

GOLDFISH SOFTWARE

DEVELOPER'S GUIDE (Version 1.0)

Revision History

Revision	1.1.1.1 Date	Name	Revision contents
1.0	3/28/2001	Rui Tang	First revision

1 Overview

1.1 Product Introduction

Goldfish software package is developed for Goldfish product, which is a multifunction handheld device. The main functions of the product include Mp3 player, digital void recorder (DVR), digital still camera (DSC), fm tuner, alarm/calendar, and personal information manager (PIM). The hardware components of the product include PalmMicro's PALM-I chip and some peripheral devices. PALM-I's unique architecture enables it to be a very cost-effective stand-alone device for hand-held devices. A full configuration of PALM-I is given in figure 1 below. However, product vendors can customize the configuration and design their own product.

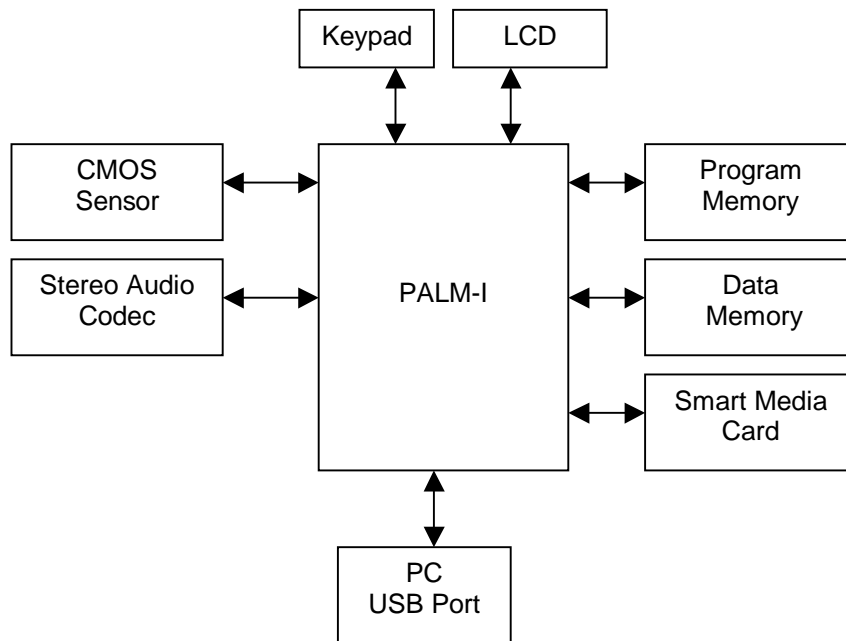


Figure 1: PA-1680 Configuration Diagram

1.2 Software overview

The goldfish product functions as a standalone device when used as a handheld device. It can also function as a PC peripheral device when connected to a host PC via a USB cable. This enables the user to exchange files and information between the host PC and the device. Goldfish software package is divided between PC/host software part and device firmware part. Figure 2 gives a brief description of relations between goldfish software modules.

1.2.1 PC/host Software Modules

PC/host software modules are compiled and linked by Microsoft Visual C++ 6.0. Developers should install Microsoft Visual C++ 6.0 or above edition to compile and link the codes. As USB drivers are needed in the projects, to install Windows 98/2000 DDK is also a must for compiling and linking.

