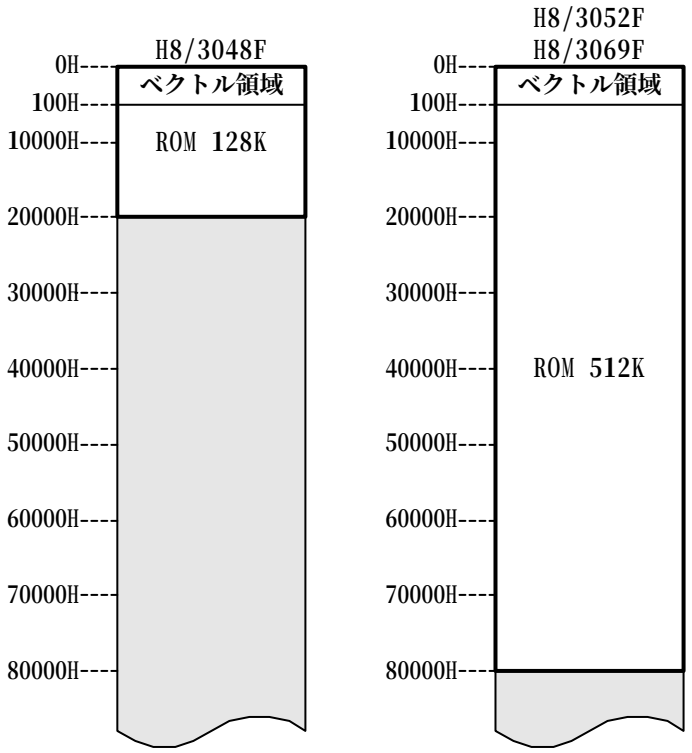
[3] メモリ容量の違いによる、プログラムへの影響：

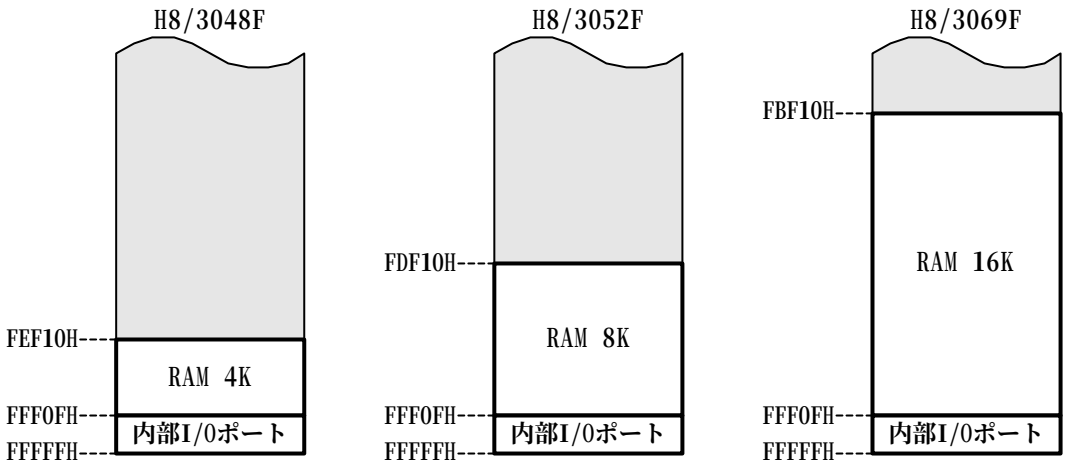
マイコンのメモリは、ROMと RAMに大別される。
RAMは、プログラムにて自由に読み書き可能であるが、ROMは、固定的なプログラムや定数のエリアであり同じメモリでも使い方が異なる。　また、また容量も制限がある。
H8シリーズでは、ROMに比べ、RAMの容量が小さいためこのあたりの配慮は必要となる。

パソコンの場合、全て RAMであり、またマイコンと比べると大量のメモリが実装されているため、ROM、RAM等の配置位置の考慮や、オブジェクトサイズを意識する必要もない。
マイコンのプログラムを作成する際に意識するのは、プログラムのどの部分を、ROMに配置するか、どの部分をRAMに配置するかである。　H8シリーズの場合、ROM容量が大きいため通常のファーム用途であれば、不足するという事はまずないと思われる。
RAMは、CPUにより、4K、8K、16Kなので、サイズ的には厳しいものがある。

ROMは、H8シリーズの、どのCPUにおいてもメモリ空間の先頭（ 0番地 ）から配置されているので、CPUの違いによる ROMの開始位置の違いはない。
ＣＰＵのアーキテクチャに関わる事として、ROMの先頭は割り込みベクトルエリアとして使用される。　通常　ROMエリアには、割り込みベクトル、プログラムのコード領域、定数、変数の初期値等が置かれる。
H8シリーズの RAMは、アドレス空間の最後に配置されるため、RAMの最終アドレスは、同じとなるが、開始アドレスは、RAMのサイズにより変わることになる。
またＣＰＵ動作モードにより、アドレス空間が、１Mbyte、16Mbyteと異なるため、RAM配置アドレスも 絶対値が異なる。
RAMも通常、先頭アドレスから順に変数やバッファ領域を確保していく。
スタックエリアだけが、最終アドレスから若い番地に向い使用される。
よって、静的変数、バッファ等の開始アドレスはCPUの違いによるRAM容量の違いを受ける。　スタックポインタの初期設定は、RAM容量の影響は受けない。

ROM先頭部分にプログラムを配置するので、プログラムサイズが数10Kbyteの場合は、のどちらのROMにおいても体制に影響は無い。





上記の16進アドレス値は、ＣＰＵモード５，７（ アドレス空間 1[Mbyte] ）の場合でありモード６（ アドレス空間 16[Mbyte] ）の場合の RAM配置アドレスは、頭に Fが付いた 6桁となる。（例： RAM終端アドレスは、FFF0FH --> FFFF0FH ）

