HT-IDE3000 User′s Guide

September 2005

Contents

P a r t I

Integrated Development Environment

Chapter 1

Overview and Installation

1

To ease the process of application development, the importance and availability of supporting tools for microcontrollers cannot be underestimated. To support its range of MCUs, Holtek is fully committed to the development and release of easy to use and fully functional tools for its full range of devices. The overall development environment is known as the HT-IDE, while the operating software is known as the HT-IDE3000. The software provides an extremely user friendly Windows based approach for program editing and debugging while the HT-ICE emulator hardware provides full real time emulation with multi functional trace, stepping and breakpoint functions. With a complete set of interface cards for its full device range and regular software Service Pack updates, the HT-IDE development environment ensures that designers have the best tools to maximize efficiency in the design and release of their microcontroller applications.

HT-IDE Development Environment

The Holtek Integrated Development Environment, otherwise known as the HT-IDE, is a high performance integrated development environment designed around Holtek′s series of 8-bit MCU devices. Incorporated within the system is the hardware and software tools necessary for rapid and easy development of applications based on the Holtek range of 8-bit MCUs. The key component within the HT-IDE system is the HT-ICE In-Circuit Emulator, capable of emulating the Holtek 8-bit MCU in real time, in addition to providing powerful debugging and trace features. The latest version of the HT-ICE In-Circuit Emulator also incorporates a complete OTP writer which provides the user with all the tools required to design, debug and program their OTP devices.

As for the software, the HT-IDE3000 provides a friendly workbench to ease the process of application program development, by integrating all of the software tools, such as editor, Cross Assembler, Cross Linker, library and symbolic debugger into a user friendly Windows based environment. In addition, the HT-IDE3000 provides a software simulator which is capable of simulating the behavior of Holtek′s 8-bit MCU range without connection to the HT-ICE. All fundamental functions of the HT-ICE hardware are valid for the simulator.

More detailed information on the HT-IDE3000 development system is contained within the HT-IDE3000 User′s Guide. Installed in conjunction with the HT-IDE3000 and to ensure that the development system contains information on new microcontrollers and the latest software updates, Holtek provides regular HT-IDE3000 Service Packs. These Service Packs, which can be downloaded from the Holtek website, do not replace the HT-IDE3000 but are installed after the HT-IDE3000 system software has been installed.

Some of the special features provided by the HT-IDE3000 include:

→ **Emulation**
- Real-time program instruction emulation

→ **Hardware**
- Easy installation and usage
- Either internal or external oscillator
- Breakpoint mechanism
- Trace functions and trigger qualification supported by trace emulation chip
- Printer port for connecting the HT-ICE to a host computer
- I/O interface card for connecting the user′s application board to the HT-ICE
- OTP writer hardware integrated within the HT-ICE

→ **Software**
- Windows based software utilities
- Source program level debugger (symbolic debugger)
- Workbench for multiple source program files (more than one source program file in one application project)
- All tools are included for the development, debug, evaluation and generation of the final application program code (mask ROM file and OTP file)
- Library for the setting up of common procedures which can be linked at a later date to other projects.
- Simulator can simulate and debug programs without connection to the HT-ICE hardware
- Virtual Peripheral Manager (VPM) simulates the behavior of the peripheral devices.
- LCD simulator simulates the behavior of the LCD panel.

Holtek In-Circuit Emulator – HT-ICE

Developed alongside the Holtek 8-bit microcontroller device range, the Holtek ICE is a fully functional in-circuit emulator for Holtek′s 8-bit microcontroller devices. Incorporated within the system are a comprehensive set of hardware and software tools for rapid and easy development of user applications. Central to the system is the in-circuit hardware emulator, capable of emulating all of Holtek′s 8-bit devices in real-time, while also providing a range of powerful debugging and trace facilities. Regarding software functions, the system incorporates a user-friendly Windows based workbench which integrates together functions such as program editor, Cross Assembler, Cross Linker and library manager. In addition, the system is capable of running in software simulation mode without connection to the HT-ICE hardware.

HT-ICE Interface Card

The interface cards supplied with the HT-ICE can be used for most applications, however, it is possible for the user to omit the supplied interface card and design their own interface card. By including the necessary interface circuitry on their own interface card, the user has a means of directly connecting their target boards to the CN1 and CN2 connectors of the HT-ICE.

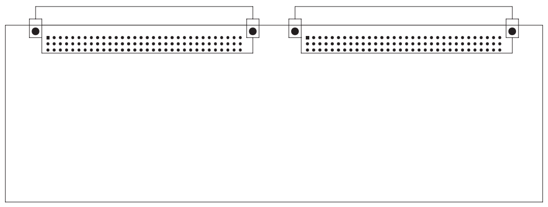


Fig 1-1

OTP Programmer

Holtek's OTP devices are fully supported by a range of programmers. For engineering level OTP device programming, Holtek supplies its stand alone programming tool which provides a quick and efficient means for low volume OTP programming. The HT-ICE In-Circuit Emulators has integrated a writer as part of the hardware package, facilitating complete design, debug and OTP device programming all within the HT-ICE. More programmers from other suppliers are available which provide more efficient and higher volume production capability. Refer to our website for further suppliers information.

OTP Adapter Card

The Holtek OTP programmers are supplied with a standard Textool chip socket. The OTP Adapter Card is used to connect the Holtek OTP programmers to the various sizes of available OTP chip packages that are unable to use this supplied socket.

System Configuration

The HT-IDE system configuration is shown below, in which the host computer is a Pentium compatible machine with Windows 95/98/NT/2000/XP or later. Note that if Windows NT/2000/XP or later systems are used, then the HT-IDE3000 software must be installed in the Supervisor Privilege mode.

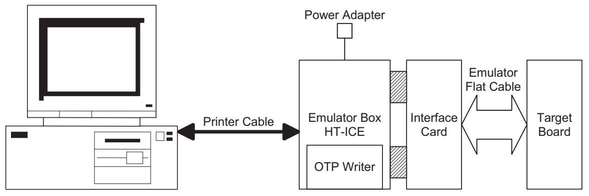
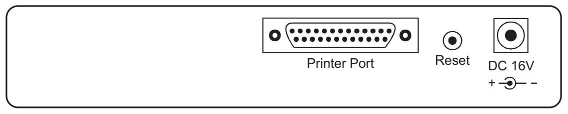


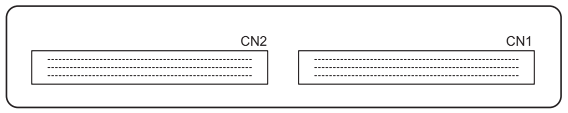
Fig 1-2

The HT-IDE system contains the following hardware components:

- The HT-ICE box contains the emulator box with 1 printer port connector for connecting to the host machine, I/O signal connector and one power-on LED
- I/O interface card for connecting the target board to the HT-ICE box
- Power Adapter, output 16V
- 25-pin D-type printer cable
- Integrated OTP writer



HT-ICE Rear View



HT-ICE Front View

Fig 1-3

Installation

System Requirement

The hardware and software requirements for installing HT-IDE3000 system are as follows:

- PC/AT compatible machine with Pentium or higher CPU
- SVGA color monitor
- At least 32M RAM for best performance
- CD ROM drive (for CD installation)
- At least 20M free disk space
- Parallel port to connect PC and HT-ICE
- Windows 95/98/NT/2000/XP

Windows 95/98/NT/2000/XP are trademarks of Microsoft Corporation.

Hardware Installation

- Step 1
  Plug the power adapter into the power connector of the HT-ICE
- Step 2
  Connect the target board to the HT-ICE by using the I/O interface card or flat cable
- Step 3
  Connect the HT-ICE to the host machine using the printer cable

The LED on the HT-ICE should now be lit, if not, there is an error and your dealer should be contacted.

Caution Exercise care when using the power adapter. Do not use a power adapter whose output voltage is not 16V, otherwise the HT-ICE may be damaged. It is strongly recommended that only the power adapter supplied by Holtek be used. First plug the power adapter to the power connector of the HT-ICE.

Software Installation

- Step1
  Insert the HT-IDE3000 CD into the CD ROM drive, the following dialog will be shown.

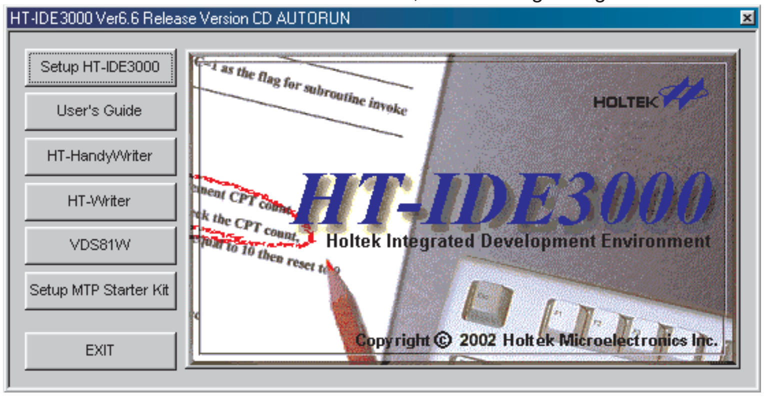
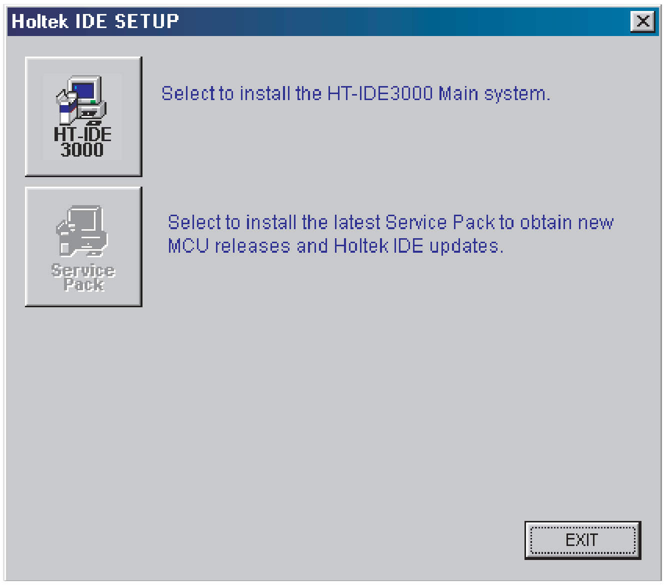

HT-IDE3000 Ver6.6 Release Version CD AUTORUN
Setup HT-IDE3000
User's Guide
HT-HandyWriter
HT-Writer
VDS81W
Setup MTP Starter Kit
EXIT
HOLTEK
HT-IDE3000
Holtek Integrated Development Environment
Copyright © 2002 Holtek Microelectronics Inc.

Fig 1-4

Click <HT-IDE3000> button and the following dialog (Fig 1-5) will be shown.



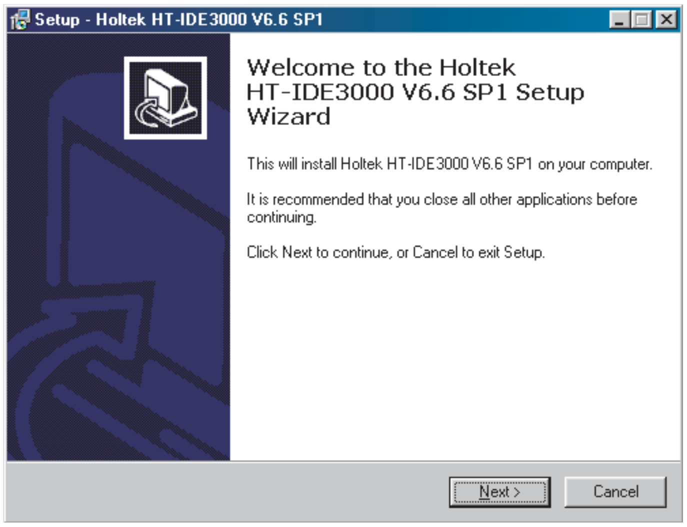
Select to install the HT-IDE3000 Main system.

Select to install the latest Service Pack to obtain new MCU releases and Holtek IDE updates.

Fig 1-5

Click <HT-IDE3000> or <Service Pack> as you want.
Here's an Example of installing HT-IDE3000
Click <HT-IDE3000> button.

- Step 2
  Press the <Next> button to continue setup or press <Cancel> button to abort.



Welcome to the Holtek HT-IDE3000 V6.6 SP1 Setup Wizard

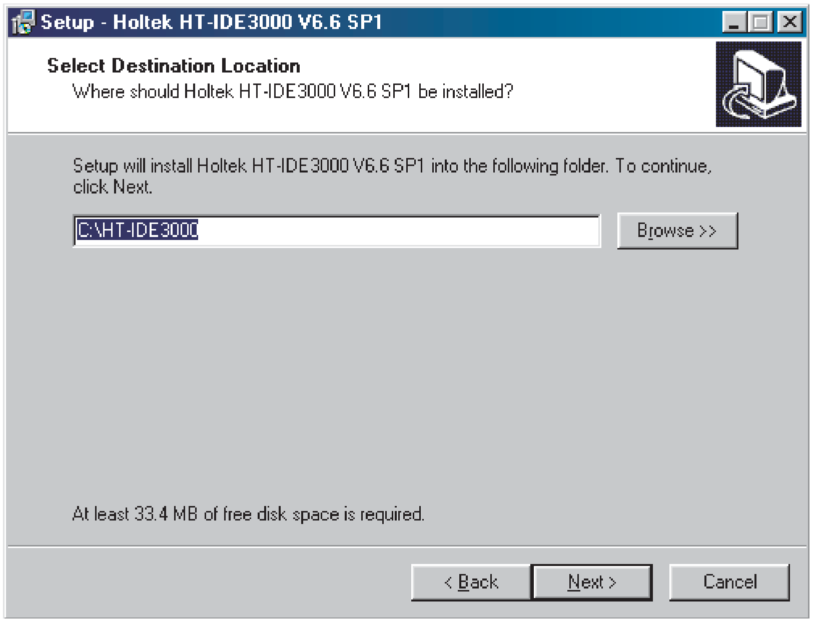
This will install Holtek HT-IDE3000 V6.6 SP1 on your computer.

It is recommended that you close all other applications before continuing.

Click Next to continue, or Cancel to exit Setup.

Fig 1-6

- Step 3
  The following dialog will be shown to ask the user to enter a directory name.



Setup - Holtek HT-IDE3000 V6.6 SP1

Select Destination Location

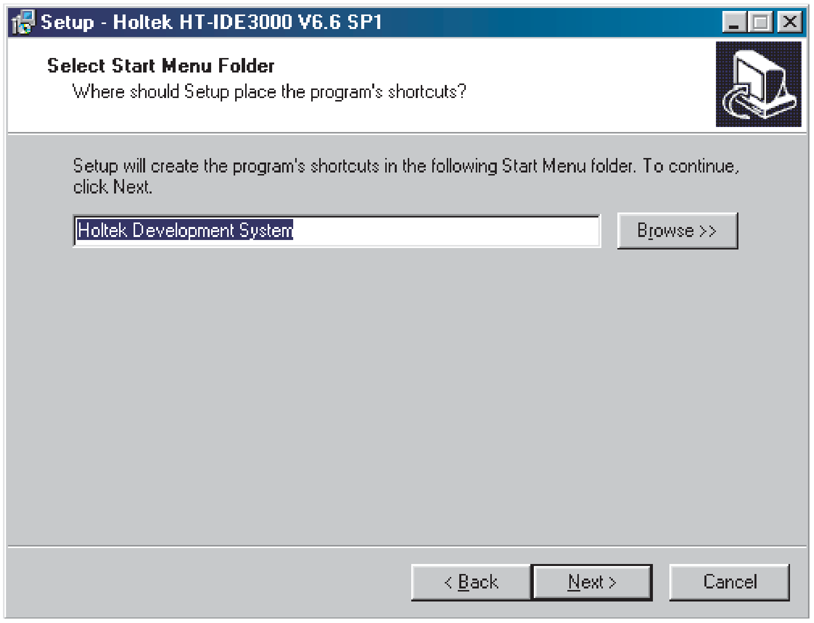
Where should Holtek HT-IDE3000 V6.6 SP1 be installed?

Setup will install Holtek HT-IDE3000 V6.6 SP1 into the following folder. To continue, click Next.

C:\HT-IDE3000

At least 33.4 MB of free disk space is required.

Fig 1-7



Setup - Holtek HT-IDE3000 V6.6 SP1

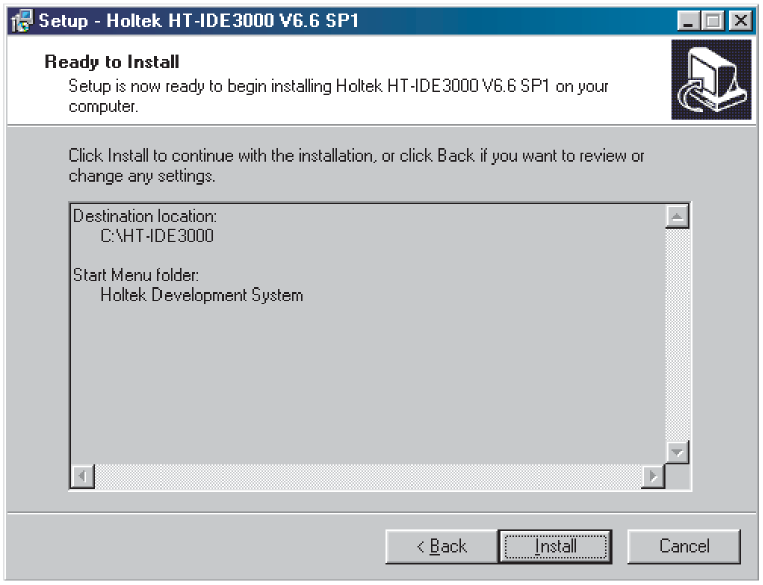
Select Start Menu Folder

Where should Setup place the program's shortcuts?

Setup will create the program's shortcuts in the following Start Menu folder. To continue, click Next.

Holtek Development System

Fig 1-8


Setup - Holtek HT-IDE3000 V6.6 SP1

Ready to Install
Setup is now ready to begin installing Holtek HT-IDE3000 V6.6 SP1 on your computer.

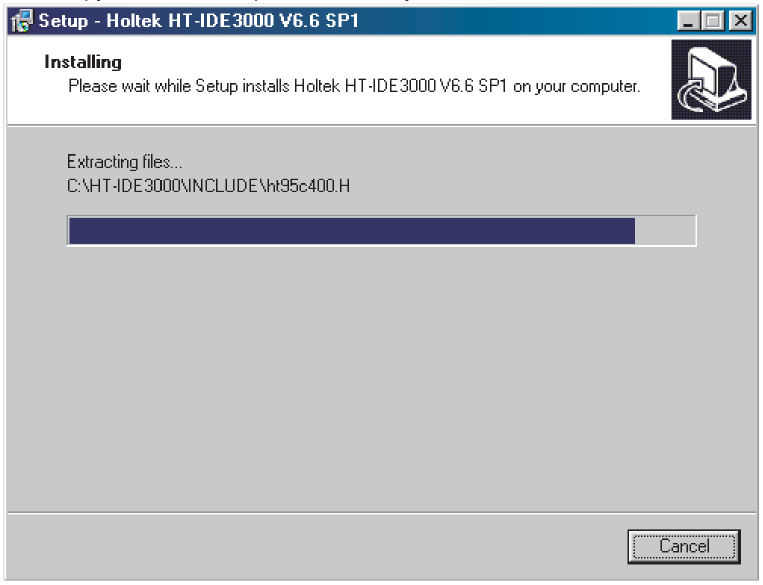
Click Install to continue with the installation, or click Back if you want to review or change any settings.

Destination location:
 C:\HT-IDE3000

Start Menu folder:
 Holtek Development System

Fig 1-9

- Step 4
  Specify the path you want to install the HT-IDE3000 and click <Next> button.
- Step 5
  SETUP will copy all files to the specified directory.


Setup - Holtek HT-IDE3000 V6.6 SP1

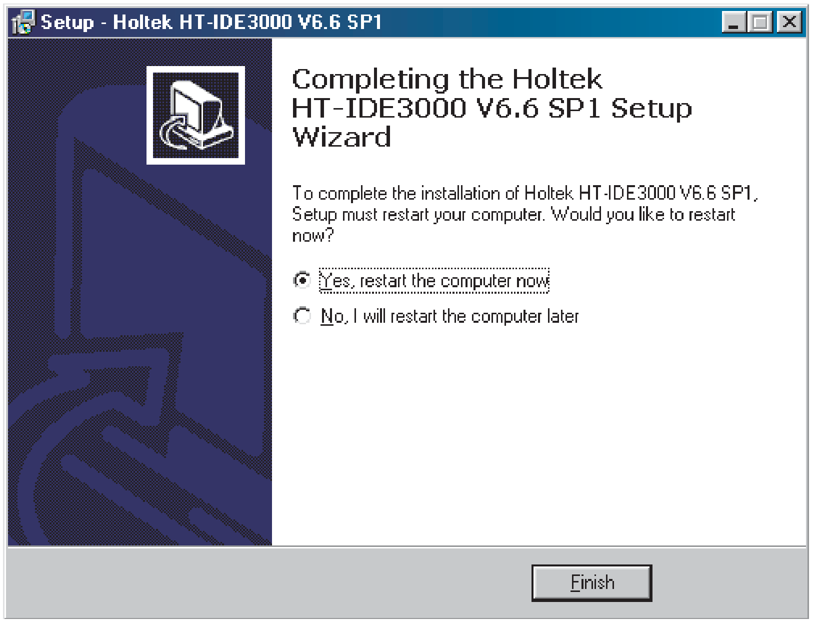
Installing
Please wait while Setup installs Holtek HT-IDE3000 V6.6 SP1 on your computer.

Extracting files...
C:\HT-IDE3000\INCLUDE\ht95c400.H

Fig 1-10

- Step 6

  If the process is successful a dialog will be shown.



Setup - Holtek HT-IDE3000 V6.6 SP1

Completing the Holtek HT-IDE3000 V6.6 SP1 Setup Wizard

To complete the installation of Holtek HT-IDE3000 V6.6 SP1, Setup must restart your computer. Would you like to restart now?

● Yes, restart the computer now

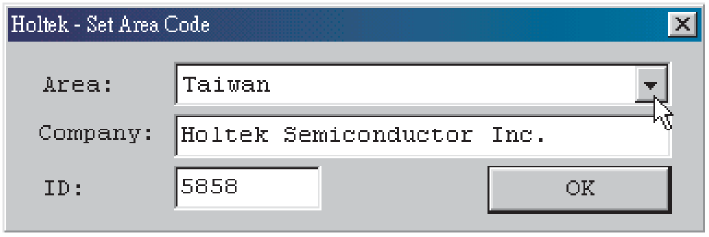
○ No, I will restart the computer later

Fig 1-11

- Step 7

  Press the Finish button and restart the computer system. Then you can run HT-IDE3000 now. SETUP will create four subdirectories, BIN, INCLUDE, LIB, SAMPLE, under the destination directory you specified in Step 4. The BIN subdirectory contains all the system executables (EXE), dynamic link libraries (DLL) and configuration files (CFG, FMT) for all supported MCU. The INCLUDE subdirectory contains all the include files (.H, .INC) provided by Holtek. The LIB subdirectory contains the library files (.LIB) provided by Holtek. The SAMPLE subdirectory contains some sample programs.

  Note that before running the HT-IDE3000 for the first time, the system will ask for company information as shown in the figure below. Select appropriate area and fill in the company name and ID. The HT-IDE3000 provider can be requested to supply an ID number.



Holtek - Set Area Code

Area: Taiwan

Company: Holtek Semiconductor Inc.

ID: 5858

Fig 1-12

Chapter 2

Quick Start

2

This chapter gives a brief description of using HT-IDE3000 to develop an application project.

Step 1 – Create a New Project

- Click on Project menu and select New command
- Enter your project name and select an MCU from the combo box
- Click OK button and the system will ask you to setup the configuration options
- Setup all configuration options and click Save button

Step 2 – Add Source Program Files to the Project

- Create your source files by using File/New command
- Write your program and save them with a file name, say TEST.ASM
- Click on Project menu and select Edit command
- An Edit Project dialog will ask you to add/delete files to/from the project
- Select a source file name, say TEST.ASM, and click Add button
- Click OK button after you setup all files in the project

Step 3 – Build the Project

- Click on Project menu and select Build command
- The system will assemble/compile all source files in the project
  - If there are some errors in the programs, double click on the error message line and the system will prompt you the position where the error happened.
  - If all the program files are error free, the system will create a Task file and download to the HT-ICE for debug.
- You may repeat this step before you finish debugging your programs

Step 4 – Programming the OTP Device

- Build the project for creating the .OTP file
- Click on Tools menu and select the Writer command to program the OTP devices

Step 5 – Transmit Code to Holtek

- Click on Project menu and select Print Option Table command
- Send the .COD file and the Option Approval Sheet to Holtek

The Programming and data flow is illustrated by the following diagram:

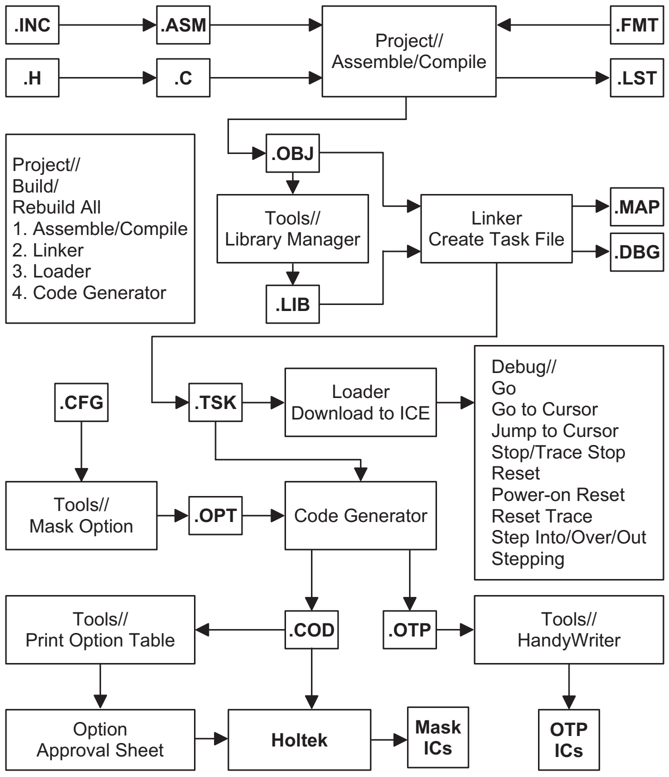


Fig 2-1

Chapter 3

Menu – File/Edit/View/Tools/Options

3

This chapter describes some of the menus and commands of the HT-IDE3000. Other menus are described in the Project, Debug and Window chapters.

Start the HT-IDE3000 System

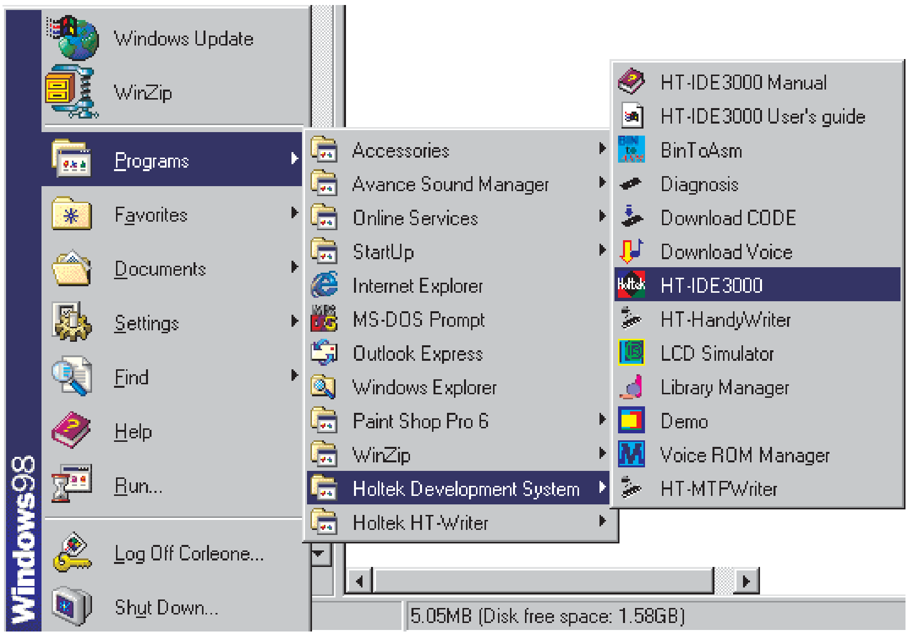
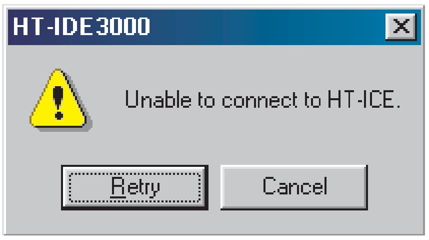


Fig 3-1

- Click Start Button, select Programs and select Holtek HT-IDE3000
  – Click the HT-IDE3000 icon

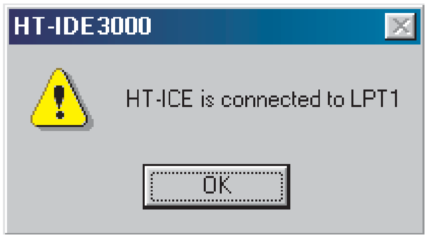
- If the last project you worked on HT-IDE3000 is in emulation mode (using HT-ICE), then Fig 3-2 will be displayed if one of the following conditions occurs.
  - No connection between the HT-ICE and the host machine or connection fails.
  - The HT-ICE is powered off.


HT-IDE3000

Unable to connect to HT-ICE.

Fig 3-2

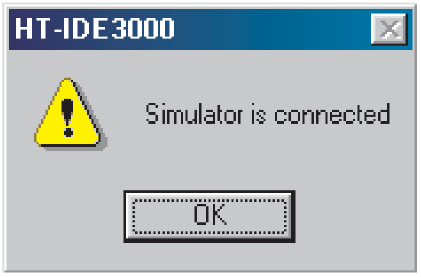
If "YES" is selected and the connection between the HT-ICE and the host machine has been made, then Fig 3-3 is displayed, the HT-IDE3000 enters the emulation mode and the HT-ICE begins to function.


HT-IDE3000

HT-ICE is connected to LPT1

Fig 3-3

- If the last project you work on HT-IDE3000 is in simulation mode (using Simulator), then Fig 3-4 will be displayed to indicate that HT-IDE3000 will enter the simulation mode.


HT-IDE3000

Simulator is connected

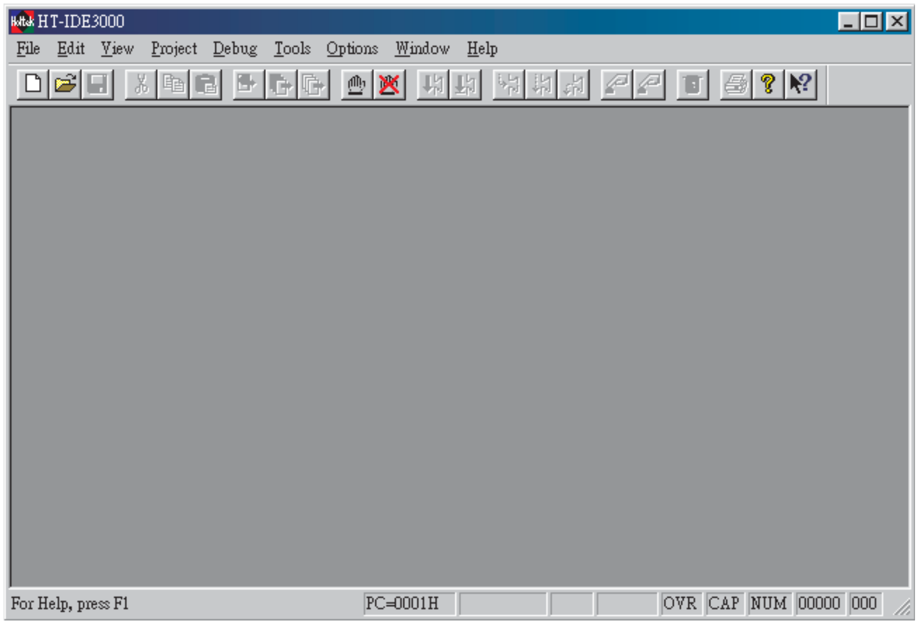
Fig 3-4

The HT-IDE3000 program supports 9 menus - File, Edit, View, Project, Debug, Tools, Options, Window and Help. The following sections describe the functions and commands of each menu.

A dockable toolbar, below the menu bar (Fig 3-5), contains icons that correspond to, and assist the user with more convenient execution of frequently used menu commands. When the cursor is placed on a toolbar icon, the corresponding command name will be displayed alongside. Clicking on the icon will cause the command to be executed.

A Status Bar, in the bottom line (Fig 3-5), displays the emulation or simulation present status and the result status of commands.

In status bar, the field (PC=0001H) displays the Program Counter while in debugging process (Debug menu).

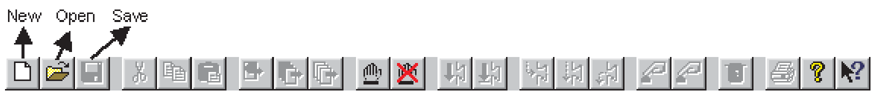


For Help, press F1 PC=0001H OVR CAP NUM 00000 000

Fig 3-5

The Status Bar contains information that may be useful during program debug. The Program Counter is used during program execution and indicates the actual present Program Counter value while the row and column indicators are used to show the present cursor position when using the program editor.

File Menu

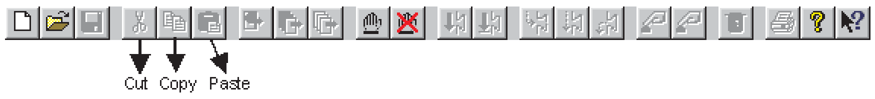


The File menu provides file processing commands, the details behind which are shown in the following list along with the corresponding toolbar icons.

- New
  Create a new file
- Open
  Open an existing file
- Close
  Close the current active file

- Save
  Write the active windows data to the active file
- Save As ...
  Write the active windows data to the specified file
- Save All
  Write all windows data to the corresponding opened files
- Print
  Print active data to the printer
- Print Setup
  Setup printer
- Recent Files
  List the most recently opened and closed four files
- Exit
  Exit from HT-IDE3000 and return to Windows

Edit Menu



- Undo
  Cancel the previous editing operation
- Redo
  Cancel the previous Undo operation
- Cut
  Remove the selected lines from the file and place onto the clipboard
- Copy
  Place a copy of the selected lines onto the clipboard
- Paste
  Paste the clipboard information to the present insertion point
- Delete
  Delete the selected information
- Find
  Search the specified word from the editor active buffer
- Replace
  Replace the specified source word with the destination word in the editor active buffer

View Menu

The View menu provides the following commands to control the window screen of the HT-IDE3000. (Refer to Fig 3-6)

- Line
  Move the cursor to the specified line (specified by line number) of the active file

- Cycle Count
  Count instruction cycles accumulatively. Press the reset button to clear the cycle count. The radio buttons Hex and Dec are used to change the radix of the count, hexadecimal or decimal. The maximum cycle count is 65535.
- Toolbar
  Display the toolbar information on the window. The toolbar contains some groups of buttons whose function is the same as that of the command in each corresponding menu item. When the mouse cursor is placed on a toolbar button, the corresponding function name will be displayed next to the button. If the mouse is clicked, the command will be executed. Refer to the corresponding chapter for the functionality of each button. The Toggle Breakpoint button will set the line specified by the cursor as a breakpoint (highlighted). The toggle action of this button will clear the breakpoint function if previously set.
- Status Bar
  Displays the status bar information on the window.

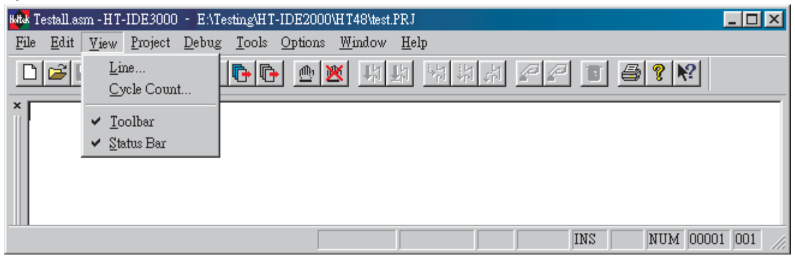

Testall.asm - HT-IDE3000 - E:\Testing\HT-IDE2000\HT48\test.PRJ
File Edit View Project Debug Tools Options Window Help
Line...
Cycle Count...
✓ Toolbar
✓ Status Bar
INS NUM 00001 001

Fig 3-6

Tools Menu

The Tools menu provides the special commands to facilitate user application debug. These commands are Configuration Option, Diagnose, Writer, Library Manager, Voice tools and LCD Simulator and virtual peripheral manager.

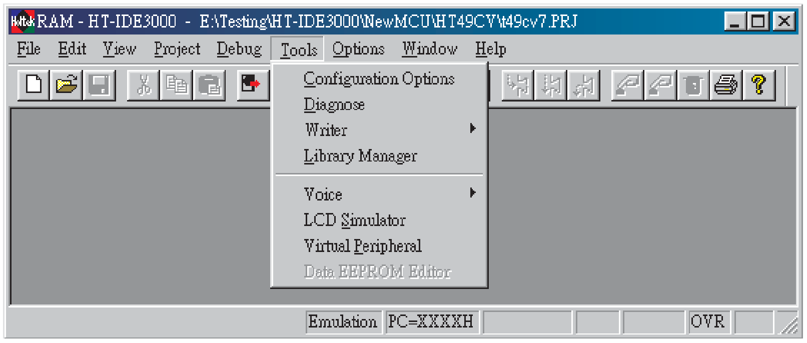


Fig 3-7

Configuration Option

This command generates an option file used by the Build command in the Project menu. The contents of the option file depend upon the specified MCU. This command allows options to be modified after creation of the project.

→ **Choosing the Clock Source**

The clock source used by the HT-ICE has to be chosen when setting the MCU options, either when creating a new project or modifying the options. The HT-ICE provides two clock sources, namely internal and external. If an external clock source is chosen, the jumper JP1 must be placed in the correct position.

- For crystal mode, add a crystal to location X1 and short positions 2 and 3 of jumper JP1 on the I/O interface card.
- For RC mode, adjust the system frequency with VR1 and short positions 1 and 2 of jumper JP1 on the I/O interface card.

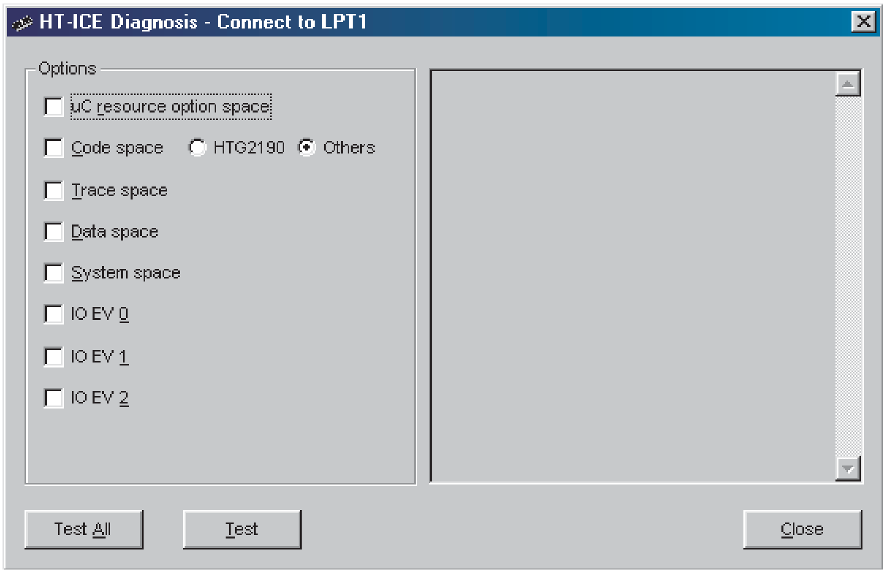
→ **Internal Clock Source**

If an internal clock source is used, the system application frequency has to be specified. The HT-IDE3000 system will calculate a frequency which can be supported by the HT-ICE, one which will be the most approximate value to the specified system frequency. Whenever the calculated frequency is not equal to the specified frequency, a warning message and the specified frequency along with the calculated frequency will be displayed. Confirmation will then be required to confirm the use of the calculated frequency or to specify another system frequency. Otherwise an external clock source is the only option. No matter which kind of clock source is chosen, the system frequency must be specified.

Diagnose

This command (Fig 3-8) helps to check whether the HT-ICE is working correctly. There are a total of 9 items for diagnosis. Multiple items can be selected by clicking the check box and pressing the Test button, or press the Test All button to diagnose all items. These items are listed below.

- MCU resource option space
  Diagnose the MCU options space of the HT-ICE.
- Code space
  Diagnose the program code memory of the HT-ICE.
- Trace space
  Diagnose the trace buffer memory of the HT-ICE.
- Data space
  Diagnose the program Data Memory of the HT-ICE.
- System space
  Diagnose the system Data Memory of the HT-ICE.
- I/O EV 0
  Diagnose the I/O EV-chip in socket 0 of the HT-ICE.
- I/O EV 1
  Diagnose the I/O EV-chip in socket 1 of the HT-ICE.
- I/O EV 2
  Diagnose the I/O EV-chip in socket 2 of the HT-ICE.
- I/O EV 3
  Diagnose the I/O EV-chip in socket 3 of the HT-ICE.



