

H8/3664FPフラットマイコン 開発キット

超小型サイズ(40 x 27mm)に部品実装済。
開発用アセンブラソフト、モニターデバッガソフト、
Cコンパイラ一式付属。



日立フラッシュマイコンH8/3664F (ROM32K・RAM2K) 使用

AKI-3664フラットマイコンキット

超小型サイズ(40mm×27mm)に部品実装済

開発用アセンブラソフト、モニターデバッカソフト、Cコンパイラ一式付属

(開発セットはソフトが付属します。ボード単体セットにはソフトは付属しません)

- ★フラッシュROM (32K) を内蔵していますので、100回以上書き替え可能です
- ★RAM (2K) ・10ビットA/Dコンバータ8ch・タイマー・SCI (RS232) 1ch・I²Cバスインターフェイス内蔵の高機能マイコンキットです。
- ★フラッシュROM書き込み制御回路、SP232 (MAX232) を装備していますので、パソコンに接続するだけでソフトの書き込みができます。
(開発セットには、パソコン接続用にベースボードパーツセットが付きます)
- ★メインクロック (1.6MHz) と1Hzが簡単に作れるサブクリスタル (32.768KHz) を両方装備しています。
- ★チップ部品の大幅採用により、外部接続用コネクタ以外は全て実装半田付け済みです。
(コネクタ付属)
- 部品表A ■ AE-3664FP (数に○が有る部品は基板に実装半田付け済みです。)

記号	種類	数	備考
IC1	HD64F3664FP	①	H8マイコン
IC2	ADM232AARN	①	RS232レベルコンバータ
IC3	TA78L05F	①	5V3端子レギュレータ
C1~5,7,9	0.1μF	⑦	チップ型積層セラミックコンデンサ
C8,10	1.0μF/10V	③	チップ型積層セラミックコンデンサ
C11	4.7μF/35V	①	電解コンデンサ
C12	100μF/10V	①	電解コンデンサ
C13,14	15pF	②	チップ型積層セラミックコンデンサ
D1	ダイオード	①	1SS388 (東芝)
R1	4.7KΩ	①	チップ型抵抗
R2	100Ω	①	チップ型抵抗
R3,4,5	10KΩ	③	チップ型抵抗
X1	セラミック発振子 16MHz	①	CSTCV16.0MXJ0C3
X2	水晶発振子 32.768KHz	①	円筒型
基板	AE-3664FP	1	部品実装済み
ピンヘッダ	26pin(13x2) CN1,2冊	2	} 80ピン(40x2)を26ピン×2ケと } 2ピン×2ケに切って使用する 2.54ピッチ
ピンヘッダ	2pin(2x1) JP2,3冊	2	
ジャンパーピン	JP2,3冊	2	
ピンソケット	26pin(13x2) CN1,2冊	2	

- 部品表B ■ ベースボード(ボードセットの場合には部品表Bは付属していません。)

記号	種類	数	備考
基板	C型ユニバーサル	1	
CP1	Dsub9Pコネクタ	1	基板取り付け用 Lアングルタイプ(メス)
SW1	タクトスイッチ	1	リセットスイッチ
R1	0Ω	1	
電池スナップ		1	電源接続用
CD-R		1	アセンブラ、モニターデバッカ、資料
CD-R		1	Cコンパイラ

■セット内容■

キットの種類により内容がちがいます。あらかじめご確認ください。

- ① H8/3664フラットマイコン開発キット
 - H8/3664フラットマイコン基板 (部品表A)
 - H8/3664用ベースボードパーツセット (部品表B)
- ② H8/3664フラットマイコンボードセット
 - H8/3664フラットマイコン基板 (部品表A)
 (部品表Bのみの販売はございません。)

■製作■

部品は予告なく相当品・互換品に変更になることがあります。メーカーにより若干型番が異なることがあります。製作前に部品表と照らし合わせてご確認ください。

部品表、回路図、部品配置図、組立図を参考にしてください。

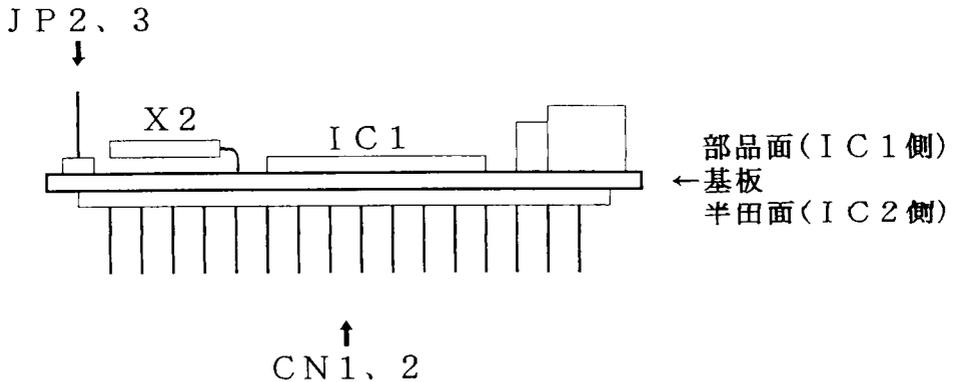
特に難しい半田箇所はありませんので、1つ1つ確実に半田付けしてください。

A、H8/3664フラットマイコン基板 (部品表A)

ほとんどの部品はすでに実装半田付け済みです。取り付ける部品はCN1、2、JP2、3のピンヘッドのみです。ピンヘッドは80ピン分(40ピン2列)が入っています。あらかじめ、26ピン(13ピン2列)を2つと、2ピンを2つに切ってください。

CN1、2は半田面(IC2側)に取付け、部品面(IC1側)から半田付けします。
 JP2、3は部品面(IC1側)に取付け、半田面(IC2側)から半田付けします。
 JP1は何も取付けません。(JP1は ■電源について■を参照してください)

ピンヘッドはお客さまのシステムにあわせて、部品面側に取り付けてもかまいませんがベースボードを使用する場合は下図のように取付けてください。



B、ベースボードパーツセット (部品表B)

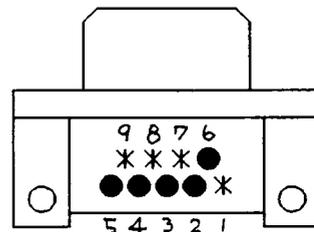
この基板はパソコンと接続して書き込みや、動作チェックなどを行なうものです。書き込み後は、ベースボード基板からはずして ユーザー基板で動作します。基板がユニバーサル基板ですので、組立図にしたがい慎重に製作してください。

1、Dsubコネクタの加工

Dsubコネクタは、ピンのピッチが基板とちがいますので、あらかじめ1、7、8、9の4つのピンを切っておきます。

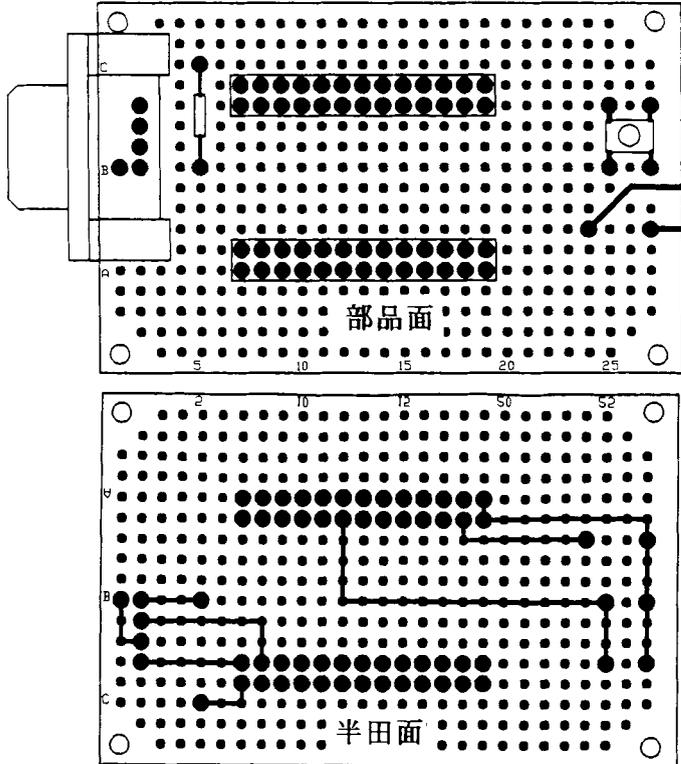
BOTTOM VIEW
(ピン側から見た図)

- = 残すピン (2,3,4,5,6)
- * = 切るピン (1,7,8,9)



Dサブ9Pコネクタ

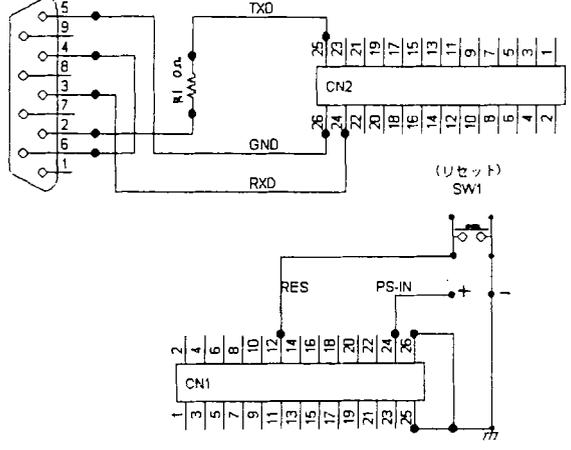
- 2、基板の部品面（5、10などの白い印刷のある面）に部品を取り付け、半田面側から半田付けします。CN1、2は部品表Aのピンソケットを取付けてください。Dsubコネクタのピンは、少し曲げるようにして基板の穴に差し込んでください。
- 3、電源接続用に電池スナップの赤を+に、黒を-に半田付けします。
- 4、メッキ線などで、半田面側で配線します。



■ベースボード部品配置図■



■ベースボード回路図■



■電源について■

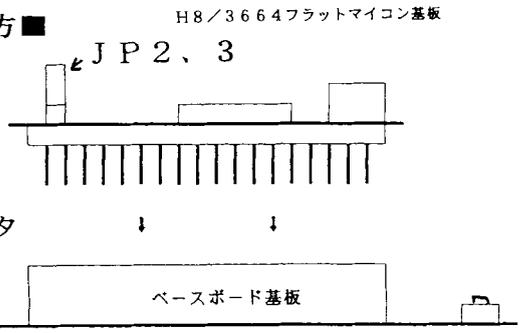
H8/3664フラットマイコンは動作、書き込み共に5Vです。
 このH8/3664フラットマイコン基板には、5Vレギュレータ（IC3）が、すでに取付けてあります。このレギュレータを使用する場合には、電源は7~12V 100mAです。

外部からの5Vを使用する場合はJP1のパターンを切ってIC3を切り離してください。（JP1は出荷時は接続されています。電源は7~12V仕様になっています）

- 1、ベースボードを使う場合（H8基板には、ベースボードから電源が供給されます。）
 7~12Vをベースボードの赤+、黒-に接続してください。
 （電池スナップに006P型9V電池を接続して書き込みや動作が出来ます。）
- 2、ベースボードから外して使う場合（7~12V電源）
 JP1は切り離さない。
 CN1-24（PS-IN） ---（+）7~12V 100mA以上
 CN1-25、26（GND） --（-）
- 3、ベースボードから外して使う場合（5V電源）
 JP1を切り離さず。
 CN1-23（5V） -----（+）5V 100mA以上
 CN1-25、26（GND） --（-）

■マイコン基板とベースボード基板の組み合わせ方■

ベースボードのDサブコネクタ側に、H8/3664基板のJP2、3がくる様に差し込んでください。



■書き込みファイル■

このキットでは、H8/3048キットとちがい、Sタイプフォーマットの、MOTファイルは使用せず、SYSROF形式の、ABSファイルで、すべておこないます。(書き込み制御ファイル3664、MOTを除く)

