HT-IDE2000 User's Guide

Fifth Edition

NOTICE

The information appearing in this Data Book is believed to be accurate at the time of publication. However, Holtek assumes no responsibility arising from the use of the specifications described. The applications mentioned herein are used solely for the purpose of illustration and Holtek makes no warranty or representation that such applications will be suitable without further modification, nor recommends the use of its products for application that may present a risk to human life due to malfunction or otherwise.

Holtek reserves the right to alter its products without prior notification. For the most up-to-date information, please visit our web site at http://www.holtek.com.tw.

Contents

Part I

Integrated Development Environment

Chapter 1

Overview and Installation

1

The HT-IDE2000 (Holtek Integrated Development Environment) is a high performance integrated development environment designed around Holtek's series of 8-bit microcontroller (μC) chips. Incorporated within the system is the hardware and software tools necessary for rapid and easy development of ASIC (Application-Specific Integrated Circuit) applications based on the Holtek range of 8-bit μCs.

The key component within the HT-IDE2000 system is the HT-ICE or In-Circuit Emulator, capable of emulating the Holtek 8-bit μC in real time, in addition to providing powerful debugging and trace features.

As for the software, the HT-IDE2000 provides a friendly workbench to ease the process of application program development, by integrating all of the software tools, such as editor, macro assembler, linker, library and symbolic debugger into a user friendly windows based environment.

In addition the HT-IDE2000 provides a software simulator which is capable of simulating the behaviour of Holtek's 8-bit μC range without using the HT-ICE. All fundamental functions of the HT-ICE hardware are valid for the simulator.

Introduction

Some of the special features provided by the HT-IDE2000 include:

→ **Emulation**

- Real-time program instruction emulation
- On-line or off-line (stand-alone) 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

→ **Software**

- Windows based software utilities
- Source program level debugger (symbolic debugger)
- Workbench for multiple source program files (more than one source progam 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)
- 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
- μC VROM editor modifies and compresses the voice data in order to generate a proper VROM (Voice ROM) size.
- LCD simulator simulates the behavior of the LCD panel.

System Configuration

The HT-IDE2000 system configuration is shown in Fig1-1, in which the host computer is a Pentium compatible machine, with windows 95/98 or later.

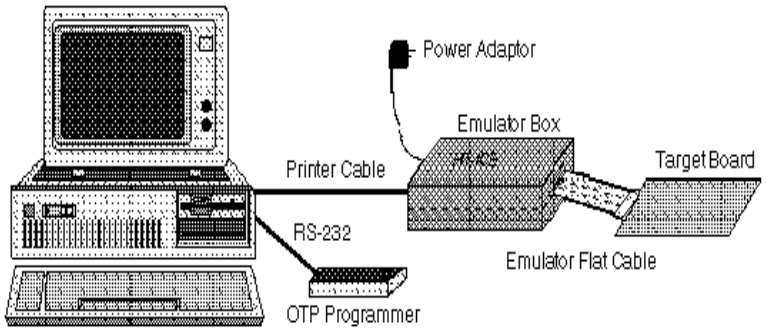


Fig 1-1

→ **The HT-IDE2000 system contains the following hardware components**

- The HT-ICE box contains PCBs (printed circuit board) with 1 printer port connector for connecting to the host machine, I/O signal connector and one LED, Fig 1-2.
- I/O interface card for connecting the target board to the HT-ICE box
- Power adaptor, output 9V
- 25-pin D-type printer cable
- OTP programmer, gang programmer (optional)

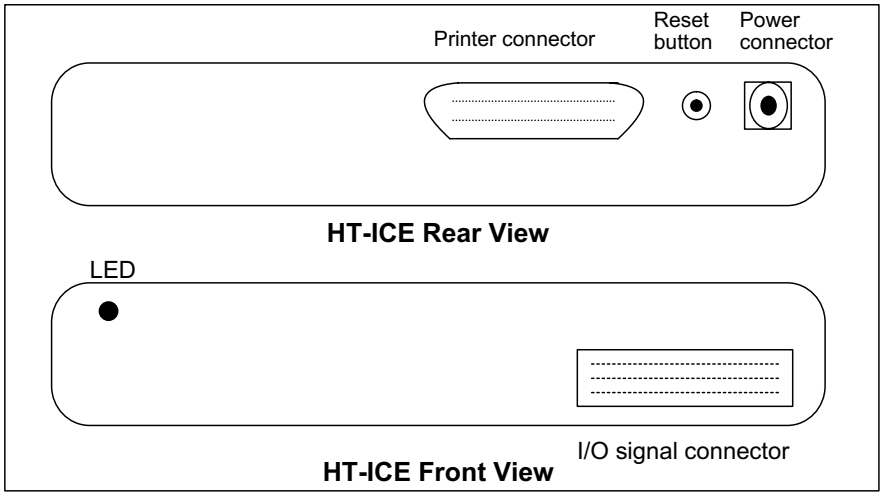


Fig 1-2

→ **I/O interface card**

The I/O interface card (Fig 1-3) is a PCB which is used to connect the HT-IDE2000 system to the user's target board. It provides the following functions:

- external clock source
- external signal trace input
- μC pin assignment

The external clock source has two modes, RC & crystal. For use with a crystal clock, short positions 1 and 2 on Jumper JP1. Otherwise for an RC clock short positions 2 and 3, and adjust the system frequency with VR1 (Fig 1-3). Refer to the Tools/Mask Option Menu for the choice of the clock source and system frequency.

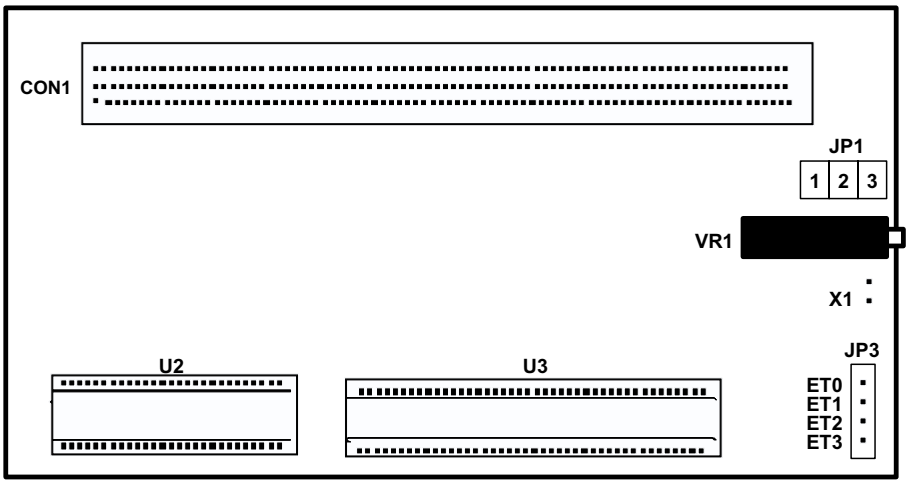


Fig 1-3

The 4 external signal trace inputs, marked as ET0 to ET3 at jumper location JP3, exist to help the user trace and digitize signals and analyse their behavior. Refer to the chapters on Breakpoint and Trace the Application Program for more information.

The μC pin assignments in location U2, U3 are defined in the same manner as the pin assignment of the HT48CX0 series according to the Holtek 8-bit μC databook. The VME connector at location CON1 is used to connect the I/O interface card to the HT-ICE.

Installation

System requirement

The hardware and software requirements for installing HT-IDE2000 system are as follows:

- PC/AT compatible machine with Pentium or higher CPU
- SVGA color monitor
- At least 32M RAM for well performance
- 3.5 floppy disk drive (for disk installation)
- CD ROM drive (for CD installation)
- At least 10M free disk space
- Parallel port to connect PC and HT-ICE
- MS-Windows 95/98/NT/2000

Note MS-Windows 95/98/NT/2000 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 9V, 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-IDE2000 CD into the CD ROM drive, the following dialog will be shown

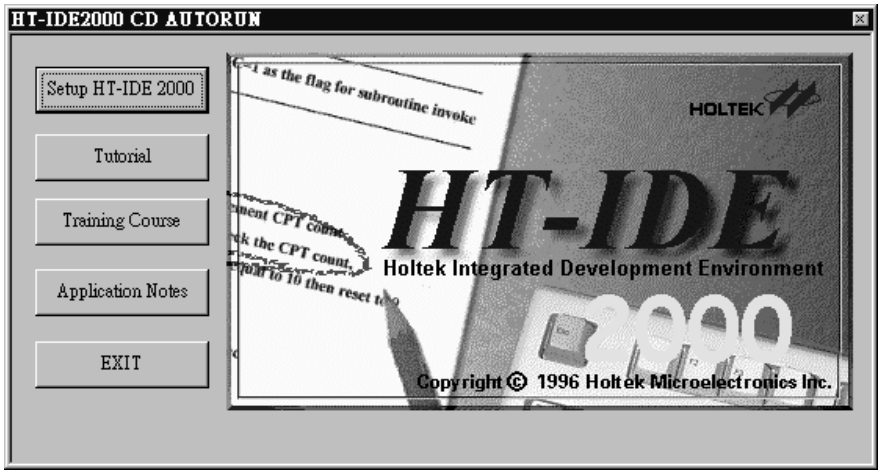


Fig 1-4

Click <Setup HT-IDE2000> button and the following dialog (Fig1-5) will be shown

Fig 1-5

- Step 2

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

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

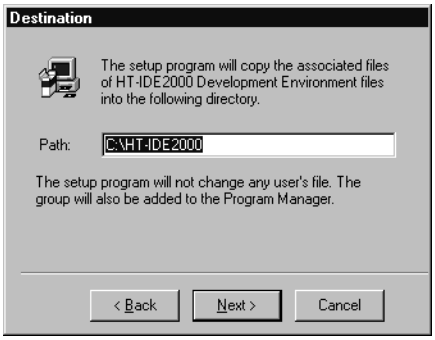


Fig 1-6

• Step 4
Specify the path you want to install the HT-IDE2000 and click <Next> button

• Step 5
SETUP will copy all files to the specified directory

• Step 6
If the process is successful a dialog will be shown

Fig 1-7

• Step 7
Press button and you can run HT-IDE2000 now.

Note 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 micro-C bodies. The INCLLUDE 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 Before you first time run HT-IDE2000, the system will ask you for company information (Fig 1-8). Select appropriate area and fill in the company name and ID. You can ask your HT-IDE2000 provider for an ID number.

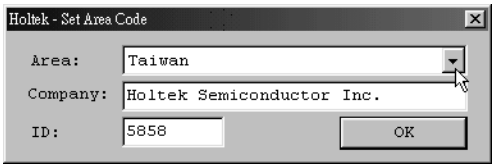


Fig1-8

Chapter 2

Quick Start

2

This chapter gives a brief description of using HT-IDE2000 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 a microcontroller from the combo box
- Click OK button and the system will ask you to setup the mask options
- Setup all mask 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 programs and save them with your 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 your 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 good, the system will create a Task file and download to the HT-ICE for debug

→ **Step 4**

Send Code to Holtek

- You may repeat Step 3 before you finish debugging your programs
- Click on Options menu and select Project command
- A Project Option dialog will be shown
- Check Generate .COD file box and click OK button
- Rebuild the project for creating the .COD file
- Click on Project menu and select Print Option Table command
- Send the .COD file and the Option Approved Sheet to Holtek

→ **Step 5**

Programming OTP IC

- Click on Options menu and select Project command
- A Project Option dialog will be shown
- Check Generate .OTP file box and click OK button
- Rebuild the project for creating the .OTP file
- Click on Tools menu and select OTP Programmer command
- Use HT-OTP to program the OTP ICs

The Programming and data flow can be illustrated by the following diagram

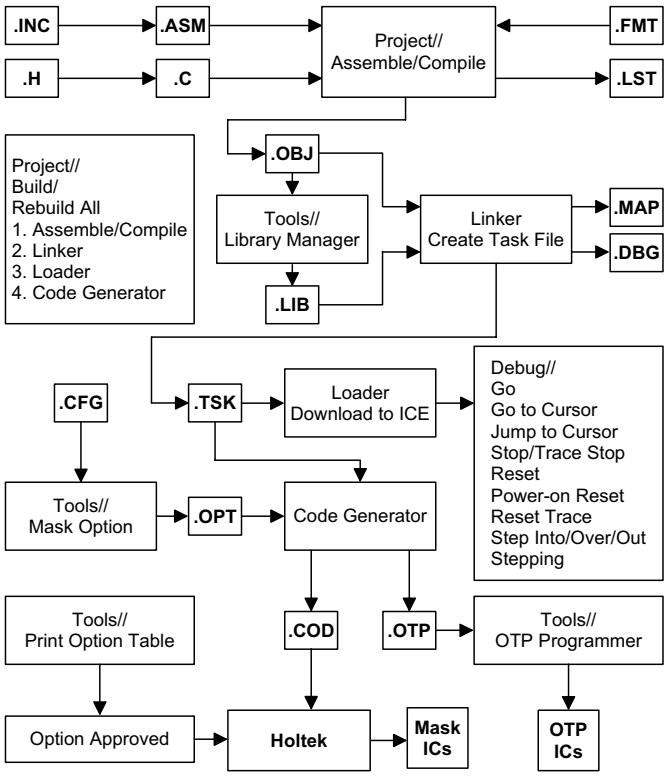

.INC
.ASM
.H
.C
Project//
Assemble/Compile
.FMT
.LST
Project//
Build/
Rebuild All
1. Assemble/Compile
2. Linker
3. Loader
4. Code Generator
.OBJ
Tools//
Library Manager
.LIB
Linker
Create Task File
.MAP
.DBG
.CFG
.TSK
Loader
Download to ICE
Debug//
Go
Go to Cursor
Jump to Cursor
Stop/Trace Stop
Reset
Power-on Reset
Reset Trace
Step Into/Over/Out
Stepping
Tools//
Mask Option
.OPT
Code Generator
Tools//
Print Option Table
.COD
.OTP
Tools//
OTP Programmer
Option Approved
Holtek
Mask
ICs
OTP
ICs

Fig 2-1

Chapter 3

HT-IDE2000 Menu – File/Edit/View/Tools/Options

3

This chapter describes some of the menus and commands of the HT-IDE2000. Other menus are described in the Project, Debug and Window chapters.

Start the HT-IDE2000 System

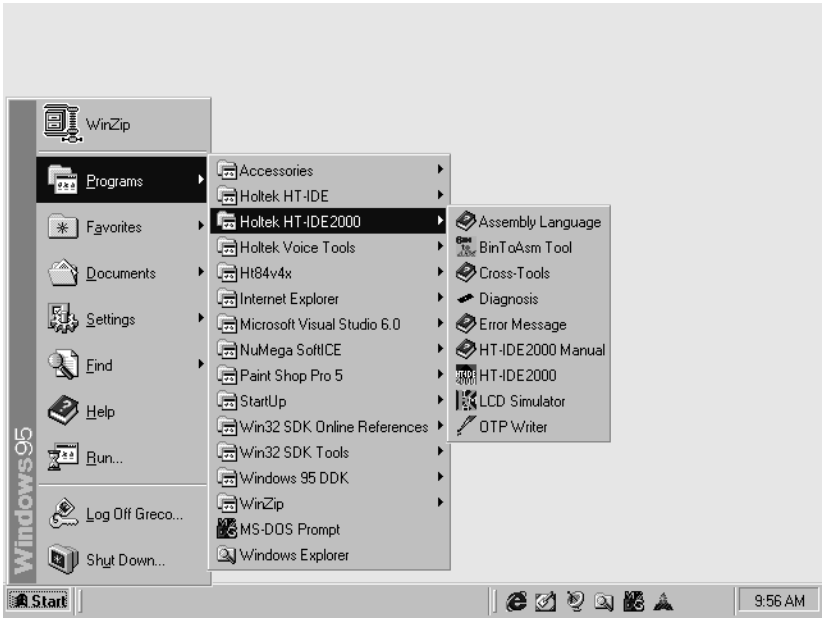


Fig 3-1

- **For windows 95/98/NT/2000**
  - Click Start Button, select Programs and select Holtek HT-IDE2000
  - Click the HT-IDE2000 icon
- If the last project you worked on HT-IDE2000 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.

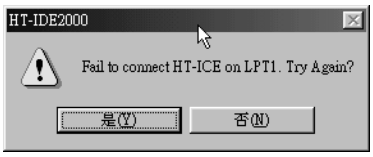


Fig 3-2

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-IDE2000 enters the emulation mode and the HT-ICE begins to function.

Fig 3-3

- IF the last project you work on HT-IDE2000 is in simulation mode (using Simulator), then Fig 3-4 will be displayed to indicate that HT-IDE2000 will enter the simulation mode.

Fig 3-4

The HT-IDE2000 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 statusbar, in the bottom line (Fig 3-5), displays the emulation or simulation present status and the result status of commands.

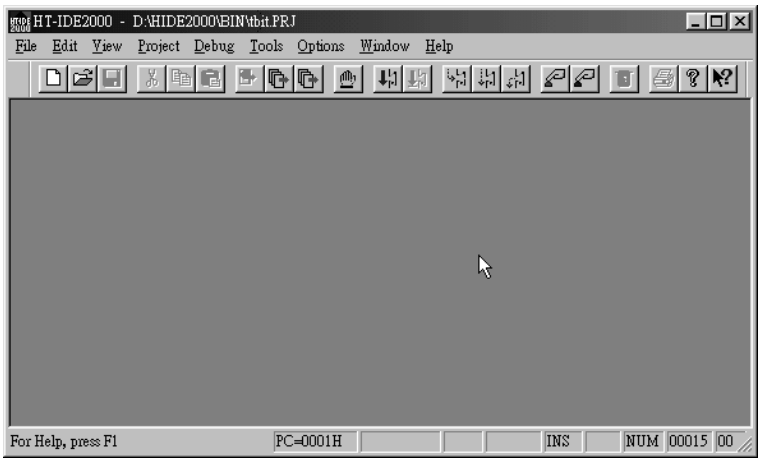


Fig 3-5

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-IDE2000 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-IDE2000. (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 8 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.

Fig 3-6

Tools Menu

The Tools menu provides the special commands to facilitate user application debug. These commands are Mask Option, Diagnose, OTP Programmer, Library Manager, Voice tools and LCD Simulator.

Fig 3-7

Mask option

This command generates an option file used by the Build command in the Project menu. The contents of the option file depends upon the specified μC. 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 μC 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 1 and 2 of jumper JP1 on the I/O interface card.
- For RC mode, adjust the system frequency with VR1 and short positions 2 and 3 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-IDE2000 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.

- μC resource option space
  Diagnose the uC 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.

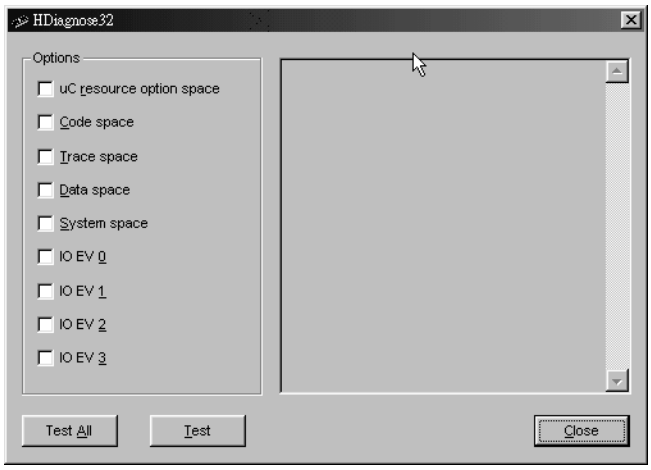


Fig 3-8

OTP programmer

The OTP (One Time Programmable) programmer supports functions for programming the OTP ICs. Refer to the chapter on usage of the HT48RX0 OTP programmer. The function of this command is the same as the program-item icon OTP writer in the HT-IDE2000 group.

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 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

Chapter 12 gives more details about these functions.

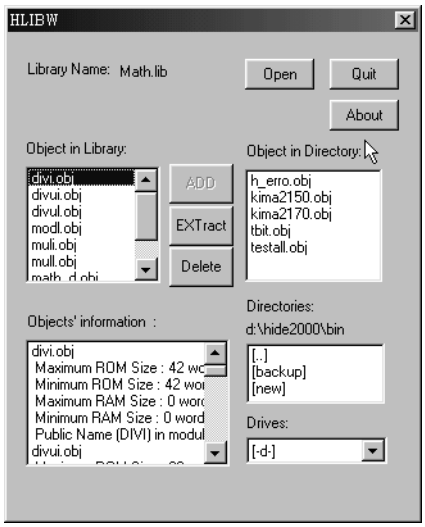


Fig 3-9

Voice/VROM editor

The VROM. Editor, in Fig 3-10 provides the following functions for μC voice

- Select the voice resource files with format .WAV or .PCM
- Modify the voice resource files
- Compress the voice resource files to a .VOC file to reduce the reaquired voice ROM size
- Modify the binary code if the VROM contains a voice program

Refer to chapter 13 for more details.

Fig 3-10

Voice/Download

This command downloads the contents of a specified voice data file .VOC to the HT-ICE for emulation. It also uploads from the HT-ICE VROM saving the data to a specified .VOC file. Fig 3-11 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 microcontroller. When uploading, a different file name from the project name may be specified to save the contents of the HT-ICE voice RAM. Ensure that the voice ROM file .VOC has been generated by the VROM Editor before downloading.

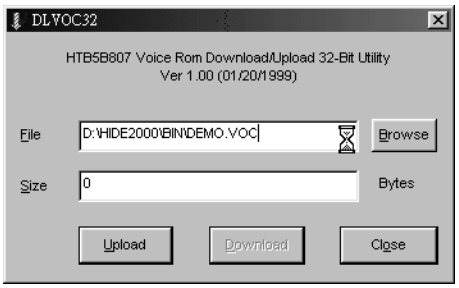


Fig 3-11

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. Chapter 14 gives more details on the LCD simulator.

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-IDE2000 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.

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

Fig 3-12

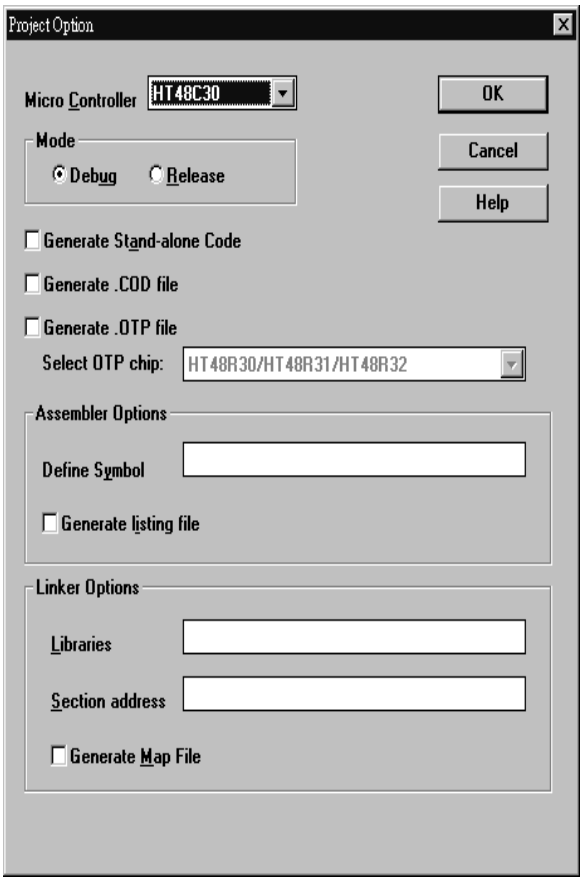


Fig 3-13

- Microcontroller

  The μC name of this project. Use a scroll arrow to browse the available μC names and select the appropriate one.

- Mode

  The processing mode of the project. The Debug mode is used during the development process. The Release mode is the same as the Debug mode except that there is no debugging information file in this mode. In the Debug mode, the contents of the source files will be displayed on the active window when the Build command of the Project menu has been executed (refer to the chapter on Build a project s task files). In Release mode, the deassembled instructions will be displayed on the Program window of the Window menu (refer to the chapter on Commands of the Window menu). There is no symbol in this window due to lack of debugging information.

- Generate stand-alone code

  Check this box to generate the stand-alone code. If the stand-alone code is generated, the user can emulate the application program in the off-line mode after loading to the HT-ICE.

- Generate .COD file

  Check this box to generate the .COD file. This file will be sent to Holtek for manufacturing the masked ROM ICs, after the program emulation and debugging tasks are completed. The OTP programmer also needs this file.

- Generate .OTP file

  Check this box to generate the .OTP file. Refer to Chapter 8 for more details on using the Holtek (OTP) Programmer. The .OTP file contains the data to be programmed into the OTP chip by using HT-OTP and the OTP programmer.

- Assembler options

  The command line options of the HASM 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.

- Linker options

  To specify the options of the HLINK cross linker. Libraries are used to specify the library files refered by HLINK. For example:

  ```
  libfile1, libfile2
  ```

  Section addresses are used to set the ROM/RAM address 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 HLINK is generated.

Debug command

This command sets the options used by the Debug menu (Chapter 5 HT-IDE2000 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.

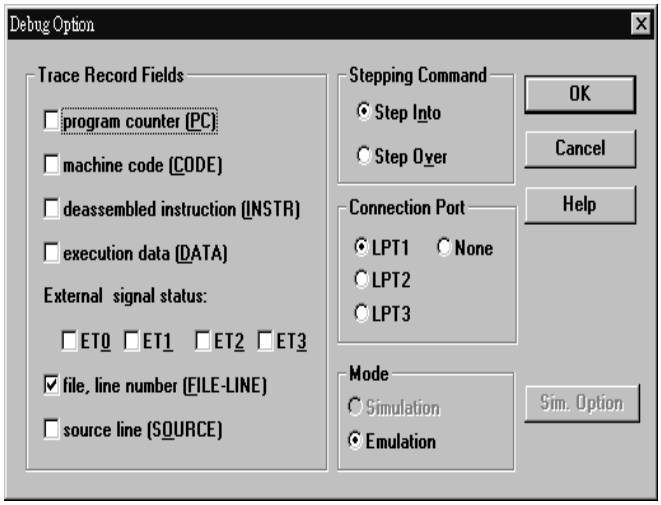


Fig 3-14

- 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 Read data if the execution is a read operation only, and it is Write 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.

- 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-IDE2000 working mode as either simulation or emulation mode. If the HT-ICE is connected to the host machine and powered on, the HT-IDE2000 can be selected to be either in emulation or simulation mode.

Directories command

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

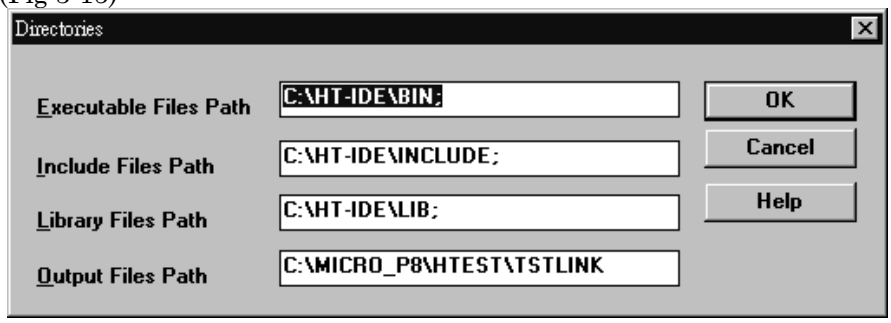


Fig 3-15

- Executable files path
  The search path referred to by the HT-IDE2000 when the executable files are called.
- Include files path
  The search path referred to by the HASM to search for the included files.
- Library files path
  The search path referred to by the HLINK to search for the library files.
- Output files path
  The directory for saving the output files of the HASM (.obj, .lst) and HLINK (.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.

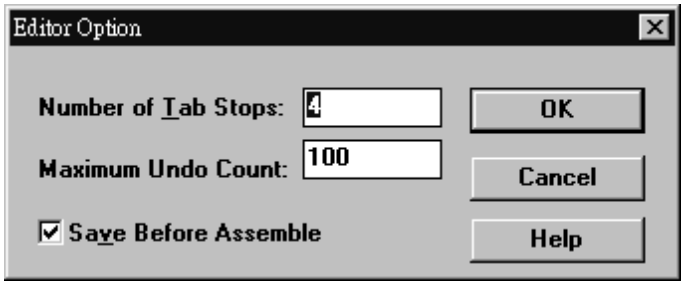


Fig 3-16

Color command

This command sets the foreground and backgroud 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 HASM output.

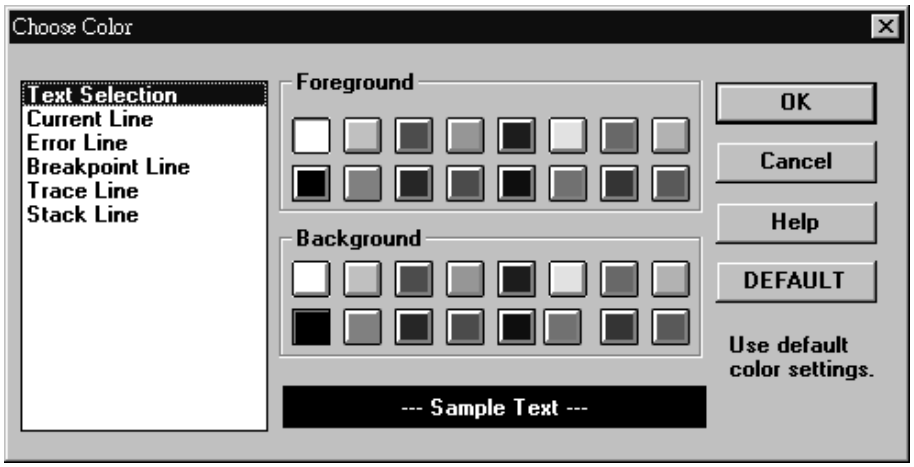


Fig 3-17

Font command

This command will change the displayed fonts.

Chapter 4

HT-IDE2000 Menu – Project

4

The HT-IDE2000 provides an example Project, which will assist first time users in quickly familiarizsing themselves with project development. It should be noted that from the standpoint of the HT-IDE2000 system, a working unit is a project with each user application described by a unique project.

When developing an HT-IDE2000 application for the first time, the development steps as described earlier, are recommended.

Note When setting the project's options, during development it is recommended to select the Debug Mode, which is the default mode.

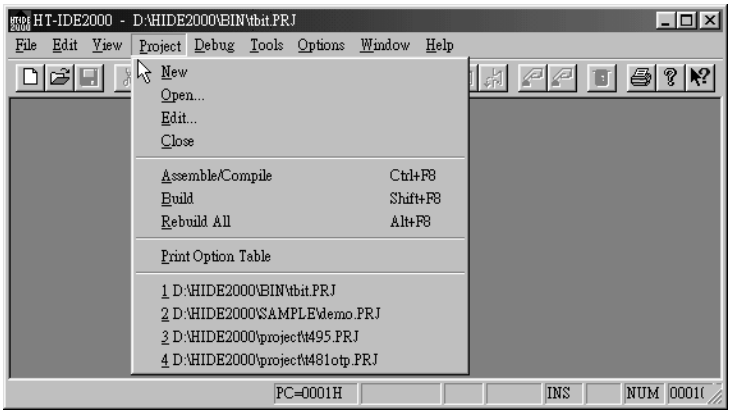
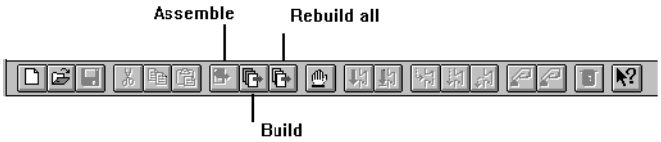


Fig 4-1



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 Microcontrollers.

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

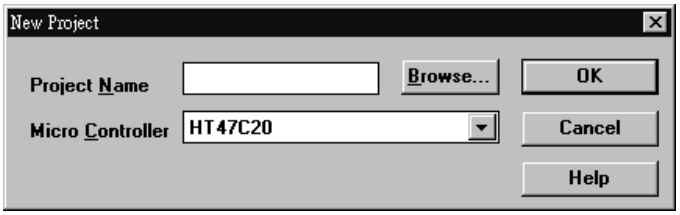


Fig 4-2

Open and Close a Project

The HT-IDE2000 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-IDE2000 window. Otherwise, the results are unpredictable.
The HT-IDE2000 will retain the opening project information if the system exits from the HT-IDE2000 without closing the opening project. This project will be opened automatically the next time the HT-IDE2000 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.