- [ ] uC resource option space
- [ ] Code space ○ HTG2190 [● Others]
- [ ] Trace space
- [ ] Data space
- [ ] System space
- [ ] IO EV 0
- [ ] IO EV 1
- [ ] IO EV 2

Fig 3-8

Writer

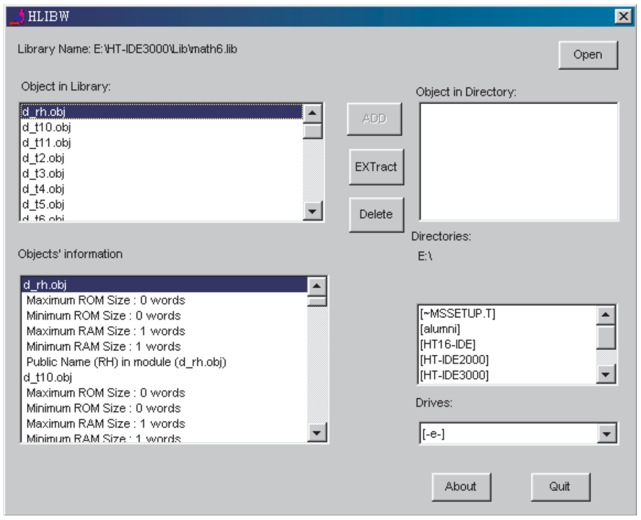
The Writer command under the Tools menu controls the OTP/MTP programming functions of the HT-ICE built-in writer. Within this command, the sub-command Handywriter is used to program all Holtek's OTP type MCU and the HT-MTPWriter is to program all Holtek's MTP type MCU. However, this command is not applicable for the other external stand-alone writer which is known as the HT-Writer. Please visit our website for the relevant information.

Library Manager

The Library Manager command, in Fig 3-9, supports the library functions. Program codes used frequently can be compiled into library files and then included in the application program by using the Project command in the Options menu. (Refer to the Cross Linker options item in Options menu, Project command). The functions of Library Manager are:

- Create a new library file or modify a library file
- Add/Delete a program module into/from a library file
- Extract a program module from a library file, and create an object file

Part III gives more details on the library manager.



Library Name: E:\HT-IDE3000\Lib\math6.lib

Object in Library:

d_rh.obj
d_t10.obj
d_t11.obj
d_t2.obj
d_t3.obj
d_t4.obj
d_t5.obj

Object in Directory:

Directories:
E:\

[~MSSETUP.T]
[alumni]
[HT16-IDE]
[HT-IDE2000]
[HT-IDE3000]

Drives:
[-e-]

Objects' information

d_rh.obj
Maximum ROM Size : 0 words
Minimum ROM Size : 0 words
Maximum RAM Size : 1 words
Minimum RAM Size : 1 words
Public Name (RH) in module (d_rh.obj)
d_t10.obj
Maximum ROM Size : 0 words
Minimum ROM Size : 0 words
Maximum RAM Size : 1 words

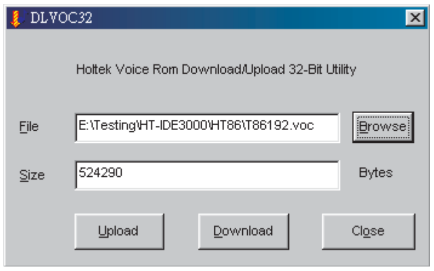
Fig 3-9

Voice/VROM Editor

Holtek provides a VROM Editor for the user to arrange the voice code for the specific MCU (eg. HT86 series)

Voice/Download

This command downloads the contents of a specified voice data file with extension name .VOC to the HT-ICE for emulation. It also uploads from the HT-ICE VROM saving the data to a specified .VOC file. Fig 3-10 displays the dialogue box which shows the name of the downloaded voice file .VOC, which was generated by the VROM Editor. The size box displays the voice ROM size in bytes for the current project¢s MCU. When uploading, a different file name from the project name may be specified to save the contents of voice ROM from the HT-ICE. Ensure that the voice ROM file .VOC has been generated by the VROM Editor before downloading.



Holtek Voice Rom Download/Upload 32-Bit Utility

File: E:\Testing\HT-IDE3000\HT86\T86192.voc

Size: 524290

Fig 3-10

LCD Simulator

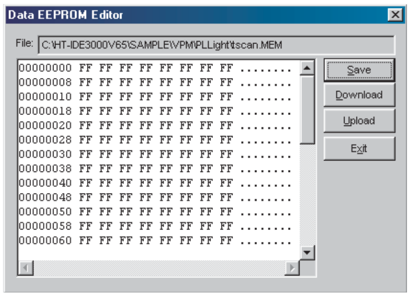
The LCD simulator HT-LCDS, provides a mechanism to simulate the output of the LCD driver. According to the designed patterns and the control programs, the HT-LCDS displays the patterns on the screen in real time. Part III gives more details on the LCD simulator.

Virtual Peripheral Manager

The Virtual Peripheral Manager (VPM) provides a mechanism to simulate the peripheral device. It must be used while the HT-IDE3000 is in simulation mode.

Data EEPROM Editor

Some Holtek's MCUs (eg. HT48E series) have internal EEPROM. The Data EEPROM Editor provides the interface for the user to arrange the data and download/upload the data to/from the HT-ICE.



File: C:\HT-IDE3000V65\SAMPLE\VPM\PLLight\tscan.MEM

```
00000000 FF FF FF FF FF FF FF FF ........
00000008 FF FF FF FF FF FF FF FF ........
00000010 FF FF FF FF FF FF FF FF ........
00000018 FF FF FF FF FF FF FF FF ........
00000020 FF FF FF FF FF FF FF FF ........
00000028 FF FF FF FF FF FF FF FF ........
00000030 FF FF FF FF FF FF FF FF ........
00000038 FF FF FF FF FF FF FF FF ........
00000040 FF FF FF FF FF FF FF FF ........
00000048 FF FF FF FF FF FF FF FF ........
00000050 FF FF FF FF FF FF FF FF ........
00000058 FF FF FF FF FF FF FF FF ........
00000060 FF FF FF FF FF FF FF FF ........
```

Fig 3-11

Options Menu

The Options menu (Fig 3-12) provides the following commands which can set the working parameters for other menus and commands.

Project Command

The Project command sets the default parameters used by the Build command in the Project menu. During development, the project options may be changed according to the needs of the application. According to the options set, the HT-IDE3000 will generate a proper task file for these options when the Build command of the Project menu is issued. The dialog box (Fig 3-13) is used for setting the options of the Project.

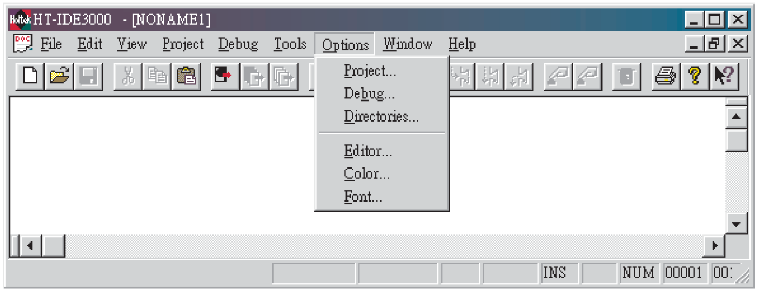


Fig 3-12

Note Before issuing the Build command, ensure that the project options are set correctly.

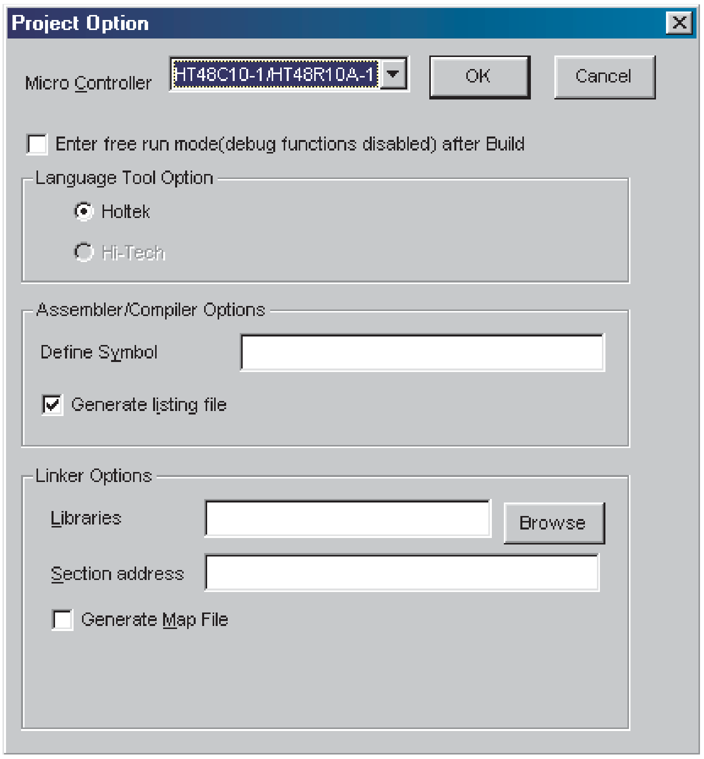


Fig 3-13

- Micro Controller
  The MCU name of this project. Use a scroll arrow to browse the available MCU names and select the appropriate one.
- Enter Free Run Mode (Debug Options Disabled) After Build
  Check this box so that HT-IDE3000 will enter free run mode after build. All the debug functions will be disabled while in free run mode.
- Language Tool Option
  Holtek permits Third Party to provide C compiler for Holtek's MCU. Currently, you can select Hi-Tech language tool as another choice.
- Assembler/Compiler Options
  The command line options of the Cross Assembler. Define symbol allows user to define value for the specified symbol which is used in the assembly program. The syntax is as follows:

```
symbol1[=value1] [,symbol2 [=value2] [,...]]
```

For example:

```
debugflag=1, newver=3
```

The check box of the Generate listing file is used to check if the source program listing file has been generated.

- Cross Linker Options
  To specify the options of the Cross Linker. Libraries are used to specify the library files refered by Cross Linker. For example:

  ```
  libfile1, libfile2
  ```

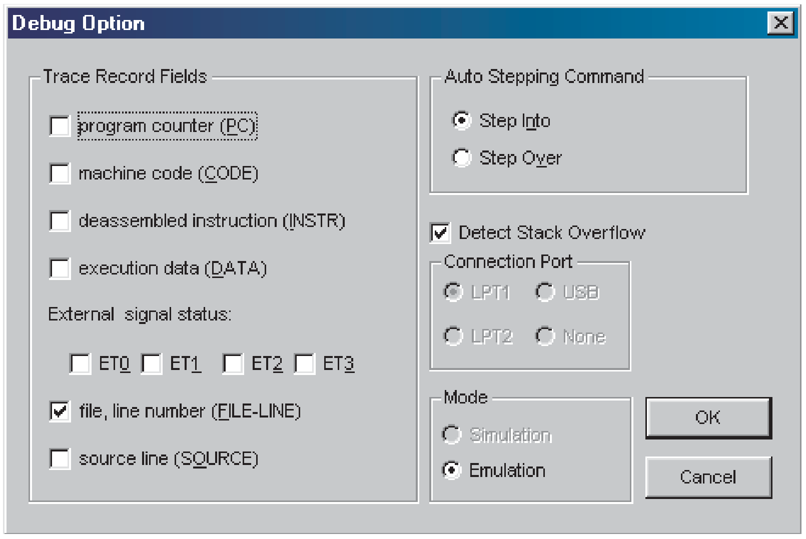
  Library files can be selected by clicking the Browse button.
  Section address is used to set the ROM/RAM addresses of the specified sections, for example:

  ```
  codesec=100, datasec=40
  ```

  The check box of the Generate map file is used to check if the map file of Cross Linker is generated.

Debug Command

This command sets the options used by the Debug menu (Chapter 5 HT-IDE3000 menu - Debug). The dialog box (Fig 3-14) lists all the debug options with check boxes. By selecting the options and pressing the OK button, the Debug menu can then obtain these options during the debugging process.


Debug Option

Trace Record Fields
- program counter (PC)
- machine code (CODE)
- deassembled instruction (INSTR)
- execution data (DATA)
- External signal status: ET0 ET1 ET2 ET3
- [✓ file, line number (FILE-LINE)]
- source line (SOURCE)

Auto Stepping Command: [✓ Step Into] Step Over

[✓ Detect Stack Overflow]

Connection Port: LPT1 USB LPT2 None

Mode: Simulation [✓ Emulation]

OK Cancel

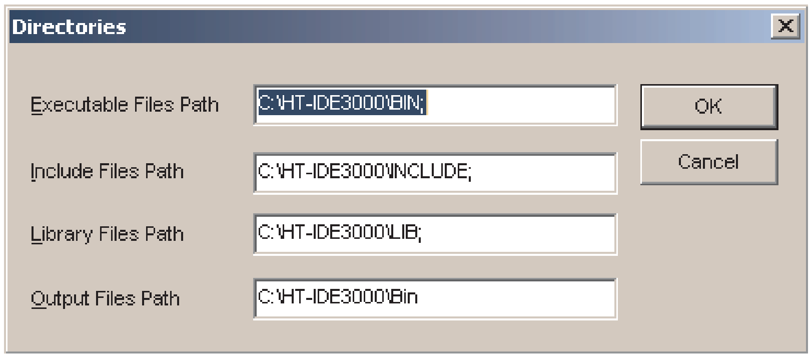
Fig 3-14

- Trace Record Fields
  This location specifies the information to be displayed when issuing the Trace List command, contained within the Window menu. For each source file instruction, the information will be displayed in the same order as that of the items in the dialog box, from top to the bottom. If no item has been selected, the next selected item will be moved forward. The default trace list will display the file name and line number only. The de-assembled instruction is obtained from the machine code, and the source line is obtained from the source file. The execution data is the read data if the execution is a read operation only, and it is the written data if the execution is a write only or read and write operation. The external signal status has no effect if the simulation mode is selected.

- Auto Stepping Command
  Selects the automatic call procedure step option, namely Step Into or Step Over. Only one option can be selected.
- Connection Port
  Selects the PC connection port for the HT-ICE. One PC parallel port, LPT1, LPT2 or LPT3 can be selected for connection to the HT-ICE. The connection port has no effect if the simulation mode is selected.
- Mode
  Selects the HT-IDE3000 working mode as either simulation or emulation mode. If the HT-ICE is connected to the host machine and powered on, the HT-IDE3000 can be selected to be either in emulation or simulation mode.
- Detect Stack Overflow
  Uncheck this box if you don't want the system to show a message while detecting stack overflow.

Directories Command

The command sets the default search path and directories for saving files. (Fig 3-15)



Executable Files Path: C:\HT-IDE3000\BIN;
Include Files Path: C:\HT-IDE3000\INCLUDE;
Library Files Path: C:\HT-IDE3000\LIB;
Output Files Path: C:\HT-IDE3000\Bin

Fig 3-15

- Executable files path
  The search path referred to by the HT-IDE3000 when the executable files are called.
- Include files path
  The search path referred to by the Cross Assembler to search for the included files.
- Library files path
  The search path referred to by the Cross Linker to search for the library files.
- Output files path
  The directory for saving the output files of the Cross Assembler (.obj, .lst) and Cross Linker (.tsk, .map, .dbg)

Editor Command

This command sets the editor options such as tab size and Undo command count. The Save Before Assemble option will save the file before assembly. The Maximum Undo Count is the maximum allowable counts of consecutive undo operations.

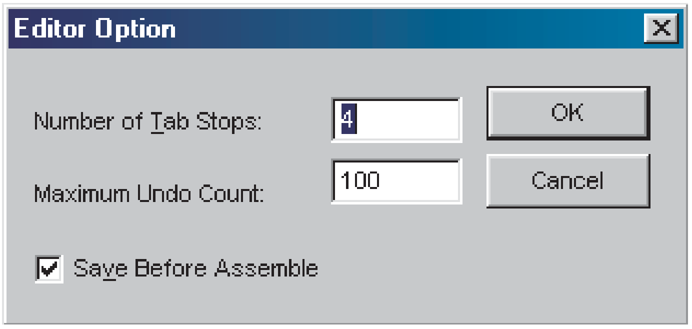

Editor Option
Number of Tab Stops: 4
Maximum Undo Count: 100
[✓] Save Before Assemble

Fig 3-16

Color Command

This command sets the foreground and background colors for the specified line. From the available options (Fig 3-17), Text Selection is used for the Edit menu, Current line, Breakpoint Line, Trace Line and Stack Line are for the Debug menu and Error line is for the Assembler output.

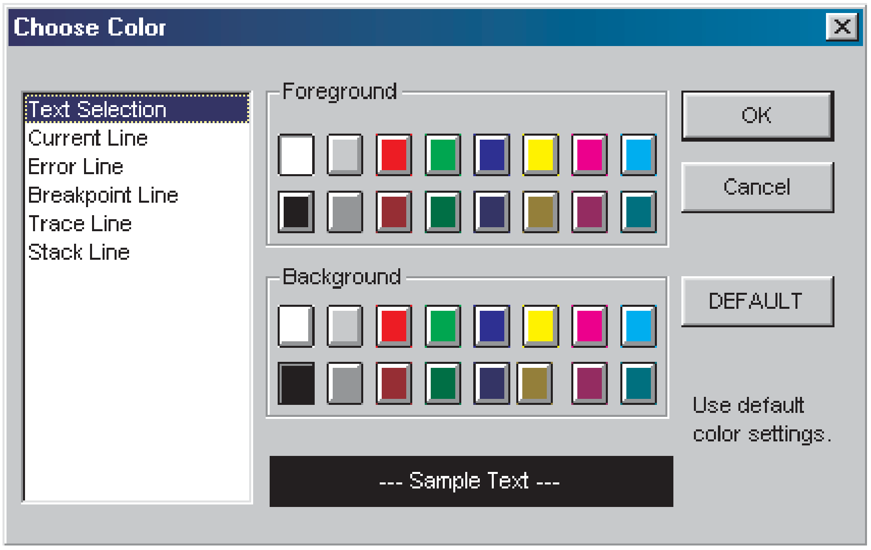


Fig 3-17

Font Command

This command will change the displayed fonts.

Chapter 4

Menu – Project

4

The HT-IDE3000 provides an example Project, which will assist first time users in quickly familiarizing themselves with project development. It should be noted that from the standpoint of the HT-IDE3000 system, a working unit is a project with each user application described by a unique project.

When developing an HT-IDE3000 application for the first time, the development steps, as described earlier, are recommended.

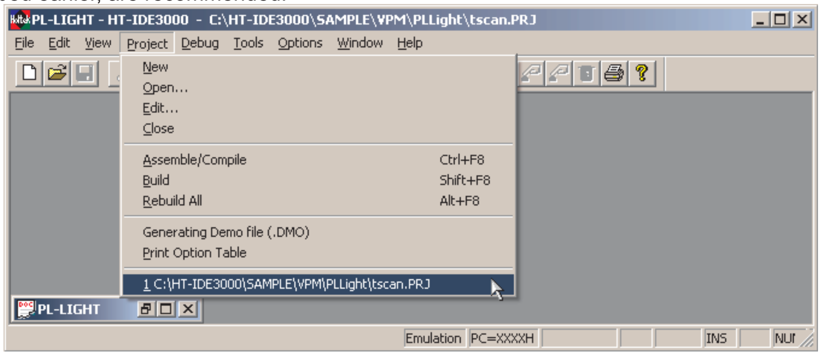
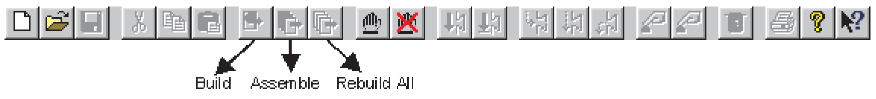

PL-LIGHT - HT-IDE3000 - C:\HT-IDE3000\SAMPLE\VPM\PLLight\tscan.PRJ
File Edit View Project Debug Tools Options Window Help
New
Open...
Edit...
Close
Assemble/Compile Ctrl+F8
Build Shift+F8
Rebuild All Alt+F8
Generating Demo file (.DMO)
Print Option Table
1 C:\HT-IDE3000\SAMPLE\VPM\PLLight\tscan.PRJ
PL-LIGHT
Emulation PC=XXXXH INS NUM

Fig 4-1

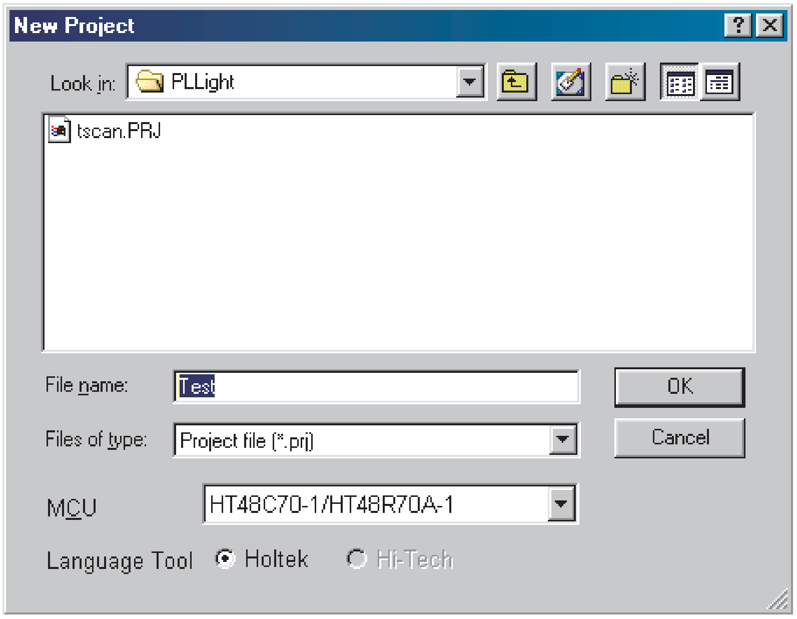
Create a New Project

In the Project menu (Fig 4-1), select the New command to create a new project. In this command, the user needs to key in or select two pieces of information for the new project, namely the Project



Name and the Micro Controller (Fig 4-2). The user may browse all directories and all existing projects and select one of them (to overwrite the old project) and to choose one of the available MCU.

Note The project name is a file name with the extension .PRJ.



File name: Test

Files of type: Project file (*.prj)

MCU HT48C70-1/HT48R70A-1

Language Tool [✓ Holtek] Hi-Tech

Fig 4-2

Open and Close a Project

The HT-IDE3000 can work with only one project at a time, which is the opening project, at any time. If a project is to be worked upon, the project should first be opened, by using the Open command of the Project menu (Fig 4-1). Then, insert the project name directly or browse the directories and select a project name. Use the Close command to close the project.

Note When opening a project, the current project is closed automatically.
Within the development period, i.e. during editing, setting options and debugging etc., ensure that the project is in the open state. This is shown by the displaying of the project name of the opening project on the title of the HT-IDE3000 window. Otherwise, the results are unpredictable.
The HT-IDE3000 will retain the opening project information if the system exits from the HT-IDE3000 without closing the opening project. This project will be opened automatically the next time the HT-IDE3000 is run.

Manage the Source Files of a Project

Use the Edit command to add or remove source program files from the opened project. The order, from top to bottom, of each source file in the list box, is the order of the input files to the Cross Linker. The Cross Linker processes the input files according to the order of these files in the box. Two buttons, namely [Move Up] and [Move Down], can be used to adjust the order of a source file in the project. Fig 4-3 is the dialog box of the Project menu¢s Edit command.

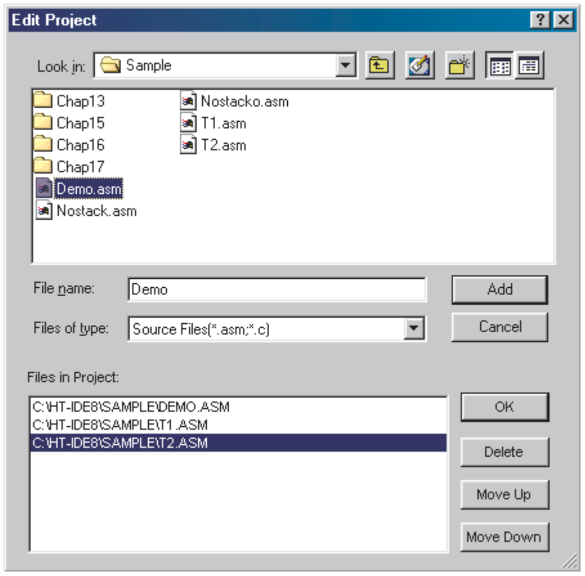


Fig 4-3

To Add a Source File to the Project

- Type the source file name into the text box of the File Name in the Edit dialog box
- Alternatively, choose the source file type and browse the List Files.
  - Choose the drive and directory where the source files are located by using the browse Drives and Directories items
  - Choose a source file name from the list box below the File name item
  - Double-click the selected file name or choose the Add button to add the source files to the project

When the selected source file has been added, this file name is displayed on the list box of the Files in project.

To Delete a Source File from the Project

- Choose the file to be deleted from the project
- Click the Delete button

Note Deleting the source files from the project does not actually delete the file but refers to the removal of the file information from the project.

To Move a Source File Up or Down

- Choose the file to be moved in the list box (Files in project), by moving the cursor to this file and clicking the mouse button
- Click the [Move Up] button or the [Move Down] button

Build a Project′s Task Files

Be sure that the following tasks have been completed before building a new project:

- The project has been opened
- The project options have all been set
- The project source files have been added
- The MCU options have been set (refer to the Tools menu chapter)

There are two commands related to the building of a project file, the Build command and the Rebuild All command.

The Project menu′s Build command performs the following operations:

- Assemble or compile all the source files of the current project, by calling the Cross Assembler or C compiler depends on the file extension .asm or .C
- Link all the object files generated by the Cross Assembler or C compiler, and generate a task file and a debugging file.
- Load the task file into the HT-ICE if it is powered-on
- Display the source program of the execution entry point on the active window (the HT-IDE3000 refers to the source files, the task file and the debugging file for emulation)

Note The Build command may or may not execute the above tasks as the execution is dependent on the creation date/time of all corresponding files.
The rules are:

- If the creation date/time of a source file is later than that of its object file, then the Cross Assembler or C compiler is called to assemble, compile this source file and to generate a new object file.
- If one of the task′s object files has a later creation date/time than that of the task file, then the Cross Linker is called to link all object files of this task and to generate a new task file.

The Build command downloads the task file into the HT-ICE automatically whether there is an action or not.

The Rebuild All command carries out the same task as the Build command. The difference is that the Rebuild All command will execute the task immediately without first checking the creation date/time of the project files.

The result message of executing a Build or Rebuild All command are displayed on the Output window. If an error occurs in the processing procedure, the actions following it are skipped, and no task file is generated, and no download is performed.

To Build a Project Task File

- Click the Open command of the Project menu to open the project
- Click either the Build command of the Project menu or the Build button on the toolbar (Fig 4-1) to start building a project

To Rebuild a Project Task File

- Click the Open command of the Project menu to open the project
- Click either the Rebuild All command of the Project menu or the Rebuild all button on the toolbar (Fig 4-1) to start building a project

Once the project task has been built successfully, emulation and debugging of the application program can begin (refer to the HT-IDE3000 menu - Debug chapter).

Assemble/Compile

To verify the integrity of application programs, this command can be used to assemble or compile the source code and display the result message in the Output window.

To Assemble or Compile a Program

- Use the File menu to open the source program file to be assembled or compiled
- Either select the Assemble/Compile command of the Project menu or click the Assemble button on the toolbar to assemble/compile this program file

If the opened file has an .asm file extension name, the Cross Assembler will execute the assembly process. If the file has a .C extension then the Holtek C compiler will compile the program.

If no errors are detected, an object file with extension .OBJ is generated and stored in the directory which is specified in the Output Files Path (refer to Options menu, Directories command). If an error occurs and a corresponding message displayed on the Output window, one of the following commands can be used to move the cursor to the error line:

- Double-click the left button of the mouse or
- Select the error message line on the Output window, and press the <Enter> key

Print Option Table Command

This command will print the current active option file to the specified printer. A printer may be selected where the options file is to be printed out. It is recommended to use a different printer port from the port which is connected to the HT-ICE.

If both the printer and the HT-ICE are using the same printer port, issuing this command will cause the loss of all debug information and corresponding data. After the printing job has finished, the user should proceed to the very beginning of the development procedure and use the Build command of the Project menu if further emulation/debugging of the application program is required.

Generate Demo File (.DMO) Command

This command will generate a file (.dmo) for HT-DEMO. User can carry HT-DEMO with the .dmo file and demonstrate his project on a PC without the installation of HT-IDE3000.

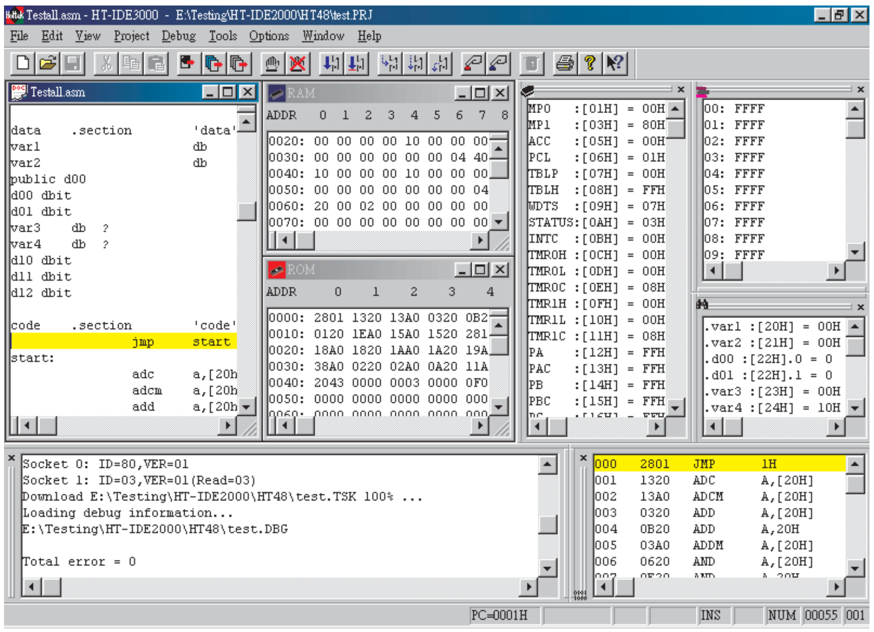
Chapter 5

Menu – Debug

5

In the development process, the repeated modification and testing of source programs is an inevitable procedure. The HT-IDE3000 provides many tools not only to facilitate the debugging work, but also to reduce the development time. Included are functions such as single stepping, symbolic breakpoints, automatic single stepping, trace trigger conditions, etc.

After the application program has been successfully constructed, (refer to the chapter on Build a project's task files) the first execution line of the source program is displayed and highlighted in the active window (Fig 5-1). The HT-IDE3000 is now ready to accept and execute the debug commands.


Testall.asm - HT-IDE3000 - E:\Testing\HT-IDE2000\HT48\test.PRJ
File Edit View Project Debug Tools Options Window Help
Testall.asm
RAM
ROM
PC=0001H
INS
NUM
00055
001

```
data    .section            'data'
var1                        db
var2                        db
public d00
d00 dbit
d01 dbit
var3    db  ?
var4    db  ?
d10 dbit
d11 dbit
d12 dbit

code    .section            'code'
                jmp         start
start:
                adc         a,[20h
                adcm        a,[20h
                add         a,[20h
```

```
ADDR   0  1  2  3  4  5  6  7  8
0020: 00 00 00 00 10 00 00 00
0030: 00 00 00 00 00 00 04 40
0040: 10 00 00 00 10 00 00 00
0050: 00 00 00 00 00 00 00 04
0060: 20 00 02 00 00 00 00 00
0070: 00 00 00 00 00 00 00 00
```

```
ADDR     0    1    2    3    4
0000: 2801 1320 13A0 0320 0B2
0010: 0120 1EA0 15A0 1520 281
0020: 18A0 1820 1AA0 1A20 19A
0030: 38A0 0220 02A0 0A20 11A
0040: 2043 0000 0003 0000 0F0
0050: 0000 0000 0000 0000 000
```

```
MP0    :[01H] = 00H
MP1    :[03H] = 80H
ACC    :[05H] = 00H
PCL    :[06H] = 01H
TBLP   :[07H] = 00H
TBLH   :[08H] = FFH
WDTS   :[09H] = 07H
STATUS:[0AH] = 03H
INTC   :[0BH] = 00H
TMR0H :[0CH] = 00H
TMR0L :[0DH] = 00H
TMR0C :[0EH] = 08H
TMR1H :[0FH] = 00H
TMR1L :[10H] = 00H
TMR1C :[11H] = 08H
PA     :[12H] = FFH
PAC    :[13H] = FFH
PB     :[14H] = FFH
PBC    :[15H] = FFH
```

```
00: FFFF
01: FFFF
02: FFFF
03: FFFF
04: FFFF
05: FFFF
06: FFFF
07: FFFF
08: FFFF
09: FFFF
```

```
.var1 :[20H] = 00H
.var2 :[21H] = 00H
.d00 :[22H].0 = 0
.d01 :[22H].1 = 0
.var3 :[23H] = 00H
.var4 :[24H] = 10H
```

```
Socket 0: ID=80,VER=01
Socket 1: ID=03,VER=01(Read=03)
Download E:\Testing\HT-IDE2000\HT48\test.TSK 100% ...
Loading debug information...
E:\Testing\HT-IDE2000\HT48\test.DBG

Total error = 0
```

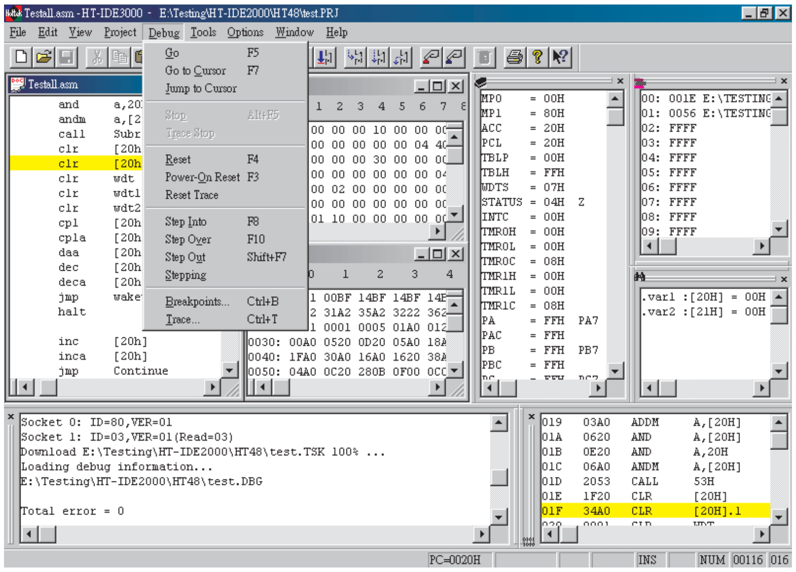
```
000     2801    JMP     1H
001     1320    ADC     A,[20H]
002     13A0    ADCM    A,[20H]
003     0320    ADD     A,[20H]
004     0B20    ADD     A,20H
005     03A0    ADDM    A,[20H]
006     0620    AND     A,[20H]
```

Fig 5-1

Reset the HT-IDE3000 System

There are 4 kinds of reset methods in the HT-IDE3000 system:

- Power-on reset (POR) by plugging in the power adapter or pressing the reset button on the HT-ICE
- Reset from the target board
- Software reset command in the HT-IDE3000 Debug menu (Fig 5-2)
- Software power-on reset command in the HT-IDE3000 Debug menu (Fig 5-2)



```
        and     a,20
        andm    a,[2
        call    Subr
        clr     [20h
        clr     [20h
        clr     wdt
        clr     wdt1
        clr     wdt2
        cpl     [20h
        cpla    [20h
        daa     [20h
        dec     [20h
        deca    [20h
        jmp     wake
        halt

        inc     [20h]
        inca    [20h]
        jmp     Continue
```

```
 1  2  3  4  5  6  7  8
00 00 00 10 00 00 00
00 00 00 00 00 04 40
00 00 00 30 00 00 00
00 00 00 00 00 00 04
00 02 00 00 00 00 00
00 00 00 00 00 00 00
01 10 00 00 00 00 00
```

```
0     1     2     3     4
1  00BF  14BF  14BF  14B
2  31A2  35A2  3222  362
1  0001  0005  01A0  012
0030:  00A0  0520  0D20  05A0  18A
0040:  1FA0  30A0  16A0  1620  38A
0050:  04A0  0C20  280B  0F00  0CC
```

```
MP0    = 00H
MP1    = 80H
ACC    = 20H
PCL    = 20H
TBLP   = 00H
TBLH   = FFH
WDTS   = 07H
STATUS = 04H  Z
INTC   = 00H
TMR0H  = 00H
TMR0L  = 00H
TMR0C  = 08H
TMR1H  = 00H
TMR1L  = 00H
TMR1C  = 08H
PA     = FFH  PA7
PAC    = FFH
PB     = FFH  PB7
PBC    = FFH
```

```
00: 001E E:\TESTING
01: 0056 E:\TESTING
02: FFFF
03: FFFF
04: FFFF
05: FFFF
06: FFFF
07: FFFF
08: FFFF
09: FFFF
```

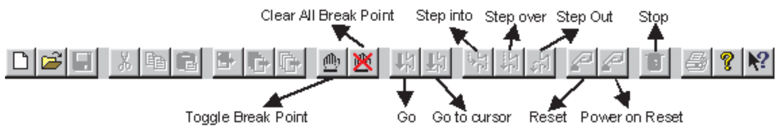
```
.var1 :[20H] = 00H
.var2 :[21H] = 00H
```

```
Socket 0: ID=80,VER=01
Socket 1: ID=03,VER=01(Read=03)
Download E:\Testing\HT-IDE2000\HT48\test.TSK 100% ...
Loading debug information...
E:\Testing\HT-IDE2000\HT48\test.DBG

Total error = 0
```

```
019   03A0   ADDM   A,[20H]
01A   0620   AND    A,[20H]
01B   0E20   AND    A,20H
01C   06A0   ANDM   A,[20H]
01D   2053   CALL   53H
01E   1F20   CLR    [20H]
01F   34A0   CLR    [20H].1
```

Fig 5-2



The effects of the above 4 types of reset are listed in table 5-1.

Reset Item	Power-On Reset	Target Board Reset	Software Reset Command	Software Power-On Reset Command
Clear Registers	(*)	(*)	(*)	(*)
Clear Options	Yes	No	No	No
Clear PDF, TO	Yes	No	No	Yes
PC Value	(**)	0	0	0
Emulation Stop	(**)	No(***)	Yes	Yes
Check Stand-Alone	Yes	No	No	No

Table 5-1

Note (*) : Refer to the Data Book of the corresponding MCU for the effects of registers under the different resets.
(**) : The PC value is 0 and the emulation stops.
(***) : If the reset is from the target board, the MCU will start emulating the application after the reset is completed.

PC - Program Counter
PDF - Power Down Flag
TO - Time-out Flag

To Reset from the HT-IDE3000 Commands

- Either choose the Reset command from Debug menu or click the Reset button on the toolbar to execute a software reset
- Either choose the Power-on Reset command from the Debug menu or click the Power-on Reset button to execute a software power on reset

To Reset from the Target Board

The target board circuit can take advantage of the μ_RES pin (pin 03-C) on the DIN connector to design a MCU reset button. The effect of this reset is listed in table 5-1.

Emulation of Application Programs

After the application program has been successfully written and assembled, the Build or Rebuild command should be executed. If successful, the first executable line of the source program will be displayed and highlighted on the active window (Fig 5-1). At this point, emulation of the application program can begin by using the HT-IDE3000 debug commands.

Note During emulation of an application program, the corresponding project has to be open.

To Emulate the Application Program

- Choose the Go command from the Debug menu
  or press the hot key F5
  or press the Go button on the toolbar

Other windows can be activated during emulation. The HT-IDE3000 system will automatically stop the emulation if a break condition is met. Otherwise, it will continue emulating until the end of the application program. The Stop button on the toolbar is illuminated with a red color while the HT-ICE is in emulation. Pressing this button will stop the emulation process.

To Stop Emulating the Application Program

There are three methods to stop the emulation, shown as follows:

- Set the breakpoints before starting the emulation
- Choose the Stop command of the Debug menu or press the hot key Alt+F5
- Press the Stop button on the toolbar

To Run the Application Program to a Line

The emulation may be stopped at a specified line when debugging a program. The following methods provide this function. All instructions between the current point and the specified line will be executed except the conditional skips. Note however that the program may not stop at the specified line due to conditional jumps or other situations.

- Move the cursor to the stopped line (or highlight this line)
- Choose the Go to Cursor command of the Debug menu
  or press the hot key F7
  or press the Go to Cursor button on the toolbar

To Directly Jump to a Line of an Application Program

It is possible to jump directly to a line, if the result of executed instructions between the current point and the specified line are not important. This command will not change the contents of Data Memory, registers and status except for the Program Counter. The specified line is the next line to be executed.

- Move the cursor to the appropriate line or highlight this line
- Choose Jump to Cursor command of the Debug menu

Single Step

The execution results of some instructions in the above section may be viewed and checked. It is also possible to view the execution results one instruction at a time, i.e., in a step-by-step manner. The HT-IDE3000 provides two step modes, namely manual mode and automatic mode.

