お客様各位

カタログ等資料中の旧社名の扱いについて

2010年4月1日を以ってNECエレクトロニクス株式会社及び株式会社ルネサステクノロジが合併し、両社の全ての事業が当社に承継されております。従いまして、本資料中には旧社名での表記が残っておりますが、当社の資料として有効ですので、ご理解の程宜しくお願い申し上げます。

ルネサスエレクトロニクス ホームページ（http://www.renesas.com）

2010年4月1日
ルネサスエレクトロニクス株式会社

【発行】ルネサスエレクトロニクス株式会社（http://www.renesas.com）

【問い合わせ先】http://japan.renesas.com/inquiry

ご注意書き

1. 本資料に記載されている内容は本資料発行時点のものであり、予告なく変更することがあります。当社製品のご購入およびご使用にあたりましては、事前に当社営業窓口で最新の情報をご確認いただきますとともに、当社ホームページなどを通じて公開される情報に常にご注意ください。
2. 本資料に記載された当社製品および技術情報の使用に関連し発生した第三者の特許権、著作権その他の知的財産権の侵害等に関し、当社は、一切その責任を負いません。当社は、本資料に基づき当社または第三者の特許権、著作権その他の知的財産権を何ら許諾するものではありません。
3. 当社製品を改造、改変、複製等しないでください。
4. 本資料に記載された回路、ソフトウェアおよびこれらに関連する情報は、半導体製品の動作例、応用例を説明するものです。お客様の機器の設計において、回路、ソフトウェアおよびこれらに関連する情報を使用する場合には、お客様の責任において行ってください。これらの使用に起因しお客様または第三者に生じた損害に関し、当社は、一切その責任を負いません。
5. 輸出に際しては、「外国為替及び外国貿易法」その他輸出関連法令を遵守し、かかる法令の定めるところにより必要な手続を行ってください。本資料に記載されている当社製品および技術を大量破壊兵器の開発等の目的、軍事利用の目的その他軍事用途の目的で使用しないでください。また、当社製品および技術を国内外の法令および規則により製造・使用・販売を禁止されている機器に使用することができません。
6. 本資料に記載されている情報は、正確を期すため慎重に作成したものですが、誤りがないことを保証するものではありません。万一、本資料に記載されている情報の誤りに起因する損害がお客様に生じた場合においても、当社は、一切その責任を負いません。
7. 当社は、当社製品の品質水準を「標準水準」、「高品質水準」および「特定水準」に分類しております。また、各品質水準は、以下に示す用途に製品が使われることを意図しておりますので、当社製品の品質水準をご確認ください。お客様は、当社の文書による事前の承諾を得ることなく、「特定水準」に分類された用途に当社製品を使用することができません。また、お客様は、当社の文書による事前の承諾を得ることなく、意図されていない用途に当社製品を使用することができません。当社の文書による事前の承諾を得ることなく、「特定水準」に分類された用途または意図されていない用途に当社製品を使用したことによりお客様または第三者に生じた損害等に関し、当社は、一切その責任を負いません。なお、当社製品のデータ・シート、データ・ブック等の資料で特に品質水準の表示がない場合は、標準水準製品であることを表します。

   標準水準： コンピュータ、OA 機器、通信機器、計測機器、AV 機器、家電、工作機械、パーソナル機器、産業用ロボット

   高品質水準： 輸送機器（自動車、電車、船舶等）、交通用信号機器、防災・防犯装置、各種安全装置、生命維持を目的として設計されていない医療機器（厚生労働省定義の管理医療機器に相当）

   特定水準： 航空機器、航空宇宙機器、海底中継機器、原子力制御システム、生命維持のための医療機器（生命維持装置、人体に埋め込み使用するもの、治療行為（患部切り出し等）を行うもの、その他直接人命に影響を与えるもの）（厚生労働省定義の高度管理医療機器に相当）またはシステム等
8. 本資料に記載された当社製品のご使用につき、特に、最大定格、動作電源電圧範囲、放熱特性、実装条件その他諸条件につきましては、当社保証範囲内でご使用ください。当社保証範囲を超えて当社製品をご使用された場合の故障および事故につきましては、当社は、一切その責任を負いません。
9. 当社は、当社製品の品質および信頼性の向上に努めておりますが、半導体製品はある確率で故障が発生したり、使用条件によっては誤動作したりする場合があります。また、当社製品は耐放射線設計については行っておりません。当社製品の故障または誤動作が生じた場合も、人身事故、火災事故、社会的損害などを生じさせないようお客様の責任において冗長設計、延焼対策設計、誤動作防止設計等の安全設計およびエージング処理等、機器またはシステムとしての出荷保証をお願いいたします。特に、マイコンソフトウェアは、単独での検証は困難なため、お客様が製造された最終の機器・システムとしての安全検証をお願いいたします。
10. 当社製品の環境適合性等、詳細につきましては製品個別に必ず当社営業窓口までお問合せください。ご使用に際しては、特定の物質の含有・使用を規制する RoHS 指令等、適用される環境関連法令を十分調査のうえ、かかる法令に適合するようご使用ください。お客様がかかる法令を遵守しないことにより生じた損害に関して、当社は、一切その責任を負いません。
11. 本資料の全部または一部を当社の文書による事前の承諾を得ることなく転載または複製することを固くお断りいたします。
12. 本資料に関する詳細についてのお問い合わせその他お気付きの点等がございましたら当社営業窓口までご照会ください。

注 1. 本資料において使用されている「当社」とは、ルネサスエレクトロニクス株式会社およびルネサスエレクトロニクス株式会社がその総株主の議決権の過半数を直接または間接に保有する会社をいいます。

注 2. 本資料において使用されている「当社製品」とは、注 1 において定義された当社の開発、製造製品をいいます。

ユーザーズ・マニュアル

RENESAS

SIMPLEHOST®

17Kシリーズ用ディバッガ

emIC-17K®/RA17K対応版

入門編（Windows™対応）

対象エミュレータ

IE-17K

IE-17K-ET

資料番号　U10445JJ2V0UM00（第2版）
（旧資料番号　EEU−5009）
発行年月　August 1997　N

NEC

ユーザーズ・マニュアル

SIMPLEHOST®

17Kシリーズ用ディバッガ

emIC-17K®/RA17K対応版

入門編（Windows™対応）

対象エミュレータ

IE-17K

IE-17K-ET

資料番号　U10445JJ2V0UM00（第2版）
（旧資料番号　EEU−5009）
発行年月　August 1997　N

〔メ モ〕

目次要約

本版で改訂された主な箇所

箇　所	内　容
全　般	バージョン　V2.0→V2.01に変更
	Windows3.1→Windows3.1およびWindows95対応に変更
	Windows3.1→Windows95上の画面に変更
	*SINPLEHOST*のEditorを削除
p.5	1.2　インストールを変更
p.41, 45	2.4.2　プログラムの実行を変更 2.4.4　実行チェックを変更
p.83	3.2.1　Traceを変更
p.97	付録B　改版履歴を追加

本文欄外の★印は，本版で改訂された主な箇所を示しています。

巻末にアンケート・コーナを設けております。このドキュメントに対するご意見をお気軽にお寄せください。

はじめに

このたびは、*SIMPLEHOST*（*emIC-17K*RA17K対応版）をお買い求めいただきまして、ありがとうございました。*SIMPLEHOST*とは、Simple & High level Operation Support Toolの略語で、4ビット・シングルチップ・マイコン17Kシリーズの開発用サポート・ツールIE-17Kのコントロール・ソフトウエアです。Windowsに対応したホスト・マシン上で動作します。

*emIC-17K*とはEmbedded Medium Level C Compiler for 17K seriesの略語で、17Kシリーズのためのcライクなコンパイラです。

RA17Kとは、17Kシリーズのためのリロケータブル・アセンブラです。

これらでコンパイルまたはアセンブルされたプログラムをディバグするためのツールが*SIMPLEHOST*（*emIC-17K*RA17K対応版）です。

このマニュアルで*SIMPLEHOST*と言った場合は*SIMPLEHOST*（*emIC-17K*RA17K対応版）のことを指します。

*SIMPLEHOST*のマニュアルは次のもので構成されています。

SIMPLEHOST *emIC-17K*RA17K対応版 ユーザーズ・マニュアル 入門編
（このマニュアル）

SIMPLEHOST *emIC-17K*RA17K対応版 ユーザーズ・マニュアル レファレンス編
（U10496J）

このマニュアルは、はじめて*SIMPLEHOST*をお使いになるときや、*emIC-17K*を使ったプログラミング、*SIMPLEHOST*を使ったディバグの基本的なことを理解したいときにご覧ください。

また、C言語と表記した場合は、ANSI（（X3J11委員会X3-159、1989）（1989.12.14））で規定化されたC言語です。

●このマニュアルの目的と構成

このマニュアルは、はじめて*emIC-17K*RA17Kでプログラムを作成する方や、はじめて*SIMPLEHOST*をお使いになり、ディバグを行う方のためのマニュアルです。プログラムのサンプルとして「タイマの検査」を使い、*emIC-17K*のプログラミングとディバグの方法について解説します。

このマニュアルは、次の内容から構成されています。

このマニュアルに記載されたレッスンの流れで、*emIC-17K*の文法に従いプログラムを作成し、*SIMPLEHOST*でプログラムのディバグをしていけば、バグの少ないプログラムが作成できるようになっています。

●入門編を読む前に

次の資料に目を通しておくことをお勧めします。

・17Kシリーズ　ユーザーズ・マニュアル　アーキテクチャ編：IEU-769

●凡例

“　”　任意の文字，画面内の項目を示します。

［　］　画面のタイトルを示します。

「　」　メニューの名称，ボタンの名称，メッセージを示します。

↵　改行キーの入力

目　　次

図 の 目 次 (1/2)

図 の 目 次 (2/2)

第1章　概　　要

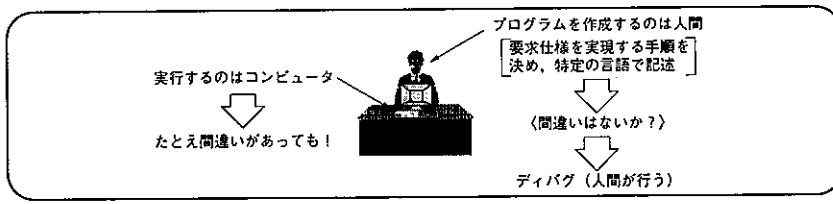
この章では、概要（お使いの前の簡単な説明）およびインストールについて説明します。

1.1　プログラムとディバグ

まずプログラムとは何でしょうか？

プログラムとは、コンピュータなどに対する命令の集まりです。 （コンピュータが認識できる言葉：言語で書かれています）

プログラムは，コンピュータが実行する処理の手順を定めたものです。プログラミングとは，プログラムを作成することです。プログラムを作成するのは人間で、実行するのはコンピュータです。



プログラムを作成しても間違いなく動作するとは限りません。プログラムの間違い（バグと呼ばれています）を取り除くことをディバグといいます。

次にコンパイラとディバッガについて少しお話ししましょう。

17 K用のコンパイラは、*emIC-17K*（Embeded Midium Level C compilerの略称，「エムエルシー」と読みます）と呼ばれています。

*emIC-17K*は，

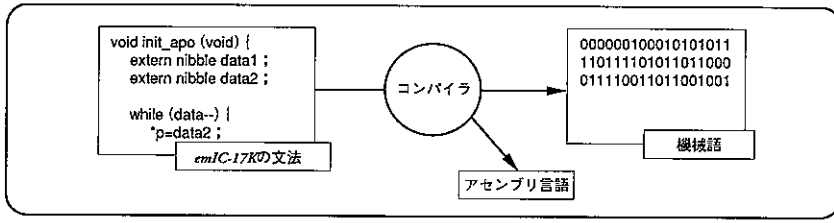
プログラムを書くための人工的な「プログラミング言語」です。

*emIC-17K*は，システム記述を目的としたC言語の流れを受けた高水準言語（より人間の話す言葉に近い言語）に近いプログラミング言語として設計されました。よって，ハードウエアに密着した命令が使える，マイクロコンピュータの性能を十分に発揮できる，そしてプログラマの論理を素直に書くことのできるプログラミング言語であるということができます。

*emIC-17K*は，

ソース・プログラムを機械語に変換する「コンパイラ」です。

アセンブラ（RA17K）が，アセンブリ言語で書かれたソース・プログラムを機械語に変換するのに対して，高水準言語で（*emIC-17K*の文法で）書かれたソース・プログラムを機械語に変換します。



*emIC-17K*を使うと，プログラマの論理を素直に書くことができ，デバイスに依存しないプログラムが書きやすいということです。

*emIC-17K*を使うことは

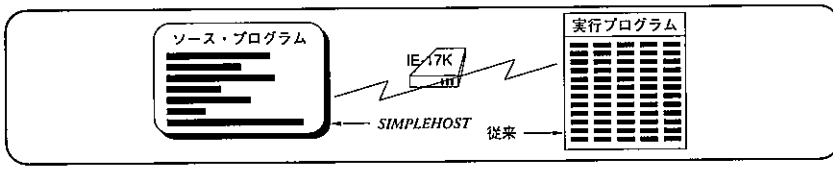
より快適なプログラミング環境を維持する（簡単につかえる）
C言語の豊富な知識と経験を活用する（だれでも使える）

ということです。

17K用のディバッガは，*SIMPLEHOST*（Simple & High level Operation Support Toolの略称で「シンプルホスト」と読みます）と呼ばれています。*SIMPLEHOST*は，プログラムのディバグ作業（バグを捜し修正する）をサポートするためのツールです。

今までは，インサーキット・エミュレータを単独で使用してプログラムのディバグ，検査を行っていました。これでは，インサーキット・エミュレータが，高機能である反面，コマンドを覚えたりするのにいささか熟練が必要でした。

そこでインサーキット・エミュレータとプログラムのインタフェース（操作／表示）をよくする手段として，*SIMPLEHOST*は開発されました。これにより，ディバッガのコマンドを意識することなくディバグに専念し，コマンドの学習時間を減らすことができました。



それでは，*SIMPLEHOST*を使うと何ができるのか次に示します。

・マニュアル・フリー

*SIMPLEHOST*は，専門家でない人がプログラムのディバグを行うことも予想して，だれでも簡単に操作できるようになっています（また，オンライン・ヘルプ機能を持っています）。

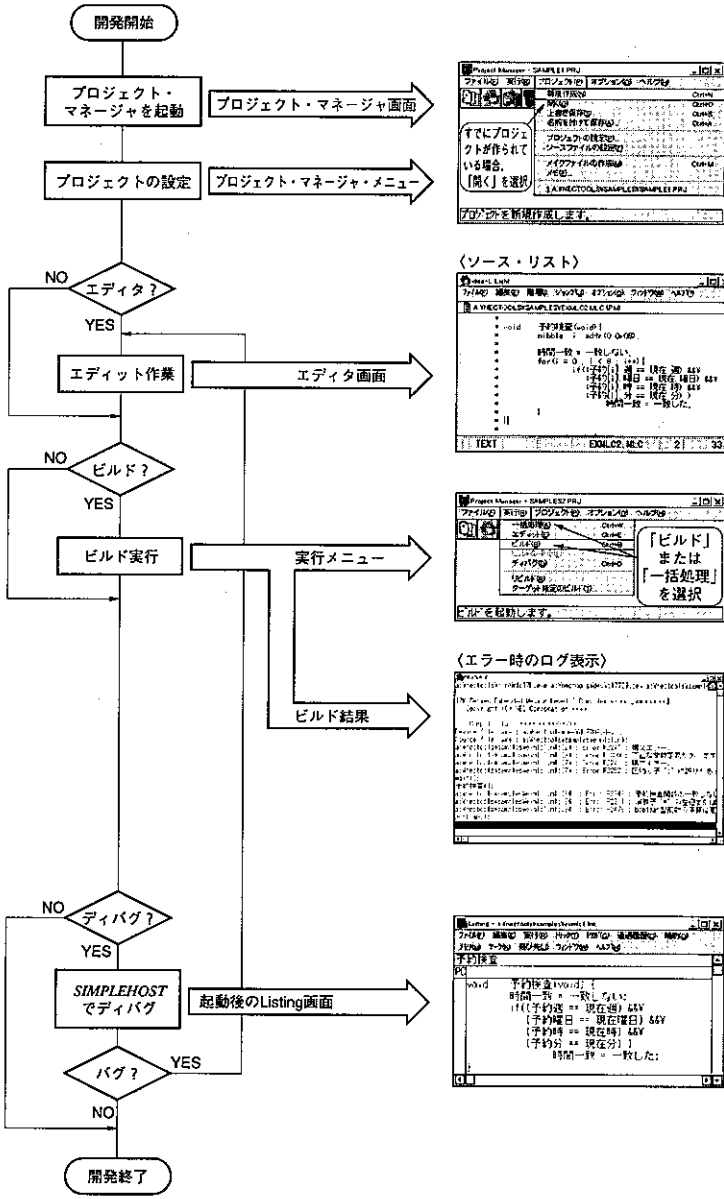
・できることがみえる／できたことがみえる

*SIMPLEHOST*は，今までのディバッガと異なり，すべてのコマンドを覚えなくても操作することができます。操作に必要な情報はすべて画面に表示されているからです。

コマンドの学習時間を可能な限り短くし，実際に操作しながら学習する。

また，*SIMPLEHOST*は，今までのディバッガと異なり数字の羅列を見なくても結果を直感的に理解することができます。操作の結果が，使う人の見やすい表やグラフなどで表示されるからです。

次に17Kシリーズ用プログラム開発の流れを示します。


開発開始
プロジェクト・マネージャを起動
プロジェクト・マネージャ画面
すでにプロジェクトが作られている場合、「開く」を選択
プロジェクトの設定
プロジェクト・マネージャ・メニュー
NO
エディタ？
YES
〈ソース・リスト〉
エディット作業
エディタ画面
NO
ビルド？
YES
「ビルド」または「一括処理」を選択
ビルド実行
実行メニュー
ビルド結果
〈エラー時のログ表示〉
NO
ディバグ？
YES
SIMPLEHOST でディバグ
起動後のListing画面
バグ？
YES
NO
開発終了

1.2 インストール

1.2.1 動作環境

*SIMPLEHOST*は，次の環境で動作します。

ホスト・マシン	PC-9800シリーズ（CPU：80386以上）/IBM PC/AT™
メモリ	メイン・メモリ 640 Kバイト
	プロテクト・メモリ 4 Mバイト以上
ハード・ディスク	5 Mバイト以上の空き容量
マウス	
OS	MS-DOS™ バージョン3.30以上/PC DOS™ バージョン5.00以上
	Windows バージョン3.1
★	またはWindows95
モニタ	Windowsアクセラレータ対応

★ 1.2.2 インストール

*SIMPLEHOST*の出荷媒体はフロッピィ・ディスク2枚で構成されています。

Windows3.1またはWindows95上で，setup.exeファイルを実行してください。*SIMPLEHOST*のインストールを開始します。

【実行例】Windows95上でインストールする場合

ドライブCより*SIMPLEHOST*のファイルを読み込み，A:¥nectools¥binにインストールする場合の実行例を次に示します。

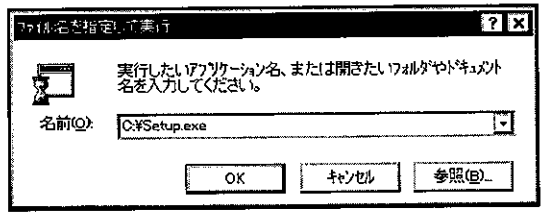
Windows95はすでに起動してあるとします。

（1）インストーラを起動します。

① “ID17K SETUP DISK#1”をフロッピィ・ディスク・ドライブに挿入します。

② スタートメニューから「ファイル名を指定して実行」を選択します。

③ “名前”欄には“C¥:setup.exe”と入力されています。



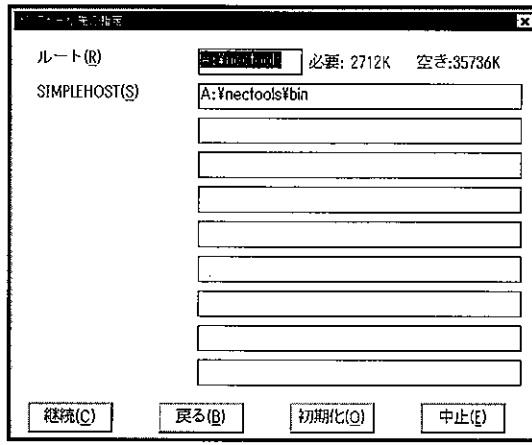
④ 「OK」を選択します。セットアップの初期化処理のあと、インストーラが起動します。