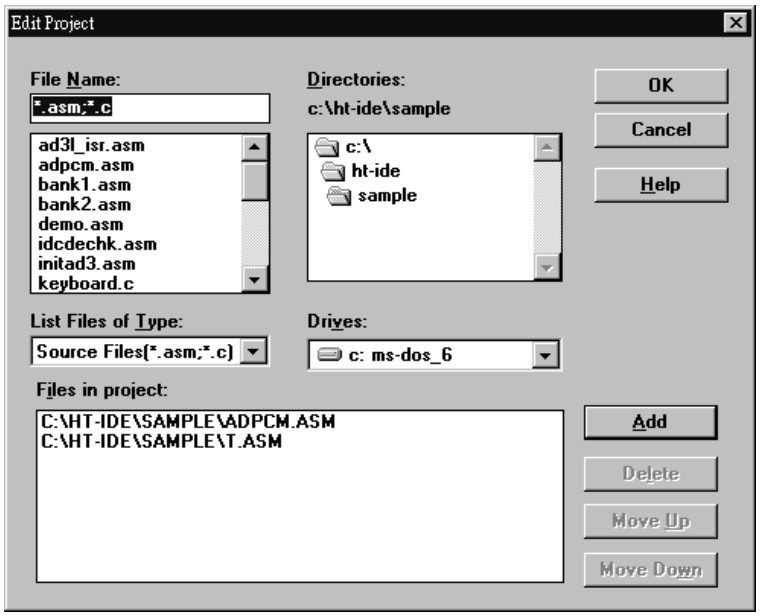


Fig 4-3

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 or ...
- Choose the source file type and browse the List Files.
- Choose the drive and directory where the source files are located 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 μC 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 HASM or C compiler, and generate a task file and a debugging file if the project is in the debug mode (please refer to Options menu, Project command)
- 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-IDE2000 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 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 Linker is called to link all object files of this task and 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 active 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 debug of the application program can begin (refer to the HT-IDE2000 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.

Chapter 5

HT-IDE2000 Menu – Debug

5

In the development process, the repeated modification and testing of source programs is an inevitable procedure. The HT-IDE2000 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 in the debug mode, (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-IDE2000 is now ready to accept and execute the debug commands.

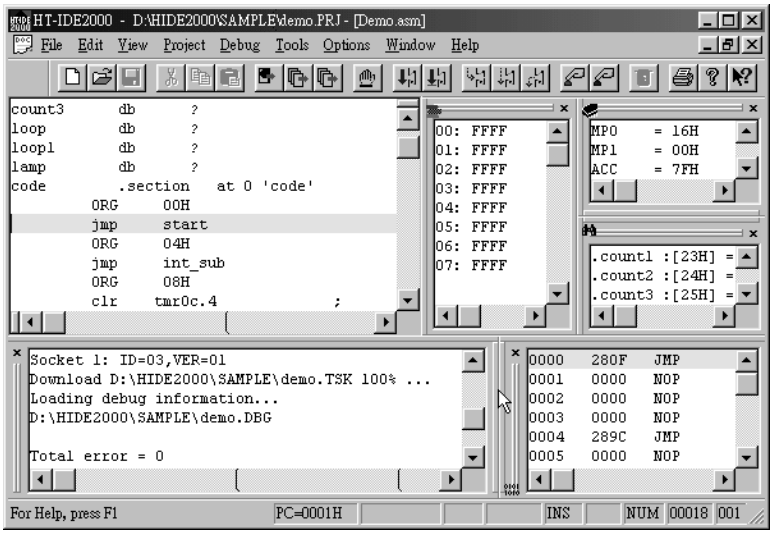


Fig 5-1

Reset the HT-IDE2000 System

There are 4 kinds of reset methods in the HT-IDE2000 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-IDE2000 Debug menu (Fig 5-2)
- Software power-on reset command in the HT-IDE2000 Debug menu (Fig 5-2)

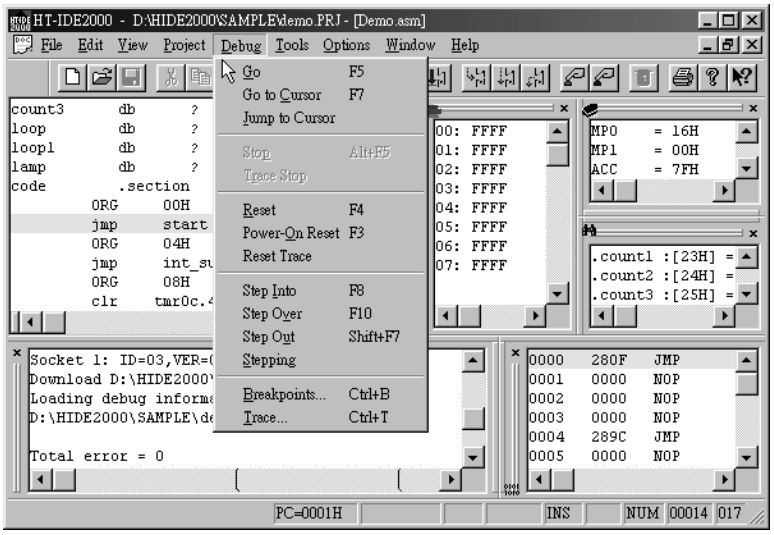
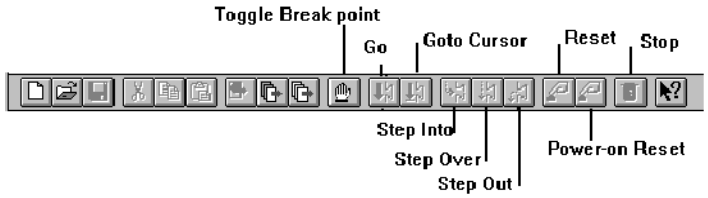


Fig 5-2



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 PD, 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 μC for the effects of registers under the different resets.
(**) : After power-on reset, the μC will check if it is in the stand-alone mode.
If so, the μC will start emulating the application program, otherwise the PC value is 0 and the emulation stops.
(***) : If the reset is from the target board, the μC will start emulating the application after the reset is completed.

PC - program counter
PD - power down flag
TO - time out flag

To reset from the HT-IDE2000 commands

- Either choose the Reset command from Debug menu or click the Reset button on the toolbar to exectute 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 μC 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 in the debug mode 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-IDE2000 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-IDE2000 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 Goto 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-IDE2000 provides two step modes, namely manual mode and automatic mode.

In the manual mode, the HT-IDE2000 executes exactly one step command each time the single-step command is executed. In the automatic mode, the HT-IDE2000 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 in each mode.

- The Step Into command executes exactly one instruction at a time, however upon encountering a CALL procedure, will enter the procedure and stop at the first 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-IDE2000 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-IDE2000 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 ! 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 progam 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 breakpoint list box (Fig 5-3), then the descriptive items must be designated first, then added to the breakpoint 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 Breakpoint 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.

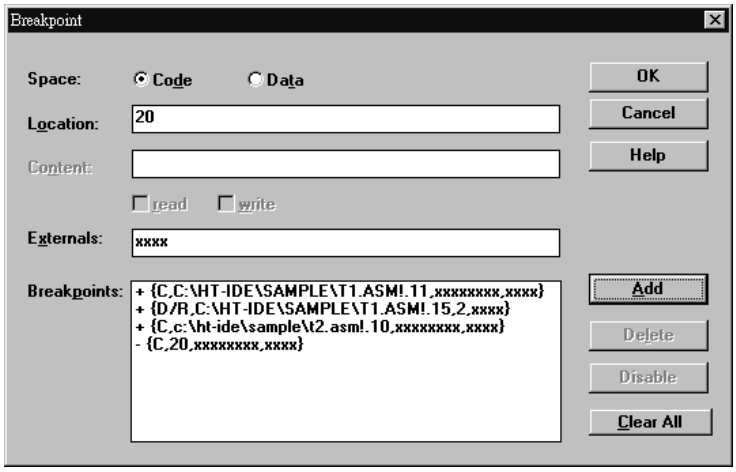


Fig 5-3

→ **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 breakpoint 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

→ **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 breakpoint 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-IDE2000 provides a powerful trace mechanism which records the execution processes and all relative information when the HT-IDE2000 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-IDE2000 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.

About the trace mechanism and its requirements

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 (don't 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: To stop 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 breakpoint list box (refer to the section on Breakpoints, Format of breakpoints list box)

To stop 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.

→ **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

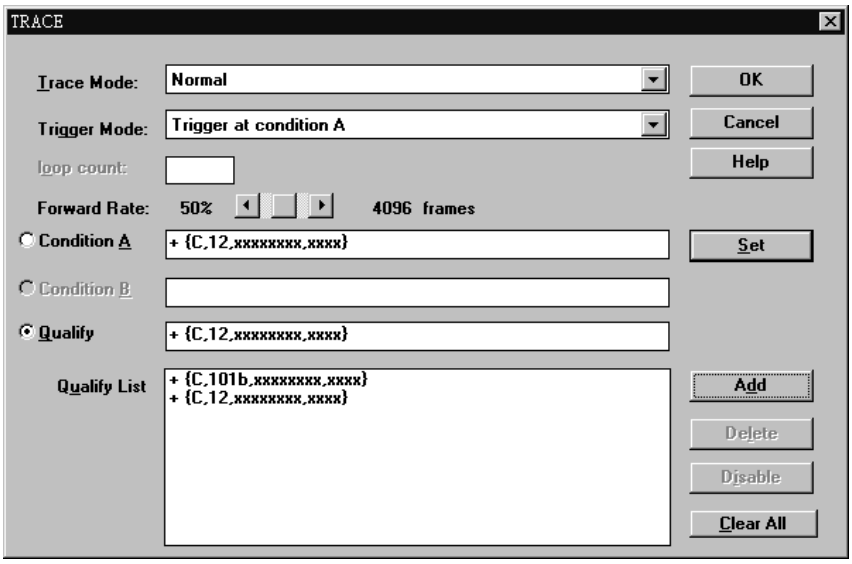


Fig 5-4

→ **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 button
  A Set 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

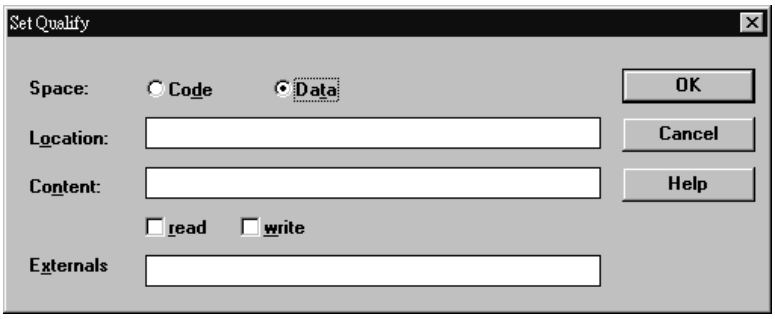


Fig 5-5

→ **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 button
  A Set dialog box is displayed as in Fig 5-5
- Enter the qualifier information
- Press the OK button to close the Set Condition dialog box
- Press the Add button to add the qualifers 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) qualifer 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.

About the 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.

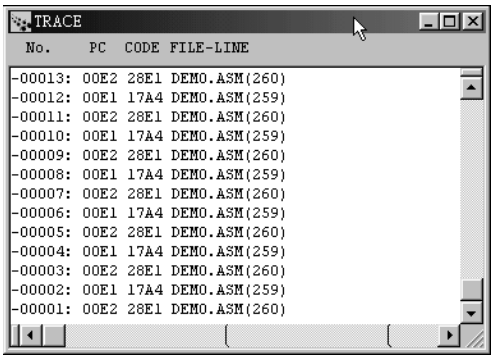


Fig 5-6

- 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 5-6. 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 5-7).
- 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-IDE2000 utility.
- Execution data (DAT)
  The data content to be executed (read/write).
- External signal status (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.

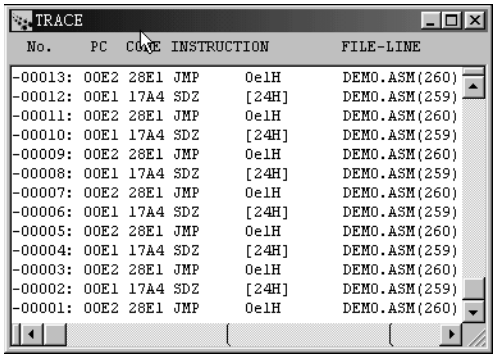


Fig 5-7

Note To set the trace record fields use the Debug sub-menu 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.

Chapter 6

HT-IDE2000 Menu – Window

6

The HT-IDE2000 provides various kinds of window 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, Sim. PAD and Sim. Result.

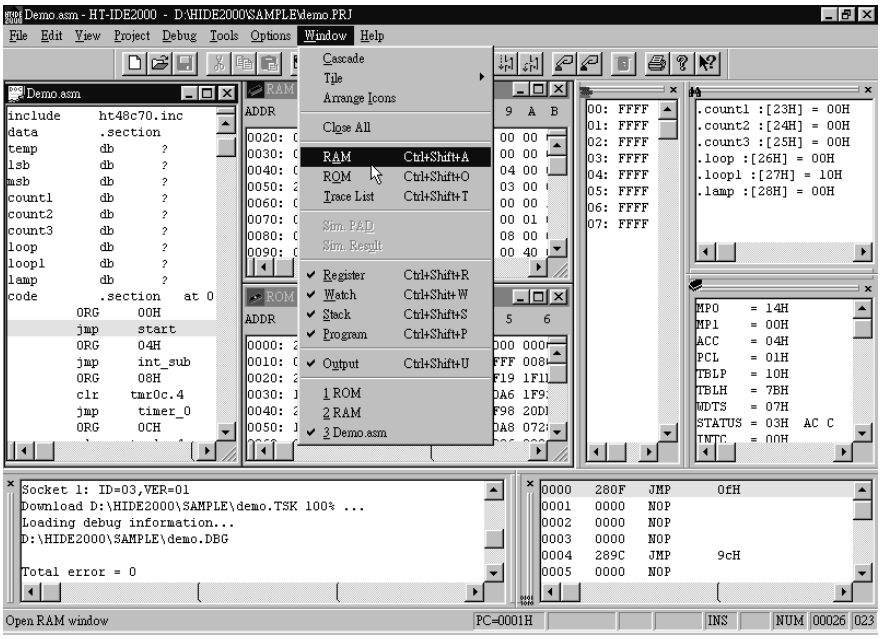


Fig 6-1

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 digit are displayed in hexadecimal format.

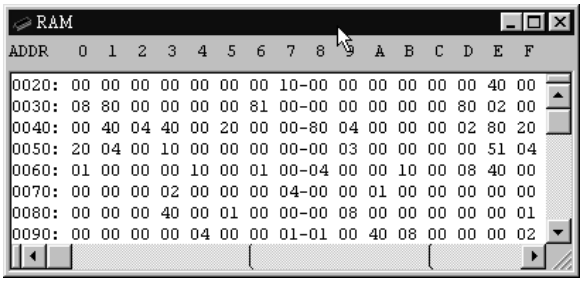


Fig 6-2

- 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 memory size 1 where the memory size is depends upon the uC 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.

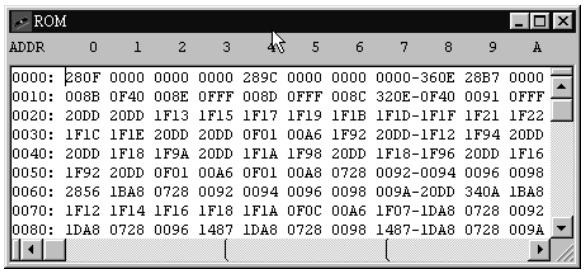


Fig 6-3

• 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.

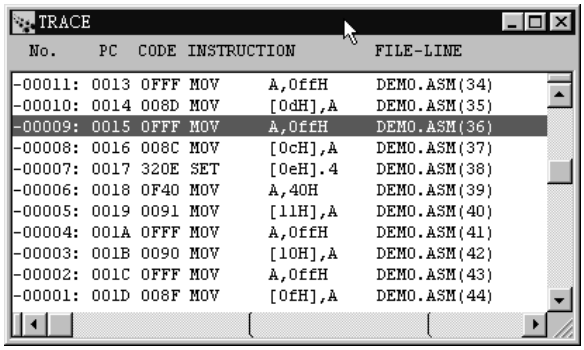


Fig 6-4

• Register

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

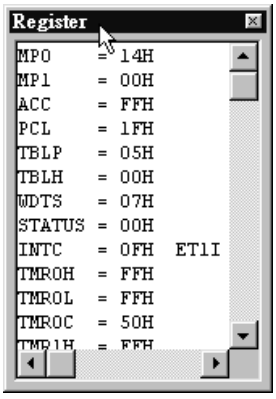


Fig 6-5

- Watch

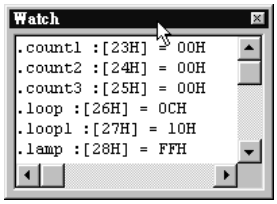
  The Watch window displays the memory addresses and contents of the specified symbols defined in the data sections, i.e., in the RAM space. The format of the symbol is:

  [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-IDE2000 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.

Fig 6-6

- Stack

  The Stack window displays the contents of the stack buffer for the uC selected in the current project. The maximum stack level is dependent upon the uC selected. Fig 6-7 shows an example of the Stack window for the HT48C70 which has an 8 level stack. The growth of the stack is numbered from 0. The number is increased by 1 for a push operation (CALL instructions 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 hex 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.

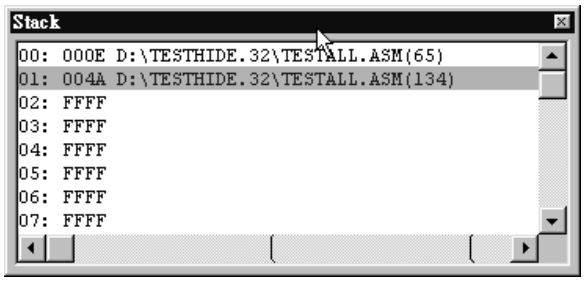


Fig 6-7

- Program

  The Program window displays the code memory or ROM in disassembly format. The address range is from 0 to memory size "1" where the memory size is depends upon the uC selected in the project.
- Sim. PAD

  The Sim. PAD window is used as the interface to set the port inputs when the project is in simulation mode. After setting the port input, the result of an input instruction for this port will remain at the same value until the input level of this port is changed.
- Sim. Result

  The Sim. Result window is used when the project is in simulation mode. This window displays the input/output levels of all ports for the current project.
- Output

  The Output window shows the system messages from the HT-IDE2000 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-IDE2000 provides a simulation mechanism for debugging application programs. The HT-IDE2000 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-IDE2000, 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 μC 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.

Start the Simulation

Upon entering the HT-IDE2000, 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-IDE2000 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-IDE2000 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-IDE2000. Fig 7-1 displays the contents of the Debug command.

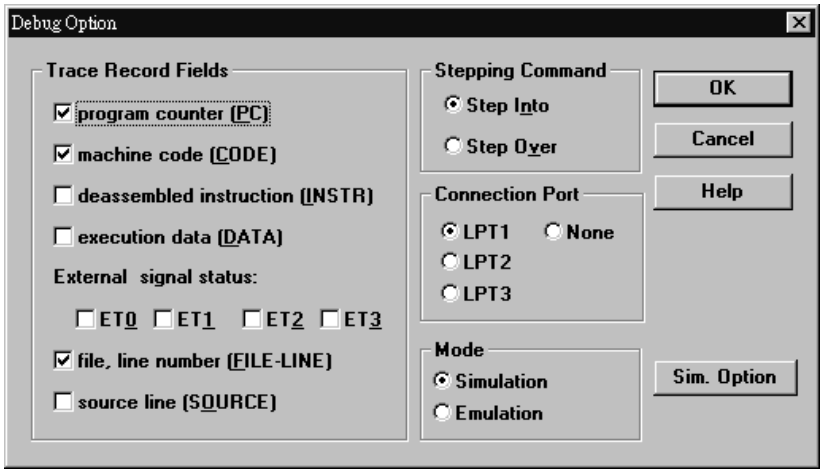


Fig 7-1

In addition to μC 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.

Reference chapter 15 for detail information of VPM.

Chapter 8

Using the OTP Programmer

8

Introduction

The Holtek HandyWriter was specifically developed to program the range of Holtek OTP microcontroller devices allowing users to easily and efficiently burn their programming code into the OTP devices. The advantages of this writer include its small and easy to manage size, ease of installation and easy to use special features. The structure of the writer includes the following components and is shown in Fig 8-1 below:

- Single 40 pin DIP TEXTOOL
- Single 25 pin printer port D-type female connector
- Single 96 pin VME connector

To use the HandyWriter requires the following:

- 16V power adapter with minimum current rating of 500mA. For best purposes please use the adapter included with the HandyWriter carton.
- IBM386 compatible or higher spec. PC
- Win95/98/NT Windows operating system
- HT-IDE2000 microcontroller development system
- If the writer is directly connected to the PC, the HT-ICE is not required.

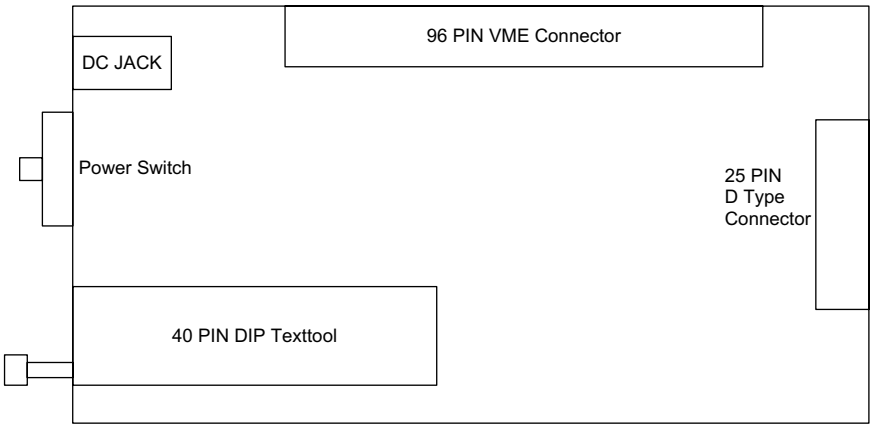


Fig 8-1

Installation

- To directly connect to a PC, use the printer cable to connect from the HandyWriter′s 25 pin D-type connector to the printer port of the PC as shown in Fig 8-2. To connect via the HT-ICE, first connect the HandyWriter to the VME 96 pin socket CN1 on the HT-ICE then connect the HT-ICE to the PC′s printer port using the printer cable as shown in Fig 8-3.
- Install the HT-IDE2000 system software, to do so please consult the HT-IDE2000 User′s Guide

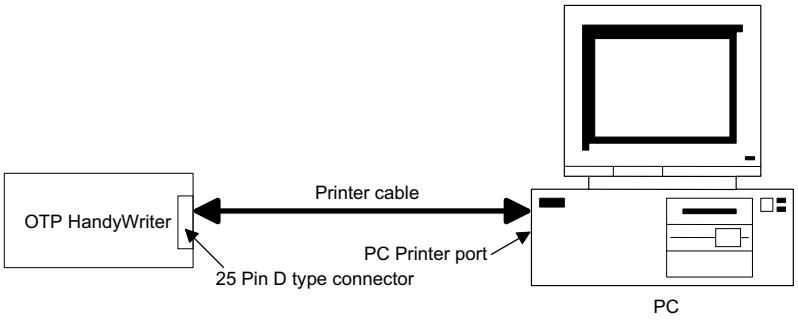


Fig 8-2

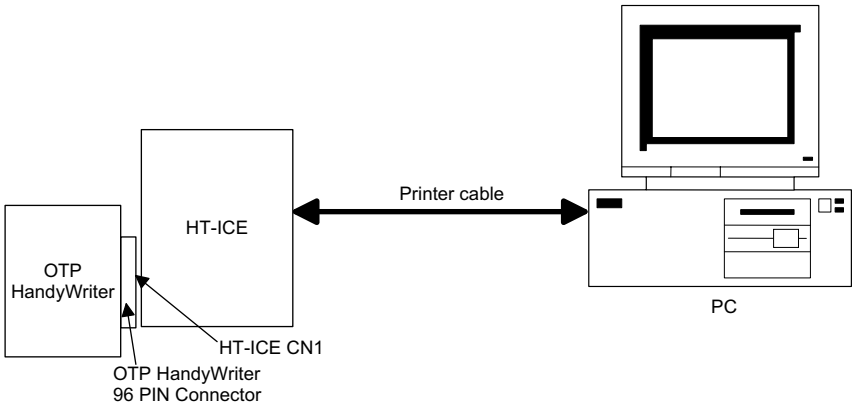


Fig 8-3

Programming an OTP chip with the HandyWriter

→ **Run the HT-HandyWriter system software**

Run the HT-HandyWriter system software under the HT-IDE2000 icon in the main Windows programs menu as shown in the Fig 8-4 below:

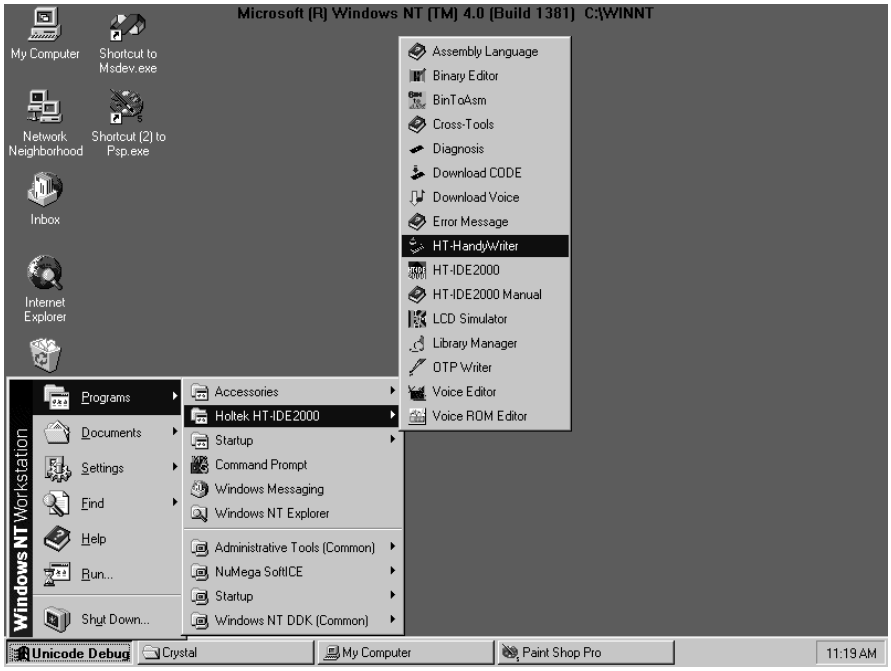


Fig 8-4

→ **LPT — Setup the Printer Port**

After running the HandyWriter program, a window as shown in Fig 8-5 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 8-6 will be displayed. From here LPT1, LPT2 or LPT3 can be chosen. If the OTP HandyWriter 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 8-6. If the OTP HandyWriter is directly connected to the PC printer port then choose the relevant printer port in the same way.

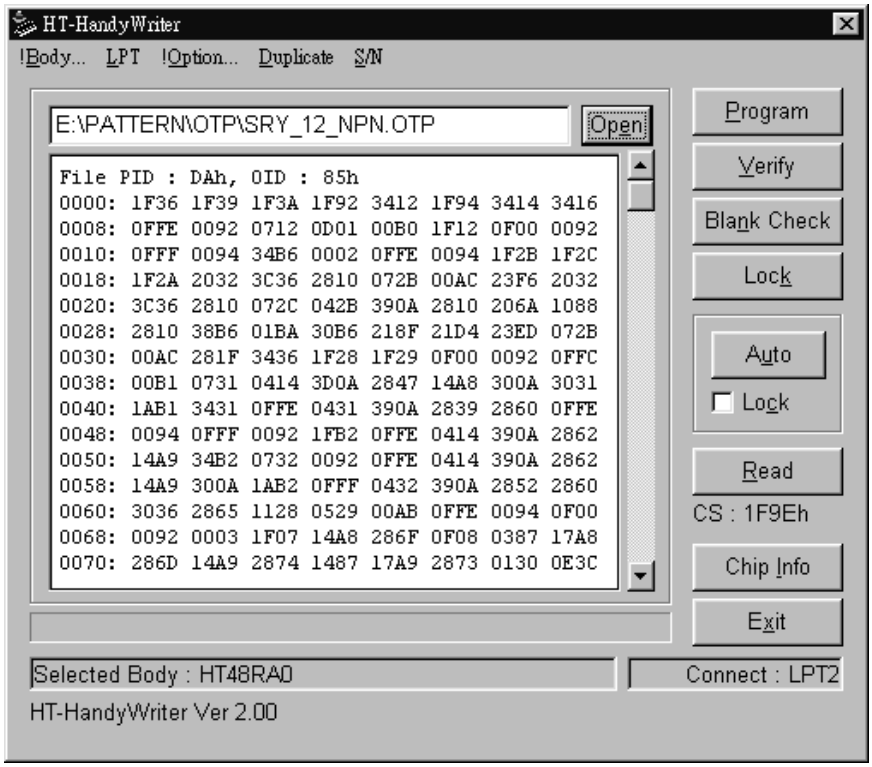


Fig 8-5

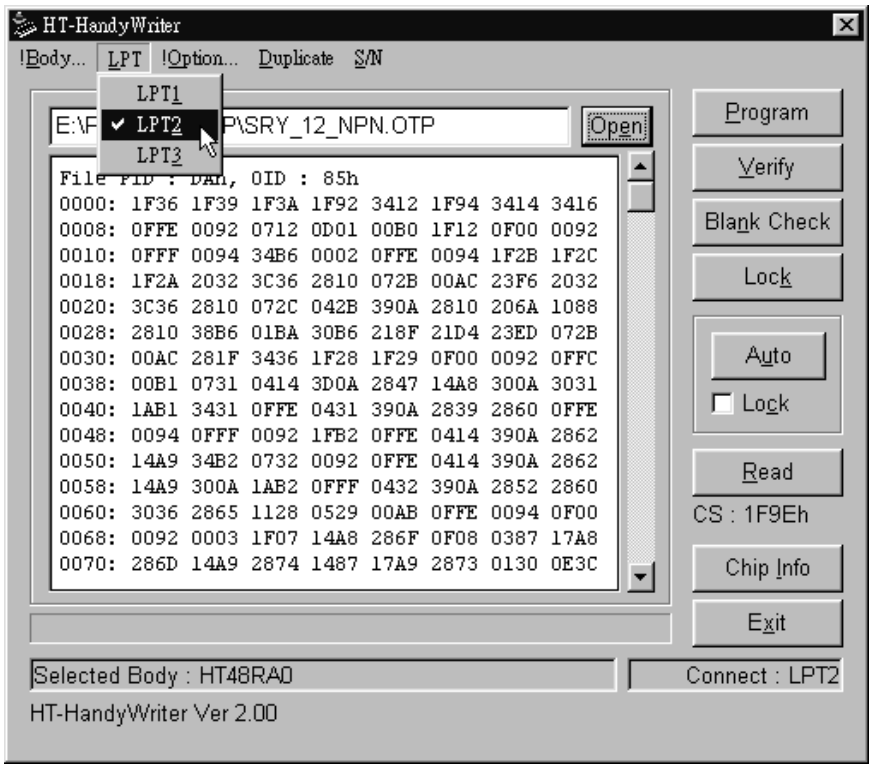


Fig 8-6

→ **!Body — Select the OTP Body Type**

By clicking on "!Body", [Set Body] dialog will be shown as Fig 8-7. If there is no IC type identifier stored in the OTP chip, all the read/write operations will be completed according to the chip type that selected by users.

Fig 8-7

→ **!Option — Check the IC Option**

By clicking on "!Option", a pop-up dialog, as shown in Fig 8-8, will be displayed. It will illustrate the option that comes from opened file or OTP chip content.

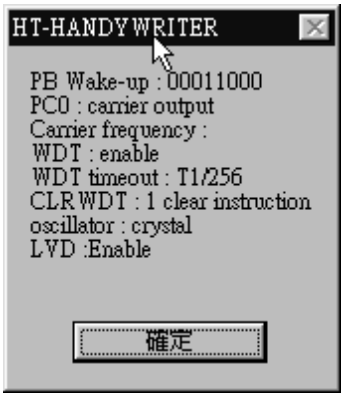


Fig 8-8

→ **HT-HandyWriter Programming Functions**

Fig 8-5 shows the internal functions of the HandyWriter. The 8 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 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 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 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 HandyWriter 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 HandyWriter 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 8-9. The Auto-Program window as shown in Fig 8-10 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 8-11, 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 8-11.