・MOTファイルを使用するとエラーとなる場合があります。

書き込みはFLASH、EXEではなく、MS-DOS用のHTERM、EXEで書き込みを行ないます。

■MS-DOS (DOSプロンプト) について■

本キットのソフトは、WINDOWS 98/95のDOSプロンプトで動作します。

フォルダ(ディレクトリ)の作成、ファイルのコピーや削除はWINDOWSで行なってください。また、各ファイル名・フォルダ名はすべて半角文字12文字以内に行なってください。

ソースファイル等の作成、編集や内容の表示は、WINDOWSの「文字を編集するソフト」メモ帳やワードパットなどで行なってください。

以下の説明で

▲は、「半角1文字分のスペース」を表します。

↓は、「リターンキーを押す」を表します。

フォルダ内のファイル名一覧を画面に表示するには、

DIR▲/W↓ とします。

フォルダを移動するには、

CD▲フォルダ名↓ (例 CD▲C:¥3664↓) とします。

■CDの内容■

¥-¥asm [アセンブラ、リンカー]

-asm38.exe (アセンブラソフト)

-lnk.exe (リンカーソフト)

-h8sman.doc (ワード形式 アセンブラ・リンカー説明)

-h8sman.pdf (PDF形式 アセンブラ・リンカー説明)

¥writer [ライター、ターミナルソフト]

-3664.mot (書き込み制御プログラム)

-hterm.exe (ターミナルソフト)

-readme.txt (テキスト形式 ターミナルソフト説明)

¥monitor [アセンブラ用モニターデバッカ]

-monitor.abs (モニタープログラム)

-readme.txt (テキスト形式 モニターソフト説明)

¥sample サンプルファイル [サンプルソースファイル]

¥data H8/3664ハードウェア、ソフトウェアマニュアル (PDF形式)

¥making 製作例やms-dosの説明、ソフトの作り方の説明

(making内の基板製作はH8/3664DIP型基板の例です

ms-dosの説明、ソフトの作り方の説明のみを参考にしてください。)

■ソフトの準備■ (すべて、WINDOWSで準備をおこないます。)

① はじめに、CD内の各説明ファイル、各マニュアルを印刷してください。

★PDF形式のファイルを開覧、印刷するには、Acrobat Readerが必要です。あらかじめ、www.adobe.comでダウンロードしてください。

② フォルダをつくる。

Cドライブに3664(3664は半角数字)と言う名前のフォルダをつくる。

このフォルダは他のフォルダ内ではなく、C:のすぐ下に作ってください。

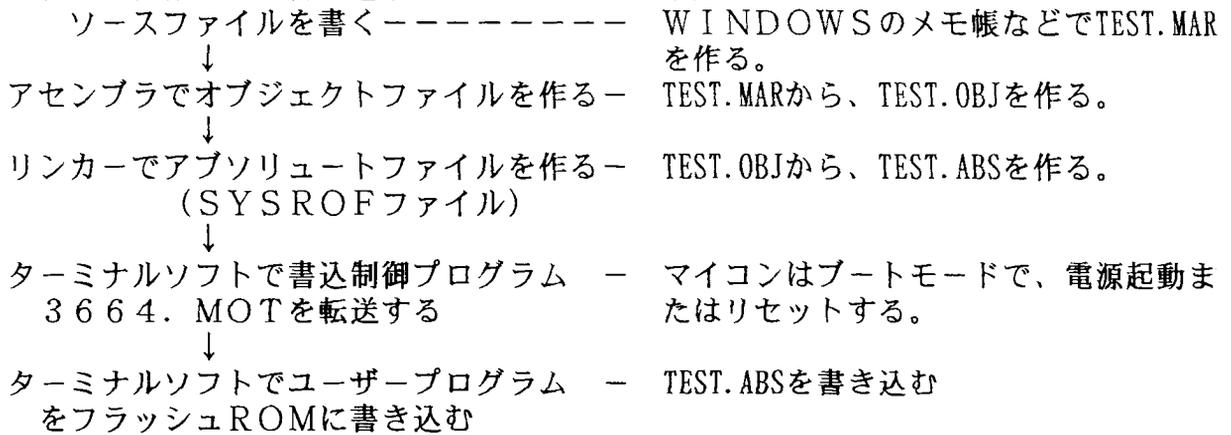
③ ソフトをコピーする

CD内の各フォルダのファイルすべて(説明ファイルは除く)をすべて、3664フォルダにコピーしてください。

④ ソースファイルの読み取り専用属性を解除する。

CD内のファイルは、属性が読み取り専用になっています。各ソフトはそのままでも良いですが、ソースファイルは編集などしますので、読み取り専用属性を解除してください。(ファイルのプロパティを開き、読み取り専用のチェックを外す)

■ソフト製作から、書き込みまでのおおまかな手順■



■アセンブラ、リンカーの使い方■

アセンブラ、リンカーの詳しい使い方はh8sman.docをごらんください。
sample.txtがアセンブラソース内でのラベルやシンボルなどの使い方例ですのであわせてごらんください。

ここでは、TEST.MARからTEST.ABSを作る手順を説明します。

①WINDOWSから、DOSプロンプトを起動し、3664フォルダに移動(チェンジディレクトリ)する。

C:¥WINDOWS>CD▲C:¥3664↓ MS-DOSのCDコマンドでC:¥WINDOWSから、
C:¥3664> C:¥3664に移動

②アセンブラソフト asm38.exeでtest.marから、test.objを作る。

C:¥3664>asm38.exe▲test.mar↓
H8S,H8/300 SERIES CROSS ASSEMBLER Ver. 2.0A Evaluation software
Copyright (C) Hitachi, Ltd. 1994,1998
Copyright (C) HITACHI MICROCOMPUTER SYSTEM LTD. 1994,1998
Licensed Material of Hitachi, Ltd.

*****TOTAL ERRORS 0

*****TOTAL WARNINGS 0

(ソースファイルtest.marの1行目に .PRINT LISTコマンドがありますので、リストファイルtest.lisが同時に作られます。
リストファイルは、プログラムが実際にどのアドレスに割り当てられたかを示すファイルで、プログラムをデバックするときに使用します。)

③リンカーソフトlnk.exeでtest.objから、test.absを作る。

C:¥3664>lnk.exe▲test.obj↓
H SERIES LINKAGE EDITOR Ver. 5.3B Evaluation software
Copyright (C) Hitachi, Ltd.1989,1998
Copyright (C) HITACHI MICROCOMPUTER SYSTEM LTD. 1990,1998
Licensed Material of Hitachi, Ltd.

LINKAGE EDITOR COMPLETED

書き込み用のtest.absが、つくられました。

■ ユーザープログラムをフラッシュROMに書き込む ■

ユーザープログラムはターミナルソフト `h term. exe` で書き込みます。
`h term. exe` の詳しい使い方は `readme. doc` をご覧ください。
ここでは、TEST. ABS を ROM に書き込む手順を説明します。

① パソコンと接続する。

H8 / 3664 フラットマイコンは、ベースボードに装着し、ベースボードの CP1 をパソコンの COMポート (RS232C) に接続します
COM1 ~ 4 に対応しています。ケーブルはストレートケーブルです。
(Dサブ9P-9Pストレートケーブル 別売1本400円で発売中)

② 基板 JP2、JP3 をブートモードに設定する。

JP2、JP3 はキット付属のジャンパーピンを差すとブートモードになります。
ここでは、まだ電源を入れないでください。

③ ターミナルソフト `h term. exe` を起動する。

`h term. exe` は、デフォルトで COM1, 19200bps ですので、そのまま起動します。

`C:\¥3664>hterm.exe ↓`

Terminal Program for H Series Monitor Ver. 5.0

と起動メッセージがでます。

④ フラッシュROM書き込みコマンド [CTRL+F] を入力する。

[CTRL+F] は、パソコンの [CTRL] キーと [F] キーを同時に押します。

Set Boot Mode and Hit Any Key.

←←押すと、このメッセージがでます。

⑤ ここで、キットの電源を入れ、その後、パソコンのキー (例えば、A) を押します。

Bitrate Adjustment Completed.

←←押すと、このメッセージがでます。

⑥ 書き込み制御ファイル名 3664. mot を入力します。

Input Control Program Name : 3664. mot ↓

←←3664. mot を入力

transmit address = FA2C

←←ここで、もたつく場合あり。

Flash Memory Erase Completed.

⑦ ユーザープログラム名 test. abs を入力します。

Input Program File Name : test. abs ↓

←←test. abs を入力

transmit address = 0017F

Program Completed.

←←書き込みされました。

パソコンの [ESC] キーを押し、H TERM. EXE を終了させます。

⑧ 一度電源を切り、JP2、JP3 のジャンパーピンを取り、再度電源を入れると、`test. mar` が動作します。

`test. mar` は、水晶発振子 X2 (32. 768 KHz) を使い、1 Hz をつくり、その 1 Hz で、ポート 1 を H/L させるソフトです。

テスターなどで、P10 (CN1-20) の電圧を測ると、1 秒ごとに、5 V と 0 V を繰り返します。

■モニターデバッカの使い方■

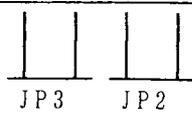
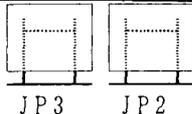
このモニターデバッカは、フラッシュROMにモニタープログラム (monitor.abs) を書き込み、ユーザーソフトをパソコンから、H8/3664のRAM(H'F780~H'FB7F)に転送してデバックします。

- ・パソコン側ソフトは、HTERM. EXEを使用します。
- ・ユーザーソフトはRAM上で動作しますので、RAMアドレスH'F780~H'FB7Fにアセンブラ・リンカーを使って作ってください。(例 testram.abs)
- ・パソコンとの接続は、フラッシュROM書き込みと同じです。