当然、モード５（アドレス空間 1[Mbyte]）と、モード６（アドレス空間 16[Mbyte]）では、同じＣＰＵでも ＲＡＭ開始アドレスは異なるアドレスに配置される。
H8/3048Fの場合、
モード５の場合 RAM開始アドレスは　FEF10Hであり、
モード６の場合 RAM開始アドレスは FFEF10Hになる。

（ モード５及び ７において、命令のオペランドで 6桁のアドレスを指定しても、上位4bitは無視されるので、FEF10Hで指定しても、FFEF10Hで指定しても同じ結果となる。
しかし、モード６においては、FEF10Hと FFEF10Hを区別するので FEF10Hでは RAMをアクセス出来ない。）

RAMの場合は、サイズの違いにより先頭アドレスが異なるため、RAMの開始アドレス及びRAMサイズを意識する必要がある。
スタックは、通常 RAMの最終アドレスに配置するので、スタックポインタの設定は、RAMの最終アドレスを初期値として設定する。

スタックのサイズは、実行するプログラムにもよるが、マイコンのシステムであれば1～2 Kbyte程度でよいと思われる。　アセンブラだけで作られたプログラムの場合はこれで十分である。
Ｃで作られたプログラムの場合、Auto変数によってもスタックを消費するので、Auto変数では、配列データ等の大きなサイズのデータを宣言する事は極力避けなければならない。
あとは、どの程度サブルーチンや割り込み処理がネストするかを考慮してスタックサイズを決める必要がある。（ ちなみに昔の MS-DOSのＣコンパイラは、2048byteが標準のスタックサイズであった。 ）

ポインタ値	Mode5、Mode7 (1 Mbyte Address)	Mode6 (16 Mbyte Address)
FEF10H	RAMをアクセス可能	RAMをアクセス不可能
FFEF10H	RAMをアクセス可能	RAMをアクセス可能

[4] ＣＰＵクロックによる違い：

ＣＰＵクロックがプログラムに与える影響は、単に実行速度が速い遅いだけの問題ではない。　ＣＰＵのクロックを分周してタイミングを作り出している内部周辺回路があり、そのタイミングに影響を与える。
最も問題となるのは、シリアル通信のボーレイトジェネレータである。
ＣＰＵのクロックが早くなれば、比例してボーレイトも上がってしまうため、外部機器と正常に通信ができなくなる。
このため、ボーレイトをＣＰＵのクロック数に影響を受けず一定の値を保つように、各ＣＰＵクロックに応じた分周値テーブルを用意する必要がある。

その他、インターバルタイマーにおいても同様の事が発生する。
ＣＰＵクロックを早くすればインターバルタイマーの周期も早くなるので、タイマーで時間を計ったりしている場合は要注意である。
よって経過時間のカウント用途でタイマーを使用する場合も、クロック毎に分周値を設定する必要がある。
今回、ＣＰＵクロックの種類は、16MHz、20MHz、25MHzの３種類とする。
その場合に、タイマー発振周波数から １[MHz]を得る場合、

25 [MHz]は　1/25、
20 [MHz]は　1/20、
16 [MHz]は　1/16 となる。

タイマーの係数として水晶発振器の周波数から １[MHz]を得るための分周値を　用意しておく。

[5] 秋月マザーボードの違い：

初期のマザーボードは、DIPスイッチ、押しボタンスイッチ、発光ダイオード、液晶表示器の I/Oが付いたもので、CPUモード７で動作させる事を想定していた。
AKI_H8__USB基板にも、DIPスイッチ、押しボタンスイッチ、発光ダイオード、液晶表示器の I/Oが付いている。　しかし、それらが接続されているＣＰＵの端子に互換性がない。
もちろんもっと大きな違いとして、ＵＳＢの周辺回路が実装されている。
あるいは、ＬＡＮの周辺回路が実装されている等があるが、これらは各ボードで排他的要素であり、ソフト的にみると、ＵＳＢやＬＡＮそれだけで大きなモジュールとなり得るので基本的な共通I/Oには含めない事とする。

ここでは、マザーボードと、AKI_H8_USBの間で、DIPスイッチ、押しボタンスイッチ、発光ダイオード、液晶表示器を同等に扱えるようなインタフェースを用意することにする。
ただし、実装されているDIPスイッチのビット数や、AKI_H8_USBでは、液晶表示とLEDのポートが共通になっているなどの問題があり全く同等には扱えない。

[6] 仕様の違いを吸収する取り決め：

(1) ROM先頭アドレスは、CPUの種類、動作モードに関わらず 0 番地から始まる。
よって、割り込みベクトルテーブル、プログラムコードの開始アドレスは
固定と考えてよい。
＜分ける必要なし＞

(2) CPUの動作モードによる RAM配置アドレスの違いは、CPUモード５，７においても
絶対アドレスを指すオペランドは、24bitで扱う。
（ これにより動作モードによる RAM、I/Oポートのアドレスの違いを
吸収できる。）
＜分ける必要なし＞

(3) RAMの開始アドレスは、各CPUのRAM容量の違いにより開始アドレスがずれている
ため、CPUの違いにより、RAMの開始アドレスを定義する必要がある。
＜ CPU毎（ 3048，3052，3069 ）に設定する必要あり、
設定そのものは、１行で済むため 共通ソース内に３行用意して
そのうちの１つを生かし他の２つはコメントにする形で対応する。＞
共通ソース名（ define_H8.src ）

例）H8/3048を有効にした場合：

```
ram_head  .EQU  h'FFFE10          ; H8/3048F
; ram_head  .EQU h'FFDF10          ; H8/3052F
; ram_head  .EQU h'FFBF10          ; H8/3069F
```

(4) ＣＰＵクロックに依存する要素への対応。
ボーレイトジェネレータの分周値と、インターバルタイマ用に水晶発振周波数
から 1[MHz]を作り出す分周値を、CPU各周波数（ 25MHz、20MHz、16MHz ）毎に
分けて用意する。これは、３本のソースを用意する。

ソース名：	cpu_16.src	（ 16 MHz用 分周値定義ファイル ）
	cpu_20.src	（ 20 MHz用 分周値定義ファイル ）
	cpu_25.src	（ 25 MHz用 分周値定義ファイル ）