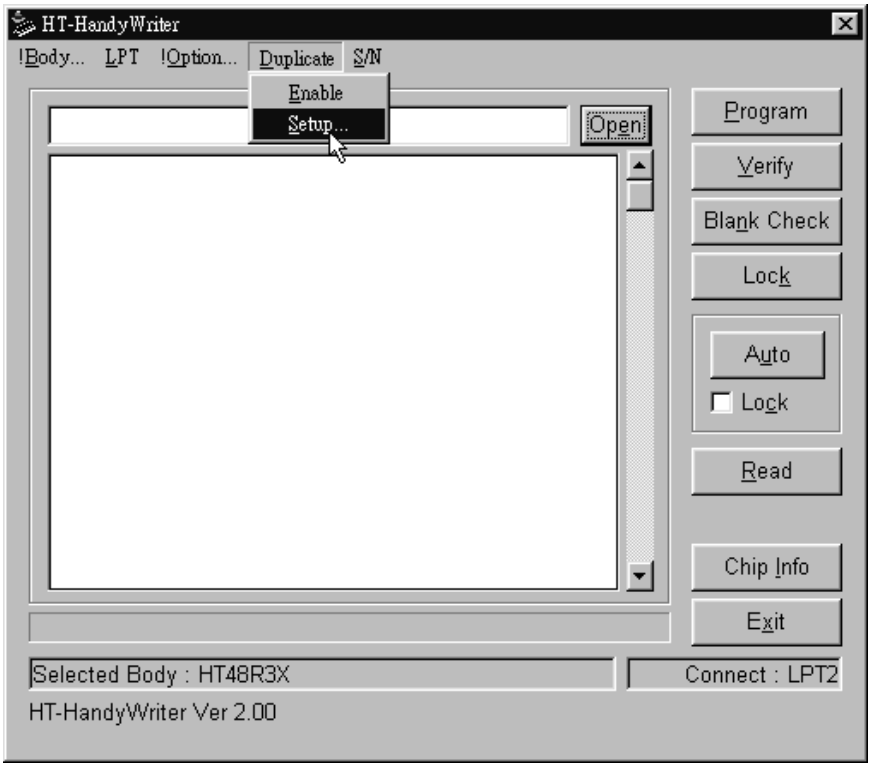


Fig 8-9

Fig 8-10

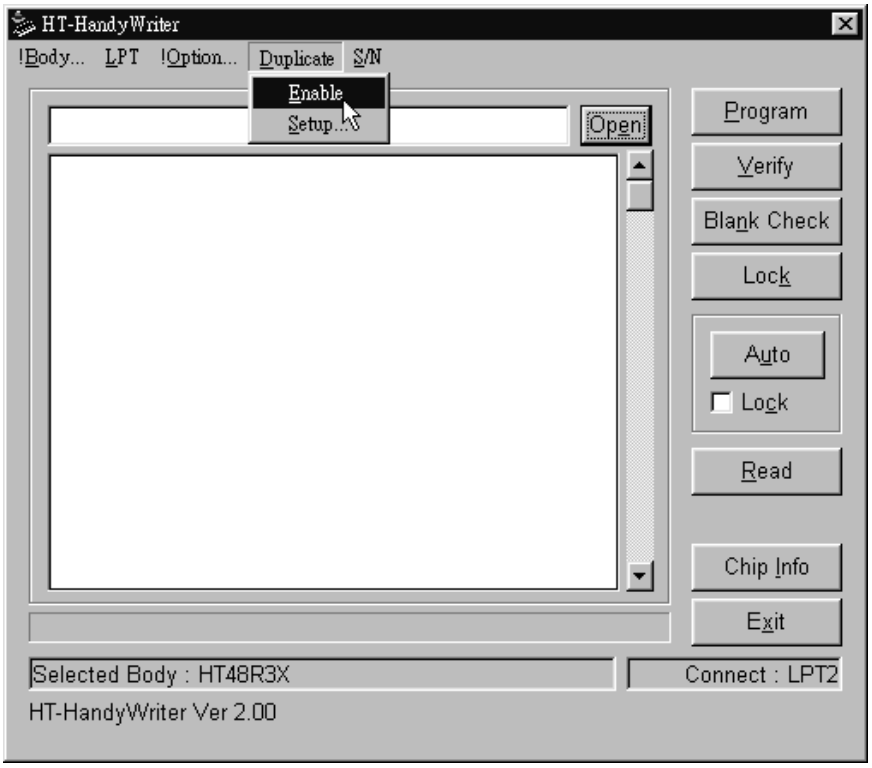


Fig 8-11

- S/N Serial Number Writing

  The Serial Number menu enables an 8-digit hexadecimal serial number to be written into a specially allocated position within the Program ROM. The Serial Number is written into the Lower byte locations of Program ROM addresses 1,2,3 and 7. Location 7 will contain the MSB of the Serial Number while locations 3 and 2 will contain the subsequent locations with location 1 containing the LSB. Note that the Serial Number will be automatically incremented by one after each device has been programmed. If this function is to be used, then the application program must not use these 4 words for other purposes, otherwise the application code will be overwritten during programming.

  First, it is necessary to setup the initial value of the serial number. To do this select the [S/N]/Setup instruction, after which a window, as shown in Fig 8-12 will be displayed where up to 8 hexadecimal digits can be written. The [S/N]/Enable function, as shown in Fig 8-13, should then be chosen to activate this function. The Serial Number will now be displayed on the lower right corner of the main menu. Now, whenever the Program function is activated, the Serial Number will be written into the Program ROM, the value of which will be incremented by one for each subsequently programmed device.

  To de-activate the write Serial Number function, the toggle action [S/N]/Enable function, as shown in Fig 8-13, should once again be selected.

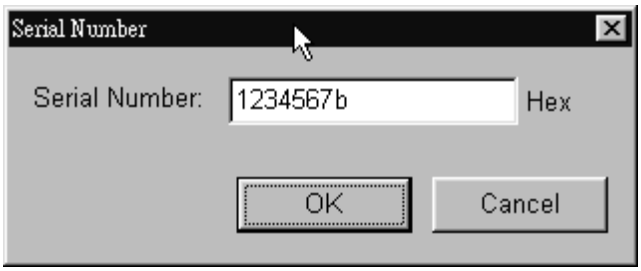


Fig 8-12

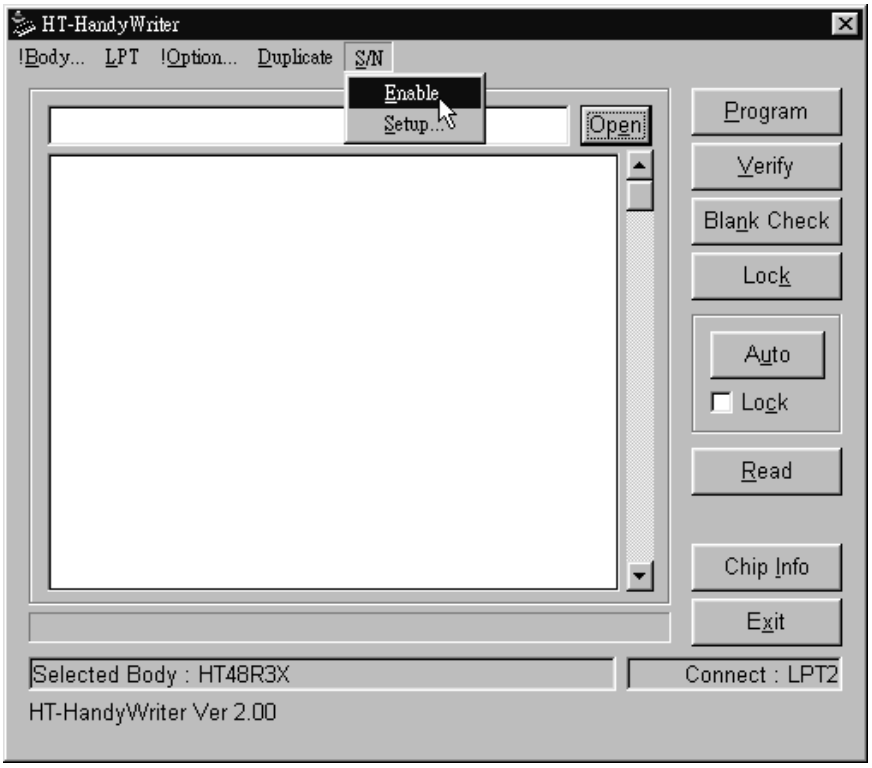


Fig 8-13

System Messages

→ **HandyWriter Connect to LPT1.**
OTP HandyWriter already connected to LPT1.

→ **Cannot connect to ICE**
Connection problems between the HandyWriter, the HT-ICE and the printer port.

→ **Invalid EV Chip!**
The HandyWriter 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 HandyWriter through ICE**
The HandyWriter is successfully connected via the HT-ICE.

→ **Cannot find HandyWriter, please connect it to ICE**
Or this HandyWriter is an old version

The HT-ICE is already connected to the printer port, but the HandyWriter is not connected to the HT-ICE. It may also be that an old version of the HandyWriter is being used (THANDYOTP-A) so the system is unable to detect a good connection. If the former case, please connect the HandyWriter 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 chip 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 chip has 400h of writable space, the file content is 800h, so the HandyWriter can only write 400h of data into the contents of the OTP chip.

→ **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 at the address xxxxh in the OTP chip is not the same as the data yyyyh in the PC ram memory.

→ **Addr: xxxxh, Data: zzzzh**
Not Blank!

The OTP chip is not blank as the address xxxxh contains the data zzzzh, inhibiting the implementation of further instructions.

→ **Chip mismatched!**

The OTP chip presently in the HandyWriter and the OTP chip mentioned in the .OTP file do not match, inhibiting the implementation of further instructions.

→ **Chip is locked!**

The OTP chip presently in the HandyWriter 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 HandyWriter system software.

Part II

Development Language and Tools

Chapter 9

Holtek C Language

9

Introduction

The Holtek C compiler is based on ANSI C. Due to the architecture of the Holtek microcomputer, 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
- Variables
- Constants
- Operators
- Program control flow
- Functions
- Pointers and arrays
- Structures and unions
- Preprocessor directives
- Holtek-C specifics

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 complier. They must be in lower case.

auto	bits	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, register** and **static** are not supported by the Holtek C compiler.

Data types and sizes

Only three basic data types are supported by the Holtek C compiler,

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 range

Data Type	Size (bits)	Range
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;
```

For example:

```
int lcd @ 0×20; /* declare the variable lcd in the offset
0×20 of RAM */
```

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.

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.

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. Since enumerator values are by default assigned increasing from 0, this is equivalent to writing

```
const PORTA=0 ;
const PORTB=1 ;
const PORTC=2 ;
```

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. The entries in the enumeration list are assigned constant integer values. These values are limited within the range 0 to 255. Although variables of the enum type may be declared, the Holtek C compiler will not check whether what was stored in such a variable is a valid value for the enumeration. Nevertheless, the enumeration variables offer the chance of checking and as a result is a better method than **#define**.

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

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 increment	
++	increment	right to left
––	dcrement	
~	complement	
!	not	
–	unary minus	
+	unary plus	
&	address of	
*	dereference	
*	multiply	left to right
/	divide	
%	modulus (remainder)	

Operators	Description	Associativity
+	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 > 128)
{
    word_count=1
    line++;
}
else
    word_count++;
```

→ for statement

- Syntax

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

  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.

- Description

  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 a statement or block of statements. 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
  See the Syntax for **switch** statement
- Description
  A label has the same form as a variable name, but followed by a colon. It can be attached to any statement in the same function as the **goto** statement. 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;
}
```

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.

- Description

  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.

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 equalent 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 refered 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

```
svariable.member1
```

where ***svariable*** 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** equal 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)
```

→ **#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 inclusion: #if #else #endif**

- Syntax

  #if *expression*
  source codes
  [**#else**
  source codes]
  #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 code below the **#if** statement will be compiled. Otherwise, the source code that follows the **#else** statement, if it exists, will be compiled.

- Example

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

→ **Conditional inclusion : #ifdef**

• Syntax

```
#ifdef symbol
    source codes
[#else
    source codes]
#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 code below the **#ifdef** statement will be compiled. Otherwise, the source code that follows the **#else** statement, if it exists, will be compiled.

• Example

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

→ **Conditional inclusion : #ifndef**

• Syntax

```
#ifndef symbol
    source codes
[#else
    source codes]
#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 code below the **#ifndef** statement will be compiled. Otherwise, the source code that follows the **#else** statement, if it exists, will be compiled.

• Example

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

→ **Conditional inclusion: #elif**

- Syntax

```
#if expression1
    source codes
#elif expression2
    source codes
[#else
    source codes]
#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 code that exists below the **#if** statement will be compiled. If **expression1** is zero, then **expression2** is checked to see if it is nonzero. If so then the source codes that follows the **#elif** statement will be compiled. Otherwise, the source code that follows the **#else** statement, if it exists, will be compiled.

- Example

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

→ **Conditional inclusion : defined**

- Syntax

```
#if defined symbol
    source codes
[#else
    source codes]
#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
```

→ **#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 in 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.inc>
#include  ht8270.inc
```

→ **Inline assembly: #asm and #endasm**

- Syntax

  Inline assembly instructions can be included as follows:

```
#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

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

→ #line

- Syntax

  #line *constant* ["*filename*"]

- Description

  The **#line** directive sets the predefined macro __LINE__, for the purpose of error diagnostics or symbolic debugging, such that the line number of the next source line is considered to be the given constant, which must be a decimal number. If the **filename** is given, __FILE__ is set to the file named. If **filename** is absent the remembered file name is not changed.

- Example

```
#line 20  ht48c10.asm
```

→ Machine dependence : #pragma

- Syntax

  #pragma *Token-String*
  #pragma *vector Symbol @ Address*

- Description

  The **#pragma** directive causes machine-dependent behavior when the Token-String is of a form recognized by the C compiler. The **#pragma** directive must end with a semicolon. An unrecognized pragma will be ignored. The vector is a valid pragma which sets up the location, Address, of an interrupt vector and assigns **Symbol** as the name of the vector. If a function of name Symbol is defined, that function executes when the corresponding interrupt occurs.

• Example

```
#pragma vector __INT @ 0x0004
void __INT(void){
...
}
```

Predefined names

Several symbols are predefined and expanded to produce special information. These symbols cannot be undefined or redefined.

__LINE__	A decimal constant containing the current source line number
__FILE__	A string literal containing the name of the file being compiled
__DATE__	A string literal containing the date of compilation, in the form Mmmdd yyyy .
__TIME__	A string literal containing the time of compilation, in the form hh mm ss
__STDC__	The constant 1. It is intended that this symbol be defined to be 1 only in standard-conforming implementation.

Holtek C Compiler Specifics

This section describes some fundamental requirements of the Holtek C compiler language. The topics are:

Using multiple source files
Input/Output ports system calls
Interrupts

Using multiple souce files

The Holtek C compiler supports multiple source files with only one source file containing the main() routine. C source files may be compiled one by one and all the object files linked to an execution file. The Holtek C compiler can compile all source files and link them all together.

Input/Output ports system calls

The Holtek C language provides the following system calls for accessing the input/output ports. These system calls are implemented without call instructions to reduce the number of stacks used.

→ **Input/Output ports**

- unsigned char peekPX()
  Read data from port X, X=A,B,C,D,E,F,G
  - Example
```
unsigned char i;
i= peekPA();  //read input value from the port A, saved in i
```
- void pokePX (unsigned char)
  Write data to port X, X=A,B,C,D,E,F,G
  - Example
```
pokePC (0×00); //write a char 0×00 to the port C
```

→ **Read/Write control registers**

- unsigned char peekPXC()
  Read data from port X control register, X= A,B,C,D,E,F,G
  - Example
```
unsigned char i;
i= peekPEC(); //read an input value from the control register
              //of port E, saved in i
```
- void pokePXC (unsigned char)
  write data to port X control register, X=A,B,C,D,E,F,G
  - Example
```
pokePBC(0×20);  //write a value of 0x20 to the control
                //register of port B
```

→ **Set/Clear bits of ports**

- void setPX()
  set port X, X=A,B,C,D,E,F,G
  - Example
```
setPD();  //set port D
```
- void setPXi()
  set bit i of port X, i=0,1,2,3,4,5,6,7 and X=A,B,C,D,E,F,G
  - Example
```
setPD3();  //set bit 3 of port D
```
- void clrPX()
  clear port X,X=A,B,C,D,E,F,G
  - Example
```
clrPC();  //clear port C
```

- void clrPXi()
  clear bit i of port X, i=0,1,2,3,4,5,6,7 and X=A,B,C,D,E,F,G
  - Example
    ```
    clrPB7(); //clear bit 7 of port B
    ```

→ **Set/Clear bits of port control register**

- void setPXC()
  set port X control register, X=A,B,C,D,E,F,G
  - Example
    ```
    setPDC(); //set port D control register
    ```
- void setPXCi()
  set bit i of port X control register, i=0,1,2,3,4,5,6,7 and X=A,B,C,D,E,F,G
  - Example
    ```
    setPEC3(); //set bit 3 of port E control register
    ```
- void clrPXC()
  clear port X control register, X=A,B,C,D,E,F,G
- vold ClrPXCi()
  clear bit i of port X control register, i=0,1,2,3,4,5,6,7 and X=A,B,C,D,E,F,G
  - Example
    ```
    clrPAC0(); //clear bit 0 of port A control register
    clrPCC4(); //clear bit 4 of port C control register
    ```

Interrupts

The Holtek C language provides a means for implementing interrupts vectors through the preprocessor directive #pragma. The directive #pragma vector is used to declare the name and address of the interrupt vectors. Any function with the same name as the interrupt vector is the interrupt service routine for the vector. The return statement within the interrupt service routine generates a RETI instruction. An example of interrupt is shown as follows:

```
#pragma vector __INT @ 0×0004
void __INT(void){
...
}
```

Difference between Holtek C and ANSI C

Keywords

The following keywords and qualifiers are not supported:

```
Keywords: float     double
Qualifiers: auto    register     static
```

Variables

All variables are static. The operator '@' can be used to specify the address of variables in the general purpose data memory. The offset of the memory starts from 0×20. The syntax is:

data_type varaible_name **@** *memory_location*

For example:

```
unsigned char flag @ 0×25; /* declare the flag in the offset
                              0x25 of RAM */
```

Constants

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

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

Functions

Avoid using reentrant and recursive code.

Arrays

Holtek C allows one dimensional arrays only. An array should be located in a contiguous block of memory and must not have more than 256 elements.

Constant variables

Constant variables must be declared in global scope and be initialized when declared. The size of all constant variables is limited to 255 bytes in the current version.

Initial value

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

For 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[]="abcde";
```

Multiply/Divide/Modulus

The multiply, divide and modulus ("*", "/", "%") operators are implemented by system calls. It is necessary to include the math.lib library if these arithmetical operators are used. To include a library, select [options] in the main menu, select [project...], put the library name in the [libraries] field.

Stack

Because the Holtek HT48CX0 microcontrollers have from 2 to 8 stacks the programmer needs to consider the function call depth to avoid stack overflow. The multiply, divide and modulus of the Holtek C language are implemented by "call" instructions, taking one stack. The input/output port system functions are implemented without "call" instructions.

Operator/System Function	Stack Needed
main ( )	0
*	1
/	1
%	1
peekPX(), X=A,B,C,D,E,F,G	0
peekPXC(), X=A,B,C,D,E,F,G	0
pokePX(), X=A,B,C,D,E,F,G	0
pokePXC(), X=A,B,C,D,E,F,G	0
setPX(), X=A,B,C,D,E,F,G	0
setPXi(), i=0,1,2,3,4,5,6,7 and X=A,B,C,D,E,F,G	0
setPXC(), X=A,B,C,D,E,F,G	0

Operator/System Function	Stack Needed
setPXCi(), i=0,1,2,3,4,5,6,7 and X=A,B,C,D,E,F,G	0
clrPX(), X=A,B,C,D,E,F,G	0
clrPXi(), i=0,1,2,3,4,5,6,7 and X=A,B,C,D,E,F,G	0
clrPX(), X=A,B,C,D,E,F,G	0
clrPXi(), i=0,1,2,3,4,5,6,7 and X=A,B,C,D,E,F,G	0
constant array	1

Holtek C Compiler

The Holtek C compiler supports the ability to write programs using mixed languages. To do this however there are certain rules that need to be followed.

ASM calls C functions

Variables in C language are case sensitive but they are non-case sensitive during the Holtek assembly process. As a result the name of the C function must be declared in capitals. Instead of using the stack, the Holtek C compiler uses RAM to pass arguments to functions. To call C functions in an assembly program, the names of functions and arguments must be declared as external. The names of C functions are translated into the name preceded with an under score. The names of arguments are translated into the function name following the number of the argument occurring. It is also necessary to declare all the internal variables (TMP, TREG, LH, LL, TLH, TLL, AREGTMP_) used by the C functions. Since the Holtek C compiler supports multiple banks, it is necessary to adjust the BP before calling the C function.

To get the return value of the C function it is necessary to know the data size of the return value. If its size is one byte then the return value is stored in a register. If it is two bytes then the high byte is stored in LH and the low byte stored in a register.

- Example 1:

  Assume there is a C function

  ```
  void MAXMIN(long val)
  ```

  We want to call it from assembly. Be aware the function name MAXMIN is capital.

  _MAXMIN ← the function name

  MAXMIN0 ← the name of val argument

```
; Declares the function name and argument
;================================
EXTERN _MAXMIN:NEAR
EXTERN MAXMIN0:BYTE    ;It is ok to declare "byte" although val
is "long" two bytes


; Declares internal variables used by C function
;================================
CDataTmp .SECTION  data
PUBLIC TMP
PUBLIC TREG
PUBLIC LH
PUBLIC LL
PUBLIC TLH
PUBLIC TLL
PUBLIC AREGTMP_
TMP          DB ?
TREG         DB ?
LH           DB ?
LL           DB ?
TLH          DB ?
TLL          DB ?
AREGTMP_     DB ?
code .section "code"
; Sets arguments
;============
MOV A,0ah
MOV MAXMIN0,A
MOV A,00h
MOV MAXMIN0[1],A
```

```
; Set BP
;=====
MOV A,    01FH
ANDM A,   [04H]
MOV A,    HIGH _MAXMIN
AND A,    0E0H
ORM A,    [04H]

; Call C function
;===========
CALL _MAXMIN
```

- Example 2:
  Assume there is a C function
  `long CFUNC(int a, int b, int c)`
  We want to call it from assembly and put the return value in retval variable.
  _CFUNC ← the function name
  CFUNC0 ← the name of a argument
  CFUNC1 ← the name of b argument
  CFUNC2 ← the name of c argument

```
; Declares the function name and argument
;==============================
EXTERN _CFUNC:NEAR
EXTERN CFUNC0:BYTE
EXTERN CFUNC1:BYTE
EXTERN CFUNC2:BYTE

; Declares internal variables used by C function
;================================
CDataTmp .SECTION  data
PUBLIC TMP
PUBLIC TREG
PUBLIC LH
PUBLIC LL
PUBLIC TLH
PUBLIC TLL
```

```
PUBLIC AREGTMP_
TMP          DB ?
TREG         DB ?
LH           DB ?
LL           DB ?
TLH          DB ?
TLL          DB ?
AREGTMP_     DB ?

retval   DW  ?   ;to keep return value

code .section  code
 .
; Sets arguments
;============
MOV A,01h
MOV CFUNC0,A
MOV A,02h
MOV CFUNC1,A
MOV A,03h
MOV CFUNC2,A

; Set BP
;=====
MOV A,    01FH
ANDM A,   [04H]
MOV A,    HIGH _CFUNC
AND A,    0E0H
ORM A,    [04H]

; Call C function
;===========
CALL _CFUNC
```

```
;get return value
MOV retval,A    ;store low byte
MOV A,LH
MOV retval[1],A ;store high byte
C calls ASM functions
```

- Example 3:
  In assembly programs

```
public _sum
public sum0, sum1
d1 .section  data
sum0 db ?
sum1 db ?
c1 .section  code
_sum  proc
  mov a, sum0
  add a, sum1
  ret
_sum  endp

public _join
public join1, join2
extern LH:byte
d2 .section  data
join1 db ?
join2 db ?
c2 .section  code
_join  proc
  mov a,join0
  mov LH, a
  mov a, join
  ret
_join  endp
```

In C program

```
int SUM(int,int);
unsigned long JOIN(int, int);
void function()
{
  int  a,b,c;
  unsigned long  d;
  a = 0x10;
  b = 0x20;
  c = SUM(a, b);
  d = JOIN(a, b);
{
```

Chapter 10

Assembly Language and Cross Assembler

10

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, giving directions to the 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
	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* [;]
[*optional items*]	In the above command line, options and; are both optional, but filename is required, except for the following two cases: 1. Brackets in the instruction operands. In this case, the brackets refer to memory address. 2. Brackets in the interactive command mode. In this case, they are message separators.

Example of convention	Description of convention
{*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.
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
- ? and $ 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 assembler, specifying the manner in which the 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 controlling 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 assembler. Any text following a semicolon is considered a comment.

Assembly Directives

Directives give direction to the assembler, specifying the manner in which the 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 by the directive EQU or the option /D of the command line.

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. HASM 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 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 subsequenct 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 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 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.

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**

.CHIP *description-file*

- Description

  This directive enables the instruction set of the given microprocessor. The description-file is the name of a file which contains all information required by the Cross Assembler to assemble the source file. If no option /CHIP=description-file exists in the command line while assembling, this directive has to be used within the source file, otherwise an error will occur.

Note This directive must be located before the first instruction statement.

→ **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 one instruction unit (14 bits for HT48100). **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 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.

Note Multiple sections can be defined in a source file, but any two sections with the same complete name are not permitted.

→ **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 microcontroller, according to the directive .CHIP. 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.

→ **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, defines 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 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 microcontroller, specified by the directive **.CHIP** or the command line parameter /CHIP=. The Cross Assembler will clear any redundant bits; expression1 has to be a value or a label. This directive may also be employed to set up 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 assembler uses the default radix (decimal). Table 9.1 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

Table 9.1 Digits Used With Each Radix

→ **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, 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 for chksum, bit 0 of location 3 for flag1, location 4 for sbuf and bit 1 of location 3 for cflag.

→ **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 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 that 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.

The syntax of a macro definition is as follows:

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

The assembler supports a directive LOCAL for the macro definition. Its syntax is

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

The LOCAL directive defines symbols available only in the defined macro. The dummy-name is a temporary name that is replaced by an 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 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 1 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 2.10
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 microcontroller used in the source program. The microcontroller type needs to be specified prior to the first instruction statement in the source program by using the directive .CHIP.

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 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 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

  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 and HIGH operator
  - Syntax

    LOW *expression*
    HIGH *expression*

The LOW/HIGH operator returns the value of an expression if the result of the expression is an immediate value. The LOW/HIGH operators will then take the low/high byte of this value. But if the expression is a label, the LOW/HIGH operator will take the values of the low/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 micro-controllers. (Note: The format of the BP might be different between micro-controllers.)

    Example 1:

```
mov A, BANK label
mov BP,A
jmp label
```

    Example 2:

```
mov A, BANK var
mov BP,A
mov A, OFFSET var
mov MP1,A
mov A,R1
```

- Operator precedence

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

Miscellaneous

Forward references

The 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 $digit. The digit can be '0', '1' ... To '9'. 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 assembler will transfer every local label into a unique label before assembling the source file. At most 10 local labels can be defined between two consecutive labels. The following is an example.

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

Reserved assembly language words

The following table lists all reserved words used by the assembly language.

- Reserved Names (directives, operators)

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

- 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

Assembler Options

The Cross Assembler options can be set via the Options menu Project command. The Assembler Options is located on the center part of the Project Option dialog box, as shown in Fig 3-13.