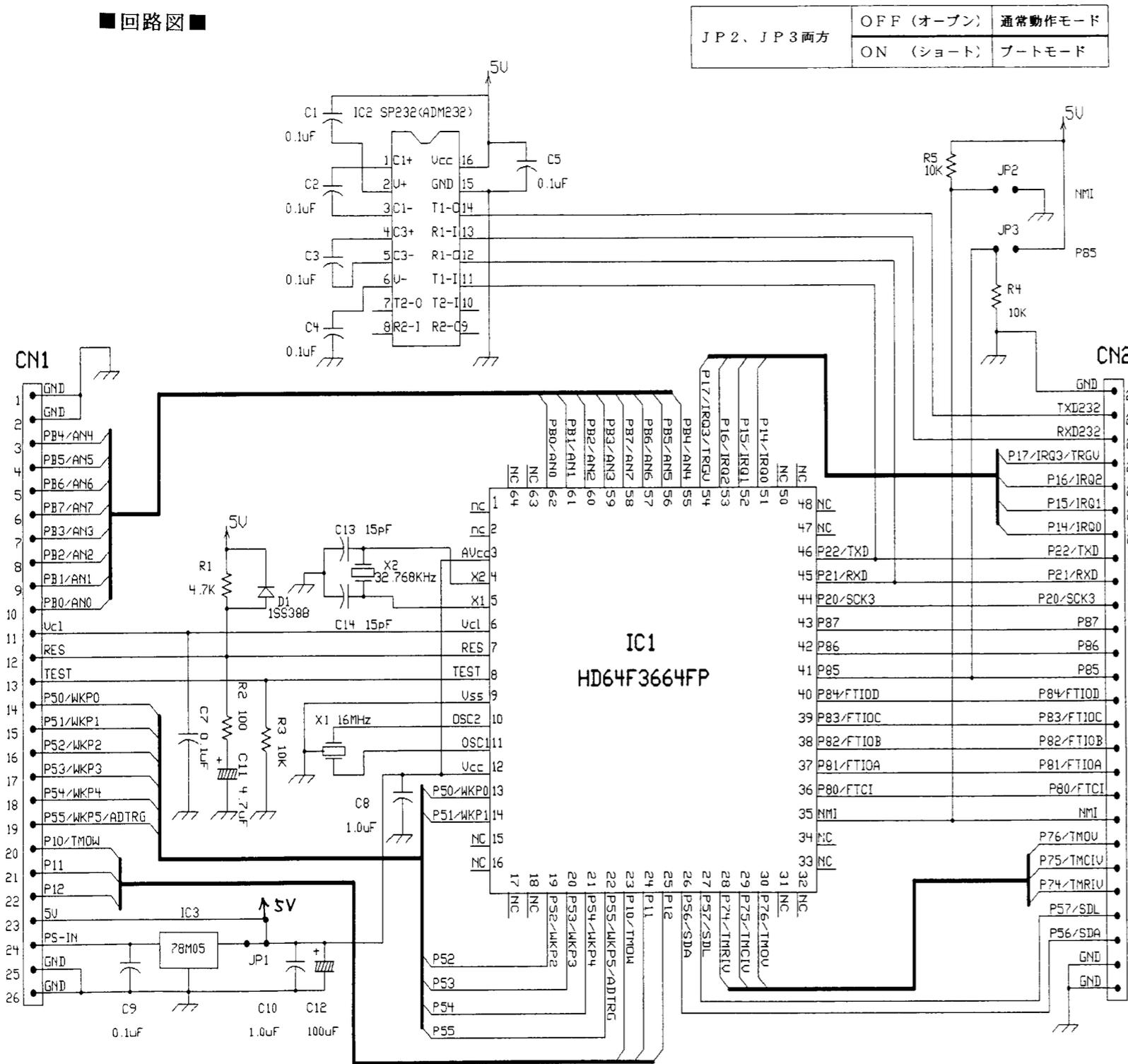
- ①フラッシュROMにモニタープログラム (monitor.abs) を書き込みむ。
先の①~⑦の手順で、monitor.absをフラッシュROMに書き込んでください。
- ②一度H8/3664マイコンの電源を切り、SW1、2をオープンとし、通常動作モードに設定してください。
- ③ターミナルソフトhterm.exeを起動する。(6ページ手順③)
Terminal Program for H Series Monitor Ver. 5.0
- ④H8/3664マイコンの電源を入れる。(すでに電源が入っている場合はリセットスイッチを押しても同じです。) パソコンに次のメッセージが出ます。
H8/3664 Series Normal Mode Monitor Ver. 1.0A
Copyright (C) Hitachi, Ltd. 2000
Copyright (C) Hitachi ULSI Systems Co., Ltd. 2000
:
- ⑤ユーザープログラム (例 testram.abs) をLコマンドでダウンロードします。
:L▲testram.abs↓
transmit address = F790
Top Address=F780
End Address=F791
- ⑥Gコマンドでユーザープログラム (例 testram.abs) を実行します。
:G▲F780↓
モニターデバッカの各コマンドの詳しい使い方は、readme.txtをごらんください。また、モニターデバッカ用のプログラムの作り方は、kakikae.txtをごらんください。

■ブートモードとJP2、JP3の設定■

H8/3664フラットマイコン基板は、基板上のJP2、JP3をOFF (オープン) した状態で「電源をいれる」または「リセットする」と通常動作モードで起動します。JP2、JP3をON (ショート) した状態で「電源をいれる」または、「リセットする」とブートモードで起動します。(ベースボードのSW1を押すとリセットします。) JP2、JP3は付属のジャンパーピンを差すとON、取るとOFFになります。

JP2、JP3	OFF (オープン)		通常動作モード
	ON (ショート)		ブートモード

■回路図■



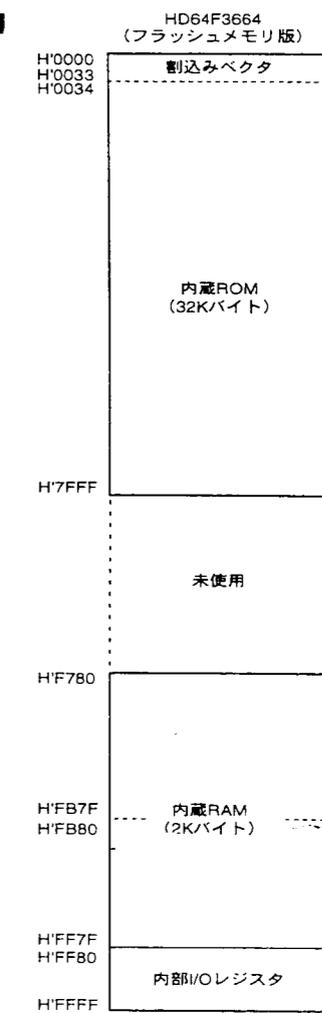
JP2、JP3両方	OFF (オープン)	通常動作モード
	ON (ショート)	ブートモード

■ I/Oなどの注意 ■

- 1、P21、P22は、RS232用IC2に接続されています。P21、P22は、汎用I/Oとしては、使用できません。
P21はRXDとして使用しない場合入力に設定してください。
P22はTXDとして使用しない場合出力に設定してください。
- 2、P85はブートモード設定ピンですので、10KΩでプルダウンされています。
- 3、AVcc (A/D変換アナログ部電源ピン) は、基板上でVCC (5V) に接続されていますので、外部リファレンス電源は使用出来ません。
- 4、P56、P57は、I²Cバスインターフェイス用ピンです。通常のI/O出力して使用する場合は、Hレベル出力電圧が、約2.5Vです。他のI/OとはHレベル出力電圧が違いますので注意してください。

■メモリーマップ■

アドレス空間



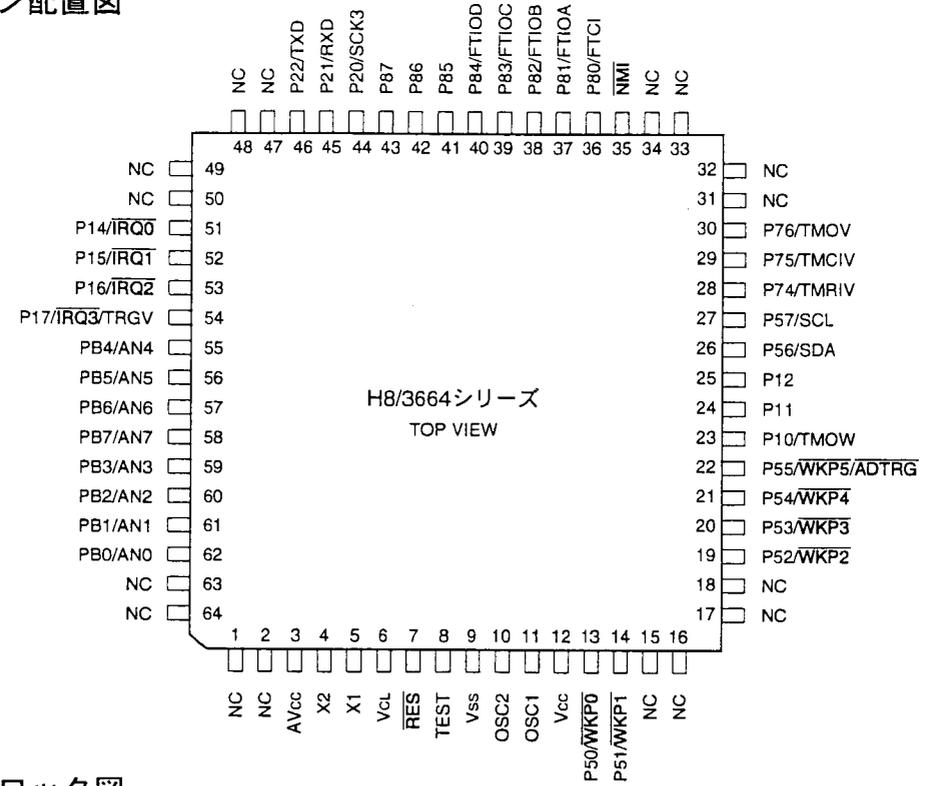
■ブートモードとJP2、JP3■
H8/3664フラットマイコン基板は、基板上のJP2、JP3をOFF (オープン) で「電源をいれる」または、「リセットする」と通常動作モードで起動します。JP2、JP3をON (ショート) で「電源をいれる」または、「リセットする」とブートモードで起動します。JP2、JP3は付属のジャンパーピンを差すとON、取るとOFFになります。ペーボードのSW1を押すとリセットになります。

JP2、JP3両方	OFF (オープン)	通常動作モード
	ON (ショート)	ブートモード

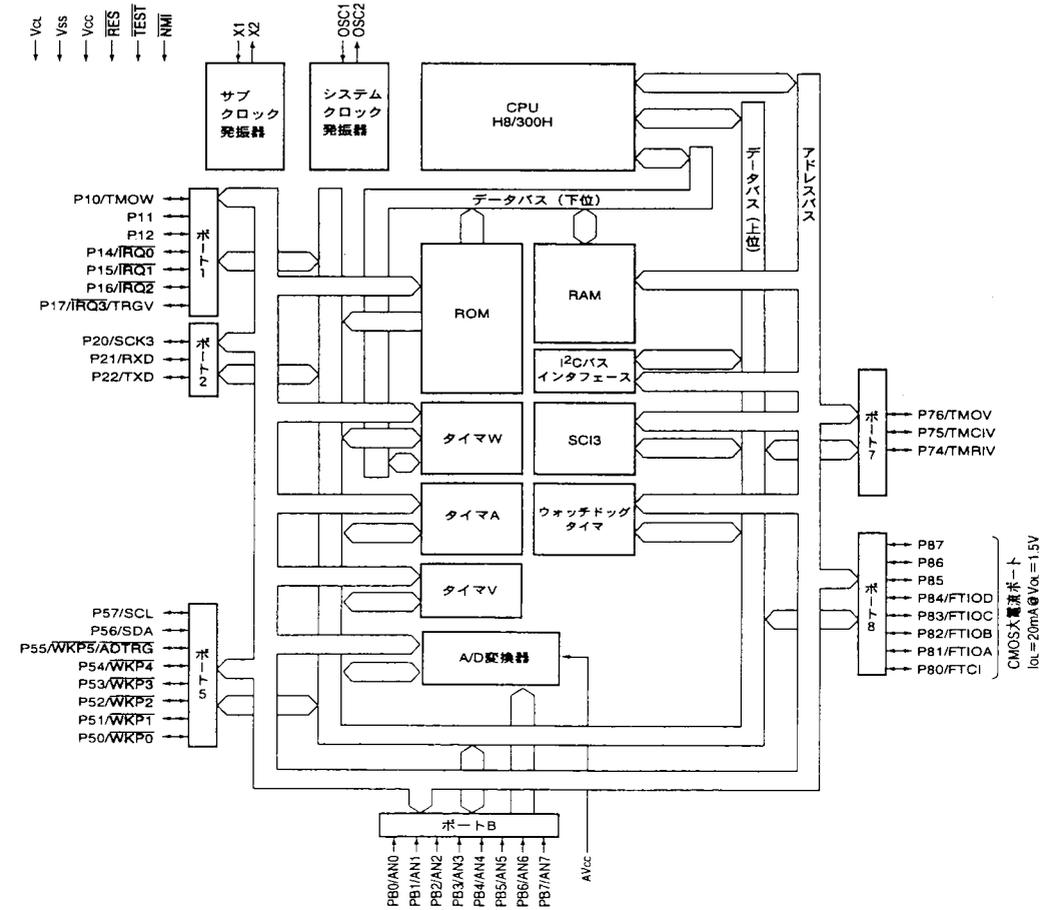
■ H8 / 3664 フラットマイコン基板ピン配置表 ■

CN1	3664ピン番	名称・機能	CN2	3664ピン番	名称・機能
1	9	V _{SS} (GND)	26	9	V _{SS} (GND)
2	9	V _{SS} (GND)	25	—	TXD (RS232Cレベル)
3	55	PB4/AN4	24	—	RXD (RS232Cレベル)
4	56	PB5/AN5	23	54	P17/IOQ3/TRGV
5	57	PB6/AN6	22	53	P16/IOQ2
6	58	PB7/AN7	21	52	P15/IOQ1
7	59	PB3/AN3	20	51	P14/IOQ0
8	60	PB2/AN2	19	46	P22/TXD
9	61	PB1/AN1	18	45	P21/RXD
10	62	PB0/AN0	17	44	P20/SCK3
11	6	V _{CL}	16	43	P87
12	7	RES (リセット)	15	42	P86
13	8	TEST	14	41	P85 (JP2)
14	13	P50/WKP0	13	40	P84/FTIOD
15	14	P51/WKP1	12	39	P83/FTIOC
16	19	P52/WKP2	11	38	P82/FTIOB
17	20	P53/WKP3	10	37	P81/FTIOA
18	21	P54/WKP4	9	36	P80/FTCI
19	22	P55/WKP5/ADTRG	8	35	NMI (JP3)
20	23	P10/TMOW	7	30	P76/TMOV
21	24	P11	6	29	P75/TMCIV
22	25	P12	5	28	P74/TMRIV
23	12	V _{CC}	4	27	P57/SCL
24	—	PS-IN (電源)	3	26	P56/SDA
25	9	V _{SS} (GND)	2	9	V _{SS} (GND)
26	9	V _{SS} (GND)	1	9	V _{SS} (GND)