⑤ 「ようこそ、NECセットアップへ。」のメッセージがでますので、「継続」を選択してください。

⑥ 「継続」を選択してください。

（2）インストール・ディレクトリを指定します。

① ［インストール先の指定］ダイアログ・ボックスが表示されます。



② インストールするディレクトリを“ルート”に入力します。

③ 「継続」を選択します。

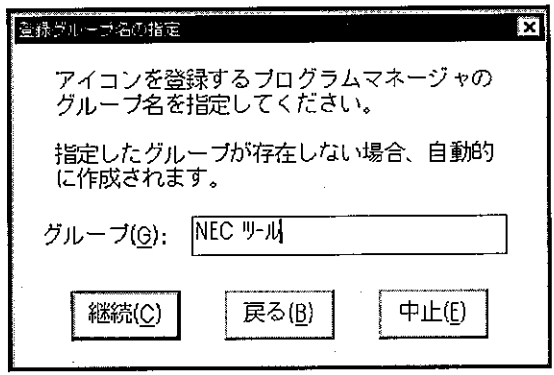
備考1．「戻る」を選択すると、［ID17K setup］ダイアログに戻ります。

2．「初期化」を選択すると、指定ディレクトリがディフォールトのディレクトリを指定します。インストール先のルートのディフォールトは、Windows95がインストールされているドライブの¥nectoolsになります。すでにインストーラでツールがインストールされている場合は、そのルートになります。

ルートを変更すると、それ以下のディレクトリも連動して変更されます。

（3）登録するグループを指定します。

① ［登録グループ名の指定］ダイアログが表示されます。



② 登録グループ名を“グループ”に入力します。

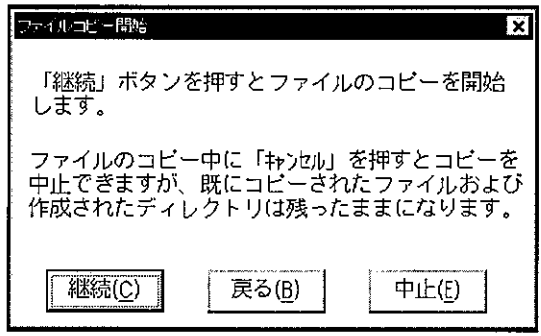
指定したグループが存在しない場合，そのグループが新規に作成されます。また，指定したグループがすでにインストーラを使用して登録してある場合，そのグループを使用します。

③ 「継続」を選択します。

備考 「戻る」を選択すると，［インストール先の指定］ダイアログに戻ります。

（4）ファイルのコピーを開始します。

① ［ファイルコピー開始］ダイアログが表示されます。



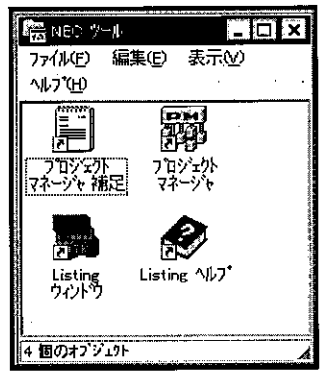
② 「継続」を選択すると、ファイルのコピーを開始します。

備考　「戻る」を選択すると、［登録グループ名の指定］ダイアログに戻ります。

（5）出荷媒体を交換します。

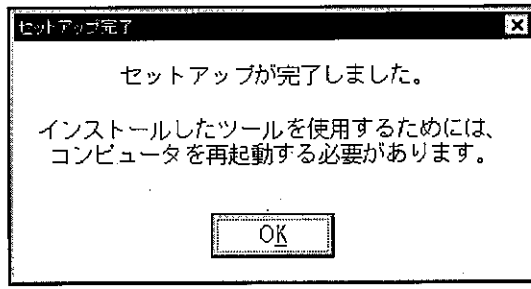
次のメッセージが表示されたら“ID17K SETUP DISK #2”をフロッピィ・ディスク・ドライブに挿入します。

（6）登録されたグループと，アイコンが作成されます。



備考 *SIMPLEHOST*のアイコンは、「Listingウインドウ」と表示されます。

（7）インストールを終了します。



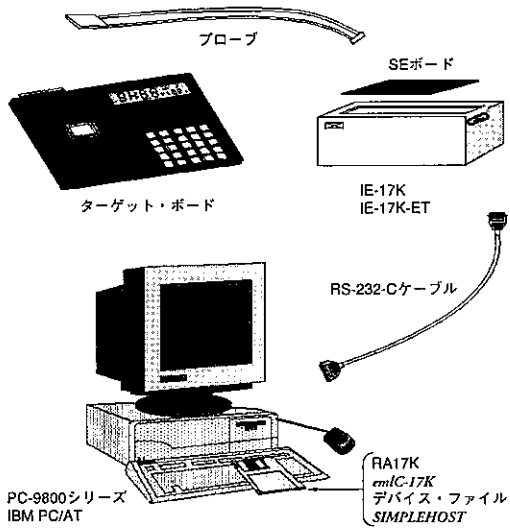
① 「OK」を選択すると、インストーラが終了します。

② コンピュータを再起動してください。

1.3　お使いの前に

1.3.1　動作環境の確認

実際の作業を始める前に，ディバグなどができる状態になっているかどうかを確認してください。



★　備考　*SIMPLEHOST*ファイルの一部（HOST17K.DLL）を取り替えることにより，インサーキット・エミュレータEMU-17Kでも*SIMPLEHOST*を使用できます。（EMU-17Kは，株式会社内藤電誠町田製作所の製品です）。

1.3.2 IE-17Kの設定

*SIMPLEHOST*は、IE-17Kと人間とのインタフェースをするためのソフトウエアです。

*SIMPLEHOST*を使用する前にホスト・マシンにIE-17Kを接続する必要があります。

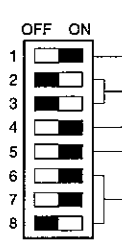
すでにIE-17Kをお使いの場合は、そのままの設定でかまいません。IE-17Kの内部設定は，IE-17K ユーザーズ・マニュアルを参照してください。

*SIMPLEHOST*がうまく動作しない場合に次の項目をチェックしてください。

□メモリ・ボードのスイッチ設定は間違っていませんか？
□スーパバイザ・ボードのスイッチ設定は間違っていませんか？
□メモリ・ボードとスーパバイザ・ボードはしっかり差し込まれていますか？
□SEボードはメモリ・ボード上に実装されていますか？
□メモリ・ボードにケーブルが接続されていますか？
□スーパバイザ・ボードにケーブルが接続されていますか？

ホスト・マシンとIE-17Kを接続します。ホスト・マシンのRS-232-CコネクタとIE-17KのRS-232-CコネクタのCHANNEL0を付属のケーブルで接続してください。

スーパバイザ・ボードのディップ・スイッチの設定を次に示します。



コントロール方式	1	設定内容		
	ON	フロー・コントロール		
	OFF	ライン・コントロール		

パリティ・ビット	2	3	設定内容
	ON	ON	奇 数
	ON	OFF	使用不可
	OFF	ON	偶 数
	OFF	OFF	な し

ストップ・ビット	4	設定内容
	ON	2ビット
	OFF	1ビット

キャラクタ長	5	設定内容
	ON	8ビット
	OFF	7ビット

ボー・レート	6	7	8	設定内容
	OFF	OFF	ON	76800 bps
	OFF	OFF	OFF	38400 bps
	ON	ON	ON	19200 bps
	ON	ON	OFF	9600 bps
	ON	OFF	ON	4800 bps
	ON	OFF	OFF	2400 bps
	ON	ON	ON	1200 bps
	OFF	ON	OFF	600 bps

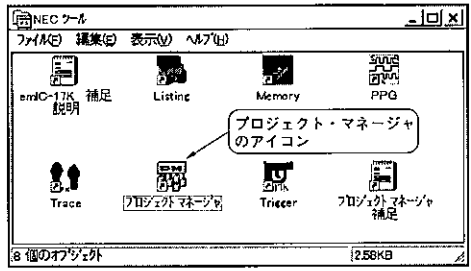
1.4 起動と終了

1.4.1 起 動

*SIMPLEHOST*は，Windows上のアプリケーション・ソフトウエアです。
必ずWindowsを起動後に実行してください。

Windows画面のNECツールのグループを選択するとアイコンが表示されます。

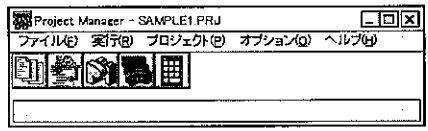
図1－1 アイコン画面



[プロジェクト・マネージャの起動]

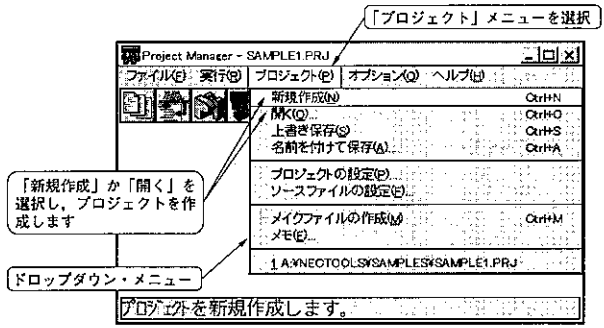
グループ内のアイコン（プロジェクト マネージャ）をダブルクリックすると，次のようなプロジェクト・マネージャの起動画面が表示されます。

図1－2 プロジェクト・マネージャ起動画面



★ プロジェクト・マネージャを起動したら，「プロジェクト」－「新規作成」／「開く」メニューによりプロジェクトを作成します。作成方法の詳細は，17Kシリーズ プロジェクト・マネージャ ユーザーズ・マニュアル レファレンス編を参照してください。

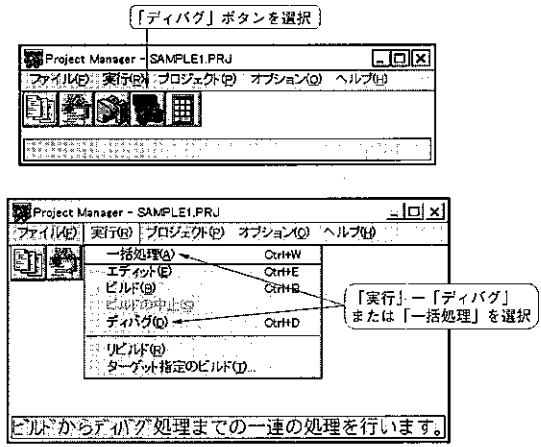
★ 図1－3　「プロジェクト」メニュー



［*SIMPLEHOST*の起動］

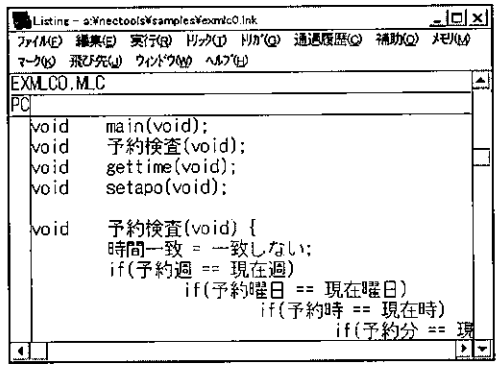
プロジェクトを作成したあとに画面上の「ディバグ」ボタンまたは、「実行」－「ディバグ」／「一括処理」メニューを選択すると*SIMPLEHOST*が起動します。

図1－4　「実行」メニュー



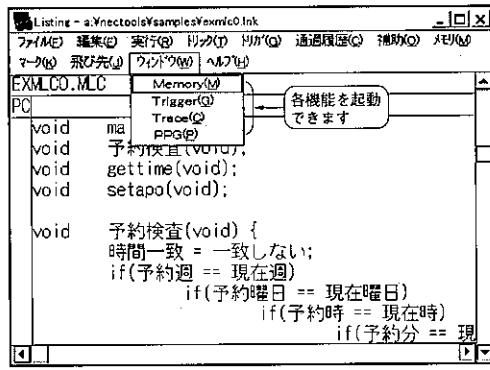
*SIMPLEHOST*の起動画面を次に示します。

図1－5 *SIMPLEHOST*起動画面



*SIMPLEHOST*は、Listingよりほかのウィンドウ（Memory、Trigger、Trace、PPG）を起動します。

図1－6 「ウィンドウ」メニュー

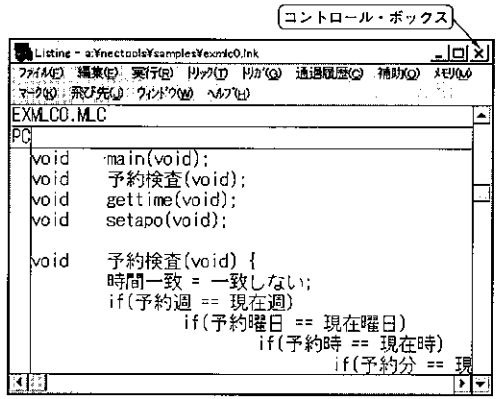


1.4.2 終　了

*SIMPLEHOST*の終了方法には2種類あります。

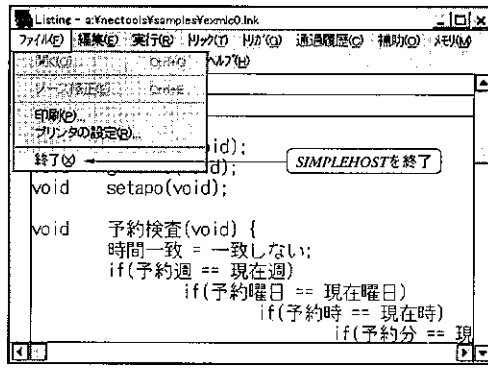
① Listing画面の右上にあるアイコン（これをコントロール・ボックスといいます）を使います。
コントロール・ボックスをマウスでクリックすることにより画面を閉じることができます。

図1－7　コントロール・ボックス



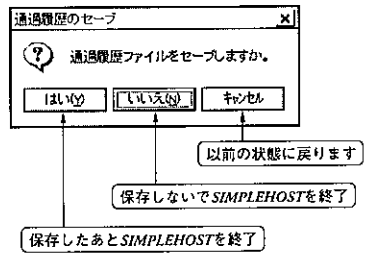
② 「ファイル」－「終了」メニューを選択することにより画面を閉じることができます。

図1－8　「ファイル」メニュー



*SIMPLEHOST*の終了時に，通過履歴をセーブするかしないかの確認メッセージを表示します。

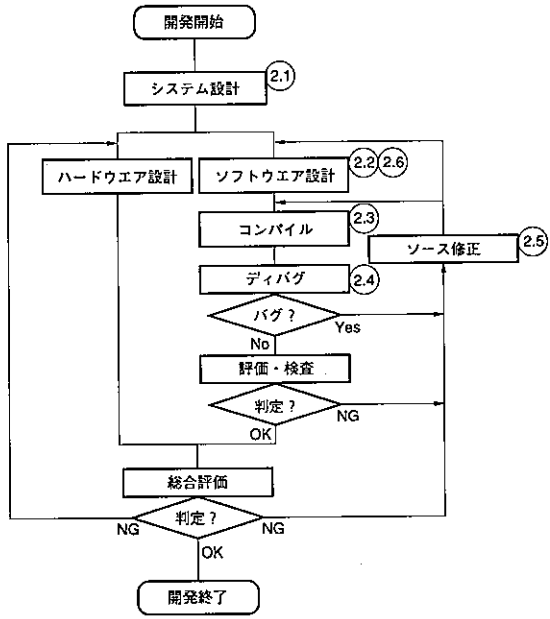
図1－9　通過履歴のセーブ



〔メ モ〕

第２章　プログラム設計

この章では，入門編を使っていくうえで重要となるサンプル・プログラムの設計を行います。
プログラムの開発手順を次に示します。



備考　○内は節番号を示します。

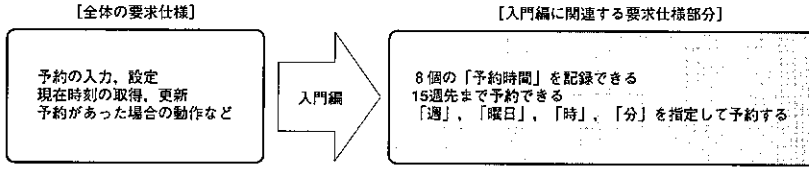
2.1　システム設計

2.1.1　要求仕様

この項では，入門編を使っていくうえで重要となるサンプル・プログラムの要求仕様を説明します。この項の「要求仕様」を理解したうえで次のステップにお進みください。

入門編では，タイマのプログラムの一部を作成します。そして，このプログラムの作成とディバグを手順を追って解説していきます。

このタイマのプログラムで今回の入門編に関連する部分を次に示します。



「要求仕様」を次に示します。

現在の時刻と予約の時刻を比較し，一致したらフラグを立てて終了する

2.1.2　プログラム設計方針

入門編で最初に作成するプログラムはプログラムとして必要な仕様を完全には満たしていません。初めは骨格となるプログラムを作成し，あとから少しずつ肉付けをして仕様を満たすようにします。

これは通常の（4ビット・マイコンの）プログラム作成手順とは多少異なるかもしれません。通常，作成するプログラムの仕様はあらかじめ完全に決められており，経験のあるプログラマならば，要求仕様を見てその仕様を完全に満たしたプログラムを作成してから，ディバグにとりかかります。

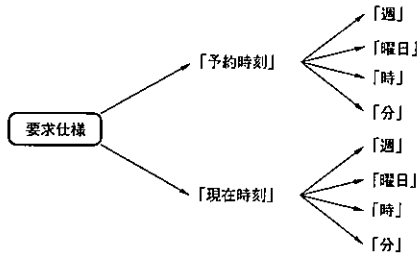
具体的には，入門編で作成するプログラムは，8個の「予約時刻」を検査し，現在時刻と一致したかどうかを検査するプログラムです。しかし，その前に，1個の「予約時刻」と現在時刻を比較することができなければ，当然8個の「予約時刻」を検査することはできません。

そこで，入門編では，1個の「予約時刻」と現在時刻を比較するプログラムを作成し，それを元に8個の「予約時刻」を検査するプログラム（要求仕様のすべてを満たすプログラム）を作成します。

2.2　ソフトウエア設計

2.2.1　データ定義

まず，実際のプログラムを作成する前にデータの定義を行うために要求仕様の分析を行います。



2つの時間を比較するのですから，4×2で合計8個のデータが必要となります。この8つのデータをそれぞれ格納するための8つの箱の名前を次のように決めます。

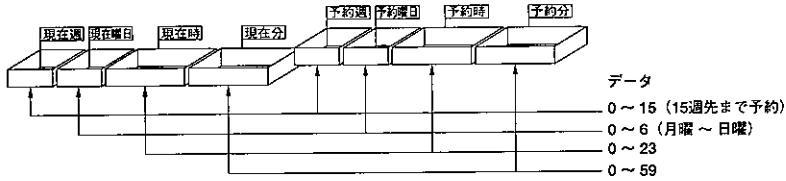
現在週，現在曜日，現在時，現在分
予約週，予約曜日，予約時，予約分

「現在時刻」は，常に変化するデータですからこれを「変数」として扱います。「予約時刻」も同様です。

［参考］　一般に変数とは，プログラムで使用するデータを格納するための箱のようなものです。これは通常，中のデータを変更することができるようになっています（変更できないようにすることもできます）。