The symbols could be defined in the *define symbol* edit box. The syntax is

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

for 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 listing file (4 digits). A line number is generated only if a cross-reference file is required
- The 2nd field – offset – is the offset from the beginning of the current section to the code (4 digits)
- The 3rd field – code – is present only if the statement generates code or data (two 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 addres (Linker must resolve)
  E → external symbol (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 (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
12345678901234567890123456789012345678901234567890123456789012345678901...
llll oooo  hhhh hhhh E  C source-program-statement
                    R  n
```

llll → line number (4 digits, right alignment)
oooo → offset of code (4 digits)
hhhh → two 4-digits for opcode
C → statement from included file
n → Macro expansion line
E → external reference
R → relocatable name

→ Summary of assembly

The total warning number and total error number is the information provided at the end of the 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 2.04   page
1    0000         .chip  ht48100
2    0000
3    0000         page   60
4    0000         message          Sample Program 1
5    0000
6    0000         pa     equ    [12h]
7    0000         pac    equ    [13h]
8    0000         pb     equ    [14h]
```

```
9  0000           pbc    equ    [15h]
10 0000           pc     equ    [16h]
11 0000           pcc    equ    [17h]
12 0000
13 0000           data   .section         'data'
14 0000 00        b1     db ?
15 0001 00        b2     db ?
16 0002 00        bit1       dbit
17 0003
18 0000           code   .section         'code'
19 0000 0F55      mov    a, 055h
20 0001 0080 R           mov    b1, a
21 0002 0FAA             mov    a, 0aah
22 0003 0080 R           mov    b2, a
23 0004 0F00             mov    a, 0
24 0005 0093             mov    pac, a
25 0006 0095             mov    pbc, a
26 0007 0097             mov    pcc, a
27 0008 0700 R           mov    a, b1
28 0009 0092             mov    pa, a
29 000A 0700 R           mov    a, b2
30 000B 0094             mov    pb, a
31 000C                  end


    0 Errors
```

Chapter 11

Cross Linker

11

What the Cross Linker Does

The Cross Linker, creates task programs from the object files generated by the Cross Assembler or the Holtek 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 HT-IDE2000 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, 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 12 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 name 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 3.20
Copyright (C) HOLTEK Microelectronics INC. 1997-1998. 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>
```

HLINK Task File and Debug File

One of the Cross Linker's output files is a task file which consists of two parts, a task header and binary code. The task header contains the Cross Linker version, the microcontroller name and the ROM code. 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 HT-IDE2000 debugging program. This information includes source file names, symbol names and line number as defined in the source files. The HT-IDE2000 will refer to the symbolic debugging function information. This file should not be deleted unless the debugging procedure is completed, otherwise the HT-IDE2000 will be unable to support the symbolic debugging function.

Chapter 12

Library Manager

12

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). 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 Set Up 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 12-1. Fig 12-2 shows the dialogue box for processing the functions of the Library Manager.

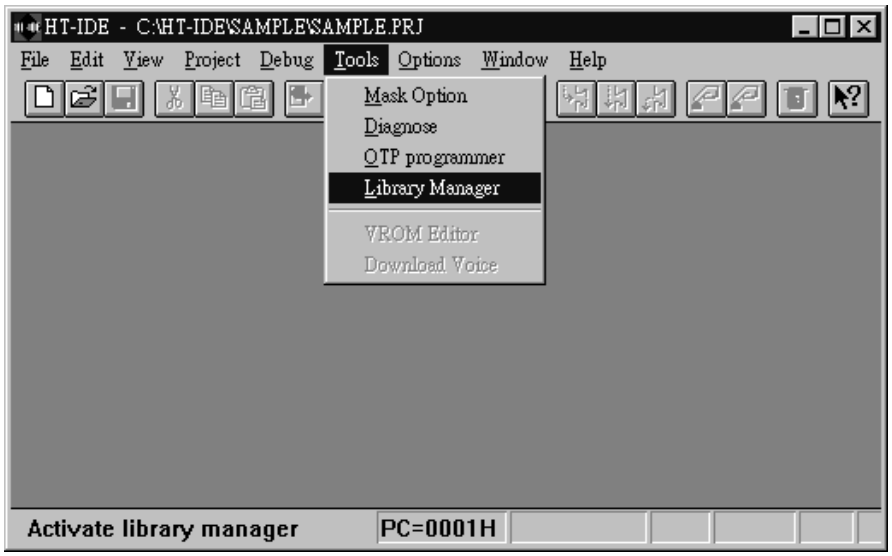


Fig 12-1

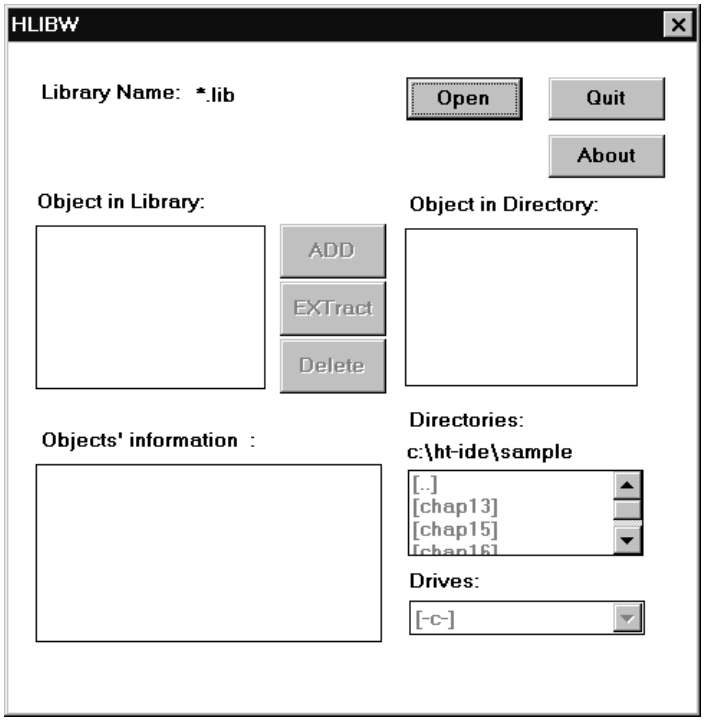


Fig 12-2

Create a new library file

Press Open button, Fig 12-3 is displayed

Type in a new library file name and press the OK button, Fig 12-4 is displayed for confirmation. If the Yes button is chosen, a new library file will be created but will not contain any program modules.

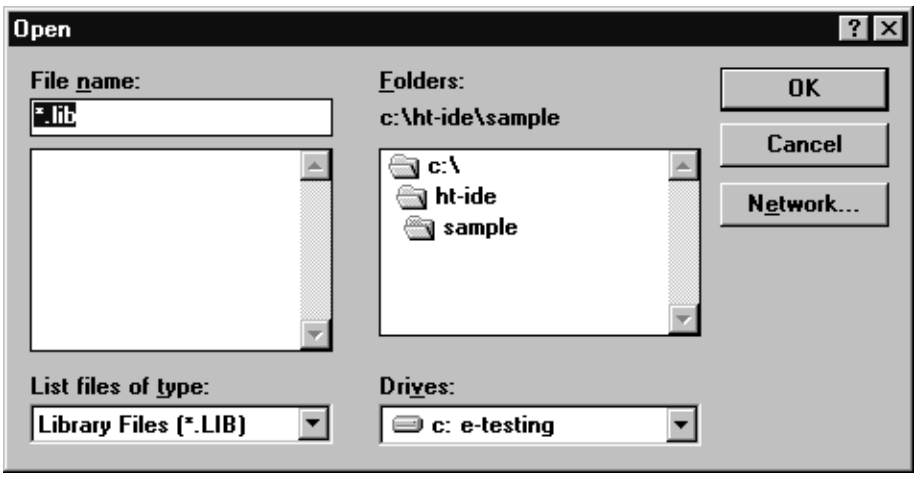


Fig 12-3

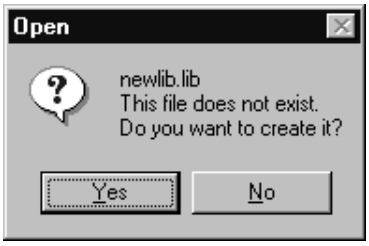


Fig 12-4

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 12-3 is displayed. Select a library file from the box below the File name box, press OK button. From Fig 12-2, 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 name of all public symbols in this object module.

Part III

Utilities

In addition to the previously discussed general purpose 8-bit microcontroller development tools, Holtek also supplies several other utilities for its range of special purpose Voice and LCD microcontroller 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 section contains all the information needed to program and debug relevant applications quickly and efficiently.

Chapter 13

μC VROM Editor (HT-VDS827)

13

Introduction

The HT827XX series of processors are 8-bit high performance microcontrollers with voice synthesizers and tone generators. Holtek provides a HT-VDS827 utility software package that compresses speech source files, with formats such as .WAV, .PCM or ADPCM files, in order to save voice ROM space. An uncompressed binary file can also be loaded as an LCD pattern file.

In the following sections, a quick overveiw for the μC voice microcontrollers is given. The voice ROM editor, HT-Voice Editor and HT-Bin Editor is explained in more detail.

Quick Start for μC Voice Microcontrollers

The purpose of this section is to quickly obtain familiarisation with the μC Voice Development System. If this is the first time that the system is used we strongly recommend that you read this section first and follow the *step-by-step guide*.

In this section, a step-by-step guide is first introduced, followed by the resources supported by the system and several examples which are explained in detail. Finally, in the last part of this section, some useful information is provided.

Step-by-Step guide

In order to create a voice microcontroller application, it is necessary to create a new project and edit sound tracks to be used within the program, before writing the application program.

Taking the HT82770 as an example:

→ **Step 1**

Enter the HT-IDE2000

- For Windows 3.1
  - C:win <Enter>
  - Double click the icon Holtek HT-IDE2000
  - Double click the icon HT-IDE2000, the HT-IDE2000 environment is displayed
  - Press the [OK] button when the dialog box Simulator is connected is displayed
- For Windows 95
  - Click the Start button, select Programs, then select Holtek HT-IDE2000
  - Click the HT-IDE2000 icon
  - Press the [OK] button when the dialog box Simulator is connected is displayed

→ **Step 2**

Create a new project

- Select the Project menu, New command.
- Type the project name (example: Sample) and select the Microcontroller (Example HT82770).
- Press the [OK] button. The project SAMPLE will be created and a Mask Option window (Mask Option for HT82770) is displayed.
- Change the option selection according to the application project.
- Press the [SAVE] button to save the mask options.

→ **Step 3**

Add the voice sources

- Invoke the voice ROM editor, HTVDS827.EXE, by selecting the Tools menu, then the VROM Editor command.
  The Body Type box located on the left bottom side will display 'HT82770' which means that voice sources for the HT82770 will be used.
- Add the voice sources
  Browse from the right hand side of the main window to locate the voice sources and add some into the Sound List box by pressing the Add button which is located in the centre of the main window.
- Store the voice ROM data
  Store the file by selecting the File menu, Save command of the HTVDS827.EXE.

- Quit the HTVDS827.EXE.

→ **Step 4**

Download the voice ROM data by selecting Tools menu, Download Voice command. The voice ROM data is now downloaded into the HT-ICE.

→ **Step 5**

Create/Modify the source files for the application. Here a source file must be provided. The following is an example ...

```
; t.asm                      ; 1
;                            ; 2
#include voice.inc            ; 3 include the file  voice.inc
code .section at 0  code     ; 4
    org    0                 ; 5
    jmp    begin             ; 6
    org    8                 ; 7
    jmp    AdpcmISR          ; 8 specify the ISR for sampling rate
begin:                       ; 9
    speech 0, 12H, 1         ; 10
    set    [22H]             ; 11 turn on the volume
    jmp    $                 ; 12
    end                      ; 13
```

This program is used to play the first track which is edited by the VROM Editor.

→ **Step 6**

Add the source files to the project

Add the above example, T.ASM, into this new project.

- Select the Project menu, Edit command
- Double click the file t.asm created on the previous step.

→ **Step 7**

Change the HT-IDE2000 working mode.

The HT-IDE2000 shall be working in emulation mode.

- Select the Options menu, Debug command
- Click the Emulation radio button in the Mode box

→ **Step 8**

Build the project

- Select the Project menu, Rebuild All command
  If everything is OK, the current line cursor will stop on the first line to be executed.

The program code has now been downloaded into the HT-IDE2000 and the voice ROM is ready. The program can now be executed.

The first track added into the voice ROM can now be heard. It is now replayed by the μC Voice System.

Resources supported by the development system

In this section, all supported resources are first introduced first after which the details behind their usage is described. The section conlcludes with a quick reference of all supported resources.

→ Compressing/Decompressing algorithms provided

Once a μC voice type IC project has been completed, for example for the HT82700, the Voice ROM Editor (HTVDS827.EXE) is then ready to edit sound tracks. There are four different compressing algorithms supported by the Voice ROM Editor, such as 3-bit ADPCM, 4-bit ADPCM, 6-bit PCM and 8-bit PCM. Different compression methods can be specified to compress sound tracks. If the Voice ROM editor is invoked and the voice ROM data is stored, several files are created. Among these files, the *voice.inc* contains the compression method used to compress tracks. According to this file, the development system can link to some corresponding decompressing algorithms. There are other functions or control flags defined in this file. With these decompressing algorithms, short and simple program can easily be written to control track playing. This file must be included in the program, if these decompressing algorithms are to be used.

Because each of these supporting decompression algorithms consumes a significant amount of limited system resources, they cannot all be combined together. That is, 3-bit ADPCM and 4-bit ADPCM cannot be combined. With this excepton, any other supported decompressing algorithms can be combined together to get the best quality for the application. The μC Voice System also provides a quick mode for either 3-bit ADPCM or 4-bit ADPCM. Though decompressing in this quick mode will consume a little more program ROM space, the sampling rate could be much higher than in the normal mode. The default status is in the normal mode.

The following lists the resources used and the maximum sampling rates of each decompression algorithm.

Decompression Algorithm	Resources Used			Maximum Sampling Rate	Possible Combined Algorithms
	RAM	ROM, total	ROM, last page		
3-bit ADPCM	18	333	49	8.0kHz	ADPCM3, PCM6, PCM8
3-bit ADPCM, QUICK	15	459	196	11.0kHz	ADPCM3, PCM6, PCM8
4-bit ADPCM	17	309	32	7.5kHz	ADPCM4, PCM6, PCM8
4-bit ADPCM, QUICK	14	509	256	11.0kHz	ADPCM4, PCM6, PCM8
PCM	12	186	0	16.0kHz	PCM6, PCM8

Table: The decompressing algorithms and the resources used.

Each of these decompressing algorithms is implemented in an interrupt service routine and shall be invoked each time the sampling rate counter overflows. In these decompressing algorithms, the voice ROM data is accessed, the decompressing action is taken and the decompressed data is then sent to the D/A output.

Since there is no other function called in the decompressing algorithms, the ISR could be invoked once when the sampling rate counter overflows and the stack is not full. In the ISR, the value of the accumulator and the status registers are preserved but the value of the following registers might be changed: TBLP(07H), TBLH(08H), DAL(20H), DAH(21H), ROMC(2CH).

Important Since the registers TBLP and TBLH might be changed in the ISR, it is important to disable the sampling rate interrupt when using table lookup instructions TABRDC or TABRDL.

→ **Decompression algorithms usage**

If it is required to use all of these decompression algorithms, the file "voice.inc" must be included and then the ISR specified for the sampling rate counter interrupt in the program. Any other resources supported are intended to assist with customising the sound track playing. A sample program follows to explain this ...

```
; t.asm                      ; 1
;                            ; 2
#include voice.inc            ; 3 include file  voice.inc
code .section at 0  code     ; 4
   org    0                  ; 5
   jmp    begin              ; 6
```

```
    org    8                ; 7
    jmp    AdpcmISR         ; 8 Specify ISR
begin:                      ; 9
speech 0, 12H, 1            ; 10
set    [22H]                ; 11 Turn on the volume
jmp    $                    ; 12
end                         ; 13
```

The file "voice.inc" is included at line 3 and the ISR is specified at line 8. The symbol AdpcmISR is predefined in the file "voice.inc". The sampling rate counter interrupt is located at interrupt vector 2 or ROM absolute address 8.

Since there is no other function called in the ISR, it is not necessary to be concerned about the status of the stack. If there are other functions that have to be performed when the sampling rate counter overflows, they can be joined together by jumping first to application specific functions and at the end of that function jump to AdpcmISR. For example,

```
; example.asm
;
code .section at 0  code
    org    0
    jmp    begin
    org    8
    jmp    MyFirstIsr
begin:
    ......
MyFirstIsr:
    ......
    jmp    MySecondIsr
    ......
MySecondIsr:
    ......
    jmp    AdpcmISR
    ......
    end
```

Note
- Since all the algorithms are implemented using a relocatable style, applications should be designed by following the same style as mentioned in the HT-IDE2000 User's Guide. This relocatable style programming is based on the section concept. Trying to specify the absolute address without following this style might result in a RAM conflict and cause run-time errors.
- The file "voice.inc" which is generated by the VROM editor is placed in the same directory with the project file. The assember source code and project file should be placed together so that the include file can be included correctly.
- Once the VROM editor has been re-invoked and another compression algorithm chosen for some sound tracks, always remember to rebuild the project again. This is because the include file "voice.inc" may have been changed.

If everything has been prepared as mentioned above, the playing sound can be controlled in the program. Here, one macro and three flags are provided, the *speech* macro, the *standby* flag, the *pause* flag and the *stop* flag. The following gives a more detailed explanations.

- The ***AdpcmQuickMode*** symbol
  The quick mode on for either 4-bit ADPCM or 3-bit ADPCM can be turned on by defining the symbol "AdpcmQuickMode" in your program. This symbol shall be defined before the "voice.inc" is included. Programs running in this mode will consume a little more program ROM space, but the sampling rate could be much higher than in the normal mode. For example:

```
#define  AdpcmQuickMode
#include         voice.inc
......
```

Note The quick mode is only available for ADPCM.

- The ***speech*** macro

  Generally, the *speech* macro is used to play edited sound tracks. It is defined in the included file, "voice.inc". The syntax of the *speech* is ...

  speech *TrackNumber, SamplingRate, VoiceDown*

  where,

  TrackNumber

  is the number of the sound edited, beginning with 0. Refer to the .NUM file for a listing of all voice resources. It is generated by the HTVDS827.EXE.

  SamplingRate

  is used to specify the sampling rate counter.

  VoiceDown

  is used to reduce power consumption.

  Each of these three parameters can be left empty.

Note that **speech** initializes certain variables and turns on the sampling rate counter interrupt. Then, when the sampling rate counter overflows, the decompression algorithm is invoked.

In the *speech* macro the sampling rate interrupt will be enabled, two level stacks will be used, the flag tempo.7 is set (setting this flag is to enable the D/A output, sampling rate counter and counter ROM) and the value of the following registers may be changed: Accumulator (05H), Status (0AH), ROMC (2CH.)

If the *TrackNumber* is left empty, the *speech* will not initialise the track number register which is used by the decompression algorithms.

Because the parameter *SamplingRate* is used to specify the sampling rate control register (located at RAM 23H), if it is left empty, *speech* will not change the value of that register. For example, if there is only one sampling rate in the application, it can be initialized at the beginning of the program and left empty in all of the following *speech* macro. The following paragraph demonstrates this ...

```
code .section at 0 "code"
   org    0
   jmp    begin
   org    8
   jmp    AdpcmISR
begin:
   mov    a, 12H
   mov    [23H], a      ; Sampling rate counter is set to 8MHz
   ......
   speech 0, ,1
```

```
......
speech 1, ,1
......
end
```

The parameter *VoiceDown* could also be left empty. If it is non-empty, no matter what symbol it has, the voice down mode is enabled. Otherwise, if it is left empty, the voice down mode is disabled. Enabling the voice down mode will cause the register DAH and DAL to be set to zero and thus reduce power consumption after the sound track is performed. For example,

```
.....
speech 0,,    ; After track 0 is played, the system will not
......        ;      go into voice down mode
speech 1,,1   ; After track 1 is played, the system goes into
......        ;      voice down mode.
```

- The ***standby*** flag

  The ***standby*** flag is a one-bit, read only flag which indicates the playing status. It will be set if the decompression algorithm is on standby and is waiting for some tracks to be played. For example to demonstrate usage of this flag, if there are five tracks recorded in the voice ROM and all have to be played one by one, the following short program can be used.

```
Z   equ    [0AH].2
#include voice.inc
code .section at 0  code
    org    0
    jmp    begin
    org    8
    jmp    AdpcmISR
begin:
    set    [22H]
    clr    TrackNo
NextTrack:
    Speech , 12H, 1
waiting:
    snz    standby
    jmp    waiting
    inc    TrackNo
    mov    a, TrackNo
    xor    a, 5
    snz    Z
    jmp    NextTrack
    jmp    $
```

Important The *standby* flag is READ ONLY. Changing it may cause unpredictable results.

- The ***pause*** flag

  The *pause* flag is a one-bit flag used to pause the playing. If it is set, the performance is stopped temporarily until it is reset. The following program use port A bit 0 to toggle pause mode when playing.

```
; Example of toggling the pause flag
; Pa.0 is configured as input, pull-high and a switch button is
; connected
; between pa.0 and VSS.
  pa    equ     [12h]
  acc   equ     [05h]
#include voice.inc
code .section at 0  code
    org     0
    jmp     begin
    org     8
    jmp     AdpcmISR
begin:
    set     [22H]
again:
    speech  0, 12H, 1
waiting:
    sz      standby
    jmp     again
    sz      pa.0
    jmp     waiting
    clr     acc
Loop:
    sdz     acc
    jmp     Loop
    sz      pa.0
    jmp     waiting
    sz      pause           ; toggle pause
    jmp     ps1
    set     pause
    jmp     Pa0Release
ps1:
    clr     pause
Pa0Release:                 ; wait until pa.0 release
    snz     pa.0
    jmp     Pa0Release
    jmp     waiting
    end
```

- The ***stop*** flag

  The *stop* flag is a one-bit flag used to terminate the playing. If it is set, the performance is stopped. Playing has to be restarted if this flag is used to terminate playing. In the following program, the playing is terminated if port A bit 0 is set low for a short period of time.

```
; Example of stop playing by setting the stop flag
; In this example, bit 0 of port A is configured as input,pull-
; high and a switch button is connected between pa.0 and VSS.
pa  equ    [12h]
acc equ    [05h]
#include voice.inc
code .section at 0 'code'
    org    0
    jmp    begin
    org    8
    jmp    AdpcmISR
begin:
    set    [22H]
again:
    speech 0, 12H, 1
waiting:
    sz     standby
    jmp    again
    sz     pa.0
    jmp    waiting
    clr    acc
Loop:
    sdz    acc
    jmp    Loop
    sz     pa.0
    jmp    waiting
    set    stop          ; stop
    jmp    $
    end
```

Quick reference

- AdpcmQuickMode

  Used to turn on quick mode for either 3-bit ADPCM or 4-bit ADPCM. Define before voice.inc is included.

  Usage,

  #define AdpcmQuickMode

  For example,

```
#define      AdpcmQuickMode
#include     voice.inc
......
```

- Pause

  A one-bit flag used to pause playing.

  Usage,

  ```
  clr    pause
  ```

  or

  ```
  set    pause
  ```

  or

  ```
  sz     pause
  ```

  or

  ```
  snz    pause
  ```

  For example, to pause play until bit 0 of port A is turned low ...

  ```
      set    pause
      sz     pa.0
      jmp    waiting
      clr    pause
      ......
  ```

- Speech

  A macro used to play the sound tracks.

  Usage,

  ```
  speech    TrackNo, SamplingRate, VoiceDown
  ```

  For example, to play the first track, 8kHz (assuming the system frequency is 4MHz), then

  ```
  speech    0, 12H, 1
  ```

- Standby

  A one-bit, read only flag indicates the status of playing. If it is set, a sound track has been playing.

  Usage,

  ```
  sz     standby
  ```

  or

  ```
  snz    standby
  ```

  For example, to set port A high when the microcontroller is on standby ...

  ```
  waiting:
      snz    standby
      jmp    waiting
      set    pa
      ......
  ```

- Stop
  A one bit flag used to stop playing.
  Usage,

```
clr    stop
```

  or

```
set    stop
```

  or

```
sz     stop
```

  or

```
snz    stop
```

For example, to stop playing when bit 0 of port A is low ...