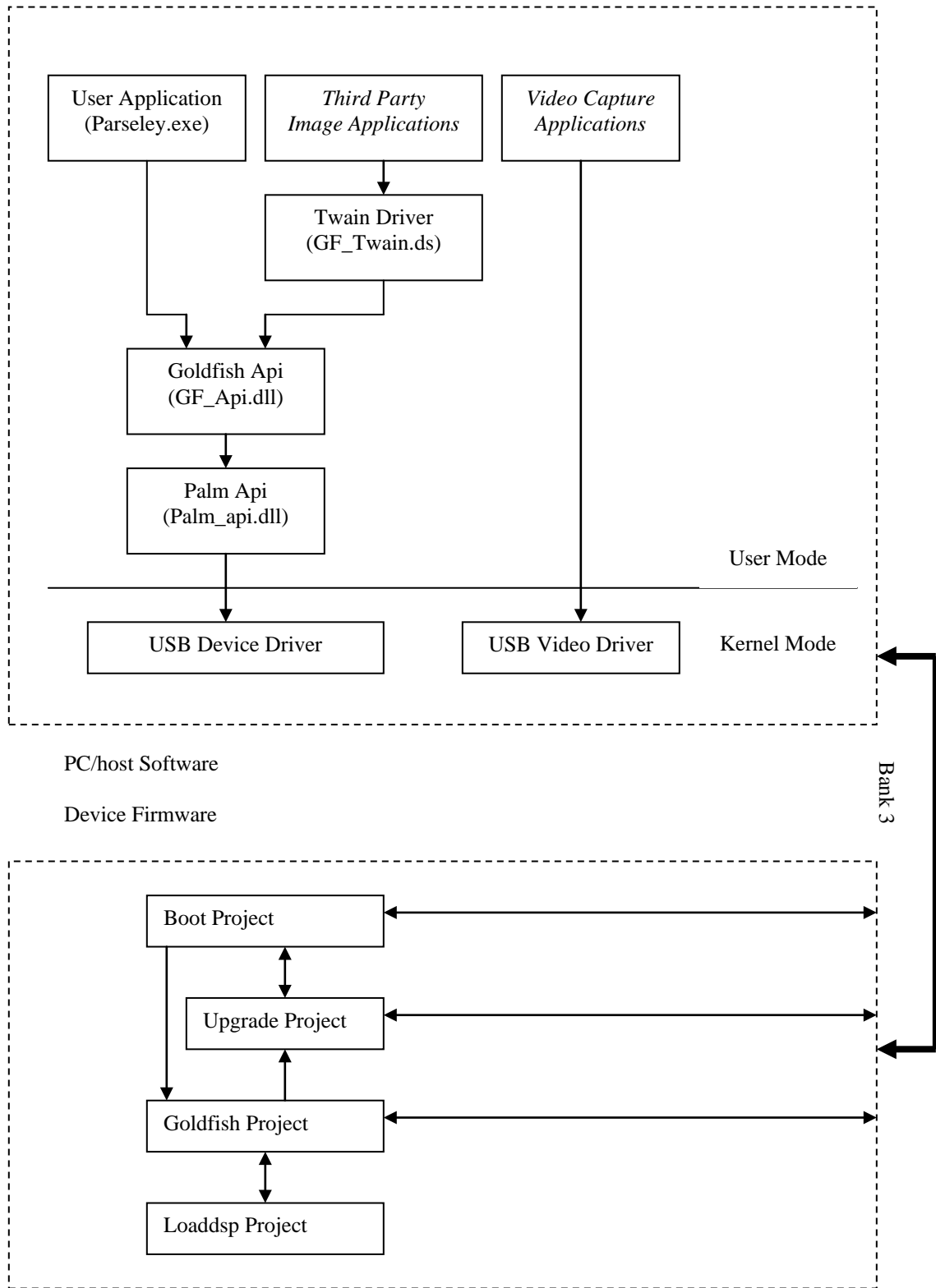


Figure 2: Software Overview

As seen in figure 2, PC/host software

modules include kernel mode USB

drivers and user mode APIs and applications. These software modules communicate with the goldfish device firmware when the device is connected via a usb cable. The host PC automatically loads one of the usb drivers (device driver and video driver) when the device is hot plugged. The choose between the two usb drivers is decided by the usb device descriptors reported from the device. To the user point of view, this option lies in the two menu items, "Camera->Video Mode->On/Off". That is to say, if the user selects the video-on mode from the menu, the host PC will load usb video driver when the device is connected. Otherwise the usb device driver will be loaded. The default setting is in video-off mode.

User mode software modules include 2 layered Api sets, the user application and the goldfish twain driver. This two-layered API organization gives other product vendors the flexibility to customize their own APIs and applications while at the same time saves their developing efforts. The lower layered Palm_Api.dll provides the API set for basic communication functions between the device and host. These functions include read/write register values and data blocks to/from the device. The higher layered GF_Api.dll wraps these basic functions into high level APIs, which enables the application to perform management like exchanging files and loading/saving device settings.

The goldfish software package also includes a demo application called *Parsley*. This windows application provides a user interface to communicate with the device.