(5) マザーボードに付属する動作確認用途の I/Oを、マザーボード、AKI_H8_USB基板
にて極力同様に使えるようにする。

ソース名：	aki_mbio.mar	（ マザーボード用 ）
	aki_usbio.mar	（ AKI_H8_USB基板用 ）

拡張子に、src と mar を 使っているが 使い分けは、
src を 定数定義ファイル（ヘッダファイル）として使用し、
mar を コード実装部ファイルとして使用している。

[7]　共通基本モジュールのファイル構成：

H8のアセンブラ A38H.exe は インクルード機能や、マクロ機能がないため、あるファイル内の任意位置に別ファイルを取り込む処理や、行の置き換え、if文による アセンブル制御を行う事が出来ない。

よってインクルードと同等の処理を行うためには、ソースファイルを連続して複数本読み込ませ、１本のオブジェクトを生成する方法を用いる。

その場合、読み込ませる順番を意識する必要がある。

ソースファイル名		説明
define_H8.src		H8シリーズCPUの内部I/Oのポートアドレス等の定義ファイル （ ram_head 定義 含む ）
どれか一つ選択する	cpu_16mhz.src	16［MHz］ＣＰＵのカウンタ分周値定義ファイル
	cpu_20mhz.src	20［MHz］ＣＰＵのカウンタ分周値定義ファイル
	cpu_25mhz.src	25［MHz］ＣＰＵのカウンタ分周値定義ファイル
Rst_Int.mar		割り込みテーブルと、リセット処理と割り込み処理、 ＣＰＵ内部 周辺回路処理
どちらか一つ選択する	aki_mbio.mar	秋月マザーボード 基本I/Oルーチン
	aki_usbio.mar	AKI_USB基板 基本I/Oルーチン
ram_head.mar		共通モジュール用変数エリア宣言

define_H8.src 、cpu_16mhz.src 、cpu_20mhz.src 、cpu_25mhz.src は、コードの実態を生成しない、値の定義ファイルのみである。

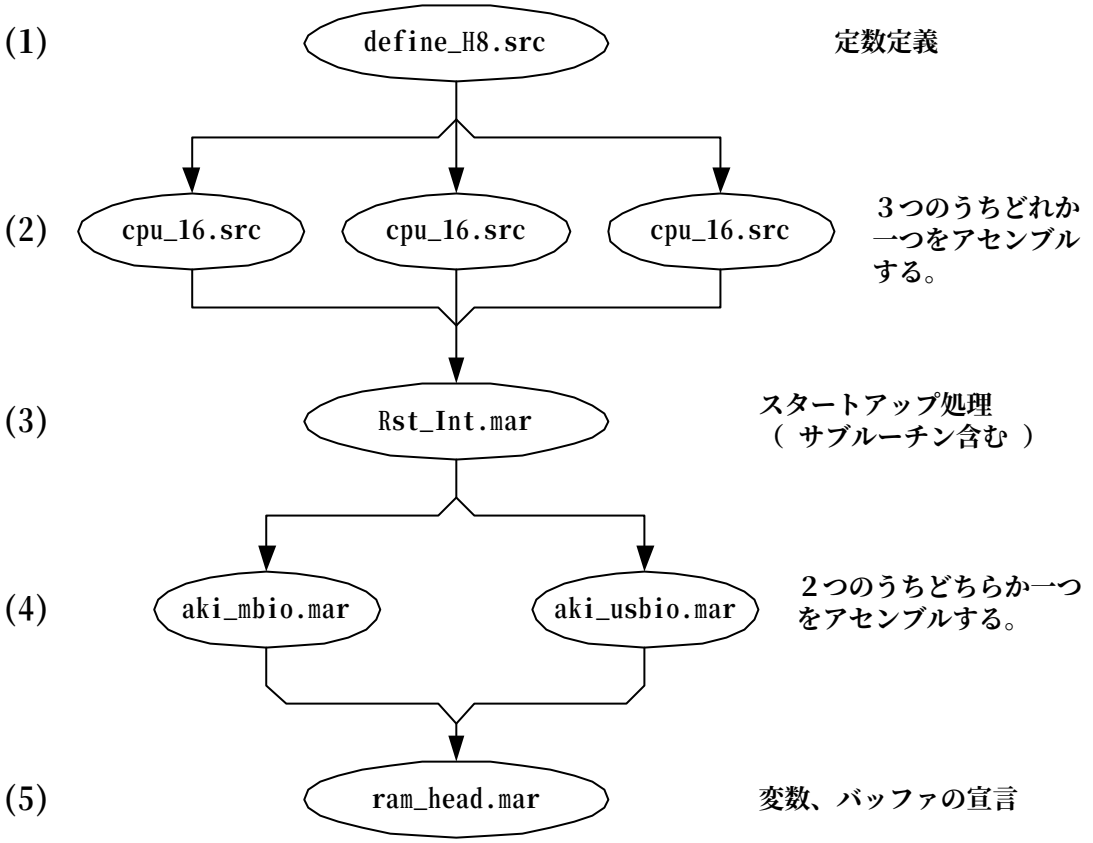
Rst_Int.mar 、aki_mbio.mar 、aki_usbio.mar 、ram_head.mar コードや変数等のオブジェクトの実体を宣言するソースファイルである。