```
Waiting:
    sz      pa.0
    jmp     waiting
    set     stop
    ......
```

Using the VROM Editor

This chapter outlines the file types that the HT-VDS827 creates, then briefly describes the menu commands and how to construct a .VPJ file.

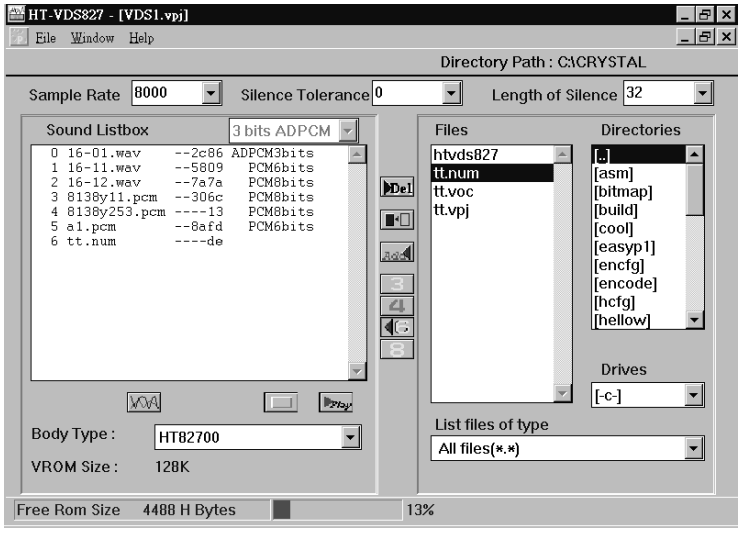


Fig 13-1: Main screen of HT-VDS827

File type

The Holtek HT-VDS827 creates three kinds of files.

- VPJ project file, which contains the voice source file and other information.
- VOC file is for downloading, which must be returned to Holtek.
- NUM text file, which contains the starting address of the voice file and the compression method used. This file can be used for error checking.

Creating a new .VPJ file

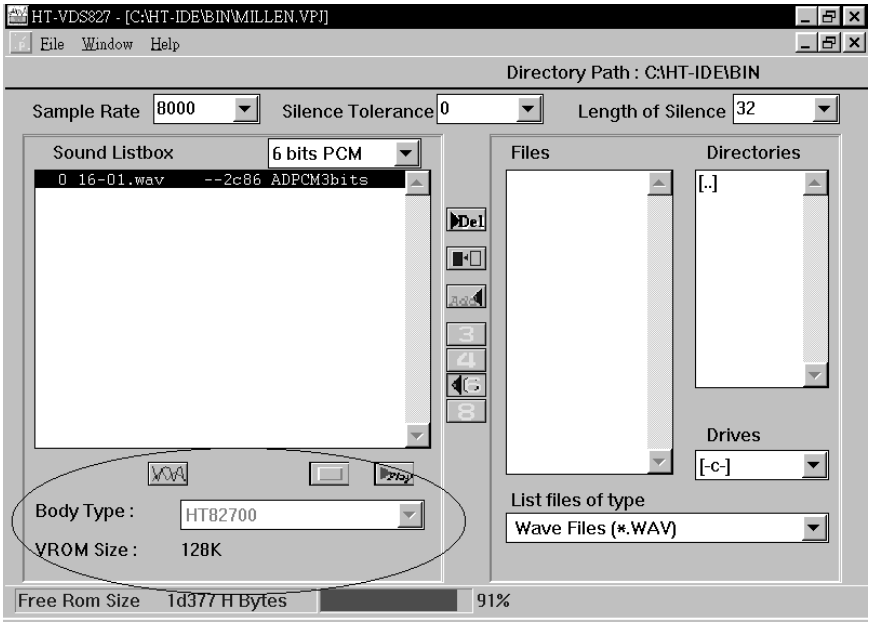
→ **Step 1**

In the main screen of the HT-VDS827, choose the New command from the File menu to create a new project.

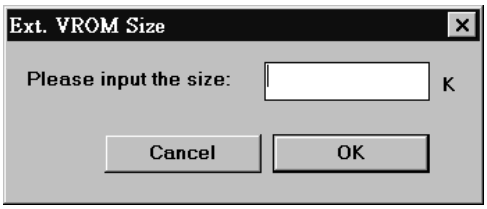
→ **Step 2**

From the Body Type Selection Box, select a body to meet the desired requirement. Each body has its corresponding voice ROM space as shown below:

Body	Voice Rom Space (Byte)
HT827A0	128K
HT827D0	64K

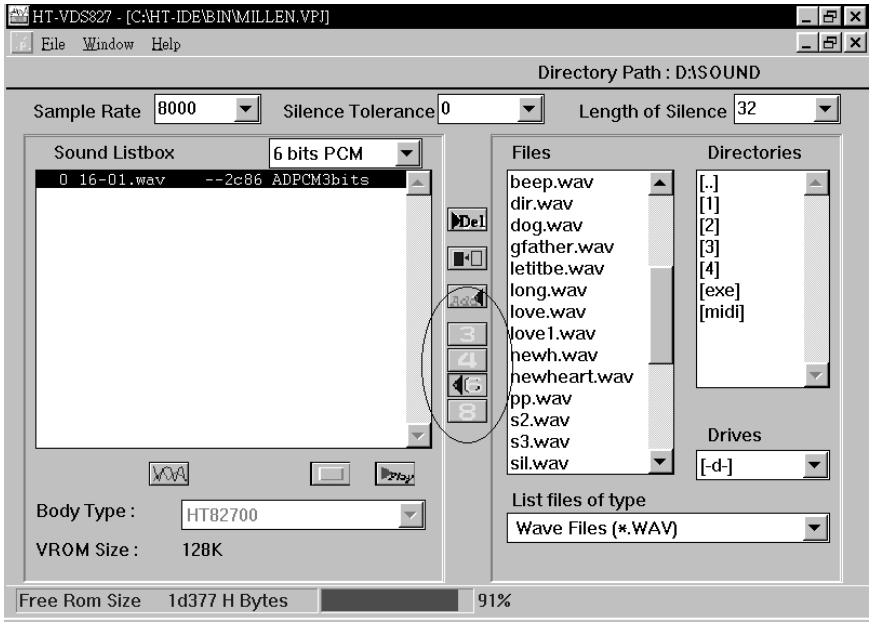


If none of the body types meet the requirements, select Ext. VROM command from the Body Type Selection Box, and fill the desired VROM size in the dialog box as shown below.



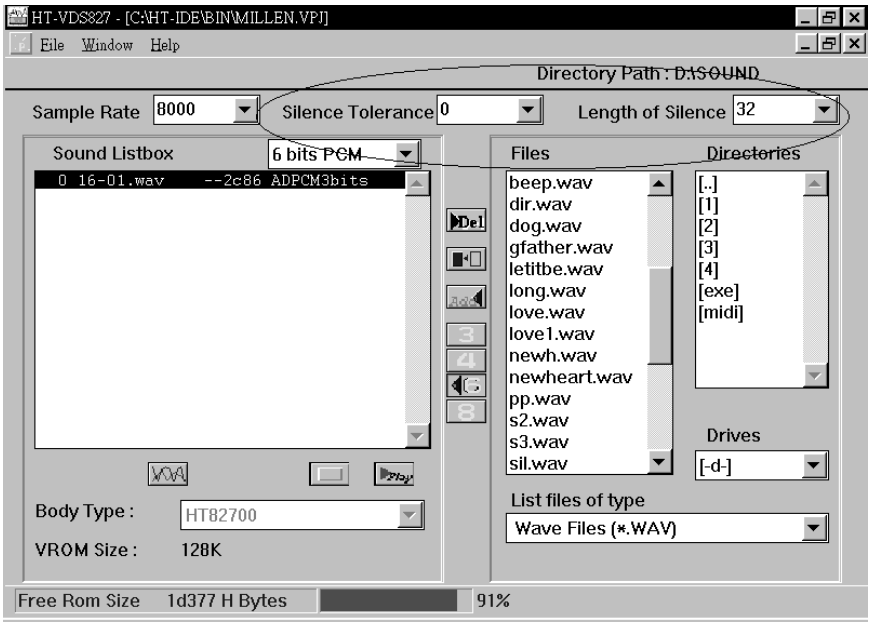
→ **Step 3**

Select the appropriate model to compress the source voice file. The voice quality must be sacrificed in order to save the voice ROM space. There is a trade-off between compression model and sound quality. Four compression models are available:
3-bit ADPCM
4-bit ADPCM
6-bit PCM
8-bit PCM



→ **Step 4**

Different settings of silence tolerance and length of silence lead to different compression results. Changing the length of silence and the silence tolerance changes the compressed file length and the free ROM size, because silence has a concise compression format. Four lengths of silence and thirteen silence tolerances are available.



→ **Step 5**

Select the working drive and directory from the right sub-window of the main window, then choose the file type of the source voice code. Three kinds of Voice files are supported:

wave file (*.WAV)
16-bit PCM files (*.PCM)
8-bit PCM files (*.PCM)

If All files (*.*) is chosen, the file will be treated as a binary file without compression.

→ **Step 6**

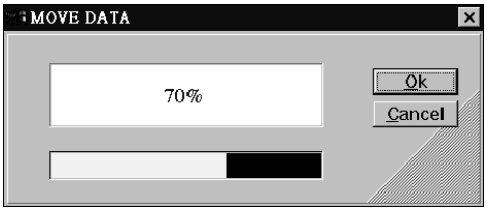
To compress the files in the files list box, double click the file, or choose the file first and then press the button . The mouse can also be dragged or press the button to select all files and then press the button to compress all files together. The compressed files are listed in the Sound List Box.

The sound list box displays the file name, compressed file size, and the compression model used in their processing order. The files with All files (*.*)/List files of type item selected, will not be compressed by any kind of compression model.

→ **Step 7**

The order of the compressed files can be changed by dragging the file with the right mouse key, and dropping it at the desired place. When dropping the file, a Move Data window appears:

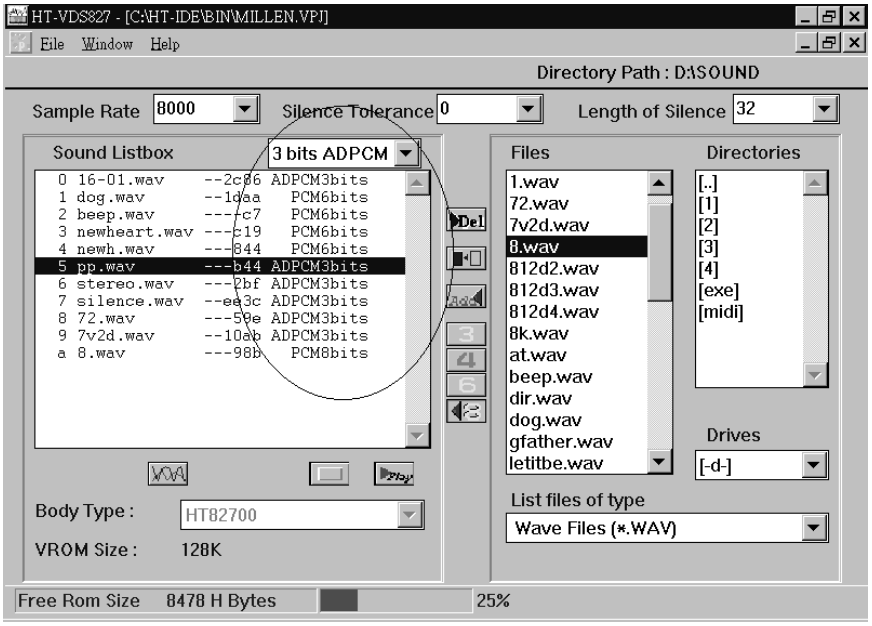
→



Step 8

The compression model of a compressed file can easily be changed by selecting the file, and choosing the desired compression model from the combo box.

Note 4-bit ADPCM and 3-bit ADPCM cannot exist at the same time.



→ **Step 9**

A voice or a data file may be edited by selecting the desired file first, and then pressing the button . The HT-VDS827 will auto-detect the file type and open the files with either the HT-Voice Editor or the HT-Binary Editor. After saving the changes and closing the editor, the HT-VDS827 will reload the data automatically.

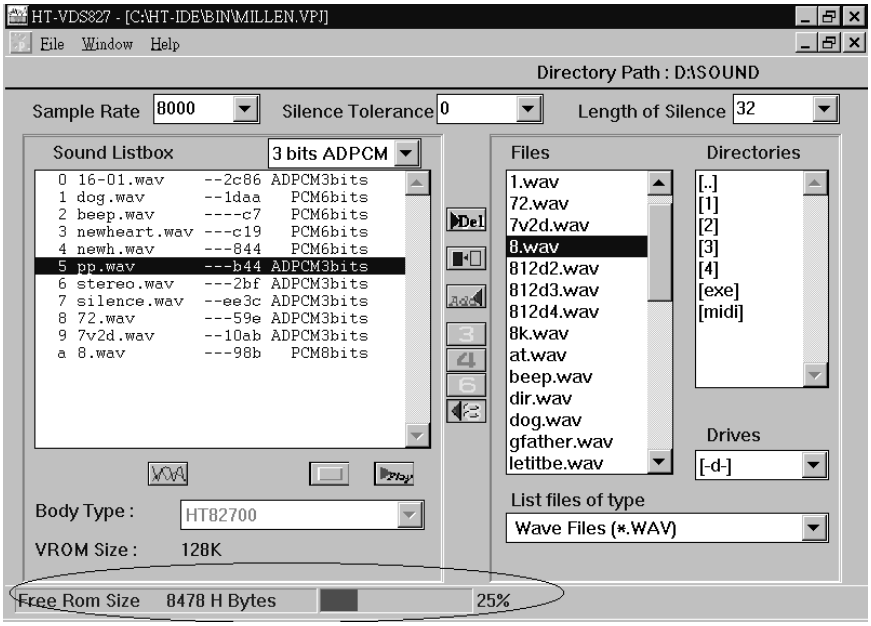
For more information about the HT-Voice Editor and the HT-Binary Editor, refer to the following sections.

→ **Step 10**

To save the current edited project, select the Save As command from the File menu. The Save As command creates three files, .VPJ, .VOC, and .NUM. Selecting the Print... command prints a table which shows the compressed file name, the starting address of the file in the voice ROM and the compression model used.

No.	File Name	Starting Address	Compression
0	7v2d.wav	00000015	3ADPCM
1	8k.wav	000010AE	3ADPCM
2	beep.wav	00001E83	3ADPCM
3	holtek0.wav	00001EEA	3ADPCM

The bottom box of the main screen displays the percentage of free ROM size in real time.



Play with sample rate

HT-VDS827 compression tool also provides an on-line play function if a sound card is installed. To listen to the sound of the compressed voice file, select a compressed file in the sound list box and press the button  . Press the button  to stop. The sampling rate of the playing sound can be adjusted by selecting the appropriate sampling rate. If it is required to listen to the original sound of the source file, double click the source file in the Files list box with the right mouse. This function allows the sound before and after compression to be compared and the best choice made.

File menu

The File pop-up menu consists of New, Open, Save as... and Print... commands.

→ **New**

Create a new project.

→ **Open**

Open an existing project.

VOC file must exist in the same directory with .VPJ file. If not, the HT-VDS827 will create a new one.

→ **Save As...**

Save the current edited project under a new file name.

→ **Print...**

Print the result including the file name, the starting address, and the compression methods used.

Window menu

The Window menu consists of Tile, Cascade, Arrange Icons, and Close All commands.

→ **Tile**
Tile all opened files on the screen.

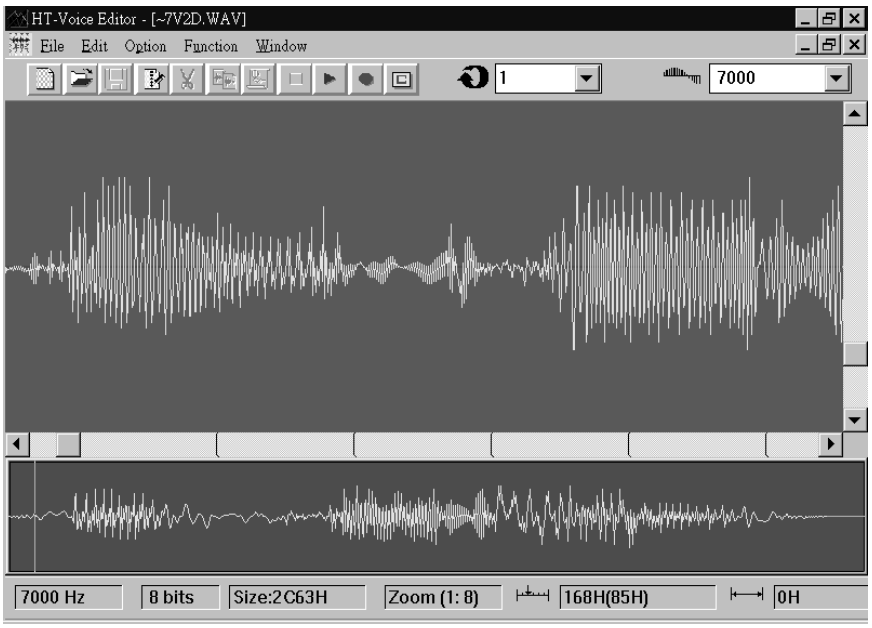
→ **Cascade**
Cascade all opened files on the screen.

→ **Close All**
Close all the opened project files.

→ **Arrange Icons**
Arrange all icons.

Using the HT-Voice Editor

The main screen of HT-Voice Editor is shown below. This chapter explains how to edit a wave file, and describes all the files and the menu commands.



New/Record command

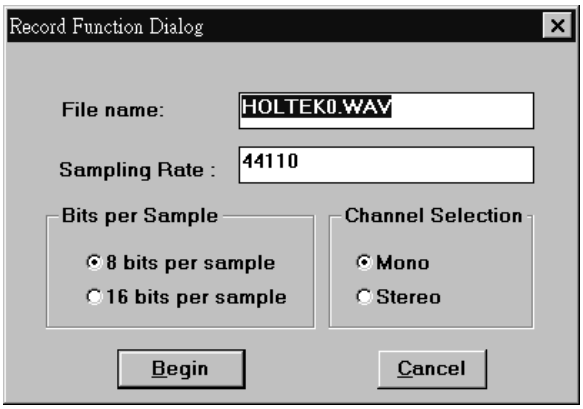
→ **Step 1**

To create a new voice file, select the New command from the File menu.

→ **Step 2**

To record, select the Record command from the Function menu, or press

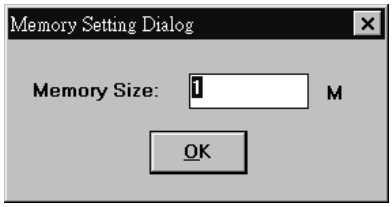
button (if a sound card and microphone are installed), a Record Function Dialog screen appears:



Fill out the dialog box: File Name, Sample Rate, and other options. Press the Begin button to start recording.

→ **Step 3**

The default longest record time is 25 sec (if the option is 8 bits per sample and Mono). If more time is needed, select the Memory.. command in the Option menu . The Memory Setting Dialog box is shown as follows. Press the End button to stop recording.



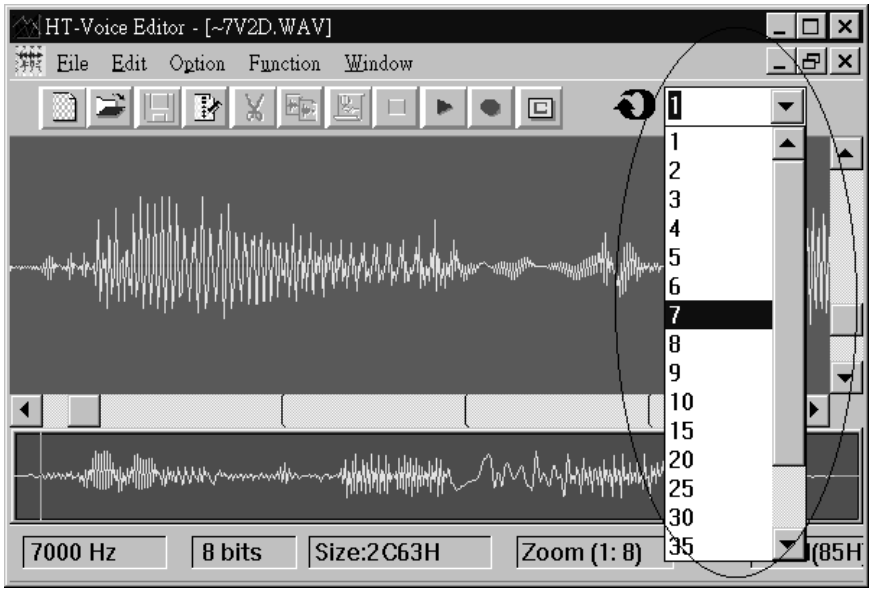
Play command sample rate

→ **Step 1**

Before playing, it is necessary to record or open a file. Select the Play command in the Function menu, or press the ▶ button to listen to the recorded sound. Press the ■ button to stop.

→ **Step 2**

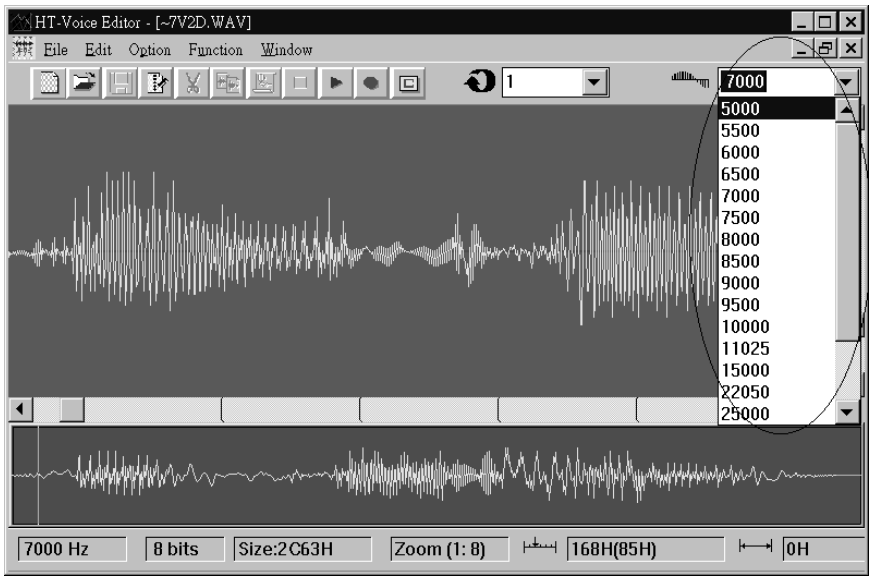
To play a song repeatedly, select the number of times to play from the following combo box.



→ **Step 3**

The Sample Rate can be adjusted to differentiate between the sounds produced.

You can also adjust the Sampling Rate, and then you can differentiate between the sounds produced. This command will not change the voice data, but the sampling rate when playing. Different sampling rate will come with difference sound in frequency and speed.



Open command

→ **Step 1**

To open a voice file, select the Open command in the File menu.

→ **Step 2 - Cut/Copy/Paste**

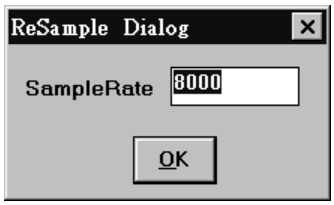
Select a range first, and then cut/copy the range by selecting the cut/copy command from the Edit menu, or press the / button. After cutting or copying, the Paste command in Edit menu or the button will be enabled. This command can be selected or button pressed to paste the range in the clipboard to the current position.

→ **Step 3 - Delete**

Mark a range first, and then select the Delete command in the Edit menu.

→ **Step 4 - Re-sample**

Sample rate of the file can be changed by selecting the [ReSample] command in the [Edit] menu. The [ReSample Dialog] is shown below:



ReSample will add/delete data points to suit the sampling rate we type in. If you decrease the sampling rate, the timbre will not as good as original voice, but the file size will be decreased.

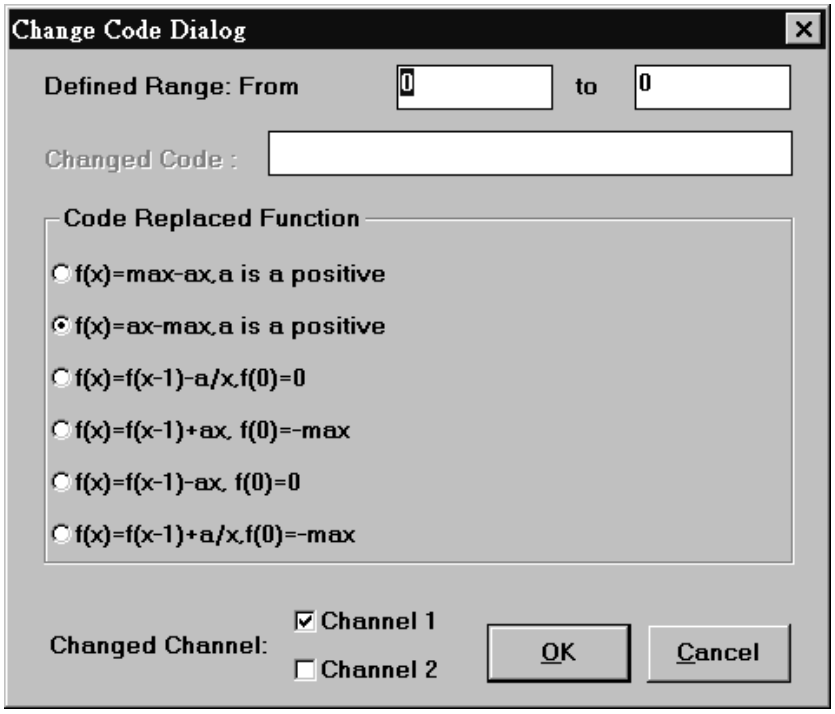
→ **Step 5 - Change Format**

The voice file format can be changed by selecting the [ChangeFormat] comman in [Edit] menu, or pressing the button. You can change the Sampling rate, Channel, and Sampling rate, Channel, and Bits per sample of this file. If a PCM file is opened, the following default parameters are taken: Sampling rate 8000Hz, 1 default setting, Sampling rate 8000Hz, 1 Channel, 8 Bits Per Sample. So we can set the correct setting by using [ChangeFormat] command.

→ **Step 6 - Change Code**

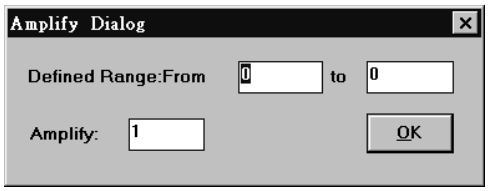
Before changing the code, a range must first be marked, or defined in the Change Code Dialog box as shown below. The Code can be changed by selecting the Change Code command in the Edit menu.

To replace the data marked with the code series, fill in the Changed Code edit box with code series, with the format data1,data2,data3 (hex). The data marked with the code function can also be replaced. Also, select which channel to be replaced by checking the Changed Channel check box.



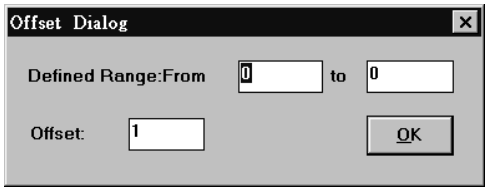
→ **Step 7 - Amplify**

Mark a range first or choose the Amplify command from the Edit menu and define the range in the Amplify Dialog window, and then type in the magnitude desired for the selected range to be amplified.



→ **Step 8 - Offset**

Mark a range first, or select the [Offset] command from the [Edit] menu and define a range in the [Offset Dialog] window, and type in the desired offset value. If the offset value is negtive, voice data will down offset, that means, if the offset value is positive, voice data will up offset.



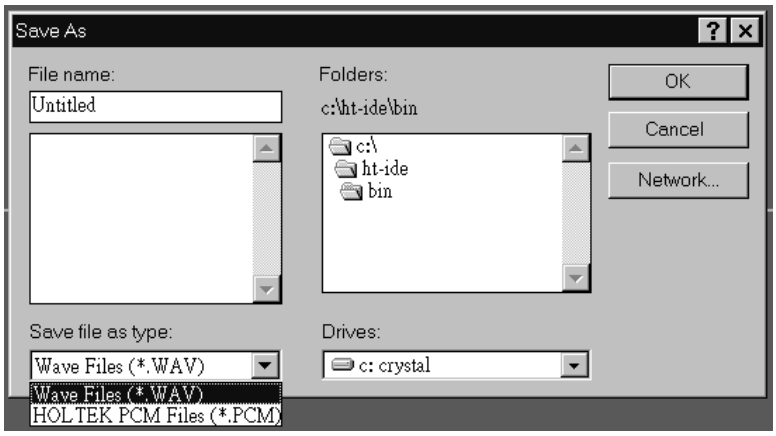
Save command Voice type

→ **Save**

To save a file, select the Save command in the File menu, or press the button. If the file name is not specified, the system will show a Save As dialogue box.

→ **Save As...**

To save the current file under another name, select the Save As command from the File menu. The System will show a Save As dialogue box as follows.



Type in the full path, file name, and file type: Wave Files .WAV or PCM File .PCM.

Other commands

→ **Short Menu or Full Menu/File**
Short menu / Full menu switch command.

→ **Exit/File**
Close the application.

→ **About HT-Voice.../File**
Information about this application.

→ **Tile or Arrange/Window**
Tile (it is called Arrange in short menu) all opened files on the screen.

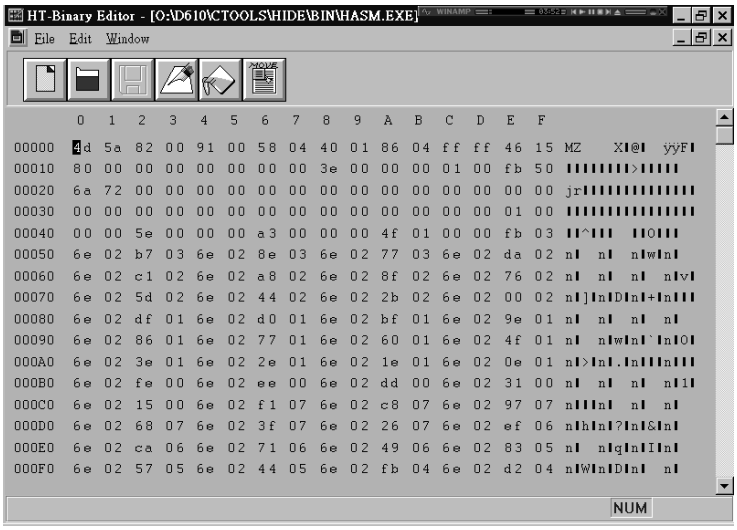
→ **Cascade/Window**
Cascade all opened files on the screen.

→ **Arrange Icons/Window**
Arrange all icons on the screen.

→ **Close All/Window**
Close all opened files.

Using the HT-Binary Editor

The main screen of the HT-Binary Editor is shown below. This is a simple binary editor used to communicate with the HT-VDS827. This chapter explains how to edit a binary file and describes the menu commands.



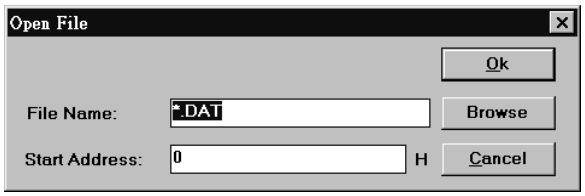
Creating a new file

To create a new file, select the New command from the File menu, or press the button. Then begin to input data (Hex) in the editing area.

Opening a file

To open a file, select the Open command from the File menu, or press the button. The following Open File dialogue box appears:

Type in the full path, or press the Browse button to select a file, and input the starting address from where the file is loaded in the Start Address edit box.

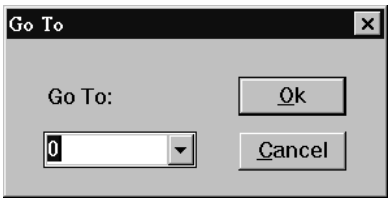


Editing

→ Go To

To go to a specific position, select the Go To command from the Edit menu, or press the button. A Go To dialogue box appears:

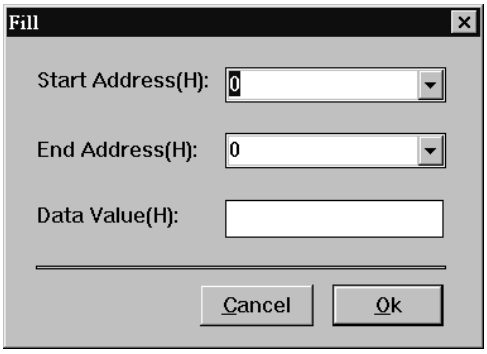
Type in the position you want to go to (in Hex), and press the OK button.



→ **Fill**

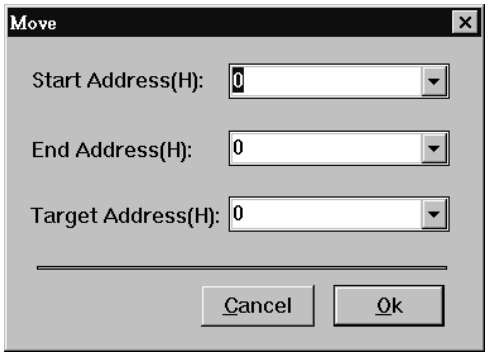
To fill a range with data (in Hex), select the Fill command in the Edit menu, or press the button. A Fill dialogue box appears:

Type in the Start Address, End Address and data in Hex, and then press the Ok button.



→ **Move**

To copy data from a specified range to a target address, select the Move command in the Edit menu, or press the button. A Move dialogue box appears:

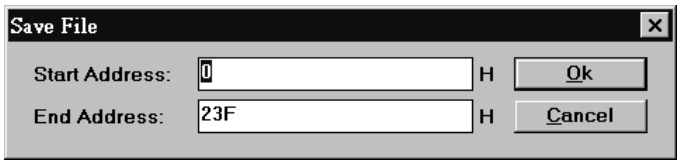


Type in the Start Address, End Address and Target Address, and then press the Ok button.

Saving

→ **Save**

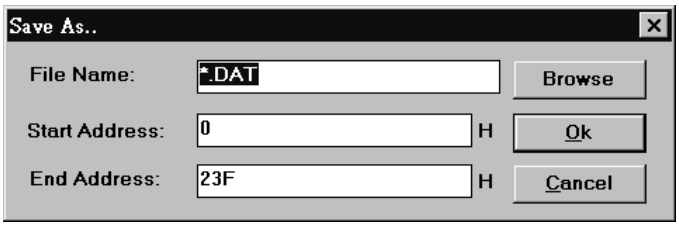
To save a file, select the Save command from the File menu, or press the button. A Save File dialogue box appears. If the file name is not specified, a Save As.. dialogue box will appear.



Type in the Start Address and End Address to be saved and then press the Ok button.

→ **Save As ...**

To save the current file under another name, select the Save As command in the File menu. A Save As ... dialogue box appears:



Type in the full path in the File Name edit box, or press the Browse button to select a file. Input the Start Address and End Address to be saved, and then press the Ok button.

Other commands

→ **Short Menu or Full Menu/File**
Short menu / Full menu switch command.

→ **Exit/File**
Close the system.

→ **About HT-Binary Editor*/File**
Show information about the system.

→ **Tile or Arrange/Window**

Tile (called Arrange in short menu) all opened files on the screen.

→ **Cascade/Window**
Cascade all opened files on the screen.

→ **Arrange Icons/Window**
Arrange all icons on the screen.

→ **Close All/Window**
Close all opened files on the screen.

Chapter 14

LCD Simulator

14

Introduction

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. It facilitates the development process even if the required LCD panel is unavailable. Note that if the micro controller of the current project does not support LCD functions, these commands are disabled.

LCD Panel File

Before starting the LCD simulation, an LCD panel file must first be set up, otherwise the HT-LCDS cannot simulate the LCD action. If the micro controller of the current project has an LCD driver, then a corresponding panel file should be setup for simulation. The LCD simulator commands within the Tools menu will then be enabled to setup the panel file and for simulation. (Fig 14-1).

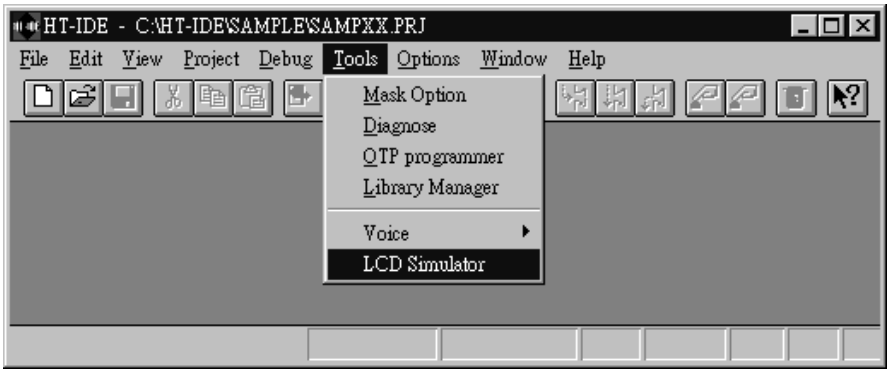


Fig 14-1

Relationship between the panel file and the current project

By default, the panel 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 file of the current project. The panel 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 file by clicking File menu, Save command or Save button on the toolbar.

When the HT-LCDS begins simulation, it refers to the current active panel file for the simulation information. The File menu, New command or Open command is used to activate a panel file whether for the same project name or not.

If the HT-LCDS was in simulation mode while exiting the previous time from the HT-IDE2000, the HT-LCDS will be automatically invoked in simulation mode the next time the HT-IDE2000 is used. In this situation, the HT-LCDS refers to the panel file with the same name as the project name.

Entry situations of the HT-LCDS

When selected from within the Tools menu, the LCD Simulator, Fig 14-2 is displayed if the corresponding panel 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 file does not exist in the project directory, both the panel screen and the COM/SEG table are not displayed.

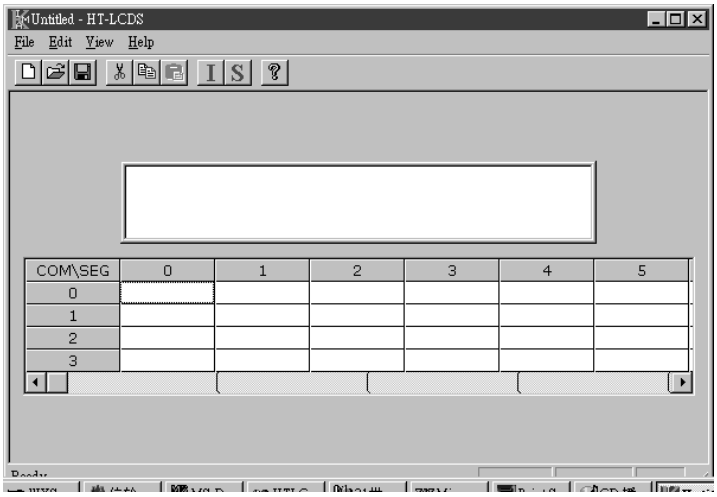


Fig 14-2

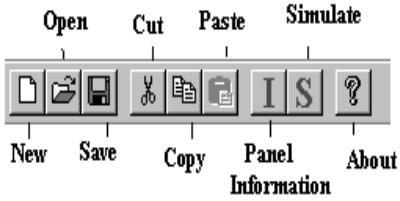


Fig 14-3

Set up the LCD Panel File

The following two steps are used to setup a panel file:

- Set the panel configurations. This data includes the segment and common number of the LCD driver as well as the width and height size of the panel in pixels. This is displayed on the screen as well as the directory of the panel file and the dot matrix mode selection.
- Select the patterns and their positions. This will setup the relationship between the patterns and the COM/SEG positions.

Set the panel configurations

The only way to set the panel configuration is to create a new panel file by selecting the File menu, New command. The Panel Configuration dialog box, in Fig 14-4, will be displayed along with the corresponding panel file name. Set the correct data for each item in the box and press the [OK] button. The screen returns to Fig 14-2 for pattern selection.

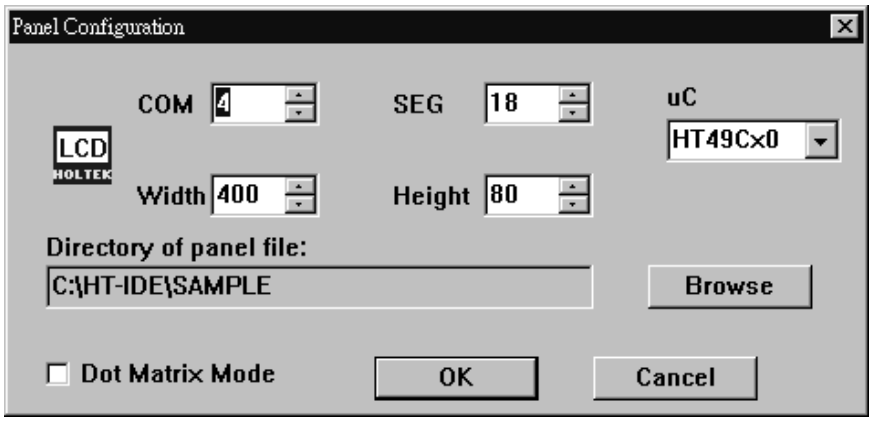


Fig 14-4

The panel configuration items are:

- μC controller. Selects the micro controller of the current project.
- COM and SEG. Select the number of the COMMON and SEGMENT of the LCD driver respectively. The default number of the LCD driver for this micro controller is displayed. Ensure that these numbers are the same as the actual setting number of the LCD driver for the micro controller.
- Width and Height. Enter the size of panel screen in pixels. They can be changed in order to adjust the panel screen.
- Directory of panel file. Select the directory where the panel file is stored. Use the the browse button to change the directory or set the same directory as the project s.
- Dot Matrix Mode. To simulate the dot matrix type of LCD panel or not. The dot matrix screen is similar to Fig 14-5.

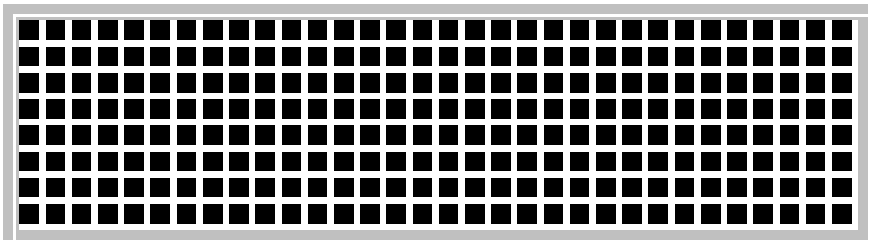


Fig 14-5

Note Do not set different COM or SEG number from the actual corresponding numbers which are set for the microcontroller LCD, otherwise the results will be unpredictable.

Select the patterns and their positions

- When Fig 14-2 appears during the panel file creation procedure, i.e. after completing the panel configuration, patterns must be selected by using the Pattern Information dialog box and selecting the COM/SEG positions for these patterns.
- If Fig 14-2 has appeared from the Open command of the File menu, then new patterns may be added, original patterns changed to new patterns or patterns deleted from the COM/SEG table. In addition, the COM/SEG positions of the patterns can be changed.

The following are the methods used to add/delete/modify the patterns and their positions:

Add a new pattern

- Select the COM/SEG position on the grid as in Fig 14-2 and double click the mouse. The Pattern Information dialog box, in Fig 14-6, is displayed. All the bitmap files in the project s directory are listed in the Pattern List box. The Size field is the bitmap size of 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, i.e. a bitmap file, from the pattern list box, or click the Browse button to change to another directory and select a pattern from it. The HT-LCDS uses 2-color bitmap files as the image source of patterns. The Preview-window zooms in the selected pattern.
- Set the X/Y positions of the selected pattern in the panel screen.
- Press the [OK] button which returns the screen to Fig 14-2, then click the Save command of the File menu or click the Save button on the toolbar. The panel file has now been created or modified.

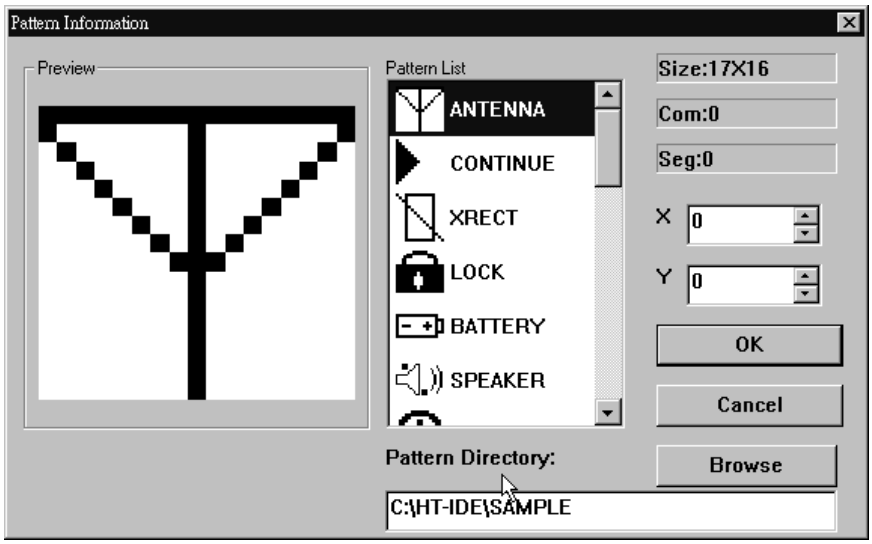


Fig 14-6

Delete a pattern

- In Fig 14-2, Select the COM/SEG position of the pattern to be deleted and press the the [Delete] key or click the Cut button on the toolbar.

Change the pattern

- Use the Delete a pattern method to delete the selected pattern, then use the Add a new pattern method to change the pattern. Or ..
- In Fig 14-2, select the COM/SEG position of the selected pattern and double click the mouse. The Pattern Information dialog box, in Fig 14-6, is then displayed. Select a pattern from the Pattern List box and press the [OK] button.

Change the pattern position

- In Fig 14-2, use the Select-Drag-Drop method to move the pattern directly onto the panel screen. Or ...
- 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 above operations have been completed and the screen returns to Fig 14-2, click the Save command of the File menu or click the Save button on the toolbar. The panel file has now been created or modified.

Simulate the LCD

Before starting the LCD simulation, ensure that the correrct panel file is referred to by the HT-LCDS. It is dependent upon the following two situations:

Still in LCD simulation mode when exiting from HT-IDE2000

If the HT-LCDS was in simulation mode the last time the HT-IDE2000 was exited, it will return to this mode the next time HT-IDE2000 is run. The HT-LCDS will reference the panel file with the same name as the project name. The LCD simulation can be terminated when starting the HT-IDE2000.

In HT-LCDS environment

Click the LCD Simulator command of the Tools menu, in Fig 14-1, to enter the HT-LCDS environment. Fig 14-2 is displayed.

- Click the S button on the toolbar. The HT-LCDS begins LCD simulation and the corresponding panel file is referenced. This will be the panel file with the same name as the project s.
- Open a panel file which may not be the corresponding panel file of the current project. Click the S button on the toolbar. The HT-LCDS will begin LCD simulation and the corresponding panel file referemced.

When the HT-LCDS begins simulation, Fig 14-7 is displayed and the most recent LCD datum displayed on the panel screen.

Stop the simulating

Click the X box on Fig 14-7, to stop the HT-LCDS simulation and exit.

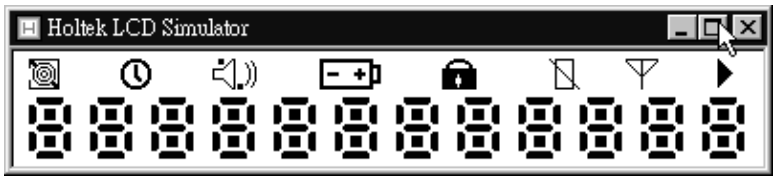


Fig 14-7

Chapter 15

Virtual Peripheral Manager 15

Introduction

In most practical applications the chosen microcontroller is connected to some form 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 microcontroller 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 microcontroller 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 a more efficient debug and implementation of user applications.

The VPM Window

Fig 15-1 shows a practical example of a VPM window. As in most window 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 microcontroller. 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 shows the total execution time and cycle count respectively while the VPM is in running mode

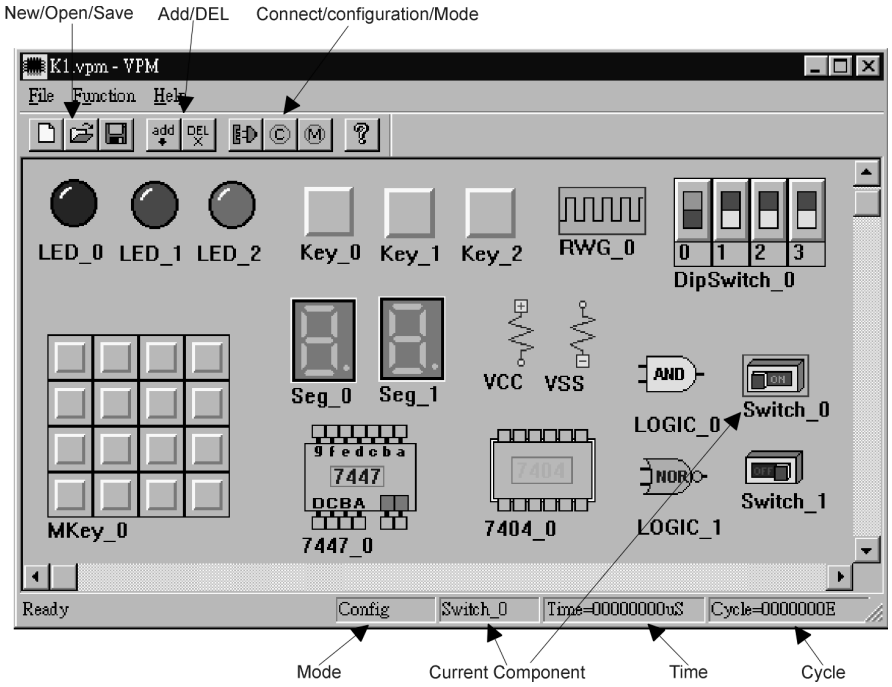


Fig 15-1

VPM Menu

File menu

There are five 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.

→ **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

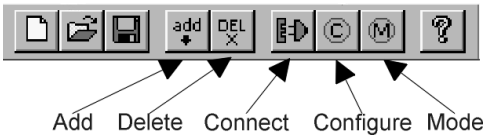


Fig 15-5

→ **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.

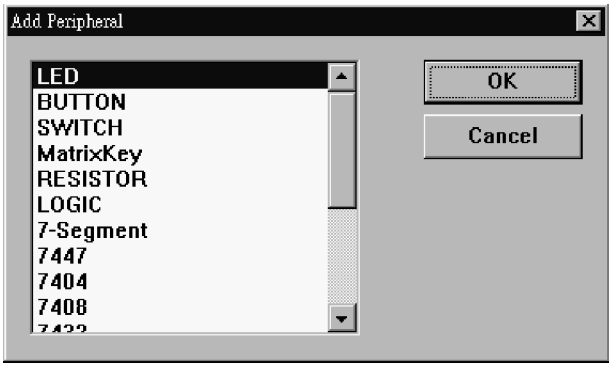


Fig 15-6

→ **Delete**

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.

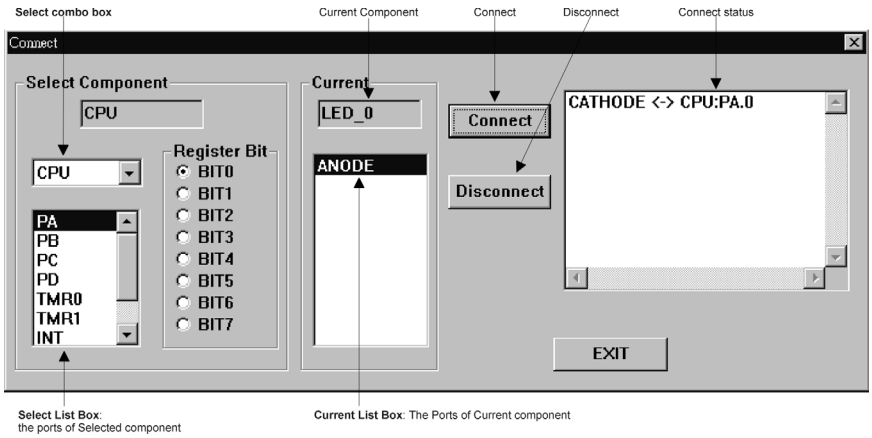


Fig 15-7

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.

→ **Configure**

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.

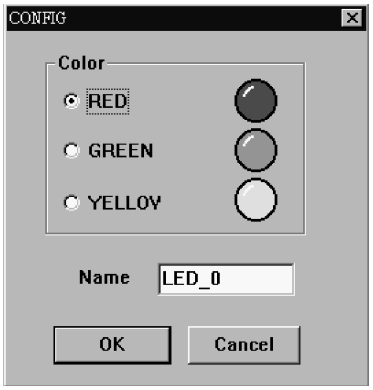


Fig 15-8

→ **Mode**

The VPM 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/delete/configure functions. In the running mode, the VPM will display the operations of these components according to their specific configurations in addition to displaying the HT-IDE2000 microcontroller 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.

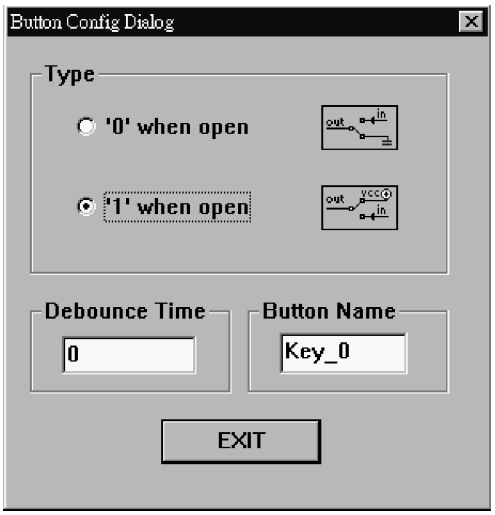


Fig 15-11

Seven segment display

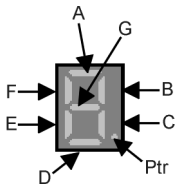


Fig 15-12

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 are connected to an input pin of the same name and also to a common pin. This common pin can be either a cathode (-) or 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.

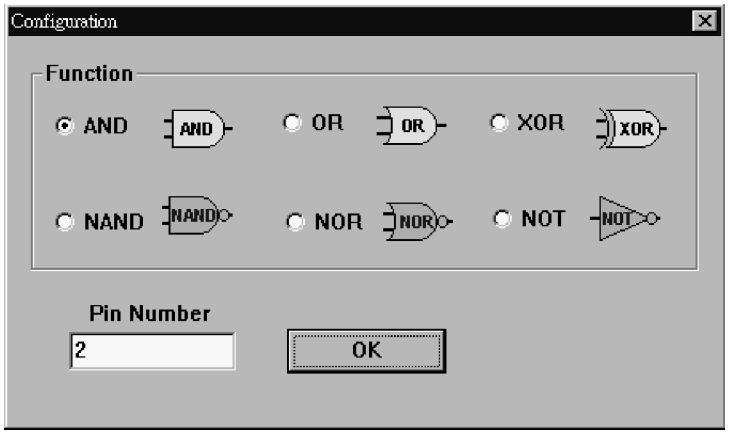


Fig 15-14

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.

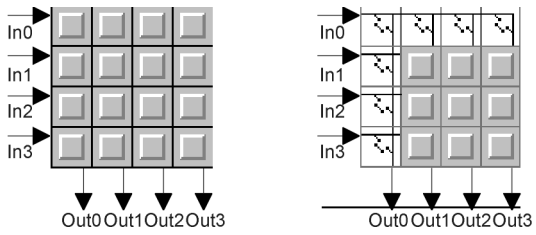


Fig 15-15

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 CPU 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 clicked twice 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 microcontroller 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 HT-IDE2000 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-IDE2000 system**

- Create a new project and select the HT48C10 body (Project/New)
- Add the source file scanning.asm to the project (Project/Edit) The file can be found in the HT-IDE2000\SAMPLE\CHAP15 directory
- Change the HT-IDE2000 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 Select the 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 CPU (n=0-7)
- The following shows how to connect LED_0's anode to VCC and its cathode to bit0 of PA on the CPU
- 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 CPU
- 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-IDE2000**

Start the debug operations — the output results for the LEDs will be shown in theVPM window.

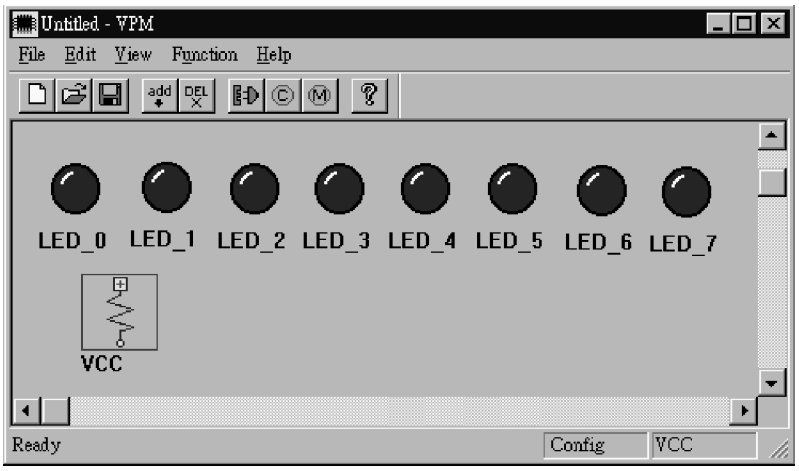


Fig 15-17

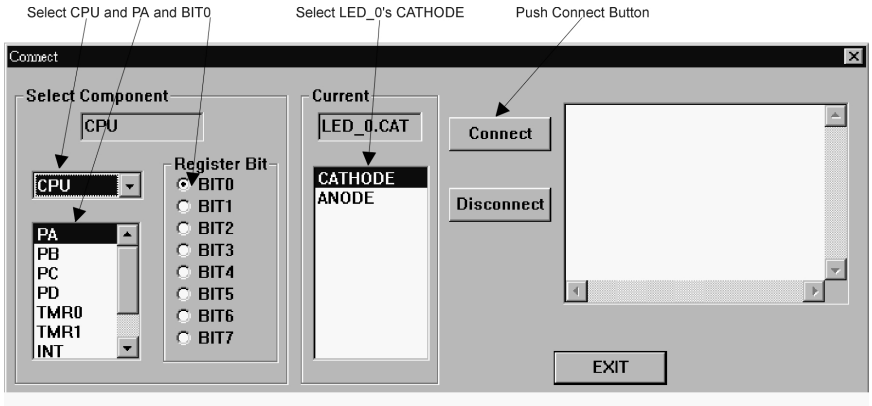


Fig 15-18

Part IV

Programs and Application Circuits

To assist in understanding the general concepts of microcontroller circuit and program design some examples are provided here for consultation. Working carefully through the examples, simultaneously looking at both the code and the explanation should give a good introduction to some useful microcontroller programming techniques. Although specific microcontrollers are chosen for the application in question, the same general programming techniques apply to other controllers.

Chapter 16

Input/Output Applications 16

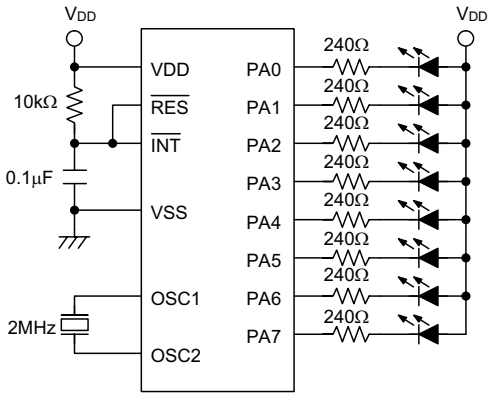
Using the HT48CX0 series of microcontrollers for I/O applications is very simple using the port I/O registers to control the input/output data. The possibility of single bit manipulation further enhances control of input and output data giving increased flexibility. Setting the port bits to either input or output is conducted under program control and normally implemented within the program initialization section. Setting the port bits can be changed again within the main program section, however this is not recommended unless the circuit design calls for special double function I/O port operation.

Scanning Light

This example gives a functional emulation of a scanning LED array. Here a row of LEDs will light in turn one after the other. The circuit uses the PA port PA0~PA7, each bit of which is connected via a 240Ω series resistor to an LED.

Circuit design

The I/O port bits PA0~PA7 are the outputs, with each output bit controlling a single LED via a 240Ω series resistor. By using the instructions RRC and RLC the illuminated LED can be made to move from left to right and vice versa. See the circuit diagram for more details.



HT48C10

Program