ピン配置図



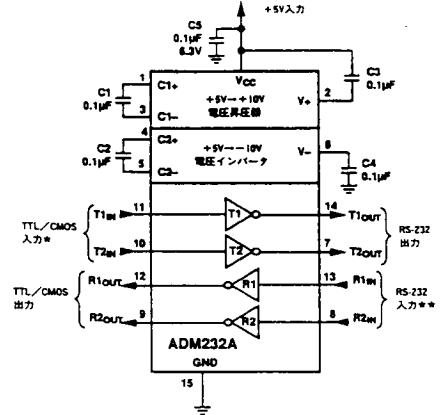
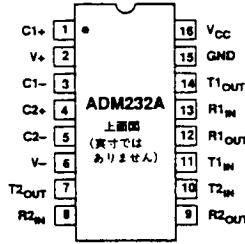
内部ブロック図



高速、+5V、0.1 μ F CMOS RS-232ドライバ/レシーバ ADM232A (SP232)

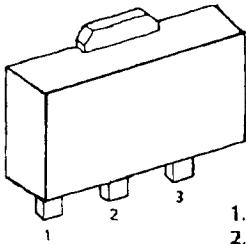
特長

- 200kB/秒の転送レート
- 小容量 (0.1 μ F) 値のチャージ・ポンプ用コンデンサ
- +5V単一電源動作
- EIA-232-EおよびV.28規格に適合
- 2個のドライバと2個のレシーバ
- DC-DCコンバータを内蔵
- +5V電源で $\pm 9V$ の出力振幅
- $\pm 30V$ のレシーバ入力レベル



TA78L05F

3端子正出力固定電圧電源

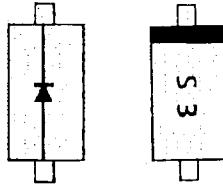


- 出力
- コモン (放熱板)
- 入力

1SS388

ダイオード

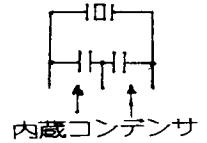
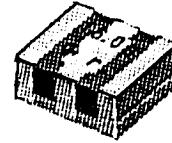
- $V_F (3) = 0.54V$ (標準)
- $I_R = 5\mu A$ (最大)



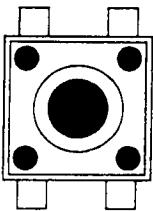
セラミック発振子

■ 負荷容量内蔵型 CSTCVシリーズ

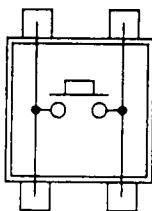
C S C T S 1 6 . 0 M



タクトスイッチ



上面図

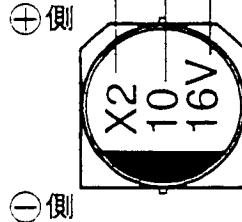


内部接続図

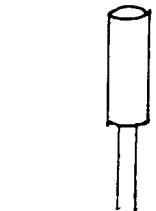
電解コンデンサ

静電容量

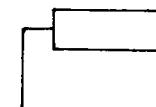
ロットNo. 定格電圧



水晶発振子 (32.768KHz)



金属ケース



AKI-H8/3664FPキット 秋月電子通商 2001/06 byKAKE
 問い合わせは往復はがきまたは返信用切手同封の封書でお願いいたします。
 電話・ファックス・Eメールでのお問い合わせは受け付けておりません。
 ☎158-0095 東京都世田谷区瀬田5-35-6 (株)秋月電子通商 質問係宛

AKI-H8/3664F タイニーマイコン専用 C コンパイラ

H8/3664用Cコンパイラをご購入いただきまして、ありがとうございます。

1、CD内のファイルの内容はつぎのとおりです。

```

CH38    EXE    : Cコンパイラ本体

C38CGN  EXE    : CH38.EXEが使用するコプログラム
C38ASM  EXE    : CH38.EXEが使用するコプログラム
C38FRNT EXE    : CH38.EXEが使用するコプログラム
C38MID  EXE    : CH38.EXEが使用するコプログラム
C38PEP  EXE    : CH38.EXEが使用するコプログラム

C38HN   LIB    : ライブラリ3664Hノーマルモード用

CTYPE   H      : ヘッダーファイル
ERRNO   H      : ヘッダーファイル
FLOAT   H      : ヘッダーファイル
LIMITS  H      : ヘッダーファイル
MATH    H      : ヘッダーファイル
SETJMP  H      : ヘッダーファイル
STDARG  H      : ヘッダーファイル
STDDEF  H      : ヘッダーファイル
STDIO   H      : ヘッダーファイル
STDLIB  H      : ヘッダーファイル
ASSERT  H      : ヘッダーファイル
INDIRECT H     : ヘッダーファイル
NO_FLOAT H     : ヘッダーファイル
MACHINE H      : ヘッダーファイル
3664F   H      : ヘッダーファイル (1/0アクセス)
    
```

```

README.TXT      : このテキストです。
ABOUTC.TXT     : Cについての説明や、3664F、Hの使い方の説明
    
```

3664F、HとABOUTC.TXTは繰り返し参照する事が多いので、プリントアウトする事をおすすめします。

サンプルフォルダには、サンプルソースファイルが入っています。
マニュアルフォルダには、Cコンパイラのマニュアルが入っています。

2、Cコンパイラの動作環境

OS WINDOVS95以降
メモリ 16M以上
WINDOVS内のDOSプロンプト上で動作する。
MS-DOSのみのマシンでは動作しません。

ソースファイルや、リンカー用のサブファイルを製作、編集するには、パソコンで文字を編集するソフト（例えば MS-DOSのエディター、WINDOVSのメモ帳やワードパットなど）をご使用ください。

3、CDのファイルをハードディスクにコピーする。

あらかじめ、ハードディスクにY3664Cというフォルダ（ディレクトリ）を作りその中にCDの中身をコピーしてください。
リンカー LNK.EXEをAKI-H8/3664キットに付属のCDからコピーしてください。

.....
 ・(注意)フォルダ名(ディレクトリ名)は小文字10文字以内にしてください。
 ・長いとコンパイル時にエラーになります。
 ・また、フォルダは C:\Y3664C の様にドライブの先頭(ルート)においてください。
 ・ROM化の為のリンカー、アセンブラ等はこのCDには付属していません。AKI-H8/3664キットに付属しています。
 ・それをご使用ください。

4、ROMライター用のファイルを作る例 CTEST.C

(あらかじめサンプルのCTESTフォルダ内のファイルをCH38.EXEとのあるフォルダC:\Y3664Cにコピーしてください。)

リセットベクトル、スタックポインタ、割り込みベクトルはアセンブラで書く必要があります。3664RES.MARがそれです。3664RES.MARからつくった3664RES.OBJをリンカーでCのオブジェクトファイルと合体させます。
Cのmain関数は3664RES.OBJが呼び出します。
3664RES.MAR、3664RES.OBJはこのDISKのSAMPLE内に入っています。

- (1) CコンパイラCH38.EXEを使い、CTEST.CからCTEST.OBJをつくる。
MS-DOSのコマンドラインで次のようにキー入力するとCTEST.OBJが作られます
CH38.EXE -INCLUDE=C:\Y3664C -CPU=300HN CTEST.C
同時にリストファイルctest.lstが作られます。
CTESTC.BATがこの内容ですのでそれをそのまま実行しても同じです。
- (2) リンカーLNK.EXEを使い、CTEST.OBJと3664RES.OBJからCTEST.ABSをつくる。
リンカーのサブコマンドとしてCTEST.SUBを指定する。
LNK.EXE -SUB=CTEST.SUB
CTESTL.BATがこの内容ですのでそれをそのまま実行しても同じです。
- (3) 生成されたCTEST.ABSをライターで書き込みます。

5、パッチファイルの説明

パッチファイルはMS-DOSで使用する物で、キーボードから入力し実行させるソフト（例えばCコンパイラ CH38.EXE）などを、あらかじめその中に書いておきパッチファイルを実行することでキーボード入力と同じ動作をするものです。

パッチファイルを使用すると、インクルードファイルの指定などを、そのつど入力する必要がなく、便利です。

(I) Cコンパイラ用パッチファイル

パッチファイルCTESTC.BATの内容は次のようになっています。

```
CH38.EXE -INCLUDE=C:\Y3664C -CPU=300HN CTEST.C
```

このパッチファイルは、CTEST.Cをコンパイルする専用のもので、お客さまが製作されたソースファイル（例えばYUUA.C）に応用するには、次の様に内容を書き替えたファイル（YUUA.C.BAT）を新に作ってください。

YUUA.C.BATの内容例

```
CH38.EXE -INCLUDE=C:\Y3664C -CPU=300HN YUUA.C
```

(II) リンカー用パッチファイル

リンカーはサブコマンドとしてサブファイルを指定しますので、YUUA.Cに応用するには、

リンカー用パッチファイルの内容
サブファイルの内容
を書き替えたファイルを作ってください。

YUUA.C.BATの内容例

```
LNK.EXE -SUB=YUUA.SUB
```

YUUA.SUBの内容例

```
OUTPUT YUUA  
PRINT YUUA  
INPUT 3664RES, YUUA  
LIB c38hn  
START P(100)  
EXIT
```

6、リンカー用サブファイルの説明

（AKI-H8/3664Fタイニーマイコンキット付属のリンカーの説明H8SWANをあわせて、ご覧ください。）

リンカーはサブコマンドとしてサブファイルを使用することで、入力するオブジェクトファイル(.OBJ)の指定や、プログラムの開始アドレスを指定します。

----CTEST.SUBの説明----

```
OUTPUT ctest      :CTEST.ABSを出力ファイルとする。  
PRINT ctest       :CTEST.MAPを出力する。  
INPUT 3664RES, ctest :3664RES.OBJ, CTEST.OBJを入力ファイルとする。  
LIB c38hn         :ライブラリファイルはC38HN（ノーマルモード用）  
START P(100)     :この行はメモリの開始アドレスを指定します。  
EXIT              :SUBファイルの終了
```

開始アドレス指定は、領域ごとに指定します。

領域には次の領域があります。

プログラムによっては、上の例のようにプログラム領域だけの場合があります。

- P プログラム領域（main関数などプログラム本体）
- C 定数領域（CONST型のデータ）
- B 未初期化領域（RAM領域）
- D 初期化データ領域

START P(100), C(2500), B(0F780) の様に指定します。

START P, C(100), B(0F780) の様に指定すると定数領域 Cはプログラム領域 Pに続いて割り当てられます

各領域の大きさはCコンパイラが出力するリストファイルをご覧ください。

■（株）日立製作所のホームページ内TinyマイコンホームページにH8/300H Tinyシリーズ アプリケーションノートが掲載されています。C言語でソフトを記述していますので ご参考にしてください。