Rst_Int.mar 、aki_mbio.mar 、aki_usbio.mar は、ROM上に配置され、ram_head.marは RAM上に配置される。

この中で Rst_Int.mar は、割り込みベクトルテーブル、Ｃ言語の main を呼び出すためのスタートアップ処理等、固定アドレスを含む処理を行っている。

その他、Ｈ８ＣＰＵに内蔵される I/Oのうちシリアル通信とタイマーを一部サポートしている。

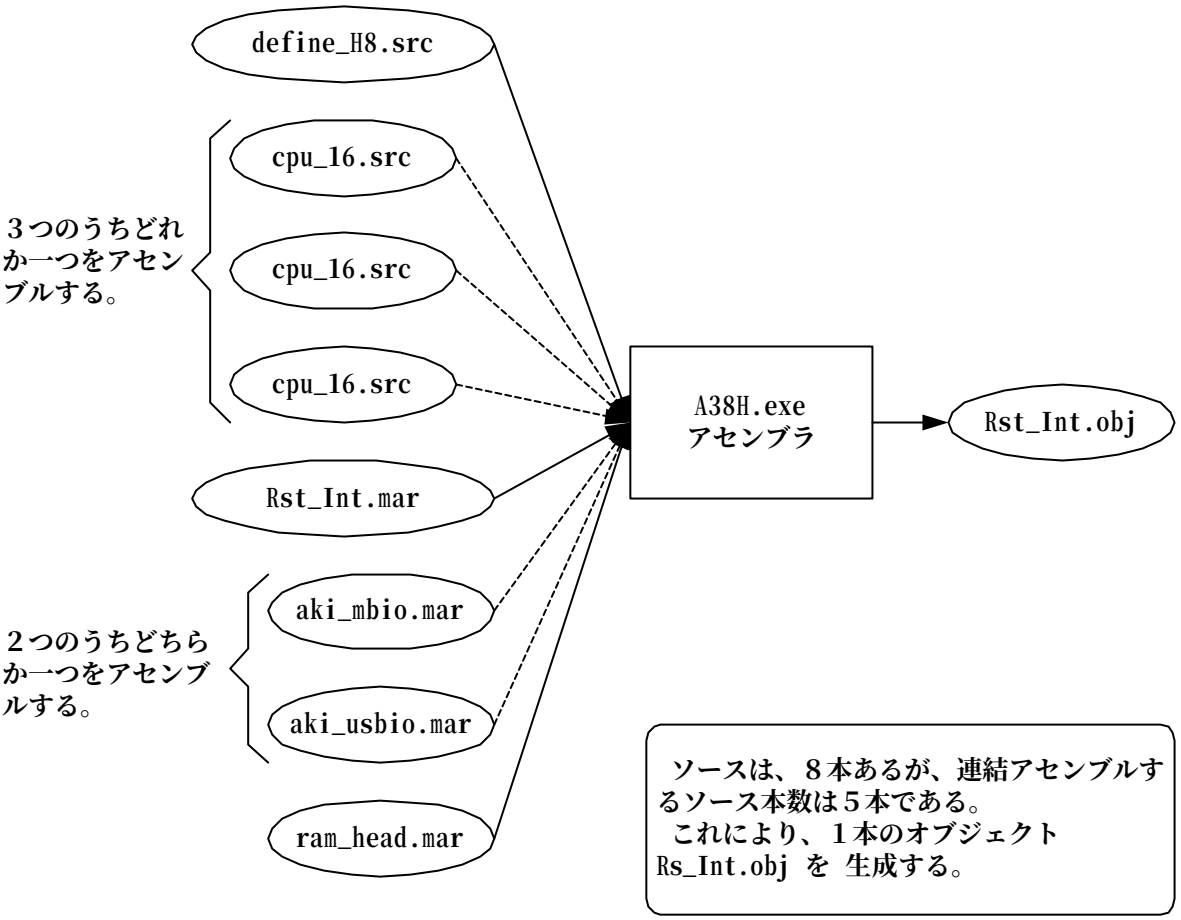
[8] アセンブルの順序：



[9] ソースとオブジェクトの関係：

以下のソースファイルを、連結、選択アセンブルする事により、オブジェクトファイル（ Rst_Int.obj ）を生成する。

Rst_Int.objは、Cオブジェクトのスタートアップモジュールなので、次の作業としてこのRst_Int.objと Cで記述されたメイン、及びサブルーチンをリンカで連結し実行モジュールを生成する。



[10] アセンブルのバッチファイル例（ rst.bat ）：

```
rem ★  H8/3048 ＋ AKI_MB
a38h define_h8,cpu_16mhz,Rst_Int,aki_mbio,ram_head -object=Rst_Int >err.txt

rem ★  H8/3052 ＋ AKI_USB
rem a38h define_h8,cpu_25mhz,Rst_Int,aki_usbio,ram_head -object=Rst_Int >err.txt

type err.txt
```

この中で、ＣＰＵの種類、基板の種類で変更になるものは、

(1)　define_H8.src内の ram_headの定義

(2)　cpu_16mhz　（ CPUのクロックにより cpu_20、cpu_25に置きかえる。 ）

(3)　aki_mbio　（ AKI_H8_USB基板を使用する場合は aki_usbio に置きかえる。 ）

[11] ターゲット環境を切り換える際の作業：

最初のデバッグの際は、モニターデバッガで、RAM上でデバッグしていて、本番のROM版オブジェクトを生成する際に変更する個所。

(1) プログラムコードのスタートアップ部分を RAM -> ROM に配置し直す。
define_H8.src 内の vct_offsetの変更
（ 使用しない方を ; でコメント化する。 ）

```
vct_offset      .EQU    h'0             ; 通常の ROMベクタオフセット
;vct_offset     .EQU    h'FFF000        ; 仮想ＲＡＭベクタオフセット
```

(2) Rst_Int.obj 生成用バッチファイル rst.bat の内容も CPUクロック及び、
使用マザーの種類に合わせる。（ 詳細は前ページ参照の事 ）

rst.bat を実行し 変更した Rst_Int.objを生成する。

(3) リンカに渡すパラメータファイル（ *.sub ）内にて Ｃ言語で使用する
コード、データ領域の配置情報を変更する。

ROMに配置する場合：
START P(000200),C(002500),B(0FFF000)

RAMに配置する場合：
START P(0FFF200)

リンカへ渡すパラメータファイルの例（ test.sub ）

```
INPUT rst_int, test
OUTPUT test
PRINT test
LIB \aki_h8\c\c38hab
START P(000200),C(002500),B(0FFF000)
EXIT
```