17Kシリーズのプログラムでは，「変数として扱う」ということは，「データ・メモリに領域を確保する」ということと同じ意味です。今まで17Kシリーズのアセンブラを使用してきた方には「MEM型のシンボルを定義する」といったほうが分かりやすいかもしれません。

では，「週」，「曜日」，「時」，「分」，それぞれにはどんなデータが入る可能性があるか考えてみましょう。



［参考］　「時」が12までとした場合は，このほかに午前か午後かを表すフラグが必要です。

数種類の大きさの箱があるので，その中からデータ量が格納できるだけの大きさの箱を指定します。これは「変数」の大きさを指定するための「型指定」と呼ばれ，プログラム上に記述するときは大きさを表す「型指定子」と呼ばれるキー・ワードを記述します。

4ビット・マイコンのため，最も効率的にデータ処理できるのは4ビットの値です。この大きさをニブル（nibble）といい，0から15までの値を処理することができます。

しかし当然これだけでは足りない可能性もあるので，この8倍（8ニブル）までの大きさの数を処理する方法も用意されています。たとえば，ニブルの2倍の領域（0から255まで）を確保するためのバイト（byte）というキー・ワードがあります。

データ定義	型指定子	データ範囲	格納有効範囲
「週」	nibble	0～15	0～15
「曜日」	nibble	0～6	0～15
「時」	byte	0～23	0～255
「分」	byte	0～59	0～255

表のように「週」と「曜日」に関してはnibbleを使いデータを格納するための領域を確保し，確保した領域に入れたデータを処理します。同様に「時」と「分」に関してはbyteを使います。

このようにして最適なデータ処理領域の大きさを決定するわけです。

名前とその名前の箱の大きさも決まったので，早速これをデータ・メモリに確保するためのプログラムを*cmlC-17K*で書いてみましょう。

その前に，データ・メモリのどこに確保するかを決めなければなりません。*emIC-17K*でデータ・メモリの場所を指定する場合は，addr()指定子を使用します。この指定子では，データ・メモリのバンクとアドレスを指定します。

［フォーマット］

<型名>　<変数名>　addr（<バンク番号>，<アドレス>）；

*emIC-17K*で，変数のための領域をデータ・メモリに確保するプログラムを書くと，次のようになります。

```
<型名>    <変数名>    addr（<バンク番号>，<アドレス>）；
 nibble     現在週          addr(0.0x00)；
 nibble     現在曜日        addr(0.0x01)；
 byte       現在時          addr(0.0x02)；
 byte       現在分          addr(0.0x04)；

 nibble     予約週          addr(0.0x10)；
 nibble     予約曜日        addr(0.0x11)；
 byte       予約時          addr(0.0x12)；
 byte       予約分          addr(0.0x14)；
```

備考　0xは，次の数値が16進数であることを示します。

これで，現在時刻と予約時刻を格納するための箱が用意されたことになります。これを一般に「変数を定義する」といいます。ここで，現在週や予約曜日などは箱に付けた「名前」です。

比較するデータを格納するための変数は，これで確保できました。実をいうとこのプログラムでは，このほかにもう1つ変数が必要になります。それは，2つの時間を比較した結果をとっておくための変数です。

先ほどの時間のデータを格納するための変数の場合と同様に「ここにどんなデータが入る可能性があるか？」ということを考えると，ここに入るのは「一致したか」，「一致しないか」の2通りの状態だけです。そこでこの変数の名前は「時間一致」という名前にします。

名前は付けましたが，この2通りの状態を記憶するにはどのような変数が必要かと考えれば，1ビットのフラグが必要なことがわかります。しかしながら，1ビットの大きさの変数（フラグ）を定義するにはflagというキー・ワードはありません。そこで*emIC-17K*では，booleanというキー・ワードを用意しました。

フラグを設定する場合も，addr ()指定子を使用します。変数を指定する場合と異なるのは，バンクとアドレス以外にビット位置を指定することです。

［フォーマット］

<型名>　<変数名>　addr（<バンク番号>．<アドレス>．<ビット位置>）；

*emIC-17K*で、データ・メモリにフラグのための領域を確保するプログラムを書くと、次のようになります。

```
<型名>    <変数名>    addr（<バンク番号>．<アドレス>．<ビット位置>）；
boolean   時間一致    addr(0.0x07.0)；
```

これですべての箱の準備は終了しました。

2.2.2 関　数

2.2.1　データ定義において変数の準備はできました。次は関数（function）をプログラムします。関数は、いくつかの処理（文の集まり）に付けた名前のこととここでは定義しておきましょう。

*emIC-17K*のプログラムは、いくつかの「文」でできています。特に指定しなければ、文を書いた順番（上から下）にプログラムは実行されます。

```
a=1；
b=a；
c=f()；
```

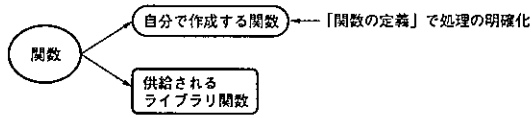
という3つの文は*emIC-17K*のプログラムです（今回は使用しません）。文はセミコロン（；）で終わります。

```
a=1;b=a;c=f()；
```

1行にいくつかの文（セミコロンで区切る）を書くこともできますが、普通は1行に1つの文だけを書きます（あとでリストを見やすくするためです）。

*emIC-17K*のプログラムの大部分は代入式と、関数呼び出しで構成されます。上の例では、f()が関数呼び出しです。必ずしも関数の結果をどこかに代入しなくてもかまいません。つまりf()；のようにも記述可能なのです。

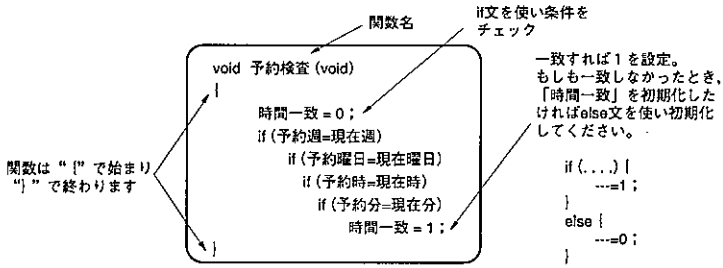
関数は2種類あり、供給されるライブラリ関数、自分で作成する関数があります。自分で関数を作る際にはどういう処理を行うかを決めておかなければなりません。



それでは，要求仕様に基づき関数を作成してみましょう。作成する関数の基本は，2つの数値の比較を行うことです。それには，「if」を使用します。

ifを使うと，ある式（比較）が真のときに行う処理と偽のときに行う処理を分けて書けます。予約があった場合（現在時刻と予約時刻が同じ場合）にすることは，「時間一致」フラグを立てることです。今回は「フラグの内容が1ならばフラグが立っている」と決めます。したがってフラグを立てるために1という数値を代入します。今回は予約がなかった場合（現在時刻と予約時刻が異なる場合）にすることはありません。

このフラグの初期化をついでにやっておきましょう。予約があった場合にフラグを1とする関数で，このフラグを初期化するには，0を代入しておきます。そうすれば，この関数を実行する前にフラグの内容が何であろうと，この関数を実行したあとでフラグの内容が0ならば「予約がなかった」，1ならば「予約があった」と判断できます。



●メイン関数

C言語のプログラムには，必ずmain()というメイン関数があります。この関数は，プログラムの一番最初に実行される特別な関数で，あらかじめ名前が決められています。したがって，プログラムは，この関数の中から呼ばれる関数や処理として記述されることになります。

普通の関数だけでメイン関数がないとプログラムはどの関数から実行していいのかわからなくなりますので，今回もメイン関数を書いておきましょう。ここで作成するのは，「予約検査」という関数を呼び出すためのダミーのメイン関数です。

●そのほかの関数

ほかに，これから比較する「現在時刻」と「予約時刻」にテストのためのデータを入力する関数を作っておきましょう。これはディバグ用の代入文があるだけの関数です。まずは，現在時刻と予約時刻に異なる値を代入し，「時間一致」フラグが変化しない（“時間一致=1;”を実行しない）ようにします。ディバグ用の現在時刻を設定する関数の名前をgettime()，予約時刻を設定する関数の名前をsetapo()とでもしておきましょう。

```
void gettime(void) {
    現在週 = 1;
    現在曜日 = 2;
    現在時 = 3;
    現在分 = 4;
}

void setapo(void) {
    予約週 = 5;
    予約曜日 = 6;
    予約時 = 7;
    予約分 = 8;
}

void main(void) {
    getime();
    setapo();
    予約検査();
}
```

ここで現在週から予約分までの変数に1から8までのデータを代入しておきます。ここで，1から8までの数値を入力するのは特に意味はありません。現在時刻と予約時刻が異なる値であればなんでもいいので，どの変数に何の値を入力したかを簡単に把握できるように順番に入れただけです。

繰り返しますが，メイン関数とgettime()，setapo()の3つの関数はディバグ用のダミーの関数です。普通は現在時刻の設定をプログラムでは行いませんし（ハードウエアから取得する），予約時刻はユーザが入力する（同様にハードウエアから取得する）ものです。

それではここでソース・プログラムの全文を見てみましょう。

```
#define 一致した          (1)
#define 一致しない        (0)

void    main(void);                     //プロトタイプ宣言
void    予約検査(void);                 //プロトタイプ宣言
void    gettime(void);                  //プロトタイプ宣言
void    setapo(void);                   //プロトタイプ宣言

        nibble  現在週                  addr(0.0x00);
        nibble  現在曜日                addr(0.0x01);
        byte    現在時                  addr(0.0x02);
        byte    現在分                  addr(0.0x04);

        nibble  予約週                  addr(0.0x10);
        nibble  予約曜日                addr(0.0x11);
        byte    予約時                  addr(0.0x12);
        byte    予約分                  addr(0.0x14);

        boolean 時間一致                addr(0.0x07.0);

void 予約検査(void){
        時間一致 = 一致しない;
        if(予約週 = 現在週)
            if(予約曜日 = 現在曜日)
                if(予約時 = 現在時)
                    if(予約分 = 現在分)
                        時間一致 = 一致した;
}

void setapo(void){
        予約週 = 5;
        予約曜日 = 6;
        予約時 = 7;
        予約分 = 8;
}

void gettime(void){
        現在週 = 1;
        現在曜日 = 2;
        現在時 = 3;
        現在分 = 4;
}

void main(void){
        getime();
        setapo();
        予約検査();
}
```

2.3　コンパイル

2.3.1　エディット

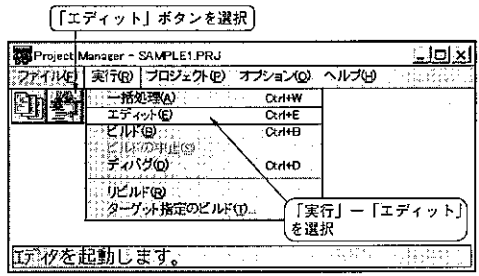
それでは，2.2　ソフトウエア設計で設計したプログラムをエディタにより入力してみましょう。

エディタは，お手持ちのものをご使用していただいてもかまいませんが，この入門編は，プロジェクト・マネージャに添付されているエディタを使用して説明していきます。

まず，エディタを使用するためにプロジェクト・マネージャを起動します。

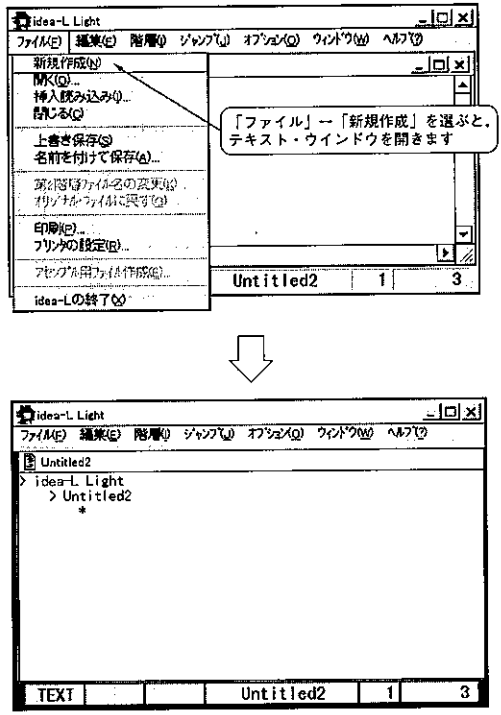
次に，「実行」－「エディット」メニューか「エディット」ボタンを選択します。

図2－1　「実行」－「エディット」メニュー



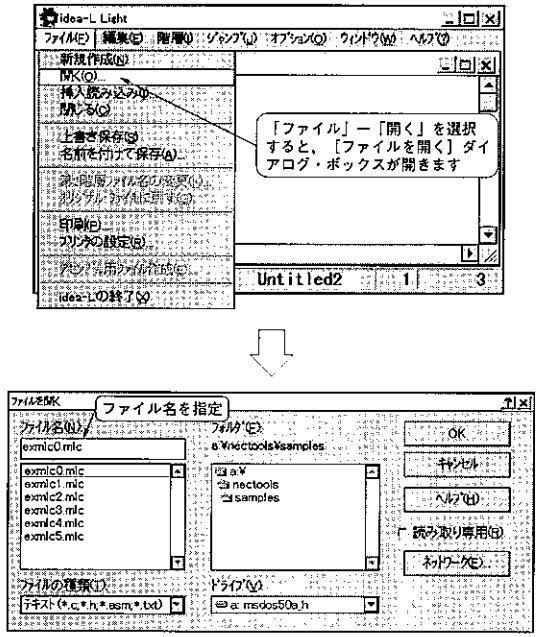
エディタが起動され，「ファイル」ー「新規作成」メニューを選ぶと，エディットのためのテキスト・ウインドウが開きますのでソース・プログラムを入力してください。

図2－2　エディット画面（新規）



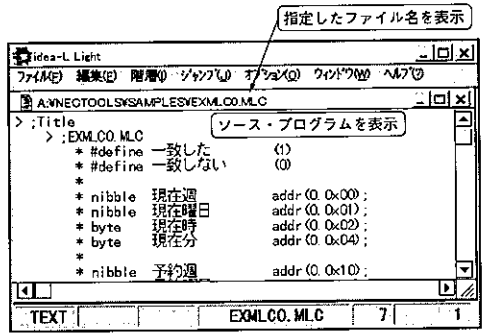
このとき，すでに存在しているソース・ファイルを修正する場合は，「ファイル」－「開く」メニューにより，ソース・ファイルを読み出して修正してください。

図２－３　ファイル名指定画面



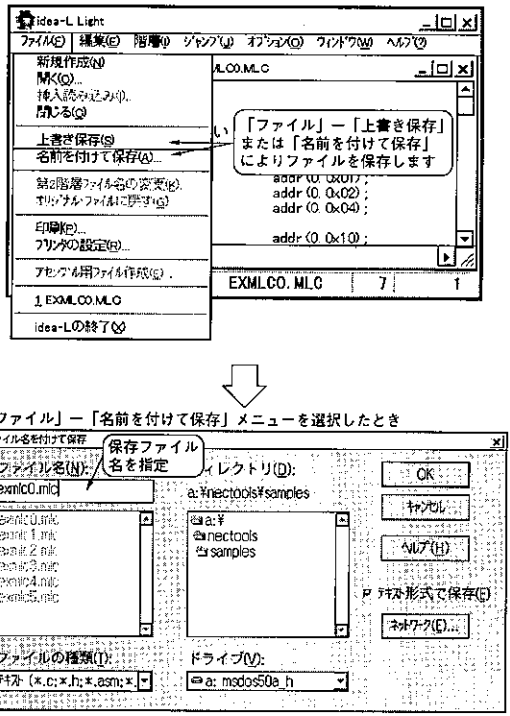
〔ファイルを開く〕ダイアログ・ボックスで指定したソース・ファイルがテキスト画面上に開かれます。

図2－4　エディット画面（修正）



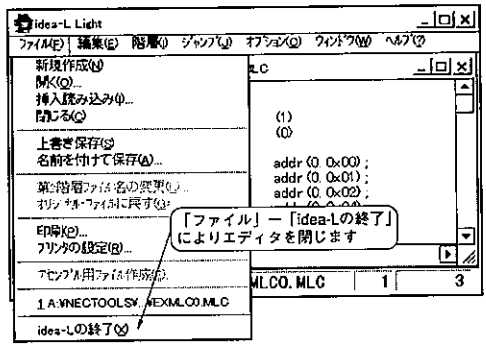
入力が終了後，「新規作成」時は「ファイル」ー「名前を付けて保存」メニュー，「開く」により読み出したときは「ファイル」ー「上書き保存」または「ファイル」ー「名前を付けて保存」メニューによりファイルを保存してください。

図2ー5　ファイルの保存



ファイルの保存が終了したら、あとはエディタを閉じるだけです。

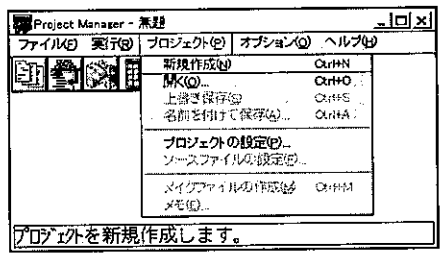
図2－6　エディタの終了



2.3.2　プロジェクトの設定

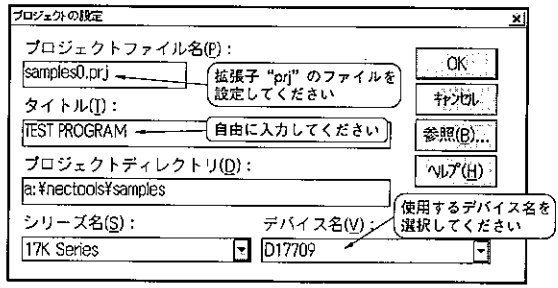
エディタが終了したらプロジェクト・マネージャに戻りますので，プロジェクトの設定を行います。「プロジェクト」－「新規作成」メニューを選ぶと，［プロジェクトの設定］ダイアログ・ボックスが開きます。

★　図2－7　「プロジェクト」－「新規作成」メニュー



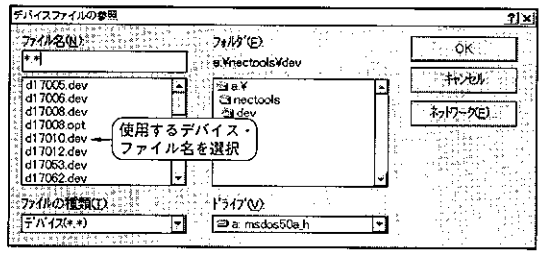
プロジェクトの設定をします。

図２－８　プロジェクトの設定



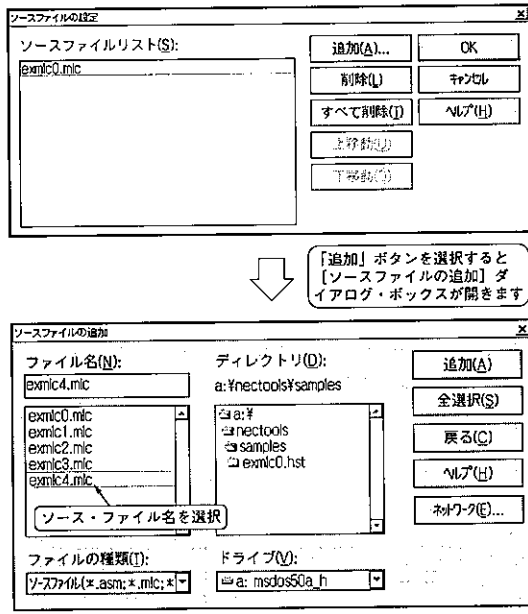
★　デバイス・ファイルを新規で登録する場合は，使用するデバイス名を“デバイス名”に入力します。登録されていないデバイス・ファイル名が入力されると［デバイスファイルの参照］ダイアログ・ボックスが開きますので，デバイス・ファイル名を選択してください。次からは“デバイス名”のドロップダウン・メニューに登録されていますので，それを選択してください。