In the manual mode, the HT-IDE3000 executes exactly one step command each time the single-step command is executed. In the automatic mode, the HT-IDE3000 executes single step commands continuously until the emulation stop command is issued, using the Stop command of the Debug menu. In the automatic mode, all user specified breakpoints are discarded and the step rate can be set from FAST, 0.5, 1, 2, 3, 4 to 5 seconds. There are 3 step commands, namely Step Into, Step Over and Step Out.

- The Step Into command executes exactly one instruction at a time, however, it will enter the procedure and stop at the first instruction of the procedure when it encounters a CALL procedure instruction.
- The Step Over command executes exactly one instruction at a time, however upon encountering a CALL procedure, will stop at the next instruction after the CALL instruction instead of entering the procedure. All instructions of this procedure will have been executed and the register contents and status may have changed.
- The Step Out command is only used when inside a procedure. It executes all instructions between the current point and the RET instruction (including RET), and stops at the next instruction after the CALL instruction.

Note The Step Out command should only be used when the current pointer is within a procedure or otherwise unpredictable results may happen.

The two step commands, Step Into and Step Over, in the automatic mode are set using the Debug sub-menu of the Options menu

- To start automatic single step mode
  Choose the Stepping command from the Debug menu
  also choose the stepping speed (the step command is set in the Debug command from the Options menu)
- To end automatic single step mode
  Choose the Stop command from the Debug menu
- To change automatic single step command for the automatic mode
  - Choose the Debug command from the Options menu
  - Choose the Step Into or the Step Over command in the Stepping command box
- To start Step Into
  Choose the Step Into command from the Debug menu
  or press the hot key F8
  or press the Step Into button on the toolbar
- To start Step Over
  Choose the Step Over command of the Debug menu
  or press the hot key F10
  or press the Step Over button on the toolbar
- To start Step Out
  Choose the Step Out command of the Debug menu
  or press the hot key Shift+F7
  or press the Step Out button on the toolbar

Breakpoints

The HT-IDE3000 provides a powerful breakpoint mechanism which accepts various forms of conditioning including program address, source line number and symbolic breakpoint, etc.

Breakpoint Features

The following are the main features of the HT-IDE3000 breakpoint mechanism:

- At most 3 breakpoints with equal priority can take effect at any instant.
- Any breakpoint will be recorded in the breakpoints list box after it is set, however this breakpoint may not be immediately effective. It can be set to be effective later, as long as it is not deleted, i.e.still in the breakpoints list box.
- It is acceptable to add at most 20 breakpoints to the list box simultaneously. At least one breakpoint should be deleted first, if a 21st breakpoint is to be added.
- Breakpoints of address or data, in binary form with don′t-care bits, are permitted.
- When an instruction is set to be an effective breakpoint, the HT-ICE will stop at this instruction, but will not execute it, i.e. this instruction will become the next one to be executed. Although an instruction is an effective breakpoint, the HT-ICE may not stop at this instruction due to execution flow or conditional skips. If an effective breakpoint is in the Data Space (RAM), the instruction that matches this conditional breakpoint data will always be executed. The HT-ICE will stop at the next instruction.

Description of Breakpoint Items

A breakpoint consists of the following descriptive items. It is not necessary to set all items, Fig 5-3:

- Space
  The location of the breakpoint, either Program Code space or Data space.
- Location
  The actual location of the breakpoint. The next paragraph will give the location format.
- Content
  The data content of breakpoint. This item is effective only when the Space is assigned to the Data space. The Read and Write check box are used for executing conditions of the breakpoint.
- Externals
  External signal breakpoint. There are 4 external signals, ET0, ET1, ET2 and ET3 at location JP3 on the I/O interface card.

→ **Format of Description Items – Location**
The allowed formats of Location items are:

- Absolute address (in code space or data space) with 4 format types, namely decimal, hexadecimal (suffix with ″H″ or ″h″), binary and don′t-care bits. For example

20, 14h, 00010100b, 10xx0011

represents decimal 20, hexadecimal 14h, binary 00010100b and don′t-care bits 4 and 5 respectively.

Note Don′t-care bits must be in binary format.

- Line number with or without source file name, the format is:

  ```
  [source_file_name!].line_number
  ```

  where the ***source_file_name*** is a name of the optional source file. If there is no file name, the current active file is assumed. The exclamation point ²!² is necessary only when a source file name is specified. The dot . must prefix the line number which is decimal.
  Example:

  ```
  C:\HIDE\USER\GE.ASM!.42
  ```

  sets the breakpoint at the 42nd line of the file GE.ASM in directory \HIDE\USER of drive C.
  Example:

  ```
  .48
  ```

  sets the breakpoint at the 48th line of the current active file.
- Program symbol with or without the source file name. The format is

  ```
  [source_file_name!].symbol_name
  ```

  All are the same as the line number location format except that the line_number is replaced with symbol_name. The following program symbols are acceptable:
  – Label name
  – Section name
  – Procedure name
  – Dynamic data symbols defined in data section

→ **Format of Description Items – Content and External Signals**
The format of the content and external signals have four digital number options, similar to the format of Location absolute address. These four types of number are decimal, hexadecimal, binary and don¢t-care bits.

→ **Format of Breakpoints List Box**
The Breakpoints list box contains all the breakpoints that have been added, including effective breakpoints and non-effective breakpoints. The Add button should be used to add new breakpoints to the list box, and the Delete button to remove breakpoints from the list box. The format of each breakpoint in the list box is as follows:

```
<status> {<space and read/write>, <location>,
<data content>, <external signal>}
```

where <status> is effective status. ″+″ is effective (enabled) and ″–″ is non-effective (disabled). <space and read/write> is the space type and operating mode. ″C″ is the code space, "D/R" is the data space with read, ″D/W″ is the data space with write, ″D/RW″ is the data space with read and write.

<location>, <data content> and <external signal> have the same data format as the input form respectively.

How to Set Breakpoints

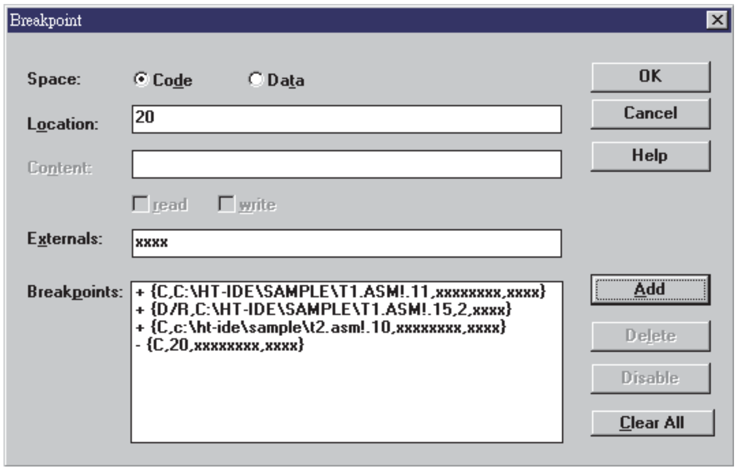
There are two methods to set/enable a breakpoint, one is by using the Breakpoint command from the Debug menu, the other is by using the Toggle Breakpoint button on the toolbar. The rules of the breakpoint mechanism are as follows:

- If the breakpoint to be set is not in the Breakpoints list box (Fig 5-3), then the descriptive items must be designated first, then added to the Breakpoints list box.
- As long as the breakpoint exists in the list box, it can be made effective by Enabling the breakpoint if it fails to be initially effective.
- Press the OK button for confirmation. Otherwise, all changes here will not be effective.
- When using the Toggle Breakpoint button on the toolbar, the cursor should first be moved to the breakpoint line, and then the Toggle Breakpoint button pressed. If an effective breakpoint is to be changed to a non-effective breakpoint, this can be achieved by merely pressing the Toggle breakpoint button.

→ **To Add a Breakpoint**

- Choose the Breakpoint command from the Debug menu (or press the hot key Ctrl+B)
  A breakpoint dialog box is displayed (Fig 5-3)
- Designate the descriptive items of the breakpoint
  Set Space, Location items
  Set Content item and Read/Write check box if Space is the data space
  Set External signals if necessary
- Press the Add button to add this breakpoint to the Breakpoints list box.
- Press the OK button to confirm

Note If the total count of the effective breakpoints is less than 3, the newly added one will take effect automatically after it has been added.
If the Breakpoints list box is full, with 20 breakpoints, the Add button is disabled and no more breakpoints can be added.



Space: (● Code) ○ Data
Location: 20
Content:
☐ read ☐ write
Externals: xxxx
Breakpoints:

```
+ {C,C:\HT-IDE\SAMPLE\T1.ASM!.11,xxxxxxxx,xxxx}
+ {D/R,C:\HT-IDE\SAMPLE\T1.ASM!.15,2,xxxx}
+ {C,c:\ht-ide\sample\t2.asm!.10,xxxxxxxx,xxxx}
- {C,20,xxxxxxxx,xxxx}
```

Fig 5-3

→ **To Delete a Breakpoint**

- Choose the Breakpoint command from the Debug menu or press the hot key Ctrl+B
  A breakpoint dialog box is displayed (Fig 5-3)
- Choose or highlight the breakpoint to be deleted from the Breakpoints list box
- Press the Delete button to delete this breakpoint from the Breakpoints list box
- Press the OK button to confirm

→ **To Delete all Breakpoints**

- Choose the Breakpoint command from the Debug menu or press the hot key Ctrl+B
  A breakpoint dialog box is displayed (Fig 5-3)
- Choose the Clear All button to delete all breakpoints from the Breakpoints list box
- Press the OK button to confirm
- You can also click the Clear All Breakpoint button on the toolbar to accomplish this task.

→ **To Enable (Disable) a Breakpoint**

- Choose the Breakpoint command from the Debug menu or press the hot key Ctrl+B
  A breakpoint dialog box is displayed (Fig 5-3)
- Choose the disabled (enabled) breakpoint from the Breakpoints list box
- Press the Enable (Disable) button, to enable or disable this breakpoint
- Press the OK button to confirm

Trace the Application Program

The HT-IDE3000 provides a powerful trace mechanism which records the execution processes and all relative information when the HT-IDE3000 is emulating the application program. The trace mechanism provides qualifiers to filter specified instructions and trigger conditions in order to stop the trace recording. It also provides a method to record a specified count of the trace records before or after a trigger point.

Note When the HT-IDE3000 starts emulating (refer to the section on Emulation of the Application Programs), the trace mechanism will begin to record the executing instructions and relative information automatically, but not vice versa.

Initiating the Trace Mechanism

The basic requirement for initializing the trace mechanism is to set the Trace Mode with or without Qualify. The Trace Mode defines the trace scope of the application program and Qualify defines the filter conditions of the trace recording.

The available Trace Modes are

- Normal
  Sets the trace scope to all application programs and is the default mode.
- Trace Main
  Sets the trace scope to all application programs except the interrupt service routine programs.
- Trace INT
  Sets the trace scope to all interrupt service routine programs.

According to Qualify, the trace mechanism decides which instructions and what corresponding information should be recorded in the trace buffer during the emulation process. The rule is that an instruction will be recorded if its information and status satisfy one of the enabled qualifiers. The format of Qualify is the same as that of the breakpoint. If all program steps are required to be recorded, then No Qualify is needed (do not set the Qualify). The default is No Qualify.

In contrast to the Trace Mode and Qualify, which specify the conditions of trace recording, both the Trigger Mode and Forward Rate specify the conditions to stop the trace recording.

The Trigger Mode specifies the kind of trigger point, and is a standard used to determine the location of the stop trace point. The Forward Rate specifies the trace scope between the trigger point and the stop trace point.

The available Trigger Modes are:

- No Trigger
  No stopping of the trace recording condition. This is the default case.
- Trigger at Condition A
  The trigger point is at condition A.
- Trigger at Condition B
  The trigger point is at condition B.
- Trigger at Condition A or B
  The trigger point is at either condition A or condition B.
- Trigger at Condition B after A
  The trigger point is at condition B after condition A has occurred.
- Trigger when meeting condition A for k times
  The trigger point is when condition A has occurred k times.
- Trigger at Condition B after meeting A for k times
  The trigger point is at condition B after condition A has occurred for k times.

Condition A and Condition B specify the trigger conditions. The format of condition A or B is the same as that of the breakpoint.

The Loop Count specifies the number of occurrences of the specified condition A. It is used only when the Trigger Mode is from one of the last two modes in the above list.

The Forward Rate specifies the approximate rate of the trace recording information between the trigger point and stop trace point in the whole trace buffer. The trigger point divides the trace buffer into two parts, before and after trigger point. The forward rate is used to limit the trace recording scope after the trigger point. The percentage is adjustable between 0 and 100%.

Note It is not necessary for the trace recording scope to be equal to the forward rate. If a breakpoint is met before reaching the trace recording scope or a trace stop command (refer to: Stopping the trace mechanism) is issued, the trace recording will be stopped.

A Qualify list box records and displays all qualifiers used by the Trace Mode. Up to 20 qualifiers can be added into the list box and and up to 6 qualifiers can be effective. A Qualifier can be disabled or deleted from the list box. The format of each qualifier in the Qualify list box has the same format as the breakpoint in the Breakpoints list box (refer to the section on Breakpoints, Format of breakpoints list box)

Stopping the Trace Mechanism

There are 3 methods to stop the trace recording mechanism:

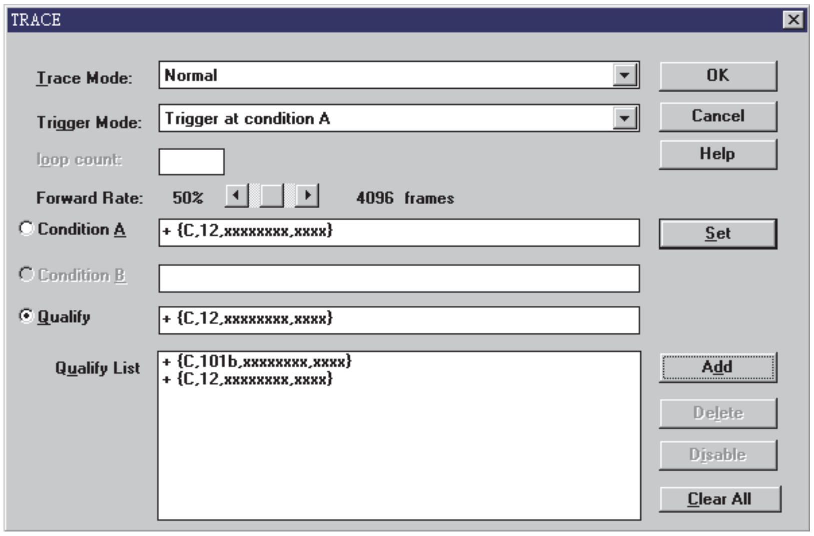
- Set the trigger point (Trigger Mode) and Forward Rate as shown above
- Set breakpoints to stop the the emulation and the trace recording.
- Issue a Trace Stop command from the Debug menu (Fig 5-2) to stop the trace recording.

Fig 5-4 lists all the requirements to use the trace mechanism. This is the result of the Trace command from the Debug menu.

Trace Start/Stop Setup

→ **To Set the Trace Mode**

- Choose the Trace command from the Debug Menu
  A Trace dialog box is displayed as in Fig 5-4.
- Choose a trace mode from the Trace Mode pull-down list box
- Press the OK button



Trace Mode: Normal
Trigger Mode: Trigger at condition A
loop count:
Forward Rate: 50% 4096 frames
Condition A + {C,12,xxxxxxxx,xxxx}
Condition B
Qualify + {C,12,xxxxxxxx,xxxx}
Qualify List
+ {C,101b,xxxxxxxx,xxxx}
+ {C,12,xxxxxxxx,xxxx}

Fig 5-4

→ **To Set the Trigger Mode**

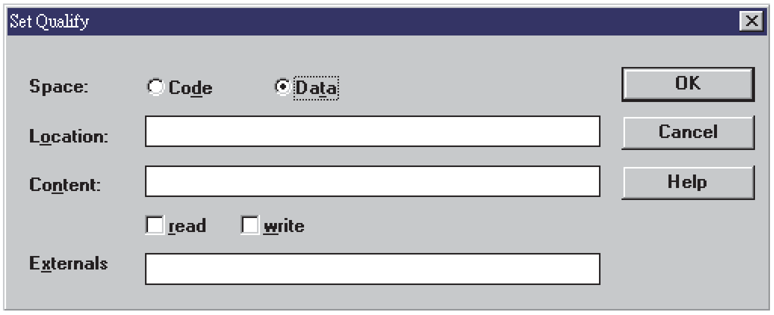
- Choose the Trace command from the Debug Menu
  A Trace dialog box is displayed as in Fig 5-4.
- Choose a trigger mode from the Trigger Mode pull-down list box
- press the OK button

→ **To Change the Forward Rate**

- Choose the Trace command from the Debug Menu
  A Trace dialog box is displayed as in Fig 5-4
- Use the Forward Rate scroll bar to specify the desired rate
- Press the OK button

→ **To Setup the Condition A/Condition B**

- Choose the Trace command of the Debug Menu
  A Trace dialog box is displayed as Fig 5-4.
- Press Condition A/Condition B radio button
- Press the Set Condition button
  A Set Qualify dialog box is displayed as in Fig 5-5.
- Enter the conditional information
- Press the OK button to close the Set Condition dialog box
- Press the OK button to close the Trace dialog box



Space: Code [✓ Data]

Fig 5-5

→ **To Add a Trace Qualify Condition**

- Choose the Trace command from the Debug Menu
  A Trace dialog box is displayed as in Fig 5-4.
- Press the Qualify radio button
- Press the Set Qualify button
  A Set Qualify dialog box is displayed as in Fig 5-5.
- Enter the qualifier information
- Press the OK button to close the Set Qualify dialog box
- Press the Add button to add the qualifiers into the Qualify list box below
- Press the OK button to close the Trace dialog box

→ **To Delete a Trace Qualify Condition**

- Choose the Trace command from the Debug Menu
  A Trace dialog box is displayed as in Fig 5-4.
- Choose the qualify line to be deleted from the Qualify list box
- Press the Delete button
- Press the OK button to confirm

→ **To Delete All Qualify Conditions**

- Choose the Trace command from the Debug Menu
  A Trace dialog box is displayed as in Fig 5-4.
- Press the Clear All button
- Press the OK button to confirm

Note If there is no qualifier, all instructions are qualified by default.

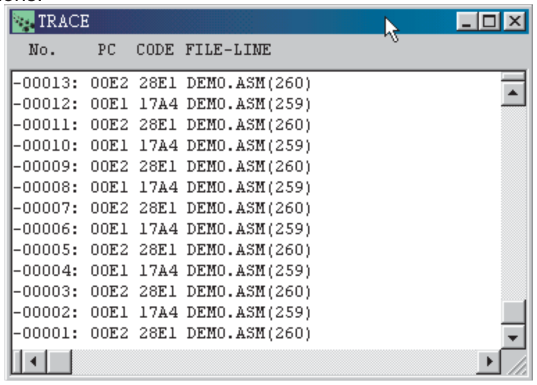
→ **To Enable (Disable) a Trace Qualify Condition**

- Choose the Trace command from the Debug Menu
  A Trace dialog box is displayed as in Fig 5-4
- Choose the disabled (enabled) qualifier line to be enabled (disabled) from the Qualify list box
- Press the Enable (disable) button
- Press the OK button to confirm

Note At most, 6 trace qualifications can be enabled at the same time.

Trace Record Format

Once the trace qualify and trigger conditions have been setup, those instructions which satisfy the qualify conditions will be recorded in the trace buffer. The Trace List command of the Window menu provides the functions to view and check the trace record information, used for debugging the program. The trace record fields may not all be displayed on the screen except for the sequence number. These fields are dependent upon the settings in the Debug sub-menu from the Options menu. The text enclosed by the parentheses are the headings shown in the Trace List command of the Window menu. Fig 5-6 and Fig 5-7 illustrate the contents of the trace list under the different debug options.



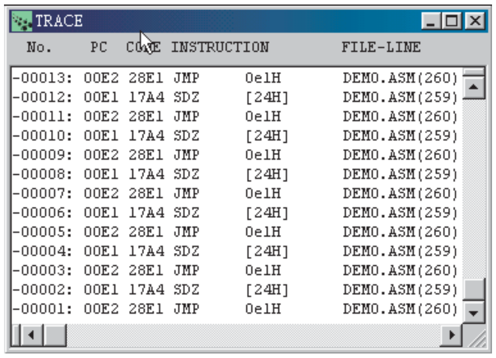
```
 No.     PC  CODE FILE-LINE
-00013: 00E2 28E1 DEMO.ASM(260)
-00012: 00E1 17A4 DEMO.ASM(259)
-00011: 00E2 28E1 DEMO.ASM(260)
-00010: 00E1 17A4 DEMO.ASM(259)
-00009: 00E2 28E1 DEMO.ASM(260)
-00008: 00E1 17A4 DEMO.ASM(259)
-00007: 00E2 28E1 DEMO.ASM(260)
-00006: 00E1 17A4 DEMO.ASM(259)
-00005: 00E2 28E1 DEMO.ASM(260)
-00004: 00E1 17A4 DEMO.ASM(259)
-00003: 00E2 28E1 DEMO.ASM(260)
-00002: 00E1 17A4 DEMO.ASM(259)
-00001: 00E2 28E1 DEMO.ASM(260)
```

Fig 5-6

- Sequence number (No)
  For any of the trigger modes, the sequence number of a trigger point is +0. The trace records before and after the trigger point are numbered using negative and positive line numbers respectively. If all the fields of the Trace Record Fields (in the Debug Option of Option menu) are selected, the result is as shown in Fig 5-7. If No trigger mode is selected or the trigger point has not yet occurred, the sequence number starts from -00001 and decreases 1 sequentially for the trace records (Fig 5-6).
- Program count (PC)
  The program count of the instruction in this trace record.
- Machine code (CODE)
  The machine code of this instruction.
- Disassembled instruction (INSTRUCTION)
  The disassembled mnemonic instruction is disassembled using an HT-IDE3000 utility.

- Execution data (DAT)
  The data content to be executed (read/write).
- External signal status (0 1 2 3)
  The external signal 0~3 denotes the external signal ET0~ET3 respectively.
- Source file name with a line number (FILE-LINE)
  The source file name and the line number of this instruction.
- Source file (SOURCE)
  The source line statement (including symbols).

All the above fields are optional except the sequence number which is always displayed.



```
 No.     PC   CODE INSTRUCTION          FILE-LINE
-00013: 00E2 28E1 JMP      0e1H         DEMO.ASM(260)
-00012: 00E1 17A4 SDZ      [24H]        DEMO.ASM(259)
-00011: 00E2 28E1 JMP      0e1H         DEMO.ASM(260)
-00010: 00E1 17A4 SDZ      [24H]        DEMO.ASM(259)
-00009: 00E2 28E1 JMP      0e1H         DEMO.ASM(260)
-00008: 00E1 17A4 SDZ      [24H]        DEMO.ASM(259)
-00007: 00E2 28E1 JMP      0e1H         DEMO.ASM(260)
-00006: 00E1 17A4 SDZ      [24H]        DEMO.ASM(259)
-00005: 00E2 28E1 JMP      0e1H         DEMO.ASM(260)
-00004: 00E1 17A4 SDZ      [24H]        DEMO.ASM(259)
-00003: 00E2 28E1 JMP      0e1H         DEMO.ASM(260)
-00002: 00E1 17A4 SDZ      [24H]        DEMO.ASM(259)
-00001: 00E2 28E1 JMP      0e1H         DEMO.ASM(260)
```

Fig 5-7

Note To set the trace record fields use the Debug command of the Options menu.
To view the trace record fields use Trace List command of the Window menu.

→ **Clear the Trace Buffer**

The trace buffer can be cleared by issuing the Reset Trace command. Hereafter, the trace information will be saved from the beginning of the trace buffer. Note that both the Reset command and the Power-On Reset command also clear the trace buffer.

Debugger Command Mode

In addition to the windows based debugging mode, the HT-IDE3000 provides an alternative debugging mode, named the Command Mode. Under this mode, the user, in addition to obtaining the same functions as the menu-driven windows based debugging mode, also has access to additional debugging functions. These added functions include the ability to save the debugging history into a log file in order to execute these debugging commands automatically again as well as the ability to execute the previous debugging command without rewriting the command.

Enter/Quit the Command Mode

→ **Enter to Command Mode**

From the Debug Menu of the HT-IDE3000 select ″Command Mode″ command. When the command mode has been entered a new screen will appear where commands can be entered after the ″HT8>″ prompt on the second line. (Fig 5-8)

→ **Command Mode Window**

- The Command Mode Title bar shows the name of the present project file.
- Any command can be entered after the ″HT8>″ prompt on the command line.
- When the command is entered the full command syntax will be displayed on the bottom status bar.
- After the command has been entered at the ″HT8> xxxx″ prompt, the next line will display the result of the command execution. (Fig 5-9)
  Another ″HT8>″ prompt will then be displayed where another command can be entered.

→ **Quit from the Command Mode**

To quit from the Command Mode the normal windows exit method can be used or a Q[uit] command can be entered at the command prompt.

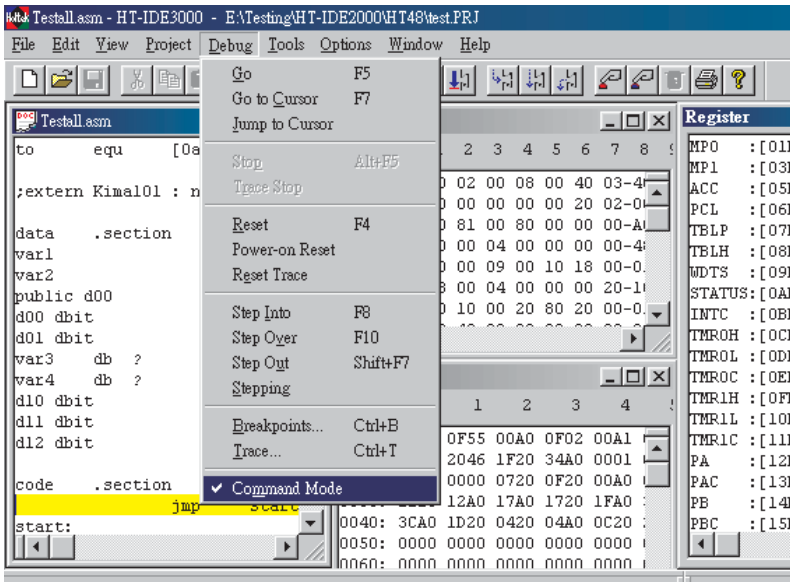
Functions Supported by the Command Mode

The following table shows the complete list of debugging statements supported by the Command Mode

Command	Function Description	Command Syntax
!	Execute a previous command	!dd
;	Comment	;
BP	Breakpoint Commands	BP {-C|-D|-E|-L} [list|*]. → list=11 12...
BP	Breakpoint Set	BP S[,RW], Location [,Data] [,Ext Sig]
DB	Dump Program Memory	DB [bank.address [,range]]
DR	Dump Data Memory	DR [bank.]address[,range]
FA	Fill string	FA {bank.address|symbol} list. →list=11 12...
FB	Fill bytes	FB {bank.address|symbol} list → list=11 12...
GO	Free run or run to the specified address	GO [address]

Command	Function Description	Command Syntax
JP	Jump to specified address directly	JP address
H	Help	H
HIS	History of commands	HIS
LF	Load and execute a log file	LF [-V] [LogFileName]
LP	Load project	LP ProjectName
Q	Quit	Q
R	Reset	R
POR	Power on reset	POR
S	Single Step (Into/Over/Out)	S [-I|-V|-O] default option: ″-I″
TR	Trace list	TR [-L] [length]
W	Open/Write/Close a Log file	W {-S|-C} [LogFileName]

In the debugging command syntax, if large brackets exist, this indicates that a parameter must be inserted otherwise an error will occur. Parameters are separated by a | symbol.



```
to       equ     [0a

;extern Kimal01 : n

data     .section
var1
var2
public d00
d00 dbit
d01 dbit
var3     db   ?
var4     db   ?
d10 dbit
d11 dbit
d12 dbit

code     .section
              jmp      start
start:
```

```
 2  3  4  5  6  7  8
 02 00 08 00 40 03-4
 00 00 00 00 20 02-0
 81 00 80 00 00 00-A
 00 04 00 00 00 00-4
 00 09 00 10 18 00-0
 04 00 00 00 20-1
 10 00 20 80 20 00-0
```

```
      1    2    3    4
 0F55 00A0 0F02 00A1
 2046 1F20 34A0 0001
 0000 0720 0F20 00A0
 12A0 17A0 1720 1FA0
0040: 3CA0 1D20 0420 04A0 0C20
0050: 0000 0000 0000 0000 0000
0060: 0000 0000 0000 0000 0000
```

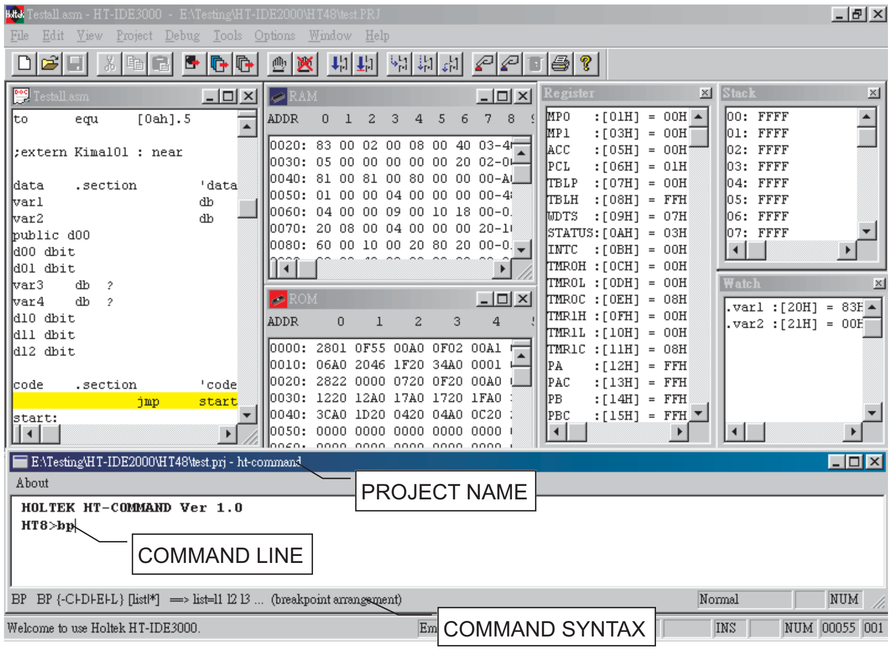
```
MP0    :[01
MP1    :[03
ACC    :[05
PCL    :[06
TBLP   :[07
TBLH   :[08
WDTS   :[09
STATUS:[0A
INTC   :[0B
TMR0H  :[0C
TMR0L  :[0D
TMR0C  :[0E
TMR1H  :[0F
TMR1L  :[10
TMR1C  :[11
PA     :[12
PAC    :[13
PB     :[14
PBC    :[15
```

Fig 5-8

→ **Breakpoint Commands**

There are two breakpoint commands, their command syntax and function is as follows:

- BP – Breakpoint Clear/Enable/Disable/List
  Syntax: BP [-C | -D | -E | -L] [list | *]
  Parameter -C is the clear breakpoint parameter. This will delete the indicated breakpoint or clear all the breakpoints shown in the Breakpoint Box. Within the list there can be from 1~20 numbers


Testall.asm - HT-IDE3000 - E:\Testing\HT-IDE2000\HT48\test.PRJ
File Edit View Project Debug Tools Options Window Help
Testall.asm
RAM
ROM
Register
Stack
Watch
E:\Testing\HT-IDE2000\HT48\test.prj - ht-command
About
PROJECT NAME
COMMAND LINE
COMMAND SYNTAX
Normal
NUM
INS
00055
001

```
to       equ      [0ah].5

;extern Kimal01 : near

data    .section         'data
var1                     db
var2                     db
public d00
d00 dbit
d01 dbit
var3     db   ?
var4     db   ?
d10 dbit
d11 dbit
d12 dbit

code    .section         'code
                 jmp      start
start:
```

```
ADDR   0  1  2  3  4  5  6  7  8
0020: 83 00 02 00 08 00 40 03-4
0030: 05 00 00 00 00 00 20 02-0
0040: 81 00 81 00 80 00 00 00-A
0050: 01 00 00 04 00 00 00 00-4
0060: 04 00 00 09 00 10 18 00-0.
0070: 20 08 00 04 00 00 00 20-1
0080: 60 00 10 00 20 80 20 00-0.
```

```
ADDR    0    1    2    3    4
0000: 2801 0F55 00A0 0F02 00A1
0010: 06A0 2046 1F20 34A0 0001
0020: 2822 0000 0720 0F20 00A0
0030: 1220 12A0 17A0 1720 1FA0
0040: 3CA0 1D20 0420 04A0 0C20
0050: 0000 0000 0000 0000 0000
```

```
MP0    :[01H] = 00H
MP1    :[03H] = 00H
ACC    :[05H] = 00H
PCL    :[06H] = 01H
TBLP   :[07H] = 00H
TBLH   :[08H] = FFH
WDTS   :[09H] = 07H
STATUS:[0AH] = 03H
INTC   :[0BH] = 00H
TMR0H  :[0CH] = 00H
TMR0L  :[0DH] = 00H
TMR0C  :[0EH] = 08H
TMR1H  :[0FH] = 00H
TMR1L  :[10H] = 00H
TMR1C  :[11H] = 08H
PA     :[12H] = FFH
PAC    :[13H] = FFH
PB     :[14H] = FFH
PBC    :[15H] = FFH
```

```
00: FFFF
01: FFFF
02: FFFF
03: FFFF
04: FFFF
05: FFFF
06: FFFF
07: FFFF
```

```
.var1 :[20H] = 83H
.var2 :[21H] = 00H
```

```
HOLTEK HT-COMMAND Ver 1.0
HT8>bp
```

```
BP  BP {-C|-D|-E|-L} [list|*]  ==> list=l1 l2 l3 ...  (breakpoint arrangement)
Welcome to use Holtek HT-IDE3000.
```

Fig 5-9

which represent the breakpoints already setup. This means that more than one can be selected. For example, the three numbers 1 3 8 each separated by a space, indicates that the 1st,, 3rd and 8th, breakpoints will be cleared. This has the same operation as the Delete function within the Debug/Breakpoint window. The star symbol * means that all the breakpoints already setup will be cleared. It has the same operation as the Clear All function within the Debug/Breakpoint window.

Parameter -D will change all the indicated breakpoints to non-active, however the breakpoints will still remain shown in the Breakpoint Box. This command is the same as the Disable function within the Debug/Breakpoint window. The star * has the same operation as that described above.

Parameter -E will change all the indicated breakpoints to active. , This command is the same as the Enable function within the Debug/Breakpoint window. The star * has the same operation as that described above.

Parameter -L will display all the presently setup breakpoints in the window, the format is consistent with the contents of the Debug/Breakpoint window, where the first column shows the breakpoint number. The user can refer to this breakpoint number to setup the required numbers in the BP -C, BP -D, BP -E statement.

Note

1. BP-L this parameter does not require list |
2. The HT-IDE3000 can only have a maximum of 3 breakpoints active at the same time.

If no C, D, E or L parameters are given then the Breakpoint command will be of the following type:

- BP – Breakpoint Set
  Syntax: BP S [,RW] ,Location [,Data] [,Ext Sig]
  The parameter within the brackets is optional however under certain conditions it must be specified.
  S denotes a Space, where a choice can be made between C or D. The letter C indicates that the breakpoint is set in Program Code Memory, while D indicates that the breakpoint is set in the Data Memory (RAM)
  If D is chosen to replace S then the read/write option [,RW] must also be specified. The user can choose from R or W or RW. This is because if the breakpoints are set in the Data Memory then the choice exists for the breakpoint to be activated on either a read, a write or both a read and write. If C is chosen to replace S, which indicates program code, then it is not necessary to setup RW.
  The ″Location″ parameter sets the position of the breakpoint, its format is:
  [SourceFileName!].LineNumber or [SourceFileName!].SymbolName
  If no SourceFileName is specified then the already opened source file will be taken as the default.
  If D is chosen to replace S then the ″Data″ parameter must be setup. The breakpoint is setup at the specified location in the Data Memory and will initiate a break when a read or write with the specified data occurs.
  Ext Sig is a parameter that can be chosen, for its use consult the HT-IDE3000 User′s Guide

→ **Comment Command**

- Syntax: ; comment string
  This command is provided to give an explanation to the Log file. Any characters found after the ; will have no functional effect.

→ **Dump Command**

- Syntax: DB bank.address ,range
  DB range
  DB
  This command will display in the window, the contents of the specified program memory area. This area is specified by indicating the address, as well as the range and bank. The data is in hex format. If the address is specified but the bank number is not specified then the bank number will be taken as that of the current bank. If neither address nor bank number is specified the bank number will be taken as that of the current bank number and the address will be taken as that of the present Program Counter. If the range is not specified then the range value will be taken as 16 words. The range is not allowed to exceed one bank (2000h). An example of this statement would be 1.0f00 which would indicate that the bank number is 1 and the address value is 0f00h.
- Syntax: DR bank.address ,range
  DR address
  This command will display in the window, the contents of the specified area of Data Memory. This data area is specified by its address, range and bank. The data is displayed in hex format. If the range is not specified then it will be set to 16 bytes. The range is not allowed to exceed one bank (100h) and the bank address is expressed in hex format.

→ **Fill Command**

This command changes the contents of the Data Memory

- Syntax: FB {bank.address | symbol}, list
  Will write the bytes specified in the list into a Data Memory area at the specified bank number and at the specified start address or symbol.
  Either a bank.address or symbol name can be used. Also the list can be more than one byte, however at least one blank must be used as a delimiter. All values are specified in hex format. The list range cannot cross over a bank boundary.
- Syntax: FA {bank.address | symbol}, string
  FA has the same function as FB except that the data is supplied in ASCII
  the user can chose one of the following symbol formats:
  .var
  filename!.var
  path\filename!.var

Note If path contains spaces then the name must be included in quotation marks otherwise an error condition will occur.

- Example: FA ″d:\tmp\test cmd\test1.asm!.count″, ″test1″

→ **Go/Jump Commands**

- Syntax: GO [address]
  If an address is specified the program will free run until the specified address is encountered. If the address is not specified the program will run to the end or until an active breakpoint is encountered.
- Syntax: JP address
  Will force a direct jump to the specified address. Note that an address must be specified.

→ **Help Command**

- Syntax: H
  This command will list in the window all of the debugging commands, their syntax and description.

→ **History Commands**

- Syntax: HIS
  This command will display in the window the last 20 commands, not including the HIS command, that were executed. At the same time the first column will display the command sequence numbers in succession.
- Syntax: !dd
  dd is the displayed command sequence number in the above mentioned HIS command. This command will execute the previously executed command again. By writing the sequence number and adding a ″!″ the same command can be executed again reducing the need to re-input commands and parameters. If no command sequence number is indicated the last command will be executed.

→ **Load Commands**

- Syntax: LF [-V] [LogFileName]
  This command will load and execute all the Debugging Commands in the Log File, specified by the LogFileName

If no LogFileName is specified, then the same name as the current Project File name will be taken as the filename.
Parameter -V indicates that the command line and the execution result should be displayed in the window.
If LF has no -V option, then the result record will be placed in a logfile of the same name with a .res file extension name.
Log file is created using the W command. The contents can be modified by using the File and Edit function within the HT-IDE3000.
However these contents must contain the correct Debugging Commands otherwise an error condition will occur, the execution will stop and return to the prompt sign

Note
1. If spaces are included in the LogFileName then the name must be included within quotation marks otherwise an error condition will occur.
2. The logfile cannot contain the LF, W or Q commands.

→ **Quit Command**
- Syntax: Q
  This command will end the Command Mode and return to the present window.

Note
1. This command has no effect in the Command Log file.
2. After quitting from the command mode all the files opened by ″LF″ and ″W -S″ will be closed and the execution of commands will stop.

→ **Reset Commands**
- Syntax: R
  The function of this command is the same as the Debug/Reset command
- Syntax: POR
  The function of this command is the same as the Debug/Power-On Reset command

→ **Step Commands**
There are 3 kinds of Single Step commands; which after execution will display the contents of the PC, STATUS and ACC
- Syntax: S {-I | -O | -V }
  Single Step Command.
  -I is Step Into, which has the same function as Debug/Step Into
  -V is Step Over, which has the same function as Debug/Step Over
  -O is Step Out, which has the same function as Debug/Step Out
  If no option has been setup the default condition will be ″S -V″

→ **Trace Command**
- Syntax: TR [-L] [length]
  The trace command will display the contents of the trace buffer in the window. Parameter -L indicates that all records will be displayed, which include Sequence number, Program count, Machine code, Disassembled instructions, Execution data, External signal, source file name with line number and source file.

If the -L parameter is not supplied, then the default condition will only display Sequence number, Program count, Machine code, Disassembled instructions and source file name with line number. The parameter ″length″ indicates the length of the displayed trace. The trace display will begin from sequence number 0 and trace back with the specified length. The length can also specify the length to trace forward. To do this the forward rate must first be setup in the system. The default length value is 5.

The Trace mode, qualify conditions and forward rate etc. parameters are directly setup within the HT-IDE3000 window, the command mode does not support these functions.