GF_Twain.ds is the twain data source driver for goldfish product. Twain drivers enables third party image software such as PhotoShop to acquire images from the device by a standardized protocol. This protocol is specified by the TWAIN Working Group. Related documentation could be found at www.twain.org.

1.2.2 Device Firmware

Device firmware modules include DSP codes and 8051 controller codes. As in the PALM-I chip architecture, the dsp core serves as a slave processor to the 8051 controller, dsp codes are converted as binary images and are linked into 8051 projects as constant arrays in header files. They are loaded into the dsp core by the controller when needed.

This developer guide focuses only in the 8051 controller codes since they are responsible for organizing the device and communicate with the host. To compile the controller code, developers should install Franklin's Proview32, which provides an integrated developing environment (IDE) for 8051 projects.

As can be seen in figure 2, goldfish software package includes 4 8051 projects, which are boot project, goldfish project, upgrade project and loaddsp project. A brief description of the functions of these four projects the and switching relations among them is given in the following list:

- Boot project. This project resides in the first page (64K bytes) of program flash. When the device is just power up or reset, the controller will begin to run in this page. Boot code is designed to guarantee safely upgrade to newer versions of firmware. Once programmed into the first page of program flash, it should not be modified thereafter, even when the user wants to upgrade new firmware to the program flash. When the device is boot up in the boot project, it first checks whether the firmware is integral. There is a flag in the program flash (four characters located at 0x8000 of the first page) to indicate the firmware integrity. This flag is erased every time an upgrade operation is performed, and is set again after the upgrade successfully ended. By this way, if an upgrade operation is failed in process, the device will know it by checking this flag after reset. After check the firmware integrity, the device will also check whether the ERASE key is pressed at the boot time. Users press the ERASE at boot time to force an upgrade operation. This is necessary when the firmware could not bring up the device successfully. For example, when the user upgraded a corrupted firmware to the program flash, although the upgrade process was successfully ended, the new firmware upgraded will cause the controller running out of order after the device leaves the boot stage. The device will switch to Goldfish project if neither of the conditions is true (not integral firmware or ERASE key is pressed). Otherwise it will stays in the boot project until it is connected to the PC host via usb for a upgrade operation.
- Goldfish project. This project covers the next 2 pages following the boot project, i.e. the 2nd and the 3rd page. The 4th page is also reserved for goldfish project but not used at this time. This is main function project of the device. All the applications run in this project. At the time the device first enters the goldfish project (after leaving the boot project), it switches to loaddsp project to load dsp codes to SDRAM. After doing that, it will switch back to goldfish project.

- Loaddsp project. This project covers the 5th and the 6th program page and is only executed once after the system is power up. The main function of this project is to load dsp codes to SDRAM so that these codes could be dma transferred to dsp core when needed. After the device first enters the goldfish project (after leaving the boot project), it switches to loaddsp project to load dsp codes to SDRAM. After doing that, it will switch back to goldfish project.
- Upgrade project. PALM-I chip has a special mode called upgrade mode. In this mode, PALM-I fetches instruction from the on-chip 4K SRAM instead of the external program flash. This 4K SRAM is used as external data memory for the controller (though it is on chip, it is still external to the controller point of view) when the program is running on program flash. While in upgrade mode, it serves as the code memory for the controller. The main purpose of introducing this mode is for upgrading firmware and saving power. When running this mode, program flash is mapped into 8051's external memory data space so that could be written with new contents. It also can be used in standby mode when most of the peripheral devices could be shut down.

2 Multi-page Programming

2.1 8051 Program Memory Space and Bank Switching

The 8051 controller directly supports a maximum of 64K bytes of code space. However, 8051 linkers like Franklin Proview32 or Keil allows 8051 programs to be created that are larger than 64K bytes by using a technique known as code banking or bank switching. Bank switching involves extra hardware implementation to select a number of banks residing on a common bank. Figure 3 gives an example of 4-bank switching with common bank size of 16K bytes.