■本CD内のCコンパイラマニュアルはPDF形式のファイルになっています。パソコンで、表示、印刷するには、Acrobat Readerが必要です。Acrobat Readerは www.adobe.co.jpでダウンロードできます。

この文書では、H8-3664F 用 C コンパイラを使ってプログラムを作るために必要な C の基礎知識について説明します。H8 のプログラムを作るために必要な最小限の内容ですから、カーニハンとリッチーの「プログラミング言語C」などと一緒に読んでいただいたほうがよいかも知れません。

1、プログラムのおおまかな構造

C は、関数を並べてプログラムを作ります。そのなかでも、main() は必ず必要な関数です。ここから処理が始まります。

まずは「hello, world」を出力するプログラムを見てみましょう。main という関数が定義されています。

```
#include (stdio.h)
main()
{
    printf('hello, world\n');
}
```

最初の行 #include (stdio.h) は、ここで stdio.h という名前のファイルを読み込む（インクルードする）ことをあらわします。読み込まれるファイル stdio.h には、入出力関数を使うために必要な情報が書き込まれています。main() ので使われている関数 printf() は文字を出力するための関数です。一般的に、関数は名前のあとに () をつけて表します。このカッコの中には、その関数に渡すパラメータを書いておきます。この例の場合の、関数 printf() に対するパラメータは

```
'hello, world\n'
```

という文字列です。最後の「\n(\n)」は改行をあらわします。

関数 printf() は、すでにだれかが作ってくれてライブラリに登録されていますが、自分で作った関数を使うときも、この printf() を使うのと全く同じように記述します。（ライブラリの関数と、自分で作った関数の区別がつかないということです）このように C では、プログラムを関数に分けて、その関数を呼ぶことで処理が進んでいきます。

printf() の前の空白や、「{」「}」の位置は自由にきめてかまいません。ただし引用符「」で囲まれた文字列の中で、行をかえることはできません。

実際にコンパイルしてみると、コンパイラは問題ありませんが、リンカでエラーが出ました。画面に文字を出力したり、ファイルアクセスのための ANSI 標準ライブラリ関数は H8/3664F C コンパイラでは使わないようにしましょう。

2、ループと分岐

条件分岐の if 文や、switch、それにループを構成する do - while などです。

2.1 ループ (for, while, do - while)

for 文の例を、これまた有名な温度換算プログラムを使って示します。

```
/* 温度換算プログラム */
```

```
#include (stdio.h)
main()
{
    int fahr;
    for (fahr = 0; fahr (<= 300; fahr += 20) {
        printf('%3d %6.1f\n',
            fahr, (5.0 / 9.0) * (fahr - 32));
    }
}
```

最初の行は注釈行です。コンパイラは /* と */ で、挟まれた部分を無視します。注釈は、複数行にまたがってもかまいません。C++ スタイルの注釈で使われる「//」は、H8/3664F C コンパイラではエラーになります。

main() の中の最初の行 int fahr; は、整数型 (int) で、fahr という名前の変数を宣言しています。この例のように、C の変数は使う前に宣言しておく必要があります。変数の型には int のほかにいくつかが用意されています。整数のバイト数はコンパイラによって異なる場合があります。ここで示した整数のバイト数は H8/3664F C コンパイラの場合です。limits.h にそれぞれの型に対する最大・最小値が決められています。

char	文字 (1バイト)
int	整数 (2バイト)
short	整数 (2バイト)
long	整数 (4バイト)
double	浮動小数点数 (?バイト)

次の for 文は繰り返しの処理で使います。この例の場合では fahr を 0 から 300 まで、20 ずつ増やしながらループしています。fahr += 20; は、fahr = fahr + 20; の意味です。

for 文の一般形と、for と同様の繰り返し処理を記述するのに使われる while 文の一般形を示します。この2つは同じ処理になります。まず式1が処理され、式2を処理し、その結果が真 (0 以外) であれば分を実行し、最後に式3が処理されます。式2は x (y などの関係式で、式1と3は代入文や関数呼び出しになります。

```
(( for 文))
for (式1; 式2; 式3)
    文
```

```
(( while 文))
式1;
while (式2) {
    文
```

式3、

もう一つの繰り返し処理 do - while は次の形になります。

```

((do - while))
do
  文
while (式);

```

do - while は、文が実行されてから終了条件の式を評価します。前の for と while は、式を評価してから文が実行されましたから、順番が逆です。必要に応じて使い分けましょう。

for の後の文と、do と while の間の文がひとつであれば {} は必要ありません。しかし、文が追加されることがよくあります。このとき、{} を付けないままにしておく、思わぬ処理内容になってしまうというバグを発生させますから、文が一つでも {} を付けるようにしておいたほうがよいでしょう。

たとえば、次のプログラムは for ループの中に y += x; と z *= x; の2つの文が入っているように見えますが、y += x; だけが繰り返し処理されて、ループが終了したところで z *= x; が一度だけ処理されます。

```

y = 0;
z = 0;
for (x = 0; x < 10; x++)
  y += x;
  z *= x; /* ここはループの外 */
printf("%d, %d\n", y, z);

```

2. 2 分岐 (if, if - else, if - else if, switch)

ある判定をして、その結果で処理を変えるときには if 文を使います。その一般的な形は次のようになります。

```

if (式)
  文1
else
  文2

```

if 文は、式を評価しそれが真 (0でない) なら文1を、偽 (0) なら文2が実行されます。else 以下は省略可能です。

もし x が 1 なら式1、2 なら式2、3 なら式3、それ以外なら式4、といった場合は else

= if を使います。

```

if (x == 1) {
  式1;
} else if (x == 2)
  式2;
} else if (x == 3) {
  式3;
} else {
  式4;
}

```

判定式 x == 1 は、x が 1 と等しければ真 (0以外)、等しければ偽 (0) になります。よくやる間違いで、x == 1 とするところを、x = 1 と書いてしまうことがあります。C では式も値を持ちますから x = 1 自体は値 1 になり、文法上は問題ありません。しかし、x = 1 は比較ではなく代入文ですから、x の値が 1 になってしまい、思わぬ結果をもたらします。最近のコンパイラは判定式を書くべきところで代入していると警告を出してくれますが、この H8/3664F C コンパイラは何も文句はいりません。

if と else の間に {} を置いています。式が一つだけであれば省力可能です。でも、つけておいたほうが無難です。

同じ変数に対して、いくつかの定数値と比較し分岐する場合は switch 文を使います。switch 文の一般的な形は次のようになります。式の値が定数式と等しいところの文が実行されます。どれにも当てはまらない場合は default の文が実行されます。break や文はなくてもかまいません。break が無いと、次の case へ処理が進みます。break があると switch を抜けます。

```

switch (式) {
  case 定数式1:
    文1
    break;
  case 定数式2:
    文2;
    break;
  default:
    文3;
    break;
}

```

先ほどの例を if ではなく switch を使って表現してみましょう。もし x が 1 なら式1、2 なら式2、3 なら式3、それ以外なら式4 を実行します。

```

switch (x) {
  case 1: /* x が1のときの処理 */

```

```

    式1;
    break;
case 2:    /* x が2のときの処理 */
    式2;
    break;
case 3:    /* x が3のときの処理 */
    式3;
    break;
default:  /* x がそれ以外のときの処理 */
    式4;
    break;
}

```

もし x が1か2なら式1, 3なら式3、それ以外なら式4を実行するという場合は、次のようになります。

```

switch (x) {
    case 1:    /* x が1のときの処理 */
    case 2:    /* x が2のときの処理 */
        式1;
        break;
    case 3:    /* x が3のときの処理 */
        式3;
        break;
    default:  /* x がそれ以外のときの処理 */
        式4;
        break;
}

```

3、関数

プログラムをいくつかの関数に分割します。それらの関数が集まって、プログラムになります。最も簡単な関数は次のような形になります。

```
smallFunc() {}
```

関数名 `smallFunc` の後ろにパラメータを記入するカッコ、そして関数の内容を記述する () が続きます。この場合は、パラメータが無く、内容も無い関数になっています。

具体的なプログラムを示して説明します。このプログラムはいよいよ H8/3664 で動作するものです。シリアルポート (SC13) から1文字読み込んで、シリアルポートから出力するというものです。その動作は、パソコンのターミナルソフトでチェックできます。`main()` と `inch()` だけを残して、後は説明の邪魔になるのでよけました。(後ほど全リストを示します)

`include` が3つあります。このプログラムをコンパイルするために必要な情報が、ここで読み込まれます。`3664f.h` は、H8/3664F の I/O ポートなどの定義をしているコンパイラ付属のインクルードファイルです。ただし、ANSI 標準ライブラリではありませんから、C の本を見てもその説明はありません。使い方は難しいのですが、これを使うとスマートに I/O ポート操作を記述することができます。

`ctype.h` は、よく使う型の定義をしています。これもコンパイラ付属のインクルードファイルです。

`myType.h` は、このプログラム独自のインクルードファイルで、`SInt16` などの、自分で作成した型の定義をしています。

```

#include '3664f.h'
#include 'ctype.h'
#include 'myType.h'

```

```

/* PROTOTYPE */
void initio(void);
SInt16 inch(void);
SInt16 inchxx(void);
void outch(short ch);
void out6h(UInt32 xx);
void out4h(UInt16 xx);
void out2h(UInt8 xx);
void out8h(UInt32 xx);
void outh(UInt8 xx);
void crlf(void);

```

突然出てきた関数プロトタイプは、このファイルの中で使われる関数を前もって宣言しておくものです。

そして `main` 関数です。みなれない `void` で囲まれています。`main` の左側の `void` は、この関数 `main` は `void` 型の値を返すということを意味していますし、`main` の右側のカッコの中にある `void` は、`void` 型のパラメータひとつを必要としているということになるんですね、一応。で、なぜ一応かというと、`void` というのは特別な型で、`void` 型を返すということは何も返さないし、`void` 型のパラメータを必要としているということは、パラメータを必要としない、という、ちょっとひねくれた表現なのでした。

```

void main(void)
{
    SInt16 ch;

    initio();
    crlf();
    outch('*');
}

```

```

do {
    ch = inch();
    outch(ch);
} while (true);
} /* main */

```

前にもいいましたように、このプログラムはシリアルポート (SC13) から1文字読み込んで、シリアルポートから出力するというものです。実際には、読み込んだ文字を大文字に変換して、出力しています。

それでは、main() の中を順番に見ていきましょう。

まず、この関数で使う変数の宣言です。Sint16 型の変数を一つ宣言しています。その変数の名前は ch です。

```
Sint16 ch;
```

文字なのになせ char 型ではなく、Sint16 という16ビットの変数になっているのでしょうか。unicode だからというわけではありません。通常、ループの終了をチェックするための文字の終わり (ファイルの終わり) を表す EOF (End Of File) が、char 型では表現できないのです。ここでは EOF チェックをしていませんから、関係はないのですが、入力する文字は、8ビットの char ではなく、16ビット型の変数を使いましょう。

次に3つの関数が並んでいます。I/O の初期化をして、改行を出力してから、プロンプト (*) を出力します。C の伝統では、関数名は小文字、定数は大文字となっていますが、そんなこと気にせず initio は Initio、init_io などとしてもかまいません。大切なのは、スタイルを統一するということです。

```

initio();
crlf();
outch('*');

```

次は do - while ループです。while のカッコの中が true (真) になっています。do - while は、while のカッコの中が真であるあいだはループしつづけますから、これは結局、無限にループをくり返すということです。

ループの中は、一文字入力して、それを出力しています。

```

do {
    ch = inch();
    outch(ch);
} while (true);