```
#include ht48c10.inc
; - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
data .section  data          ;== data section ==
count1 db  ?                 ;delay loop counter 1
count2 db  ?                 ;delay loop counter 2
lamp   db  ?                 ;lamp register
; - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
code .section at 0  code     ; == program section ==
    org    00H               ;
    jmp    start             ;
    org    04h               ;external interrupt subroutine
    reti                     ;for guarantee
    org    08h               ;timer/event 0 interrupt subroutine
    reti                     ;for guarantee
    org    0ch               ;timer/event 1 interrupt subroutine
    reti                     ;for guarantee
start:                       ;
    clr    intc              ;initialize registers
    clr    tmrc              ;to guarantee performance
    clr    tmr               ;(interrupts)
    set    pac               ;(ports)
    set    pbc               ;(input mode)
    set    pcc               ;
main:
    mov    a,0       ;(1)    ;
    mov    pac,a             ;set port A to output port
    mov    pa,a              ;zero port A (all lamp on)
    mov    a,0feh    ;(2)    ;1111 1110 (1:OFF ,0:ON)
    mov    lamp,a            ;set initial lamp state
llamp:                       ;shift lamp left loop
    mov    a,lamp            ;load lamp state
    mov    pa,a              ;output lamp state to port A
```

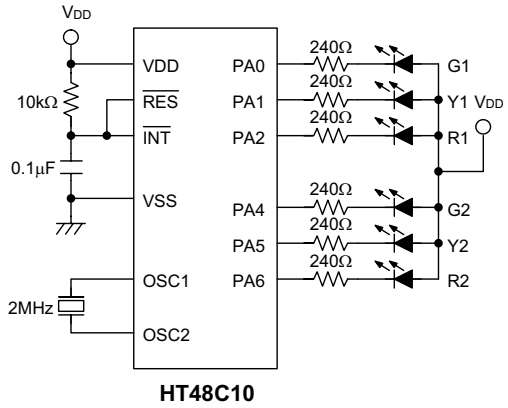
```
    call   delay      ;(3)   ;delay for a while
    set    c                 ;;shift lamp state left through
    rlc    lamp       ;(4)   ;;carry flag (fill LSB 1)
    sz     c                 ;if all LEDs have been lit?
    jmp    llamp      ;(5)   ;;no. continue shift left loop
    rrc    lamp       ;(6)   ;;yes. restore lamp state
rlamp:                       ;shift lamp right loop
    mov    a,lamp            ;load lamp state
    mov    pa,a              ;output lamp state to port A
    call   delay             ;delay for a while
    set    c                 ;;shift lamp state right through
    rrc    lamp       ;(7)   ;;carry flag (fill MSB 1)
    sz     c                 ;if all LEDs have been lit?
    jmp    rlamp             ;;no. continue shift right loop
    rlc    lamp              ;;yes. restore lamp state
    jmp    llamp      ;(8)   ;repeat from shift lamp left loop
delay  proc                  ;delay subroutine
    mov    a,2fh             ;; set counters
    mov    count1,a          ;;
    mov    count2,a          ;;
d1:
    sdz    count2            ;count down count2
    jmp    d1                ;
    sdz    count1            ;count down count1
    jmp    d1                ;
    ret
delay endp
    end
```

Program description

This short program allows an array of 8 LEDs to be illuminated in turn first in a left moving direction and then in a contrary right moving direction. The program begins by first setting up the ports as either output or input as shown in section (1). Here, as all ports are to be set as outputs, the control register for PA, known as PAC should only contain zeros. In section (2) bit 0 of Port A is set low which illuminates the first LED. Section (3) is a time delay to allow the LED illumination to be visible. In section (4) the RLC or rotate left instruction is used to move the illuminated LED one step to the left using the lamp register and the accumulator. At section (5) the lamp register is tested for all zeros. If this is the case, all the LEDs have been illuminated and the program jumps to the next section illuminating the LEDs in turn, one step to the right. This time the RRC or rotate right command is used as shown in section (6). A process similar to the above is repeated as shown in section (7). Finally the lamp register is checked for all zeros, which is an indication that all the LEDs have been illuminated. If so the program repeats the cycle jumping back to the beginning as shown in section (8) and starting the pattern with the left moving LEDs.

Traffic Light

This application uses red, green and yellow LEDs to simulate a crossroads traffic light function. Initially R1 and G2 are illuminated. After a delay the green light flashes followed by the yellow light. After another delay R2 and G1 are illuminated. This cycle will continue in this way indefinitely in the same way as the traffic lights at a traffic crossroad intersection. Within the application the different time durations for the red and green light as well as the flashing time can be programmed.



Circuit design

The circuit uses the two port sections PA0~PA2 and PA4~PA6 with each one representing a set of traffic lights on each road at a crossroad intersection. The operation of the circuit will be self explanatory from the contents of the program. See the circuit diagram for more details of the hardware.

Program

```
#include ht48c10.inc
; - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
data .section  data         ;== data section ==
count1 db ?                 ;delay loop counter 1
count2 db ?                 ;delay loop counter 2
count3 db ?                 ;delay loop counter 3
flash  db ?                 ;light flash register
rglight db ?                ;light register
; - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
code .section at 0  code  ;== program section ==
    org    00H              ;
    jmp    start            ;
    org    04h              ;external interrupt subroutine
    reti                    ;for safeguard
```

```
    org    08h              ;timer/event 0 interrupt subroutine
    reti                    ;for safeguard
    org    0ch              ;timer/event 1 interrupt subroutine
    reti                    ;for safeguard
start:                      ;
    clr    intc             ;initialize registers
    clr    tmrc             ;to guarantee performance
    clr    tmr              ;(interrupts)
    set    pac              ;(ports)
    set    pbc              ;(input mode)
    set    pcc              ;
main:
    mov    a,0       ;(1)   ;
    mov    pac,a            ;set port A to output port
    mov    pa,a             ;zero port A (all light on)
loop:                       ;light loop
    mov    a,0              ;;load table-read pointer
    mov    tblp,a           ;;
    tabrdl rglight   ;(2)   ;load light state by looking up table
    mov    a,rglight ;(3)   ;;output light state to port A
    mov    pa,a             ;;
    call   delayl    ;(4)   ;delay for a 'long' while
    inc    tblp      ;(5)   ;

    mov    a,7              ;load flash counter
    mov    flash,a          ;
flash1:                     ;flash1 loop
    tabrdl rglight          ;load light state
    mov    a,rglight        ;;
    mov    pa,a             ;;output light state to port A
    call   delays    ;(6)   ;delay for a 'little' while
    inc    tblp             ;
    sdz    flash            ;if flash light over?
    jmp    flash1           ;no. flash again
    tabrdl rglight          ;(yes. go ahead) load light state
    mov    a,rglight        ;;output light state to port A
    mov    pa,a             ;;
    call   delaym    ;(7)   ;delay for a 'medium' while
    inc    tblp             ;
; - - - - - - - - - - - - - - - - - - - - - - - - - - - - ;
    tabrdl rglight          ;load light state
    mov    a,rglight        ;;output light state to port A
    mov    pa,a             ;;
    call      delayl        ;delay for a 'long' while
    inc    tblp             ;

    mov    a,7              ;;load flash counter
    mov    flash,a          ;;
flash2:                     ;flash2 loop
    tabrdl rglight          ;load light state
```

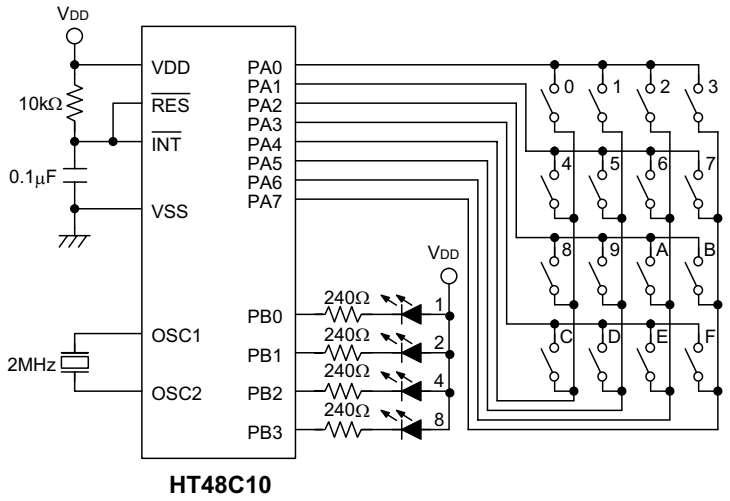
```
    mov    a,rglight        ;;output light state to port A
    mov    pa,a             ;
    call   delays           ;delay for a 'little' while
    inc    tblp             ;
    sdz    flash            ;if flash light over?
    jmp    flash2           ;no. flash again
    tabrdl rglight          ;(yes. go ahead) load light state
    mov    a,rglight        ;;output light state to port A
    mov    pa,a             ;;
    call   delaym           ;delay for a 'medium' while
    jmp    loop             ;repeat from light loop
delayl proc                 ;'long' delay subroutine
    mov    a,0fh            ;;load counters
    mov    count1,a         ;;
    mov    count2,a         ;;
    mov    count3,a         ;;
d1:
    sdz    count3           ;;count down count3
    jmp    d1
    sdz    count2           ;;count down count2
    jmp    d1
    sdz    count1           ;;count down count1
    jmp    d1
    ret
delayl endp
delaym proc                 ;'medium' delay subroutine
    mov    a,07h            ;;load counters
    mov    count1,a         ;;
    mov    a,0ffh           ;;
    mov    count2,a         ;;
    mov    count3,a         ;;
d2:
    sdz    count3           ;;count down count3
    jmp    d2
    sdz    count2           ;;count down count2
    jmp    d2
    sdz    count1           ;;count down count1
    jmp    d2
    ret
delaym endp
delays proc                 ;'little' delay subroutine
    mov    a,0ffh           ;;load counters
    mov    count1,a         ;;
    mov    count2,a         ;;
d3:
    sdz    count2           ;;count down count2
    jmp    d3
    sdz    count1           ;;count down count1
    jmp    d3
    ret
```

```
delays endp

    org    300h             ;TABLE
                            ; RYG  RYG
    dc     0EBh             ;1110 1011 G R
    dc     0FBh             ;1111 1011 O R
    dc     0EBh             ;1110 1011 G R
    dc     0FBh             ;1111 1011 O R
    dc     0EBh             ;1110 1011 G R
    dc     0FBh             ;1111 1011 O R
    dc     0EBh             ;1110 1011 G R
    dc     0FBh             ;1111 1011 O R
    dc     0DBh             ;1101 1011 Y R
    dc     0BEh             ;1011 1110 R G
    dc     0BFh             ;1011 1111 R O
    dc     0BEh             ;1011 1110 R G
    dc     0BFh             ;1011 1111 R O
    dc     0BEh             ;1011 1110 R G
    dc     0BFh             ;1011 1111 R O
    dc     0BEh             ;1011 1110 R G
    dc     0BFh             ;1011 1111 R O
    dc     0BDh             ;1011 1101 R Y

    end
```

Program description

The program begins (1) by setting the value of the port control register to determine which port bits are output and inputs. In this case each port bit of port control register PAC is set to 0 as all bits in this port are used as outputs. As the illuminated condition of the red, yellow and green lights is fixed a table read instruction can be used to determine their values as shown in section (2), however the values must first be setup. Because the last page TABRDL instruction is used the table data is setup from address 300h. The highest address is 3FFh. In section (5) the table pointer is incremented. In section (2) the display status of the LEDs is obtained from the table, this value being placed on the output port as shown in (3). Due to different timing delays being required for correct traffic light operation several procedures exist within the program to provide different timings as shown in (4) delayl, (7) delaym and (6) delays.

Keyboard Scanner



This unit uses a 4×4 keyboard matrix, giving a total of 16 keys with each key representing a single hexadecimal value as shown in the diagram. The microcontroller program scans the keyboard matrix to detect which key was pressed and after detection displays on the LED display the corresponding hex code. There are 4 LEDs, so a range of values from 0000 to 1111 can be displayed. During the scanning process, if two keys are pressed simultaneously only the first key scanned will be detected and displayed. By using this method 8 logic lines can control up to 16 switches with required values assigned to each key.

Circuit design

PA0~PA3 are assigned as outputs and PA4~PA7 assigned as inputs, together forming a 4×4 matrix. Note that during creation of the project, PA4~PA7 should have the pull-high option selected from the mask option. The program detects which key was pressed while a look up table defines the value of each key. PB0~PB3 are defined as outputs and represent a 4 bit hex code giving 16 different values with each value representing a single key.

Program

```
#include ht48c10.inc
; - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
data .section 'data'          ;== data section ==
temp    db  ?                 ;temporary data register
```

```
disp   db ?                ;key display register
count1 db ?                ;delay loop counter
mask   db ?                ;mask register
matrix db ?                ;key matrix register
; - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
code .section at 0  code  ;== program section ==
    org    00h             ;
    jmp    start           ;
    org    04h             ;external interrupt subroutine
    reti                   ;for safeguard
    org    08h             ;timer/event 0 interrupt subroutine
    reti                   ;for safeguard
    org    0ch             ;timer/event 1 interrupt subroutine
    reti                   ;for safeguard
start:                     ;
    clr    intc            ;initialize registers
    clr    tmrc            ;to guarantee performance
    clr    tmr             ;(interrupts)
    set    pac             ;(ports)
    set    pbc             ;(input mode)
    set    pcc             ;
main:
    set    pac       ;(1)  ;set port A to input mode
    clr    pbc             ;set port B to output mode
    clr    pa              ;zero port A (latch=0)
    set    pb              ;off LEDs
keyloop:                   ;
    mov    a,0feh    ;(2)  ;scan first row of keys
    mov    matrix,a        ;hold scan code
    mov    pac,a           ;pa.0 output 0 (latch)
    mov    a,pa            ;read input state
    cpl    acc             ;;complement input state
    and    a,0f0h          ;;
    sz     acc             ;if any input?
    jmp    get_key         ;yes. get input info
    mov    a,0fdh    ;(2)  ;no. scan second row
    mov    matrix,a        ;hold scan code
    mov    pac,a           ;pa.1 output 0 (latch)
    mov    a,pa            ;read input state
    cpl    acc             ;;complementl input state
    and    a,0f0h          ;;
    sz     acc             ;if any input?
    jmp    get_key         ;yes. get input info
    mov    a,0fbh    ;(2)  ;no. scan third row
    mov    matrix,a        ;hold scan code
    mov    pac,a           ;pa.2 output 0 (latch)
    mov    a,pa            ;read input state
    cpl    acc             ;;complement input state
    and    a,0f0h          ;;
```

```
    sz     acc              ;if any input?
    jmp    get_key          ;yes. get input info
    mov    a,0f7h    ;(2)   ;no. scan fourth row
    mov    matrix,a         ;hold scan code
    mov    pac,a            ;output pa.3 0 (latch)
    mov    a,pa             ;read input state
    cpl    acc              ;;complement input state
    and    a,0f0h           ;;
    sz     acc              ;if any input?
    jmp    get_key          ;yes. get input info
    jmp    keyloop          ;repeat from keyloop
get_key:                    ;get input key info
    call   delays           ;debounce
    mov    a,pa             ;test port A
    or     a,0fh            ;
    cpl    acc              ;
    sz     acc              ;any key hold?
    jmp    go_on            ;yes. go on (some key is pressed)
    jmp    keyloop          ;no. return to scan key again
go_on:
    call   key_in    ;(3)   ;calculate table index
    tabrdl disp      ;(10)  ;load display data
    mov    a,disp           ;;output data to port B
    mov    pb,a      ;(11)  ;;
    jmp    keyloop          ;repeat keyloop
key_in proc                 ;get key number
    mov    a,pa             ;;hold port A state
    mov    temp,a    ;(4)   ;;
get_release:                ;wait for the key to be released
    mov    a,pa             ;test port A state
    cpl    acc              ;
    and    a,0f0h           ;
    sz     acc       ;(6)   ;if release?
    jmp    get_release      ;no. keep up waiting
    mov    a,0fh            ;yes. calculate key number
    andm   a,matrix  ;(7)   ;mask low nibble of scan code
    mov    a,0              ;keep table index at register A
get_row:                    ;calculate row number
    rrc    matrix           ;check each bit to get row number
    snz    status.0         ;
    jmp    get_next         ;if zero goto get_next
    clr    c                ;
    add    a,4h      ;(8)   ;table index +4 (4 keys a row)
    jmp    get_row          ;continue calculating
get_next:                   ;
    mov    tblp,a           ;hold table index at register TBLP
    mov    a,0efh           ;
    mov    mask,a           ;mask=0111 1111
    mov    a,0fh            ;
    orm    a,temp           ;temp=XXXX 1111
```