リンカパラメータ（ ?.subファイル ）を変更して、
リンク、コンバート作業を実行し、Sフォーマット16進ファイルを生成する。

尚、リンクパラメータの STARTパラメータは、プログラムのサイズ、使用ＣＰＵにより変更する方が望ましい。

B(??????)は、実質RAMの開始アドレスになるが、FFF000は H8/3048の場合であり3052、3069でもそのままで動作するが、RAMを有効利用しているとはいえない。
RAMを大量に使用する場合は、調整の必要性が生じる。
及び、 C(??????) の値もコードサイズが巨大になった場合は、調整が必要となる。

[12]　ターゲットモジュールが変更されない場合：

ソースモジュール、及びコンパイル、リンクのパラメータを変えてターゲットモジュールを再構築しているにもかかわらず書き換わってないような場合が発生する場合があります。
現象としては、デバッグのためROM版から、RAM版に変更した際に MOTファイルを転送すると、0 番地から転送を始めていたり、あるいは、デバッガで 逆アセンブルすると、内容が 明らかに変だったり訳の無からない現象が発生する場合がまれにあります。

当方で確認した現象としては、MOTファイルが何らかのタイミングで、ファイルオープン中のままになってしまい、排他制御によりファイルを書き換える事が出来なくなる現象を２回ほど確認しました。
この状態になると、MOTファイルを更新できないため、修正内容がターゲットに反映されなくなる。　MOTファイルを消す事も出来ないのでその場合は、開発用パソコンを再起動するしかありません。

再構築する際は、拡張子が、OBJ、ABS、MOT、MAP、LIS のファイルを削除するバッチファイルを用意して削除に成功したのを確認してから、再構築を行うほうがいいと思われます。

[A]　H8/3048と H8/3052の 書き込み時の電圧：

書き込みに関わる端子は、VPP(FWE)と MD2の２つである。

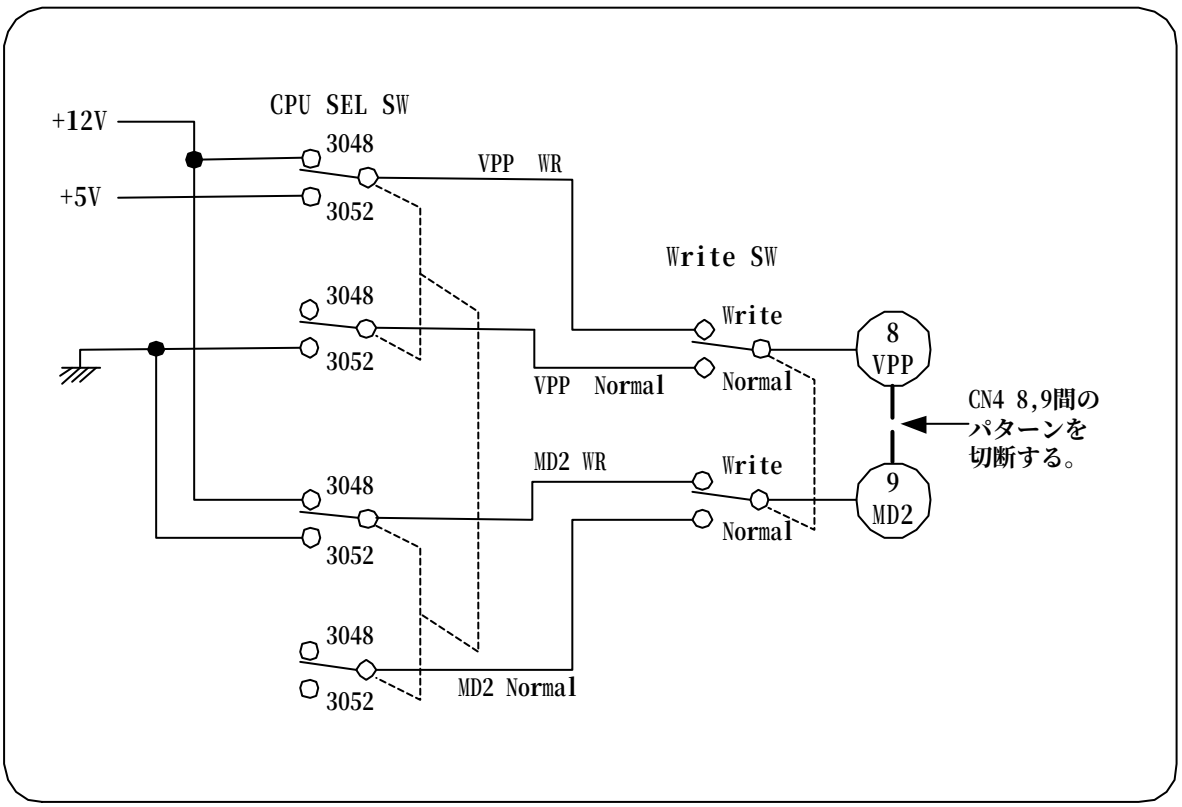
H8/3048F / H8/3052F	8 PIN VPP (FWE)	9 PIN MD2
書き込み時	+12V / +5V	+12V / GND
通常時	NC / GND	NC / NC

[B]　AKIマザーの改造：

AKIマザーボードにおいて、H8/3048と H8/3052の両方の書き込みに対応出来るような改造を行う場合はどのようにするかを検討する。

まず、旧マザーにおいては、8.VPPと 9.MD2がコネクタ部パターンで接続されているためこれをパターンカットする必要がある。

次に以下のようなスイッチによる切換を行えばよいと思われる。



[C] C と アセンブラのインタフェース：

C言語から、アセンブラのルーチンを呼び出す場合に、引数と関数値の引渡しをどのように行うかを明確にしておく必要がある。
渡す引数が、4 byte以内の整数、ポインタの場合、
第１引数が ER0レジスタ、第２引数が ER1レジスタにて渡される。
関数値は、4 byte以内の整数、ポインタの場合、ER0レジスタが 使用される。

Ｈ８のＣコンパイラでは、 int で宣言された変数は、16 bit で 扱われるようで
引数で渡す場合、R0 または、R1にて渡され、 関数値も R0 にて 渡される。