図２－９　デバイス・ファイルの設定



次にソース・ファイル名の設定です。「プロジェクト」－「ソースファイルの設定」メニューを選ぶと，［ソースファイルの設定］ダイアログ・ボックスが開きますので，使用するソース・ファイルを選択してください。

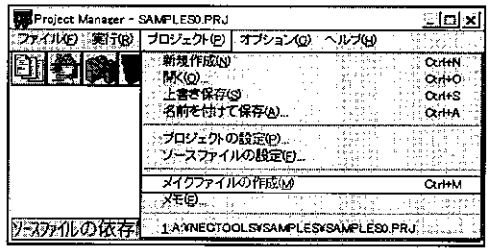
図2－10　ソース・ファイルの設定



ソース・ファイルの設定が終わったらメイク・ファイルの作成を行います。

「プロジェクト」－「メイクファイルの作成」メニューを選ぶと，自動的に作成されます。これで作成したプログラムをコンパイルすることが可能になります。

図2－11　メイク・ファイルの作成

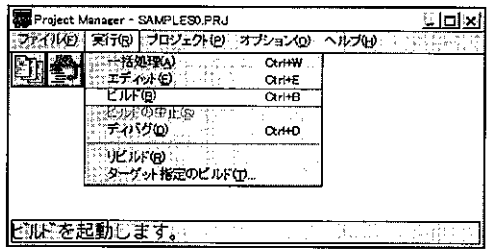


2.3.3　ビ ル ド

メイク・ファイルが作成できたら次はコンパイル作業です。

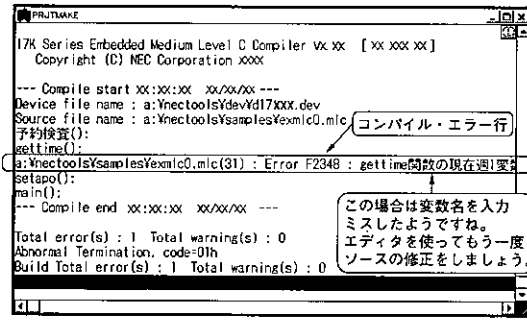
「実行」－「ビルド」メニューによりビルドを実行します。これにより，コンパイル→リンクの一連作業が行われます。

図2－12　ビ ル ド



コンパイル時にエラーがあった場合次のような画面が表示されます。

図２－13　ビルド（エラー）結果表示画面



コンパイルの段階では，ソース・プログラムの文法的なチェックが行われます。タイプ・ミスや文法的に許されない記述があると，エラー・メッセージが出力されます。しかしここで出力されるのは，文法的な間違いのみです。論理的な間違いなどは発見されません。

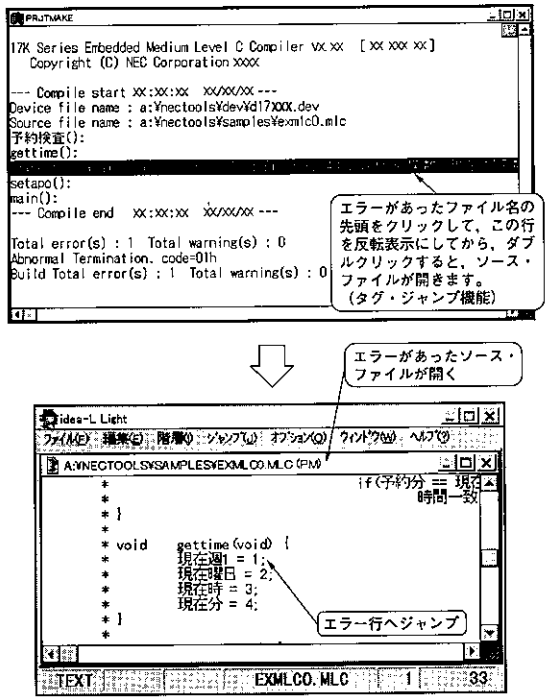
エラー・メッセージが出力された場合は，そのメッセージの行頭にエラーの発生したファイル名と行番号が出力されます。それに続いてエラーと判断された原因を出力します。

＜ファイル名＞（行番号）：＜エラー番号＞：＜エラー内容＞

コンパイラが出力したメッセージをたよりに，エラーとなってしまう箇所をなくし（エディタを使ってソース修正），再ビルドしてください。

ソース修正を行う場合は，タグ・ジャンプ機能を使用すると便利です。タグ・ジャンプ機能はリンク・エラー時には使用できませんので注意してください。

図2－14　タグ・ジャンプ機能



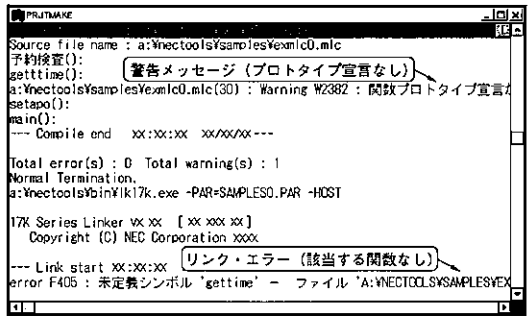
エラーの原因は1つでも，複数のエラー・メッセージが出力される場合があります。たとえば，変数の定義で名前をタイプ・ミスすると，その変数を使用しているすべての箇所で「変数名が定義されていない」というエラー・メッセージが出力されるかも知れません。逆に，出力されたエラー・メッセージは1つなのに，その原因はソースの複数の箇所にあったり，1つのソースの修正を行うと，その修正がほかの箇所に影響し，大幅な修正をしなければならないこともあります。

もしコンパイラがエラーを発見したら，「なぜその間違いがソース・プログラムに入りこんだのか？」を考え，「同じような間違いをしている可能性はないか？」，「この修正をすることで影響される範囲は？」を考慮するようにすると，早い段階からバグの少ないプログラムを作成できるようになります。

●エラーは出力されましたか？

先ほど掲載したプログラムにはエラーがあります。したがってエラー・メッセージが出力されます。エラー・メッセージは次のようなものです。

図２－15　エラー表示画面



これ以外のエラー・メッセージが出力された場合は、それはこのプログラムのせいではありません。どこかでタイプ・ミスしているか、コンパイルを行う環境が異なるためだと思われます。

このマニュアルのとおりに環境を設定しましたか？

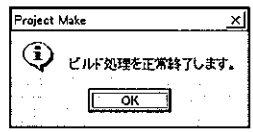
このエラーはファイル名「emlc0.mlc」の30行目で発生しています。このエラー・メッセージは、それが*emlc-17K*のプログラムの中でどういう意味を持つ言葉であるのかが示されていないということです。47行目はgetime(); になっています。

これは、その前に定義した関数gettimeを呼び出しています。定義した関数は「gettime」で、「getime」ではありません。

getime(); をgettime(); に変更して再ビルドしてください。

ビルドして正常終了すると次のメッセージが表示されます。

図２－16　終了メッセージ



2.4　ディバグ

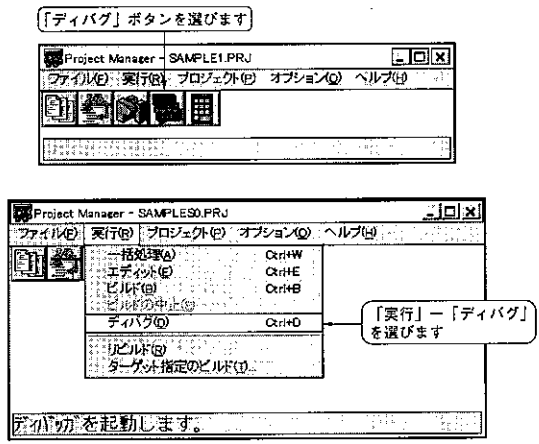
それでは、2.3　コンパイルでビルドしたプログラムをディバグしてみましょう。

2.4.1　*SIMPLEHOST*の起動

最初に、*SIMPLEHOST*を起動します。

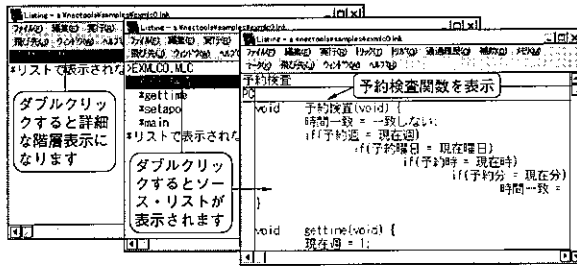
プロジェクト・マネージャの「実行」－「ディバグ」メニューか「ディバク」ボタンを選びます。

図2－17　「実行」－「ディバグ」メニュー



*SIMPLEHOST*が起動されると、Listing画面が開きます。

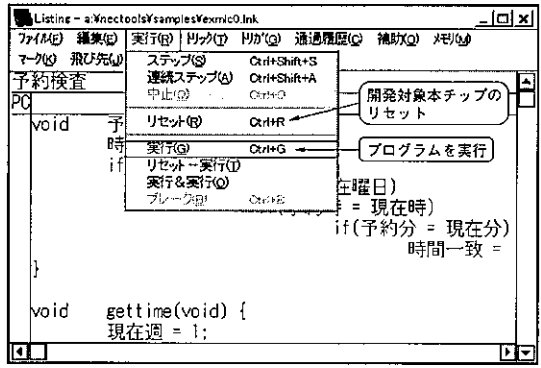
図2－18　Listing画面



2.4.2　プログラムの実行

たいていのディバッガには，「プログラムの実行」というコマンドがあります。当然，*SIMPLEHOST*のメニューにも「実行」というメニューがあります。

図2－19　「実行」－「実行」メニュー



また，*SIMPLEHOST*のようなインサーキット・エミュレータを使用するディバッガの場合は，「リセット」という項目もあります。

インサーキット・エミュレータを使用したディバグは，非常に詳細なプログラムの動作やそのときのシステムの状態を観察することができます。そのようなことができる環境で，ディバッガを立ちあげた直後とターゲット・システム電源を入れた直後の状態が異なるのではせっかくの機能も宝の持ち腐れです。

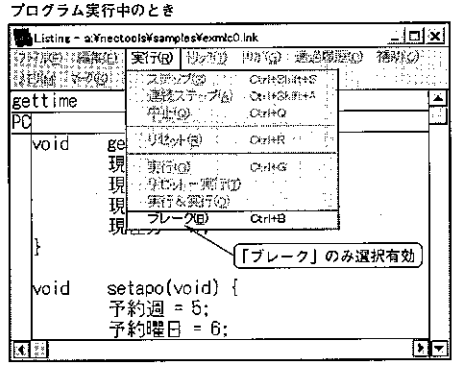
そこで，本チップをターゲット・システムで電源を入れた場合と同じような状態にするためにプログラムを実行する前に「リセット」を行います。これは，開発対象の本チップのリセットです。Windowsや*SIMPLEHOST*が動いているコンピュータのリセットではありませんので，安心してリセットを実行できます。

リセットしたら次に行うのはプログラムの「実行」です。「実行」メニューの中にある「リセット→実行」を選べば，「実行」メニューの「リセット」と「実行」メニューの「実行」という2つの動作をいっぺんにやってくれます。ここでは，

「実行」－「リセット→実行」メニューを選びます。

本チップをリセットするとプログラム・カウンタは0番地を指します。ここからプログラムが開始されます。ソース・リスト上でいうとメイン関数の先頭からプログラムは実行されます。プログラム実行中は，「実行」メニューの「ブレーク」のみ有効になり，「ブレーク」以外は選べません。また，メニュー・バーの「実行」，「ウインドウ」，「ヘルプ」以外は，淡色表示になります。

★　図2－20　「実行」－「ブレーク」メニュー



プログラムがどこまで動いているかこのままでは分かりません。そこで，一度このプログラムを止めてから，どこまで実行されているのかを確認してみましょう。

「リセット→実行」や「実行」を使って動かしたプログラムを止めるには，「実行」－「ブレーク」メニューを使います。

★　「実行」－「ブレーク」メニューを選びます。

メニュー・バーの表示が元に戻りました。この操作を強制ブレークといいます。「リセット→実行」や「実行」では，*SIMPLEHOST*は，ブレーク条件が成立したらプログラムを止めるという動作をします。今はブレーク条件は特に設定しなかったので強制的にブレーク（中断）させたわけです。

それでは，順番にディバグを行っていきましょう。

2.4.3　変数の表示

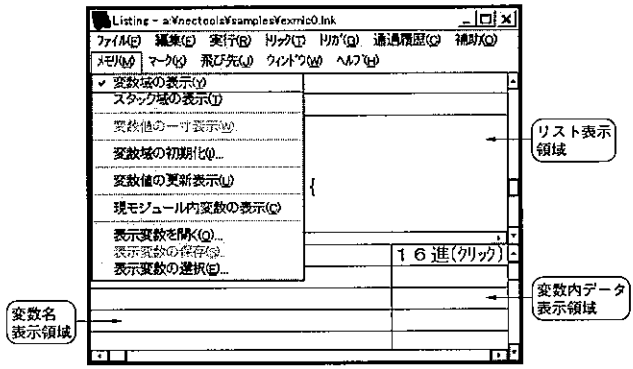
今，ディバグ用の関数で現在時刻と予約時刻に異なる値を代入して，「時間一致」が0になるように設計してあります。では，一度プログラムを実行したらこの「時間一致」フラグはどうなっているでしょうか？それよりもこのフラグを変更する文は実行されているのでしょうか？

「時間一致」はデータ・メモリに領域を確保した(フラグの大きさの)変数です。この変数の値を確認する方法としては，2通りあります。1つは，Memoryを使用する方法で，もう1つはListingの「変数域」を使用する方法です。ここでは，Listingを使って変数の値を確認します。

●変数域の表示

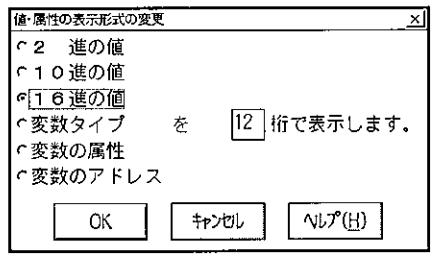
Listingをリスト画面にします。「＊」の表示のある行をダブル・クリックすると，“予約検査()”関数のある行を開きます。リスト画面に変ったら，「メモリ」メニューの「変数域の表示」を選びます。

図２－21　変数域の表示



Listing画面が３つに分割されます。１つは今までのリスト画面，もう１つは「変数名」という欄，最後の１つは「変数名」に対応した「16進(クリック)」という欄です。まず「16進(クリック)」をクリックしてみましょう。

図２－22　［値・属性の表示形式の変更］ダイアログ・ボックス



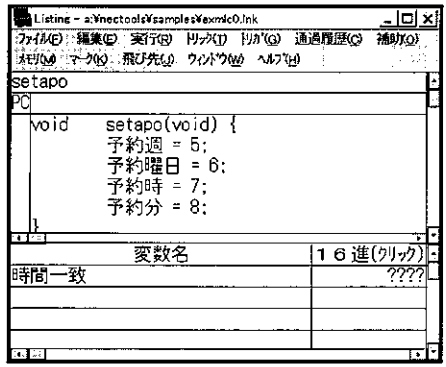
すると［値・属性の表示形式の変更］ダイアログ・ボックスが表示されますので，「２進の値」を選んでおきましょう。選んだあとに「OK」ボタンを選択するとこのダイアログ・ボックスを閉じ，欄のタイトルが「２進（クリック）」に変わります。

*SIMPLEHOST*では，これらの２つの欄を合わせて「変数域」と呼んでいます。変数域の左の欄（「変数名」の欄）に，プログラムで使用している変数名を入力すると，変数域の右の欄にその変数の値が表示されます。

●変数域に変数名を入力する

変数域の空欄の部分を選択すると，その位置に（文字入力のための）カーソルが表示されます。そこでキー・ボードから変数名を入力し，リターン・キーを押すとその値が表示されます。まだ値が定まっていない（プログラムを実行していない）場合は「分かりません」という意味で「？？？？」という表示が行われます。変数名を間違えて入力すると「XXXX」という表示が行われます。ではここで，値を調べる変数の名前，「時間一致」を入力しておきましょう。

図２－23　変数の表示



変数域に変数名を入力し，リターン・キーを押すとその右側に値が表示されます（変数名を入力する前に２進数で表示するように設定しておいたので，２進数で表示されます）。

ここで，「時間一致」フラグの値は，

0にはなっていません。1になっています。

ということは，“時間一致＝１；”の文は実行されているということになります。

2.4.4　実行チェック

もう一度プログラムを動かして“時間一致＝1；”が実行されるところを見てみましょう。

●ステップ

「リセット一実行」メニューでプログラムを実行したら、いつまでたっても止まりませんでした。“時間一致＝1；”の文がどの時点で実行されたのかも分かりませんでした（ブレーク条件を設定しなかったので）。

ステップを使うと、1ステップずつプログラムを追っていくことができます。

とりあえず、「実行」－「リセット」メニューを選び、さらに「実行」－「ステップ」メニューを選択します。（リセット直後のプログラム・カウンタを示すアイコンは、メイン関数行にあります）。

ステップ実行をするとアイコンは次の行に移ります。次の行は関数ですので、もう一度「ステップ」メニューを選ぶと、その関数を定義したところにジャンプします。このように「ステップ」メニューは、プログラムを1ステップずつ目で確認しながら追うのに最適です。

●連続ステップ

でも関数1つくらいなら1ステップずつ追うのもいいですが、たくさんある場合はいささか大変です。そこで、もう少し便利な「連続ステップ」メニューを選びます。

手の形のアイコンがPC欄を「タッタッタッ」と動きます。画面の下の方までいくと「スッ」と画面がスクロールします。連続ステップの実行中は、メニュー・バーの「実行」メニューの「中止」のみ有効になります。連続ステップを止めるときにこの「実行」－「中止」メニューを選びます。

★

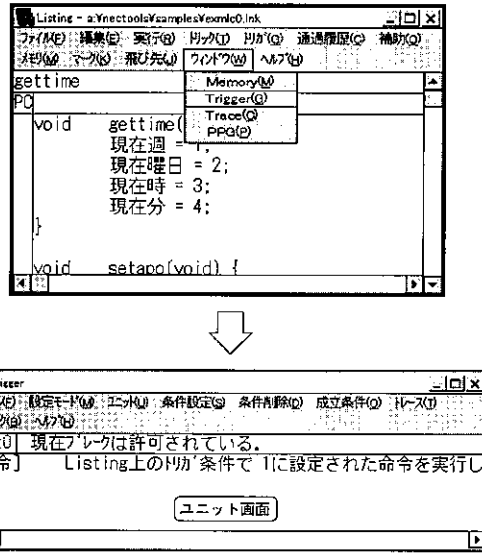
2.4.5　トリガ条件の設定

連続ステップを行っていると、プログラムの動作がダイナミックに見えて楽しいのですが、“時間一致＝1；”の文を実行したのかしていないのか分からないうちに通り越して、プログラムの終わりにまで行ってしまいます。これを“時間一致＝1；”の文を実行したときにストップさせる方法があります。それには、Triggerの機能を使います。

●Triggerを起動する

Triggerの起動は簡単です。メニュー・バーの「ウインドウ」－「Trigger」により起動されます。Listingのようにファイルを指定したりする必要はありません。Triggerでは、トリガ情報ファイルというファイルを使用しますが、起動のときには使用しません。