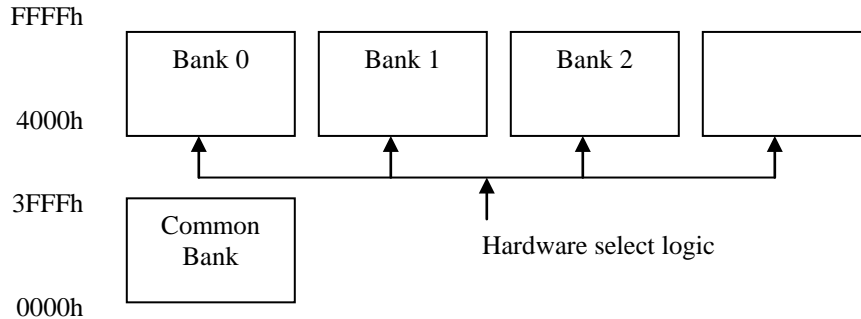


Figure 3: Bank Switching

The basic idea of bank switching is to switch code between banks seamlessly by using a common bank. For example, when the code being executed in bank 0 need to call a function that resides on bank 2, since it is unable to jump directly to the targeted function, it will first jump back to the common bank. In the common bank, it saves the current bank number and current address for later return. Then it commands the hardware to select the new bank and jumps from the common bank to the new bank. When returning from the new bank, old bank number and returning address are restored. Once again, it first jumps to the common bank and command the hardware logic to select the correct bank for executing. The linker takes care of all the process, i.e., it is transparent to user source code. There is no necessary to modify the code for bank switching. The only thing need to do is to specify in project setting the bank to reside on for each module.

2.2 Bank switching with PALM-I

PALM-I has an 8051 core as a controller. Goldfish products use program flash to store controller codes. As goldfish firmware takes 6 64K bytes program pages, 512K bytes or bigger size program flash is needed to accommodate all the controller codes.

A register named PROG_PAGE is used to select the program page for the controller. According to the content of this register, different program flash spaces are mapped into the 8051 64K code memory space. For example, when PROG_PAGE is 3, then the 4th page of the program flash (from 0x30000 ~ 0x3ffff) is the current controller code memory space.

Writing to PROG_PAGE causes either a reset to the controller or an external interrupt 0 to the controller. However, neither of them is desirable for a seamless bank switching. Therefore, to use the bank-switching feature, the user must be sure that not to reset the controller and the external interrupt 0 is disabled. Bit 4 of register PROG_DATA_MODE decides which feature is selected. When this bit is set, writing to PROG_PAGE causes external interrupt 0 to the controller. When this bit is reset, writing to PROG_PAGE causes the controller to be reset. A macro to enable bank switching is as follows:

```
#define ChipEnableBankSwitch() PROG_DATA_MODE |= 0x10
```

This macro is invoked in the loadmain.c of the loaddsp project, which is a 2-page project. For the goldfish project, which also takes 2 pages, similar assembly code is used in the file of startup.a51:

```
MOV     DPTR, #PROG_DATA_MODE
MOVX   A, @DPTR
ORL    A, #010H
MOVX   @DPTR, A
```

2.3 Bank Switching Settings in Proview32 Project

Let us take the goldfish as an example to illustrate it. The first thing is to enable bank switching in project setting. Check the “Use Bank switching” option shown in the “Options”->”Project”->”L51”->”Linker” dialog. Figure 4 shows the selection.

The second thing to do is to decide the common bank size and the locations of the software modules. This is set in the “Options”->”Project”->”L51”->”Linker” dialog.

As illustrated figure 5, goldfish project takes 2 banks (pages). The common banks size is 16K bytes, starting from 0x0000 to 0x4000.