```
get_column:                 ;calculate column number
    mov    a,temp           ;load temp
    xor    a,mask           ;test column number
    snz    z                ;
    jmp    index            ;no. test next column
    ret                     ;yes. return (TBLP)
index:                      ;next column
    inc    tblp      ;(9)   ;table index+1
    set    c                ;
    rlc    mask             ;shift mask left (LSB=1)
    jmp    get_column       ;repeat get_column key_in endp
delays proc                 ;delay subroutine
    mov    a,0ffh           ;load counter
    mov    count1,a         ;
d1:
    sdz    count1           ;count down count1
    jmp    d1
    ret
delays endp
    org    300h             ;display data table
    dc     0fh,0eh,0dh,0ch  ;key0, key1, key2, key3
    dc     0bh,0ah,09h,08h  ;key4, key5, key6, key7
    dc     07h,06h,05h,04h  ;key8, key9, keyA, keyB
    dc     03h,02h,01h,00h  ;keyC, keyD, keyE, keyF
    end
```

Program description

Section (1) defines whether the port bits are defined as inputs or outputs. The program enters a loop to determine which key is pressed. The code works by first scanning line by line to determine if any key has been pressed in that line. Because four keys are connected together in the same line, the software must determine the exact key, which has been pressed. For example in section (2) the first line is scanned to see if key 0~3 has been pressed. If so this code then jumps out of this scanning loop to a subsection (3), to determine which key connected to this line has been pressed. If not the code moves on to look at the line containing keys 4-7 and so on. After entering this code subsection, the code first stores the key value in a temporary register as shown in section (4). There then follows a short time delay (5), to take account of switch bounce and then code to determine when the switch has been released (6). The code will not continue until the key has been released. The next part is to determine which row has been pressed (7). When jumping from row to row the corresponding table address jumps by 4 bits each time (8). After determining the row, the pressed key connected to this row has to be found. To do this the value in the accumulator jumps in increments of one bit until the correct key is located, as shown in (9). The correct key has thus been found by first looking at the rows and

then at the columns and making the appropriate steps through the table values. In section (10) the table read instruction is used to determine the displayed value of the individual key, which is then placed on Port B which illuminates the corresponding LEDs.

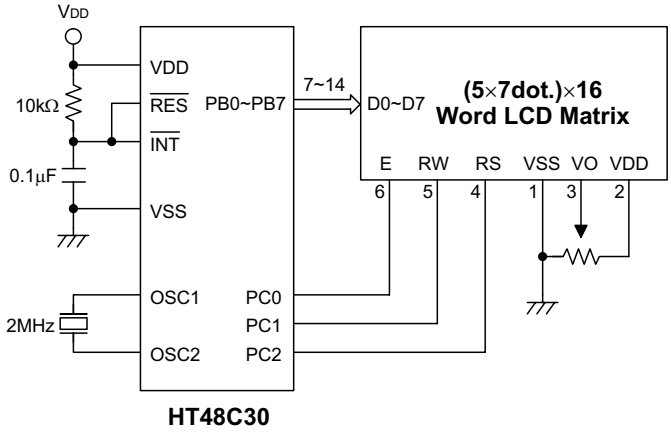
LCM

This unit describes the use of an 8-bit microcontroller used in conjunction with a DV16100NRB liquid crystal display. This LCM is driven and controlled by an internal Hitachi HD44780 device. In this application only the timing requirements of the LCM need to be considered to produce the correct microcontroller signals. For more detailed timing and instruction information, the LCM manufacturer's data should be consulted first.

LCMs can operate in either 4 bit or 8 bit mode. Using a 4 bit mode of operation, transmitting a character or an instruction to the module requires two transmission events to complete the operation. With an 8-bit mode of operation only one transmit event is required, however an extra 4 I/O lines are required. This section shows the use of the statements #define, if, else and endif.

Circuit design

PB0~PB7 are setup as I/O bits while PC0~PC2 as the LCM control lines are setup as outputs. These can be setup according to the specific user requirements.



Program

```
;==========
;= LCM.inc =
;==========
;this header file depends on what type of uC is used
ifndef HEADER_HT48C30
    #define HEADER_HT48C30
    #include ht48c30.inc
endif
; - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;for   DV-16100NRB
LCM_CLS       EQU 01H
CURSOR_HOME       EQU 02H
CURSOR_SR         EQU 14H
CURSOR_SL         EQU 10H
INCDD_CG_SHF_C    EQU 06H
TURN_ON_DISP      EQU 0FH
LCD_ON_CSR_OFF    EQU 0CH
; - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
LCM_DATA          EQU pb    ;define port B equal LCM data port
LCM_DATA_CTRL     EQU pbc   ;
LCM_CTRL          EQU pc    ;define port C eaual LCM control port
LCM_CTRL_CTRL     EQU pcc   ;
; - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;LCM Display Commands and control Signal name.
E                 EQU 0     ;signal pin number
RW                EQU 1     ;
RS                EQU 2     ;
;===========
;= main.asm =
;===========
#define HEADER_HT48C30
#include ht48c30.inc
#include lcm.inc
; - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;#define four_bit           ;define four_bit for 4-bit mode
extern busy_chk:near        ;import external module
extern delay:near           ;
extern write_char:near      ;
extern snd_cmd:near         ;
; - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
data .section 'data'        ;== data section ==
counter0   db ?
counter1   db ?
msg        db ?
tmp        db ?
; - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
code .section at 0 'code' ;== program section ==
    org    00H              ;
```

```
    jmp    start            ;
    org    04h              ;external interrupt subroutine
    reti                    ;for safeguard
    org    08h              ;timer/event 0 interrupt subroutine
    reti                    ;for safeguard
    org    0ch              ;timer/event 1 interrupt subroutine
    reti                    ;for safeguard
start:                      ;
    clr    intc             ;initialize registers
    clr    tmrc             ;to guarantee performance
    clr    tmr              ;(interrupts)
    set    pac              ;(ports)
    set    pbc              ;(input mode)
    set    pcc              ;
main:
    clr    LCM_DATA_CTRL    ;set LCM data port to output port
    clr    LCM_CTRL_CTRL    ;set LCM control port to output port
    clr    LCM_DATA
    clr    LCM_CTRL
DISPLAY_INIT:
ifdef four_bit              ;
    mov    a,20h            ;4-bit mode
else
    mov    a,30h            ;8-bit mode
endif
    mov    LCM_DATA,a
    set    LCM_CTRL.E       ;write instruction code to
    clr    LCM_CTRL.E       ;initalize LCM
LCM_DELAY:                  ;delay for LCM setup timing
    mov    a,0ffh           ;need about 4.5ms
    mov    counter1,a
    mov    counter0,a
lp0:
    sdz    counter1
    jmp    lp0
    sdz    counter0
    jmp    lp0
CMD_SEQ:
ifdef four_bit              ;
    mov    a,28h            ;4-bit mode and 2 lines (2 pass/byte)
else                        ;28h for 2 lines and 20h for 1 line
    mov    a,38h            ;8-bit mode and 2 lines
endif                       ;38h for 2 lines and 30h for 1 line
    mov    LCM_DATA,a       ;
    set    LCM_CTRL.E
    clr    LCM_CTRL.E
ifdef four_bit
    mov    a,80h            ;4-bit high nibble (2nd pass)
endif
    mov    LCM_DATA,a       ;write instruction code
```

```
    set    LCM_CTRL.E       ;
    clr    LCM_CTRL.E           ;
    call   busy_chk             ;check busy flag
    mov    a,LCM_CLS            ;clean display
    call   snd_cmd              ;
    call   busy_chk             ;check busy flag
    mov    a,TURN_ON_DISP       ;turn on display
    call   snd_cmd              ;
    call   busy_chk             ;check busy flag
    mov    a,INCDD_CG_SHF_C     ;auto increase mode
    call   snd_cmd              ;(cursor left and DD RAM address+1)
;  - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;inc addr of DD ram & shift
;the cursor to the right at
;the time of write to DD/CG
;RAM.
;  - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    call   busy_chk             ;check busy flag
    mov    a,LCM_CLS            ;clear display
    call   snd_cmd              ;
    call   busy_chk             ;check busy flag
    mov    a,CURSOR_HOME        ;cursor home
    call   snd_cmd              ;
    call   busy_chk             ;check busy flag
    clr    tblp                 ;load table pointer (=0)
;start show "HOLTEK 8 bit μC"
agn:
    tabrdl msg                  ;load message data
    mov    a,msg                ;
    mov    tmp,a                ;
    mov    a,24h                ;line end=24h
    xorm   a,msg                ;
    sz     msg                  ;if line end?
    jmp    agn1                 ;no. show next char
    jmp    secn_line            ;yes. show next line
agn1:
    call   busy_chk             ;check busy flag
    mov    a,tmp                ;
    call   write_char           ;write char to LCM
    inc    tblp                 ;next char (tblp+1)
    jmp    agn                  ;repeat from agn
secn_line:
    inc    tblp                 ;next char (tblp+1)
    call   busy_chk             ;check busy flag
    mov    a,0c0h               ;move cursor to 2nd line
    call   snd_cmd              ;(1st line:00h~, 2nd line:40h~)
    call   busy_chk
snd_line:
    tabrdl msg                  ;load message data
    mov    a,msg                ;
```

```
    mov    tmp,a                ;
    mov    a,24h                ;line end=24h
    xorm   a,msg                ;
    sz     msg                  ;if line end?
    jmp    snd_lin1             ;no. show next char
    mov    a,LCD_ON_CSR_OFF ;yes. hide cursor
    call   snd_cmd
    jmp    lp                   ;goto lp (end)
snd_lin1:
    mov    a,tmp                ;
    call   write_char           ;write char to LCM
    call   busy_chk             ;check busy flag
    inc    tblp                 ;tblp+1
    jmp    snd_line             ;repeat from snd_line
lp:
    jmp    lp                   ;end program
; - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
holtek_tbl .section at 700h 'code'    ;table at last page
htk_tbl:
    dc  0048h,004fh,004ch,0054h,0045h,004bh,0020h,0038h,0024h
    dc  0020h,0062h,0069h,0074h,0020h,0075h,0043h,0024h
end ;module
;===========
;= LCM.asm =
;===========
include lcm.inc
; - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;#define four_bit             ;define this for 4-bit mode
public busy_chk               ;
public delay                  ;
public write_char             ;
public snd_cmd                ;
; - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
dataLCM .section 'data'
dtmp   db ?
dtmp2  db ?
; - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
codeLCM .section 'code'
;=== snd_cmd ===
snd_cmd:                      ;=write instruction code=
ifdef four_bit                ;4-bit mode
    mov    dtmp,a
    and    a,0f0h             ;filter high nibble
endif
    mov    LCM_DATA,a         ;latch command
    clr    LCM_CTRL.RW        ;RW=0
    clr    LCM_CTRL.RS        ;RS=0
    set    LCM_CTRL.E         ;high
    clr    LCM_CTRL.E         ;low (trigger)
ifdef four_bit
```

```
    swapa  dtmp                 ;4-bit low nibble (two pass)
    and    a,0f0h
    mov    LCM_DATA,a           ;latch command
    set    LCM_CTRL.E           ;high
    clr    LCM_CTRL.E           ;low (trigger)
endif
    ret
;== busy_chk ==
busy_chk:                       ;=test busy flag=
    clr    LCM_CTRL.E           ;ready to pulse (low)
    set    LCM_DATA_CTRL        ;set LCM data port to input port
    clr    LCM_CTRL.RS          ;RS=0
    set    LCM_CTRL.RW          ;RW=1
    set    LCM_CTRL.E           ;pulse (high)
    mov    a,LCM_DATA           ;load busy flag
    clr    LCM_CTRL.E           ;pulse (low)
ifdef four_bit
    and    a,0f0h               ;4-bit mode high nibble (1st pass)
    mov    dtmp,a               ;
    set    LCM_CTRL.E           ;pulse (high)
    swapa  LCM_DATA             ;a.3~a.0.7~m.4+, a.7~a.4.3~m.0
    clr    LCM_CTRL.E           ;pulse (low)
    and    a,0fh                ;4-bit mode low nibble (2nd pass)
    or     a,dtmp               ;combine 2 pass
endif
    sz     acc.7                ;is busy?
    jmp    busy_chk             ;yes. check again
    clr    LCM_CTRL.RW          ;no. go ahead
    clr    LCM_DATA_CTRL        ;set LCM_DATA to output port
    ret
;== write_char ==
write_char:                     ;=write data to LCM=
ifdef four_bit                  ;4-bit mode (2 pass)
    mov    dtmp,a
    and    a,0f0h               ;filter high nibble
endif
    mov    LCM_DATA,a           ;latch data
    clr    LCM_CTRL.RW          ;RW=0
    set    LCM_CTRL.RS          ;RS=1 (write operation)
    set    LCM_CTRL.E           ;high
    clr    LCM_CTRL.E           ;low (trigger)
ifdef four_bit
    swapa  dtmp                 ;4-bit mode (2nd pass)
    and    a,0f0h               ;filter low nibble
    mov    LCM_DATA,a           ;latch data
    set    LCM_CTRL.E           ;high
    clr    LCM_CTRL.E           ;low (trigger)
endif
    ret
;== delay ==
```

```
delay:                              ;=delay for a while=
    mov    dtmp,a
    drep:
    sdz    dtmp2                    ;count down dtmp2
    jmp    drep
    sdz    dtmp                     ;count down dtmp
    jmp    drep
    ret
end ;module
```

Program description

The program begins by calling in include files and by defining the LCM to be on Port B. The LCM control lines are defined on Port C. External modules are also declared to define whether the LCM is in 4-bit or 8-bit mode.

The LCM will automatically conduct an internal reset during power-up. However most program controlled LCMs will still utilize software for their initialization. In this example the code begins at start by running some initialization code. According to the data for the HD44780, there needs to be at least 4.5 ms delay between each program. This is the function of the LCM_DELAY code section. Before the LCM initialization has been carried out it is not possible to check the status of BUSY. To issue instructions to the LCM refer to the HD44780 instruction definitions. The code section LCM.INC contains several often used instructions. Before the LCM writes instructions it must first check whether LCM is in a busy condition. The code section BUSY_CHK exists to check the BUSY status of the LCM. After checking this, data can then be sent to the LCM. The ASCII codes to be displayed should be placed in the last page of the program, and a table look up method used to place the data in the accumulator, from where the code section WRITE_CHAR can display it on the LCM.

Operation truth table of signal RS, R/W and E:

RS	RW	E	Operation
0	0		Write instruction code
0	1		Read busy flag & address counter
1	0		Write data
1	1		Read data

Using an I/O Port as a Serial Application

This section shows code to simulate serial port operation. This can be used as a basis for the development of simple serial port applications such as 8-bit communication, non-parity, single stop bit applications. Here the example is given as a project with the user left to decide which pins are send and receive and the baudrate decided by the system frequency. Below are the steps taken and other points to be noted.

- Create a new file called DFSERIPT.INC in the directory \HT-IDE2000\INCLUDE\Here must be included ...
  - Baudrateconst XXX. This value XXX can be taken from a table or calculated (ignore the decimal point)
  - TXPIN ..... the defined transmit pin
  - RXPIN ..... the defined receive pin
- The serial port program uses 4 RAM locations, 2 I/O pins and 49 program memory locations. The TXPIN must be defined as an output and RXPIN defined as an input.
- Before Call or Receive, the RXPINs stop bit condition must be checked Calling these two subroutines will change the condition of the carry flag.
- When using Routines, the program must first declare them as follows:

```
EXTERN      TRANSMIT: NEAR
EXTERN      RECEIVE: NEAR
```

- Below the main program serial.asm must be added.
  The parameter baudrateconst is calculated from the baudrate as follows.
  Baudrateconst = (Fsys/(baudrate*12))-3 (ignore the decimal point)

Note The best value for baudrateconst is within a range of 7 to 256, the bigger the value the smaller the error. The following table gives some direct values.

Refer to the table below:

Baudrate/Fsys	4MHz	2MHz	1MHz
9600	31	14	X
7200	43	20	8
4800	66	31	14
3600	89	43	20
2400	135	66	31
2000	163	80	38
1800	182	89	43
1200	X	135	66

Note X means unable to use, reduce either Fsys or the baudrate.

Program

```
;================
;= Dfseript.inc =
;================
BAUDRATECONST  EQU  66        ;table look-up (4MHz, 4800 bits/second)
                              ;or calculate (see HT-IDE2000 User's Guide)
TXPIN   EQU PA.3              ;transmit pin
RXPIN   EQU PA.2              ;receive pin
;============
;= Main.asm =
;============
#define HEADER_HT48C70
#include ht48c70.inc
#include dfseript.inc
;#define  TRANSMIT_MODE       ;define TRANSMIT_MODE for transmit mode
                              ;default is receive mode
extern transmit:near          ;external functions
extern receive:near
; - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
data .section 'data'          ;== data section ==
transmit_data    db ?
receive_data     db ?
counter       db ?
; - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
code .section at 0 'code'     ;== program section ==
    org    00H                ;
    jmp    start              ;
    org    04h                ;external interrupt subroutine
    reti                      ;for safeguard
    org    08h                ;timer/event 0 interrupt subroutine
    reti                      ;for safeguard
    org    0ch                ;timer/event 1 interrupt subroutine
    reti                      ;for safeguard
start:                        ;
    clr    intc               ;initialize registers
    clr    tmr0c              ;to guarantee performance
    clr    tmr0h              ;(interrupts)
    clr    tmr0l              ;
    clr    tmr1c              ;
    clr    tmr1h              ;
    clr    tmr1l              ;
    set    pac                ;(ports)
    set    pbc                ;(input mode)
    set    pcc                ;
    set    pdc                ;
```

```
    set    pec              ;
    set    pfc              ;
    set    pgc              ;
main:
    set    pac.2            ;set receive pin to input mode
    clr    pac.3            ;set transmit pin to output mode
loop:
ifdef  TRANSMIT_MODE        ;transmit mode
    mov    a,32             ;transmit 32 bytes
    mov    counter,a
    clr    tblp             ;tble=0 (data pointer)
again:
    tabrdl transmit_data    ;load transmit data
    mov    a,transmit_data
    call   transmit         ;transmit
    inc    tblp             ;tble+1 (point to next)
    sdz    counter          ;transmit over? (counter-1)
    jmp    again            ;no. next byte
    jmp    loop             ;repeat from loop else
    mov    a,40h            ;receive mode
    mov    mp0,a            ;mp0-receive buffer (40h~5Fh)
    mov    a,32             ;load counter
    mov    counter,a        ;
again:
    call   receive          ;receive 1 byte
    mov    r0,a             ;put received data to buffer
    inc    mp0              ;buffer index+1
    sdz    counter          ;receive over?
    jmp    again            ;no. continue receiving
    jmp    $                ;yes. stop program
endif
test .section at 1f00h 'code'
test_table:                 ;data fo transmitting (32 bytes)
    dc  012h,034h,056h,078h,09ah,0bch,0deh,0f0h,011h,022h,033h,
        044h,055h,066h,077h,088h
    dc  099h,0aah,0bbh,0cch,0ddh,0eeh,0ffh,000h,055h,0aah,055h,
        0aah,055h,0aah,055h,0aah
    end
;==============
;= Serial.asm =
;==============
;serial port library
ifndef HEADER_HT48C70       ;depends on what type of IC is selected
#define HEADER_HT48C70
#include ht48c70.inc
endif
#include dfseript.inc
public transmit             ;external functions
public receive              ;
baudrate   equ  baudrateconst   ;replace baudrate with
```

```
baudrateconst
tx          equ    txpin          ;replace txpin with tx
rx          equ    rxpin          ;replace rxpin with rx
sdata .section  data         ;
count   db  ?                     ;serial bit counter
txreg   db  ?                     ;transmit data register
rcreg   db  ?                     ;receive data register
delay   db  ?                     ;delay counter
serial .section 'code'            ;
transmit proc                     ;transmit a byte (Acc)
    mov    txreg,a                ;hold Acc at txreg
    mov    a,baudrate             ;load delay counter
    mov    delay,a                ;
    clr    tx                     ;send start bit '0'
    mov    a,9                    ;load bit counter
    mov    count,a                ;
txdelay:                          ;
    sdz    delay                  ;delay to fit baudrate
    jmp    txdelay                ;
    mov    a,baudrate             ;reload delay counter
    mov    delay,a                ;
    sdz    count                  ;if transmit over?
    jmp    sendbit                ;no. send next bit
    jmp    endtx                  ;yes. go ahead
sendbit:
    rrc    txreg                  ;shift right through carry flag
    snz    c                      ;is '1'
    jmp    lobit                  ;no. goto lobit
    set    tx                     ;yes. Send '1'
    jmp    txdelay                ;repeat from txdelay
lobit:                            ;
    clr    tx                     ;send '0'
    jmp    txdelay                ;repeat from txdelay
endtx:                            ;
    nop                           ;delay for a while
    nop                           ;
    set    tx                     ;send stop bit '1'
t1:                               ;
    sdz    delay                  ;delay between bits
    jmp    t1                     ;(timing adjustment)
    mov    a,baudrate             ;
    mov    delay,a                ;
t2:                               ;
    sdz    delay                  ;
    jmp    t2                     ;
    ret                           ;
transmit   endp                   ;
receive proc                      ;receive a byte
    sz     rx                     ;if start bit '0'
    jmp    receive                ;no. test again
```

```
    mov    a,9                ;yes. start receiving
    mov    count,a            ;load bit counter
    mov    a,baudrate+1       ;load delay counter
    mov    delay,a            ;(+1 for timing adjustment)
rxdelay:                      ;
    sdz    delay              ;delay to fit baudrate
    jmp    rxdelay            ;
    mov    a,baudrate+1       ;reload delay counter
    mov    delay,a            ;(+1 for timing adjustment)
    sdz    count              ;if receive over?
    jmp    rxbit              ;no. receive next bit
    mov    a,rcreg            ;yes. put received data to Acc
    ret                       ;
rxbit:                        ;
    set    c                  ;c=1
    snz    rx                 ;if received '1'?
    clr    c                  ;no. c=0
    rrc    rcreg              ;shift left through carry flag
    jmp    rxdelay            ;repeat from rxdelay
receive endp                  ;
    end
```

Program description

The most important element here is the baudrate parameter because the transmiting and receiving of data have to be coordinated with the baudrate speed. For this reason a formula as well as a table is provided to define this constant. Because in this example the baudrate is not defined by using the timer/counter some small discrepencies may exist, however according to our tests using an 8051 the values given in the table are error free. The higher the baudrate parameter the lower the error rate so by adjusting the value of baudrate and system frequency the baudrate parameter value can be raised.

Chapter 17

Interrupt and Timer/Counter Applications

17

There are many different methods of using a timer/counter method to implement an interrupt. For example, if it is required to have a timed signal, or after a fixed or non-fixed period of time to have an event occur, the timer counter can be used. During the specified time or count period, the main program can continue with other functions, when the time or count period is over an interrupt will be activated. This interrupt can then be used to run special interrupt code or trigger other special functions. When the interrupt function ends the code returns to running the main program.

The Holtek HT48X00 microcontroller series timer/counter possess either 16 bit or 8 bit counters. All are count up types. The values are first converted into 2 s complement and then the 16 or 8 bit value loaded into the timer/counter. Additionally the timer/counters can be divided up into three types, event counter, timer or pulse width measurement type. The event counter type receives generated signals from outside while the timer type uses the internal system clock as its base timing.

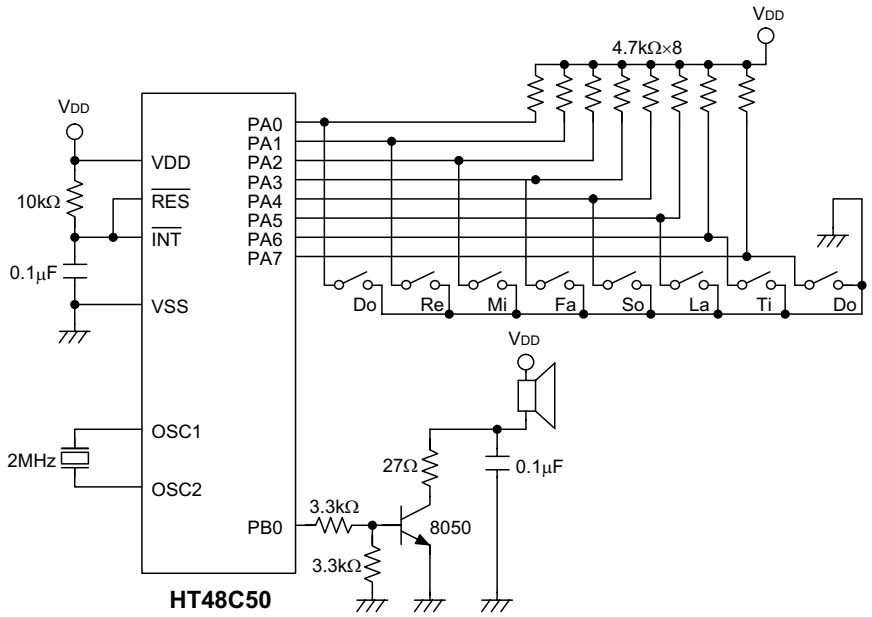
Electronic Piano

This unit describes how to implement a scanning keyboard and then from the pressed key generate a corresponding defined sound frequency. Each time a key is pressed the corresponding frequency value is placed into the timer/counter register. When this counter counts to its maximum value an internal interrupt is generated and the interrupt routine is run. At this point the timer/counter register value is reloaded and the counting continues. In this way, by programming different values into the timer/counter register, different values of frequency can be generated. The internal interrupt routine contains code to change the state of the output port and thus

generate the required frequency on a corresponding pin and create the desired note. By adding a suitable amplifier and speaker the system is complete. The important points of the software is to use the timer/counter as a counter to control the output frequency. This frequency has to be calculated.

Circuit design

PA0~PA7 are setup as inputs with each line connected to a pull up resistor. Pressing a key will bring the corresponding line low. PB0 is setup as an output and is the line where the required frequency appears. By changing this line from hi to lo and vice-versa the required frequency can be generated.



Program

```
#include ht48c50.inc
; - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
data .section  data          ;== data section ==
temp    db  ?                ;hold temporary data
sound   db  ?                ;hold freq.
; - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
code .section at 0  code     ;== program section ==
    org    00h               ;
    jmp    start             ;
    org    04h               ;external interrupt subroutine
    reti                     ;for safeguard
```

```
    org    08h              ;timer/event 0 interrupt subroutine
    cpl    pb        ;(6)   ;generate square wave
    reti                    ;end timer0 ISR
    org    0ch              ;timer/event 1 interrupt subroutine
    reti                    ;for safeguard
start:                      ;
    clr    intc             ;initialize registers
    clr    tmr0c            ;to guarantee performance
    clr    tmr0h            ;(interrupts)
    clr    tmr0l            ;
    clr    tmr1c            ;
    clr    tmr1             ;
    set    pac              ;(ports)
    set    pbc              ;(input mode)
    set    pcc              ;
    set    pdc              ;
main:
    set    pac       ;(1)   ;set port A to input port
    clr    pbc              ;set port B to output port
    clr    pb               ;
keyloop:             ;(2)   ;
    mov    a,pa             ;test any input
    cpl    acc              ;
    sz     acc              ;if any?
    call   whichkey         ;yes. find out which key
    jmp    keyloop          ;no. repeat from keyloop
whichkey proc        ;(3)   ;find out which key
    mov    temp,a           ;hold Acc content
    mov    a,0              ;zero table index
    mov    tblp,a           ;
    clr    c                ;c=0 (check each bit(key) by carry
flag)
keynext:                    ;
    rrc    temp             ;shift right through carry flag
    sz     status.0         ;if carry? (some key was pressed)
    jmp    timerset         ;yes. output sound
    inc    tblp      ;(4)   ;no. table index point to next
    inc    tblp             ;(2 bytes/key)
    jmp    keynext          ;check next bit (key)
timerset:                   ;set timer to generate sound
    mov    a,5              ;enable timer0
    mov    intc,a           ;
    mov    a,80h            ;set timer0 mode (internal clock)
    mov    tmr0c,a          ;
    tabrdl sound     ;(5)   ;load freq.
    mov    a,sound          ;(low byte)
    mov    tmr0l,a          ;
    inc    tblp             ;
    tabrdl sound     ;(6)   ;(high byte)
    mov    a,sound          ;
```

```
    mov    tmr0h,a          ;
    set    tmr0c.4          ;start timer0
key_halt:            ;(7)   ;check key holding state
    mov    a,pa             ;test pa
    cpl    acc              ;
    sz     acc              ;is holding?
    jmp    key_halt         ;yes. check again
    clr    tmr0c.4   ;(8)   ;no. stop timer (stop sound two)
    clr    pb               ;
    ret                     ;
whichkey endp               ;
    org 0f00h               ;sound freq.
    dc  21h,0feh,58h,0feh
    dc  84h,0feh,99h,0feh
    dc  0c1h,0feh,0e3h,0feh
    dc  02h,0ffh,11h,0ffh
    end
```

Program description

The program begins (1) by setting all pins on Port A to inputs by setting the Port A control register PAC high. Port B is setup as outputs by clearing its control register PBC. After this the program enters a program loop (2) to detect if a key has been pressed. If a key has been pressed the program jumps out of this loop otherwise it will remain in this loop until a key is pressed. The next stage is a program to determine which key was pressed (3). Determining which key is pressed enables the correct frequency value to be obtained from the table. This is done by incrementing the table pointer until the correct location is reached. The value is divided into a high byte and low byte, these two values have to be placed in the correct high/low byte position of the timer/counter (5). After the value is placed and counted an interrupt is generated by the timer/counter and the interrupt routine is run and the correct output (6) placed on Port B. In this way the correct frequency can be generated. The Port B output pin is connected to a suitable amplifier and speaker to generate the correct tone.

After detecting the key and obtaining the correct frequency value the key is again examined (7) to see if it has been released. This is because the timer/counter will continue to be reloaded so if the key is not released the note will continue to appear on Port B. If the key has been released (8) then the control bit is set to turn off the timer/counter.

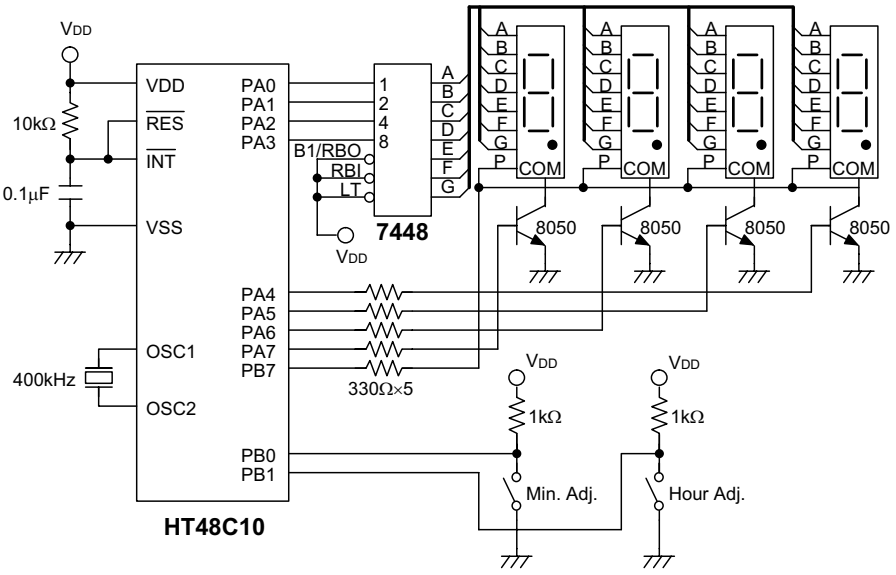
Clock

This application shows the use of the 16 bits of the timer counter to generate internal interrupts and consequently generate a timing function. This

application depends upon the system clock frequency as a basis for its timing. The application shown here uses a 400kHz system clock which will generate a 100kHz timer/counter clock due to the internal divide by four operation. With a 16 bit counter the maximum count is 65536, this would generate an internal interrupt every 0.65536 seconds. However for a clock function a basic time unit of 1 second is required so for this reason the timer/counter is setup to record a basic timing of 0.5 seconds. In this case an interrupt will be generated every 0.5 seconds, so by counting two interrupts a means of obtaining the basic timing unit of 1 second is obtained. The application shown uses 4 seven segment displays to display a clock in 24 hour format, displaying both hours and minutes. Two keys are provided to provide for adjustment of hours and minutes.

Circuit design

PA0~PA7 are setup as outputs with PA0~PA3 setup as the display data. PA4~PA7 provide scanning inputs to the control transistors for the segment displays. These will scan the individual displays one after the other. PB0 and PB1 are setup as inputs for the switches which enable the hours and minutes to be preset.



Program