```

次にもう一つの関数 inch() を見てみましょう。

```
Sint16 inch(void)
```

です。パラメータなしで、Sint16 型の値を返す関数です。Sint16 というのは、符号付き16ビット整数の意味です。自分で作って myTypes.h で宣言してあります。ほかにもご想像のとおり、UInt16 とか、Sint32、Sint8 などがあります。こうやって、自分につこうのいい型を作ることできます。

```

Sint16 inch(void)
{
    Sint16 ch;

    ch = inchxx();
    if (islower(ch)) {
        ch = toupper(ch);
    }
    return (ch);
} /* inch */

```

さて、inch() の内部に目をやると、先ほどの main() と同じ Sint16 型の変数 ch を宣言しています。同じ型同じ名前なのですが、main() の中で使われる変数 ch とは別のものです。{|} の中で宣言された変数は、その外の変数とは独立しているのです。

```
Sint16 ch;
```

次の4行は、関数 inchxx() の返した値が小文字なら大文字にする、というところです。islower() と toupper() は、コンパイラ付属のインクルードファイル ctype.h で宣言されている ANSI 標準ライブラリ関数です。これは、書店で販売されている C の教科書に、説明が載っているということの意味します。

inchxx() は自分で作った関数で、islower() と toupper() は、はじめからライブラリに用意されている関数だなんて、ちょっと見ただけでは (よく見ても) 分かりません。ある関数があったら、その関数がどこにあるのか探せる環境を作ってください。

islower() と toupper() は、ANSI 標準ライブラリ関数です。この関数のプロトタイプは、このファイルに含まれていません。では、いったいどこにあるのでしょうか。はい、それは、最初にインクルードした ctype.h に含まれるのです。toupper を検索して、それが ctype.h にあるということがすぐにわかるようにしておきましょう。

たとえば、ゲームを作るときに必ず必要となる乱数を発生する関数 rand() をプログラムの中で使ったとします。その rand() のプロトタイプ宣言が含まれるヘッダファイルを、そのプログラムの最初でインクルードしなければなりません。rand() を使うためには、その関数がどのヘッダファイルで宣言されているのか調べなければならないのです。

話は戻りまして、inch() の中の話です。関数 inchxx() の返した値が小文字なら大文字にするというところです。inchxx() は、SC13 から一文字入力し、その文字を返す関数です。その中身については、最後にソースリストを載せておきましたので、御参照ください。ii

文の判定式部分にある `islower(ch)` は、渡した値が小文字なら真になりますから、`ch` が小文字なら `if` 文の中が実行され、`ch` を `toupper(ch)` の値にします。`toupper(ch)` は、`ch` を大文字にした値を返します。

```
ch = inchxx();
if (islower(ch)) {
    ch = toupper(ch);
}
```

最後の `return (ch);` は、関数のように見えますが、関数ではありません。実はカッコは不要なのですが、なんとなくカッコを付けたくなるんですね。ここは、この関数が `ch` の値を返すということです。

```
return (ch);
```

ま、このように、プログラムを「I/O を初期化するところ」「一文字入力するところ」「一文字出力するところ」などに分けたものを関数にして、その関数を組み合わせてプログラムを作っていきます。

プログラムをわかりやすくするために一つの関数の行数はすくなめのほうがよいようです。たとえば一つの画面に表示されるくらいとか、プリンとして1ページくらいなど・・・

このプログラムの全リストはこの文書の最後に載せておきます。

4、変数の型

4、1 型の種類

いままでいくつかの変数の型が出てきました。そもそも C に用意されている型は次のものがあります。

<code>char</code>	1バイトの文字
<code>int</code>	整数
<code>float</code>	単精度浮動小数点数
<code>double</code>	倍精度浮動小数点数

さらに修飾子として、`short` と `long`、さらに `signed` と `unsigned` があります。HB/3664F C コンパイラでは `int` と `short int` は16ビット、`long int` は32ビットです。ここで、`short int` や、`long int` は通常 `int` が省略されて `short` と `long` となります。`short char` や、`long double` がサポートされているか実験してみましょう。

4、2 型変換（キャスト）

変数には、`int` とか `char` とかの型があります。ま、アセンブラ的に考えると、型というのは、変数のサイズのようなものです。（同じサイズでも型が違うということもありますから、この説明は正確には間違いです。）

サイズが異なる変数間で代入が行われる場合、型変換（キャスト）が発生します。自動的にコンパイラが型変換してくれるときと、エラーになる場合があります。故意に型変換を行いたい場合には、キャスト演算子を使います。

型変換が必要などときには変換したい式にキャスト演算子をつけます。キャスト演算子は、変換先の型をカッコで括ったものです。例を次に示します。ここでは、`int` 型の変数 `xx` を型変更し、`char` 型にしています。

構造体を `int` に変換する、という場合のように、キャスト演算子をつけても変換できず、エラーになる場合もあります。

```
int xx;
char yy;
```

```
xx = 100;
yy = (char)xx;
```

4、3 定数

定数は次のように宣言します。この例では、`kMaxNumber` が 200 に、`kTestString` は "Test String" に、`kLongVal` は 40000 になります。

40000L の L は、`long` の L で、長整数型の文字列として扱われます。

```
#define kMaxNumber 2000
#define kTestStr "Test String"
#define kLongVal 40000L
#define kCRcode 0x0d
#define kTABcode '\t'
```

アセンブラで使われる16進数は `0x` あるいは `0X` を付けてあらわします。たとえば、アセンブラでの16進数 `h'12ab` は、`0x12ab` となります。`0x12abl` とすると、長整数型（H8/3664のCコンパイラでは32ビット）になります。この例では、`kCRcode` が16進数です。

文字列と文字は区別されます。2重引用符「`''`」で囲まれていれば文字列、単一引用符「`'`」であれば文字です。文字は数値なのですが、文字列は（後で説明しますが）ポインタですから、普通の値のように扱うことはできません。

`'\t'`（`'\t'`）はタブコードをあらわします。このほかにも改行 `\n` などがあります。

列挙定数というものもあります。定数なんですが、ひとつひとつの値自体は何でもかまわないという時に使います。

たとえば、真と偽を意味する `true` と `false` です。

```
enum { false, true };
```

こうすると、`false` は0に、`true` は1になります。

次の例のように、値を指定することもできます。`shirobon` は、`kurobon + 1` になります。`aoobon` は 100 です。

```
enum { kurobon = 1, shirobon, akabon, aobon = 100 };
```

定数のように #define を使うマクロがあります。簡単な関数をマクロで表現することができます。マクロを使った関数は、実際に関数ではありません。マクロを使った場所で展開されます。三項演算子の説明を参照してください。

4. 4 演算子

2項演算子には +、-、*、/、% があります。% は余りです。7 % 3 は 1 になります。

比較などに使われる >、<、>=、<=、==、!= もあります。

C++ という具合に、言語の名前にもなってしまったインクリメントとデクリメント演算子は、C++ であれば、C = C + 1 と同じ意味です。C += 1 と書くこともできます。

```
x = 0;
y = 5;
z = x++ + --y;
```

さて、x、y、z はどんな値になっているのでしょうか。注意するところは、インクリメント演算子とデクリメント演算子が、変数のどちらについているかということです。x++ のように、右側についている場合は、x の値を参照してから x を更新します。左側であれば、更新してから、その値を参照します。ですから、z = x++ + --y; は、z = 0 + 4; となって、z は 4 になります。そして、z に代入後、zx は 1 に、y は 4 になります。

アセンブラ的な演算子として、ビット演算子があります。

```
&   ビットごとの AND
|   ビットごとの OR
~   ビットごとの Ex-OR
<<  左シフト
>>  右シフト
~   1 の補数
```

あまり好きではないのですが、三項演算子 ? : があります。if 文の代わりですが、複数行にわたる if 文が、一つの式で表現できてしまいます。たとえば、前に出てきた大文字に変換する式は次のようになります。

```
upch = islower(c) ? c - 'a' + 'A' : c;
```

一般的な形は次のようになっています。式 1 が真であれば式 2 が、偽であれば式 3 が全体の式の値になります。

式 1 ? 式 2 : 式 3

先ほどの大文字変換の式を if 文に直すと次のようになります。

```
if (islower(c)) {
    upch = c - 'a' + 'A';
} else {
    upch = c;
}
```

定数のように #define を使うマクロについて説明しておきます。簡単な関数をマクロで表現することができます。マクロを使った関数は、実際に関数ではありません。マクロを使った場所で展開されます。なんだか分からないと思うので、例を示します。

```
#define toupper(c) (islower(c) ? (c) - 'a' + 'A' : (c))
```

先ほどの、大文字に変換する式をマクロ関数にしました。こうすると、toupper() を普通の関数のように使うことができます。ちょっと注意が必要なんですが・・・この定義を見て、余計な () が増えたことにお気づきでしょう。これは、() が無い場合に、c に相当する部分が、複雑な式だったりすると、マクロを展開したときに意味が変わってしまうことがあるためです。自分でマクロを作るときには注意しましょう。

また、マクロにはもう一つ注意しなければならないことがあります。次の式はどのようなでしょう。

```
x = 'a';
y = toupper(++x);
```

toupper() が純粋な関数であれば y は 'B' になるはずですが、ところが、マクロですから展開されると次のようになります。

```
x = 'a';
y = (islower(++x) ? ((++x) - 'a' + 'A') : (++x));
```

y がどんな値になるか考えてみましょう。++x が一箇所のように見えますが、マクロが展開されると 3 箇所にもなっています。

5. 構造体と共有体 (struct, union)

良く使います。これが分からないと、プログラムが作れません。

H8/3664F の I/O ポートの定義をしている 3664I.h をちょっと見てください。長いので途中まで次に示します。

```
struct st_sci3 {
    union {
        unsigned char BYTE;
        struct {
            /* struct SCI3 */
            /* SMR          */
            /* Byte Access */
            /* Bit Access  */

```

```

unsigned char COM :1;      /* COM    */
unsigned char CHR :1;      /* CHR    */
unsigned char PE  :1;      /* PE     */
unsigned char PM  :1;      /* PM     */
unsigned char STOP:1;      /* STOP   */
unsigned char MP  :1;      /* MP     */
unsigned char CKS :2;      /* CKS    */

```

このように、struct と union の嵐となっています。

5. 1 構造体 (struct)

パラレルポートを考えましょう。ポート1は8ビットのレジスタ2つから構成されます。ひとつはポートコントロールレジスタ (PCR1) で、もうひとつはデータレジスタ (PDR1) です。

それを (36641.h の定義とは異なりますが) 構造体で表現すると次のようになります。#define のところは、こーゆーもんだと思っていてください。

```

struct st_io {
    union {
        unsigned char PDR1;
        unsigned char PCR1;
    };
#define IO (*(volatile struct st_io *)0xFFD0) /* IO Address*/

```

define 文と組み合わせることで、ポート1を使うにはこんなふうにします。PCR1 に 0x1f を書き込んで、8ビットすべてを出力ポートにして、PDR1 を0にすることで、ポート1から0を出力します。

```

IO.PCR1 = 0x1f;
IO.PDR1 = 0;

```

このように、構造体を使うといくつかの変数をまとめて扱うことができるのです。

構造体の中の変数をメンバーといいます。

よく、例として使われるのが、座標情報です。2次元平面での x y 空間での点を次のように表現できます。

```

struct point {
    int h;
    int v;
};

```

このままなら、こんなふうに使います。point 構造体の変数 pt を用意し、座標を { 100, 250 } にして、線をひく関数 lineno() に渡しています。

```

void lineno(struct point pt); // プロトタイプ宣言

```

```

void lineno(struct point pt)
{
    /* それなりの処理 */
}

```

```

main()
{
    struct point pt;

    pt.h = 100;
    pt.v = 250;
    lineno(pt);
}

```

struct といちいち書くのはわずらわしいので、通常は次のように typedef と組み合わせます。

```

typedef struct {
    int h;
    int v;
} point;

```

```

void lineno(point pt); /* prototype */

```

```

main()
{
    point pt;

    pt.h = 100;
    pt.v = 250;
    lineno(pt);
}