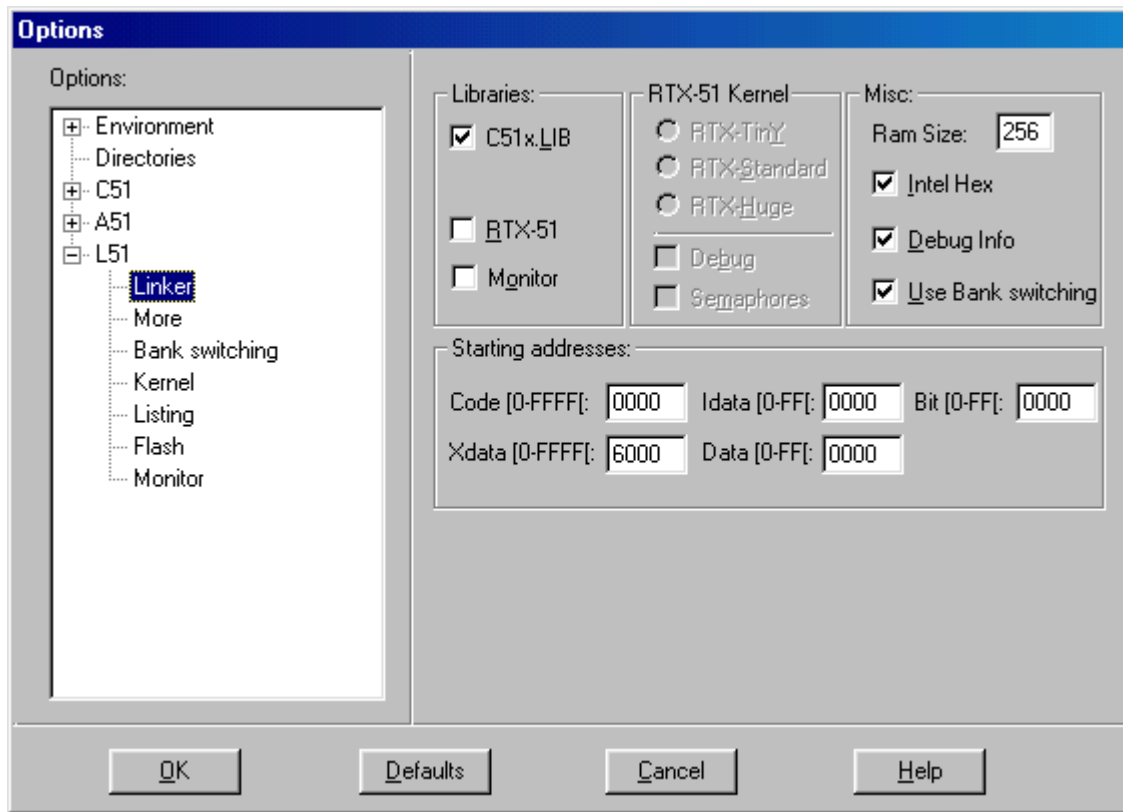


Figure 4: Enable Bank Switching Option

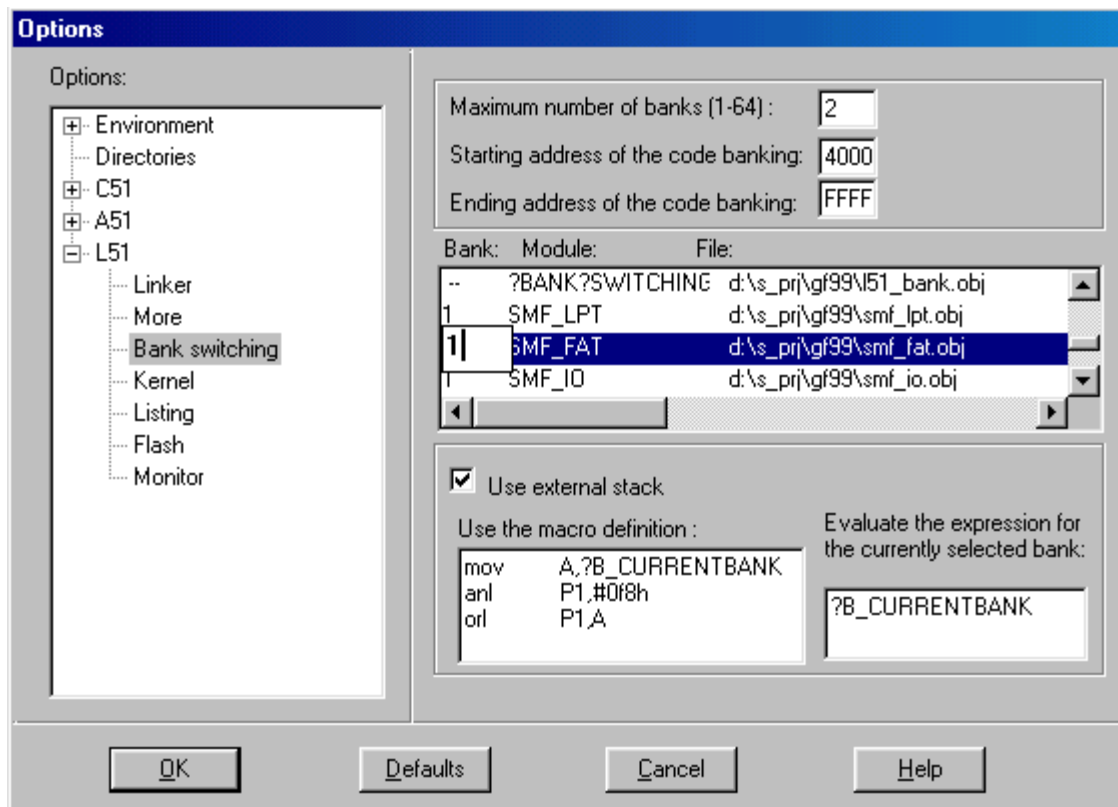


Figure 5: Bank Switching Settings of Goldfish Project

Some principles apply when locating the software modules:

- Locating software modules that impact or rely greatly on system performance to the common bank. Since modules located in the common bank do not suffer from the overhead for bank switching, it is helpful to better performance to do so. However, the common bank is a limited resource because it is needed in each 64K-byte program page. Therefore, if the common bank size is too big, more program spaces are wasted. Developers should make their own decision on this trade-off. Goldfish project set common bank size to 16K bytes and puts fat related software modules in the common bank.
- Locating software modules that are more likely to be executed together in a same bank. This also helps to reduce the overhead for bank switching. For example, goldfish project puts all the smartmedia related software modules into bank 1.

After setting the bank switch settings, build the project. Proview32 will generate a file named *l51_bank.a51* and a file named *bankswitch.mac* under the project folder. However, these two files cannot be directly used in the IDE environment because of some inconsistent linking options. I believe that it is a bug left by Franklin Company. So I modified the *l51_bank.51* file and replaced the *bankswitch.mac*, which is included by the *l51_bank.a51*, with a new file named *bankswitch.inc*.

The content of *bankswitch.inc* is really a simple, it just defines a macro to switch between pages like the following:

```

PUSH    DPL
PUSH    DPH
MOV     A, ?B_CURRENTBANK
INC     A
MOV     DPTR, #08100H
MOVB   @DPTR, A