→ **Write Command**

- Syntax: W [-S | -C] [LogFileName]
  This command will write the debugging commands and its corresponding results into the Log File. The Log File will terminate whenever a W -C or Q command is encountered or if the command mode is terminated.
  -S will create a Log File in which all following commands and results will be written
  -C will close the previously created Log File, no further commands will be written into the Log File
  If the indicated Log File is already saved, then the system will require confirmation before overwriting and continuing with the next step. It is not necessary to add a file extension name.
  If the Log File name does not exist, then the file name will take the same name as the project with an added .CMD file extension name.

Note
1. If spaces are included in the LogFileName then the name must be included within quotation marks otherwise an error condition will occur.
2. After executing the W -S command the LF or W -S command cannot be executed.

Log File Format

The Log File is a text file that can be modified by any text editor including the editor contained within the HT-IDE3000. This editor can be accessed by selecting Edit from the main menu. Its format is that every Debugging command will occupy one line.

command: W -S LogFileName will clear the contents of the Log File, and after write the new commands and results.

If the command string, has been created by the ″W -S″command then note that prompt signs will also be written into the Log File. However, the next time it is read by the debugger command these previously written prompt signs will be ignored automatically. For the case where the command strings are generated using an editor, note that it is not necessary to enter any prompt signs into the Log File.

If the Log File has been created by the ″W -S″command then before each command execution result a ″;″ will be automatically inserted making the execution result into an annotated note.
In this way when the next upload is executed only the command string will be executed, the result string will be ignored.

HT-COMMAND Error Messages

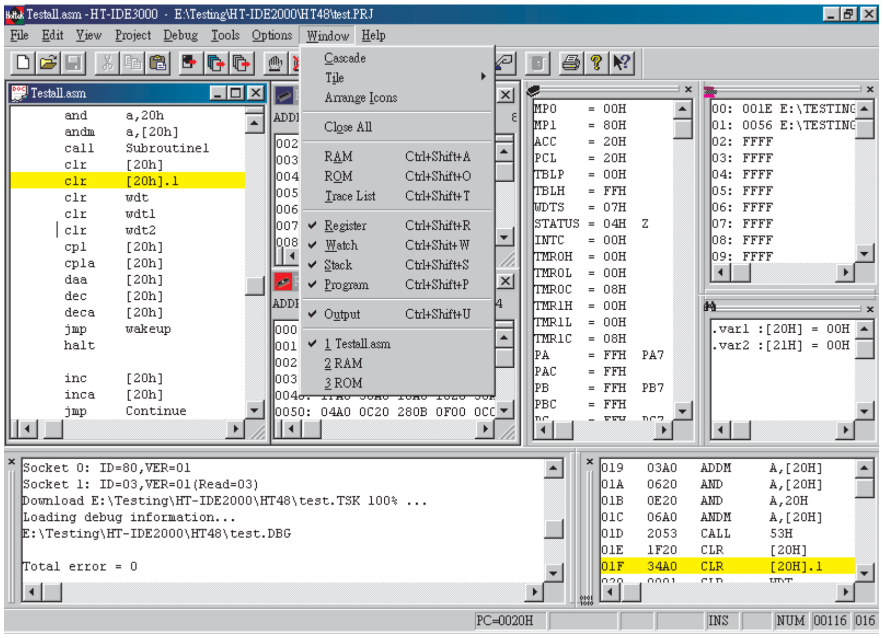
Error Message	Description
Invalid Command	The command just entered is illegal
Can not find HT-IDE	The present environment is not the HT-IDE3000
Syntax error	The input syntax is incorrect
No project for debug	No project file has been opened in the HT-IDE3000
ROM bank out of range	The Program Memory dump has exceeded its range
RAM bank Out of range	The Data Memory dump has exceeded its range
Can not run xxx command in emulation mode	The xxx command cannot be executed
Can not run xxx command in load file mode	The xxx command cannot be executed
Can not run xxx command in write file mode	The xxx command cannot be executed
Unterminated string	The character string definition requires balanced quotes
No Command in history buffer	History buffer empty
Open xxx log file error	Cannot open the log file
Close xxx log file error	Cannot close the log file
Read xxx log file error	Cannot read the log file
Write xxx log file error	Cannot write to the log file
Not in emulation status	Before executing this command first enter emu-lation mode
Sources have been modified, please rebuild	The original source file has been modified requir-ing the files to be rebuilt
Stop by user	User has stopped execution
Get PC failed	Reading the value of the Program Counter has failed
Stack overflow	The stack has exceeded its capacity
No debug info	The setup breakpoints have no debug informa-tion
Cannot find the symbol	The indicated symbol cannot be found
Cannot find the register	The indicated register cannot be found

Chapter 6

Menu – Window

6

The HT-IDE3000 provides various kinds of windows which assist the user to emulate or simulate application programs. These windows (as shown in Fig 6-1) include program Data Memory (RAM), program code memory (ROM), Trace List, Register, Watch , Stack, Program, Output, etc.



```
        and     a,20h
        andm    a,[20h]
        call    Subroutine1
        clr     [20h]
        clr     [20h].1
        clr     wdt
        clr     wdt1
        clr     wdt2
        cpl     [20h]
        cpla    [20h]
        daa     [20h]
        dec     [20h]
        deca    [20h]
        jmp     wakeup
        halt

        inc     [20h]
        inca    [20h]
        jmp     Continue
```

```
0050: 04A0 0C20 280B 0F00 0CC
```

```
MP0    = 00H
MP1    = 80H
ACC    = 20H
PCL    = 20H
TBLP   = 00H
TBLH   = FFH
WDTS   = 07H
STATUS = 04H  Z
INTC   = 00H
TMR0H  = 00H
TMR0L  = 00H
TMR0C  = 08H
TMR1H  = 00H
TMR1L  = 00H
TMR1C  = 08H
PA     = FFH  PA7
PAC    = FFH
PB     = FFH  PB7
PBC    = FFH
```

```
00: 001E E:\TESTING
01: 0056 E:\TESTING
02: FFFF
03: FFFF
04: FFFF
05: FFFF
06: FFFF
07: FFFF
08: FFFF
09: FFFF
```

```
.var1 :[20H] = 00H
.var2 :[21H] = 00H
```

```
Socket 0: ID=80,VER=01
Socket 1: ID=03,VER=01(Read=03)
Download E:\Testing\HT-IDE2000\HT48\test.TSK 100% ...
Loading debug information...
E:\Testing\HT-IDE2000\HT48\test.DBG

Total error = 0
```

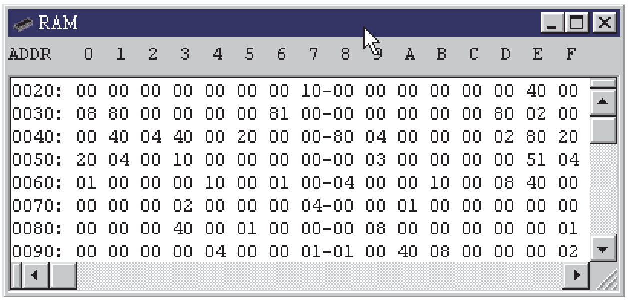
```
019   03A0   ADDM    A,[20H]
01A   0620   AND     A,[20H]
01B   0E20   AND     A,20H
01C   06A0   ANDM    A,[20H]
01D   2053   CALL    53H
01E   1F20   CLR     [20H]
01F   34A0   CLR     [20H].1
```

Fig 6-1

Window Menu Commands

- RAM

  The RAM window display the contents of the program Data Memory space as shown in Fig 6-2. The address spaces of the registers are not included in the RAM window because they are displayed in the register window. The contents of the RAM window can be modified directly for debugging purpose. The address displayed vertically is the base address while the horizontal single digit address is the offset. All the digits are displayed in hexadecimal format.

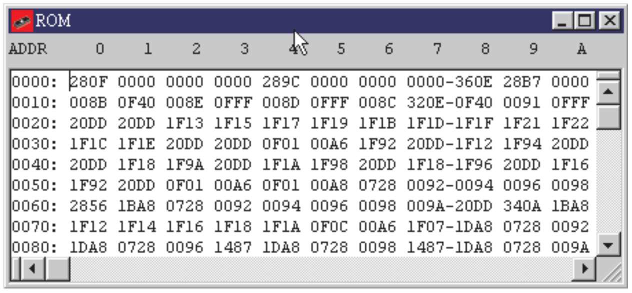


```
ADDR   0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0020: 00 00 00 00 00 00 00 10-00 00 00 00 00 00 40 00
0030: 08 80 00 00 00 00 81 00-00 00 00 00 00 80 02 00
0040: 00 40 04 40 00 20 00 00-80 04 00 00 00 02 80 20
0050: 20 04 00 10 00 00 00 00-00 03 00 00 00 00 51 04
0060: 01 00 00 00 10 00 01 00-04 00 00 10 00 08 40 00
0070: 00 00 00 02 00 00 00 04-00 00 01 00 00 00 00 00
0080: 00 00 00 40 00 01 00 00-00 08 00 00 00 00 00 01
0090: 00 00 00 00 04 00 00 01-01 00 40 08 00 00 00 02
```

Fig 6-2

- ROM

  The ROM window displays the contents of the program code memory space as shown in Fig 6-3. The ROM address range is from 0 to last address where the last address depends upon the MCU selected in the project. The horizontal and vertical scrollbars can be used to view any address in the ROM window. The contents in ROM window are displayed in hexadecimal format and cannot be modified.

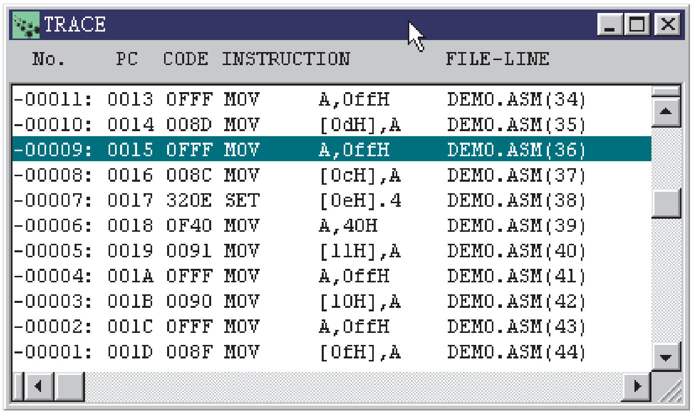


```
ADDR     0    1    2    3    4    5    6    7    8    9    A
0000: 280F 0000 0000 0000 289C 0000 0000 0000-360E 28B7 0000
0010: 008B 0F40 008E 0FFF 008D 0FFF 008C 320E-0F40 0091 0FFF
0020: 20DD 20DD 1F13 1F15 1F17 1F19 1F1B 1F1D-1F1F 1F21 1F22
0030: 1F1C 1F1E 20DD 20DD 0F01 00A6 1F92 20DD-1F12 1F94 20DD
0040: 20DD 1F18 1F9A 20DD 1F1A 1F98 20DD 1F18-1F96 20DD 1F16
0050: 1F92 20DD 0F01 00A6 0F01 00A8 0728 0092-0094 0096 0098
0060: 2856 1BA8 0728 0092 0094 0096 0098 009A-20DD 340A 1BA8
0070: 1F12 1F14 1F16 1F18 1F1A 0F0C 00A6 1F07-1DA8 0728 0092
0080: 1DA8 0728 0096 1487 1DA8 0728 0098 1487-1DA8 0728 009A
```

Fig 6-3

- Trace List

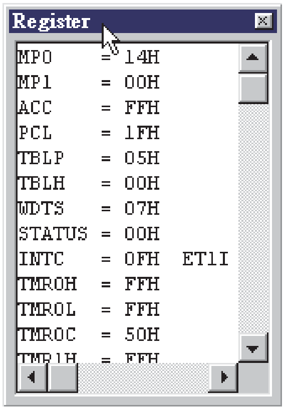
  The Trace List window displays the trace record information as shown in Fig 6-4. The contents of the trace record can be defined in the Debug command in the Options menu. Double click the trace record in the Trace List window will activate the source file window and the cursor will stop at the corresponding line.



```
TRACE
 No.     PC   CODE  INSTRUCTION          FILE-LINE
-00011: 0013 0FFF MOV      A,0ffH       DEMO.ASM(34)
-00010: 0014 008D MOV      [0dH],A      DEMO.ASM(35)
-00009: 0015 0FFF MOV      A,0ffH       DEMO.ASM(36)
-00008: 0016 008C MOV      [0cH],A      DEMO.ASM(37)
-00007: 0017 320E SET      [0eH].4      DEMO.ASM(38)
-00006: 0018 0F40 MOV      A,40H        DEMO.ASM(39)
-00005: 0019 0091 MOV      [11H],A      DEMO.ASM(40)
-00004: 001A 0FFF MOV      A,0ffH       DEMO.ASM(41)
-00003: 001B 0090 MOV      [10H],A      DEMO.ASM(42)
-00002: 001C 0FFF MOV      A,0ffH       DEMO.ASM(43)
-00001: 001D 008F MOV      [0fH],A      DEMO.ASM(44)
```

Fig 6-4

- Register
  The Register window displays all the registers defined in the MCU selected in the project. Fig 6-5 shows an example of the Register window of HT48C70-1. The contents of the Register window can be modified for debugging. Note that the Register window is dockable.



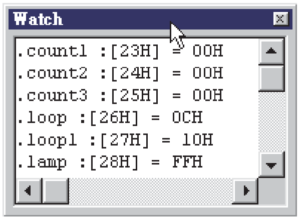
```
Register
MP0     = 14H
MP1     = 00H
ACC     = FFH
PCL     = 1FH
TBLP    = 05H
TBLH    = 00H
WDTS    = 07H
STATUS  = 00H
INTC    = 0FH  ET1I
TMR0H   = FFH
TMR0L   = FFH
TMR0C   = 50H
TMR1H   = FFH
```

Fig 6-5

- Watch
  The Watch window displays the memory addresses and contents of the specified symbols defined in the data sections, i.e., in the RAM space. The format of the symbol is:
  `[source_file_name!].symbol_name`

  The contents of the registers can also be displayed by first typing a period then typing the symbol name or register name and pressing the Enter key. The memory address and contents of the specified symbol or register will be displayed to the right of the symbol as shown in the following format:
  `:[address]=data contents`

Note that both address and data are displayed in hexadecimal format as shown in Fig 6-6. The symbol and their corresponding data will be saved by the HT-IDE3000 and displayed the next time the Watch window is opened. The symbols can be deleted from Watch window by pressing the delete key. Note that the Watch window is dockable.



```
.count1 :[23H] = 00H
.count2 :[24H] = 00H
.count3 :[25H] = 00H
.loop :[26H] = 0CH
.loop1 :[27H] = 10H
.lamp :[28H] = FFH
```

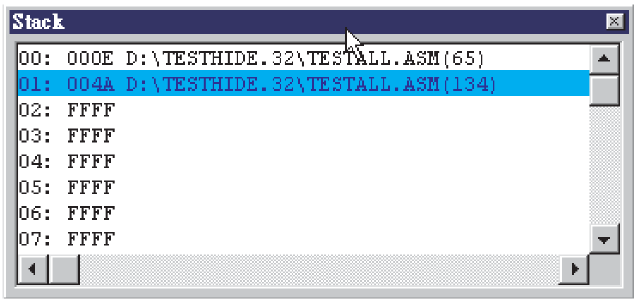
Fig 6-6

- Stack

The Stack window displays the contents of the stack buffer for the MCU selected in the current project. The maximum stack level is dependent upon the MCU selected. Fig 6-7 shows an example of the Stack window. The growth of the stack is numbered from 0. The number is increased by 1 for a push operation (CALL instruction or interrupt) and decreased by 1 for a pop operation (RET or RETI instructions). The top stack line is highlighted. E.g. The 01: shown in Fig 6-7 is the top stack line. While executing a RET or RETI instruction, the program line number specified in the top stack line (134 in this example) will be used as the next instruction line to be executed. Also, the line above the top stack line (00: in this example) will be used as the new top stack line. If there is no stack line anymore, no line in the Stack window will be highlighted. The format of the stack line is:

`Stack_level: program_counter source_file_name(line_number)`

where the stack_level is the level number of the stack, program_counter is the hexadecimal return address of the calling procedure or the program address of the interrupted instruction, source_file_name is the complete name of the source file containing the calling or interrupted instruction, and line_number is the decimal line number of the instruction after the call instruction or interrupted instruction in the source file.



```
00: 000E D:\TESTHIDE.32\TESTALL.ASM(65)
01: 004A D:\TESTHIDE.32\TESTALL.ASM(134)
02: FFFF
03: FFFF
04: FFFF
05: FFFF
06: FFFF
07: FFFF
```

Fig 6-7

- Program
  The Program window displays the program code memory or ROM in disassembly format. The address range is from 0 to last address where the last address depends upon the MCU selected in the project.
- Output
  The Output window shows the system messages from the HT-IDE3000 when the Build/Rebuild All commands are executing. By double clicking on the error message line, the window containing the source file will be displayed and the corresponding line containing the error highlighted.

Chapter 7

Simulation

7

The HT-IDE3000 provides a simulation mechanism for debugging application programs. The HT-IDE3000 simulator provides the same functions as the HT-ICE, but does not require the actual presence of the HT-ICE to function. In the HT-IDE3000, all the debugging and window functions for the HT-ICE are valid for the simulator. In addition, the simulator provides an interface for the input and output ports. Although the simulator provides many functions, some hardware characteristics of the MCU cannot be simulated. It is therefore recommended that emulation is carried out on the application program using the HT-ICE before manufacture of the masked IC.

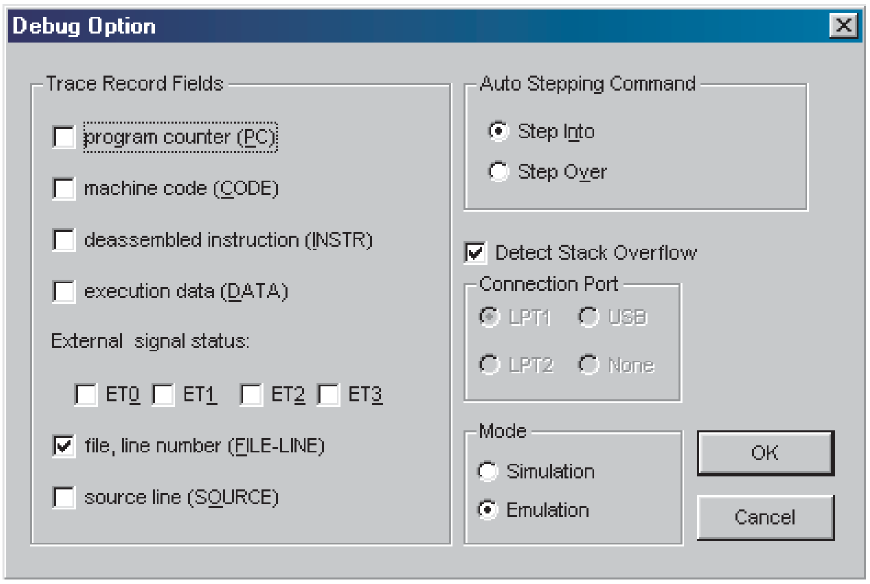
Some MCU series support emulation mode only and some support simulation mode.

Start the Simulation

Upon entering the HT-IDE3000, two situations may occur. The first is when a project has already been opened, and the second is when no project has been opened. In the first case, the working mode of the HT-IDE3000 depends upon the working mode of this project. In the latter case, the working mode will be in simulation. Even if the working mode of a project is in emulation, it can be changed by the user to be in simulation. In addition, the working mode of the HT-IDE3000 will be in simulation when the following situations occur.

- No connection between the HT-ICE and the host machine or when the connection fails.
- The HT-ICE is powered off.

The Debug command in the Option menu provides the function to set the working mode of the HT-IDE3000. Fig 7-1 displays the contents of the Debug command.



Debug Option

Trace Record Fields

- [ ] program counter (PC)
- [ ] machine code (CODE)
- [ ] deassembled instruction (INSTR)
- [ ] execution data (DATA)

External signal status:

- [ ] ET0 [ ] ET1 [ ] ET2 [ ] ET3

- [✓] file, line number (FILE-LINE)
- [ ] source line (SOURCE)

Auto Stepping Command

- (●) Step Into
- ( ) Step Over

[✓] Detect Stack Overflow

Connection Port

- (●) LPT1 ( ) USB
- ( ) LPT2 ( ) None

Mode

- ( ) Simulation
- (●) Emulation

OK Cancel

Fig 7-1

In addition to MCU simulator, Holtek provides a Virtual Peripheral Manager (VPM) which enable the user to directly drive and monitor the simulation of inputs and outputs on PC.

Part III gives more details on the VPM.

Chapter 8

OTP Programming

8

Introduction

Holtek's OTP writer was specifically developed for the range of Holtek OTP (One-Time Programmable) MCU devices, allowing users to easily and efficiently burn their programming code into the OTP devices. Its small and easy to manage size, ease of installation and easy-to-use special features are among the advantages of using this OTP writer. Furthermore, the OTP writer has been integrated on board in the recent version of HT-ICE Emulator, thus, making it more convenient for users during product development.

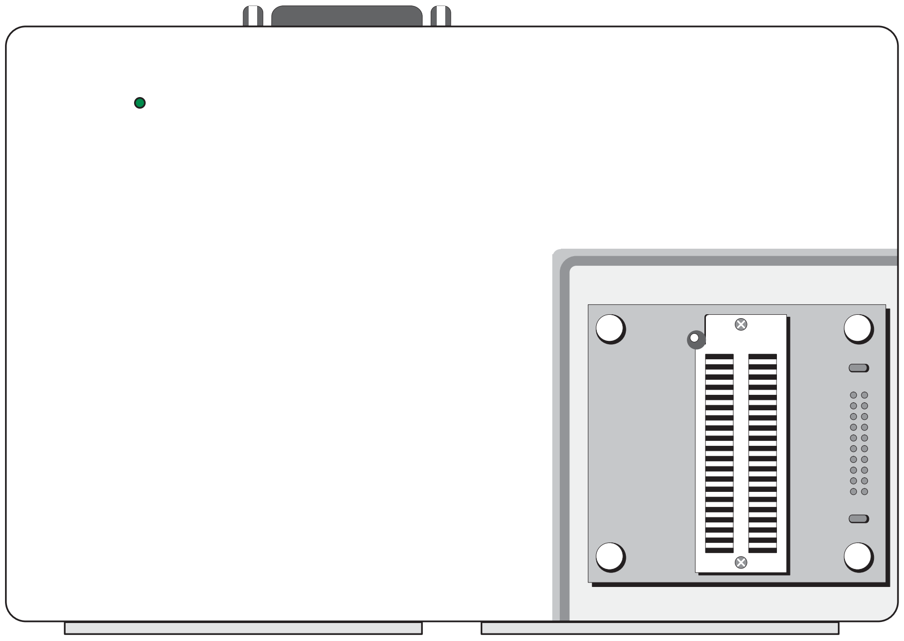


Fig 8-1

Installation

Since the OTP writer is built-in on the HT-ICE box, after the completion of HT-ICE installation, the OTP burning function is ready to be used within the HT-IDE3000 software with no further installation procedure needed. Refer to Chapter 1 — Overview and Installation.

Adapter Card

The HT-ICE emulator is shipped with a 40-pin TEXTOOL Adapter Card. If the device package format doesn′t match with this Adapter Card, user has to change the Adapter Card by himself. Refer to other Holtek Technical Document or visit our website for further information on selecting Adapter Cards.

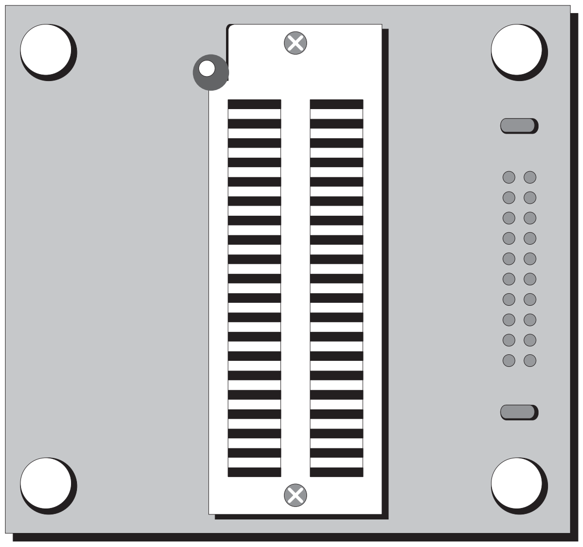


Fig 8-2

Programming an OTP Device with the HT-HandyWriter

→ **Run the HT-HandyWriter Software**
Run the HT-HandyWriter software under the Holtek Development System icon in the main Windows programs menu as shown in the Fig 8-3 below:

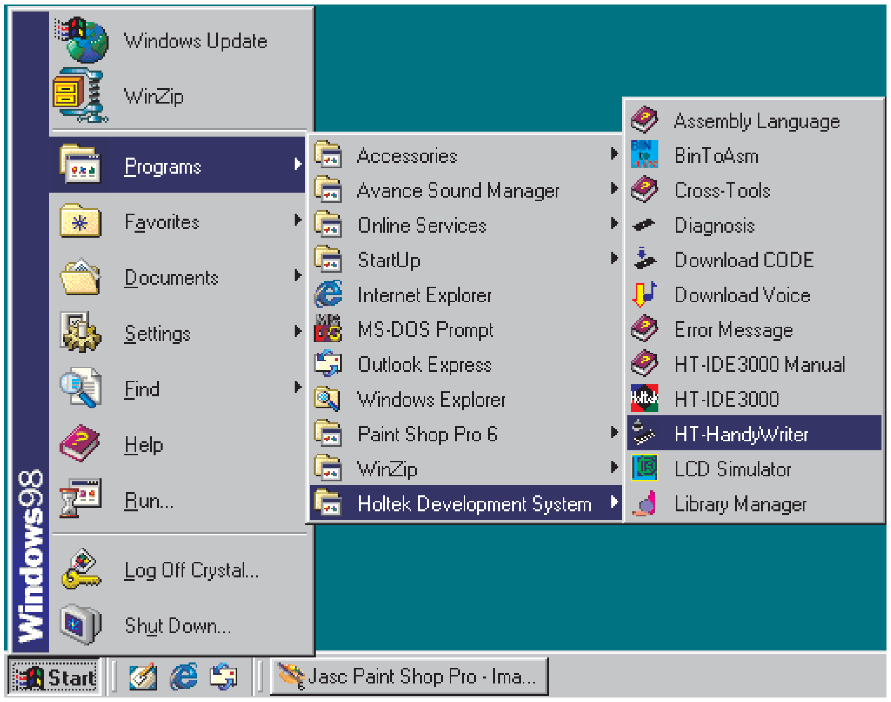
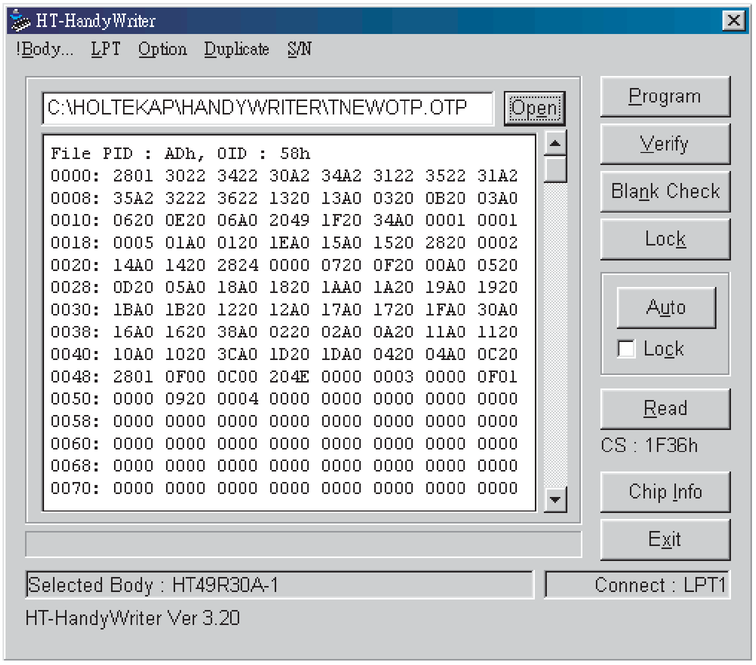


Fig 8-3

→ **LPT — Setup the Printer Port**
After running the HT-HandyWriter software program, a window as shown in Fig 8-4 will be shown, however it is first necessary to setup the correct printer port. By selecting "LPT" command, a sub menu as shown in Fig 8-5 will be displayed. From here LPT1, LPT2 or LPT3 can be chosen. If the OTP writer is connected to the HT-ICE, then select the printer port to which the HT-ICE is connected. For example if the HT-ICE is connected to LPT1 then select LPT1 from Fig 8-5. If the OTP writer is directly connected to the PC printer port then choose the relevant printer port in the same way.



C:\HOLTEKAP\HANDYWRITER\TNEWOTP.OTP

```
File PID : ADh, OID : 58h
0000: 2801 3022 3422 30A2 34A2 3122 3522 31A2
0008: 35A2 3222 3622 1320 13A0 0320 0B20 03A0
0010: 0620 0E20 06A0 2049 1F20 34A0 0001 0001
0018: 0005 01A0 0120 1EA0 15A0 1520 2820 0002
0020: 14A0 1420 2824 0000 0720 0F20 00A0 0520
0028: 0D20 05A0 18A0 1820 1AA0 1A20 19A0 1920
0030: 1BA0 1B20 1220 12A0 17A0 1720 1FA0 30A0
0038: 16A0 1620 38A0 0220 02A0 0A20 11A0 1120
0040: 10A0 1020 3CA0 1D20 1DA0 0420 04A0 0C20
0048: 2801 0F00 0C00 204E 0000 0003 0000 0F01
0050: 0000 0920 0004 0000 0000 0000 0000 0000
0058: 0000 0000 0000 0000 0000 0000 0000 0000
0060: 0000 0000 0000 0000 0000 0000 0000 0000
0068: 0000 0000 0000 0000 0000 0000 0000 0000
0070: 0000 0000 0000 0000 0000 0000 0000 0000
```

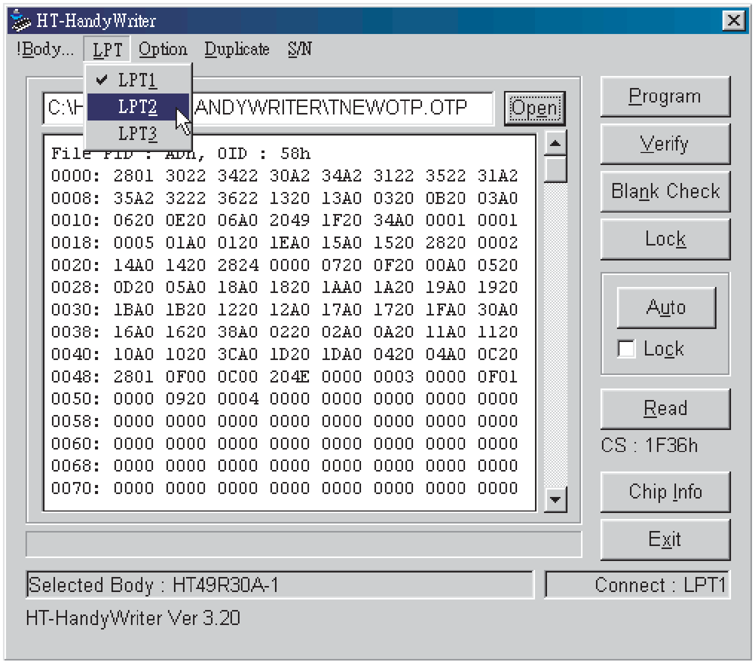
CS : 1F36h

Selected Body : HT49R30A-1

Connect : LPT1

HT-HandyWriter Ver 3.20

Fig 8-4



C:\H ANDYWRITER\TNEWOTP.OTP

```
File PID : ADh, OID : 58h
0000: 2801 3022 3422 30A2 34A2 3122 3522 31A2
0008: 35A2 3222 3622 1320 13A0 0320 0B20 03A0
0010: 0620 0E20 06A0 2049 1F20 34A0 0001 0001
0018: 0005 01A0 0120 1EA0 15A0 1520 2820 0002
0020: 14A0 1420 2824 0000 0720 0F20 00A0 0520
0028: 0D20 05A0 18A0 1820 1AA0 1A20 19A0 1920
0030: 1BA0 1B20 1220 12A0 17A0 1720 1FA0 30A0
0038: 16A0 1620 38A0 0220 02A0 0A20 11A0 1120
0040: 10A0 1020 3CA0 1D20 1DA0 0420 04A0 0C20
0048: 2801 0F00 0C00 204E 0000 0003 0000 0F01
0050: 0000 0920 0004 0000 0000 0000 0000 0000
0058: 0000 0000 0000 0000 0000 0000 0000 0000
0060: 0000 0000 0000 0000 0000 0000 0000 0000
0068: 0000 0000 0000 0000 0000 0000 0000 0000
0070: 0000 0000 0000 0000 0000 0000 0000 0000
```

CS : 1F36h

Selected Body : HT49R30A-1

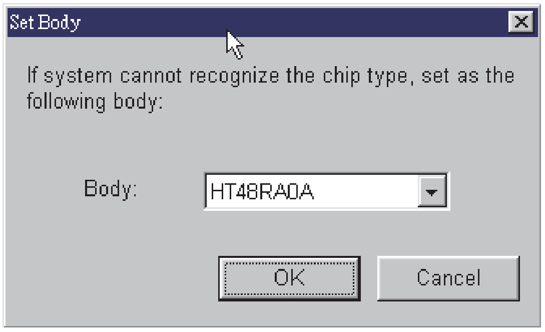
Connect : LPT1

HT-HandyWriter Ver 3.20

Fig 8-5

→ **!Body — Select the OTP MCU Type**

By clicking on "!Body", [Set Body] dialog will be shown as Fig 8-6. If there is no MCU type identifier stored in the OTP device, all the read/write operations will be completed according to the chip type that selected by users.

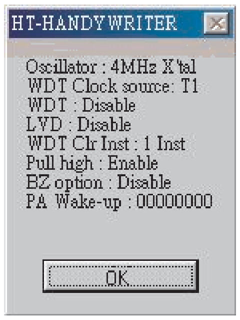


If system cannot recognize the chip type, set as the following body:

Fig 8-6

→ **Option — Check the MCU Option**

- Option
  When the [Option]/Option instruction is selected, a pop-up dialog, as shown in Fig 8-7, will be displayed. It will illustrate the option that comes from opened file or OTP device content.
- Print
  This instruction will print the option comes from opened file or OTP device content.



```
Oscillator : 4MHz X'tal
WDT Clock source: T1
WDT : Disable
LVD : Disable
WDT Clr Inst : 1 Inst
Pull high : Enable
BZ option : Disable
PA Wake-up : 00000000
```

Fig 8-7

→ **HT-HandyWriter Programming Functions**

Fig 8-4 shows the internal functions of the HT-HandyWriter. The 9 buttons shown at the right hand side of this window each represent an instruction, the function of which is explained below:

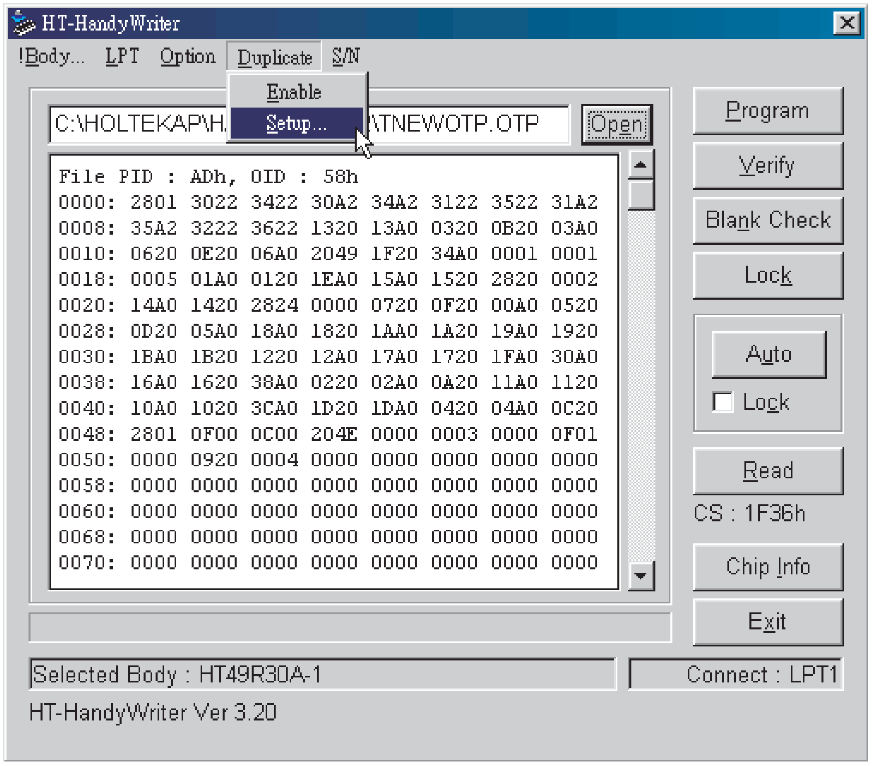
- Open
  This opens a file with the .OTP suffix, which will load the program contents into the PC ram memory. This data will be accessed when programming the relevant OTP device. After selecting "Open", the file dialogue box will be displayed from which the correct folder and file name can be chosen. The file content will be displayed in the message window after being opened, and the checksum of the opened file will be shown underneath the "Read" button.

- Program
  This instruction encompasses two functions. The first is to place the program data in the PC ram memory into the OTP device, the second is a verification check to verify that the actual data burned into the OTP device is the same as that in the PC ram memory data. After verification the result of this process will be shown on the HT-HandyWriter display.
- Verify
  The contents of the presently loaded OTP device will be read and checked that it is the same as the data loaded into the PC ram memory, the results of which will be displayed on the HT-HandyWriter display.
- Blank Check
  Check that the presently loaded OTP device has not previously been written to. The results of this check will be displayed on the HT-HandyWriter display. If the device is not empty, the memory area that has been written to will also be shown on the display.
- Lock
  This instruction will implement the protect function in the OTP device preventing the contents of this IC from being read. After programming an OTP device, this instruction can then be used to protect the contents.
- Auto
  This instruction will execute in order the three instructions Blank Check, Program and Verify. If any of the instructions do not execute correctly, the process will be halted and the following instruction not executed. There is also a lock function, which can be selected to prevent the data from being read out after programming. This lock function should first be selected before the Auto button pressed.
- Read
  This instruction will read out the contents of the OTP device presently loaded into the OTP writer and store them in the PC ram memory. This instruction will also cause the file checksum to be displayed underneath the ″Read″ button. If required, this data can also be stored in a file with the .OTP file suffix.
- Chip Info
  This instruction will read power-on ID, software ID, ROM size, option size from IC and display ″Get info from chip″ message to inform users the listed information comes from IC interior. If there is no such information inside IC, the specification defined by ″!Body″ command will be shown. It will display ″Get info from ini″ to inform users that above information comes from system setting.

→ **HT-HandyWriter Additional Functions**

- Duplicate – automatic OTP detection and duplication
  This function enables multi-OTPs of the same type to be continuously programmed. After opening the file using the Open instruction and inserting the OTP into the TEXTOOL socket, the OTP writer will automatically detect the device and then proceed to implement the functions that have been setup. In this way, after the desired .OTP file has been opened, it is only necessary to place the correct device in the socket to program a large number of devices.
  Before using this function, it is first necessary to setup the Auto-Program functions that are required. To setup these functions, select the [duplicate]/Setup instruction as shown in Fig 8-8. The Duplicate Setup window as shown in Fig 8-9 will then be displayed from which the user can select the required functions from the Blank Check, Program, Verify and Lock list.