```

```

void lineno(point pt)
{
    /* それなりの処理 */
}

```

構造体は、C とならぶ構造化言語として有名な Pascal ではレコード型といいます。住所録や、会員名簿の個人データなど、いくつかの項目 (フィールド) がひと組のデータとして扱われるときに使われるのです。

ひとつの構造体中のデータが、全て同じ型であれば配列でも表現できますが、構造体は、ひとつひとつの項目の型が異なってもかまいません。

構造体を使うと便利ことがあります。それは、修正が簡単だということです。たとえば、先ほどの、平面座標を表す Point 構造体です。これは水平座標値 *h* と、垂直座標値 *y* という2つの値を持っていました。

さて、これを修正します。座標情報に、色を追加するのです。構造体を使っていれば、構造体の定義に色情報を追加し、色の処理を追加します。関数のパラメータは Point 構造体を渡すということで、修正は必要ありません。

もし、線を引く関数 *lineo* が構造体ではなく、次のように水平座標と垂直座標をバラバラに渡していたとします。

```
void lineo(int h, int v);
```

これに色を渡すように修正するので、

```
void lineo(int h, int v, int color);
```

と、なります。これだけなら簡単ですが、プログラムの中から *lineo* を使っているところを全てさがし出して、*color* を追加しなければなりません。これはたいへんです。

構造体を使っていれば、*color* を追加する前とあとは次のようになります。

<<追加前>>

```
void lineo(Point pl);
```

<<追加後>>

```
void lineo(Point pl);
```

同じなんです。ただし、構造体の宣言が変わります。

```
typedef struct {
    int h;
    int v;
    int color;
} point;
```

5.2 共有体 (union)

さて、次は共有体です。

共有体は構造体と見かけ上ほとんど変わりありませんが、大きな違いがあります。構造体では、各メンバーに独立した領域が確保されますが、共有体では、すべてのメンバーが領域を共有するのです。

何の目的でこんなものが考えだされたのかというと、そもそもはだいたい同じ項目の並んでいる構造体があって、その異なる部分を共有体にするすることで、記録に必要な領域を減らそうとしたのです。

アセンブラに近いプログラムを作る場合にでてくる共有体はちょっと違います。つぎの共有体を見て下さい。

```
union charToShort {
    char ch[2];
    short sh;
};
```

こうしておいて、次のプログラムで何がプリントされるでしょう。

```
main()
{
    union charToShort xx;

    xx.sh = 0x1234;
    printf("%02x %02x\n", xx.ch[0], xx.ch[1]);
}
```

16ビット変数の *xx.sh* に *0x1234* が代入され、次に、*xx.ch[0]* と *xx.ch[1]* をプリントしています。H8/3664F では、上位桁がメモリの最初に格納されていますから、*xx.ch[0]* は *xx.sh* の上位バイト、すなわち *0x12* が、*xx.ch[1]* には *xx.sh* の下位バイト、*0x29* が入っているのです。

このように、ある領域を16ビットデータとしてあつかったり、8ビットデータとしてみるのが、共有体を使うことで、可能となるのです。

H8/3664F の C コンパイラでは、H8 の I/O 領域の定義に共有体をたくさん使っています。これは、レジスタが16ビットアクセスのほうがつごうが良かったり、8ビットや、1ビットごとのアクセスのほうが便利だったりするためです。

3664I.h で定義されているパラレルポート P1 の構造体を、次に示します。

先ほどの例では2バイト目に PDR1 がありましたが、それが共有体になっています。共有体のメンバーは、1バイト単位の *unsigned char BYTE* と、各ビットごとの構造体です。このようにすると、1バイトまとめて入出力するときは *BYTE* を使い、ビット単位なら *BIT* 構造体を使えばよいのです。

BIT の中にでてくる *:1* は、ビットフィールドということで、1ビット単位でアクセスするメンバーです。

```
struct si_io {
    /* struct IO */
    /*
    途中省略
    */
    union {
        /* PDR1 */

```

```

unsigned char BYTE; /* Byte Access */
struct { /* Bit Access */
    unsigned char 87:1; /* Bit 7 */
    unsigned char 86:1; /* Bit 6 */
    unsigned char 85:1; /* Bit 5 */
    unsigned char 84:1; /* Bit 4 */
    unsigned char 83:1; /* Bit 3 */
    unsigned char 82:1; /* Bit 2 */
    unsigned char 81:1; /* Bit 1 */
    unsigned char 80:1; /* Bit 0 */
} PDR1;
/*
途中省略
unsigned char PCR1; /* PCR1 */

```

こんなふうに使います。まず、コントロールレジスタ (PCR1) に 0x11 を書き込んで、ポート 1 を出力ポートに設定します。次に、データレジスタ (PDR1) の 8 ビット全体に 0 を出力し、次に、bit 7 を 1 にしています。

```

IO.PCR1 = 0x11; /* bit 7..0 : Output */
IO.PDR1.BYTE = 0; /*
IO.PDR1.BIT_87 = 1; /*

```

6. 配列とポインタ

同じようで違う配列とポインタです。アセンブラ的な考え方をすると理解しやすいところです。そもそもポインタは何のために使うのでしょうか。いろいろあるのですが、ひとつには、C の関数はパラメータを値でわたすということに関係があります。値で渡すということは、その変数のコピーを渡すことです。コピーを修正してもオリジナルには何の影響もありません。ポインタを渡せば、渡された関数はそのポインタが指している場所を操作することができます。渡された関数で、値を変更できるのです。例えば、ファイルからデータを読み込む関数は、読み込んだときに発生したエラーを返しますが、読み込んだデータはパラメータによって指定された場所 (ポインタによって指されたところ) へ格納します。というふうに、ポインタを使うと、関数が複数の値を返すこともできるようになります。

変数 x のポインタを $\&x$ で表現します。 $\&x$ は変数 x の番地です。 px をポインタとすると、 $px = \&x$ と書くことができます。 x の番地を px に代入しています。 $\&$ の逆が $*$ です。 $*px$ でポインタ px が指している場所の内容ということになります。 $x = *px$ と書くことができます。ポインタ px が指している場所の内容を x に代入しています

これだけなら話はそう複雑ではないのですが、変数には型があります。これがちょっと、話をややこしくします。型というのは、例えば、H8/3664F では 2 バイトの `int` とか、1 バイトの `char` などです。そのため、ポインタにも型があります。`int` のポインタとか、`char` のポインタと呼ばれます。`*px` が `int` になったり、`char` になったりするのは、

`char` 型の変数はそのサイズが 1 バイトです。`int` は 2 バイトになります。`char` へのポインタの値を +1 すると、1 バイト分アドレスが増えます。`int` へのポインタを +1 すると、`int` は 2 バイトなので、2 バイト分アドレスが増えます。このところは、またあとで説明します。

6. 1 ポインタ

`int` 型の変数 x にたいして、 $\&x$ は変数 x のポインタです。 p が `int` へのポインタとすると $*p$ は、ポインタの指している中身の値になります。すなわち `int` です。 $p = \&x$; あるいは $x = *p$; という式が成立します。関数が `func(int *z)` となっていたとき、 z は `int` のポインタです。 $\&$ がつくるとポインタだってさっきいったばかり...。もう一度よく見てください。 x が `int` 型するとき、 $\&x$ が `int` のポインタです。 x だけなら `int` です。はい、ひと呼吸。`int` のポインタを p とすると $*p$ は `int` になります。それと同じで、`func(int *z)` も $*z$ が `int` である、と読むことができます。 $*z$ すなわち、 z の指している中身が `int` なんだから、 z は `int` のポインタということです。

2つの整数を交換する関数を考えます。C のパラメータは値渡しなので、値を交換する関数 `swap()` を `swap(int *xx, int *yy)` のようにすると、交換してくれません。値を交換するためには、値の入っている場所を渡さなければなりません。それが、この `swap(int *xx, int *yy)` です。`int *xx` は、 xx が `int` 型のポインタであることを意味します。ポインタというのはアドレスということです。`swap()` を呼ぶところで `swap(&ii, &jj);` としています。 $\&ii$ は、変数 ii へのポインタです。変数 ii の番地と同じです。

```

#include <stdio.h>
void swap(int *xx, int *yy);

void main()
{
    int ii = 10;
    int jj = 20;

    swap(&ii, &jj);
    printf("%d %d\n", ii, jj);
}

void swap(int *xx, int *yy)
{
    int temp;

```

```

temp = *xx;
*xx = *yy;
*yy = temp;
}

```

6. 2 文字列型変数

あ、C には文字列型という型はありませんでした。通常は文字の配列として文字列を実現します。たとえばこんな感じです。'hello!Yn' という文字列を考えてみましょう。

```

0  1  2  3  4  5  6  7
h  e  l  l  o  !  Yn Yo
104 101 106 106 111 33 10 0

```

下の列の数字は文字に対する ASCII コードを10進数にしたものです。(改行コードは UNIX の改行である 0x0a になってます)

ch を char 型の配列であると宣言します。宣言で値も書いていますから、その値で配列が初期化されます。

```
char ch[] = 'hello!Yn'
```

配列ですから、ch[0] は 'h'、ch[1] は 'e'、ch[2] は 'l' という具合になっています。最後に文字列の終わりを示す 0 が自動的に入ります。ch[7] が 0 になります。

ch[8] はなんでしょう？ なんだか分かりませんが、ch[8] を使ってもエラーにはなりません。エラーにならないので、これがシステムダウンの原因の一つになることがあります。

文字列のコピーを考えましょう。

strcpy(char *dst, char *src) は、src を dst へコピーします。

```
#include <stdio.h>
```

```
void strcpy(char *dst, char *src);
```

```
void main()
```

```
{
    char str[20];

    strcpy(str, 'hello!Yn');
    printf("%s", str);
}
```

```
void strcpy(char *dst, char *src)
```

```
{
    char ch;

    do {
        ch = *src++;
        *dst++ = ch;
    } while (ch != 'Y0');
}
```

関数 strcpy(char *dst, char *src) は、そのパラメータとして char へのポインタを2つ必要とします。その2つは文字列で、src 文字列から dst 文字列へコピーします。文字列の最後は 'Y0' で終了しているはずですから、'Y0' をコピーしたらおしまいです。

この行を考えてみましょう。

```
ch = *src++;
```

ch は char 型で、src は char のポインタです。*src は char になるので char 型の変数 ch へ代入できます。(ちなみに、ch = src; はコンパイラに型が一致しないぞ! と文句をいわれてしまいます。)

*src++; の ++ はどんな役目なのでしょう。後ろについていますから、値を参照してから更新されます。下の図のように、src が 'o' を指していたとします。ch には 'o' が代入されます。その後、インクリメント演算子 ++ によって、*src ではなく、src が +1 され、'!' を指すようになります。

```

0  1  2  3  4  5  6  7
h  e  l  l  o  !  Yn Yo

src

```

という具合に、一文字ずつ src 側から dst 側へコピーされていきます。'Y0' をコピーしたところでループは終了します。

コンパイラが出力するアセンブラのソースを見ると勉強になるのですが、残念ながら H8/36 64F 用 C コンパイラはアセンブラのソースを出力してくれません。これは大きな欠点です。そのため、このコンパイラを仕事で使おうとする人は少ないでしょう。

文字列は char の配列であるといいつつ、strcpy() は strcpy(char *dst, char *src) というわけで、ポインタを使ってます。ちょっとだらしないです。配列は配列、ポインタはポインタとして区別するべきでした。strcpy(char dst[], char src[]) が、本来の姿でしょうか。でも普通はポインタを使います。

このように、ポインタと配列はだらしなくしていると同じになってしまいます。

ここで、ちょっと問題です。

```
void main()
{
    char str[20];

    strcpy(str, 'hello!Yn');
    printf("%s", str);
}
```

としていましたが、それを次のように修正します。

```
void main()
{
    char *str;

    strcpy(str, 'hello!Yn');
    printf("%s", str);
}
```

`char str[20];` を `char *str;` に変更しました。配列とポインタは同じですから、これでもエラーは発生せずに無事にコンパイルを終了します。実行すると、`char` のポインタ `str` にコピーしようとして、いえ、コピーしてしまいます。