MOVB   @DPTR, A
POP     DPH
    
```

POP DPL

Notice that the goldfish project runs at the 2nd and 3rd program page. So to set current to bank 0 of the project, (?B_CURRENTBANK = 0), 1 should be written to 0x8100, which is the address of register PROG_PAGE as mentioned in the previous section. Therefore, there is an instruction to increment the accumulator (*INC A*) before setting the current bank. Developers creating their own bank switching project should follow the steps below:

- After the linker generate the l51_bank.a51 and bankswitch.mac, replace the l51_bank.a51 with the one in the goldfish project and remove bankswitch.mac.
- Write a new bankswitch.inc referring the one I used in goldfish project. There may be some difference since your project may start at another page. For example, if your project starts from the 1st page, the instruction of *INC A* should not be used. While if your project starts from, for example, the 5th page, you should replace the *INC A* with an instruction of *ADD A, #04H*.

2.4 Switching Between Projects

Bank switching provides the way to switch program within a same project, therefore enabling to program a large project with multiple pages. However, it is too difficult to write all the firmware into one project. Goldfish firmware includes 4 proview32 projects: *boot*, *goldfish*, *loaddsp* and *upgrade*. The following table summarizes the location information of them. Notice that the upgrade project does not reside on program flash but on the 4K SRAM in the PALM-I chip.

The method to switch to a new project is to write to PROG_PAGE, which causes the controller to be reset and run on a new program page. Column 3 of the table is the value to be written into register PROG_PAGE when switching to the project.