プロトタイプが宣言されてない場合（ 引数の型の宣言が無い場合 ）は long として
扱われ、ER0、ER1で引数は 渡される。

３番目以降の引数が有る場合は、スタック上に引数は、積み上げられる。
この場合、Word の データは、 push.w で 積み上げられ、
long の データは、push.l で 積み上げられる。
（ byteのデータは、 Word として積み上げられると思われる。 ）

注意：
引数が３つ以上あっても、ER0、ER1に 詰め込む事が可能であれば
詰め込む仕様になっている。

引数が全て byteの場合： dummy(char, char, char, char)
R0L、 R0H、 R1L、 R1H

引数が全て int の場合： dummy(int, int, int, int)
R0、 E0、 R1、 E1

混在引数の例１： dummy(char*, int, int)
ER0、 R1、 E1

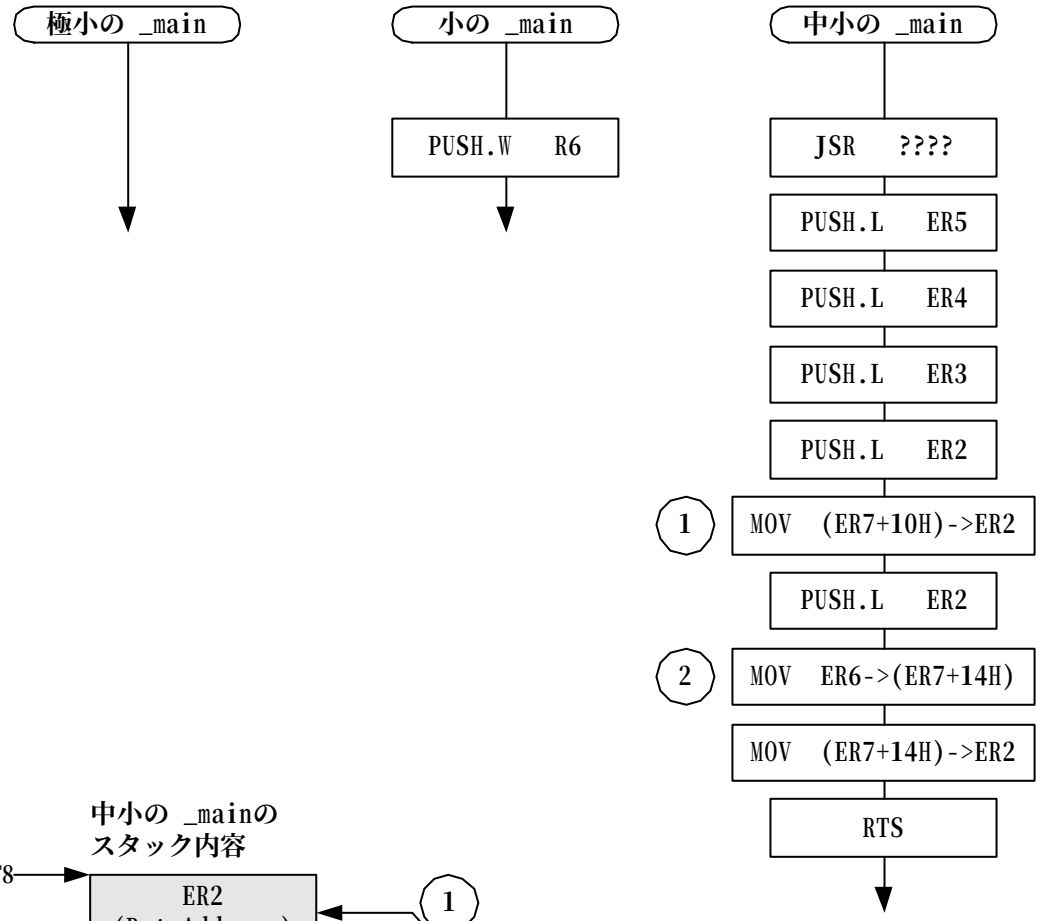
混在引数の例２： dummy(char, int, char, char)
R0L、E0、 R0H、 R1L

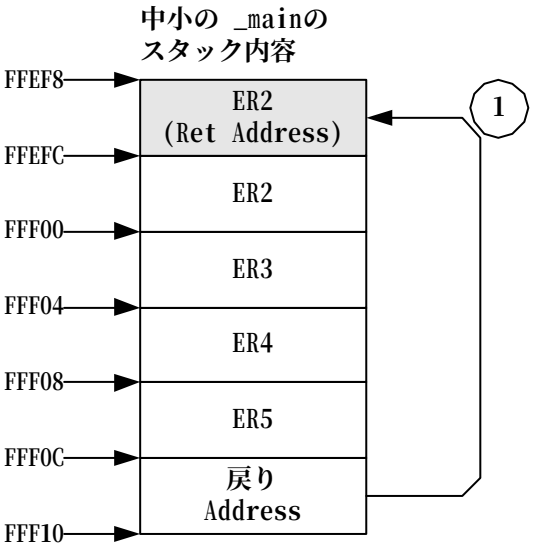
混在引数の例３： dummy(char*, char*, int)
ER0、 ER1、 push.w
エントリ直後の、３番目の引数位置は ER7 + 4

混在引数の例４： dummy(char*, char*, int, int)
ER0、 ER1、 push.w、push.w
エントリ直後の、３番目の引数位置は ER7 + 4
エントリ直後の、４番目の引数位置は ER7 + 6

[D]　Cの初期化処理：

　アセンブラのスタートアップルーチンから、C言語にて作成された _mainを
呼び出しているが、_mainの エントリ部分で暗黙に呼び出される処理が存在する。
但し、これは main関数あるいは、Cプログラムの大きさにより変わるようで
極めて小さなmain関数の場合、生成されない。
少しmain関数の記述が増えると、R6レジスタがプッシュされてから処理が開始される。
更に main関数の記述が増えるとスタックを操作するサブルーチンが呼び出される。
スタック操作のサブルーチンは ER5、ER4、ER3、ER2 をスタックに積み上げたまま
リターンするようになっている。（ コンパイラが処理を行う上で必要となる内部変数（レジスタ変数）を確保するための処理と思われる。）
この、３段階の暗黙のエントリ処理はどのような判断で挿入されるかは不明。
main関数の大きさの表現は、感覚的であるが以下の状況を確認した。