When the [Duplicate]/Enable instruction is selected as shown in Fig 8-10, the Auto-Program function will be activated. After this instruction has been activated, it is now possible to proceed with multi-chip programming. After the chips have all been programmed, the Auto-Program function can be switched off, by again selecting the toggle action [Duplicate]/Enable instruction as shown in Fig 8-10.



```
File PID : ADh, OID : 58h
0000: 2801 3022 3422 30A2 34A2 3122 3522 31A2
0008: 35A2 3222 3622 1320 13A0 0320 0B20 03A0
0010: 0620 0E20 06A0 2049 1F20 34A0 0001 0001
0018: 0005 01A0 0120 1EA0 15A0 1520 2820 0002
0020: 14A0 1420 2824 0000 0720 0F20 00A0 0520
0028: 0D20 05A0 18A0 1820 1AA0 1A20 19A0 1920
0030: 1BA0 1B20 1220 12A0 17A0 1720 1FA0 30A0
0038: 16A0 1620 38A0 0220 02A0 0A20 11A0 1120
0040: 10A0 1020 3CA0 1D20 1DA0 0420 04A0 0C20
0048: 2801 0F00 0C00 204E 0000 0003 0000 0F01
0050: 0000 0920 0004 0000 0000 0000 0000 0000
0058: 0000 0000 0000 0000 0000 0000 0000 0000
0060: 0000 0000 0000 0000 0000 0000 0000 0000
0068: 0000 0000 0000 0000 0000 0000 0000 0000
0070: 0000 0000 0000 0000 0000 0000 0000 0000
```

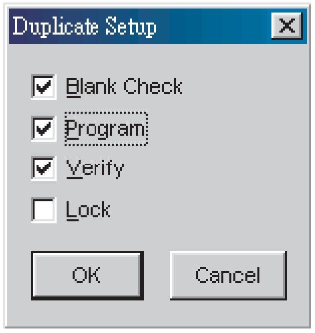
CS : 1F36h

Selected Body : HT49R30A-1

Connect : LPT1

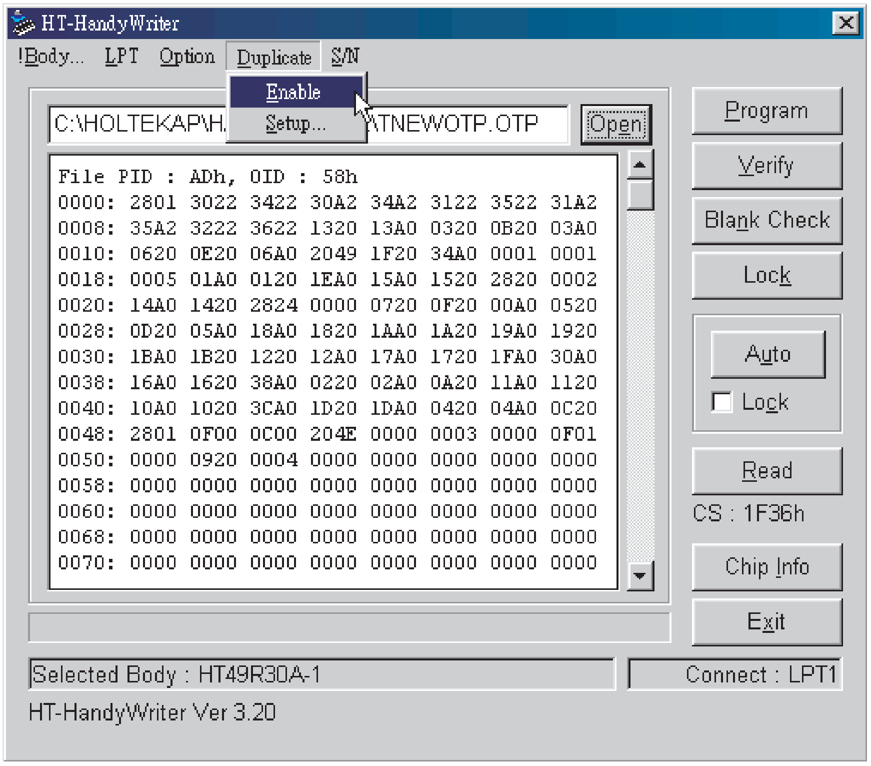
HT-HandyWriter Ver 3.20

Fig 8-8



☑ Blank Check
☑ Program
☑ Verify
☐ Lock

Fig 8-9



C:\HOLTEKAP\H. \TNEWOTP.OTP

```
File PID : ADh, OID : 58h
0000: 2801 3022 3422 30A2 34A2 3122 3522 31A2
0008: 35A2 3222 3622 1320 13A0 0320 0B20 03A0
0010: 0620 0E20 06A0 2049 1F20 34A0 0001 0001
0018: 0005 01A0 0120 1EA0 15A0 1520 2820 0002
0020: 14A0 1420 2824 0000 0720 0F20 00A0 0520
0028: 0D20 05A0 18A0 1820 1AA0 1A20 19A0 1920
0030: 1BA0 1B20 1220 12A0 17A0 1720 1FA0 30A0
0038: 16A0 1620 38A0 0220 02A0 0A20 11A0 1120
0040: 10A0 1020 3CA0 1D20 1DA0 0420 04A0 0C20
0048: 2801 0F00 0C00 204E 0000 0003 0000 0F01
0050: 0000 0920 0004 0000 0000 0000 0000 0000
0058: 0000 0000 0000 0000 0000 0000 0000 0000
0060: 0000 0000 0000 0000 0000 0000 0000 0000
0068: 0000 0000 0000 0000 0000 0000 0000 0000
0070: 0000 0000 0000 0000 0000 0000 0000 0000
```

CS : 1F36h

Selected Body : HT49R30A-1

Connect : LPT1

HT-HandyWriter Ver 3.20

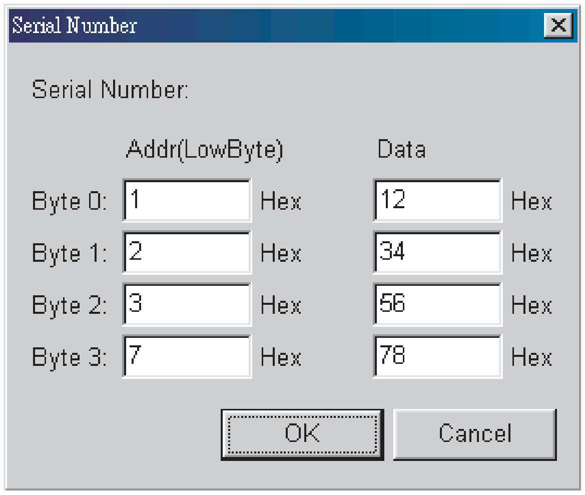
Fig 8-10

- S/N – Writing Serial Numbers

  The serial number function allows a user specified serial number to be written into each device. This serial number and its address is specified by the user and is written into the lower byte address of the Program ROM for each device. After a serial number is written into a device, an auto-incrementing function ensures that subsequently programmed devices will contain serial numbers incremented by one each time.

  First it is required to setup the initial data and fixed address of the first serial number. After selecting [S/N]/Setup, the window, as shown in Fig 8-11, can be used to input the initial serial number's data and its corresponding address.

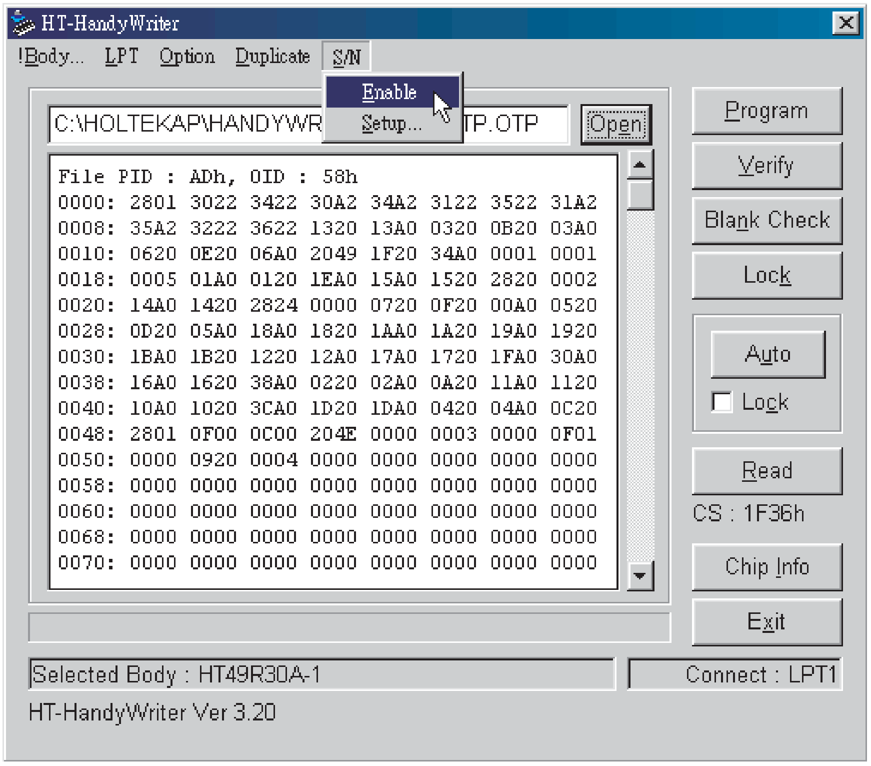
  After the initial data and address information has been setup, [S/N]/Enable should be selected to activate the serial number function. When the serial number function is activated, the present serial number's corresponding address and data will be displayed at the lower right hand side of the main window. During the programming stage the first device to be programmed will contain the previously setup serial number data in its Program ROM at the indicated address. Subsequent devices will contain serial numbers incremented by one for each additional device. To reset the order of serial numbers, again select the [S/N]/Setup function.



Serial Number:

	Addr(LowByte)		Data	
Byte 0:	1	Hex	12	Hex
Byte 1:	2	Hex	34	Hex
Byte 2:	3	Hex	56	Hex
Byte 3:	7	Hex	78	Hex

Fig 8-11



C:\HOLTEKAP\HANDYWR TP.OTP

```
File PID : ADh, OID : 58h
0000: 2801 3022 3422 30A2 34A2 3122 3522 31A2
0008: 35A2 3222 3622 1320 13A0 0320 0B20 03A0
0010: 0620 0E20 06A0 2049 1F20 34A0 0001 0001
0018: 0005 01A0 0120 1EA0 15A0 1520 2820 0002
0020: 14A0 1420 2824 0000 0720 0F20 00A0 0520
0028: 0D20 05A0 18A0 1820 1AA0 1A20 19A0 1920
0030: 1BA0 1B20 1220 12A0 17A0 1720 1FA0 30A0
0038: 16A0 1620 38A0 0220 02A0 0A20 11A0 1120
0040: 10A0 1020 3CA0 1D20 1DA0 0420 04A0 0C20
0048: 2801 0F00 0C00 204E 0000 0003 0000 0F01
0050: 0000 0920 0004 0000 0000 0000 0000 0000
0058: 0000 0000 0000 0000 0000 0000 0000 0000
0060: 0000 0000 0000 0000 0000 0000 0000 0000
0068: 0000 0000 0000 0000 0000 0000 0000 0000
0070: 0000 0000 0000 0000 0000 0000 0000 0000
```

CS : 1F36h

Selected Body : HT49R30A-1

Connect : LPT1

HT-HandyWriter Ver 3.20

Fig 8-12

System Messages

→ **HT-HandyWriter Connect to LPT1.**
OTP writer already connected to LPT1.

→ **Cannot Connect to ICE**
Connection problems between the OTP writer, the HT-ICE and the printer port.

→ **Invalid EV Chip!**
The OTP writer is unable to support the EV chip in the HT-ICE. The HT-ICE must be changed for correct operation to take place.

→ **Connect to HT-HandyWriter Through ICE**
The OTP writer is successfully connected via the HT-ICE.

→ **Cannot find HT-HandyWriter, Please Connect It to ICE**
Or This HT-HandyWriter is an Old Version
The HT-ICE is already connected to the printer port, but the OTP writer is not connected to the HT-ICE. It may also be that an old version of the OTP writer is being used (THANDYOTP-A) so the system is unable to detect a good connection. If the former case, please connect the OTP writer directly to the ICE.

→ **File PID: ADh, OID: 50h**
The opened files recorded power-on ID is ADh, the software ID is 50h.

→ **Invalid OTP File Format**
The opened file format is incorrect.

→ **The Chip PID: ADh, OID: 50h Doesn t Match with the File PID: ADh, OID: 51h**
Are You Sure to Continue?
The type of OTP device and the chip supported by the opened file does not match.

→ **Chip ROM Size: 0400h, File ROM Size: 0800h. System Will Set ROM Size as 0400h.**
Are You Sure to Continue?
The OTP device has 400h of writable space, the file content is 800h, so the OTP writer can only write 400h of data into the contents of the OTP device.

→ **Addr: xxxxh, Data: yyyyh, Rdata: zzzzh**
Program/Option Verify Failed!
Errors exist in either the program or option verification information. The reason is because the data zzzzh at the address xxxxh in the OTP device is not the same as the data yyyyh in the PC ram memory.

→ **Addr: xxxxh, Data: zzzzh**
Not Blank!
The OTP device is not blank as the address xxxxh contains the data zzzzh, inhibiting the implementation of further instructions.

→ **Chip Mismatched!**
The OTP device presently in the OTP writer and the OTP device mentioned in the .OTP file do not match, inhibiting the implementation of further instructions.

→ **Chip is Locked!**
The OTP device presently in the OTP writer is locked, inhibiting the implementation of further instructions.

→ **No Data to Verify/Program!**

Before executing the Verify or Program instruction, the .OTP file must be loaded using the ″Open″ function in the HT-HandyWriter system software.

Part II

Development Language and Tools

Chapter 9

Assembly Language and Cross Assembler

9

Assembly-Language programs are written as source files. They can be assembled into object files by the Holtek Cross Assembler. Object files are combined by the Cross Linker to generate a task file.

A source program is made up of statements and look up tables, giving directions to the Cross Assembler at assembly time or to the processor at run time. Statements are constituted by mnemonics (operations), operands and comments.

Notational Conventions

The following list describes the notations used by this document.

Example of convention	Description of convention
[*optional items*]	Syntax elements that are enclosed by a pair of brackets are optional. For example, the syntax of the command line is as follows: **HASM** [*options*] *filename* [;] In the above command line, *options* and semicolon; are both optional, but *filename* is required, except for the following case: Brackets in the instruction operands. In this case, the brackets refer to memory address.
{*choice1* | *choice2*}	Braces and vertical bars stand for a choice between two or more items. Braces enclose the choices whereas vertical bars separate the choices. Only one item can be chosen.

Example of convention	Description of convention
Repeating elements...	Three dots following an item signify that more items with the same form may be entered. For example, the directive PUBLIC has the following form: **PUBLIC** *name1* [,*name2* [,...]] In the above form, the three dots following *name2* indicate that many names can be entered as long as each is preceded by a comma.

Statement Syntax

The construction of each statement is as follows:

[*name*] [*operation*] [*operands*] [;*comment*]

- All fields are optional.
- Each field (except the comment field) must be separated from other fields by at least one space or one tab character.
- Fields are not case-sensitive, i.e., lower-case characters are changed to upper-case characters before processing.

Name

Statements can be assigned labels to enable easy access by other statements. A name consists of the following characters:

A~Z a~z 0~9 ? _ @

with the following restrictions :

- 0~9 cannot be the first character of a name
- ? cannot stand alone as a name
- Only the first 31 characters are recognized

Operation

The operation defines the statement action of which two types exist, directives and instructions. Directives give directions to the Cross Assembler, specifying the manner in which the Cross Assembler is to generate the object code at assembly time. Instructions, on the other hand, give directions to the processor. They are translated to object code at assembly time, the object code in turn controls the behavior of the processor at run time.

Operand

Operands define the data used by directives and instructions. They can be made up of symbols, constants, expressions and registers.

Comment

Comments are the descriptions of codes. They are used for documentation only and are ignored by the Cross Assembler. Any text following a semicolon is considered a comment.

Assembly Directives

Directives give direction to the Cross Assembler, specifying the manner in which the Cross Assembler generates object code at assembly time. Directives can be further classified according to their behavior as described below.

Conditional Assembly Directives

The conditional block has the following form:

IF
statements
[**ELSE**
statements]
ENDIF

→ **Syntax**
`IF` *`expression`*
`IFE` *`expression`*

- Description
  The directives **`IF`** and **`IFE`** test the *`expression`* following them.
  The **`IF`** directive grants assembly if the value of the *`expression`* is true, i.e. non-zero.
  The **`IFE`** directive grants assembly if the value of the *`expression`* is false, i.e. zero.
- Example

```
IF  debugcase
        ACC1    equ 5
        extern  username: byte
ENDIF
```

In this example, the value of the variable `ACC1` is set to 5 and the `username` is declared as an external variable if the symbol `debugcase` is evaluated as true, i.e. nonzero.

→ **Syntax**
`IFDEF` *`name`*
`IFNDEF` *`name`*

- Description
  The directives **`IFDEF`** and **`IFNDEF`** test whether or not the given *`name`* has been defined. The **`IFDEF`** directive grants assembly only if the *`name`* is a label, a variable or a symbol. The **`IFNDEF`** directive grants assembly only if the *`name`* has not yet been defined. The conditional assembly directives support a nesting structure, with a maximum nesting level of 7.
- Example

```
IFDEF       buf_flag
            buffer  DB  20 dup(?)
ENDIF
```

In this example, the `buffer` is allocated only if the `buf_flag` has been previously defined.

File Control Directives

→ **Syntax**
`INCLUDE` *`file-name`*
`or`
`INCLUDE` "*`file-name`*"

- Description
  This directive inserts source codes from the source file given by *`file-name`* into the current source file during assembly. Cross Assembler supports at most 7 nesting levels.
- Example
  ```
  INCLUDE macro.def
  ```
  In this example, the Cross Assembler inserts the source codes from the file `macro.def` into the current source file.

→ **Syntax**
`PAGE` *`size`*

- Description
  This directive specifies the number of the lines in a page of the program listing file. The page size must be within the range from 10 to 255, the default page size is 60.
- Example
  ```
  PAGE 57
  ```
  This example sets the maximum page size of the listing file to 57 lines.

→ **Syntax**
`.LIST`
`.NOLIST`

- Description
  The directives **`.LIST`** and **`.NOLIST`** decide whether or not the source program lines are to be copied to the program listing file. **`.NOLIST`** suppresses copying of subsequent source lines to the program listing file. **`.LIST`** restores the copying of subsequent source lines to the program listing file. The default is **`.LIST`**.
- Example
  ```
  .NOLIST
  mov a, 1
  mov b1, a
  .LIST
  ```

In this example, the two instructions in the block enclosed by **`.NOLIST`** and **`.LIST`** are suppressed from copying to the source listing file.

→ **Syntax**
`.LISTMACRO`
`.NOLISTMACRO`

- Description
  The directive **`.LISTMACRO`** causes the Cross Assembler to list all the source statements, including comments, in a macro. The directive **`.NOLISTMACRO`** suppresses the listing of all macro expansions. The default is **`.NOLISTMACRO`**.

→ **Syntax**
`.LISTINCLUDE`
`.NOLISTINCLUDE`

- Description
The directive **`.LISTINCLUDE`** inserts the contents of all included files into the program listing. The directive **`.NOLISTINCLUDE`** suppresses the addition of included files. The default is **`.NOLISTINCLUDE`**.

→ **Syntax**
`MESSAGE` '*`text-string`*'

- Description
The directive **`MESSAGE`** directs the Cross Assembler to display the *`text-string`* on the screen. The characters in the *`text-string`* must be enclosed by a pair of single quotation marks.

→ **Syntax**
`ERRMESSAGE` '*`error-string`*'

- Description
The directive **`ERRMESSAGE`** directs the Cross Assembler to issue an error. The characters in the *`error-string`* must be enclosed by a pair of single quotation marks.

Program Directives

→ **Syntax (comment)**
`;` *`text`*

- Description
A comment consists of characters preceded by a semicolon (;) and terminated by an embedded carriage-return/line-feed.

→ **Syntax**
`name` **`.SECTION`** [*`align`*] [*`combine`*] '*`class`*'

- Description
The **`.SECTION`** directive marks the beginning of a program section. A program section is a collection of instructions and/or data whose addresses are relative to the section beginning with the name which defines that section. The *`name`* of a section can be unique or be the same as the name given to other sections in the program. Sections with the same complete names are treated as the same section.
The optional *`align`* type defines the alignment of the given section. It can be one of the following:

BYTE	uses any byte address (the default align type)
WORD	uses any word address
PARA	uses a paragraph address
PAGE	uses a page address

For the CODE section, the byte address is in a single instruction unit. **BYTE** aligns the section at any instruction address, **WORD** aligns the section at any even instruction address, **PARA** aligns the section at any instruction address which is a multiple of 16, and **PAGE** aligns the section at any instruction address with a multiple of 256.

For DATA sections, the byte address is in one byte units (8 bits/byte). **BYTE** aligns the section at any byte address, **WORD** aligns the section at any even address, **PARA** aligns the section at any address which is a multiple of 16, and **PAGE** aligns the section at any address which is a multiple of 256.

The optional *`combine`* type defines the way of combining sections having the same complete name (section and class name). It can be any one of the following:

– COMMON
  Creates overlapping sections by placing the start of all sections with the same complete name at the same address. The length of the resulting area is the length of the longest section.

– AT *address*
  Causes all label and variable addresses defined in a section to be relative to the given address. The *address* can be any valid expression except a forward reference. It is an absolute address in a specified ROM/RAM bank and must be within the ROM/RAM range.

If no *`combine`* type is given, the section is combinative, i.e., this section can be concatenated with all sections having the same complete name to form a single, contiguous section.

The *`class`* type defines the sections that are to be loaded in the contiguous memory. Sections with the same class name are loaded into the memory one after another. The class name **CODE** is used for sections stored in ROM, and the class name **DATA** is used for sections stored in RAM. The complete name of a section consists of a section name and a class name. The named section includes all codes and data below (after) it until the next section is defined.

→ **Syntax**
`ROMBANK` *`banknum section-name`* `[,`*`section-name`*`,...]`

• Description
This directive declares which sections are allocated to the specified ROM bank. The *`banknum`* specifies the ROM bank, ranging from 0 to the maximum bank number of the destination MCU. The *`section-name`* is the name of the section defined previously in the program. More than one section can be declared in a bank as long as the total size of the sections does not exceed the bank size of 8K words. If this directive is not declared, bank 0 is assumed and all CODE sections defined in this program will be in bank 0. If a CODE section is not declared in any ROM bank, then bank 0 is assumed.

→ **Syntax**
`RAMBANK` *`banknum section-name [,section-name,...]`*

• Description
This directive is similar to **`ROMBANK`** except that it specifies the RAM bank, the size of RAM bank is 256 bytes.

→ **Syntax**
`END`

• Description
This directive marks the end of a program. Adding this directive to any included file should be avoided.

→ **Syntax**
`ORG` *`expression`*

- Description
  This directive sets the location counter to *`expression`*. The subsequent code and data offsets begin at the new offset specified by *`expression`*. The code or data offset is relative to the beginning of the section where the directive **`ORG`** is defined. The attribute of a section determines the actual value of offset, absolute or relative.
- Example

```
ORG 8
mov A, 1
```

In this example, the statement `mov A, 1` begins at location 8 in the current section.

→ **Syntax**
`PUBLIC` *`name1`* `[,`*`name2`* `[,...]]`
`EXTERN` *`name1`*`:`*`type`* `[,`*`name2`*`:`*`type`* `[, ...]]`

- Description
  The **`PUBLIC`** directive marks the variable or label specified by a name that is available to other modules in the program. The **`EXTERN`** directive, on the other hand, declares an external variable, label or symbol of the specified name and type. The type can be one of the four types: **BYTE**, **WORD** and **BIT** (these three types are for data variables), and **NEAR** (a label type and used by call or jmp).
- Example

```
PUBLIC  start, setflag
EXTERN  tmpbuf:byte
CODE      .SECTION 'CODE'
start:
      mov   a, 55h
      call  setflag
      ....
setflag     proc
      mov   tmpbuf, a
      ret
setflag     endp
end
```

In this example, both the label `start` and the procedure `setflag` are declared as public variables. Programs in other sources may refer to these variables. The variable `tmpbuf` is also declared as external. There should be a source file defining a byte that is named `tmpbuf` and is declared as a public variable.

→ **Syntax**

```
name    PROC
name    ENDP
```

- Description

  The **PROC** and **ENDP** directives mark a block of code which can be called or jumped to from other modules. The **PROC** creates a label *name* which stands for the address of the first instruction of a procedure. The Cross Assembler will set the value of the label to the current value of the location counter.

- Example

```
toggle      PROC
mov         tmpbuf, a
mov         a, 1
xorm        a, flag
mov         a, tmpbuf
ret
toggle      ENDP
```

→ **Syntax**

```
[label:] DC expression1 [,expression2 [,...]]
```

- Description

  The **DC** directive stores the value of *expression1*, *expression2* etc. in consecutive memory locations. This directive is used for the CODE section only. The bit size of the result value is dependent on the ROM size of the MCU. The Cross Assembler will clear any redundant bits; *expression1* has to be a value or a label. This directive may also be employed to setup the table in the code section.

- Example

```
table1: DC  0128h, 025CH
```

  In this example, the Cross Assembler reserves two units of ROM space and also stores 0128H and 025CH into these two ROM units.

Data Definition Directives

An assembly language program consists of one or more statements and comments. A statement or comment is a composition of characters, numbers, and names. The assembly language supports integer numbers. An integer number is a collection of binary, octal, decimal, or hexadecimal digits along with an optional radix. If no radix is given, the Cross Assembler uses the default radix (decimal). The table lists the digits that can be used with each radix.

Radix	Type	Digits
B	Binary	01
O	Octal	01234567
D	Decimal	0123456789
H	Hexadecimal	0123456789ABCDEF

→ **Syntax**

```
[name] DB  value1 [,value2 [, ...]]
[name] DW  value1 [,value2 [, ...]]
[name] DBIT
[name] DB  repeated-count DUP(?)
[name] DW  repeated-count DUP(?)
```

- Description

These directives reserve the number of bytes/words specified by the repeated-count or reserve bytes/words only. *value1* and *value2* should be ? due to the microcontroller RAM . The Cross Assembler will not initialize the RAM data. **DBIT** reserves a bit. The content ? denotes uninitialized data, i.e., reserves the space of the data. The Cross Assembler will gather every 8 **DBIT** together and reserve a byte for these 8 **DBIT** variables.

- Example

```
DATA        .SECTION   'DATA'
tbuf        DB  ?
chksum      DW  ?
flag1       DBIT
sbuf        DB  ?
cflag       DBIT
```

In this example, the Cross Assembler reserves byte location 0 for `tbuf`, location 1 and 2 for `chksum`, bit 0 of location 3 for `flag1`, location 4 for `sbuf` and bit 1 of location 3 for `cflag`.

→ **Syntax**

```
name    LABEL   {BIT|BYTE|WORD}
```

- Description

The *name* with the data type has the same address as the following data variable

- Example

```
lab1        LABEL       WORD
d1          DB  ?
d2          DB  ?
```

In this example, `d1` is the low byte of `lab1` and `d2` is the high byte of `lab1`.

→ **Syntax**

```
name    EQU     expression
```

- Description

The **EQU** directive creates absolute symbols, aliases, or text symbols by assigning an *expression* to *name*. An absolute symbol is a name standing for a 16-bit value; an alias is a name representing another symbol; a text symbol is a name for another combination of characters. The *name* must be unique, i.e. not having been defined previously. The *expression* can be an integer, a string constant, an instruction mnemonic, a constant expression, or an address expression.

- Example

```
accreg  EQU   5
bmove   EQU   mov
```

In this example, the variable `accreg` is equal to 5, and `bmove` is equal to the instruction `mov`.

Macro Directives

Macro directives enable a block of source statements to be named, and then that name to be re-used in the source file to represent the statements. During assembly, the Cross Assembler automatically replaces each occurrence of the macro name with the statements in the macro definition.

A macro can be defined at any place in the source file as long as the definition precedes the first source line that calls this macro. In the macro definition, the macro to be defined may refer to other macros which have been previously defined. The Cross Assembler supports a maximum of 7 nesting levels.

→ **Syntax**

```
name    MACRO [dummy-parameter [, ...]]
        statements
        ENDM
```

The Cross Assembler supports a directive **LOCAL** for the macro definition.

→ **Syntax**

```
name    LOCAL dummy-name [, ...]
```

- Description

  The **LOCAL** directive defines symbols available only in the defined macro. It must be the first line following the **MACRO** directive, if it is present. The *dummy-name* is a temporary name that is replaced by a unique name when the macro is expanded. The Cross Assembler creates a new actual name for *dummy-name* each time the macro is expanded. The actual name has the form `??digit`, where digit is a hexadecimal number within the range from 0000 to FFFF. A label should be added to the **LOCAL** directive when labels are used within the **MACRO**/**ENDM** block. Otherwise, the Cross Assembler will issue an error if this **MACRO** is referred to more than once in the source file.

  In the following example, `tmp1` and `tmp2` are both dummy parameters, and are replaced by actual parameters when calling this macro. `label1` and `label2` are both declared **LOCAL**, and are replaced by `??0000` and `??0001` respectively at the first reference, if no other **MACRO** is referred. If no **LOCAL** declaration takes place, `label1` and `label2` will be referred to labels, similar to the declaration in the source program. At the second reference of this macro, a multiple define error message is displayed.

```
Delay   MACRO   tmp1, tmp2
    LOCAL       label1, label2
    mov     a, 70h
    mov     tmp1, a
label1:
    mov     tmp2, a
label2:
    clr     wdt1
    clr     wdt2
    sdz     tmp2
    jmp     label2
    sdz     tmp1
    jmp     label1
    ENDM
```

The following source program refers to the macro Delay ...

```
; T.ASM
;  Sample program for MACRO.
.ListMacro
Delay MACRO  tmp1, tmp2
      LOCAL  label1, label2
      mov    a, 70h
      mov    tmp1, a
label1:
      mov    tmp2, a
label2:
      clr    wdt1
      clr    wdt2
      sdz    tmp2
      jmp    label2
      sdz    tmp1
      jmp    label1
      ENDM

data .section 'data'
BCnt db ?
SCnt db ?

code .section at 0 'code'
Delay BCnt, SCnt
end
```

The Cross Assembler will expand the macro Delay as shown in the following listing file. Note that the offset of each line in the macro body, from line 4 to line 17, is 0000. Line 24 is expanded to 11 lines and forms the macro body. In addition the formal parameters, `tmp1` and `tmp2`, are replaced with the actual parameters, `BCnt` and `SCnt`, respectively.

```
File: T.asm         Holtek Cross-Assembler  Version 2.80      Page 1

   1  0000                ; T.ASM
   2  0000                ;  Sample program for MACRO.
   3  0000                .ListMacro
   4  0000                Delay MACRO  tmp1, tmp2
   5  0000                      LOCAL  label1, label2
   6  0000                      mov    a, 70h
   7  0000                      mov    tmp1, a
   8  0000                label1:
   9  0000                      mov    tmp2, a
  10  0000                label2:
  11  0000                      clr    wdt1
  12  0000                      clr    wdt2
  13  0000                      sdz    tmp2
  14  0000                      jmp    label2
  15  0000                      sdz    tmp1
  16  0000                      jmp    label1
  17  0000                      ENDM
  18  0000
  19  0000                data .section 'data'
  20  0000  00            BCnt db ?
  21  0001  00            SCnt db ?
  22  0002
  23  0000                code .section at 0 'code'
  24  0000                Delay BCnt, SCnt
  24  0000  0F70      1         mov    a, 70h
  24  0001  0080     R1         mov    BCnt, a
  24  0002            1   ??0000:
  24  0002  0080     R1         mov    SCnt, a
  24  0003            1   ??0001:
  24  0003  0001      1         clr    wdt1
  24  0004  0005      1         clr    wdt2
  24  0005  1780     R1         sdz    SCnt
  24  0006  2803      1         jmp    ??0001
  24  0007  1780     R1         sdz    BCnt
  24  0008  2802      1         jmp    ??0000
  25  0009                end

        0 Errors
```

Assembly Instructions

The syntax of an instruction has the following form:

[*name*:] *mnemonic* [*operand1*[,*operand2*]] [;*comment*]

where

name:	→ label name
mnemonic	→ instruction name (keywords)
operand1	→ registers memory address
operand2	→ registers memory address immediate value

Name

A name is made up of letters, digits, and special characters, and is used as a label.

Mnemonic

Mnemonic is an instruction name dependent upon the type of the MCU used in the source program.

Operand, Operator and Expression

Operands (source or destination) are the argument defining values that are to be acted on by instructions. They can be constants, variables, registers, expressions or keywords. When using the instruction statements, care must be taken to select the correct operand type, i.e. source operand or destination operand. The dollar sign $ is a special operand, namely the current location operand.

An expression consists of many operands that are combined to describe a value or a memory location. The combined operators are evaluated at assembly time. They can contain constants, symbols, or any combination of constants and symbols that are separated by arithmetic operators.

Operators specify the operations to be performed while combining the operands of an expression. The Cross Assembler provides many operators to combine and evaluate operands. Some operators work with integer constants, some with memory values, and some with both. Operators handle the calculation of constant values that are known at the assembly time. The following are some operators provided by the Cross Assembler.

- Arithmetic operators + - * / % (MOD)
- SHL and SHR operators
  - Syntax
    expression **SHR** *count*
    expression **SHL** *count*

  The values of these shift bit operators are all constant values. The *expression* is shifted right **SHR** or left **SHL** by the number of bits specified by *count*. If bits are shifted out of position, the corresponding bits that are shifted in are zero-filled. The following are such examples:

```
mov A, 01110111b SHR 3   ; result ACC=00001110b
mov A, 01110111b SHL 4   ; result ACC=01110000b
```

- Bitwise operators NOT, AND, OR, XOR
  - Syntax
    NOT *expression*
    expression1 **AND** *expression2*
    expression1 **OR** *expression2*
    expression1 **XOR** *expression2*

NOT	is a bitwise complement.
AND	is a bitwise AND.
OR	is a bitwise inclusive OR.
XOR	is a bitwise exclusive OR.

- OFFSET operator
  - Syntax
    `OFFSET` *`expression`*

  The **`OFFSET`** operator returns the offset address of an *`expression`*. The *`expression`* can be a label, a variable, or other direct memory operand. The value returned by the OFFSET operator is an immediate operand.
- LOW, MID and HIGH operator
  - Syntax
    `LOW` *`expression`*
    `MID` *`expression`*
    `HIGH` *`expression`*

  The **`LOW/MID/HIGH`** operator returns the value of an *`expression`* if the result of the *`expression`* is an immediate value. The **`LOW/MID/HIGH`** operators will then take the low/middle/high byte of this value. But if the *`expression`* is a label, the **`LOW/MID/HIGH`** operator will take the values of the low/middle/high byte of the program count of this label.
- BANK operator
  - Syntax
    `BANK` *`name`*

  The **`BANK`** operator returns the bank number allocated to the section of the *`name`* declared. If the *`name`* is a label then it returns the rom bank number. If the *`name`* is a data variable then it returns the ram bank number. The format of the bank number is the same as the BP defined. For more information of the format please refer to the data sheets of the corresponding MCUs. (Note: The format of the BP might be different between MCUs.)
  Example 1:

```
mov A, BANK start
mov BP,A
jmp start
```

  Example 2:

```
mov A, BANK var
mov BP,A
mov A, OFFSET var
mov MP1,A
mov A,IAR1
```

- Operator precedence

Precedence	Operators
1 (Highest)	( ), [ ]
2	+, – (unary), LOW, MID, HIGH, OFFSET, BANK
3	*, /, %, SHL, SHR
4	+, – (binary)
5	> (greater than), >= (greater than or equal to), < (less than), <= (less than or equal to)
6	== (equal to), != (not equal to)
7	! (bitwise NOT)
8	& (bitwise AND)
9 (Lowest)	|(bitwise OR), ^(bitwise XOR)

Miscellaneous

Forward References

The Cross Assembler allows reference to labels, variable names, and other symbols before they are declared in the source code (forward named references). But symbols to the right of **EQU** are not allowed to be forward referenced.

Local Labels

A local label is a label with a fixed form such as $number. The number can be 0~29. The function of a local label is the same as a label except that the local label can be used repeatedly. The local label should be used between any two consecutive labels and the same local label name may used between other two consecutive labels. The Cross Assembler will transfer every local label into a unique label before assembling the source file. At most 30 local labels can be defined between two consecutive labels.
Example.

```
Label1:                             ; label
        $1:                         ;; local label
                mov a, 1
                jmp $3
        $2:                         ;; local label
                mov a, 2
                jmp $1
        $3:                         ;; local label
                jmp $2
Label2:                             ; label
                jmp $1
        $0:                         ;; local label
                jmp Label1
        $1:     jmp $0
Label3:
```

Reserved Assembly Language Words

The following tables list all reserved words used by the assembly language.

- Reserved Names (directives, operators)

$	DUP	INCLUDE	NOT
*	DW	LABEL	OFFSET
+	ELSE	.LIST	OR
–	END	.LISTINCLUDE	ORG
.	ENDIF	.LISTMACRO	PAGE
/	ENDM	LOCAL	PARA
=	ENDP	LOW	PROC
?	EQU	MACRO	PUBLIC
[ ]	ERRMESSAGE	MESSAGE	RAMBANK
AND	EXTERN	MID	ROMBANK
BANK	HIGH	MOD	.SECTION
BYTE	IF	NEAR	SHL
DB	IFDEF	.NOLIST	SHR
DBIT	IFE	.NOLISTINCLUDE	WORD
DC	IFNDEF	.NOLISTMACRO	XOR

- Reserved Names (instruction mnemonics)

ADC	HALT	RLCA	SUB
ADCM	INC	RR	SUBM
ADD	INCA	RRA	SWAP
ADDM	JMP	RRC	SWAPA
AND	MOV	RRCA	SZ
ANDM	NOP	SBC	SZA
CALL	OR	SBCM	TABRDC
CLR	ORM	SDZ	TABRDL
CPL	RET	SDZA	XOR
CPLA	RETI	SET	XORM
DAA	RL	SIZ	
DEC	RLA	SIZA	
DECA	RLC	SNZ	

- Reserved Names (registers names)

A	WDT	WDT1	WDT2

Cross Assembler Options

The Cross Assembler options can be set via the Options menu Project command in HT-IDE3000. The Cross Assembler Options is located on the center part of the Project Option dialog box, as shown in Fig 3-12.

The symbols could be defined in the *Define Symbol* edit box.

→ **Syntax**

```
symbol1[=value1] [, symbol2[=value2] [, ...]]
```

• Example,

```
debugflag=1, newver=3
```

The check box of the *Generate listing file* is used to decide whether the listing file should be generated or not. If the check box is checked, the listing file will be generated. Otherwise, it won't be generated.

Assembly Listing File Format

The Assembly Listing File contains the source program listing and summary information. The first line of each page is a title line which include company name, the Cross Assembler version number, source file name, date/time of assembly and page number.

Source Program Listing

Each line in the source program has the following syntax:

```
line-number offset [code] statement
```

- `Line-number` is the number of the line starting from the first statement in the assembly source file (4 decimal digits).
- The 2nd field – `offset` – is the offset from the beginning of the current section to the code (4 hexadecimal digits)
- The 3rd field – `code` – is present only if the statement generates code or data (two hexadecimal 4-digit data)

  The `code` shows the numeric value in hexadecimal if the value is known at assembly time. Otherwise, a proper flag will indicate the action required to compute the value. The following two flags may appear behind the code field.

R	→ relocatable address (Cross Linker must resolve)
E	→ external symbol (Cross Linker must resolve)

  The following flag may appear before the code field

=	→ **EQU** or equal-sign directive

  The following 2 flags may appear in the code field

----	→ section address (Cross Linker must resolve)
nn[xx]	→ **DUP** expression: nn **DUP**(?)

- The 4th field – `statement` – is the source statement shown exactly as it appears in the source file, or as expanded by a macro. The following flags may appear before a statement.

n	→ Macro-expansion nesting level
C	→ line from **INCLUDE** file

- Summary

```
0         1         2         3         4         5         6
123456789012345678901234567890123456789012345678901234567890...
llll  oooo hhhh hhhh EC source-program-statement
                     Rn
```

llll → line number (4 digits, right alignment)

oooo → offset of code (4 digits)

hhhh → two 4-digits for opcode

E → external reference

C → statement from included file

R → relocatable name

n → Macro-expansion nesting level

Summary of Assembly

The total warning number and total error number is the information provided at the end of the Cross Assembler listing file.

Miscellaneous

If any errors occur during assembly, each error message and error number will appear directly below the statement where the error occurred.

→ **Example of Assembly Listing File**

```
File: SAMPLE.ASM    Holtek Cross-Assembler  Version 2.86      Page 1


  1  0000              page 60
  2  0000              message    'Sample Program 1'
  3  0000
  4  0000              .listinclude
  5  0000              .listmacro
  6  0000
  7  0000              #include "sample.inc"

  1  0000            C pa    equ    [12h]
  2  0000            C pac   equu   [13h]
  3  0000            C pb    equ    [14h]
  4  0000            C pbc   equ    [15h]
  5  0000            C pc    equ    [16h]
  6  0000            C pcc   equ    [17h]
  7  0000            C

  8  0000
  9  0000              extern extlab : near
 10  0000              extern extb1 : byte
 11  0000
 12  0000              clrpb macro
 13  0000              clr pb
 14  0000              endm
 15  0000
 16  0000              clrpa macro
 17  0000              mov a, 00h
 18  0000              mov pa, a
 19  0000              clrpb
 20  0000              endm
 21  0000
 22  0000              data .section 'data'
 23  0000  00          b1    db ?
 24  0001  00          b2    db ?
 25  0002  00          bit1  dbit
 26  0003
 27  0000              code .section 'code'
 28  0000  0F55        mov  a, 055h
 29  0001  0080      R mov  b1, a
 30  0002  0080      E mov  extb1, a
 31  0003  0FAA        mov  a, 0aah
 32  0004  0093        mov  pac, a
 33  0005              clrpa
 33  0005  0F00      1 mov a, 00h
 33  0006  0092      1 mov [12h], a
 33  0007            1 clrpb
 33  0007  1F14      2 clr [14h]
 34  0008  0700      R mov  a, b1
 35  0009  0F00      E mov  a, bank extlab
 36  000A  0F00      E mov  a, offset extb1
 37  000B  2800      E jmp  extlab
 38  000C
 39  000C  1234 5678   dw   1234h, 5678h, 0abcdh, 0ef12h
           ABCD EF12
 40  0010              end

        0 Errors
```

Chapter 10

Holtek C Language

10

Introduction

The Holtek C compiler is based on ANSI C. Due to the architecture of the Holtek microcontroller, only a subset of ANSI C is supported. This chapter describes the C programming language supported by the Holtek C compiler.

This chapter covers the following topics:

- C program structure
- Identifiers
- Data types
- Constants
- Operators
- Program control flow
- Functions
- Pointers and arrays
- Structures and unions
- Preprocessor directives
- Holtek C language extensions and restrictions

C Program Structure

A C program is a collection of statements, comments, and preprocessor directives.