コピー前に `str` はどこを指しているのでしょうか？そこが問題です。どこを差しているのか分からないのです。どこか分からない場所に文字列データがコピーされてしまいます。コピーされた場所はプログラムのスタックエリアかも知れませんが、作業領域あるいは、I/Oポートのある場所かも知れません。そんな状態ではプログラムが正しく動作するわけがありません。

ここは初期化されていないポインタを渡してはいけません。どこか文字列をコピーできるだけの領域を指しているポインタを渡すべきです。このように、どこを指しているのか分からないポインタはすぐに作ることができます。でも、配列は（要素が0でない限り）領域込みで定義されるので、どこを指しているのか分からない、ということは発生しません。ポインタと配列は全く同じではないのです。

6.3 配列

ポインタと配列は同じものではないということを頭の隅に感じながら、今度はポインタと配列は同じであるという説明をします。正確にはポインタでも配列でも同じことを表現できる、ということですね。

<<charの配列>>

```
char str[100];
char *pStr;
```

であるとき、`str` は配列の先頭アドレスです。言葉をかえると `str` は配列の先頭へのポインタと同じ値です。配列の先頭のアドレスは、その文字どおり `0` で表現すると `&(str[0])` です。`&str[0]` と同じですが、気持ち悪いので `&(str[0])` とします。

こんどは、

```
pStr = str;
```

とすると、`pStr` は配列の先頭へのポインタになります。

```
pStr = &(str[0]);
```

と同じですね。

さて、準備は整いました。配列とポインタが同じことを表現できることを説明します。

配列の最初の要素 `str[0]` は `*pStr` と同じです。`pStr` は配列の先頭へのポインタですから、その中身は最初の要素、すなわち `str[0]` となるのです。

次の要素はどうなるでしょう。`str[1]` です。これは、`pStr` を使うと `*(pStr + 1)` となります。このように、`str[n]` は `*(pStr + n)` と同じことで、配列とポインタは表現の形式が違うけれど、その実体は同じであることが分かります。

実は、この配列が `char`、すなわち1バイトの値を要素に持っているから、ポインタ（すなわちアドレス）に1を加えれば、次のアドレス（すなわち配列の次の要素）になる、というのが何となく理解できると思います。

では、`char` ではなく、`int` ではどうでしょう。`int` は2バイトです。

<<intの配列の場合・・・>>

`char` の配列と同様のプログラムにします。

```
int intArray[100];
int *intPtr;
```

`intArray` は、配列 `intArray` の先頭へのポインタと同じ値になります。`&(intArray[0])` です。

```
intPtr = intArray;
```

こうすると、`intPtr` は `intArray` の先頭へのポインタとなります。

配列の最初の要素、`intArray[0]` と `*intPtr` は同じものになることに異義はないでしょう。次に、`intArray[1]` です。`char` の配列であれば一つの要素は1バイトですから、次の要素はポインタ（アドレス）に1を加えたものになることは直感的に理解できます。実は、配列の型に関係なく `intArray[n]` は `*(intPtr + n)` となるのです。

これは驚きです。intPtr + n の値は intPtr + n ではなく、intPtr + n * 2 になっているのです。

ポインタに対して加えた値、例えば intPtr + n の n は、「n 番地移動」ではなく、あくまでも「n 番目の要素」となり、番地ではないのです。こう考えると「ポインタと配列は同じだが、ポインタとアドレスは違うものだ」という結論になってしまいます。

int の配列の要素 0 から 99 までの合計を求めるには、ポインタを使って次のようにすることができます。

```
int getSumOfIntArray(int intArray[])
{
    int *intPtr;
    int sum = 0;
    int xx = 0;

    intPtr = intArray;
    for (xx = 0; xx < 100; xx++) {
        sum += *intPtr++;
    }

    return (sum);
}
```

intPtr++ は intPtr を次の要素を指し示すように値をひとつ増加させるという意味です。

配列とポインタは同じようなものなのですが、次の例はエラーとなります。どこがエラーなんでしょうか。100個の int 型の要素を持つ array という名前の配列を作りました。

```
int array[100];
```

array だけなら、&(array[0]) と同じものです。配列 array の最初の要素へのポインタと同じ値です。同じ値だからといって、ポインタのように使うことはできません。

```
sum += *array++;
```

*array はエラーになりませんが、array++ がいけません。array はどこかのメモリに置かれた領域の番地です。その番地は固定ですから array は、定数として扱われるのです。そのため、array++ は定数を変更するわけですから、エラーとなります。

```
int array[100];
```

```
int sum = 0;
int xx = 0;

for (xx = 0; xx < 100; xx++) {
    sum += *array++; /* ERROR !! */
}
```

6. 4 I/O アクセス

I/O のレジスタに値を設定したり、読み出したりするときにポインタを使います。I/O アクセスを簡単化するために 36641.h というインクルードファイルが用意されていますが、ここではポインタを使って I/O にアクセスしてみましょう。

H8 の I/O は、メモリと同じアドレス空間に置かれていますから、メモリと同様に読み書きします。そこで問題になるのが、対象となる I/O レジスタのサイズによって扱い方が異なるということです。

レジスタが 1 バイトなら、1 バイトの変数となる char あるいは unsigned char へのポインタを使いますし、レジスタが 2 バイトなら 2 バイト変数となる int あるいは unsigned int へのポインタを使います。

例えば、パラレルポート 1 のデータレジスタ (PDR1) に 0x1f を書き込むときには次のようにします。

まず、パラレルポート P1 のデータレジスタのアドレスは 0x1f1d4 ですから、1 バイトの変数へのポインタ pi を用意し、それにアドレス 0x1f1d4 を代入しておきます。pi はポインタですから、*pi はポインタが指している内容になります。そこで、*pi = 0x1f; とすることで、pi が指している番地の内容を 0x1f に設定したことになります。

注釈になっている行は、36641.h で宣言されている構造体を使った I/O アクセスの表現方法です。どちらも同じことをしています。

```
/* IO.PDR1.BYTE = 0x1f; */
char *p1dr = (char *)0x1f1d4;
*p1dr = 0x1f;
```

2 バイトレジスタは次のようになります。タイマー W の 2 バイトレジスタ GRA に値を設定しています。タイマー W の GRA は番地 0x1f188 です。そこに 2 バイトの値 0x9c40 を書き込んでいる例です。

注釈になっている行は、36641.h で宣言されている構造体を使った I/O アクセスの表現方法です。どちらも同じことをしています。

```
/* TW.GRA = 0x9c40; */
int *gra0 = (int *)0x1f188;
*gra0 = 0x9c40;
```

36641.h を使うと、その I/O レジスタのアドレスを調べなくてもよくて便利なのですが、本当にそのアドレスになっているのかアセンブラ出力がないので、確認する方法がありません。プログラムの動作がおかしいときにはポインタを使って、レジスタのアドレスを直接表現したほうが良いでしょう。

7、そのほか

何か書き落としたことはないでしょうか。

C はトリッキーなプログラムを作ることもできます。また、プログラマーはついつい「自分は、こんなこともできるんだぞー。どうだすごいだろう。」的なプログラムを書いてしまいがちです。そんなプログラマーに C はびったりなんです。

しかし、それではいけません。そんな状態に陥らないように、分かりやすいプログラムを作ることを心がけましょう。三項演算子よりは || 文を使うようにしたり、|| 文の条件式には代入文はかかないとか、注意すべきところはたくさんあります。演算子を評価する優先順位を覚えていないと理解できないような式は、() を使って評価の優先順位を明確にすることも大切でしょう。

C で書かれたプログラムは、CPU が変わってもほとんど修正せずに使うことができます。しかし、レジスタの設定など、CPU に依存した記述をプログラムのあちこちにばらまいてはいけません。CPU などの環境が変わっても、その変化に対するプログラムの修正を少なくするためには、I/O の初期化や、ビットのテストなど、その CPU 特有の処理は、なるべく一箇所にまとめておき、簡単に変更できるようにしておきます。また、入出力を使わない内部処理だけのプログラムであれば、パソコン上の C でチェックできます。デバッグ環境の貧弱さを十分カバーしてくれます。

ま、そんなわけで、C を好きになるのはかまいませんが、プログラムを作ることが目的であって、C はそのための道具であることをお忘れなく。

最後にシリアルポートから入出力するプログラムの C ソースを示しておきます。

```
/*
 *      sci.c
 *      for H8/3664
 *      set lab 4
 */

#include '3664f.h'
#include 'ctype.h'          /* islower() and toupper() */
#include 'myType.h'

/* CONST */
#define kCRcode      (0x0d)
#define kLFcode     (0x0a)

/* TYPE */
/* VAR */
/* PROTOTYPE */
void initio(void);
SInt16 inch(void);
```

```
SInt16 iachxx(void);
void outch(short ch);
void out8h(UInt32 xx);
void out6h(UInt32 xx);
void out4h(UInt16 xx);
void out2h(UInt8 xx);
void outh(UInt8 xx);
void crlf(void);
```

/* PROGRAM */

```
void main(void)
{
    SInt16 ch;

    initio();
    crlf();
    outch('*');
    do {
        ch = inch();
        outch(ch);
    } while (true);
} /* main */
```

void initio(void)

```
{
    SInt16 xx;
    IO.PMR1.BIT.TXD = 1;
    SCI3.SCR3.BYTE = 0; /* clear all flags */
    SCI3.SMR.BYTE = 0; /* Ascnc, 8bit, NoParity, (Even), stop1, 1

/1 */
    SCI3.BRR = 25; /* 19200baud (CPU=16MHz) */
    for (xx = 0; xx < 280; xx++) {
        /* wait 1 bit time (1/9600 sec) */
    }
    SCI3.SCR3.BYTE = 0x30; /* scr=0011 0000 (TE=1, RE=1) */
    xx = SCI3.SSR.BYTE; /* Dummy Read */
    SCI3.SSR.BYTE = 0x80; /* Clear Error Flag (TORE=1) */
} /* initio */
```

SInt16 iach(void)

```
{
    SInt16 ch;
```

```

ch = inchxx();
if (islower(ch)) {
    ch = toupper(ch);
}
return (ch);
/* inch */

```

```

static inline void
inchxx(void)
{
    static char ch;

    do {
        ch = SC13.SSR.BYTE;
    } while ((ch & 0x78) == 0); /* RDRF=ORER=FER=PER=0 */

    if (SC13.SSR.BIT.RDRF == 0) {
        ch = -1; /* error */
        SC13.SSR.BYTE = 0x80;
    } else {
        ch = SC13.RDR;
        SC13.SSR.BIT.RDRF = 0;
    }
    return (ch);
/* inchxx */
}

```

```

void outch(short ch)
{
    static int xx;

    do {
        xx = SC13.SSR.BIT.TDRE;
    } while (xx == 0);
    SC13.TDR = ch;
    SC13.SSR.BIT.TDRE = 0;
/* outch */
}

```

```

void out8h(uint32_t xx)
{
    out4h((uint16_t)(xx >> 16));
    out4h((uint16_t)(xx & 0xffff));
/* out8h */
}

```

```

void out16h(uint32_t xx)
{
    out2h((uint8_t)((xx >> 16) & 0xff));
    out4h((uint16_t)(xx & 0xffff));
/* out16h */
}

```

```

void out4h(uint16_t xx)
{
    out2h((uint8_t)(xx >> 8));
    out2h((uint8_t)(xx & 0xff));
/* out4h */
}

```

```

void out2h(uint8_t xx)
{
    outb((uint8_t)(xx >> 4));
    outb((uint8_t)(xx & 0xf));
/* out2h */
}

```

```

void outh(uint8_t xx)
{
    xx &= 0xf;
    xx |= '0';
    if (xx > '9') {
        xx += 7;
    }
    outch(xx);
/* outh */
}

```

```

void crlf(void)
{
    outch(kCRcode);
    outch(kLFcode);
/* crlf */
}

```

```

/* end of sci.c */

```