図2－24　Trigger画面



起動したばかりのTrigger画面には、ユニット画面が表示されています。

ここには、各条件項目ごとのブレーク条件が表示されています。

●Triggerで何をするか

Trigger画面では、ブレーク条件の設定とトレース条件の設定ができます。

17Kシリーズのインサーキット・エミュレータであるIE-17Kは，ブレーク条件，トレース条件を細かく設定できます。*SIMPLEHOST*のTrigger機能は、このIE-17Kのブレーク条件，トレース条件の設定を簡単に行えることです。

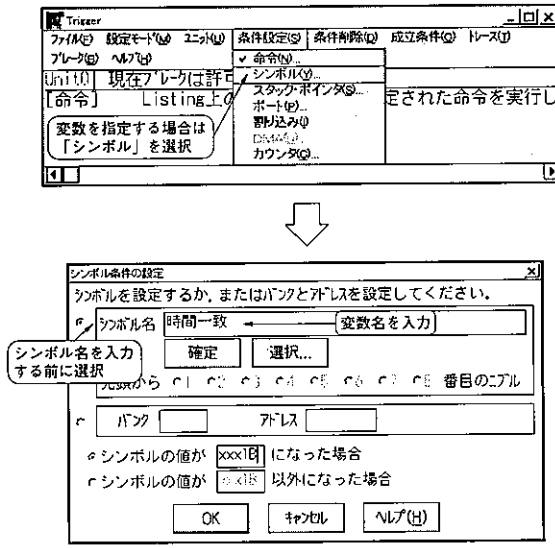
今回のサンプル・プログラムで行うことは、「時間一致」という変数に対する書き込みを監視して、この変数に書き込みを行ったらプログラムをストップさせることです。このようにプログラムをストップさせるための条件を設定することを「ブレーク条件の設定」といいます。

●「時間一致」がセットされたところでブレークする

まず最初に確認しておきましょう。「時間一致」というフラグをセットするということは、変数を書き換えるということです。換えるには、Triggerの「条件設定」－「シンボル」メニューを選択します。すると、［シンボル条件の設定］ダイアログ・ボックスが表示されました。このダイアログ・ボックスでは，変数名とどの値になった場合にブレーク条件とするかを設定します（変数名以外でもバンクやアドレスを数値で指定することもできます）。

「確定」ボタンを選択すると，指定された変数がどの値になった場合にブレークをするかを設定するテキスト・ボックスに自動的に「XXX1B」という値が入ります。通常はここに「2」とか「3」とかいう数値を入力するのですが，今回の変数はフラグ型ですので，そのフラグが定義されているビットだけ参照するように自動的に設定されます。

図2－25　メモリ書き込み条件の設定



これでダイアログ・ボックスの設定は終了ですので，「OK」を選択します。すると，ダイアログ・ボックスは消え，Trigger画面の［シンボル］という項目には今ダイアログ・ボックスで設定した内容が日本語の文章で表示されます。

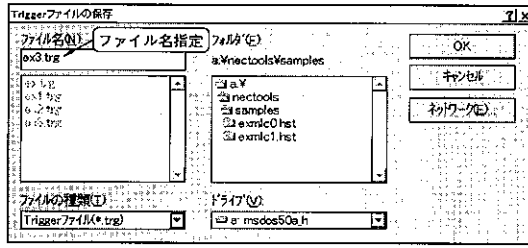
●ブレークする

これでブレーク条件の設定は終わりですが，最後に設定した条件でブレークが許可されていることを確認します。メニュー・バーの下の表示が「Unit0　現在ブレークは許可されている」となっていることを確認してください。もし万が一「許可されていない」と表示されていたら，「ブレーク」－「ブレーク条件とする」メニューを選ぶことにより許可されます。

●トリガ条件の保存

設定した項目が少ないので，また今度同じ設定が必要になったら，また同じ操作をすればいいのですが，とりあえず今設定したトリガ条件を保存しておきましょう。Trigger画面で設定したブレーク条件などはトリガ情報ファイルに保存することができます。「ファイル」－「名前を付けて保存」メニューを選びます（すでに存在するファイルに上書きしたい場合は，「ファイル」－「上書き保存」メニューを選択してください）。

図2－26　トリガ条件の保存



●アイコンにする

ブレーク条件の設定も済みTrigger画面も不要なので，これをアイコン表示にして画面の隅に片づけておきましょう。ウィンドウのコントロール・メニューの最小化を選んでもいいのですが，ここでは簡単に，ウィンドウの右上にある最小化ボタンを選択します。これでウィンドウがアイコン表示になります。

2.4.6　バグの発見

ブレーク条件が設定できたので，「リセット一実行」メニューを選びます。ブレーク条件が成立したらプログラムは勝手にストップします。

さて，目的の所（“時間一致＝１；”）でプログラムをストップすることができるようになったのはいいのですが，なぜ“時間一致＝１；”を実行してしまうのかがいまだよく分かりません。

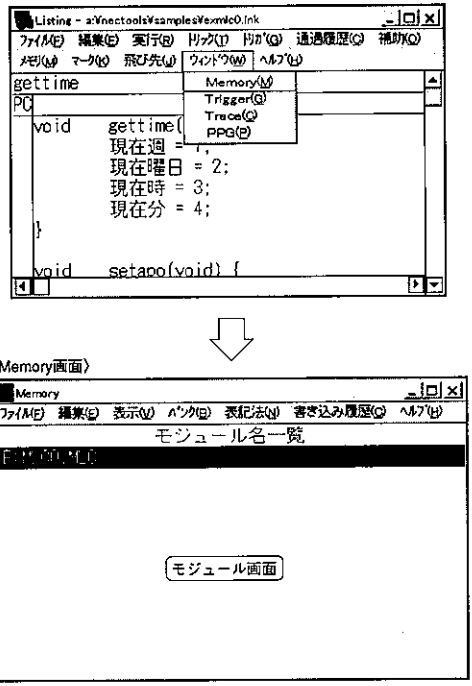
この文の実行をコントロールする（実行するかしないかを決定する）のは，if文です。ifで，２つの値の比較を行い，それが４組とも一致したらこの文を実行するように設計したつもりです。この４組の値はすべて異なるデータをセットしてあります。では，このifを実行するときにこの変数の値はどうなっているのでしょうか？これを確認できれば，予想したとおりに動作しない原因がつかめるかも知れません。

先ほど，変数の値を確認する方法には２つあると説明し，「時間一致」の確認には変数域を使用しました。ここでは，Memoryを使用してみましょう。

●Memoryを起動する

Memoryの起動は簡単です。「ウインドウ」－「Memory」メニューにより起動されます。

図2－27　「Memory」画面

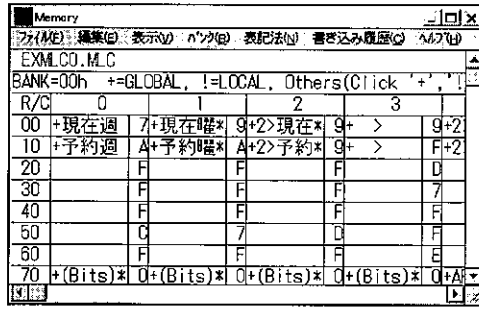


起動したばかりのMemory画面には，モジュール画面が表示されています。ここには，ソース・ファイルの名前が表示されているだけです。ここで，「表示」メニューを選ぶと，「メモリ」，「フラグ」，「コントロール・レジスタ」，「拡張RAM」などのメニューが表示されます。今回はこの中で「表示」－「メモリ」メニューを選択します。

これを見ると，変数名らしきものがいくつか表示されています。問題の「現在時刻」と「予約時刻」の8つの変数もあるようです（17Kシリーズのアーキテクチャを少しでもご存じの方ならば一目で理解できるデータ・メモリのメモリ・イメージです）。

では，変数名と対になっている欄にある数値はその変数の値では・・・という想像は簡単に付くのですが，肝心の予約時刻が設定した値になっていません。

図2－28　データ・メモリの表示



今プログラムは“時間一致＝１；”の文を実行しストップしています。当然，この関係を実行する前にテスト・データを設定するためのダミーの関数も実行しているはずです。テスト・データには１から８までの数値を入力しました。しかしテスト・データの値を見ると，「予約時刻」と「現在時刻」が同じ値になってしまっています。

「予約時刻」の値が「現在時刻」の値に変化している

●「予約時刻」が変わっている

このプログラムで「予約時刻」を使っているのは，setapo()関数と予約検査()関数の２箇所でした。setapo()関数ではディバグのためのデータを「予約時刻」に設定しているだけなので問題なさそうです。

予約検査()関数で「予約時刻」を使用しているのは，「現在時刻」との比較のところでした。本来この関数ではデータ設定はありえません。どうやらこの辺に「予約時刻」を変更している（バグがある）可能性があります。

予約検査()関数の比較がおかしい

ということになりそうです。

●数値の比較

先ほど作成した「予約検査」の関数では，２つの数値の比較を次のようにやっていました。２つの数値「予約週」と「現在週」が等しければというプログラムのつもりです。

if（予約週＝現在週）

実は*emIC-17K*における２つの数値の比較はこのように記述しません。次のように書きます。

```
if（予約週 == 現在週）
```

等号(=)単独で記述すると，それは代入文として処理されます。つまり先ほどのプログラムでは，「現在週」と「予約週」を比較するのではなく，「予約週」に「現在週」を代入していたのです。

*emIC-17K*では，2つの数値の比較に等号を2つ重ねたもの「==」を使用します。これは，結構よくあるバグで知名度も高いので，最初のソース・プログラムを見てすぐに気付いた方もいらっしゃると思います。

2.5　ソースの修正

●比較を修正する

間違いが分かったら早速修正です。*SIMPLEHOST*では，「ファイル」－「ソース修正」メニューがあります。これは，プロジェクト・マネージャに添付されているエディタと連動し，ソース修正を可能にしています。

しかしながら，ソース修正が終了してもコンパイル／アセンブルが行われなければ新しいオブジェクトができていないので，続けてディバグできません。

それではどうしたらよいのでしょう。実はこの一連の動作を管理しているのがプロジェクト・マネージャなのです。

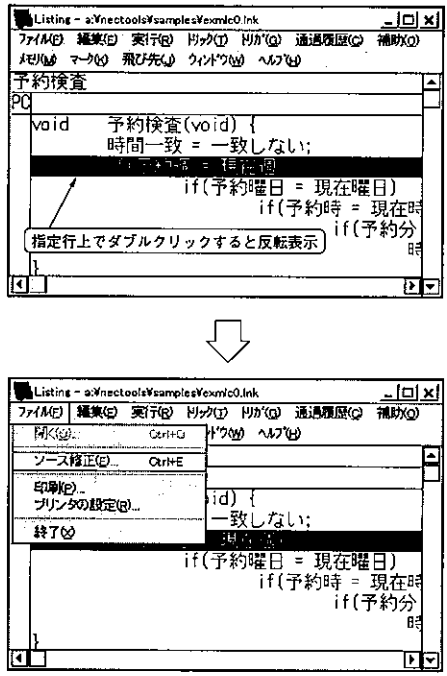
では，順を追って説明をしていきます。

まず，間違いである「=」を「==」に修正します。

「=」の行にマウス・カーソルを動かし，ダブルクリックします。

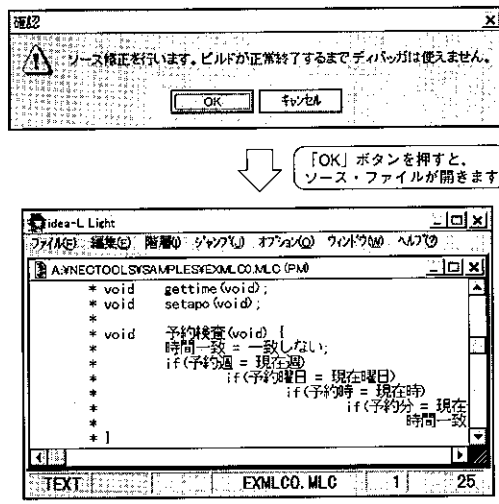
マウス・カーソルをソース・イメージの上に持って行きダブルクリックするとその行が反転表示になります。ここで，「ファイル」－「ソース修正」メニューを選択すると，ディバッガからプロジェクト・マネージャに対してソース修正の要求が出されます。

図2－29　ソース修正画面



プロジェクト・マネージャは，この要求を受けると指定されたファイル名でエディタを開きます。ソースを修正しましょう。

図2－30　エディタ画面



キーボードから「=」を入力します。

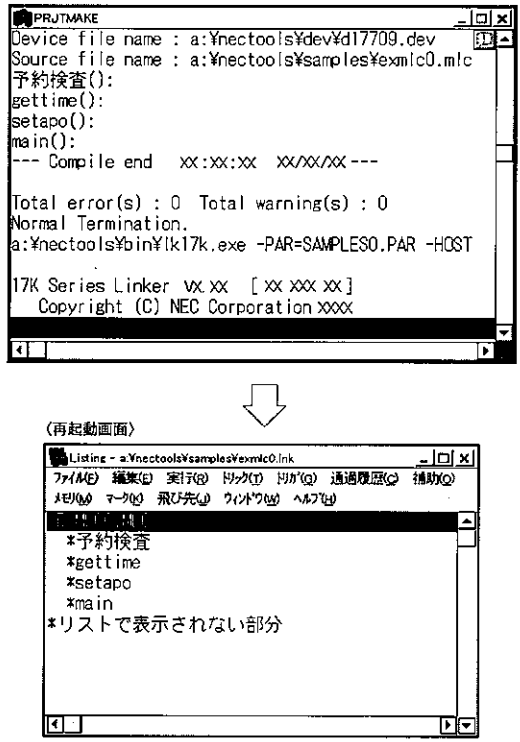
数値の比較は4回行っていますので，これを4回繰り返します。修正したプログラムは次のようになります。

```
if（予約週 == 現在週）
  if（予約曜日 == 現在曜日）
    if（予約時 == 現在時）
      if（予約分 == 現在分）
```

ソースの修正が済んだらファイルに保存します。保存されるとプロジェクト・マネージャはオブジェクトを作るためにビルド（コンパイル／アセンブル→リンク）に入ります。

図2－31　再ビルド画面

〈再ビルド画面〉



これでソースの修正は終わりました。再修正した部分が正しく動作するかどうか確かめてみましょう。

●実行して確かめる

さあ，「実行」－「連続ステップ」メニューによりプログラムを実行して確かめてみましょう。

実行後の変数域を見てみましょう。「時間一致」のフラグは，「0」のままでしたか？

今回は意図したとおりに動いたようです。

●ifの条件を1回にする

ほかにプログラムの修正をする箇所はないでしょうか？「予約検査()」の関数でifを4つもネストする（重ねる）のはあまりスマートとは思えません。そこで，このifの判定の方法を変えてみたいと思います。

「週」のデータが一致したら「曜日」のデータの比較をするというように順番に考えた結果がifの4重ネストでした。

この比較のあとですることは，

(全部一致した場合に)フラグを立てる

ことだけです。

したがって，「この4つの比較が全部一致した場合に（4つの条件式が真の場合に）フラグを立てる」と考えると，「条件式の論理的ANDをとってそれをif文の条件式とすれば，ifは1つで済む」ということになります。

*emIC-17K*で，条件式の論理的ANDをとったプログラムに変更する（ifの条件式を1回にする）とプログラムは次のようになります。

```
if((予約週 == 現在週)&&¥
   (予約曜日 == 現在曜日)&&¥
   (予約時 == 現在時)&&¥
   (予約分 == 現在分))
```

ここで「&&」は論理的AND演算子，「¥」は継続行の指定です。

●ついでにプログラムを読みやすくする

プログラム中に直接数値を記述することが必要な場合は少なくありません。ただし，経験のあるプログラマはそのような場合でも，処理するところに数値を記述することはできるだけ避けるようにします。

たとえば，今回のプログラムで言えば，“時間一致＝1；”“時間一致＝0；”とかいう場合の数字です。

せっかく変数に名前を付けプログラムを抽象化し，読みやすいプログラムを作成できるような環境が提供されているのに，ここであまり意味のない数値を持ち出してプログラムの可視性を落とすのは適切ではありません。

ここで「1」という数字はどういう意味を持つのか，それが分かるような名前を数字の「1」に付けるために，プログラムの先頭に次のような記述を付け加えます。

```
#define 一致した　（1）
#define 一致しない（0）
時間一致 = 一致しない；
if((予約週 == 現在週)&&¥
　(予約曜日 == 現在曜日)&&¥
　(予約時 == 現在時)&&¥
　(予約分 == 現在分))
　　時間一致 = 一致した；
```

#define は，「プリプロセッサ指令」と呼ばれるもので，普通の文とは区別されます。したがって，文のようにセミコロン(；)で終わりません。プリプロセッサ指令は，(物理)行の終わりをその指令の終わりと判断します。

「一致した」，「一致しない」の2つは，文法的には「記号定数」と呼ばれるものです。このプリプロセッサ指令以降，この記号定数を使用することができます。

たとえば「時間一致」フラグの初期化やセットならば，

```
時間一致 = 一致しない；
時間一致 = 一致した；
```

と記述することができます。今回は時間一致というフラグの状態を示すためだけに使用しますので，ユニークな（ほかにはない）名前を使用しますが，フラグのオン，オフを1，0で代用する場合，次のように記号定数を定義するのが一般的です。

```
#define　ON　　（1）
#define　OFF　（0）
```

ちなみに記号定数は，ほかの変数や識別子と区別する意味で大文字で記述するのが習慣となっています。

ifの条件判定を4重のネストから論理的ANDを使用した1回の判定に変更してみました。ソース・プログラムが格段に読みやすくなれば，それは意味があることですが，あの程度では可読性がそんなに向上したとは思えません。

このように修正した結果，実行スピードが上がったり，生成される命令が少なくなったりすれば，「修正は必要だった」ということができます。

●命令の表示

生成される命令の数やスピードなどは，アセンブラ・レベルで比較しないと正確には把握できません。

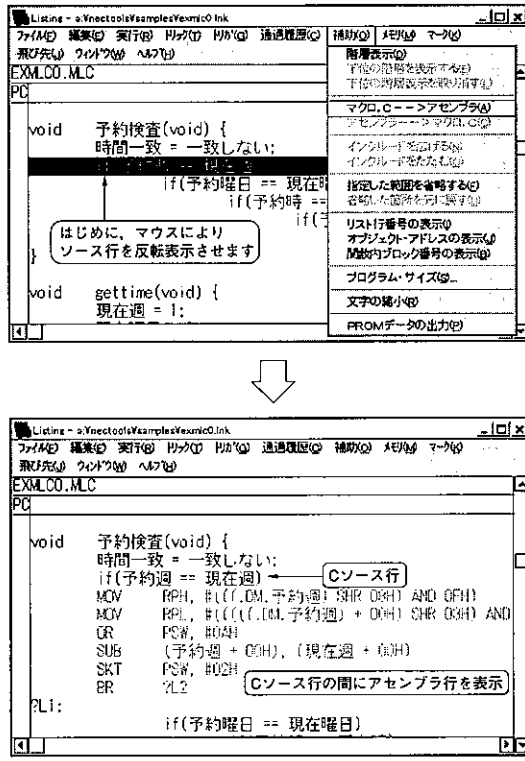
今，*SIMPLEHOST*は，*emIC-17K*形式のソース・イメージを表示しています。このソース・イメージをアセンブラの形式で表示できれば問題は解決できます。修正した行を含めてその近辺の何行かを範囲指定します（アセンブラで表示したい行の先頭にマウスを動かし，マウスのボタンを押したままマウス・カーソルを画面の下の方に動かします。するとマウス・カーソルが動いたところにある行は反転表示されます。これで，範囲指定ができたことになります）。