Statements

Statements, which may consist of variables, constants, operators and functions, are terminated with a semicolon and perform the following operations:

- Declare data variables and data structures
- Define data space
- Perform arithmetic and logical operations
- Perform program control operations

One line can contain more than one statement. Compound statements are one or more statements contained within a pair of braces and can be used as a single statement. Some statements and preprocessor directives are required in the Holtek C source files. The following is a shell:

```
void main()
{
/* user application source code */
}
```

The *main* function is defined within the user application source code. There may be more than one source file for an application, but only one source file can contain the *main* function.

Comments

Comments are used to document the meaning and operation of the source statements and can be placed anywhere in a program except for the middle of a C keyword, function name or variable name. The C compiler ignores all comments. Comments cannot be nested. The Holtek C compiler supports two kinds of comments, block comment and line comment.

→ **Block Comment**

The block comment begins with **/*** and ends with ***/**, for example:

```
/* this is a block comment */
```

A block comment′s end character */ may be placed in a different line from the beginning block comment characters. In this case all the characters between the starting comment characters and end comment characters, are treated as comments and ignored by the C compiler.

→ **Line Comment**

A line comment begins with **//** and comments out all characters to the end of the line, for example

```
// this is a line comment
```

Identifiers

The name of an identifier contains a sequence of letters, digits, and under scores with the following rules:

- The first character must not be a digit
- Only the first 31 characters are significant
- Upper case and lower case letters are different
- Reserved words cannot be used

Reserved Words

The following are the reserved words supported by the Holtek C compiler. They must be in lower case.

auto	bit	break	case	char
const	continue	default	do	else
enum	extern	for	goto	if
int	long	return	short	signed
static	struct	switch	typedef	union
unsigned	void	volatile	while	

The reserved words **double, float** and **register** are not supported by the Holtek C compiler.

Data Types

Data Types and Sizes

Four basic data types are supported by the Holtek C compiler,

bit	a single bit
char	a single byte holding one character
int	an integer occupying one byte
void	an empty set of values, used as the type returned by functions that generate no value

The following qualifiers are allowed

Qualifier	Applicable Data Type	Use
const	any	place the data in a ROM space
long	int	create a 16-bit integer
short	int	create an 8-bit integer
signed	char, int	create a signed variable
unsigned	char, int	create an unsigned variable

The following are the data types, sizes and ranges.

Data Type	Size (bits)	Range
bit	1	0,1
char	8	–128~127
unsigned char	8	0~255
int	8	–128~127
unsigned	8	0~255
short int	8	–128~127
unsigned short int	8	0~255
long	16	–32768~32767
unsigned long	16	0~65535

Declaration

Variables must be declared before being used as this defines the data type and the size of the variable. The syntax of variable declaration is:

```
data_type variable_name [,variable_name...];
```

where *data_type* is a valid data type and *variable_name* is the name of the variable. The variables declared in a function are private (or local) to that function and other functions cannot access these variables directly. The local variables in a function exist and are valid only when this function is called, and are non-valid when exiting from the function. If the variable is declared outside of all functions, then it is global to all functions.

The qualifier **const** can be applied to a declaration of any variable, to specify that the value of the variable will not be changed. The variables declared with **const** are placed within the ROM space. The **const** qualifier can be used in array variables. A **const** variable must be initialized upon declaration, followed by an equal sign and an expression. Other variables cannot be initialized when declared.

A variable can be declared in a specified RAM address by using the @ character; the syntax is:

```
data_type variable_name @ memory_location;
```

The *memory_location* specifies the address variable located. To allocate a variable above the RAM bank 0 in the multiple RAM banks MCU, you might specify the bank no. in the high byte of *memory_location*. You should check the data sheet of the Holtek MCUs to get the information of the available RAM space.

For example:

```
int v1 @ 0×40; // declare v1 in the RAM bank 0 offset 0×40
int v2 @ 0×160; // declare v2 in the RAM bank 1 offset 0×60
```

Also, an array can be declared in a specified location:

```
int port[8] @ 0×20;    // array port takes memory location
                       // 0×20 through 0×27
```

All variables implemented by the Holtek C compiler are static unless they are declared as external variables. Note that both static and external variables will not be initialized to zero by default.

Note Declaring a variable as unsigned type will get more efficient code than as signed.

Constants

A constant is any literal number, single character or character string.

Integer Constants

An integer constant is evaluated as int type, a long constant is terminated with l or L. Unsigned constants are terminated with a u or U, the suffix ul or UL indicates unsigned long. The value of an integer constant can be specified with the following forms:

Binary constant: preceding the number by 0b or 0B
Octal constant: preceding the number by 0 (zero)
Hexadecimal constant: preceding the number by 0x or 0X
Others not included above are decimal

Character Constants

A character constant is an integer, which is denoted by a single character enclosed by single quotes. The value of a character constant is the numeric value of the character in the machine′s character set. ANSI C escape sequences are treated as a single character constant.

Escape Character	Description	Hex Value
\a	alert (bell) character	07
\b	backspace character	08
\f	form feed character	0C
\n	new line character	0A
\r	carriage return character	0D
\t	horizontal tab character	09
\v	vertical tab character	0B
\\	backslash	5C
\?	question mark character	3F
\′	single quote (apostrophe)	27
\″	double quote character	22

String Constants

String constants are represented by zero or more characters (including the ANSI C escape sequences) enclosed in double quotes. A string constant is an array of characters and has an implied null (zero) value after the last character. Hence, the total required storage is one more than the number of the characters within the double quotes.

Enumeration Constants

Another method for naming integer constants is called enumeration. For example:

```
enum {PORTA, PORTB, PORTC} ;
```

defines three integer constants called enumerators and assigns values to them.

The enumeration constants have type **int** (-128~127).
An explicit integer value might be associated with an enumeration constants. For example,

```
enum {BIG=10, SMALL=20};
```

The first enumeration constant has the value 0 if no explicit value is specified.
Subsequent enumeration constants without explicit associations receive an integer value one greater than the value associated with the previous enumeration constant.

An enumeration can be named. For example:

```
enum boolean {NO, YES};
```

The first name (NO) in an **enum** statement has the value 0, the next has the value 1.

Operators

An expression is a sequence of operators and operands that specifies a computation. An expression follows the rules of algebra, may result in a value and may cause side effects. The order of evaluation of subexpressions is determined by the precedence and grouping of the operators. The usual mathematical rules for associativity and commutativity of operators may be applied only where the operators are really associative and commutative. The different types of operators are discussed in the following.

Arithmetic Operators

There are five arithmetic operators,

+	addition
–	subtraction
*	multiplication
/	division
%	modulus (the remainder of division, always positive or zero)

The modulus operator %, can only be used with integral data types.

Relational Operators

The relational operators compare two values and return either a TRUE or FALSE result based on the comparison.

>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to

Equality Operators

The equality operators are exactly analogous to the relational operators

= =	equal to
!=	not equal to

Logical Operators

The logical operators support the logical operations AND, OR and NOT. They create a TRUE or FALSE value. Expressions connected by && and || are evaluated from left to right. The evaluation stops as soon as the result is known. The numeric value of a relational or logical expression is 1 if the relation is true, and 0 otherwise. The unary negation operator ! converts a non-zero operand into 0 and a zero operand into 1.

&&	logical AND
||	logical OR
!	logical NOT

Bitwise Operators

There are six operators for manipulating bit-by-bit operations. The shift operators >> and << perform the right and left shifts of the left operand by the number of bit positions given by the right operand, which must be positive. The unary ~ yields the one′s complement of an integer, converts every 1-bit to a 0-bit and vice versa.

&	bitwise AND
|	bitwise OR
^	bitwise XOR
~	one′s complement
>>	right shift
<<	left shift

Assignment Operators

There are a total of 10 assignment operators for expression statements. For simple assignment, the equal sign is used with the value of the expression replacing the variable, in the left operand. This also provides a shortcut for modifying a variable by performing an operation on itself.

<var>	+ = <expr>	add the value of <expr> to <var>
<var>	- = <expr>	subtract the value of <expr> from <var>
<var>	* = <expr>	multiply <var> by the value of <expr>
<var>	/ = <expr>	divide <var> by the value of <expr>
<var>	% = <expr>	modulus, remainder when<var>is divided by <expr>
<var>	& = <expr>	bitwise AND <var> with the value of <expr>
<var>	| = <expr>	bitwise OR <var> with the value of <expr>
<var>	∧ = <expr>	bitwise XOR <var> with the value of <expr>
<var>	>> = <expr>	right shift <var> by <expr> positions
<var>	<< = <expr>	left shift <var> by <expr> positions

Increment and Decrement Operators

The increment and decrement operators can be used in a statement by themselves, or can be embedded within a statement with other operators. The position of the operator indicates whether the increment or decrement is to be performed before (prefix operators) or after (postfix operators) the evaluation of the statement it is embedded within.

++ <var>	pre-increment
<var> ++	post-increment
––<var>	pre-decrement
<var>––	post-decrement

Conditional Operators

The conditional operator **?:** is a shortcut for executing a statement between two selectable statements according to the result of the expression.

<*expr*> ? <*statement1*> : <*statement2*>

If <*expr*> evaluates to a nonzero value, <*statement1*> is executed. Otherwise, <*satement2*> is executed.

Comma Operator

A pair of expressions separated by a comma is evaluated from left-to-right and the value of the left expression is discarded. All side effects of the left expression are performed before the evaluation of the right expression. The type and value of the result are the type and value of the right operand. For example,

```
    f(a, (t=3, t+2), c) ;
```

has three arguments, the second of which has the value 5.

Precedence and Associativity of Operators

The following table lists the precedence and associativity of operators. The precedence is from the highest to the lowest. Each box holds operators with the same precedence. Unary and assignment operators are right associative, all others are left associative.

Operators	Description	Associativity
[ ]	subscription	left to right
( )	parenthesis	
–>	structure pointer	
.	structure member	
sizeof	size of type	

Operators	Description	Associativity
++	increment	right to left
––	decrement	
~	complement	
!	not	
–	unary minus	
+	unary plus	
&	address of	
*	dereference	
*	multiply	left to right
/	divide	
%	modulus (remainder)	
+	add (binary)	left to right
–	subtract (binary)	
<<	shift left	left to right
>>	shift right	
<	less than	left to right
<=	less than or equal to	
>	greater than	
>=	greater than or equal to	
==	equal	left to right
!=	not equal	
&	bitwise AND	
^	bitwise XOR (exclusive OR)	
|	bitwise OR	
&&	logical AND	
||	logical OR	
?:	conditional expression	
=	simple assignment	right to left
*=	multiply and assign	
/=	divide and assign	
%=	modulus and assign	
+=	add and assign	
–=	subtract and assign	
<<=	left shift and assign	
>>=	right shift and assign	
&=	bitwise AND and assign	
|=	bitwise OR and assign	
^=	bitwise XOR and assign	
,	comma	left to right

Type Conversions

The general rule for type conversion is to convert a "narrower" operand into a "wider" one without losing information, such as converting an integer into a long integer. The conversion from **char** to **long** is sign extension. Explicit type conversion can be forced in any expression, with a unary operator called a cast. In the example:

```
    (type-name) expression
```

the `expression` is converted to the named type.

Program Control Flow

The statements in this section are used to control the flow of execution in a program. The use of relational and logical operators with these control statements and how to execute loops are also described.

→ **if-else statement**

- Syntax

```
if (expression)
statement1;
[else
statement2;
]
```

- Description

  The **`if-else`** statement is a conditional statement. The block of statements executed depends on the result of the condition. If the result of the condition is nonzero, the block of its associated statements is executed. Otherwise, the block of statements associated with the **`else`** statement is executed if the **`else`** block exists. Note that the **`else`** statement and its block of statements may not exist as it is optional.

- Example

```
if (word_count > 80)
{
    word_count=1;
    line++;
}
else
    word_count++;
```

→ **for statement**

- Syntax

```
for(initial-expression; condition-expression;
update-expression)statement;
```

- Description

  The *`initial-expression`* is executed first and only once. It is used to assign an initial value to a loop counter variable. This loop counter variable must be declared before the for loop.
  The *`condition-expression`* is evaluated prior to each execution of the loop. If the *`condition-expression`* is evaluated to be nonzero, the statement in the loop is executed. Otherwise, the loop exits and the first statement encountered after the loop is executed next. The *`update-expression`* executes after the statement of the loop.
  The **`for`** statement is used to execute a statement or block of statements repeatedly.

- Example

```
for (i=0;i<10;i++)
    a[i]=b[i]; // copy elements from an array to another array
```

→ **while statement**

- Syntax

  ```
  while (condition-expression)
      statement;
  ```

- Description

  The **`while`** statement is another kind of loop. When the *`condition-expression`* is nonzero, the while loop executes the *`statement`*. The *`condition-expression`* is checked prior to each execution of the *`statement`*.

- Example

  ```
  i=0;
  while (b[i] !=0)
  {
      a[i]=b[i];
      i++;
  }
  ```

→ **do-while statement**

- Syntax

  ```
  do
      statement;
  while (condition-expression);
  ```

- Description

  The **`do-while`** statement is another kind of while loop. The *`statement`* is always executed before the *`condition-expression`* is evaluated. Hence, the *`statement`* executes at least once, then checks the *`condition-expression`*.

- Example

  ```
  i=0;
  do
  {
      a[i]=b[i];
      i++;
  }while (i<10);
  ```

→ **break and continue statement**

- Syntax

  ```
  break;
  continue;
  ```

- Description

  The **`break`** statement is used to force an immediate exit from **`while`**, **`for`**, **`do-while`** loops and **`switch`**. The **`break`** statement bypasses normal termination and returns control to the previous nesting level if a **`break`** occurs within a nested loop.

  The **`continue`** statement orders the program to skip to the end of the loop and begins the next iteration of the loop. In the **`while`** and **`do-while`** loops, the continue statement forces the *`condition-expression`* to be executed immediately. In the **`for`** loop, control passes to the *`update-expression`*.

- Example

```
char a[10],b[10],i,j;
for (i=j=0;i<10;i++)// copy data from b[ ] to a[ ],skip blanks
{
    if (b[i]==0) break;
    if (b[i]==0×20)continue;
    a[j++]=b[i];
}
```

→ **goto statement and label**

- Syntax

```
goto label;
```

- Description

  A label has the same form as a variable name, but followed by a colon. The scope of a label is the entire function.

- Example

  See the **switch** statement example

→ **switch statement**

- Syntax

```
switch (variable)
{
    case constant1:
        statement1;
        break;
    case constant2:
        statement2;
        goto Label1;
    case constant3:
        statement3;
        break;
    default:
        statement;
Label1: statement4;
        break;
}
```

- Description

  The **switch** variable is tested against a list of constants. When a match is found, the statements with that constant are executed until a **break** statement is encountered. If no **break** statement exists, execution flows through the rest of the statements until the end of the **switch** routine. If no match is found, the statements associated with the **default** case are executed. The **default** case is optional.

  The **if-else** statement can be used to select between a pair of alternatives, but becomes cumbersome when many alternatives exist. The **switch** statement is an alternative multi-way decision method that evaluates if an expression matches one of many alternatives, and branches accordingly. It is equivalent to multiple **if-else** statements.

  The **switch** statement's limitation is that the **switch** variable must be an integral data type, and can only be compared against constant values.

• Example

```
for (i=j=0;i<10;i++)
{
    switch (b[i])
    {
        case 0: goto outloop;
        case 0×20:break;
        default:
            a[j]=b[i];
            j++;
            break;
    }
}
outloop:
```

Functions

In the C language, all executable statements must reside within a function. Before a function is used or called, it must be either defined or declared, otherwise a warning message will be issued by the C compiler. Two syntax forms, namely classic and modern, are supported for function declaration and definition. Unlike the variable, there in no need and no way to assign a function in a specific bank for the MCU having multi-bank of ROM. The Cross Linker will locate functions into a appropriate ROM bank.

Classic Form

```
return-type function-name (arg1, arg2,...)
var-type arg1;
var-type arg2;
```

Modern Form

```
return-type function-name (var-type arg1, var-type arg2, ...)
```

In both forms, the `return-type` is the data type of the function returned value. If functions do not return values, then `return-type` must be declared as **void**. The `function-name` is the name of this function and is equivalent to a global variable of all other functions. The arguments, *arg1*, *arg2*, etc, are the variables to be used in this function. Their data type must be specified. These variables are defined as formal parameters to receive values when the function is called.

→ **Function Declaration**

```
// classic form
  return-type function-name (arg1, arg2, ...);
// modern form
  return-type function-name (var-type arg1, var-type arg2,...);
```

→ **Function Definition**

```
// classic form
return-type function-name (arg1, arg2, ...)
var-type arg1;
var-type arg2;
{
    statements;
}
// modern form
return-type function-name (var-type arg1, var-type arg2, ...)
{
    statements;
}
```

→ **Passing Arguments to Functions**

There are two methods for passing arguments to functions.

- Pass by value.
  This method copies the argument values to the corresponding formal parameters of the function. Any changes to the formal parameters will not affect the original values of the corresponding variables in the calling routine.
- Pass by reference.
  In this method, the address of the argument is copied to the formal parameters of the function. Within the function, the formal parameters can access the actual variables within the calling routine. Hence, changes to the formal parameters can be made to the variables.

→ **Returning Values From Functions**

By using the **return** statement, a function can return a value to the calling routine. The returned value must be of a data type specified within the function definition. If `return-type` is **void**, it means no return value, therefore no value should be in the **return** statement. When a **return** statement is encountered, the function returns immediately to the calling routine. Any statements after the **return** statement are not executed.

Pointers and Arrays

Pointers

A pointer is a variable that contains the address of another variable. For example, if a pointer variable, namely varpoint, contains the address of a variable var, then varpoint points to var. The syntax to declare a pointer variable is

```
    data-type *var_name;
```

The `data-type` of a pointer is a valid C data type. It specifies the type of variable that `var_name` points to. The asterisk (*) prior to `var_name` tells the C compiler that `var_name` is a pointer variable. Two special operators, the asterisk (*) and ampersand (&), are associated with pointers. The address of a variable can be accessed by preceding this variable with the & operator. The * operator returns the value stored at the address pointed to by the variable.

In addition to * and &, there are four operators that can be applied to the pointer variables: +, ++, -, . Only integer quantities may be added or subtracted from pointer variables. An important point to remember when performing pointer arithmetic is that the value of the pointer is adjusted according to the size of the data type it is pointing to.

Arrays

An array is a list of variables that are of the same type and which can be referenced by the same name. An individual variable in the array is called an array element. The first element of an array is defined to be at an index of 0 and the last element is defined to be at an index of the total elements minus one. C stores one-dimensional arrays in contiguous memory locations. The first element is at the lowest address. C does not perform boundary checking for arrays.

Assignment from an entire array to another array is not allowed. To copy, each individual element must be copied one by one from the first array into the second array. Any array element can be used anywhere a variable or a constant can be used.

Structures and Unions

→ **Structures**

- Syntax
```
struct struct-name
{
    data-type member1;
    data-type member2;
    ...
    data-type membern;
} [variable-list];
```
- Description
A structure is a collection of one or more variables, possibly of different types, grouped together under a single name for convenient handling. Structures may be copied and assigned to, passed to functions and returned by functions. C allows bit fields. Nested structures are also allowed.
The reserved word **struct** indicates a structure is to be defined while *struct-name* is the name of the structure. Within the structure, *data-type* is one of the valid data types. Members within the structure may have different data types. The *variable-list* declares variables of the type *struct-name*. Each item in the structure is referred to as a member.
After defining a structure, other variables of the same type are declared with the following syntax:
```
struct struct-name variable-list;
```
To access a member of a structure, specify the name of the variable and the name of member separated by a period. The syntax is
```
variable.member1
```
where *variable* is the variable of structure type and *member1* is a member of the structure. A structure member can have a data type with a previously defined structure. This is referred to as a nested structure.

- Example
```
struct person_id
{
    char id_num[6];
    char name[3];
    unsigned long birth_date;
} mark;
```

→ **Unions**

```
union union-name
{
    data-type member1;
    data-type member2;
    ...
    data-type memberm;
} [variable-list];
```

- Description

  Unions are a group of variables of differing types that share the same memory space. A union is similar to a structure, but its memory usage is very different. In a structure, all the members are arranged sequentially. In a union, all members begin at the same address, making the size of the **`union`** equals to the size of the largest member. Accessing the members of a union is the same as accessing the members of a structure.

  `union` is a reserved word and *`union-name`* is the name of the union. The *`variable-list`*, which is optional, contains the variables that have the same data type as *`union-name`*.

- Example

```
union common_area
{
    char name[3];
    int id;
    long  date;
} cdata;
```

Preprocessor Directives

The preprocessor directives give general instructions on how to compile the source code. It is a simple macro processor that conceptually processes the source codes of a C program before the compiler properly parses the source program. In general, the preprocessor directives do not translate directly into executable code. It removes preprocessor command lines from the source file and expands macro calls that occur within the source text and adds additional information, such as the **#line** command, on the source file. The preprocessor directives begin with the **#** symbol. A line that begins with a **#** is treated as a preprocessor command, and is followed by the name of a command. The following are the preprocessor directives:

→ **Macro Substitution: #define**

- Syntax

```
#define name        replaced-text
#define name [(parameter-list)] replaced-text
```

- Description

  The **`#define`** directive defines string constants that are substituted into a source line before the source line is evaluated. The main purpose is to improve source code readability and maintainability. If the `replaced-text` requires more than one line, the backslash (\) is used to indicate multiple lines.

- Example

```
#define TOTAL_COUNT    40
#define USERNAME       "Henry"
#define MAX(a,b)       (((a)>(b))?(a):(b))
#define SWAP(a,b)      {int tmp;\
                       tmp=b;\
                       b=a;\
                       a=tmp;}
```

→ **#error**

- Syntax

```
#error "message-string"
```

- Description

  The **#error** directive generates a user-defined diagnostic message, *message-string*.

- Example

```
#if     TOTAL_COUNT > 100
#error      "Too many count."
#endif
```

→ **Conditional Compilation: #if #else #endif**

- Syntax

```
#if expression
    source codes1
[#else
    source codes2]
#endif
```

- Description

  The **#if** and **#endif** directives pairs are used for conditionally compiling code depending upon the evaluation of the *expression*. The **#else** which is optional provides an alternative compilation method. If the expression is nonzero, then the *source codes1* will be compiled. Otherwise, the *source codes2*, if it exists, will be compiled.

- Example

```
#define MODE 2
#if MODE > 0
    #define DISP_MODE  MODE
#else
    #define DISP_MODE 7
#endif
```

→ **Conditional Compilation: #ifdef**

- Syntax

```
#ifdef symbol
    source codes1
[#else
    source codes2]
#endif
```

- Description

  The **#ifdef** directive is similar to the **#if** directive, except that instead of evaluating the expression, it checks if the specified *symbol* has been defined or not. The **#else** which is optional provides alternative compilation. If the *symbol* is defined, then the *source codes1* will be compiled. Otherwise, the *source codes2*, if it exists, will be compiled.

- Example

```
#ifdef DEBUG_MODE
#define TOTLA_COUNT 100
#endif
```

→ **Conditional Compilation: #ifndef**

- Syntax

```
#ifndef symbol
    source codes1
[#else
    source codes2]
#endif
```

- Description

  The **`#ifndef`** directive is similar to the **`#ifdef`** directive. The **`#else`** which is optional provides alternative compilation. If the *`symbol`* has not been defined, then the *`source codes1`* will be compiled. Otherwise, the *`source codes2`*, if it exists, will be compiled.

- Example

```
#ifndef DEBUG_MODE
#define TOTAL_COUNT 50
#endif
```

→ **Conditional Compilation: #elif**

- Syntax

```
#if expression1
    source codes1
#elif expression2
    source codes2
[#else
    source codes3]
#endif
```

- Description

  The **`#elif`** directive is accompanied with the **`#if`** directive. It provides other compilation conditions in addition to the usual two. If the *`expression1`* is nonzero, then the *`source codes1`* will be compiled. If *`expression1`* is zero, then *`expression2`* is checked to see if it is nonzero. If so then the *`source codes2`* will be compiled. Otherwise, the *`source codes3`*, if it exists, will be compiled.

- Example

```
#if MODE==1
#define DISP_MODE 1
#elif MODE==2
#define DISP_MODE 7
#endif
```

→ **Conditional Compilation: defined**

- Syntax

```
#if defined symbol
    source codes1
[#else
    source codes2]
#endif
```

- Description

  The unary operator **`defined`** can be used within the directive **`#if`** or **`#elif`**. A control line of the form

  ```
  #ifdef symbol
  ```

  is equivalent to

  ```
  #if defined symbol
  ```

  A line of the form

  ```
  #ifndef symbol
  ```

  is equivalent to

  ```
  #if !defined symbol
  ```

- Example

  ```
  #if defined DEBUG_MODE
  #define TOTAL_COUNT 50
  #endif
  ```

→ **Conditional Compilation: #undef**

- Syntax

  `#undef` *`symbol`*

- Description

  The **`#undef`** directive causes the symbol's preprocessor definition to be erased. Once defined, a preprocessor symbol remains defined and in scope until the end of the compilation unit or until it is undefined using an **`#undef`** directive.

- Example

  ```
  #define TOTAL_COUNT 100
  ...
  #undef TOTAL_COUNT
  #define TOTAL_COUNT 50
  ```

→ **File Inclusion: #include**

- Syntax

  `#include` <*`file-name`*>
  or
  `#include` "*`file-name`*"

- Description

  `#include` inserts the entire text from another file at this point into the source file. When <*`file-name`*> is used, the compiler looks for the file in the directory specified by the environment variable INCLUDE. If the INCLUDE is not defined, the C compiler looks for the file in the path. When "*`file-name`*" is used, the C compiler looks for the file as specified. If no directory is specified, the current directory is checked.

- Example

  ```
  #include <ht48c10-1.h>
  #include "my.h"
  ```

Holtek C Language Extensions and Restrictions

Holtek C language provides a number of extensions for ANSI C. Most of these provide support for elements of the Holtek microcontroller architecture. Due to the limited resource of the microcontroller, there are also some restrictions you should take care.

Keywords

The following is a list of the keywords available in Holtek C.

@	**bit**	**norambank**	**rambank0**	**vector**

The following keywords and qualifiers are not supported:

double	**float**	**register**

Memory Bank

For variables located in high banks (not bank 0), they should be accessed through indirect addressing mode. To achieve the efficiencies, you might locate the most frequently used variables in Ram bank0. The Holtek C provides you a **`rambank0`** keyword to declare variables in bank0.

- Syntax
  ```
  #pragma rambank0
  //data declarations
  #pragma norambank
  ```
- Description
  The rambank0 keyword directs the Holtek C to declare subsequent variables to be located in Ram bank0 until the **`norambank`** keyword meets. For the single Ram bank MCU, these two keywords will be ignored.
- Example
  ```
  #pragma rambank0
  unsigned int i,j;        //i, j located in Ram bank0
  long len;                //len located in Ram bank0

  #pragma norambank
  unsigned int iflag;      //iflag's bank number is unknown

  #pragma rambank0
  int tmp;                 //tmp located in Ram bank0
  . . .
  i=1;                     //MOV A, 1
                           //MOV _i, A

  iflag=1                  //MOV A,BANK _iflag
                           //MOV[04H], A
                           //MOV A,OFFSET _iflag
                           //MOV[03H],A
                           //MOV A,1
                           //MOV[02H],A
  ```

Bit Data Type

Holtek C provides you with a bit data type which may be used for variable declarations, argument lists, and function return values. A bit variable is declared just as other C data types are declared. For the multiple ram/rom bank MCU, you should declare the bit variables in the ram bank 0 (**`#pragma rambank0`**).

- Example

```
#pragma rambank0
bit test_flag;       //bit var should locate in rambank0

bit testfunc(        //bit function
 bit f1,             //bit arguments
 bit f2)
{
     . . .
     return 0;       //return bit value
}
```

- Restriction
  – To get the benefit of the bit data type, it is not recommended to declare a bit array variable.
  – There is no bit pointer.

In Line Assembly

- Syntax

```
#asm
[label:] opcode [operands]
. . .
#endasm
```

- Description
  The **`#asm`** and **`#endasm`** are the inline assembly preprocessor directives. The **`#asm`** directive inserts Holtek's assembly instruction(s) after this directive (or within the directive **`#asm`** and directive **`#endasm`**) into the output file directly.
- Example

```
// convert low nibble value in the accumulator to ASCII
#asm
; this is an inline assembly comment
and a, 0fh
sub a, 09h
sz c
add a, 40h-30h-9
add a, 30h+9
#endasm
```

Interrupt

The Holtek C language provides a means for implementing interrupt service routines (ISRs) through the preprocessor directive **`#pragma`**. The directive **`#pragma vector`** is used to declare the name and address of the ISRs. Any function declared later with the same name as defined with **`#pragma vector`** is the ISR for the vector. The return statement within the ISR generates a **RETI** instruction.

- Syntax
  #pragma vector *symbol @ address*
- Description
  symbol is the name of the interrupt service routine.
  address is the interrupt address. The reset vector (address 0) is reserved for *main* function.
- Restriction
  There are four restrictions you should keep in mind when writing an ISR.
  - There is no parameter for ISR; the return type is **void**.
  - The ISR is not reentrant. Do not enable the interrupt in the ISR.
  - Do not call the ISR explicitly in your programs. It should always be invoked implicitly by the system while the interrupt coming.
  - Do not call the user defined function written in C within the ISR. It is safe to use the system calls. If you want to call a function in the ISR, then write it in assembly. It is safety to call the built-in function in the ISR.
- Example

```
#pragma vector timer0 @ 0x8
extern void ASM_FUNCTION();
void setbusy(){
. . .
}

void timer0(){
    . . .
    ASM_FUNCTION();      //The ASM_FUNCTION should be
                         //an assembly function

    _delay(3)            //Ok; built-in function

    setbusy();           //Wrong! do not call C function
}
```

Variables

The operator ″@″ can be used to specify the address of variable in the Data Memory.

- Syntax
  data_type varaible_name **@** *memory_location*
- Description
  The *memory_location* specifies the address of the variable located. For a single bank of RAM/ROM, the *memory_location* is one byte. For multiple banks of RAM/ROM, the *memory_location* is two bytes, the high byte is the bank number. The data sheet of the Holtek microcontrollers should be referred to for information on the available RAM space.
- Example

```
    int v1 @ 0x5B;       //declare v1 in the RAM bank 0 offset 0x5B
    int v2 @ 0x2F0;      //declare v2 in the RAM bank 2 offset 0xF0
```

Static Variables

Holtek C supports file scope static variables while local static variables does not.

• Example

```
static  i;           //file scope static
void f1(){
     i=1;            //OK
}
void f2(){
     static int j;   //Wrong
                     //local static variable is not supported
     . . .
}
```

Constants

Holtek C supports binary constants. Any string that begins with 0b or 0B will be treated as a binary constant.

• Example

```
0b101= 5
0b1110= 14
```

Functions

Avoid using reentrant and recursive code.

Arrays

An array should be located in a contiguous block of memory and must not have more than 256 elements. To speak precisely, the size of an array is limited to the size of the RAM bank of the Holtek MCU you used.

Constant Variables

Constant variables must be declared in global scope and be initialized when declared. A constant variable could not be declared as external.

A constant array would specify the array size otherwise an error generated.

```
const char carray[]= {1,2,3};   //wrong
const char carray[3]= {1,2,3};  //right
```

A constant string must be used in the C file with the main function.

```
//test.c
void f1(char *s);
void f2(){
     f1(″abcd″);      //″abcd″ is a constant string
                      // If there is no main() function declared
                      // in test.c then the Holtek C compiler would
                      // generate an error.
     . . .
}
. . .
void main(){
     . . .
}
```

Pointer

Pointer can not apply to constant and bit variables

Initial Value

Global variables cannot be initialized when declared. Local variables do not have this constraint. Constant variables must be initialized when declared.

• Example

```
unsigned int i1= 0;          //illegal declaration; can not be
                             //initialized
unsigned int i2;
const unsigned int i3;       //illegal declaration; should be
                             //initialized
const unsigned int i4=5;
const char a1[5];            //illegal declaration; should be
                             //initialized
const char a2[5]={0×1,0×2,0×3,0×4,0×5};
const char a3[4]=″abc″;      //={′a′, ′b′, ′c′,0}
const char a4[3]=″abc″;      //={′a′, ′b′, ′c′}
const char a5[2]=″abc″;      //array size mismatched
```

Multiply/Divide/Modulus

The multiply, divide and modulus (″*″, ″/″, ″%″) operators are implemented by system calls.

Built-in Functions

• WDT & halt & nop

C system call	Assembly code
void _clrwdt( )	CLR WDT
void _clrwdt1( )	CLR WDT1
void _clrwdt2( )	CLR WDT2
void _halt( )	HALT
void _nop( )	NOP

- Rotate right/left

```
void _rr(int*);        //rotate 8 bits data right
void _rrc(int*);       //rotate 8 bits data right through carry
void _lrr(long*);      //rotate 16 bits data right
void _lrrc(long*);     //rotate 16 bits data right through carry
void _rl(int*);        //rotate 8 bits data left
void _rlc(int*);       //rotate 8 bits data left through carry
void _lrl(long*);      //rotate 16 bits data left
void _lrlc(long*);     //rotate 16 bits data left through carry
```

- Swap nibble

```
void _swap(int *);     //swap nibbles of 8 bits data
```

- Delay cycle

```
void _delay(unsigned long);   //delay n instruction cycle
```

The _delay function forces the MCU to execute the specified cycle count. A value of zero causes an endless loop. The parameter of the _delay could be constant value only. It does not accept a variable.

- Example 1

```
//assume the watch dog timer is enabled
//and is using one instruction
void error(){
        _delay(0);   //infinite loop, same as while(1);
}
void dotest(){
        unsigned int ui;
        ui = 0x1;
        _rr(&ui);   //rotate right
        if (ui != (unsigned int)0x80) error();
        ui = 0xab;
        _swap(&ui);
        if (ui != (unsigned int)0xba) error();
}
void main(){
        unsigned int i;
        for(i= 0; i<100; i++){
                _clrwdt();
                _delay(10);  //delay 10 instruction cycle
                dotest();
        }
}
```

- Example 2

```
//assume the watch dog timer is enable
//and using two instructions
void do test(){
...
}
void main(){
        unsigned int i;
        for(i= 0; i<100; i++){
                _clrwdt1();
                _clrwdt2();
                dotest();
        }
}
```

Stack

Because the Holtek microcontrollers have limited stack depth, it is necessary to consider the function call depth to avoid stack overflow. The multiply, divide, modulus, and const variables are implemented by using ″call″ instructions, each taking one stack level.

Operator/System Function	Stack Requirements
`main ( )`	0
`_clrwdt( )`	0
`_clrwdt1( )`	0
`_clrwdt2( )`	0
`_halt( )`	0
`_nop( )`	0
`_rr(int*)`	0
`_rrc(Int*)`	0
`_lrr(long*)`	0
`_lrrc(long*)`	0
`_rl(int*)`	0
`_rlc(int*)`	0
`_lrl(long*)`	0
`_lrlc(long*)`	0
`_swap(int*)`	0
`_delay(unsigned long)`	1
`*`	1
`/`	1
`%`	1
`constant array`	1

Chapter 11

Mixed Language

11

The Holtek Cross Tools (Cross Assembler, Cross Linker, Library and Holtek C compiler) provide methods to program with mixed languages, Holtek assembly language and C language. That means a project can consist of source files programming with assembly language and C language. However, the programmer should conform to some rules when programming with mixed languages. In order to facilitate the program coding, this chapter describes the conventions that Holtek C compiler compiles a C program into the assembly language, how to define the subroutine names, etc. The following are the topics included:

- Little endian
- Naming rule of functions and parameters
- Parameter passing
- Return value
- Preserving registers
- Calling assembly function from C program
- Calling C function from assembly program
- Programming ISR with assembly language

Little Endian

The data format adopted by the Holtek C compiler is Little-Endian, i.e. the low byte of a WORD is the WORD′s least significant byte, and the high byte is the most significant. In memory allocation, the low byte occupies the lower address and high byte occupies the higher address.

For example

```
long var @ 0x40;
var = 0x1234;
```

Then the address 0x40 contains 0x34, and the address 0x41 contains 0x12.

Naming Rule of Functions and Parameters

The Holtek Cross Assembler is non case-sensitive when handling symbol names. Actually, all symbol names are translated into uppercase no matter what the original form is. But the Holtek C language is case-sensitive. Due to the difference of these two languages, the variables and functions which are defined in C source files and referred by the assembly program should be defined as uppercase.

The names of the global variables and functions in C language are prefixed with an underscore when the C compiler translates them into assembly language. For local variables, if a local variable is declared without being referenced, the C compiler won't reserve memory space for it. By checking the assembly file generated by the Holtek C compiler, the programmer can find out what the translated name of the C local variable is.

Global Variable

A global variable in a C file is translated into the same case letters with a prefixed underscore.

For example

```
TimerCt
TMP
```

will be translated into

```
_TimerCt
_TMP
```

Local Variable

If a local variable in a C function is not referenced by other programs, it will not be translated into assembly language. Check the assembly file to find out what the result is.

```
void main( ){
    int i, j, k;      ; k is not used
    long m;
    char c;
    i = j = m = c = 2;
    #asm
    set CR3[1].2      ;set bit 10 of m, i.e. m |= 0x400
    #endasm
}
```

The corresponding part of the assembly file looks like the following:

```
#line 2 "C:\Holtek IDE\SAMPLE\NAME.C"
LOCAL CR1 DB ? ; i
#pragma debug variable 2 CR1 i
#line 2 "C:\Holtek IDE\SAMPLE\NAME.C"
LOCAL CR2 DB ? ; j
#pragma debug variable 2 CR2 j
#line 3 "C:\Holtek IDE\SAMPLE\NAME.C"
LOCAL CR3 DB 2 DUP (?) ; m
#pragma debug variable 2 CR3 m
#line 4 "C:\Holtek IDE\SAMPLE\NAME.C"
LOCAL CR4 DB ? ; c
#pragma debug variable 2 CR4 c
```

The second and third line indicates that the *i* is translated into *CR1* in the assembly file. By the same way, *j* is translated into *CR2*, *m* is *CR3* and *c* is *CR4*. The *k* is not referenced so it is not translated.

Note If local variables are added to or removed from or arranged the order, then the translated names may be changed by the C compiler.

For the above sample code, if the microcontroller supports multiple RAM banks, then the instruction

```
set CR3[1].2
```

can execute correctly or not. The program will be corrupted if CR3 is allocated in a high bank. But this phenomenon cannot happen, because the local variable is defined with a **LOCAL** directive in the translated assembly file and instructs the Cross Assembler to allocate the variable in the RAM bank 0. Hence it can execute correctly in the same way as a variable does in a single RAM bank.

Function

Like the global variable, a function in a C file is translated into the same case letters with a prefixed underscore.

For example

```
GetKey
IsBusy
```

will be translated into

```
_GetKey
_IsBusy
```

Function Parameters

The names of the function parameters in a C file are translated into the function name following the number of the parameters occurring, indexed from 0.

For example

```
GetKey(int row, long col)
```

row is translated into GetKey0
col is translated into GetKey1

Parameter Passing

Due to the microcontroller resource′s limitation, the Holtek C compiler passes parameters to a function via the RAM space instead of the stack. The naming of the function parameters are the function name appended with the number of the parameters occurring, indexed from 0. Like the local variable, the function parameters are also allocated in RAM bank 0.

For example

```
void  function (int a, int b)
```

Then the parameter `a` will be translated into `function0, b` will be `function1`.

For mixed languages, the data type of the function parameters should always be declared as BYTE in assembly, if there are more than one byte, e.g. WORD (2 bytes), the programmer should use the instruction ″DB n DUP(?)″ to declare it.

Return Value

The return value of a C function is located in the **A** register or in the **RH** system variable. If the size of the return value is one byte (e.g. **char, unsigned char, int, unsigned int, short, unsigned short**), then the value is stored in the **A** register. If it is two bytes (e.g. **long, unsigned long**, pointer), then the high byte is stored in the **RH** and the low byte is stored in the **A** register.

Note The **RH** variable is located in RAM bank 0.

Preserving Registers

Except for the ISR, there is no need to preserve the registers when writing a function in assembly. If users write an ISR in assembly language, then it is their responsibility to preserve the registers used in the ISR.

Calling Assembly Function from C Program

This section describes the steps to call an assembly function from a C program. The steps are divided into two parts, one is for the assembly files, the other is for the C files.

→ **In Assembly File**

- Declare **RH** as an external byte variable if the return value is two bytes.
- Declare the function name with prefixed underscored as public.
- Declare the function parameters, if they exist, in the RAM bank 0 as public. Beware of the naming of parameters.
- Put the return values into **A** or **RH**.