	Location	PROG_PAGE
Boot Project	Program flash: 0x00000 ~ 0x0FFFF	0x00
Goldfish Project	Program flash: 0x10000 ~ 0x2FFFF	0x01
Loaddsp Project	Program flash: 0x40000 ~ 0x5FFFF	0x04
Upgrade Project	On-chip 4K SRAM	0x80

Table 1: Firmware Projects

Since the switching between 2 projects is implemented by resetting the controller, all the 8051 registers are also reset. Therefore it is necessary to save the context before switching and restore it after the device runs in the new project. A software module named ModeMgr is used manage the switching context. Developers can find a copy of *modemgr.a51* is linked in each of the 4 projects. The header file *modemgr.h* is put under the include directory. The following 3 functions are defined in *modemgr.h*:

```
bit p_GetSwitchContext();
void p_SetSwitchContext(uchar nMode, uchar nNextMode, uchar nTask);
void p_SwitchProgram(c_uchar * pCode);
```

An example of using these functions is like the following as used in *psetting.c* of Goldfish project:

```
...
p_SetSwitchContext(iMode, iNextMode, iTTask);
p_SwitchProgram(prog_51);
...
```

The function *p_SwitchProgram* will cause the program to switch to a new project according to the *iMode*, *iNextMode* and *iTask* set in the former call of *p_SetSwitchContext*. The parameter passed in *p_SwitchProgram*, *prog_51*, is the name of a constant array, which is defined in *upgrade.h*. This array contains the binary code of upgrade project. *p_SwitchProgram* will copy the array of *prog_51* to the on-chip 4K SRAM buffer if *iMode* is set to

MODE_STANDBY, i.e., the program is going to be switched to run in standby/upgrade mode. If iMode is not MODE_STANDBY, the prog_51 passed in is ignored.

3 GFTool User Guide

3.1 Introduction

GFTool is a windows application developed for ease of goldfish firmware development. This toolbox collects conversions between hex, binary, and header files. Figure 6 shows an outlook of the application.

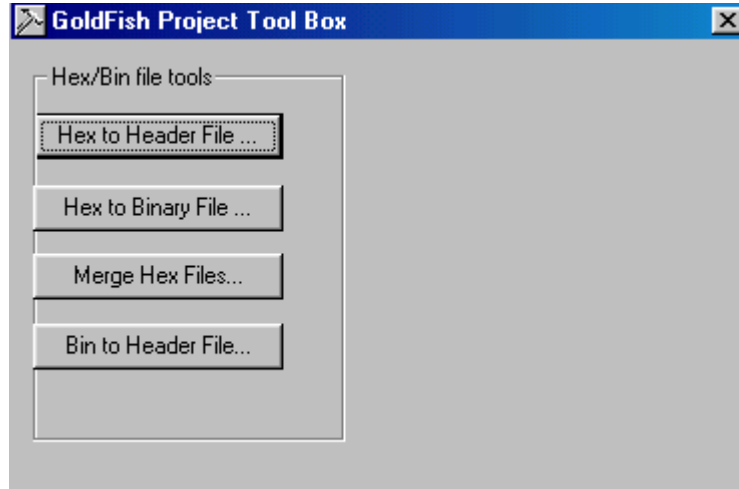


Figure 6: Goldfish Project Tool Box

3.2 Hex to Header Files Conversion

The first button in the toolbox, as its name suggests, converts a hex file to a c language header file.