この状態で，

「補助」－「マクロ, C-->アセンブラ」メニューを選びます。

すると，今まで青色の文字で表示されていた*emIC-17K*のソース・イメージの各行に対して，複数行のピンク色の文字で新たな行が表示されます。

図2－32　アセンブラ表示



これが*emIC-17K*のソース・プログラムをコンパイルすると生成されるアセンブラ行です。これを元の状態に戻すには、*emIC-17K*のソース・イメージを含んで範囲指定を行い，次のような操作を行います。

「補助」－「アセンブラ-->マクロ、C」メニューを選びます。

アセンブラの形式で表示し、変更したメリットがあったかどうかの判断はみなさんにお任せします。このように中身の確認も行えます。

これで要求仕様を完全に満たしていませんがバグのないプログラムができました。

ソース・プログラムの全文を見てみましょう。

```
#define 一致した          (1)
#define 一致しない        (0)

void    main(void);                     //プロトタイプ宣言
void    予約検査(void);                 //プロトタイプ宣言
void    gettime(void);                  //プロトタイプ宣言
void    setapo(void);                   //プロトタイプ宣言

        nibble  現在週                  addr(0.0x00);
        nibble  現在曜日                addr(0.0x01);
        byte    現在時                  addr(0.0x02);
        byte    現在分                  addr(0.0x04);

        nibble  予約週                  addr(0.0x10);
        nibble  予約曜日                addr(0.0x11);
        byte    予約時                  addr(0.0x12);
        byte    予約分                  addr(0.0x14);

        boolean 時間一致                addr(0.0x07.0);

void 予約検査(void){
        時間一致 = 一致しない;
        if((予約週 == 現在週)&&¥
          (予約曜日 == 現在曜日)&&¥
          (予約時 == 現在時)&&¥
          (予約分 == 現在分)){
              時間一致 = 一致した;
        }
}

void setapo(void){
        予約週 = 5;
        予約曜日 = 6;
        予約時 = 7;
        予約分 = 8;
}

void gettime(void){
        現在週 = 1;
        現在曜日 = 2;
        現在時 = 3;
        現在分 = 4;
}

void main(void){
        gettime();
        setapo();
        予約検査();
}
```

2.6　データ表現の変更

ここまでは，「現在時刻」という1つのモノ（概念）を4つの変数に分けました。

ここで扱う「時間」には，「週」，「曜日」，「時」，「分」の4つの要素があったからです。しかし，もともとは1つのモノを4つの変数に分割して別々に命名，定義を行っていることは，プログラムの保守という面で好ましいことではありません。そこで，4つの要素があるということを変更することはできませんがプログラム言語上の「表現」を変えることで，「現在時刻」という「意味」を持たせることができます。言い換えれば，もう一度「現在時刻」を1つのモノ（データ）として扱えるようにプログラムを変更（データ表現の変更）するということです。

*emlC-17K*には，「構造体」というデータの構造を定義する機能があります。この機能を使うことで上で説明したようなことを実現できます。

●構造体を使って「現在時刻」を定義する

データの構造を定義する構造体で特徴的なことは，異なる形式のデータをまとめて扱うことができるということです。*emlC-17K*で構造体を使ってサンプル・プログラムの「現在時刻」を定義すると次のようになります。

```
struct  日時tag {
            nibble  週;
            nibble  曜日;
            byte    時;
            byte    分;
} 現在  addr(0.0x00);
```

構造体を使わない場合と何が違うかというと，まず新しいキー・ワード（予約語）が出ています。「struct」です。あとは新しい名前が2つ登場しています。「日時tag」と「現在」です。「日時tag」を構造体タグといい，「現在」を構造化変数（または構造体名）といいます。

この6行の定義で次の3つのことを行っています。

① データの構造を宣言する

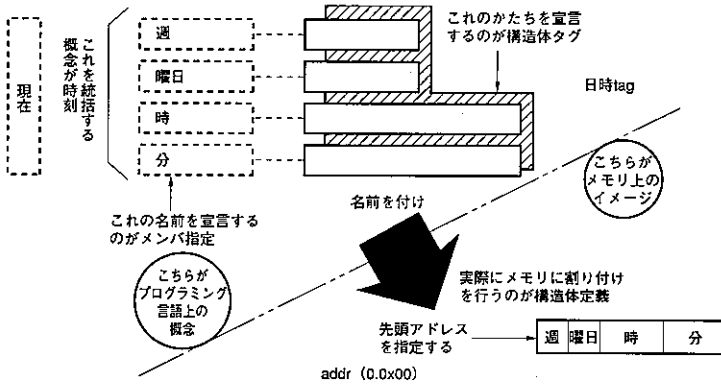
データ構造は4つの変数で構成され，それぞれニブル，ニブル，バイト，バイトの大きさを持ちます。*emlC-17K*では，構造体のそれぞれの変数名を「メンバ」と呼びます。ここでは，アドレスを指定しません。

② データの構造につけた名前を宣言する

このデータの構造を使うときの名前が構造体タグであり，先ほどの例では「日時tag」という名前でした。

③　構造体に名前をつけ，データ・メモリに割りつける

「現在」という名前を付け，それをバンク0のアドレス0x00番地に割り付けます。



構造体は，1つのデータの塊で扱うことも個々のメンバで扱うこともできるということです。

●構造体を使って参照する

今回のサンプル・プログラムで構造体の各メンバを参照する場合は，次のように行います（ここでは予約時刻の定義は変更していません）。

```
void　予約検査(void){
        時間一致 = 一致しない;
        if((予約週 == 現在．週)&&¥
          (予約曜日 == 現在．曜日)&&¥
          (予約時 == 現在．時)&&¥
          (予約分 == 現在．分))
                  時間一致 = 一致した;
}
```

このプログラムでは，構造体の名前は「現在」，各メンバの名前は「週」，「曜日」，「時」，「分」です。前回のプログラムとの違いは，構造体の名前とメンバの名前の間にピリオド（．）があります。

*emIC-17K*の文法では，ピリオドが演算子を示します。何をするのかというと，「演算」のイメージとは程遠い，「構造体のメンバを参照する」ということを行います。これでも文法上は「演算子」なのです。正式には，直接メンバ参照演算子といいます。

なぜ構造体の定義のところで，「週」や「曜日」などどこにでもある（もしかしたらこのあと「予約週」とバッティングするかも知れない）名前を使用したのかというと，構造体のメンバは一般に

<構造体名>．<メンバ名>

という参照のしかたをするからです。構造体のメンバに使用した変数の名前は通常の変数と区別して管理されます。したがって，通常の変数と同じ名前でメンバの名前を定義してもエラーにはならず，別の変数であると認識されます。

●配列を使う

現在のプログラムは，要求仕様を満たしていません。それは8個の予約を比較しなければならないのに，1個の予約の比較しかしていないためです。それでは，8個の予約の比較に拡張していきたいと思います。

8個の予約のように，たくさんの同じ形式のデータを扱うには配列が一番です。配列では，個々の要素に名前を付けるのではなく，同じ形式のデータをまとめたものに名前を付け，そのX番目の要素といった参照のしかたをします。

*emlC-17K*で配列を定義するには，次のようにしました。

```
nibble  予約週［8］      addr(0.0x10)；
```

前回のプログラムでは，「予約時刻」を次のように定義しました。

```
nibble  予約週           addr(0.0x10)；
nibble  予約曜日         addr(0.0x11)；
byte    予約時           addr(0.0x12)；
byte    予約分           addr(0.0x14)；
```

これを8個用意するのですから，

```
nibble  予約週［8］      addr(0.0x10)；
nibble  予約曜日［8］    addr(0.0x20)；
byte    予約時［8］      addr(0.0x30)；
byte    予約分［8］      addr(0.0x50)；
```

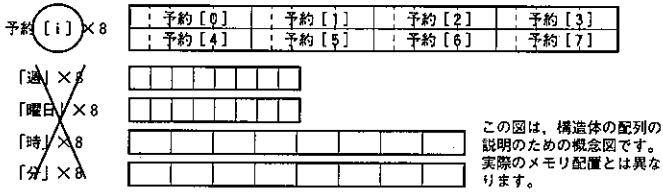
となります。でも今回はこの方法を使わずに，構造体を使ってこれを定義します。

*emlC-17K*で構造体の配列は次のように定義します。

```
struct　日時tag 予約［8］　addr(0.0x10)；
```

構造体の配列とは，配列の各要素が構造体であるモノを意味します。

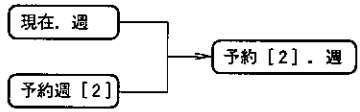
まず，「予約」という構造体を8個定義しています。ここで，必ず押さえておかなければいけないこととして，「予約」という構造体の配列が定義されているということです。「週」や「曜日」といった変数（メンバ）の配列が定義されているのではありません。



●構造体の配列を使った要素の参照

先ほど構造体のメンバの参照には，ピリオド（．）を使うと説明しました。また配列の要素の参照は，要素の位置を数字（または値を持った変数）で指定します。

上記2点より，構造体の配列のメンバの参照は簡単です。この2つの方法を組み合わせればいいのですから，次のようになります。



これで，「予約」という構造体の配列の3番目の要素の「週」というメンバを参照できます。

「予約」という構造体が（配列として）8個定義されていることを思い出してください。「予約」という構造体の「週」というメンバが8個定義されているわけではありません。

一般に，「予約」という構造体の配列のｉ番目の要素を参照するように，「予約検査」の関数の一部を修正すると，次のようになります。

```
void　予約検査(void) |
        時間一致 = 一致しない；
        if((予約 [i] . 週 == 現在. 週)&&¥
          (予約 [i] . 曜日 == 現在. 曜日)&&¥
          (予約 [i] . 時 == 現在. 時)&&¥
          (予約 [i] . 分 == 現在. 分))
                        時間一致 = 一致した；
}
```

でも、これでは「予約」という構造体の配列のどの要素と比較するかがまったく無視されています（何番目の要素かを指定する変数「i」を指定していません）。

●要素の比較（8回）

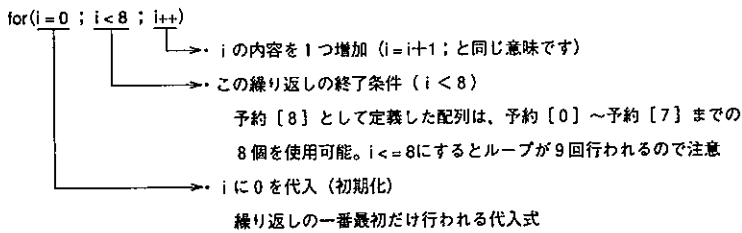
配列の任意の要素を（i番目の要素という形で）指定できたら、今度はそれを8回繰り返して比較すれば、すべての予約状況を検査できます。*emIC-17K*の形式で繰り返しの文を書くと次のようになります。

```
for(i = 0 ; i<8 ; i++) |
        /*繰り返す処理*/
}
```

ここで「繰り返す処理」というのは、先ほどのif文のことです。では、繰り返しの説明に移りましょう。

*emIC-17K*の繰り返しには3通りの方法があります。1つは、先ほどの「for」を使うような繰り返す回数が決まっている場合です。残りの2つは、繰り返す条件を先に検査してから「繰り返す処理」を行うwhileと、繰り返す条件を「繰り返す処理」のあとで検査するdo-whileです。この2つは、今回のサンプル・プログラムには登場しません。今回のサンプル・プログラムで登場するforですが、これは非常に簡単です。まずforの中に登場する「i」を説明します。

ここで、「i」は一般にループ・カウンタと呼ばれるものです。ループ・カウンタは、アセンブラなどでも頻繁に出てくるので理解しやすいでしょう。要するにループする回数を制御するための変数です。



forは，この3つの式を括弧の中にセミコロンで区切って書くのが基本です。式を省略することもありますが，このような使い方が最もよく使われます。

これで，比較の繰り返しの部分は全部書けたことになります。もう一度「予約検査」の関数の比較の部分を掲載しておくと

```
void　予約検査(void){
        時間一致 = 一致しない；
        for(i = 0 ； i < 8 ； i++){
                if((予約［i］．週 == 現在．週)&&¥
                  (予約［i］．曜日 == 現在．曜日)&&¥
                  (予約［i］．時 == 現在．時)&&¥
                  (予約［i］．分 == 現在．分))
                                時間一致 = 一致した；
        }
}
```

となります。これで「できた！」と思ってコンパイルを行うと，エラーが発生します。

```
Source file name : exmlc2. mlc
予約検査()：
exmlc2. mlc(28)：Error F2348 : 予約検査関数の i 変数は宣言されていません．
exmlc2. mlc(29)：Error F2348 : 予約検査関数の i 変数は宣言されていません．
```

これは変数「i」の定義がされていないためです。

●ループ・カウンタの定義

「i」というループ・カウンタ用の変数を定義しましょう。

「i」をどこで定義するかという問題は，その「i」がどこで必要とされるかによって異なります。もし，「i」がこのモジュールのほかの関数で使用する必要があるならば，「予約」や「現在」などと一緒にモジュールの先頭で定義するのもいいでしょう。しかし，この場合の「i」は，ほかの関数では必要とされず，この関数のforループの中だけで必要とされる変数なのです。したがって，関数の中で定義を行います。関数の中で変数の定義を行うことで，その変数は関数の中だけで使用できます。

文法的には次のように書きます。

```
void　予約検査(void)｜
        nibble　i addr(0.0x08)；

        時間一致＝一致しない；
        for(i＝0 ； i＜8 ； i++)｜
              if((予約［i］．週 == 現在．週)&&¥
                (予約［i］．曜日 == 現在．曜日)&&¥
                (予約［i］．時 == 現在．時)&&¥
                (予約［i］．分 == 現在．分))
                                時間一致＝一致した；
        ｝
｝
```

この変数「ｉ」の定義の下に１行の何も書かない行を設けたのは、それより上は変数定義、そこから下は関数の動作ということを明白にするためです。ちなみに「ｉ」というどこでも使いそうな名前にしたのは、ほかの関数からこの変数を使用できないし、ほかの関数で同じ名前の「ｉ」という変数を使用しても、きちんと異なるものであると認められるからです。つまりこの「ｉ」は、関数の中だけの「ｉ」になるのです。

●関数の修正

「予約検査」という関数では、新しいデータ構造に合わせて修正を行いました。でもこのデータ構造を使用するほかの関数、サブルーチンでは修正はしていません。とりあえずテスト・データ用のダミーの関数を変更しておかないといけません。

```
void setapo(void)｜
        nibble　i addr(0.0x08)；

        for(i＝0 ； i＜8 ； i++)｛
              予約［i］．週＝i；
              予約［i］．曜日＝i；
              予約［i］．時＝i；
              予約［i］．分＝i；
      ｝
｝
```

```
void gettime(void) {
        現在．週 = 1
        現在．曜日 = 2
        現在．時 = 3
        現在．分 = 4
}
```

データの構造を変えることは，それなりのリスクが生じます。通常，多くの人が関わっているプログラム開発プロジェクトで，ほかのモジュールからも参照する可能性のあるデータ構造を1プログラムの思い付きで簡単に変更することは許されません。

したがって，このようなモジュール間で共有するようなデータ構造を設計する人は，このマニュアルで説明している程度のことは完全に理解してからデータ構造を設計する必要があります。

それでは，ここでもう一度，変更したソース・プログラムの全文を見てみましょう。

```
#define 一致した          (1)
#define 一致しない        (0)

void    main(void);                     //プロトタイプ宣言
void    予約検査(void);                 //プロトタイプ宣言
void    gettime(void);                  //プロトタイプ宣言
void    setapo(void);                   //プロトタイプ宣言

        struct  日時tag {
                nibble  週;
                nibble  曜日;
                byte    時;
                byte    分;
        } 現在  addr(0.0x00);

        struct    日時tag     予約[8]   addr(1.0x00);
        boolean 時間一致                addr(0.0x07.0);

void 予約検査(void) {
        nibble  i     addr(0.0x08);

        時間一致 = 一致しない;
        for(i = 0; i < 8; i++) {
                if((予約[i].週 == 現在.週)&&¥
                  (予約[i].曜日 == 現在.曜日)&&¥
                  (予約[i].時 == 現在.時)&&¥
                  (予約[i].分 == 現在.分))
                      時間一致 = 一致した;
        }
}

void setapo(void) {
        nibble  i     addr(0.0x08);

        for(i = 0; i < 8; i++) {
              予約[i].週 = i;
              予約[i].曜日 = i;
              予約[i].時 = i;
              予約[i].分 = i;
        }
}

void gettime(void) {
        現在.週 = 1;
        現在.曜日 = 2;
        現在.時 = 3;
        現在.分 = 4;
}

void main(void) {
        gettime();
        setapo();
        予約検査();
}
```

「予約検査」関数は予約状況に関わらず8回の検査を行うようになっています。実際すべての予約状況の検査が必要な場合もあるでしょうが，今回は，予約が1つでもあったらそれ以降は検査する必要はないので，検査を打ち切ってしまってもかまいません。そうするためには，「時間一致＝一致した」を代入した場合にこの処理を打ち切ることが必要です。

ここで，処理を打ち切るには2つの方法があります。1つはforループを抜ける方法，もう1つは「予約検査」関数を終わる方法です。

処理を打ち切る前に関数の処理が終わったらどうなるでしょうか？　関数定義の中に記述されたすべての文を実行すると，関数を呼び出したところに戻り，関数呼び出しの次の文を実行するようになっています。しかし，関数定義の中では，関数の終わりを明示的に書くことが推奨されています。また，今回のサンプル・プログラムの関数は値を返しませんが，値を返す関数では返す値を指定しなければなりません。

それらのことを行うキー・ワードが「return」です。

●関数を抜ける

（1）forループを抜ける

forループを途中で中断する場合は，「break」というキー・ワードを使います。breakを書いた位置のfor（またはwhile/do-while）ブロックのループを中断できます。

```
void　予約検査(void){
        nibble　i addr(0.0x08);

        時間一致＝一致しない;
        for(i＝0 ; i＜8 ; i++){
              if((予約 [i] . 週 == 現在. 週)&&¥
                (予約 [i] . 曜日 == 現在. 曜日)&&¥
                (予約 [i] . 時 == 現在. 時)&&¥
                (予約 [i] . 分 == 現在. 分)){
                      時間一致＝一致した;
                      break;
                }
        }
}
```

（2）関数を終わる

「時間一致＝一致した」の次の文で処理を打ち切るのには，「return」を記述して関数を終わる方法もあります。

```
void　予約検査(void){
        nibble　i addr(0.0x08);

        時間一致＝一致しない;
        for(i＝0 ; i＜8 ; i++){
            if((予約［i］．週 == 現在．週)&&¥
              (予約［i］．曜日 == 現在．曜日)&&¥
              (予約［i］．時 == 現在．時)&&¥
              (予約［i］．分 == 現在．分)){
                時間一致＝一致した;
                return;
            }
        }
}
```

ただし，構造化プログラミングという観点から見ると，「プログラムのサブルーチン（この場合は関数）の入り口と出口は1つずつにするべきだ」ということで，好ましくありません。

●何をしたのかハッキリさせる

*emIC-17K*では，関数が処理する値を引数として渡すことはできません。外部変数として定義するようにします。ただしその代わり，関数の定義，宣言で関数が入出力する値を明示的に指定できます。

今回はINPUT()は使用しませんが，OUTPUT()は使用します。

今回のサンプル・プログラムで「予約検査」の関数が出力するのは，「時間一致」フラグです。したがってこれを“OUTPUT(boolean　時間一致)”と関数の引数を指定するところに書きます。

```
void　予約検査(OUTPUT(boolean　時間一致)){
        nibble　i addr(0.0x08);

        時間一致 = 一致しない;
        for(i = 0 ; i < 8 ; i++){
            if((予約 [i]．週 == 現在．週)&&¥
              (予約 [i]．曜日 == 現在．曜日)&&¥
              (予約 [i]．時 == 現在．時)&&¥
              (予約 [i]．分 == 現在．分)){
                時間一致 = 一致した;
                break;
            }
        }
}
```