→ **In C File**

- Declare the prototype of the external function name with uppercase letters
- Call it

The following function is defined in assembly file and called by a C program.

```
long  KEYIN(int row, long col);
```

In assembly file

```
;;Declare external byte variable RH
EXTERN RH:BYTE

;;Declare function name & parameters as public
PUBLIC _KEYIN, KEYIN0, KEYIN1

;;Declare parameters
RAMBANK0 KEYINDATA  ;suppose the MCU has multiple ram banks
KEYINDATA .section 'data'
KEYIN0 DB?              ;row
KEYIN1 DB 2 DUP (?)     ;col, don't use 'KEYIN1 DW ?'

;function body
CODE .section 'code'
_KEYIN:
. . .
MOV A, KEYIN0           ;retrieve row
. . .
MOV A, KEYIN1           ;retrieve low byte of col
. . .
MOV A, KEYIN1[1]        ;retrieve high byte of col
. . .
;; Put the return values into A and RH
MOV A,0A0H              ;suppose the return value is 0xA010
MOV RH,A                ; store high byte 0xA0 to RH
MOV A,10H               ; store low byte 0x10 to A
RET
```

In C file

```
// Declare the external function name with uppercase
extern long KEYIN(int row, long col);
long rc;
. . .
// Call it
rc = KEYIN(10, 20L);
```

Calling a C Function from an Assembly Program

This section describes the steps to call a C function from an assembly program. For the microcontroller with multiple ROM banks, it is important to set the BP (bank pointer) before calling the function.

→ **In the C File**

- Declare the function name with uppercase

→ **In the Assembly File**

- Declare **RH** as an external byte variable if the return value is two bytes.
- Declare the external function name with prefixed underscore
- Declare the function parameters as external if they exist. Beware of the naming of parameters.
- Set function parameters if they exist
- Call C function
  Call the C function directly if the microcontroller supports a single RAM/ROM bank.
  Set BP to the bank of function first, then call the C function if the microcontroller supports multiple RAM/ROM banks.
- Get return value from **A** or **RH**

The following function is defined in C language and called by the assembly program

long KEYIN(int row, long col);

and the microcontroller has single ROM bank.

```
// -----------------------------------------------------------------
// In C file, function definition
// -----------------------------------------------------------------
long KEYIN(int row, long col){
. . .
}
void main( ){
. . .
}

;; -----------------------------------------------------------------
;; In assembly file
;; -----------------------------------------------------------------

;;Declare external byte variable RH
EXTERN RH:BYTE

;; Declare the external function name with prefixed
;; underscore
extern _KEYIN: near ;; underscore and function name

;; Declare the function parameters as external variables
extern KEYIN0:byte  ; function parameter:row
extern KEYIN1:byte  ; function parameter:col
                    ; declare it as BYTE, although
                    ; it's 2 bytes
code_ki .section 'code'

;; Set function parameters for calling  KEYIN(0x10, 0x200L)
mov a,10H
mov KEYIN0,a        ; put 10H to function parameter: row
mov a,2H
mov KEYIN1[1],a     ; put 02H to high byte of parameter:col
clr KEYIN1          ; put 00H to low byte of parameter:col
```

```
;; Call C function
call _KEYIN
;; Get return value from A or RH
;; A register keeps low byte of return value
;; RH keeps high byte of return value
```

The following function is defined with C language and called by the assembly program
long KEYIN(int row, long col);
and the microcontroller supports multiple ROM banks.

```
// -----------------------------------------------------------------
// In C file
// -----------------------------------------------------------------
long KEYIN(int row, long col){
. . .
}

;-----------------------------------------------------------------
;In assembly file
;-----------------------------------------------------------------
;;Declare external byte variable RH
EXTERN RH:BYTE

;; Declare the external function name with prefixed
;; underscore
extern _KEYIN:near

;; Declare the function parameters as external variables
extern KEYIN0:byte  ; parameter: row
extern KEYIN1:byte  ; parameter: col, although it's
                    ; 2 bytes,
                    ; only declare one BYTE

code_ki  .section  'code'

;;Set function parameters for calling  KEYIN(0x10, 0x200L)
mov  a,10
mov  KEYIN0,a       ; parameter: row
mov  a ,2
mov  KEYIN1[1],a    ; high byte of the parameter: col
clr  KEYIN1         ; low byte of the parameter: col

;; Call C function in multiple ROM banks
;; Set BP to the bank of function first

mov     a, bank _KEYIN
mov     bp , a      ; change the bank number
call    _KEYIN

;; Get return value from A or RH
;; A register keeps low byte of return value
;; RH keeps high byte of return value
```

Programming the ISR with Assembly Language

An ISR (Interrupt Service Routine) is invoked by a hardware interrupt. It should not be explicitly called by user, hence it doesn′t have parameters to be passed nor return values. When an ISR is written in assembly there is no correlation with other C files. It is only necessary to add the assembly file into the project. Please refer to the assembly language user′s guide for more information about ISR programming.

Do not call a C function from an ISR, no matter whether the ISR is written in assembly or C.

Chapter 12

Cross Linker

12

What the Cross Linker Does

The Cross Linker creates task files from the object files generated by the Cross Assembler or the C compiler. The Cross Linker combines both code and data in the object files and searches the named libraries to resolve external references to routines and variables. It also locates the code and data sections at the specified memory address or at the default address, if no explicit address is specified. Finally, the Cross Linker copies both the program codes and other information to the task file. It is this task file that is loaded by the Holtek IDE Holtek Integrated Development Environment, into the Holtek HT-ICE In-Circuit Emulator, for debugging. The libraries included by the Cross Linker were generated by the Holtek library manager.

Cross Linker Options

The options specify and control the tasks performed by Cross Linker. In chapter 3, Option Menu, Project command provides a dialog box, Cross Linker Options, to specify these options to the Cross Linker. These options are:

Libraries

- Syntax
  `libfile1[,libfile2...]`
  This option informs the Cross Linker to search the specified library files if the input object files refer to a procedure or variable which is not defined in any of the object files. If a module of a library file contains the referred procedure or variable, then only this module, not the whole library file will be included in the output task file. (refer to Chapter 13 Library Manager)

Section Address

- Syntax
  `section_name = address [,section_name = address]...`
  This option specifies the address of the sections; section_name is the name of the section that is to be addressed. The section_name must be defined in at least one input object file, otherwise a warning will occur. The address is the specified address whose format is xxxx in hexadecimal format.

Generate Map File

The check box of this option is to specify whether the map file is generated or not.

Map File

The map file lists the names and loads the addresses and lengths of all sections in a program as well as listing the messages it encounters. The Cross Linker gives the address of the program entry point at the end of the map file. The map file also lists the names and loads addresses of all public symbols. The names and file names of the external symbols or procedures are recorded in the map file if no corresponding public symbol or procedure can be found. The contents of the map file are as follows.

```
Holtek (R) Cross Linker Version 7.34
Copyright (C) HOLTEK Microelectronics INC. 2002-2003. All
rights reserved.
Input Object File: C:\SAMPLE\T2.OBJ
Input Object File: C:\SAMPLE\T1.OBJ
Start      End     Length     Class     Name
0000h      00F2h   00F3h      CODE      TEXT      (C:\SAMPLE\T1.OBJ)
00F3h      0114h   0022h      CODE      SUB       (C:\SAMPLE\T2.OBJ)
0000h      0063h   0064h      DATA      DAT       (C:\SAMPLE\T1.OBJ)

Address Public by Name
001Ch      BREAKL
00A4h      CHKSTACK
0042h      FAC_DB

Address Public by Value
001Ch      BREAKL
0042h      FAC_DB
00A4h      CHKSTACK

HLINK:  Program entry point at section code(address 0) of file
        C:\SAMPLE\T1.OBJ
<EOF>
```

Cross Linker Task File and Debug File

One of the Cross Linker's output files is the task file which consists of two parts, a task header and binary code. The task header contains the Cross Linker version, the MCU name and the ROM code size. The binary code part contains the program codes. The other Cross Linker output file is the debug file which contains all information referred to by the Holtek IDE debugging program. This information includes source file names, symbol names and line numbers as defined in the source files. The Holtek IDE will refer to the symbolic debugging function information. This file should not be deleted unless the debugging procedure is completed, otherwise the Holtek IDE will be unable to support the symbolic debugging function.

Part III

Utilities

In addition to the previously discussed general purpose 8-bit MCU development tools, Holtek also supplies several other utilities for its range of special purpose Voice and LCD MCU devices by supplying all the necessary tools and step by step guide for relevant simulation of voice synthesis and tone generator applications as well as the tools for real time hardware LCD panel simulation. This part contains all the information needed to program and debug relevant applications quickly and efficiently.

Chapter 13

Library Manager

13

What the Library Manager Does

The Library Manager provides functions to process the library files. The library files are utilized in the creation of the output file by the Cross Linker. A library is a collection of one or more object modules which are assembled or compiled and ready for linking. It stores the modules that other programs may require for execution.

By using the Library Manager, library files can be created. Object files including common routines may be added to the library files. Before creating these object files, the names of all common routines must be made public by using the assembly directive PUBLIC (refer to the chapter on Assembly Language and Cross Assembler). The Cross Assembler generates the output object file (.OBJ) while the Library Manager adds this object file into the specified library file. When the Cross Linker has found unresolved names in a program during the linking process, it will search the library files for these unresolved names, and extracts a copy of the module containing that name. If an unresolved name has been found in this library module, the module will be linked to the program.

To Setup the Library Files

The Library Manager provides the following functions:

- Create new library files
- Add/Delete a program module to/from a library file
- Extract a program module from a library file, and create an object file

To select use the Tools Menu and the Library Manager command as shown in Fig 13-1. Fig 13-2 shows the dialogue box for processing the functions of the Library Manager.

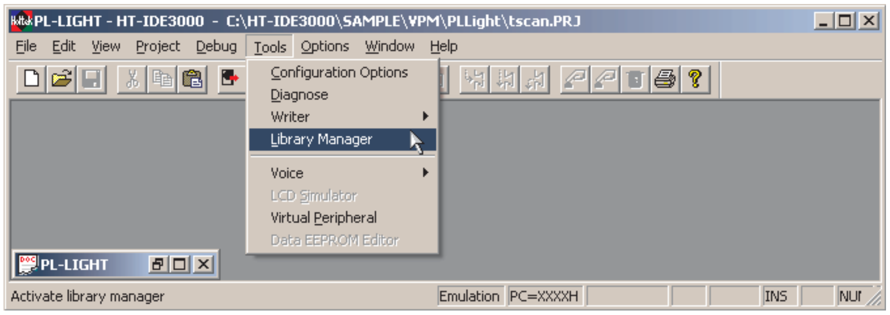
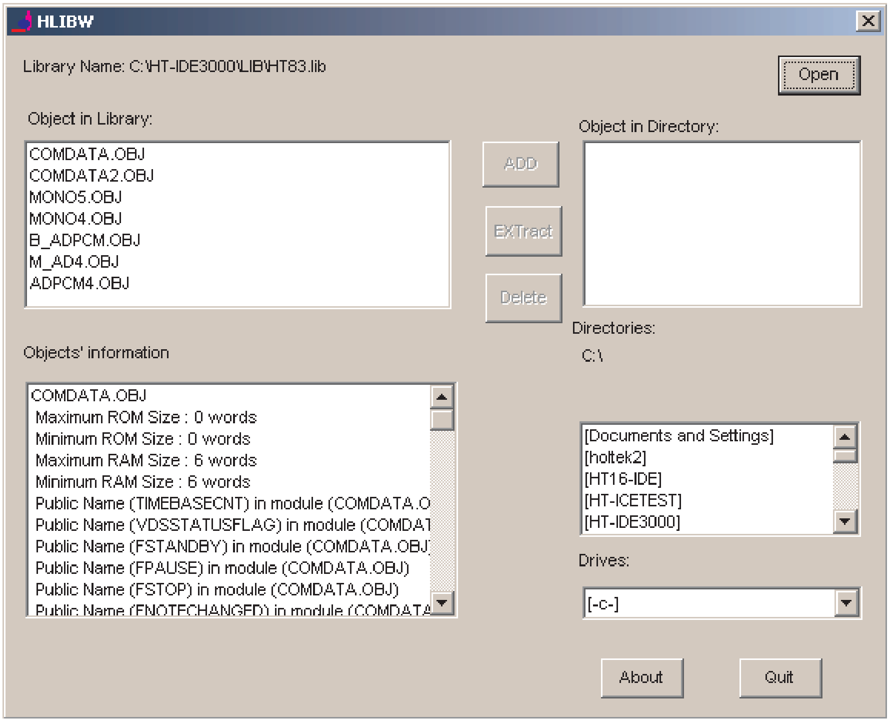

PL-LIGHT - HT-IDE3000 - C:\HT-IDE3000\SAMPLE\VPM\PLLight\tscan.PRJ
File Edit View Project Debug Tools Options Window Help
Configuration Options
Diagnose
Writer
Library Manager
Voice
LCD Simulator
Virtual Peripheral
Data EEPROM Editor
PL-LIGHT
Activate library manager
Emulation PC=XXXXH INS NUM

Fig 13-1



Library Name: C:\HT-IDE3000\LIB\HT83.lib

Object in Library:

COMDATA.OBJ
COMDATA2.OBJ
MONO5.OBJ
MONO4.OBJ
B_ADPCM.OBJ
M_AD4.OBJ
ADPCM4.OBJ

Object in Directory:

Objects' information

```
COMDATA.OBJ
 Maximum ROM Size : 0 words
 Minimum ROM Size : 0 words
 Maximum RAM Size : 6 words
 Minimum RAM Size : 6 words
 Public Name (TIMEBASECNT) in module (COMDATA.O
 Public Name (VDSSTATUSFLAG) in module (COMDAT
 Public Name (FSTANDBY) in module (COMDATA.OBJ
 Public Name (FPAUSE) in module (COMDATA.OBJ)
 Public Name (FSTOP) in module (COMDATA.OBJ)
 Public Name (FNOTECHANGED) in module (COMDATA
```

Directories:

C:\

[Documents and Settings]
[holtek2]
[HT16-IDE]
[HT-ICETEST]
[HT-IDE3000]

Drives:

[-c-]

Fig 13-2

Create a New Library File

Press Open button, Fig 13-3 is displayed

Type in a new library file name and press the OK button, Fig 13-4 is displayed for confirmation. If the Yes button is chosen, a new library file will be created but will not contain any program modules.

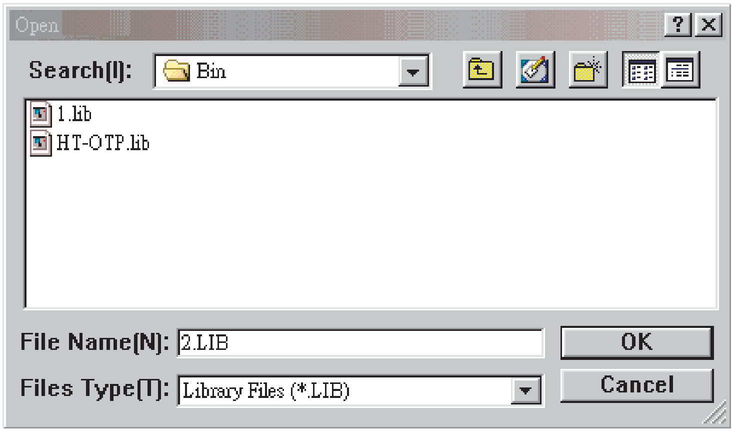


Fig 13-3

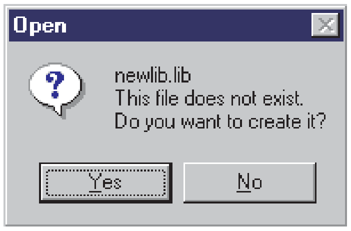

newlib.lib
This file does not exist.
Do you want to create it?

Fig 13-4

Add a Program Module into a Library File

Select an object module from the ″Object in Directory″ box, and press the [ADD] button to add this object module into this library file.

Delete a Program Module from a Library File

Select an object module from the ″Object In Library″ box, and press the [Delete] button to delete this object module from the library file.

Extract a Program Module from Library and Create An Object File

Select an object module from the ″Object in Library″ box, and press [ExTract] button. A file will then be created with the same name and same content as the selected object module. It is displayed on the ″Object in Directory″ box.

Object Module Information

Press the Open button, Fig 13-3 is displayed. Select a library file from the box below the File Name box, press OK button. From Fig 13-2, all the object modules of the selected library file are listed in the ″Object in Library″ box. The following information about each object module is also listed in the ″Objects′ Information″ box.

- Maximum ROM size
  The maximum size used by this object module program code. Dependent upon the code section align type.
- Minimum ROM size
  The minimum actual size used by this object module program code
- Maximum RAM size
  The maximum size used by this object module program data. It depends on the data section align type.
- Minimum RAM size
  The minimum, actual size used by this object module program data.
- Public Name
  The names of all public symbols in this object module.

Chapter 14

LCD Simulator

14

Introduction

The Holtek LCD simulator, known as the HT-LCDS, provides a mechanism allowing users to simulate the output of LCD drivers. According to the user designed patterns and the control programs, the HT-LCDS displays the patterns on the screen in real time. It facilitates the development process even if the actual LCD hardware panel is unavailable. Note that if the current project's microcontroller does not support LCD functions, these commands are disabled.

LCD Panel Configuration File

Before starting the LCD simulation, an LCD panel configuration file must first be setup. The HT-LCDS will obtain the LCD data and display LCD patterns on the screen according to the LCD panel configuration file. The HT-LCDS cannot simulate the LCD action if this file is absent. For microcontrollers possessing an LCD driver, the corresponding panel configuration file has to be setup for LCD simulation. The LCD simulator command within the Tools menu will then be enabled to setup the panel configuration file and for simulation (Fig14-1). The LCD panel configuration file contains two kinds of data, panel configuration data and pattern information, which users can setup using the HT-LCDS.

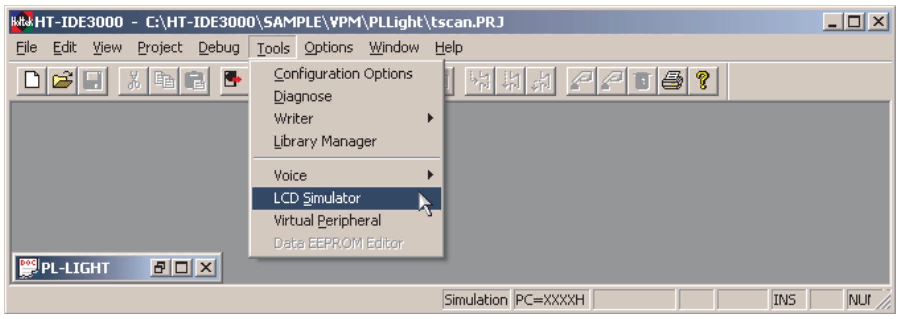


Fig 14-1

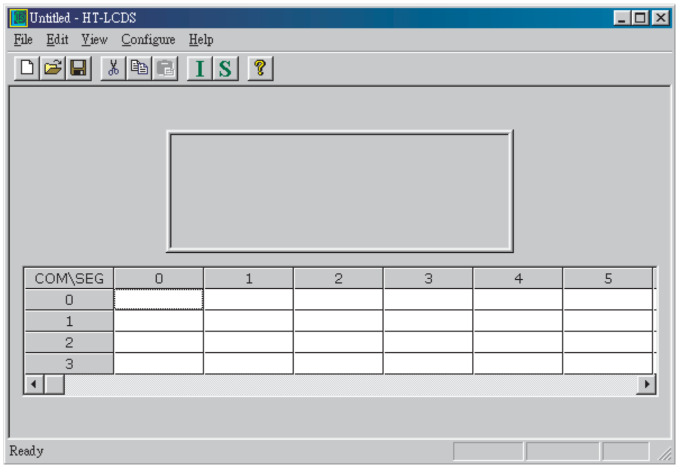
Relationship Between the Panel File and the Current Project

By default, the panel configuration file has the same file name as the current project name except for the extension name, which is .lcd. The HT-LCDS assumes this file to be the corresponding panel configuration file of the current project. The panel configuration file is generated by the HT-LCDS File menu, New command or the New button on the toolbar. A different file name from the current project name can be assigned to the panel configuration file by clicking File menu, Save command or Save button on the toolbar.

When the HT-LCDS begins simulation, it references the current active panel configuration file to obtain its simulation information. The LCD panel configuration file is activated by selecting the New or Open command of the HT-LCDS File menu. The file name of the LCD panel configuration file may be the same as the current project name or a different name can be chosen.

Selecting the HT-LCDS

When selected from within the Tools menu, the LCD simulator as shown in Fig 14-2 is displayed if the corresponding panel configuration file of the current project exists. The file name of each bitmap pattern is shown at the specified COM/SEG position of the table. At the same time, these patterns are shown on the above panel screen. If the corresponding panel configuration file does not exist within the project directory, both the panel screen and the COM/SEG table will not be displayed. Fig 14-3 shows the HT-LCDS menu bar information.



COM\SEG	0	1	2	3	4	5
0						
1						
2						
3						

Fig 14-2

The Fig below shows the HT-LCDS menu bar information.

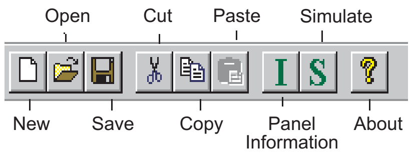


Fig 14-3

New: create a new panel configuration file
Open: open an existing panel configuration file
Save: save the panel configuration file
Cut: delete a pattern
Copy: copy a pattern to the clipboard
Paste: add the copied pattern to the panel
I: panel information dialog
S: enter the LCD simulation mode

LCD Panel Picture File

The LCD panel picture (pattern) file is a bitmap file (.bmp) which represents the practical patterns and their positions on the panel. The bitmap file can be created using any bitmap editor and provides another method of setting up the LCD panel pattern information by using the HT-LCDS Edit menu, Panel Editor command. The bitmap file is optional, users can setup the LCD panel pattern information even if the LCD panel picture file is absent.

Setup the LCD Panel Configuration File

The following two steps are used to setup a panel configuration file:

- Setup the panel configurations, including the segment and common number of the LCD driver as well as the width and height size of the panel in pixels. Also, the directory of the panel configuration file and the dot matrix mode can be selected.
- Select the patterns and their positions. This will setup the relationship between the patterns and the COM/SEG positions.

Setup the Panel Configurations

To setup the panel configurations by selecting the HT-LCDS File menu, New command. The Panel Configuration dialog box (Fig 14-4) will be displayed. Setup the correct LCD driver data, COM/SEG number, Width, Height and Directory of the pattern, then press the [OK] button. After setting up the panel configuration, the system returns to Fig 14-2 for pattern selection.

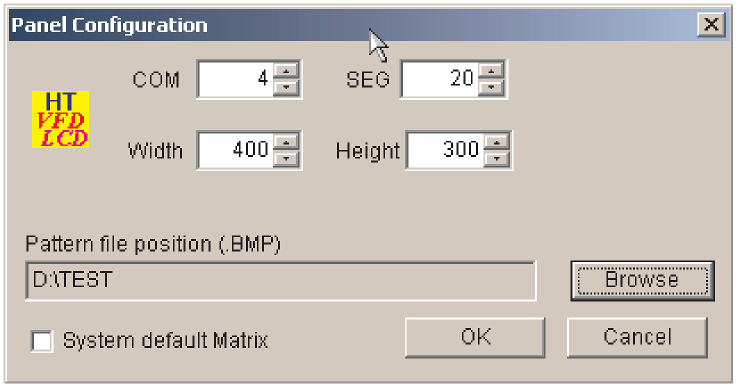

Panel Configuration
COM 4
SEG 20
Width 400
Height 300
Pattern file position (.BMP)
D:\TEST
☐ System default Matrix

Fig 14-4

The panel configurations include:

- COM and SEG. To set the LCD driver total COMMON number and SEGMENT number. The default number of the LCD driver for this microcontroller is displayed when Fig14-4 is displayed. To ensure that these numbers are the same as the actual setting number of the LCD driver for the micro controller.
- Width and Height. These are the size of the panel screen in pixels and can be changed to adjust the panel screen.
- Panel configuration file directory. Select the directory where the panel configuration file is stored using the browse button or setup to have the same directory as the project.
- Dot Matrix Mode. To simulate dot matrix type LCD panels. Fig 14-5 shows the dot matrix screen.

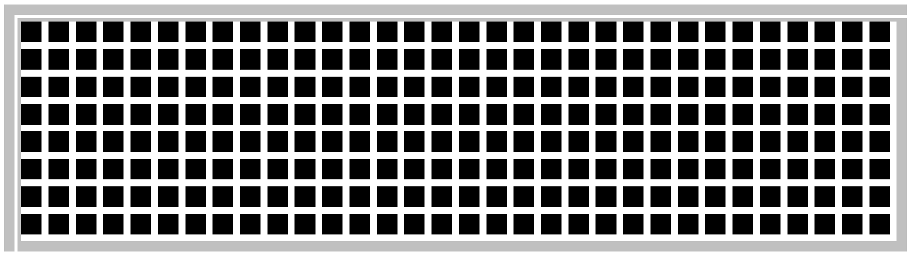


Fig 14-5

Note It is important not to set different COM or SEG number from the actual corresponding LCD driver numbers, otherwise unpredictable results will occur.

Select the Patterns and Their Positions

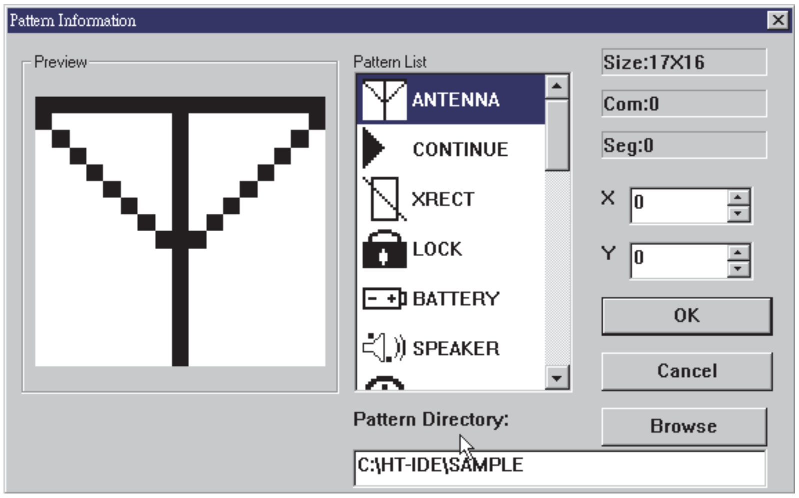
The following methods show the steps of selecting the patterns and their positions

- To create a new panel configuration file using the HT-LCDS File menu New command. After having set the panel configuration, Fig 14-2 is displayed. The user then has to select the patterns from the Pattern Information dialog box (Fig 14-6) and set the COM/SEG positions. The section, Add a new pattern, describes the procedure in detail.
- To open an existing panel configuration file using the HT-LCDS File menu Open command. The patterns are displayed as shown on the panel screen in Fig 14-2 and the pattern file names are displayed as shown in the Fig14-2 COM/SEG table position. Users can add/delete/change the pattern information, including the pattern file and pattern positions.
- To open a panel picture file using the HT-LCDS Edit menu Panel Editor command. If this panel picture file has been setup already, then it is not necessary to select the patterns, it is only necessary to select the pattern positions. The section, Define the pattern using the Panel Editor, describes the procedure in detail.

Add a New Pattern

- Move the cursor to a COM/SEG position on the grid as shown in Fig 14-2 and double click the mouse. The Pattern Information dialog box, as shown in Fig 14-6, is displayed. All the pattern files (.bmp) in the project's directory are listed in the Pattern List box. The Size field is the bitmap size of the selected pattern, Com and Seg fields are the numbers of the selected COM/SEG position of this pattern. None of these three fields can be modified.

- Select a pattern, a bitmap file, from the Pattern List box, or click the Browse button to change to another directory and select a pattern from that directory. The HT-LCDS uses 2-color bitmap files as the image source of patterns. The Preview-window zooms into the selected pattern.
- Set the X/Y positions in the panel screen for the selected pattern.
- Press the [OK] button and return to Fig 14-2, then click the File menu, Save command or click the Save button on the toolbar. The panel file has now been created or modified.



Size:17X16
Com:0
Seg:0
X 0
Y 0
Pattern Directory: C:\HT-IDE\SAMPLE

Fig 14-6

Delete a Pattern

- As shown in Fig 14-2, select the COM/SEG position of the pattern to be deleted and press the [Delete] key or click the Cut button on the toolbar.

Change the Pattern

- Delete the selected pattern first, then add a new pattern to change the pattern.
- Alternatively, as shown in Fig 14-2, select the COM/SEG position of the selected pattern and double click the mouse. The Pattern Information dialog box, as shown in Fig 14-6, is displayed. Select a pattern from the Pattern List box and press the [OK] button.

Change the Pattern Position

- As shown in Fig 14-2, use the Select-Drag-Drop method to move the pattern directly onto the panel screen.
- Alternatively, as shown in Fig 14-2, double click the COM/SEG position of the selected pattern. The Pattern Information dialog box, in Fig 14-6, is displayed. Set the X, Y value of the new position and press the [OK] button.

When the above operations have been completed and the system has returned to that shown in Fig 14-2, click the HT-LCDS File menu, Save command or click the Save button on the toolbar. The panel file has now been created or modified.

How to Add a User-define Matrix

The HT-LCDS supports a mapping strategy (File menu, Import user matrix command) which can help define a new matrix if the COM/SEG number is not equal to the ROW/COL number of the LCD panel. For example,
Assume there is an LCD panel of 2 COMs and 6 SEGs, and assuming this LCD panel is a 3 ROWs×4 COLs matrix, as shown in the following mapping

COM0-SEG0	COM0-SEG1	COM0-SEG2	COM0-SEG3
COM1-SEG0	COM1-SEG1	COM1-SEG2	COM1-SEG3
COM0-SEG4	COM0-SEG5	COM1-SEG4	COM1-SEG5

A definition file for the above matrix can be defined as follows,

```
; MATRIX.DEF
; Comment line
ROW = 3
COLUMN = 4
; mapping syntax: ROW,COL => COM,SEG
0 , 0  => 0 , 0  ; Map Row0 Col0 to COM0 SEG0
0 , 1  => 0 , 1  ; Map Row0 Col1 to COM0 SEG1
0 , 2  => 0 , 2  ; Map Row0 Col2 to COM0 SEG2
0 , 3  => 0 , 3  ; Map Row0 Col3 to COM0 SEG3
1 , 0  => 1 , 0  ; Map Row1 Col0 to COM1 SEG0
1 , 1  => 1 , 1  ; Map Row1 Col1 to COM1 SEG1
1 , 2  => 1 , 2  ; Map Row1 Col2 to COM1 SEG2
1 , 3  => 1 , 3  ; Map Row1 Col3 to COM1 SEG3
2 , 0  => 0 , 4  ; Map Row2 Col0 to COM0 SEG4
2 , 1  => 0 , 5  ; Map Row2 Col1 to COM0 SEG5
2 , 2  => 1 , 4  ; Map Row2 Col2 to COM1 SEG4
2 , 3  => 1 , 5  ; Map Row2 Col3 to COM1 SEG5
```

Define the Pattern Using the Panel Editor

The HT-LCDS supports a full panel edit interface to define the LCD panel patterns. If a panel picture file has been drawn already, then it is not necessary to set all pattern files in the panel respectively. The only requirement is to select the pattern positions.

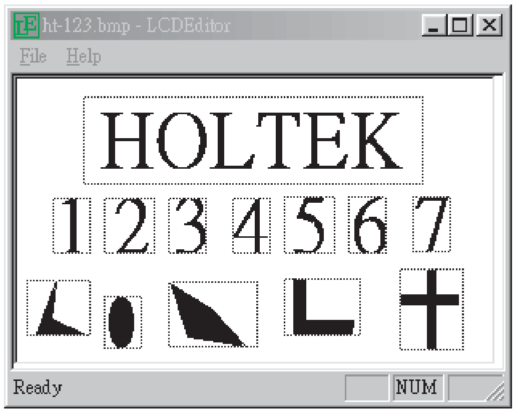


Fig 14-7

The following steps select the pattern positions for all the patterns in the LCD panel

- Invoke the Panel Editor by selecting the Edit command, Panel Editor command after having set the panel configuration
- Select the File menu, Open command in the Panel Editor to open the panel picture file (.bmp)

Note Supports 2-color .BMP only

- The panel will be displayed in the window as in Fig 14-7
- Select the pattern for each COM/SEG by using double-click or drag-and-drop methods. The Save Pattern dialog box will be displayed after which the pattern information can be entered.
- Repeat the above step for all patterns in the panel.
- After having set the pattern information for all patterns, return to the Panel Editor window and save all the settings using the File menu Save command.
- Exit the Panel Editor and return to the HT-LCDS, the panel will now display the new settings.

Add New Pattern Items Using a Batch File

The HT-LCDS provides a method to add pattern items from a batch file using the Edit menu and Add Item Batch command. The batch file is a text file with an extension name .BTH. All the pattern items in the batch file will define the pattern file name and its positions. After selecting a batch file using the Edit menu′s Add Item Batch command, the HT-LCDS adds all patterns depicted in the batch file at the specified positions of the panel. The following is an example of a .BTH file.

```
; this is a comment line.
; item syntax: BMPfile.bmp, COM, SEG, X, Y
CRYSTAL.BMP,  0, 2,  120, 30
FION.BMP,     2, 3,  200, 50
CLIN.BMP,     3, 2,  130, 90
STEVE.BMP,    4, 4,   20, 40
```

Selecting Color for an LCD Panel

The HT-LCDS provides a palette dialog, as shown in Fig.14-8, for selecting the colors of the panel using the HT-LCDS Configure menu and Set Panel Color command.

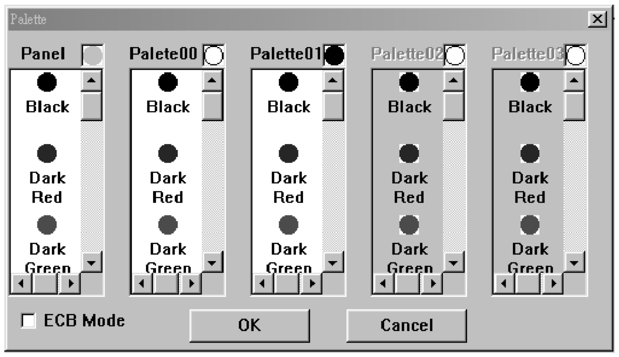


Fig 14-8

Note The ECB mode is for HTG21x0 color LCD only.

Setting Pattern Color for VFD Panel

The HT-LCDS provides an interface, as shown in Fig.14-9, for setting the color of each pattern for Holtek's VFD MCU (eg. HT49CVX series) Select Configure menu and execute the Set VFD pattern Color command to accomplish this setting.

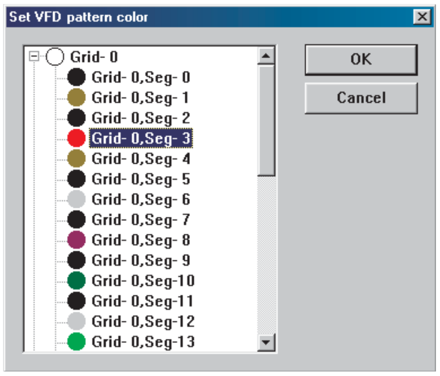


Fig 14-9

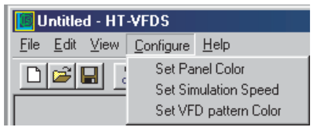


Fig 14-10

Simulating the LCD

Before starting the LCD simulation, ensure that the HT-LCDS refers to the correct panel configuration file. Enter the HT-LCDS environment by selecting the Tools menu, LCD Simulator command as shown in Fig 14-1 and Fig 14-2.

- Click once the S button on the toolbar allowing the HT-LCDS to begin LCD simulation while referring to the corresponding panel configuration file.
- Open a panel configuration file which is not the corresponding panel configuration file of the current project and click the S button on the toolbar. The HT-LCDS will then begin LCD simulation while referring to the opened panel configuration file.

When the HT-LCDS begins simulation, a window as shown in Fig 14-11 will be displayed while the most recent LCD patterns will be displayed on the panel screen.

Stop the Simulation

Double click the title bar of the LCD simulation window to make the HT-LCDS return to the edit mode.

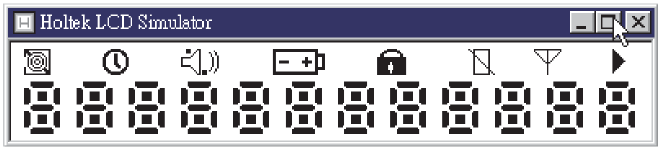


Fig 14-11

Chapter 15

Virtual Peripheral Manager

15

Introduction

In most practical applications the chosen MCU is connected to some forms of external hardware to implement the necessary user functions, however the inclusion of this external hardware in the simulation process is usually outside the scope of most MCU simulators. To overcome this problem, Holtek has developed a Virtual Peripheral Manager, or VPM, which enables the user to add a range of external peripheral devices to the MCU project. Used in conjunction with the HT-IDE simulator, the VPM enables the user to directly drive and monitor the inputs and outputs of these external hardware devices allowing for more efficient debugging and implementation of user applications.

The VPM Window

Fig 15-1 shows a practical example of a VPM window. As in most Windows applications the VPM window incorporates a toolbar for the function menus and a status bar to indicate program information with the main screen area displaying the peripherals or devices which have been added to the project.

The peripherals added to the project are known as components in the VPM. Components can be selected by clicking the mouse left button on the component required. Within this document the selected component will be referred to as the current component. By double clicking on the current component a connect dialog box will be displayed which permits the necessary connections to be made between the component and the MCU. By clicking the right mouse key, on certain current components a configuration dialog box will be displayed allowing attributes to be setup for that particular component.

In the status bar there are four fields, Mode, Current Component, Time and Cycle. The Mode field indicates whether the VPM is currently in configuration mode or running mode. The Current Component field shows the name of the current component. The Time field and Cycle field show the total execution time and cycle count respectively while the VPM is in running mode.

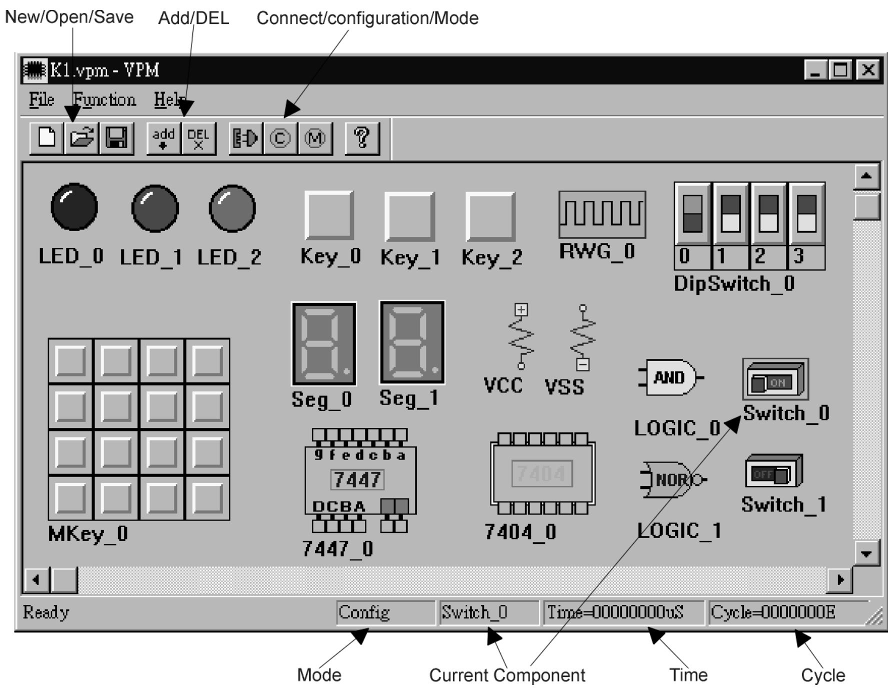


Fig 15-1

VPM Menu

File Menu

There are six functions in the File menu as shown in Fig 15-2. Three of the main functions can also be found on the toolbar as shown in Fig 15-3.

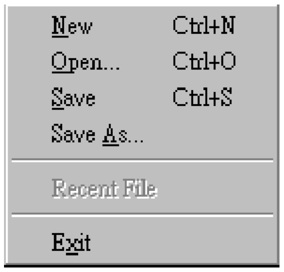


Fig 15-2

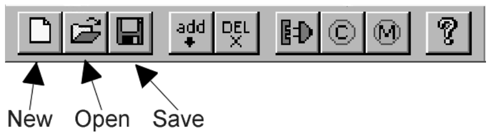


Fig 15-3

→ **New**
Create a new VPM project. Each time the VPM is entered the system automatically creates a new project.

→ **Open**
Open an existing VPM project.

→ **Save**
Save current project to file.

→ **Save As**
Save current project with another file name to file.

→ **Recent File**
List the most recently opened and closed four files.

→ **Exit**
Exit VPM and return to Windows.

Function Menu

There are five functions in the Function menu as shown in Fig 15-4. All of these functions can also be found on the toolbar as shown in Fig 15-5.

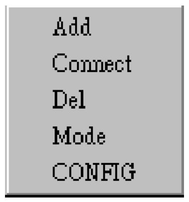


Fig 15-4

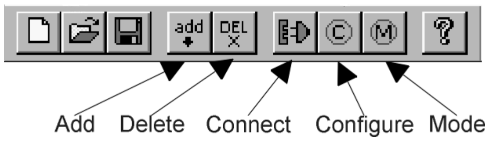


Fig 15-5

→ **Add**
Add a new peripheral to the project.
Click the Add button on the toolbar. An Add Peripheral dialog will be displayed as shown in Fig 15-6. Select the peripheral desired and click the OK button.