2

ER6 の 内容を、サブルーチンの（元の）戻りAddress（ FFF0C ）に書き込み、その値を再度、ER2に読み込み直している。

[E] ターミナルの [BackSpace] 処理：

ターミナル処理にて文字を入力し間違えた時に、[BackSpace] Keyにて訂正出来るが、ただ単に [BackSpace] Keyをエコーバックするだけでは バックスペースの動作にならない。

まずは、パソコンのターミナルソフト側にてどのようなコードが出力されるのか確認する必要がある。 調査した結果は、

[BackSpace] ＝ 08H
[↑] ＝ 1AH
[↓] ＝ 1AH
[→] ＝ 1AH
[←] ＝ 1AH
[Esc] ＝ 1AH

矢印 key や [Esc] key は 区別がつかないので 無視した方が良いと思われる。

パソコンのターミナル画面上に 08Hを返すと、カーソルポジションは、一つ左に戻るがカーソル位置の文字は消えない。

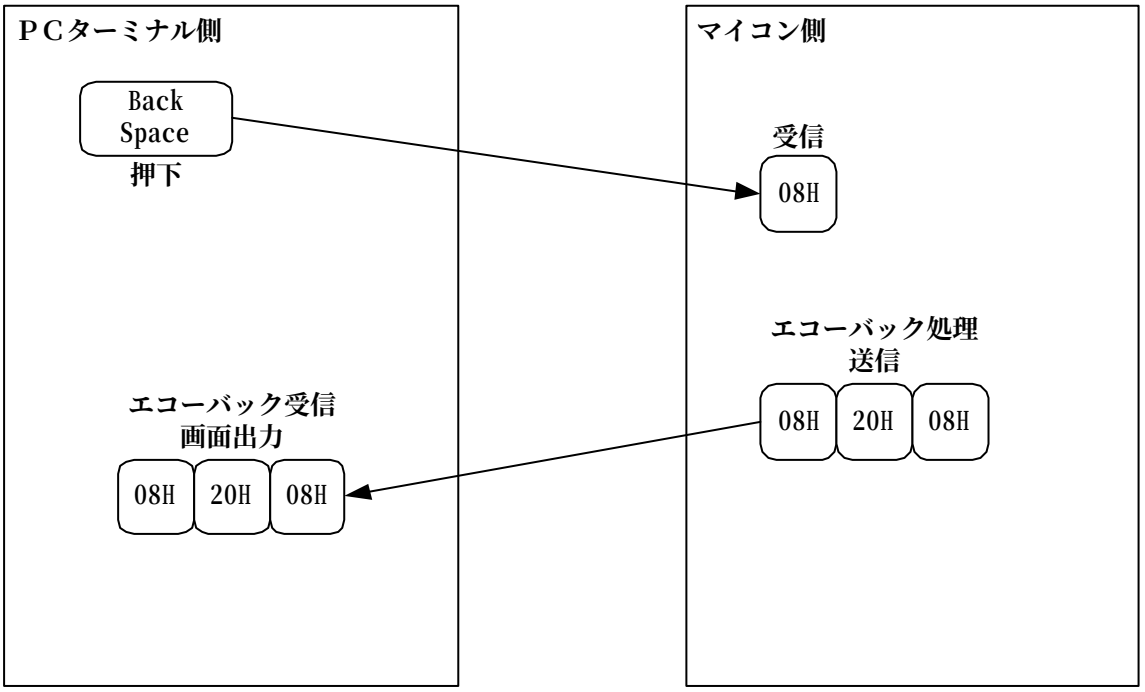
[Back Space] Key 押下時、カーソル位置の文字が消えるようにするには、その位置でスペースコードを１文字出力して文字を消し、再度 08H を 出力する事によりカーソル位置を戻す。

マイコン側での処理：

入力： 08H

に対するエコーバック処理は

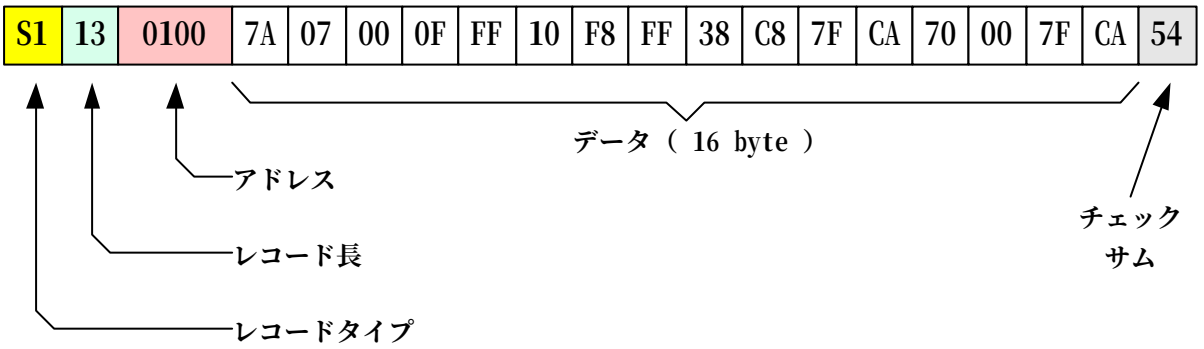
出力： 08H、20H、08H の 3byte 出力する。



[F]　モトローラＳフォーマット（ 16進ファイル ）：

　Ｈ８に関わるユーティリティは、Ｈ８マイコンに転送するファイルとしてモトローラＳフォーマットを採用している。
　自前でローダなどを作成する場合はＳフォーマットの内容を理解しておく必要がある。
Ｓフォーマットファイルの例

```
S00E00006C656420202020204D4F542D
S107000000000100F8
S11301007A07000FFF10F8FF38C87FCA70007FCA54
S113011072107A01000F42401B7146FC7FCA7200C5
S11301207FCA70107A01000F42401B7146FC5A00CF
S1050130010ABF
S9030100FC
```