今までのサンプル・プログラムで「void」と書いてあったのは，入出力引数は存在しないという意味でした。これでは適切でないので，「時間一致」を出力引数に指定します。ついでに，メイン関数での関数呼び出しも修正しておきましょう。

```
予約検査(OUTPUT(boolean　時間一致));
```

それではここでもう一度，変更したソース・プログラムの全文を見てみましょう。

```
#define 一致した          (1)
#define 一致しない        (0)

        struct  日時tag {
                nibble  週;
                nibble  曜日;
                byte    時;
                byte    分;
        } 現在  addr(0.0x00);

        struct    日時tag     予約[8]   addr(0.0x10);
        boolean  時間一致               addr(0.0x07.0);

void    main(void);                                          //プロトタイプ宣言
void    予約検査(OUTPUT(boolean 時間一致));                  //プロトタイプ宣言
void    gettime(void);                                       //プロトタイプ宣言
void    setapo(void);                                        //プロトタイプ宣言

void 予約検査(OUTPUT(boolean 時間一致)) {
        nibble  i     addr(0.0x08);

        時間一致 = 一致しない;
        for(i = 0; i < 8; i++) {
                if((予約[i].週 == 現在.週)&&¥
                  (予約[i].曜日 == 現在.曜日)&&¥
                  (予約[i].時 == 現在.時)&&¥
                  (予約[i].分 == 現在.分)) {
                        時間一致 = 一致した;
                         break;
                }
        }
}

void setapo(void) {
        nibble  i     addr(0.0x08);

        for(i = 0; i < 8; i++) {
              予約[i].週 = i;
              予約[i].曜日 = i;
              予約[i].時 = i;
              予約[i].分 = i;
        }
}

void gettime(void) {
        現在.週 = 1;
        現在.曜日 = 2;
        現在.時 = 3;
        現在.分 = 4;
}

void main(void) {
        gettime();
        setapo();
        予約検査(OUTPUT(boolean 時間一致));
}
```

〔メ　モ〕

第３章　検査および機能

この章では，プログラムの検査およびその他の機能について説明します。

３．１　検　　査

３．１．１　通過履歴

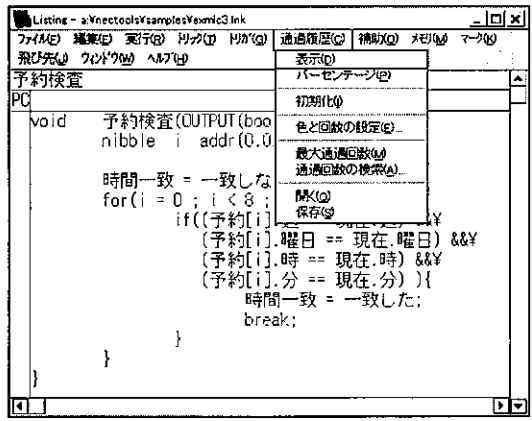
プログラムの通過履歴とは，プログラム中の各文を何回実行したかの記録です。たとえば，if文の４番目の比較「(予約 [i] .分 == 現在.分))」という文を何回実行したかを記録しています。通過履歴には２つの使い方があり，１つはプログラム開発工程の管理です。

●プログラム開発工程の管理

*SIMPLEHOST*では，プログラムの通過履歴を回数で表示するだけでなく，色でも表示し，プログラムでどこの部分を通過していないかを一目で見つけることができます。また通過回数の多いルーチンや文を見抜き，どの文が多くのステップ数を消費しているかを理解し，その部分をアセンブラで書いたりして，プログラム・サイズのダイエットに役立てることができます。

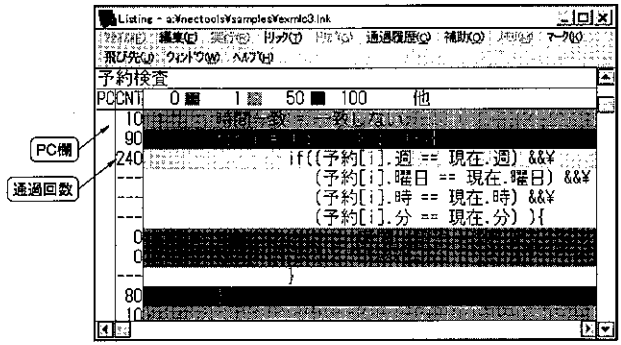
この機能を使うのは簡単で、Listing画面の「通過履歴」メニューを選びます。

図３－１　「通過履歴」メニュー



リスト画面では，行ごとのプログラムの通過回数を色分けして表示し，PC欄とソース・イメージの間に通過回数を数値で表示します。

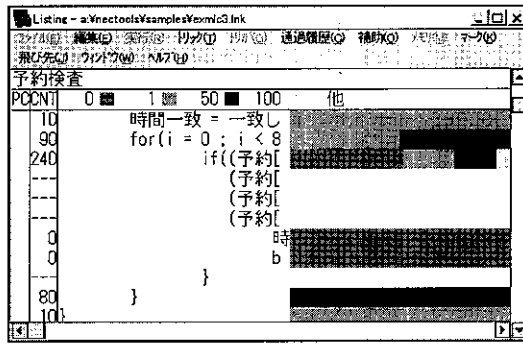
図３－２　通過履歴表示画面



*emIC-17K*の１行が複数のアセンブラの命令で構成されている場合は，１行の中でも複数の実行回数を持つ場合があります。そんなときに最大値を表示するか，最小値を表示するかを選択できます。これには，「通過履歴」－「最大通過回数」メニューを使用します。

またモジュール画面では，プログラムの通過回数を現在表示しているレベルごとにグラフ化します。

図３－３　グラフ表示画面



●ディバグ・ツール

通過履歴の2つめの使い方は，1つめの使い方に比べるといささか技巧的ですが，特定の回数を繰り返すルーチンのディバグに使えます。サンプル・プログラムのfor文は，最大8回ループを繰り返すはずでした。8回ループを繰り返した場合，「現在時刻」と一致する「予約時刻」はないことになります。ループが8回，回らなかった場合，どこかに「現在時刻」と一致する「予約時刻」がありループを中断した可能性があります（もし，「時間一致」フラグがセットされなくてループが8回未満で中断していたらそれはバグです）。したがって，この「ループした回数」というのが重要になってきます。

これは，ループ・カウンタ「i」の値を調べることでもチェックできますが，通過履歴を使えばそのループが何回，回ったかも簡単に知ることができます。

このように通過履歴は使い方によっては有効なディバグ・ツールともなり得ます。

［参考］　プログラムのディバグ時間は限られています。捜して見つかるか見つからないか分からないものをいつまでも捜しているわけにはいきませんが，存在するバグは取り除く必要があります。

何をもってディバグの終了と判断するのか？

その1つの判断基準に「通過履歴」，「書き込み履歴」というものがあります。つまり

すべての文を最低1度は実行する

ことにより，そのプログラムはディバグが済んだといえます。プログラムを実行したかしないか，何回実行したかの情報を「通過履歴」といいます。同様に使用しているメモリに対する書き込み動作をチェックすることで，ディバグの進歩状況を把握できます。

3.1.2　通過履歴を調べる

今回のサンプル・プログラムでは，「現在時刻」の取得と「予約時刻」のセットは，ほかのモジュールの分担ということになっていました（サンプル・プログラムで使用したgettime()とsetapo()は，ダミーの関数でした）。

一番最初の（前半のプログラムで使用した）テスト・データは，「予約時刻」と「現在時刻」がまったく異なる場合のテスト・データです。今「現在時刻」には（gettime関数で）「1」，「2」，「3」，「4」が代入され，「予約時刻」には配列の添え字（i）と同じ数値が代入されています。この場合も「予約時刻」と「現在時刻」は異なることになります。この条件下でプログラムを実行すると，「時間一致＝一致した」は実行されないはずです。やってみてください。

●時間が一致する場合

では，「現在時刻」を「1」，「1」，「1」，「1」として，プログラムが実行することを考えてみましょう。まず「現在時刻」の値を変更する方法ですが，これには3通りあります。

（1）ソース・プログラムを変更してもう一度コンパイル（Listingの「ファイル」－「ソース修正」メニューを選択）

これはいささか強引なやり方です。この方法では，テスト・データを変更するたびに，コンパイルをしなおさなくてはなりません。

（2）Memoryを使用

いちいちコンパイルをしなくても変数の値は変更することができます。でもMemoryを起動するというほんのわずかな手間がかかることも事実です。

（3）変数域を使用

変数域に名前を入力しその値を参照するだけでしたが，その値のところにマウス・カーソルを動かしクリックし，数値を入力することで値を変更することができたのです。

では，（3）の方法を使って「現在時刻」の値を変数域に入力して値を変更し，プログラムを実行してみてください。「時間一致＝一致した」を実行しましたか。サンプル・プログラムの変更を間違えていなければ，「時間一致＝一致した」を実行するはずです。

●あと検査すべきケースは？

これで，「予約時刻」と「現在時刻」が一致する場合と一致しない場合の両方の場合について，プログラムが予想どおり動いたことになります。では，あと検査すべきケースというのはあるでしょうか？

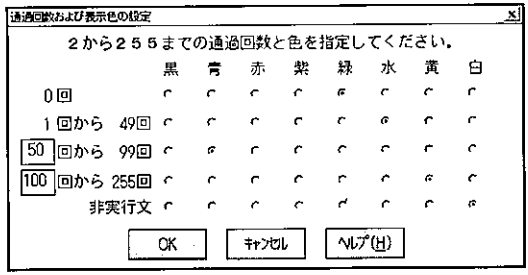
今回のテストで「予約時刻」と「現在時刻」の値が一致したのは，予約「1」で一致していることになります。これを予約「0」から予約「7」までのすべてのケースにおいて検査するのもいいでしょう（少なくとも「バグの可能性がわずかながら減った」と自信を持つことができます）。

また，上記の各ケースにおいて一致する場合は，「週」から「分」まですべての比較を行いますが，一致しない場合に，どこまで実行しているのか調べるのもテスト項目の1つです。

●通過履歴と書き込み履歴の設定

通過履歴の設定は，Listingの「通過履歴」－「色と回数の設定」メニューによりダイアログ・ボックスが表示されます。

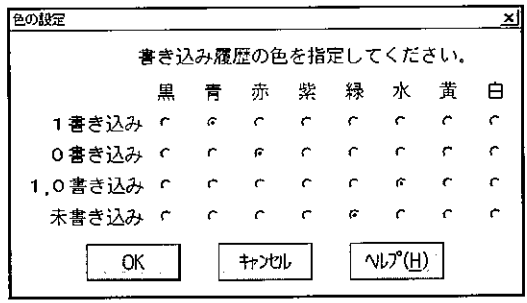
図３－４　通過回数と表示色の設定



このダイアログ・ボックスでは，通過回数の分割点と色を設定します。通過回数の分割点とは，１から255回に通過回数を３分割するための２点を指します。

★　書き込み履歴の色の設定は，Memoryの「書き込み履歴」－「色の設定」メニューを選びます。これを選ぶと［色の設定］ダイアログ・ボックスが表示されます。

図３－５　色の設定



このダイアログ・ボックスでは，“１書き込み”，“０書き込み”，“1,0書き込み”，“未書き込み”の４種類について色を設定します。

３.１.３　履歴の保存

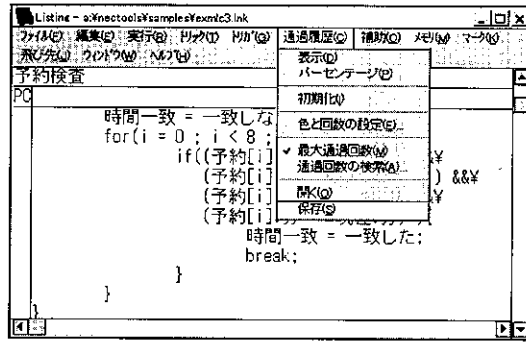
17Kシリーズで大きなプログラムや，複雑に分岐するプログラムを作成した場合，通過履歴の確認をはじめさまざまなディバグ作業が一日では済まない場合があります。*SIMPLEHOST*では，検査が数日に渡った場合を想定して，さまざまなデータをロード，セーブできます。ここでは，通過履歴をはじめ各種データの保存方法を簡単に説明します。

●通過履歴と書き込み履歴

Listingで扱う通過履歴とMemoryで扱う書き込み履歴は１つのファイルに保存されます（拡張子はCOV）。保存の手順は、Listingで、「通過履歴」－「保存」メニューを選択します。

逆に保存した通過履歴を使ってディバグを再開するときは、「通過履歴」－「開く」メニューを選択します。

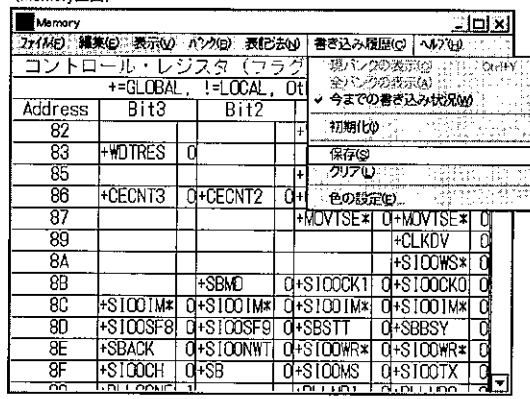
図３－６　「通過履歴」－「保存」メニュー



また、書き込み履歴の保存はMemoryで、「書き込み履歴」－「保存」メニューを選択します。「書き込み履歴」－「今までの書き込み状況」メニューを選択すると、前回ファイルに保存した書き込み履歴と現在のIE-17Kの書き込み履歴を合わせて表示します。

図３－７　「今までの書き込み状況」メニュー

〈Memory画面〉

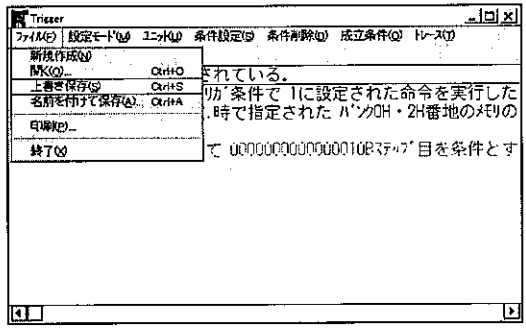


そのほかのデータとしては，表示変数ファイル，トリガ情報ファイルやトレース情報ファイル，PPG情報ファイルがあります。もしも，情報を保存しなければ*SIMPLEHOST*を起動するたびに同じ設定を行うことになりますので，トリガやトレースの設定を行ったら，設定した条件などを保存したほうが作業効率が良いでしょう。

●トリガ情報

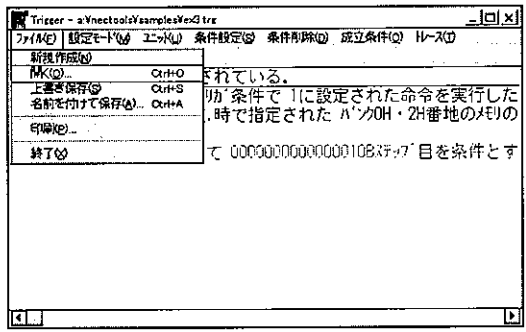
トリガ情報の保存はTriggerで，「ファイル」－「名前を付けて保存」または「上書き保存」メニューを選択します。

図3－8　「ファイル」－「上書き保存」メニュー



トリガ情報ファイルは，「名前を付けて保存」することができ，必要に応じて呼び出すことができます。Triggerで，「ファイル」－「開く」メニューを選択します。

図3－9　「ファイル」－「開く」メニュー

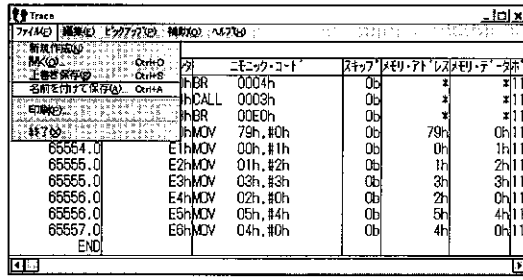


以前，ファイルに保存したトリガ設定を読み込みます。

●トレース情報

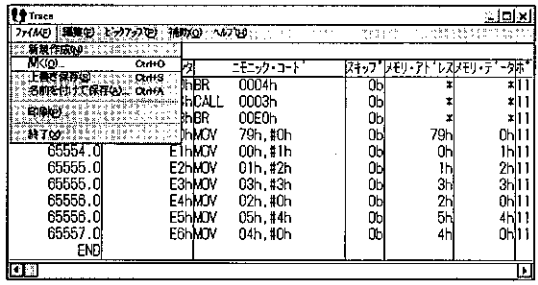
トレース情報の保存はTraceで，「ファイル」－「名前を付けて保存」または「上書き保存」メニューを選択します。

図3－10　「ファイル」－「名前を付けて保存」メニュー



トレース情報ファイルは，「名前を付けて保存」することができ，必要に応じて呼び出すことができます。Traceで，「ファイル」－「開く」メニューを選択します。

図3－11　「ファイル」－「開く」メニュー



以前，ファイルに保存したトレース情報を読み込みます。

ただし1つしてはいけないことは，Windowsや*SIMPLEHOST*を起動したままホスト・マシンの電源を切ったり，ホスト・マシンのリセット・ボタンを押すことです。この場合は，使用中のデータが失われる可能性があるどころか機械そのもの（特にハード・ディスク）が壊れる可能性さえあります。

３．２　その他の機能

３．２．１　Trace

このステップではTraceに関して簡単に説明します。

Traceは，Listingのようにプログラムを実行するとか，Triggerのようにブレーク条件を設定するとかいったことはしません。しかしトレース機能はプログラムのディバグに絶対必要な機能の１つです。

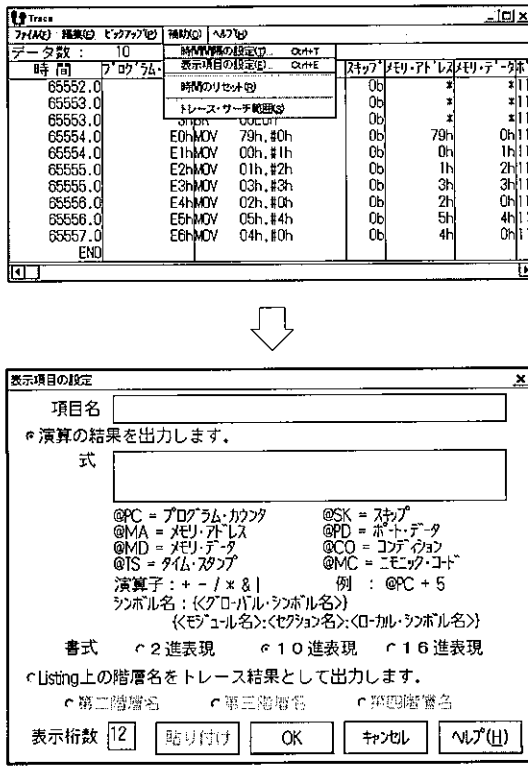
*SIMPLEHOST*のTraceは，データ・ベース的な機能を兼ね備えた高機能のトレース機能を持っています。

Traceは，Listingが実行したプログラムの結果のレポートを作ります。これは，ある時間のプログラム・カウンタやスキップの有無，メモリ・アドレスやメモリ・データ，プローブ，コンディションなどの情報を記録・表示します。また必要があれば，これらの項目を組み合わせて新しい項目を作ることもできます。