```
#include ht48c50.inc
; - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
data .section  data          ;== data section ==
second db  ?                 ;hold second
minl   db  ?                 ;hold minute low byte
minh   db  ?                 ;hold minute high byte
hourl  db  ?                 ;hold hour low byte
hourh  db  ?                 ;hold hour high byte
count1 db  ?                 ;delay counter
mask   db  ?                 ;hold mask
disp   db  ?                 ;hold display data
; - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
code .section at 0  code     ;== program section ==
    org    00h               ;
    jmp    start             ;
    org    04h               ;external interrupt subroutine
    reti                     ;for safeguard
    org    08h               ;timer/event 0 interrupt subroutine
    inc    second            ;second+1 (unit: 0.5second)
    cpl    pb                ;flash  dot  (on 0.5s then off 0.5s)
    reti                     ;end ISR
    org    0ch               ;timer/event 1 interrupt subroutine
    reti                     ;for safeguard
start:                       ;
    clr    intc              ;initialize registers
    clr    tmr0c             ;to guarantee performance
    clr    tmr0h             ;(interrupts)
    clr    tmr0l             ;
    clr    tmr1c             ;
    clr    tmr1              ;
    set    pac               ;(ports)
    set    pbc               ;(input mode)
    set    pcc               ;
    set    pdc               ;
main:
    clr    pac         ;(1)  ;set port A to output port
    mov    a,7fh             ;set port B to input port
    mov    pbc,a             ;exclude pb.7
    clr    pb          ;(2)  ;zero variables
    clr    pa                ;
    clr    minl              ;
    clr    minh              ;
    clr    hourl             ;
    clr    hourh             ;
    clr    second            ;
    mov    a,05h             ;enable timer0
    mov    intc,a            ;
    mov    a,80h             ;set timer0 mode (internal clock)
```

```
    mov    tmr0c,a          ;
    mov    a,0b0h    ;(5)   ;load timer0 counter (0.5 second)
    mov    tmr0l,a          ;(low byte)
    mov    a,3ch            ;
    mov    tmr0h,a          ;(high byte)
    set    tmr0c.4          ;start timer0
loop:                ;(3)   ;
    mov    a,0              ;zero table index
    mov    tblp,a           ;
    mov    a,minl           ;load display data
    mov    disp,a           ;( low  minute)
    call   show_clock       ;show displaying up (4th 7-segment)
    inc    tblp             ;
    mov    a,minh           ;load displaying data
    mov    disp,a           ;( high  minute)
    call   show_clock       ;show displaying up (3rd 7-segment)
    inc    tblp             ;
    mov    a,hourl          ;load displaying data
    mov    disp,a           ;( low  hour)
    call   show_clock       ;show displaying up (2nd 7-segment)
    inc    tblp             ;
    mov    a,hourh          ;load displaying data
    mov    disp,a           ;( high  hour)
    call   show_clock       ;show displaying up (1st 7-segment)
    jmp    loop             ;repeat from loop
; - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
cal_number proc             ;
    inc    minl             ;minl+1
    mov    a,minl           ;
    sub    a,0ah            ;
    sz     acc              ;if over 10 minutes?
    ret                     ;no. return
    clr    minl             ;yes. minl=0
    inc    minh             ;minh+1
    mov    a,minh           ;
    sub    a,06h            ;
    sz     acc              ;if over 60 minutes?
    ret                     ;no. return
    clr    minh             ;yes. minh=0
    mov    a,hourh          ;
    sub    a,02h            ;
    sz     acc              ;if over 20 hours?
    jmp    h_20             ;no. goto h_20
    inc    hourl            ;yes. hourl+1
    mov    a,hourl          ;
    sub    a,04h            ;
    sz     acc              ;if over 24 hours?
    ret                     ;no. return
    clr    hourl            ;yes. hourl=0
```

```
    clr    hourh            ;hourh=0
    ret                     ;return
h_20:                       ;
    inc    hourl            ;hour1+1
    mov    a,hourl          ;
    sub    a,0ah            ;
    sz     acc              ;if over 10 hours?
    ret                     ;no. return
    clr    hourl            ;yes. hourl=0
    inc    hourh            ;hourh+1
    ret                     ;return
cal_number endp             ;
; - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
show_clock proc             ;
    mov    a,1fh            ;load counter
    mov    count1,a         ;
    tabrdl mask             ;load mask
    mov    a,mask           ;(active some 7-segment LED)
    or     a,disp           ;mask displaying data
    mov    pa,a             ;(update the 7-segment LED)
d1:                         ;
    snz    pb.0      ;(4)   ;if key Min. Adj. is being pressed?
    jmp    min_inc          ;yes. deal with it
    snz    pb.1      ;(4)   ;no. if key Hour Adj is beeing
pressed?
    jmp    hour_inc         ;yes. deal with it
    mov    a,second         ;no. check sceond overflow
    clr    c                ;
    sub    a,78h     ;(6)   ;78h=120 (unit: 0.5 second)
    sz     acc              ;if overflow?
    jmp    scan_next        ;no. continue scanning
    clr    second           ;yes. secon=0
    call   cal_number ;(7)  ;calculate clock digits
    ret                     ;
scan_next:                  ;
    sdz    count1           ;if count over (counter-1)
    jmp    d1               ;no. scan keys again
    ret                     ;yes. return
min_inc:                    ;
    call   delays           ;delay for key releasing
    snz    pb.0             ;if key was released?
    jmp    min_inc          ;no. test again
    call   inc_min          ;yes. minute+1
    clr    second           ;second=0 (reset secnod)
    clr    tmr0c.4          ;stop timer0
    mov    a,0b0h    ;(5)   ;reload timer0 counter
    mov    tmr0l,a          ;
    mov    a,3ch            ;(0.5 second)
    mov    tmr0h,a          ;
    set    tmr0c.4          ;restart timer0
```

```
    ret                     ;
hour_inc:                   ;
    call   delays           ;delay for key releasing
    snz    pb.1             ;if key was released?
    jmp    hour_inc         ;no. test again
    call   inc_hour         ;yes. hour+1
    clr    second           ;second=0
    clr    tmr0c.4          ;stop timer0
    mov    a,0b0h    ;(5)   ;reload timer0 counter
    mov    tmr0l,a          ;
    mov    a,3ch            ;(0.5 second)
    mov    tmr0h,a          ;
    set    tmr0c.4          ;restart timer0
    ret                     ;
show_clock endp             ;
; - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    org    0f00h            ;mask data
    dc     10h,20h,40h,80h
; - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
delays proc                 ;
    mov    a,7fh            ;load counter
    mov    count1,a         ;
d2:                         ;
    sdz    count1           ;count down count1
    jmp    d2               ;
    ret                     ;
delays endp                 ;
; - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
inc_hour proc               ;
    mov    a,hourh          ;
    sub    a,02h            ;
    sz     acc              ;if over 20 hours
    jmp    h_201            ;no. goto h_201
    inc    hourl            ;yes. hourl+1
    mov    a,hourl          ;
    sub    a,04h            ;
    sz     acc              ;if over 24 hours?
    ret                     ;no. return
    clr    hourl            ;yes. hourl=0
    clr    hourh            ;hourh=0
    ret                     ;return
h_201:                      ;
    inc    hourl            ;hourl+1
    mov    a,hourl          ;
    sub    a,0ah            ;
    sz     acc              ;if over 10 hours?
    ret                     ;no. return
    clr    hourl            ;yes. hourl=0
    inc    hourh            ;hourh=0
```

```
    ret                     ;
inc_hour  endp              ;
;  - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
inc_min proc                ;
    inc    minl             ;minl+1
    mov    a,minl           ;
    sub    a,0ah            ;
    sz     acc              ;if over 10 minutes?
    ret                     ;no. return
    clr    minl             ;yes. minl=0
    inc    minh             ;minh=0
    mov    a,minh           ;
    sub    a,06h            ;
    sz     acc              ;if over 60 minutes?
    ret                     ;no. return
    clr    minh             ;yes. minh=0 (don t care hour)
    ret                     ;return
inc_min endp                ;
    end
```

Program description

The program begins (1) by defining A as outputs, achieved by setting the control register PAC to 00. With the exception of bit 7, all bits in Port B are set as inputs, achieved by setting the control register PBC to 7F. The next stage (2) is to clear various internal locations of the RAM and enable the interrupt and timer/counter. Following on from this (3) is the program to display the time and the program to determine if any key has been pressed. If the time is to be adjusted because a key has been pressed then the program calls a routine (4) to adjust either the minutes or the hour. A value of one will be added to either the minute or hour value. If the minute value has reached 59 the next value will be set to 00 but the hour value will not be increased. Similarly if the hour value has reached 23 the next value will be 00 but the minute value will not be changed. If no key has been pressed the program remains in the program loop to display the time. Because the program has been setup to provide an interrupt every 0.5 seconds, a 1 minute time interval is generated every 120 interrupts. Each time this happens the minutes count will be increased by one. During the program loop to display the time, the program will look at the interrupt count number to determine if a value of 120 has been reached (6). If so then the minutes count will be increased by one (7). Every 60 minutes will become a count of one hour and after 23 hours the hour count will reset to 00 hours. The information containing the time for the display is placed upon 4 bits of Port A and the scanning operation on the remaining 4 bits. In this way the display is only active for a certain period of time, but because the scanning speed is high the display appears to be showing its data continuously.

Chapter 18

Parallel Port

18

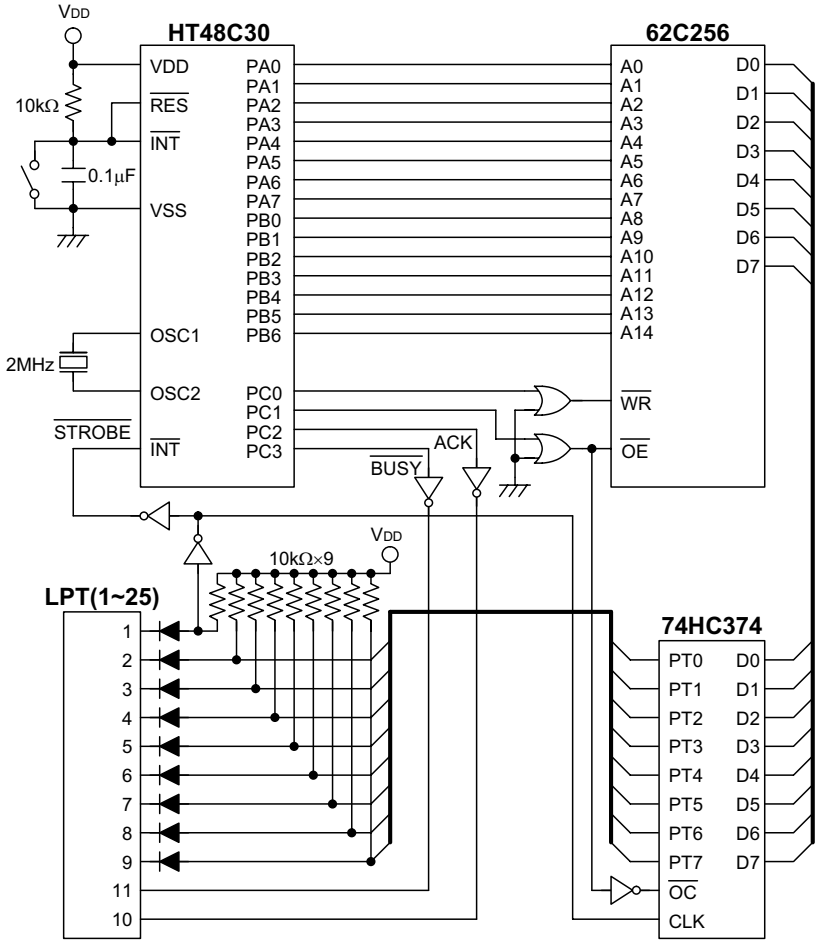
This section describes a parallel port application. The speed of parallel ports is higher than serial ports as there are more lines involved in the data transfer. Most parallel port line applications use TTL standard voltage levels, therefore operate best within a line range of several feet. Lines of excessive length, due to the increases resistance of the lines, can generate erroneous data transfers. For these reasons, parallel port applications are normally only used for short length data transfer operations. Because parallel port applications require several data lines to transmit data and because several control lines are also required the application is well suited for the HT48x00 series of controllers.

ROM Emulator

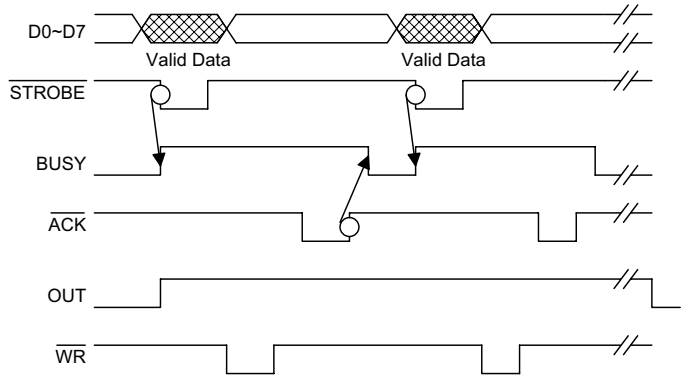
During the first stages of system development it is normally impractical to burn the required ROM. The usual procedure is to use a suitable EPROM to replace the ROM during the stages of program and system development. In this way, by using an ultra-violet light source, the internal EPROM programmed data can be erased and the device reused. The inconvenience in this approach is however, using an ultraviolet light source and the time required to erase the data. An alternative solution, which overcomes these EPROM limitations, is to use a ROM emulator. This application uses the PCs printer port to download data into the ROM emulator SRAM. Such a ROM emulator can reduce significantly early program development effort. In this application the design is implemented using the HT48300, 28-pin microcontroller.

Circuit design

I/O lines PA0~PA7, PB0~PB7 and PC0~PC3 are defined as outputs. Port A is defined as the low order bits while Port B is defined as the high order bits. Port C is for the control lines. To use in a practical application it is necessary to connect both the address and data lines of the SRAM to the ROM socket of the application.



The program has to follow the parallel port conventions and timing diagram. For this reason the timing specification of the PC printer port can be consulted to achieve the correct operation.



Program

```
#include      ht48300.inc
; - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
data   .section  'data'        ; data section
addrl  db ?                    ; low byte address register
addrh  db ?                    ; high byte address register
timer_ov  db ?                 ; timer overflow register
; - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;      PC0       WR
;      PC1       OE
;      PC2       ACK
;      PC3       BUSY
; - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
code   .section  at 0 'code'   ; program section
    org    00h                 ; ISR address setup
    jmp    start
    org    04h                 ; external INT ISR
    jmp    int_sub             ;
    org    08h                 ; timer ISR
    jmp    timer_sub           ;
    org    0Ch
int_sub:                       ;
    mov    a,00000011b         ; move acc=00000011,
                               ; (BUSY=0, ACK=0, OE=1, WR=1)
    mov    pc,a          ;(5)   ; output acc to port c
    reti                       ; return from external ISR
timer_sub:                     ;
    inc    timer_ov            ; increment timer overflow regis-
ter
    mov    a,timer_ov          ;
```

```
    xor    a,20h                ; timer overflow register "XOR"
                                ; with 20h
    sz     acc                  ; check over 20h
    jmp    timer_nov            ; if not over 20h , jump timer_nov
    mov    a,00001011b          ;
    reti                 ;(4)   ; return from ISR
timer_nov:                      ; timer_nov
    clr    acc                  ;
    reti                        ; return ISR
start:                          ; program start
    mov    a,07h         ;(1)   ; setup INTC (external INT ON,
                                ; timer ON)
    mov    intc,a               ;
    mov    a,80h                ; setup timer mode (internal
clock)
    mov    mrc,a                ;
    clr    pcc                  ; set port C as output port
top:                    ;(13)  ; top
    set    pac          ;(12)  ; set port A as input port
    set    pbc                  ; set port B as input port
    clr    addrl                ; clear low byte address register
    clr    addrh                ; clear high byte address register
    mov    a,00001001b          ; move acc=00001001,
                                ; (BUSY=1, ACK=0, OE=0, WR=1)
    mov    pc,a          ;(3)   ; output acc to port C
    clr    acc                  ;
store:                          ; store
    snz    acc.0         ;(4)   ; check acc.0="0".if external INT
                                ; interrupt,will change acc.0="1"
    jmp    store                ; if acc.0="0" jump store
    clr       pac        ;(5)   ; if acc.0="1", set port A as
                                ; output port
    clr    pbc                  ; set port B as output port
next:                           ; next
    mov    a,addrl       ;(6)   ;
    mov    pa,a                 ; output low byte address register
                                ; to port A
    mov    a,addrh              ;
    mov    pb,a                 ; output low byte address register
                                ; to port B
    inc    addrl                ; increment low byte address reg-
ister
    sz     addrl                ; check if equal to 0
    jmp    no_inc               ; if not 0 jump no_inc
    mov    a,1                  ;
    addm   a,addrh              ; increment high byte address
                                ; register
no_inc:                         ; no_inc
    mov    a,00000010b          ; move acc=00000010,
                                ; (BUSY=0, ACK=0, OE=1,WR=0)
```

```
    mov    pc,a          ;(7)   ; output acc to port C
    mov    a,10h                ; setup delay time
    call  delays         ;(8)   ; call delays subroutine
    mov    a,00000011b          ; move acc=00000011,
                                ; (BUSY=0, ACK=0, OE=1, WR=1)
    mov    pc,a                 ; output acc to port C
    mov    a,7                  ; setup delay time
    call   delays               ; call delays subroutine
    mov    a,00000111b          ; move acc=00000111,
                                ; (BUSY=0, ACK=1, OE=1, WR=1)
    mov    pc,a                 ; output acc to port C
    mov    a,5                  ; setup delay time
    call   delays               ; call delays subroutine
    mov    a,00000011b          ; move acc=00000011,
                                ; (BUSY=0, ACK=0, OE=1, WR=1)
    mov    pc,a          ;(9)   ; output acc to port C
    mov    a,5                  ; setup delay time
    call   delays               ; call delays subroutine
    clr    tmr                  ; clear timer
    set    tmrc.4        ;(10)  ; setup timer control reg. start
                                ; counting
    mov    a,00001011b          ; move acc=00001011,
                                ; (BUSY=1, ACK=0, OE=1, WR=1)
    mov    pc,a          ;(11)  ; output acc to port C
    clr    acc                  ; clear acc

store1:                         ; store1
    snz    acc.0         ;(4)   ; check if acc.0="0" .if external
                                ; INT
                                ; interrupt, will change acc.0="1"
    jmp    store1               ; if "0" jump store1
    clr    tmrc.4               ; set timer control reg. stop
                                ; counting
    snz    acc.3         ;(12)  ; check acc.3=1 (acc.3="1" timer
                                ; time-out,acc.3="0"external INT)
    jmp    next                 ; if not 1 jump next
    jmp    top                  ; acc.3=1 jump top
delays proc                     ; delay subroutine
d1:                             ; d1
    sdz    acc                  ;
    jmp    d1                   ;
    ret                         ;
delays endp                     ;
    end                         ;
```

Program description

The program begins (1) by setting up the interrupt control. The external interrupt and timer/counter are enabled and the timer/counter is setup to be

an internal type. Section (2) sets Port A and B to be inputs and Port C to be outputs and clears the high and low order bytes. In (3) the control bits are initially set as follows BUSY=0, ACK=1, $\overline{OE}$=0, $\overline{WR}$=1 by placing these values on Port C. ACC is then cleared and the program waits for an interrupt to appear. Only when an interrupt occurs will the value of ACC change (4). After an interrupt occurs, the program will jump to the interrupt service routine, and the control lines on port C change to the following: BUSY=1, ACK=1, $\overline{OE}$=1, $\overline{WR}$=1. Pin 1 of the printer port $\overline{STROBE}$ is connected to the external interrupt input INT. So only when a $\overline{STROBE}$ occurs will the interrupt service routine be run. Before the next $\overline{STROBE}$ occurs it is necessary to check the condition of the BUSY signal. So all the data transfer must have been completed.

The Printer Port s $\overline{STROBE}$ signal indicates that there is data ready to be transmitted. In (5) Ports A and B are first setup as outputs and in part (6) the data is placed on these two ports, the low order byte on A and the high order byte on B. The address is then increased by one. At the same time the $\overline{STROBE}$ signal is setup. After the 74HC374 receives a clock signal the data on pins 2~9 of the printer port will appear on the output pins of the 74HC374. After this the program will activate the $\overline{WR}$ line (7) to write the data into the SRAM and after a time delay (8) will return to its original inactive value. After this the ACK line is activated (9) to confirm the data transfer and again after a short time delay it returns to its original inactive state. At this point the write cycle is complete. The BUSY signal is now returned to its inactive state (11) permitting further data to be written. When the $\overline{STROBE}$ signal is received the timer/counter starts counting (10).

At this time the ACC is checked to see if it is zero. Two different events will allow the program to jump out of the program loop (4), these are an external interrupt or an interrupt created by the timer/counter. These two interrupts will affect ACC in different ways. This can be detected by looking at ACC bit 3, as shown in (12). Whether it is an external interrupt or an interrupt due to the timer/counter, both indicate that the data transfer is complete. After this an $\overline{OE}$ signal will be generated indicating that the SRAM data is valid and available for use by the hardware. The next action is to return Ports A and B from their original condition as output ports to input ports (13). Care has to be taken here to avoid conflicts between the microcontroller address bus and the system bus, which is connected to the same lines. For this reason Ports A and B must be defined as inputs to place them in a high impedance state before the system takes control of the address lines.

Part V

Appendix

Appendix A

Reserved Words Used By Assembler

A

Registers

The following list describes the registers used by the assembler

Register Name	Memory Address
A	05H

Instruction Sets

Instruction		Description
ADC	A,[m]	add the data memory and carry to the accumulator
ADCM	A,[m]	add the accumulator and carry to the data memory
ADD	A,[m]	add the data memory to the accumulator
ADD	A,x	add immediate data to the accumulator
ADDM	A,[m]	add the accumulator to the data memory
AND	A,[m]	logical AND the accumulator with the data memory
AND	A,x	logical AND immediate data to accumulator
ANDM	A,[m]	logical AND the data memory with the accumulator
CALL	addr	subroutine call
CLR	[m]	clear the data memory
CLR	[m].i	clear a bit of the data memory

CLR WDT		clear the watch-dog timer
CLR WDT1		clear the watch dog timer (except 48050)
CLR WDT2		clear the watch dog timer (except 48050)
CPL	[m]	complement the data memory
CPLA	[m]	complement the data memory, store the result to the accumulator
DAA	[m]	decimal-adjust accumulator for addition (except 48050)
DEC	[m]	decrement the data memory
DECA	[m]	decrement the data memory, store the result to the accumulator
HALT		enter the power down mode
INC	[m]	increment the data memory
INCA	[m]	increment the data memory, store the result to the accumulator
JMP	addr	direct jump
MOV	A,[m]	move the data memory to the accumulator
MOV	A,x	move an immediate data to the accumulator
MOV	[m],A	move the accumulator to the data memory
NOP		no operation
OR	A,[m]	logical OR the accumulator with the data memory
OR	A,x	logical OR immediate data to the accumulator
ORM	A,[m]	logical OR the data memory with the accumulator
RET		return from the subroutine
RET	A,x	return and place immediate data in the accumulator
RETI		return from interrupt (except 48050)
RL	[m]	rotate data memory left
RLA	[m]	rotate data memory left, save the result to the accmulator
RLC	[m]	rotate data memory left with carry

RLCA	[m]	rotate data memory left with carry, save the result to the accumulator
RR	[m]	rotate data memory right
RRA	[m]	rotate data memory right, save the result to the accmulator
RRC	[m]	rotate data memory right with a carry
RRCA	[m]	rotate data memory right with carry, save the result to the accumulator
SBC	A,[m]	subtract the data memory and carry from the accumulator
SBCM	A,[m]	subtract the data memory and carry from the accumulator, save the result in the data memory
SDZ	[m]	skip if the decrement data memory is zero
SDZA	[m]	decrement the data memory contents, save the result to the accumulator, skip if the result is zero
SET	[m]	set the data memory
SET	[m].i	set a bit of the data memory
SIZ	[m]	skip if the increment data memory is zero
SIZA	[m]	increment te data memory-place the result in the accumulator, and skip if zero
SNZ	[m].i	skip if bit i of the data memory is not zero
SUB	A,[m]	subtract the data memory contents from the accumulator
SUB	A,x	subtract immediate data from the accumulator
SUBM	A,[m]	subtract the data memory contents from the accumulator and save the result to the data memory
SWAP	[m]	swap nibbles within the data memory (except 48050)
SWAPA	[m]	swap the data memory-place, save the reult in the accumulator (except 48050)
SZ	[m]	skip if the data memory is zero
SZ	[m].i	skip if bit i of the data memory is zero
SZA	[m]	move the data memory to the accumulator, skip if zero

TABRDC	[m]	move the ROM code (current page) to the TBLH and the data memory (except 48050)
TABRDL	[m]	move the ROM code (last page) to the TBLH and the data memory (except 48050)
XOR	A,[m]	logical XOR the accumulator with the data memory
XOR	A,x	logical XOR immediate data to the accumulator
XORM	A,[m]	logical XOR the accumulator with the data memory

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. HASM does not accept multiple symbol definitions.

A0012 Undefined symbol in EQU directive
HASM does not accept undefined symbols to the right of directive EQU, even for forward references.

A0013 Expression syntax error
Syntax error in expression.

A0014 HASM internal stack overflow
This error is due to HASM 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 definition
The name of the section is the same as previously defined section.
The section name must be unique in a source file.

A0021 DBIT can 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 parethensis 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 10 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.

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.

A4006 No .CHIP directive
No directive .CHIP in the source file.

A4007 Bad instruction format file
The instruction description file is incorrect.

A4008 HASM internal fatal error
HASM failure, please contact dealer.

A4009 Out of memory
Not enough memory for HASM 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 HLINK syntax, refer to chapter 10.

L1002 Object file filename.obj not found in pass1
The specified object file filename.obj is not found in HLINK 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 (HLINK)
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. Refer to chapter 10, section 10.2

L1005 Illegal command option 'option'
The specified option (option) in the command line is illegal
Refer to chapter 10, section 10.2 for legal options.

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.
Please refer to chapter 10, section 10.2 for correct syntax.

L1009 Illegal object file filename.obj
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'
HLINK has failed to close the specified object file (system error)
Contact dealer.

L1011 Record 'rec-name' check sum error
HLINK found a check sum error in the record 'rec-name' of the specified object file
Check if this object file is generated by Cross Assembler (HASM) or not.

L1012 Microcontroller information mismatch
file 'filename1.obj' and 'filename2.obj'
Two object files with different uC configurations during assembly
Ensure the same uC 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 Cross Library (HLIB).
Check if the library file (libname.lib) is in the working directory.

L1014 Cannot close the library file 'filename.lib'
HLINK has failed to close the specified file.
Contact dealer.

L1015 Library file 'libname.lib' not found
HLINK cannot re-open the specified library file libname.lib while processing the link work
Contact dealer.

L1016 Object file 'filename.obj' not found in pass2
The specified object file filename.obj not found in the HLINK pass2
Contact dealer.

L1017 Cannot write the checksum of record 'xx'H
HLINK fails to write check sum of record (xxH) to the output file
Contact dealer.

L1018 Cannot write data of record 'xx'H
HLINK 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'
HLINK 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'
HLINK 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'
HLINK 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'
HLINK failed to create the debug file debugname.tsk
Check the PC file system and working directory or contact dealer.

L1023 Cannot create the task file 'taskname.tsk'
HLINK 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'
HLINK fails to create the map file mapname.map
Check the PC file system and working directory or contact dealer.

L1025 Program code too large
The total size of program code is larger than the uC 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 μC 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
Refer to chapter 10, Cross Linker for correct syntax

L1028 Cannot close the batch file 'batch.bat'
HLINK failed to close the specified batch file
Contact dealer

L1031 Public symbols are duplicated
Public symbol 'sym1' in module 'mod-name1'
Public symbol 'sym1' in module 'mod-name2'
HLINK 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
HLINK fails to convert the local file index to the global file index
Contact dealer.

L1033 Internal error when obtaining the global index
HLINK failed to get the global file index
Contact dealer.

L1034 Illegal class type for section 'sec-name' in the file 'filename.obj'
HLINK 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 (HASM). Otherwise, contact dealer.

L1035 Internal error when section 'sec-name' of the file 'filename.obj' is located
HLINK failed to find the section (sec-name) while in section allocation.
Contact dealer.

L1036 No free memory for section 'sec-name' of the file 'filename.obj'
HLINK failed to find enough ROM or RAM space for the section (sec-name) of the file (filename.obj) while in absolute section allocation.
Check if the address and the length of the section (sec-name) of the input file (filename.obj) are outwith the ROM or RAM range. Modify the program or specify a correct starting address for this section.

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 (HASM).

L1038 Memory allocation failed for section 'sec-name' in the file 'filename.obj'
HLINK 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
HLINK internal error
Contact dealer.

L1040 Bank number exceeds 8
HLINK found a bank number larger than or equal to 8, maximum is 7
Modify the directive ROMBANK in the source program

L1041 Can't find SECDEF in BNKDEF
HLINK failed to find the SECDEF record for a bank member, internal error
Contact dealer

L1042 Failed to move the write pointer for task file
HLINK internal error
Contact dealer

L1043 Illegal Fixupp record in the file 'binary.obj'
HLINK internal error
Contact dealer.

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'
HLINK 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, HLINK does the allocation work as if this option has not been issued.

L3002 Specified address 'xxxx' for 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, HLINK does the allocation work as if this option has not been issued.

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. HLIB 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. HLIB will check when processing DELETE, REPLACE, EXTRACT operations

U0016 Not enough memory
The user system has not enough memory for HLIB

U0017 Bad object file
The file to be added to the library file has a bad object format. It may not be generated by HASM or a disk error

U0018 No public name in the specified module
qIf a symbol needs to be public, refer to chapter 8, 8.2.3 program directive for PUBLIC directive, and re-assemble the source file, then use HLIB to replace the new object file with the old module

U0019 Illegal operation

U0020 Fail to close a file

U0021 Check sum is incorrect
HLIB internal error

U0022 Fail to out record to the library file
HLIB internal error

U0023 Out checksum error
HLIB internal error

U0024 Fail to seek file
HLIB 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
C1023 ## occurs at border of replacement
C1024 Stringified macro argument is too long

C1025 Unknown internal macro
C1026 Unterminated string or char const
C1027 Undefined expression value
C1028 Bad ?: in **#if/#endif**
C1029 Unknown operator in **#if**
C1030 Bad number *number* in **#if/#elsif**
C1031 Empty character constant
C1032 Syntax error
C1033 Internal Error in **#if/#elsif**
C1034 String in **#if/#elsif**
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 t*ype* 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 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 type error in argument *number* in *identifier*; type is illegal
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
C2058 left operand of . has incompatible type *type*
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
C2066 expression too complicated
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 too many initializers
C2072 unclosed comment
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*
C2201 insufficient memory
C2202 read error
C2206 should specify ROM address

Warning Code

C3001 **#error** directive: *error*
C3002 **#line** specifies number out of range
C3003 Bad token produced by ##
C3004 Bad digit in number *number*
C3005 EOF inside comment
C3006 Wide char constant value undefined
C3007 Unknown preprocessor control *control*
C3008 Undefined escape in character constant
C3009 Multibyte character constant undefined
C3010 Character constant taken as not signed
C3011 C preprocessor internal error
C3012 Illegal option – *option*
C4001 empty declaration
C4002 empty input file
C4003 missing prototype
C4004 inconsistent linkage for *identifier* previously declared at *file_line_no*
C4005 more than 511 external identifiers
C4006 declaration of *identifier* does not match previous declaration at *file_line_no*
C4007 more than 32767 bytes in *type*
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
C4025 unsigned operand of unary -
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
C4040 overflow in octal 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

C5000 **#if** too deeply nested
C5001 Out of memory
C5002 Illegal -D or -U argument *argument*
C5003 Too many macro arguments
C5004 EOF in string or char constant
C5005 **#include** too deeply nested
C5006 Too many -I directives
C5007 Unable to open input file *filename*
C5008 Unable to open output file *filename*
C6001 function not supported yet

Warning Code

C3001 **#error** directive: *error*
C3002 **#line** specifies number out of range
C3003 Bad token produced by ##
C3004 Bad digit in number *number*
C3005 EOF inside comment
C3006 Wide char constant value undefined
C3007 Unknown preprocessor control *control*
C3008 Undefined escape in character constant
C3009 Multibyte character constant undefined
C3010 Character constant taken as not signed
C3011 C preprocessor internal error
C3012 Illegal option – *option*
C4001 empty declaration
C4002 empty input file
C4003 missing prototype
C4004 inconsistent linkage for *identifier* previously declared at *file_line_no*
C4005 more than 511 external identifiers
C4006 declaration of *identifier* does not match previous declaration at *file_line_no*
C4007 more than 32767 bytes in *type*
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
C4025 unsigned operand of unary -
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
C4040 overflow in octal 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

C5000 **#if** too deeply nested
C5001 Out of memory
C5002 Illegal -D or -U argument *argument*
C5003 Too many macro arguments
C5004 EOF in string or char constant
C5005 **#include** too deeply nested
C5006 Too many -I directives
C5007 Unable to open input file *filename*
C5008 Unable to open output file *filename*
C6001 function not supported yet