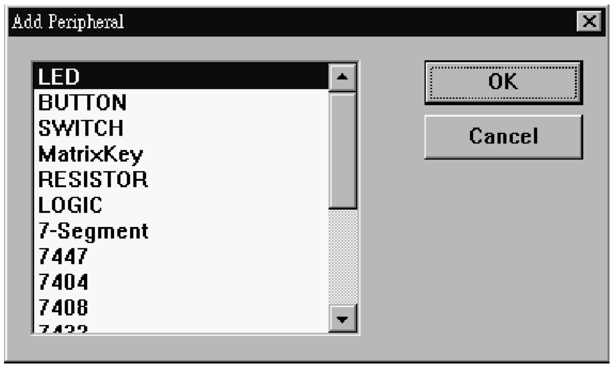


Fig 15-6

→ **Del**
Delete a peripheral from the project. Select the component to be deleted and click the Del button. The selected component will be removed from the project.

→ **Connect**
Select a component and click the Connect button on the toolbar. A Connect Dialog will be displayed like Fig. 15-7. The connection status of the current component will be displayed in Connect status list box. The Connect/Disconnect button can be used again to adjust the connection status between components.

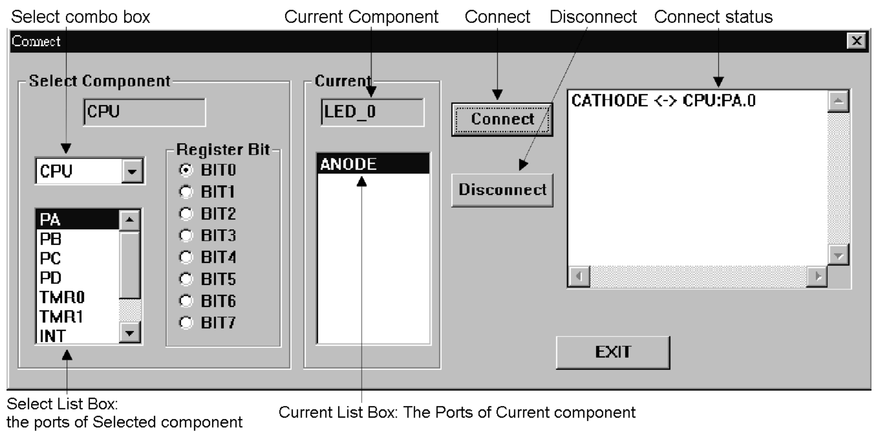
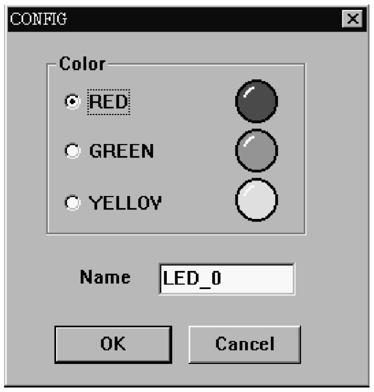


Fig 15-7

As an example, Fig. 15-7 shows the Connect dialog box for an LED component named LED_0. In this example, the current component is LED_0. The Select combo box will display all the components in this project that can be connected to LED_0. The Select List Box will display all the ports of the selected component. The Register Bit shows the port information details. The peripheral of an LED has two pins, one anode and one cathode. In this example, LED_0′s CATHODE pin has been connected to the CPU Port A bit0.

→ **CONFIG**

Some peripherals include some user adjustable attribute options. To do this the component should first be selected and then the Configure button pressed. If the component has attribute options, the Configuration Dialog box will be displayed. Fig. 15-8 shows an example of an LED configuration dialog box.



Name: LED_0

Color: [✓ RED] GREEN YELLOV

Fig 15-8

→ **Mode**

There has two modes, configuration mode and running mode. By clicking on the mode button, or selecting mode item from the function menu, the system will toggle the VPM between these two modes. In the configuration mode, the virtual external circuit can be edited using the Add/Del/CONFIG functions. In the running mode, the VPM will display the operations of these components according to their specific configurations in addition to displaying the Holtek IDE MCU simulation results.

The VPM Peripherals

LED

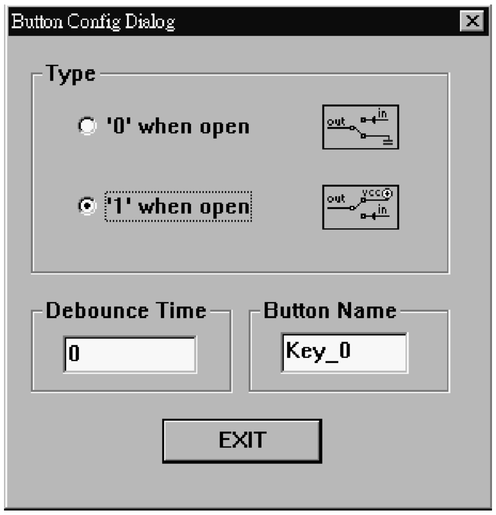
Fig 15-9

The LED has two pins, one cathode and one anode. When the cathode =0 and the anode =1, the LED will be illuminated. The LED has a colour option as shown in the configuration dialog box.

Button/Switch

Fig 15-10

The BUTTON/SWITCH has two options, the debounce time and the switch status when in the open position. The debounce time units are in milliseconds. The BUTTON has a non-latching momentary operation while the SWITCH has a latching non-momentary operation. The DipSwitch peripheral offers a means of providing multiple switches in a single package, the size of which is adjustable.


Button Config Dialog

Type
- '0' when open
- [✓ '1' when open]

Debounce Time: 0

Button Name: Key_0

Fig 15-11

Seven Segment Display

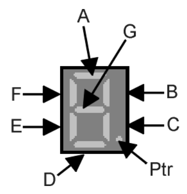


Fig 15-12

A seven segment display is formed from eight individual leds known as A, B, C, D, E, F, G and ptr. Each of these individual leds is connected to an input pin of the same name and also to a common pin. This common pin can be either a cathode (-) or an anode (+) connection which determines the polarity of the display.

→ **Resistor**

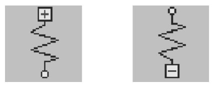
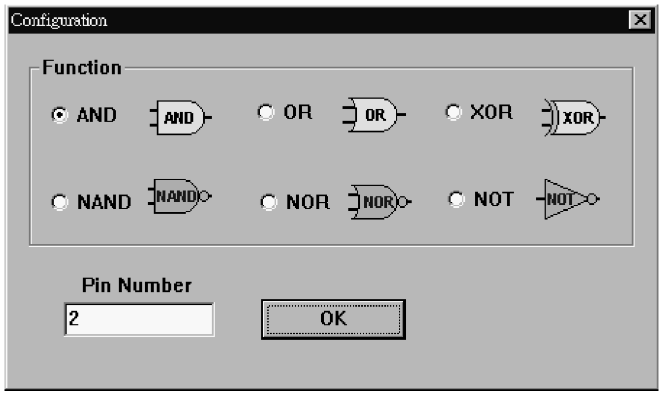


Fig 15-13

The resistors exist to provide a pull-up or pull-down function and are connected to either VCC or VSS respectively. The required configuration is set using their respective configuration dialog box.

→ **Logic Gate**

Logic gates are provided to give a total of six logic functions.



Configuration

Function

[✓ AND] OR XOR NAND NOR NOT

Pin Number: 2

Fig 15-14

Select a logic gate using the add function. If the logic gate that is displayed is not the required one, pressing the right key on the mouse will display a range of logic gates as shown in the figure. The desired logic gate can then be selected. The Pin Number input area determines the number of input pins to each gate. The value set here is reflected in the number of pins available in the connect dialog box.

→ **Matrix Key**

The Matrix key provides a standard matrix key peripheral device, the size of which can be setup from the configuration dialog box. The debounce time can be set for the matrix switches with the units in milliseconds. Note that the columns of the matrix are either connected to VCC or VSS, an option which is set in the attribute dialog box of the matrix peripheral.

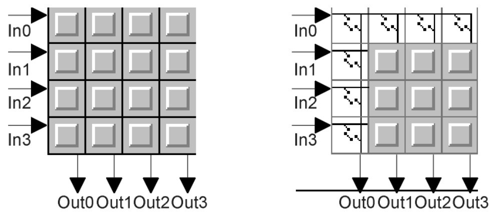


Fig 15-15

If, for example, the user sets up the matrix key with row = 4 and column =4, there will be 4 input pins or rows and 4 output pins or columns.

→ **Rectangle Wave Generator**

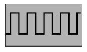


Fig 15-16

The rectangle wave generator is used to generate rectangular waves, the frequency of which is dependent upon the MCU frequency. In the attribute dialog box of this peripheral the cycle input dictates how many instruction cycles are required for an input waveform transition. If for example the cycle value is set to 2, then every 2 machine cycles the rectangular waveform generator input will toggle. The period of this input is therefore twice the cycle value. Note that if the rectangular wave generator is selected and the left key double clicked to display the connect dialog box, the generator can only connect to one device. However if the devices to be connected to are selected and their connect dialog box displayed then more than one device can be connected to the same wave generator. If more than one pin on the MCU is to be connected to the same wave generator then it is necessary to add further wave generators to achieve this.

Quick Start Example

From the examples provided in the Holtek IDE3000 User′s Guide, one has been chosen as a practical example to illustrate how to construct a virtual external circuit.

Scanning Light

→ **From within the HT-IDE3000 System**

- Create a new project and select the HT48C10-1 MCU (Project/New)
- Add the source file scanning.asm to the project (Project/Edit) The file can be found in the Holtek IDE\SAMPLE\CHAP15 directory
- Change the Holtek IDE to simulation mode.(Options/Debug/Mode)
- Build the project.(Project/Build)

→ **From within the VPM**

- Create a new VPM project.
- Add 8 LEDs to the project by repeatedly clicking the Add button and selecting LED 8 times
- Add a resistor to the project - click the Add button and select RESISTOR just added and double click the mouse left button - then setup the resistor′s name with VCC
- Connect all of the LED anode pins to VCC and connect all of the LED′s cathode pins to bit n of PA on the MCU (n=0-7). The following shows how to connect LED_0′s anode to VCC and its cathode to bit0 of PA on the MCU
  - Click the mouse left button on LED_0 to select it
  - Click the mouse right button on LED_0 to display the connect dialog box as shown as Fig 15-18
  - Connect the cathode of LED_0 to PA bit0 on the MCU
  - Repeat the above to setup all other LED_n connections
- Push the Mode button to change the VPM mode from configuration mode to running mode

→ **From within the HT-IDE3000**

Start the debug operations — the output results for the LEDs will be shown in the VPM window.

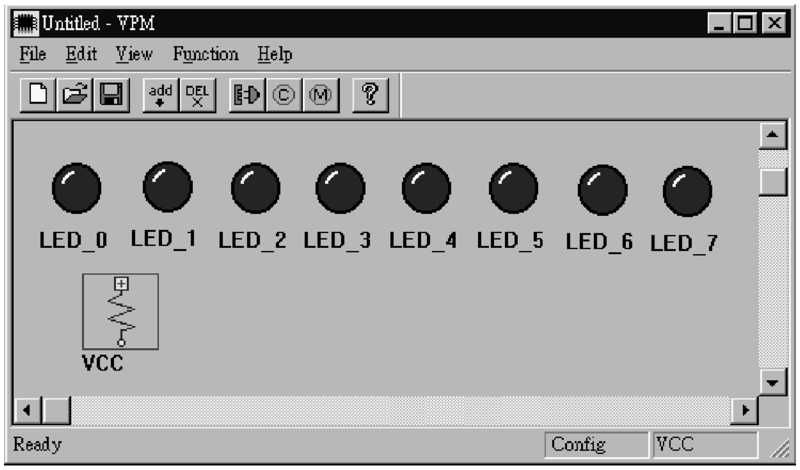


Fig 15-17

Select CPU and PA and BIT0 Select LED_0's CATHODE Push Connect Button

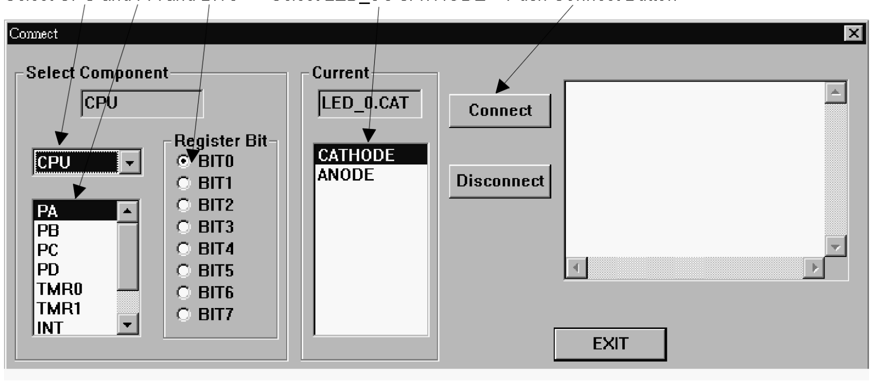


Fig 15-18

P a r t I V

Appendix

Appendix A

Reserved Words Used By Cross Assembler

A

Reserved Assembly Language Words

The following table lists all reserved words used by the assembly language.

- Reserved Names (directives, operators)

$	DUP	INCLUDE	NOT
*	DW	LABEL	OFFSET
+	ELSE	.LIST	OR
–	END	.LISTINCLUDE	ORG
.	ENDIF	.LISTMACRO	PAGE
/	ENDM	LOCAL	PARA
=	ENDP	LOW	PROC
?	EQU	MACRO	PUBLIC
[ ]	ERRMESSAGE	MESSAGE	RAMBANK
AND	EXTERN	MID	ROMBANK
BANK	HIGH	MOD	.SECTION
BYTE	IF	NEAR	SHL
DB	IFDEF	.NOLIST	SHR
DBIT	IFE	.NOLISTINCLUDE	WORD
DC	IFNDEF	.NOLISTMACRO	XOR

- Reserved Names (instruction mnemonics)

ADC	HALT	RLCA	SUB
ADCM	INC	RR	SUBM
ADD	INCA	RRA	SWAP
ADDM	JMP	RRC	SWAPA
AND	MOV	RRCA	SZ
ANDM	NOP	SBC	SZA
CALL	OR	SBCM	TABRDC
CLR	ORM	SDZ	TABRDL
CPL	RET	SDZA	XOR
CPLA	RETI	SET	XORM
DAA	RL	SIZ	
DEC	RLA	SIZA	
DECA	RLC	SNZ	

- Reserved Names (registers names)

A	WDT	WDT1	WDT2

Instruction Sets

Arithmetic Instructions

ADD A,[m]	Add Data Memory to ACC
ADDM A,[m]	Add ACC to Data Memory
ADD A,x	Add immediate data to ACC
ADC A,[m]	Add Data Memory to ACC with carry
ADCM A,[m]	Add ACC to Data Memory with carry
SUB A,x	Subtract immediate data from ACC
SUB A,[m]	Subtract Data Memory from ACC
SUBM A,[m]	Subtract Data Memory from ACC with result in Data Memory
SBC A,[m]	Subtract Data Memory from ACC with carry
SBCM A,[m]	Subtract Data Memory from ACC with carry and result in Data Memory
DAA [m]	Decimal adjust ACC for addition with result in Data Memory

Logic Operation Instructions

AND A,[m]	AND Data Memory to ACC
OR A,[m]	OR Data Memory to ACC
XOR A,[m]	Exclusive-OR Data Memory to ACC
ANDM A,[m]	AND ACC to Data Memory
ORM A,[m]	OR ACC to Data Memory
XORM A,[m]	Exclusive-OR ACC to Data Memory
AND A,x	AND immediate data to ACC
OR A,x	OR immediate data to ACC
XOR A,x	Exclusive-OR immediate data to ACC
CPL [m]	Complement Data Memory
CPLA [m]	Complement Data Memory with result in ACC

Increment & Decrement Instructions

INCA [m]	Increment Data Memory with result in ACC
INC [m]	Increment Data Memory
DECA [m]	Decrement Data Memory with result in ACC
DEC [m]	Decrement Data Memory

Rotate Instructions

RRA [m]	Rotate Data Memory right with result in ACC
RR [m]	Rotate Data Memory right
RRCA [m]	Rotate Data Memory right through carry with result in ACC
RRC [m]	Rotate Data Memory right through carry
RLA [m]	Rotate Data Memory left with result in ACC
RL [m]	Rotate Data Memory left
RLCA [m]	Rotate Data Memory left through carry with result in ACC
RLC [m]	Rotate Data Memory left through carry

Data Move Instructions

MOV A,[m]	Move Data Memory to ACC
MOV [m],A	Move ACC to Data Memory
MOV A,x	Move immediate data to ACC

Bit Operation Instructions

CLR [m].i	Clear bit of Data Memory
SET [m].i	Set bit of Data Memory

Branch Instructions

JMP addr	Jump unconditionally
SZ [m]	Skip if Data Memory is zero
SZA [m]	Skip if Data Memory is zero with data movement to ACC
SZ [m].i	Skip if bit i of Data Memory is zero
SNZ [m].i	Skip if bit i of Data Memory is not zero
SIZ [m]	Skip if increment Data Memory is zero
SDZ [m]	Skip if decrement Data Memory is zero
SIZA [m]	Skip if increment Data Memory is zero with result in ACC
SDZA [m]	Skip if decrement Data Memory is zero with result in ACC
CALL addr	Subroutine call
RET	Return from subroutine
RET A,x	Return from subroutine and load immediate data to ACC
RETI	Return from interrupt

Table Read Instructions

TABRDC [m]	Read ROM code (current page) to Data Memory and TBLH
TABRDL [m]	Read ROM code (last page) to Data Memory and TBLH

Miscellaneous Instructions

NOP	No operation
CLR [m]	Clear Data Memory
SET [m]	Set Data Memory
CLR WDT	Clear Watchdog Timer
CLR WDT1	Pre-clear Watchdog Timer
CLR WDT2	Pre-clear Watchdog Timer
SWAP [m]	Swap nibbles of Data Memory
SWAPA [m]	Swap nibbles of Data Memory with result in ACC
HALT	Enter Power Down Mode

Appendix B

Cross Assembler Error Messages

B

A0005	**Undefined symbol** The specified symbol is not defined in this file.
A0010	**Unexpected symbol** The symbol is redundant.
A0011	**Symbol already defined elsewhere** Re-defined symbol. Cross Assembler does not accept multiple symbol definitions.
A0012	**Undefined symbol in EQU directive** Cross Assembler does not accept undefined symbols to the right of directive EQU, even for forward references.
A0013	**Expression syntax error** Syntax error in expression.
A0014	**Cross Assembler internal stack overflow** This error is due to Cross Assembler processes the expression analysis.
A0016	**Duplicated MACRO argument** Two formal arguments in the MACRO definition line with the same name.
A0017	**Syntax error in MACRO parameters** Syntax error in MACRO formal parameters (expression).
A0018	**Wrong number of parameters** The total number of MACRO formal parameters is not equal to the total number of MACRO actual parameters (reference number is not equal to definition number).
A0019	**Redefined EQU** The symbol to the left of the directive EQU has been previously defined.
A0020	**Multiple section defined** The name of the section is the same as previously defined section. The section name must be unique in a source file.
A0021	**DBIT could be used in data section only** This directive can not be used in the code section.

A0022	**DB could be used in data section only** This directive can not be used in the code section.
A0024	**Syntax error** Syntax error in statement.
A0025	**MACRO too deep** Too many MACRO reference nesting levels. The maximum number of nesting levels is 7 (refer to other MACROs recursively).
A0026	**INCLUDE too deep** Too many INCLUDE file nesting levels. The maximum number of INCLUDE nesting levels is 7 (include other files recursively).
A0027	**IF too deep** Too many IF/ENDIF pair nesting levels. The maximum nesting level is 7.
A0028	**ELSE without IF** No directive IF before the directive ELSE (IF/ELSE/ENDIF pair is unbalanced).
A0029	**ELSE after ELSE** No directive ENDIF or IF after the directive ELSE. (IF/ELSE/ENDIF pair is unbalanced).
A0030	**ENDIF without IF** No directive IF before the directive ENDIF (IF/ELSE/ENDIF pair is unbalanced).
A0031	**Open conditional** The conditional directives pair (IF/IFE/ENDIF) is unbalanced.
A0032	**(expected** Left parenthesis is missing, should be added to the expression.
A0033	**ORG overlay** The memory address of directive ORG is overlaid with previously defined code.
A0034	**Value out of range** The specified value exceeds the allowed range.
A0035	**RAM-space limit exceeded** The total memory size of data sections exceeds the allowed RAM size.
A0036	**ROM-space limit exceeded** The total memory size of code sections exceeds the allowed ROM size.
A0037	**DC could be used in code section only** This directive can not be used in the data section.
A0038	**End of file encountered in MACRO definition** The directive ENDM is missing in the MACRO definition block (unbalanced).
A0039	**Constant expected** A constant is required in the expression.

A0040	**Open procedure** A directive ENDP is required to match the previous PROC.
A0041	**Block nesting error** The block nesting of directive PROC/ENDP is illegal.
A0042	**′ expected** The single quote ′ is missing.
A0043	**Non-digit in number** The number token contains a non-digit character.
A0044	**EXTERN needs an identifier** There is no identifier specified in the EXTERN directive.
A0045	**Data type expected** The data type of the identifier should be declared.
A0046	**Unknown data type** The data type is unknown.
A0047	**″:″ expected** The ″:″ is missing.
A0048	**Too many local labels** Too many local labels defined. At most 30 local labels are permitted between two labels.
A0049	**Redefined Section in ROMBANK is inconsistent** A section has already been declared in another ROMBANK directive.
A0050	**Bank out of range** The bank number specified in the ROMBANK directive exceeds the maximum bank number.
A0051	**Section Undefined in ROMBANK directive** The directive ROMBANK contains an undefined section name.
A0052	**Too many errors** There are too many errors encountered while assembling the source file.
A0053	**LABEL could be used in data section only** This directive can not be used in the code section.
A0054	**ROMBANK/RAMBANK shall be defined before SECTION is declared** A section with a specified ROM/RAM bank should be declared with ROMBANK/RAMBANK directive first.
A0055	**Record length overflow** The record length of the output object file is overflow.
A4001	**Incorrect command line option** The command line option is illegal.

A4002	**Redefined symbol** The specified symbol is defined already.
A4003	**No source file name** No source file name in the command line.
A4004	**Incorrect command line syntax** The command line syntax is illegal.
A4005	**Could not find file** The specified file is not found.
A4007	**Bad instruction format file** The instruction description file is incorrect.
A4008	**Cross Assembler internal fatal error** Cross Assembler failure, please contact dealer.
A4009	**Out of memory** No enough memory for Cross Assembler to process the source file.

Appendix C

Cross Linker Error Messages

C

L1001	**No object files specified** No object file is specified in the command line or the batch file. Check the command line syntax.
L1002	**Object file filename.obj is not found in pass1** The specified object file filename.obj is not found in Cross Linker pass1. Check if the file (filename.obj) is in the working directory, otherwise contact dealer.
L1003	**Out of memory** No enough memory space for Cross Linker. Check the total system free memory.
L1004	**Illegal section address ″dddd″** The section address specified in the command line option/ADDR is illegal. The address dddd should be in hex.
L1005	**Illegal command option ″option″** The specified option (option) in the command line is illegal.
L1006	**Batch file ″lbatch.bat″ is not found** The specified batch file lbatch.bat is not found. Check if the batch file (lbatch.bat) is in the working directory.
L1007	**Illegal file name ″filename.obj″** The specified file filename.obj contains illegal characters. Correct the characters of the file filename.obj.
L1008	**Command line syntax error** The syntax of the command line is incorrect.
L1009	**Illegal object file ″filename.obj″, RecType=xx** The format of the specified object file (filename.obj) is incorrect. Check if this object file has been generated by Holtek′s Cross Assembler.

L1010	**Cannot close object file ″filename.obj″** Cross Linker has failed to close the specified object file (system error). Contact dealer.
L1011	**Record ″xx″H check sum error** Cross Linker found a check sum error in the record (xxH) of the specified object file. Check if this object file is generated by Cross Assembler or not.
L1012	**MCU information mismatch file ″filename1.obj″ and ″filename2.obj″** Two object files with different MCU configurations during assembly. Ensure the same MCU configuration during assembling.
L1013	**Library file ″libname.lib″ does not exist** The specified library file libname.lib does not exist or the library file has not been generated by Holtek′s Library Manager. Check if the library file (libname.lib) is in the working directory.
L1014	**Cannot close the library file ″filename.lib″** Cross Linker has failed to close the specified file. Contact dealer.
L1015	**Library file ″libname.lib″ is not found** Cross Linker cannot re-open the specified library file libname.lib while processing the link work. Contact dealer.
L1016	**Object file ″filename.obj″ is not found in pass2** The specified object file filename.obj is not found in the Cross Linker pass2. Contact dealer.
L1017	**Cannot write the checksum of record ″xx″ H** Cross Linker fails to write check sum of record (xxH) to the output file. Contact dealer.
L1018	**Cannot write data of record ″xx″ H** Cross Linker fails to write record (xxH) data to the output file. Check the PC file system and working directory or contact dealer.
L1019	**Cannot open the debug file ″debugname.dbg″** Cross Linker failed to open the debug file debugname.dbg. Check the PC file system and working directory or contact dealer.
L1020	**Cannot open the task file ″taskname.tsk″** Cross Linker failed to open the task file taskname.tsk. Check the PC file system and working directory or contact dealer.
L1021	**Cannot open the map file ″mapname.map″** Cross Linker failed to open the map file mapname.map. Check the PC file system and working directory or contact dealer.

L1022	**Cannot create the debug file ″debugname.dbg″** Cross Linker failed to create the debug file debugname.dbg. Check the PC file system and working directory or contact dealer.
L1023	**Cannot create the task file ″taskname.tsk″** Cross Linker fails to create the task file taskname.tsk.. Check the PC file system and working directory or contact dealer.
L1024	**Cannot create the map file ″mapname.map″** Cross Linker fails to create the map file mapname.map. Check the PC file system and working directory or contact dealer.
L1025	**Program code is too large** The total size of program code is larger than the MCU ROM size. Check and Modify the program code (in CODE sections).
L1026	**Program data is too large** The total size of the program data sections is larger than the MCU RAM size. Check and Modify the DATA sections, omit some data in the RAM.
L1027	**Syntax error in batch file ″batch.bat″** The command syntax in the batch file is incorrect.
L1028	**Cannot close the batch file ″batch.bat″** Cross Linker failed to close the specified batch file. Contact dealer.
L1029	**Cannot open the binary file**
L1030	**Cannot create the binary file**
L1031	**Public symbols are duplicated** **Public symbol ″sym1″ in module ″mod-name1″** **Public symbol ″sym1″ in module ″mod-name2″** Cross Linker found a symbol named ″sym1″ that is declared as a public symbol in both modules, ″mod-name1″ and ″mod-name2″. Change one public symbol and all external references to this symbol to another name.
L1032	**Internal error for File Record** Cross Linker fails to convert the local file index to the global file index. Contact dealer.
L1033	**Internal error when obtaining the global index** Cross Linker failed to get the global file index. Contact dealer.
L1034	**Illegal class type for section ″sec-name″ in the file ″filename.obj″** Cross Linker found that the class name of section (sec-name) in the file (filename.obj) is illegal (neither CODE nor DATA). Check if the file (filename.obj) is generated by Holtek Cross Assembler. Otherwise, contact dealer.

L1035	**Internal error when section ″sec-name″ of the file ″filename.obj″ is located** Cross Linker failed to find the section (sec-name) while in section allocation. Contact dealer.
L1036	**The absolute address for the section is illegal**
L1037	**Two sections are overlapping** **Section ″sec-name1″ in the file ″filename1.obj″** **Section ″sec-name2″ in the file ″filename2.obj″** The ROM or RAM space allocated for the section ″sec-name1″ in the file ″filename1.obj″ overlaps with the ROM or RAM space of the section ″sec-name2″ in the file ″filename2.obj″. Check the address and length of these two sections. Refer to the listing file *.lst generated by Cross Assembler.
L1038	**ROM/RAM (Bank xx) memory allocation failed for section ″sec-name″ (size xxH) in the file ″filename.obj″** Cross Linker fails to find enough ROM or RAM space for the section (sec-name) of the file (filename.obj) while in public section allocation. Check the length of all sections in the input object files. Also, check or modify the align type of some sections to compact the sections space. Otherwise contact dealer.
L1039	**Internal error, failed to get SECDEF** Cross Linker internal error. Contact dealer.
L1040	**Illegal ROM bank number**
L1041	**A section in ROM bank is not defined**
L1042	**Failed to move the write pointer for task file** Cross Linker internal error. Contact dealer.
L1043	**Illegal Fixupp record in the file ″binary.obj″** Cross Linker internal error. Contact dealer.
L1044	**Illegal LIBHED record in the file**
L1045	**Illegal LIBNAM record in the file**
L1046	**Illegal LIBDIC record in the file**
L1047	**Caller is not a local section**
L1048	**Procedure (″proc-name″) is redefined**
L1049	**Illegal extern index**
L1050	**Local section name not in LNAMES**
L1051	**No corresponding section for extern index**

L1052	**Fail to get the global block ID**
L1053	**MCU RAM information mismatch**
L1054	**Illegal RAM bank number**
L1055	**A section in RAM bank is not defined**
L1056	**Both banks ″bank-no1″ and ″bank-no2″ contains the section ″sec-name″**
L1057	**Total length of combined sections exceeds the bank size**
L1058	**The specified section address conflicts with the absolute section**
L1059	**The bank number of specified section address conflicts with the section**
L1060	**Error Fixmth data is referred by bank Fixupp**
L2001	**Unresolved external symbol ″ext-symbol″ in file ″filename.obj″** No public symbol named ″ext-symbol″ in the file filename.obj has been found in either the input object files or the specified library files. Link the object file that defines a public symbol named ext-symbol into the command line, or include a library file defining a public symbol named ext-symbol.
L2002	**Symbol type mismatch** **Public symbol ″symbol1″ in module ″mod-name1″** **External symbol ″symbol1″ in module ″mod-name2″** Cross Linker found that an external symbol and a public symbol have the same name, but have a different symbol type. Check the symbol type of this external symbol, modify the source file, re-assemble the file and re-link.
L3001	**Specified section ″sec-name″ does not exist** The specified section (sec-name) in the command line option/ADDR does not exist Input the correct section name in the command line or ignore this section. This is a warning message, Cross Linker does the allocation work as if this option has not been issued.
L3002	**Specified address ″xxxx″ for the code section ″sec-name″ is illegal** The specified address of the specified section (sec-name) in the command line option /ADDR is illegal (not a hexadecimal digit or exceeds the legal range). Input the correct address in the command line or ignore this section. This is a warning message, Cross Linker does the allocation work as if this option has not been issued.
L3003	**Specified address ″xxxx″ for the data section ″sec-name″ is illegal**
L3004	**Recursive situation occurred in procedure ″proc-name″**

Appendix D

Cross Library Error Messages

D

U0001	**No library file name**
U0002	**Library file does not exist**
U0003	**Library file exists already**
U0004	**The contents of the library file will be discarded if operation is executed**
U0005	**Can′t open the library file**
U0006	**Can′t create a library file**
U0007	**Can′t create a TMP library file**
U0008	**Incorrect library file**
U0009	**Can′t open the list file**
U0010	**Can′t insert a new module to library**
U0011	**Can′t open the object file**
U0012	**Delete operation fails**
U0013	**Replace operation fails**
U0014	**A module with the same name exists in library already** In any library file, there cannot exist two modules with the same name. Library Manager will check this situation when processing ADD operation.
U0015	**The module doesn′t exist in library** The specified module is not in the specified library file. Library Manager will check when processing DELETE, REPLACE, EXTRACT operations.
U0016	**No enough memory** The user system has no enough memory for Library Manager.

U0017	**Bad object file** The file to be added to the library file has a bad object format. It may not be generated by Cross Assembler or a disk error.
U0018	**No public name in the specified module** If a symbol needs to be public, refer to chapter 9, program directive for PUBLIC directive, and re-assemble the source file, then use Library Manager to replace the new object file with the old module.
U0019	**Illegal operation**
U0020	**Fail to close a file**
U0021	**Check sum is incorrect** Library Manager internal error.
U0022	**Fail to out record to the library file** Library Manager internal error.
U0023	**Out checksum error** Library Manager internal error.
U0024	**Fail to seek file** Library Manager internal error.

Appendix E

Holtek Cross C Compiler Error Messages

E

Error Code

C1000	Unterminated conditional in #**include**
C1001	Unterminated #**if**/#**ifdef**/#**ifndef**
C1002	Unidentifiable control line
C1003	Could not find include file *filename*
C1004	Illegal operator * or & in #**if**/#**elsif**
C1005	Bad operator *(operator)* in #**if**/#**elsif**
C1007	#**elif** with no #**if**
C1008	#**elif** after #**else**
C1009	#**else** with no #**if**
C1010	#**else** after #**else**
C1011	#**endif** with no #**if**
C1012	#**defined** token is not a name
C1013	#**defined** token *token* cannot be redefined
C1014	Incorrect syntax for ″**defined**″
C1015	Bad syntax for control line
C1016	Preprocessor control *control* not yet implemented
C1017	Duplicate macro argument
C1018	Syntax error in macro parameters
C1019	Macro redefinition of *macro-name*
C1020	Disagreement in number of macro arguments
C1021	EOF in macro argument list
C1022	# not followed by macro parameter
C1024	Macro argument is too long
C1025	Unknown internal macro
C1026	Unterminated string or char const
C1027	Undefined expression value
C1028	Bad ?: in #**if**/#**endif**
C1030	Bad number *number* in #**if**/#**elsif**
C1031	Empty character constant
C1034	String in #**if**/#**elsif**
C1035	Syntax error in #**undef**
C1036	Syntax error in #**else**

C1037	Syntax error in #**line**
C1038	Syntax error in #**ifdef/#ifndef**
C1040	Syntax error in #**if/#elsif**
C1042	Syntax error in #**include**
C1043	Syntax error in #**if/#endif**
C1044	Syntax error in #**endif**
C1045	Lexical error in preprocessor
C1046	Internal error in #**if**/#**elsif**
C1047	EOF inside comment
C1048	#**error** directive: "err-string"
C1049	#**line** specifies number out of range
C2001	unrecognized declaration
C2002	invalid use of **auto/register**
C2004	invalid use of *specifier*
C2005	invalid type specification
C2006	invalid use of **typedef**
C2007	missing identifier
C2008	redeclaration of *identifier*
C2009	empty declaration
C2010	invalid storage class
C2011	redeclaration of *identifier* previously declared at file_line_no
C2012	redefinition of *identifier* previously defined at file_line_no
C2013	illegal initialization for *identifier*
C2014	undefined size for type *identifier*
C2015	extraneous identifier *identifier*
C2016	*size* is an illegal array size
C2017	illegal formal parameter types
C2018	missing parameter type
C2019	expecting an identifier
C2020	extraneous old-style parameter list
C2021	illegal initialization for parameter *identifier*
C2022	invalid *operator* field declarations
C2023	missing *operator* tag
C2024	*type* is an illegal bit-field type
C2025	*size* is an illegal bit-field size
C2026	field name missing
C2027	*type* is an illegal field type
C2028	undefined size for field type *identifier*
C2029	size of *type* exceeds *number* bytes
C2030	illegal use of incomplete type *type*
C2031	conflicting argument declarations for function *identifier*
C2032	missing name for parameter *number* in function *identifier*
C2033	undefined size for parameter *type identifier*
C2034	declared parameter *identifier* is missing
C2035	undefined static *type identifier*
C2036	undefined label *identifier*
C2037	expecting an enumerator identifier
C2038	underflow/overflow in value for enumeration constant *identifier*
C2039	unknown enumeration *identifier*
C2040	type error in argument *number* to *identifier*; found *type1* expected *type2*
C2041	too many arguments in *identifier*
C2042	insufficient number of arguments in *identifier*

C2043	unknown size for type *type*
C2044	assignment to const identifier *identifier*
C2045	assignment to const location
C2046	addressable object required
C2047	operands of *identifier* have illegal types *type1* and *type2*
C2048	operand of unary *operator* has illegal type *type*
C2049	syntax error; found *token1* expecting *token2*
C2050	too many errors
C2051	skipping *token*
C2053	invalid operand of unary **&**; *identifier* is declared register
C2054	invalid type argument *type* to **sizeof**
C2055	**sizeof** applied to a bit field
C2056	cast from *type1* to *type2* is illegal
C2057	found *type* expected a function
C2059	field name expected
C2060	left operand of -> has incompatible type *type*
C2061	illegal use of type name t*ype*
C2062	illegal use of argument
C2063	illegal expression
C2064	lvalue required
C2065	unknown field *identifier* of type
C2067	initializer must be constant
C2068	cast from *type1* to *type2* is illegal in constant expressions
C2069	invalid initialization type; found *type1* expected *type2*
C2070	cannot initialize undefined *type*
C2071	missing { in initialization of *type*
C2072	number of initializers not matched
C2073	illegal character@
C2074	invalid hexadecimal constant *identifier*
C2075	invalid binary constant *identifier*
C2076	invalid octal constant *identifier*
C2077	missing *character*
C2078	*identifier* literal too long
C2079	missing *'*
C2080	illegal character *character*
C2081	*identifier1* is a preprocessing number but an invalid *identifier2 constant*
C2082	invalid floating constant *identifier*
C2083	ill-formed hexadecimal escape sequence
C2084	integer expression must be constant
C2085	illegal **break** statement
C2086	illegal **continue** statement
C2087	illegal **case** label
C2088	**case** label must be a constant integer expression
C2089	illegal **default** label
C2090	extra **default** label
C2091	extraneous return value
C2092	missing label in **goto**
C2093	unrecognized statement
C2094	illegal statement termination
C2095	redefinition of label *identifier* previously defined at *life_line_no*
C2096	illegal *type* type in **switch** expression
C2097	duplicate **case** label *value*

C2098	illegal return type; found *type1* expected *type2*
C2099	type error: pointer expected
C2100	illegal type ″array of *type*″
C2101	missing array size
C2102	type error: array expected
C2103	illegal type *type*
C2104	type error: function expected
C2105	duplicate field name *identifier* in type
C2106	illegal initialization of **extern** *identifier*
C2107	**#endasm** expected
C2109	variable with initialized value must be declared as constant.
C2110	ROM constant variable must be initialized
C2111	constant variable must be declared as global
C2112	overflow in octal escape sequence
C2113	bit variable cannot be declared as constant
C2114	unclosed comment
C2115	illegal operation for bit variable
C2116	bit pointer not allowed
C2117	invalid pragma string
C2118	bit member in structure not allowed
C2119	ROM constant variable must not be declared as **extern**
C2120	vector function must not have parameters
C2121	vector function must be **void** type
C2122	const string must be used in the C file having main function
C2123	array should specify the size
C2124	size of ″array of *type*″ exceeds *n* bytes
C2125	should specify ROM address
C2126	RAM address ″@″ cannot be used with constant variables
C2127	variable with specific RAM address ″@″ should be declared as global
C2128	left operand of . has incompatible type
C2129	undeclared identifier
C2130	array size exceeds 255
C2131	more than 255 bytes in *type*
C2132	invalid initial value
C2133	bit array not allowed
C2134	redefinition of vector
C2135	invalid vector
C2136	vector is used
C2137	syntax error ; redundant tokens after #**asm**/#**endasm**
C2138	in-line asm should be put within a function
C2200	internal error
C2201	insufficient memory
C2202	read error

Warning Code

C4001	empty declaration
C4002	empty input file
C4003	missing prototype
C4004	inconsistent linkage for *identifier* previously declared at *file_line_no*
C4006	declaration of *identifier* does not match previous declaration at *file_line_no*

C4008	register declaration ignored for *type identifier*
C4009	extraneous 0-width bit field *type identifier* ignored
C4010	more than 127 fields in *type*
C4011	more than 31 parameters in function *identifier*
C4012	old-style function definition for *identifier*
C4013	compatibility of *type1* and *type2* is compiler dependent
C4014	*identifier* is a non-ANSI definition
C4015	missing return value
C4016	static *type identifier* is not referenced
C4017	parameter *type identifier* is not referenced
C4018	local *type identifier* is not referenced
C4019	register declaration ignored for *type identifier*
C4020	more than 127 enumeration constants in *type*
C4021	non-ANSI trailing comma in enumerator list
C4022	more than 31 arguments in a call to identifier
C4023	assignment between *type1* and *type2* is compiler-dependent
C4024	*identifier* used in a conditional expression
C4026	conversion from *type1* to t*ype2* is compiler dependent
C4027	*type* used as an lvalue
C4028	conversion from *type1* to *type2* is undefined
C4029	more than 511 external identifiers
C4030	initializer exceeds bit-field width
C4031	missing ″ in preprocessor line
C4033	unrecognized control line
C4034	more than 509 characters in a string literal
C4035	string/character literal contains non-portable characters
C4036	excess characters in multibyte character literal *token* ignored
C4037	overflow in constant *token*
C4039	overflow in hexadecimal escape sequence
C4041	unrecognized character escape sequence *character*
C4042	overflow in constant expression
C4043	result of unsigned comparison is constant
C4044	shifting a type by *number* bits is undefined
C4045	unreachable code
C4046	more than 15 levels of nested statements
C4047	switch statement with no cases
C4048	more than 257 cases in a switch
C4049	switch generates a huge table
C4050	pointer to a parameter/local *identifier* is an illegal return value
C4051	source code specifies an infinite loop
C4052	more than 127 identifiers declared in a block
C4053	reference to *type* elided
C4054	reference to **volatile** *type* elided
C4055	declaring type array of *type* is undefined
C4056	qualified function type ignored
C4057	unnamed *operator* in prototype

Fatal Code

C6001	function not supported yet