★　●トレース項目の追加，削除，変更

トレース項目は，「補助」－「表示項目の設定」メニューにより追加することができます。

★　図３－12　「表示項目の設定」メニュー



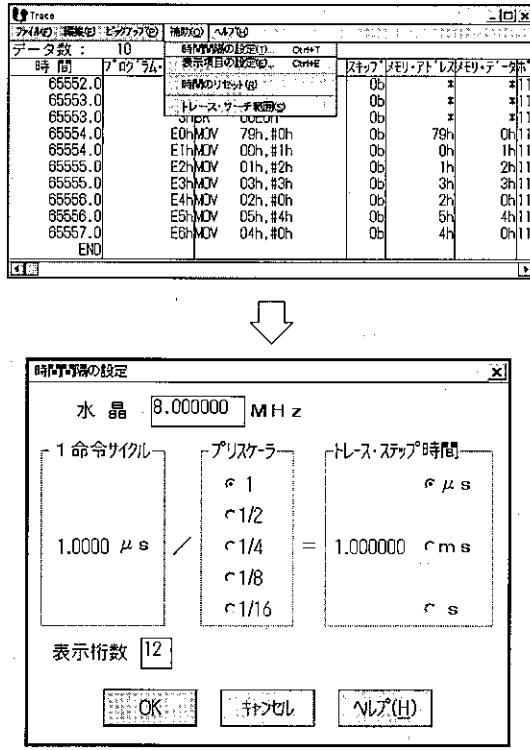
★　［表示項目の設定］ダイアログ・ボックスで、トレース項目の追加、削除、変更を行います。「追加」はいいとしてなぜ「変更」なのかというと、Trace画面にすでに存在する項目名をマウスでダブルクリックすると、その項目の内容を変更することができるからです。

●時間間隔の設定

Traceではプログラムの実行経過を時間軸とした表の形で表示します。この時間は，チップの水晶振動子の周波数を基準に算出されています。この時間は最初から正しい周波数が設定されているとは限りませんので，時間間隔の設定で設定しておきましょう。

「補助」－「時間間隔の設定」メニューを選択します。

図3－13　「時間間隔の設定」メニュー



この“水晶”という項目に開発対象デバイスの周波数をMHz単位で入力してください。この項目の周波数と実際の水晶の周波数が異なるとトレース・ステップ時間（時間欄に表示される時間の増加する場合）が正しく計測されません。“水晶”の項目に入力された周波数から1命令サイクルが算出され，その命令サイクルがトレース・ステップ時間となるからです。

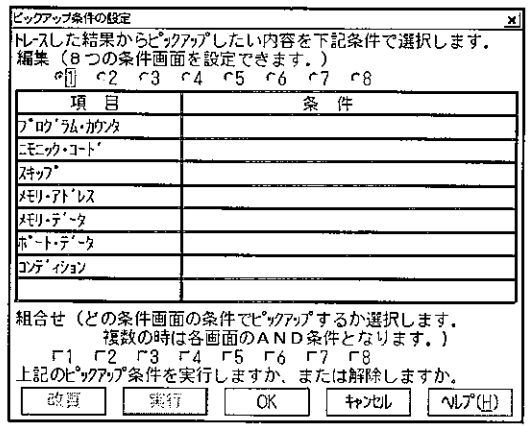
ついでに解説しておくと，これに「プリスケーラ」と「単位時間」を組み合わせることにより，トレース・ステップ時間を変えることができます。たとえば，プリスケーラが「1」，トレース・ステップ時間の単位が，［μs］の場合には，ほぼ1命令に1回の割合でトレース結果を得ることができます。通常はこのままでもかまいませんが，大きなプログラムの場合はプリスケーラを「1/16」に指定すると，ほぼ16命令に1回の割合でトレース結果を得ることができます。

●しぼり込む

大きなプログラムをディバグしていると，プログラムの実行結果も膨大な量になる可能性があります。IE-17Kには，32Kステップのトレース結果を格納するためのトレース・メモリがあるのですから，この中からプログラムのディバグに必要な部分を手作業で捜すのは困難です。

このような作業を行うために，Traceにはしぼり込みの機能があります。「ピックアップ」－「条件設定」メニューを選択してください。

図3－14　ピックアップの条件設定画面

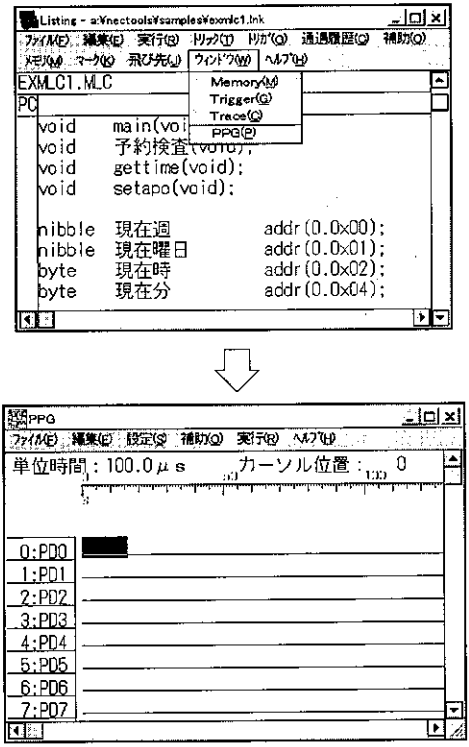


この機能は，膨大な量のトレース結果から必要なものだけをしぼり込むために使用します。

3.2.2　PPG

PPGとは，Programmable Pattern Generatorの省略名です。PPGの起動もほかのウインドウと同じで，Listingの「ウィンドウ」―「PPG」メニューで起動します。

図3－15　PPG画面

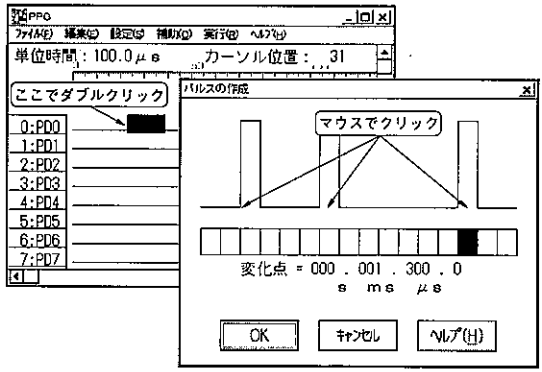


これでPPGが起動しPPGウインドウが表示されます。この画面をパルス画面といい，パルス・パターンが表示されています。

●パルス・データの作成

この画面でパルス・データを作成し出力することができます。まずはパルス・データの作成です。パルス画面のパルス・パターンをダブルクリックします。

図3－16　パルス・パターン画面

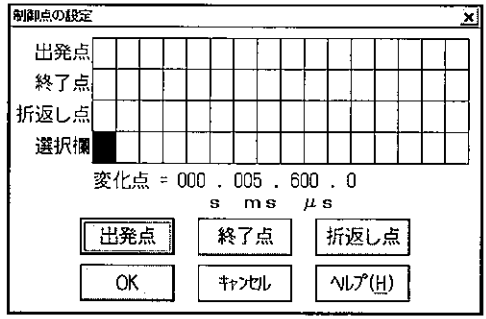


ダブルクリックした部分のパルス・パターンを拡張表示したものが、［パルスの作成］ダイアログ・ボックスの上部に表示されます（まだ何も設定していないのでただの直線です）。

あとはこのダイアログボックスで拡大表示されたパターンをマウスでクリックして波形を作り、「OK」ボタンを選択してその部分のパターン作成を終了し、また別の部分のパルス・パターンをクリックして・・・という風に作成していきます。

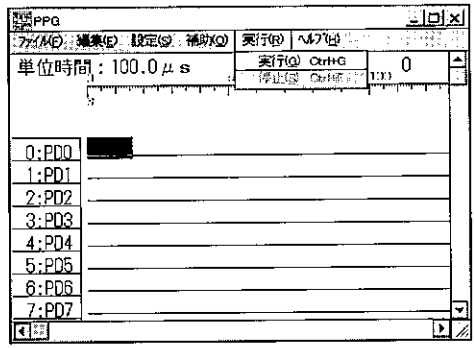
これに制御点を設定するには、パルス画面上部の目盛りをダブルクリックします。

図3－17　制御点の設定



すると［制御点の設定］ダイアログ・ボックスが表示されます。このマス目の中をマウスでクリックすれば制御点の設定ができます（黒く反転表示します）。制御点の設定の終わりも「OK」ボタンです。制御点の設定を行ったパルス画面上部の目盛りには、出発点が「s」、終了点が「e」、折り返し点が「r」と表示されます。パルス・データができたら、あとはPPGの実行だけです。もう画面を見ればすぐに分かりますよね。「実行」－「実行」メニューを選択してください。

図3－18　PPGの実行



3.2.3　Cコンパイラ機能

*emIC-17K*には、入門編ではとても説明しきれないほどのたくさんの便利な機能があります。
ここでは、それらの中でも特徴的ないくつかの機能について説明します。

●最適化

*emIC-17K*はコンパイラです。コンパイラはソース・プログラムに記述された内容を実行プログラムに「翻訳」しますが、条件によってはこの翻訳の方法を変えることができます。その条件とは、「大きさよりもスピードを優先させる」とか、この逆で「スピードよりも小さくすることに重点を置く」といったことです（別にこれは「翻訳」のスピードをどうこうするわけではありません。「翻訳」した結果の実行プログラムの大きさやスピードに対する条件です）。

このような、ソース・プログラムを実行プログラムに「翻訳」するときに行われる（よりよいプログラムを目指しての）変更を「最適化」といいます。*emIC-17K*のプログラムで最適化を指定するのは簡単です。プログラムの一番最初で

```
# pragma optimize STEP
```

または

```
# pragma optimize SPEED
```

を指定すればいいのです。あとの面倒なことはすべてコンパイラが行ってくれます（通常は、プログラム開発の初期段階ではコンパイラによる最適化は行いません。ある程度ディバグが終了してから最適化を指定するようにします）。

●gettime()/setapo()が実装された場合

このモジュールでは現在時刻の取得と予約の設定は行いません。だれかがほかのモジュールで定義していることになっています。ですから，gettime()/setapo()がほかのモジュールで実装された場合は，リンク処理を行う前にこのモジュールから2つの関数定義（代入をしている部分）を削除する必要があります。

しかし，このモジュールのメイン関数では，予約検査()の前で，この2つの関数を呼び出しています。したがって，この2つの関数がほかのモジュールで定義されていることを宣言しなければなりません。そこで，サンプル・プログラムのメイン関数を流用する場合，

```
extern void gettime();
extern void setapo();
```

この2行をexample.mlcに追加することで，ほかのモジュール（ファイル）で実装された関数を使用することができるようになります。externは，「ほかのモジュールで定義されています」という意味です。

●プロトタイプ宣言

プログラムで関数の定義を行う前にその関数を呼び出すようなプログラムを書くと，定義されていない関数を使用するという状態が発生します。関数や変数など識別子は実際に使用する前に定義されていなければなりません。

ある名前の関数を使用すること，その関数の戻り値の型や引数の型をあらかじめ宣言しておくことをプロトタイプ宣言といいます。

今回のサンプル・プログラムの「予約検査()」という関数でプロトタイプ宣言を行うと，次のようになります。

```
void 予約検査(OUTPUT(boolean 時間一致));
```

今回のサンプル・プログラムでは関数はすべて使用する前に定義されています（メイン関数で「予約検査」を呼び出す前に定義している）ので，このプロトタイプ宣言がなければエラーが発生するというわけではありませんが，プログラムを作るときの常とう手段として記述することをお勧めします。

●自分で作成した関数を公開する

今度は，ほかのモジュールから今回作成した予約検査()という関数が使用されるケースを考えてみましょう。使用するモジュールで

```
extern void 予約検査(OUTPUT(boolean 時間一致));
```

とすれば，予約検査という関数を使用できます。externは先ほどもでてきましたが，別のモジュール（今回はexample.mlc）で定義されているという意味です。これで2つのモジュールで合計3つの関数が使用されることになります。

通常このようなモジュール間で共通の宣言などは，ヘッダ・ファイルにひとまとめにします。

```
extern void gettime(void)；
extern void setapo(void)；
extern void 予約検査(OUTPUT(boolean 時間一致))；
```

これだけでもいいのですが，記号定数やほかの宣言なども一緒にヘッダ・ファイルにまとめることも考えてみましょう。

[例1]

```
#define 一致した　　(1)
#define 一致しない　(0)

        struct 日時tag {
                nibble 週；
                nibble 曜日；
                byte 時；
                byte 分；
        }；
```

[例2]

```
#include <example.h>
        struct 日時tag 現在          addr(0.0x00)；
        struct 日時tag 予約[8]  addr(0.0x10)；
```

●「時間一致」はどうするか？

今回作成したモジュールで，「時間一致」というフラグが残っていますが，このフラグはどうすればよいのでしょうか？

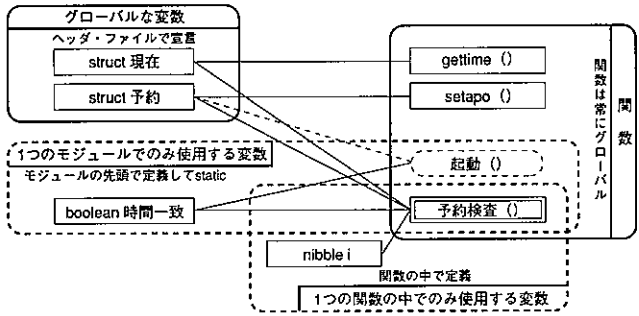
今回作成した「予約検査」のあとでどういう処理を行うかプログラムの仕様の中に明示されていません。この「予約検査()」のあとで「起動()」という関数を実行すると仮定した場合，「起動()」では「時間一致」というフラグを参照する必要があるはずです。

この「起動()」という関数をどのモジュールに配置するのが最も適切かといえば，同じフラグを使用する「予約検査()」という関数と一緒のモジュールに配置するのが最適といえます。なぜならば，1つのモジュールの中で使用する変数（この場合はフラグ）は，それをstatic宣言することで，ほかのモジュールから見えなくすることができるからです。

```
static boolean  時間一致 addr(0.0x07.0) ;
```

とすることで，ほかのモジュールからのアクセス（参照や変更）を防ぐことができます。つまり，このモジュールで定義される(であろう)2つの関数「予約検査()」と「起動()」は，ほかの関数と比べて，関数の結びつきが強い（このモジュールだけで必要とされる変数が存在する）ということができます。

逆にいえば，ほかのモジュールで設計・定義されている(であろう)2つの関数setapo()とgettime()は，「予約検査()」と「起動()」の結びつきに比べて，関数の結びつきが弱い(すべてのモジュール(関数)から参照される)ということもいえます。



〔メ 干〕

付録A　索　　引

〔メ　モ〕

付録B　改版履歴

これまでの改版履歴を次に示します。なお、適用箇所は各版での章を示します。

版　数	前版からの主な改版内容	適用箇所
第2版	バージョン　V2.0→V2.01に変更	全　般
	Windows3.1→Windows3.1およびWindows95対応に変更	
	Windows3.1→Windows95上の画面に変更	
	*SIMPLEHOST*のEditorを削除	
	1.2　インストールを変更 プログラム構成およびバックアップの項を削除	第1章　概　要
	2.4.2　プログラムの実行を変更 「実行」メニューに「ブレーク」を追加	第2章　プログラム設計
	2.4.4　実行チェックを変更 「実行」メニューに「中止」を追加	
	3.2.1　Traceを変更 「表示項目の追加／変更」メニュー名を「表示項目の設定」に変更（削除機能追加）	第3章　検査および機能

お問い合わせは，最寄りのNECへ

【営業関係お問い合わせ先】

半導体第一販売事業部 半導体第二販売事業部 半導体第三販売事業部	〒108-01 東京都港区芝五丁目7番1号（NEC本社ビル）	東京 (03)3454-1111	(大代表)
中部支社 半導体第一販売部 半導体第二販売部	〒460 名古屋市中区錦一丁目17番1号（NEC中部ビル）	名古屋 (052)222-2170 名古屋 (052)222-2190	
関西支社 半導体第一販売部 半導体第二販売部 半導体第三販売部	〒540 大阪市中央区城見一丁目4番24号（NEC関西ビル）	大阪 (06) 945-3178 大阪 (06) 945-3200 大阪 (06) 945-3208	

北海道支社	札幌 (011)231-0161	太田支店	太田 (0276)46-4011	富山支店	富山 (0764)31-8461
東北支社	仙台 (022)267-8740	宇都宮支店	宇都宮 (028)621-2281	三重支店	津 (0592)25-7341
岩手支店	盛岡 (019)651-4344	小山支店	小山 (0285)24-5011	京都支社	京都 (075)344-7824
山形支店	山形 (0236)23-5511	長野支社	松本 (0263)35-1662	神戸支社	神戸 (078)333-3854
郡山支店	郡山 (0249)23-5511	甲府支店	甲府 (0552)24-4141	中国支社	広島 (082)242-5504
いわき支店	いわき (0246)21-5511	埼玉支社	大宮 (048)641-1411	鳥取支店	鳥取 (0857)27-5311
長岡支店	長岡 (0258)36-2155	立川支社	立川 (0425)26-5981	岡山支店	岡山 (086)225-4455
土浦支店	土浦 (0298)23-6161	千葉支社	千葉 (043)238-8116	四国支社	高松 (0878)36-1200
水戸支店	水戸 (029)226-1717	静岡支社	静岡 (054)255-2211	新居浜支店	新居浜 (0897)32-5001
神奈川支社	横浜 (045)324-5524	北陸支社	金沢 (0762)23-1621	松山支店	松山 (089)945-4149
群馬支店	高崎 (0273)26-1255	福井支店	福井 (0776)22-1866	九州支社	福岡 (092)271-7700

【本資料に関する技術お問い合わせ先】

半導体ソリューション技術本部 マイクロコンピュータ技術部	〒210 川崎市幸区堀越三丁目484番地	川崎	(044)548-7950
半導体販売技術本部 東日本販売技術部	〒108-01 東京都港区芝五丁目7番1号（NEC本社ビル）	東京	(03)3798-9619
半導体販売技術本部 中部販売技術部	〒460 名古屋市中区錦一丁目17番1号（NEC中部ビル）	名古屋	(052)222-2125
半導体販売技術本部 西日本販売技術部	〒540 大阪市中央区城見一丁目4番24号（NEC関西ビル）	大阪	(06) 945-3383

半導体
インフォメーションセンター
FAX(044)548-7900
(FAXにてお願い致します)

アンケート記入のお願い

お手数ですが，このドキュメントに対するご意見をお寄せください。今後のドキュメント作成の参考にさせていただきます。

［ドキュメント名］ *SIMPLEHOST* ユーザーズ・マニュアル *emIC-17K*/RA17K対応版 入門編
（U10445JJ2V0UM00（第2版））

［お名前など］（さしつかえのない範囲で）

御社名（学校名，その他）（　　）
ご住所（　　）
お電話番号（　　）
お仕事の内容（　　）
お名前（　　）

1．ご評価（各欄に○をご記入ください）

項　　目	大変良い	良　い	普　通	悪　い	大変悪い
全体の構成					
説明内容					
用語解説					
調べやすさ					
デザイン，字の大きさなど					
その他（　　）					
（　　）					

2．わかりやすい所（第　章，第　章，第　章，第　章，その他　　）
理由［　　］

3．わかりにくい所（第　章，第　章，第　章，第　章，その他　　）
理由［　　］

4．ご意見，ご要望

5．このドキュメントをお届けしたのは
NEC販売員，特約店販売員，NEC半導体ソリューション技術本部員，
その他（　　）

ご協力ありがとうございました。
下記あてにFAXで送信いただくか，最寄りの販売員にコピーをお渡しください。

NEC半導体インフォメーションセンター
FAX：（044）548-7900

キリトリ