レコードタイプ （先頭は必ず Ｓ ）	S0	スタートレコード
	S1	データレコード（ 16 bit アドレス ４文字で指定 ）
	S2	データレコード（ 24 bit アドレス ６文字で指定 ）
	S3	データレコード（ 32 bit アドレス ８文字で指定 ）
	S4	シンボルレコード（ LSI拡張 ）
	S5	今までに出てきたデータレコード数
	S6	未使用
	S7	Ｓ３フォーマット（ 32bitアドレス ）の終了
	S8	Ｓ２フォーマット（ 24bitアドレス ）の終了
	S9	Ｓ１フォーマット（ 16bitアドレス ）の終了
レコード長	以下に続くレコードで表されるデータの数（ Byte数 ）を２文字で示す。 （ レコード長、アドレス、データ 部が 対象となる。）	
アドレス	SIレコードは 4 文字 、S2レコードは 6 文字、S3レコードは、8 文字にて格納されるアドレスを 指定。	
データ	２文字で、1 byteの データを 表す	
チェックサム	２文字（ 1 byte ）のデータ。 レコード長、アドレスフィールド、データフィールドの各バイト値の合計の２の補数 （ インターネット上の資料では、１の補数としてある資料もあった。 ）	

[G] Word、DWord変数の配置アドレスの注意：

インテル手巣 i86系ＣＰＵの場合は、あまり問題にならなかったが、Ｈ８シリーズのＣＰＵの場合、プログラムも、Word、DWord変数に関しても、Word境界を意識しなければならない。　つまり、必ず偶数アドレスに配置する必要がある。
これを、怠って配置された Wordデータをアクセスする際に、その変数が奇数アドレスに配置されていると、1 byte若いアドレス（アドレス情報の最下位 bit が切り捨てられた状態）になり、隣り合う変数の値を取り込んだり壊したりする恐れがある。

具体的には、ａというWord変数が、３番地に配置されていた場合は、２番地をアクセスしてしまう現象が発生する。
基本的に Wordマシン（ 16bit幅でメモリアクセスする ）なので Word、DWordの変数は、Word境界に配置する。

Word変数を、３番地（奇数アドレス）に配置した例。

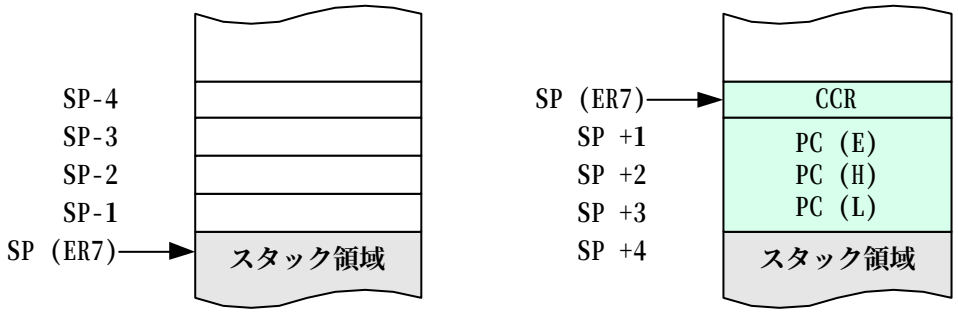
0 番地	1 番地
2 番地	3 番地
4 番地	5 番地
6 番地	7 番地

上のように変数が、Word境界を跨ぎ段違い状に配置された場合に、
2,3番地をアクセスする事になる。

Word変数を、２番地（偶数アドレス）に配置した例。

0 番地	1 番地
2 番地	3 番地
4 番地	5 番地
6 番地	7 番地

[H] 割り込み、例外処理発生時のスタック：



[I]　Ｃ言語の 初期化変数の ROM化における注意点：

RAM上でデバッグしていた時には、正常に動作していたのに、ROM化のためアドレスを変更しROMにプログラムを焼き込み、実行Dすると動作がおかしい。
という現象が発生する場合がある。

これは、初期化されているはずのデータが初期化されない状態で実行した場合の症状として現れる。

アセンブラでプログラムを作成する時は、ROM,RAMを意識してプログラムを作成するのでこの現象は起こりにくいがＣの場合、どこまでがROMに配置され、どこからがRAMに配置されるかが、コーディング上では直接見えない。

基本的に変数は、RAMに配置される。　その中で、Ｃの文法上変数の宣言と初期化が同時に行えるようになっているがこれが問題となる。
デバッグ時、ターゲットプログラムをRAM上にロードしそのまま実行するばあいは、初期化される変数の値もそのまま、RAMに転送されて、その状態で実行する関係上、初期化されるべき変数の値が正常に初期化されている。

しかし、書き込みモードで、ターゲットプログラムをROMに書き込み、一旦電源を切って、VPP(WE)のOFF、ＣＰＵモードのジャンパーによる切換えを行い、再度電源ＯＮするとROM上のプログラムコードは、残っているがRAM上の変数は初期化されない状態になっている。

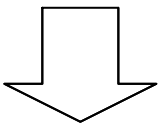
この現象を回避するには、２通りのやり方があり、一つは、リンカのパラメータで、Ｒセクションを宣言する事により、どうにか出来るらしいが詳しい説明がない。

もう一つの方法は、変数の宣言と初期化を同時に行わない事。

① 変数の宣言だけ行う。
② 初期化処理（ 実行文 ）の中で、変数に 定数を代入する。
　　（ 定数は、書き換えられる事がないので、ROM上に配置される。 ）

ROMに焼き込んでから、正常に動作しないコーディング例：

```
static char    *msg = "Test Program";
static int     no = 20;
```



上記の場合は、このように変更する。

```
static char    *msg;            /*  変数は宣言のみ行い、ここでは   */
static int     no;              /*  初期化は、行わない事           */


/*  初期化ルーチン       */
void  syokika( void )
{
       msg = "Test Program";    /*  初期化処理の実行段階で         */
       no = 20;                 /*  初期値を代入する事             */
}
```