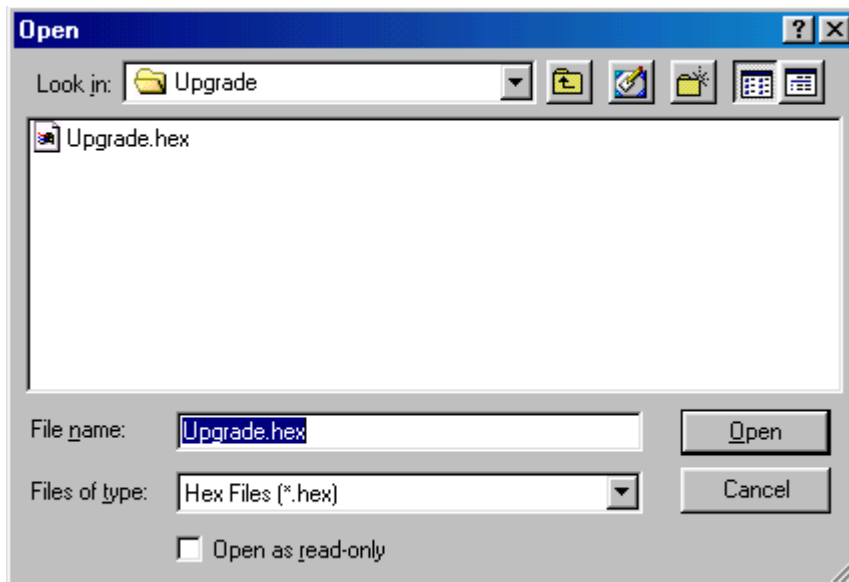


Figure 7: Select Hex File for Conversion to Header File

As described in the previous chapter, the upgrade project treated as a constant binary array and included as a c language header files in other projects (goldfish and boot project). As the linker (proview32) will only generate an intel hex format file, we have do the conversion by ourself. After select this function, user is first prompted to locate the hex file to be converted, as illustrated in figure 7.

The user is then prompted to input the constant array name used in the header file, as illustrated in the figure 8. As in goldfish project, the name of prog_51 is used.

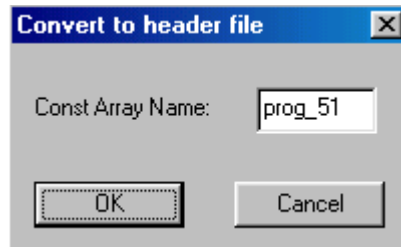


Figure 8: Specify Constant Array Name in Header File

Then the user is prompted for the header file name, as illustrated in figure 9. As in goldfish project, upgrade.h is used for the header file name.

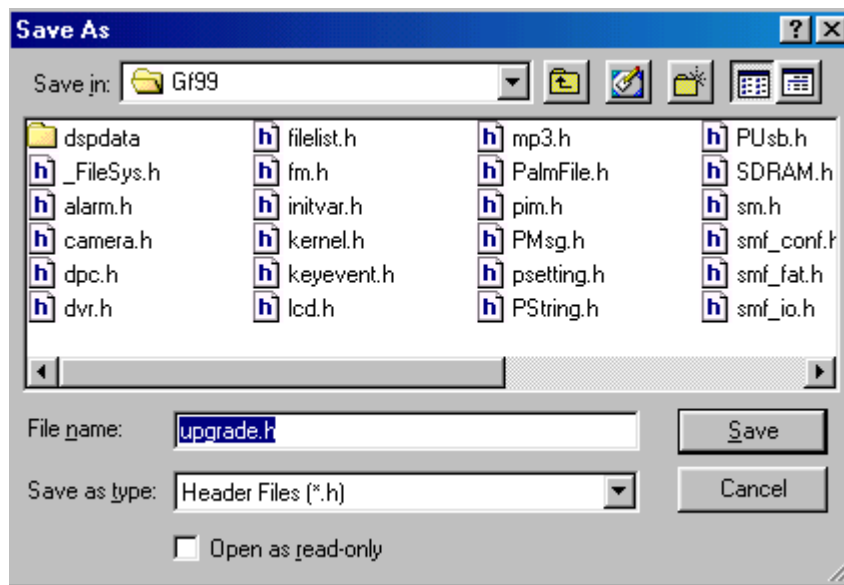


Figure 9: Specify Header File Name

3.3 Hex to Binary Files Conversoin

The second button in the toolbox, as its name suggests, converts a hex file to a binary.

3.4 Merge Hex Files into One Binary File

The third button in the toolbox, as its name suggest, merge multiple hex files into one binary file.

This function is used to generate a single binary file for upgrade firmware. The application *Parseley* provides a way to upgrade new firmware into the device. This file contains the new firmware should be in binary format or intel hex format. As the program to be upgraded include goldfish project and loaddsp project (boot project should be kept untouched, while upgrade project is linked into goldfish project). It is convenient and necessary (as we only allow to

upgrade one file to a predefined address) to convert linker generated goldfish.hex and loaddsp.hex into one binary file.

The user is first prompted to select the first hex, as illustrated below in figure 10. In this example, goldfish.exe is selected because it resides in front of the loaddsp project

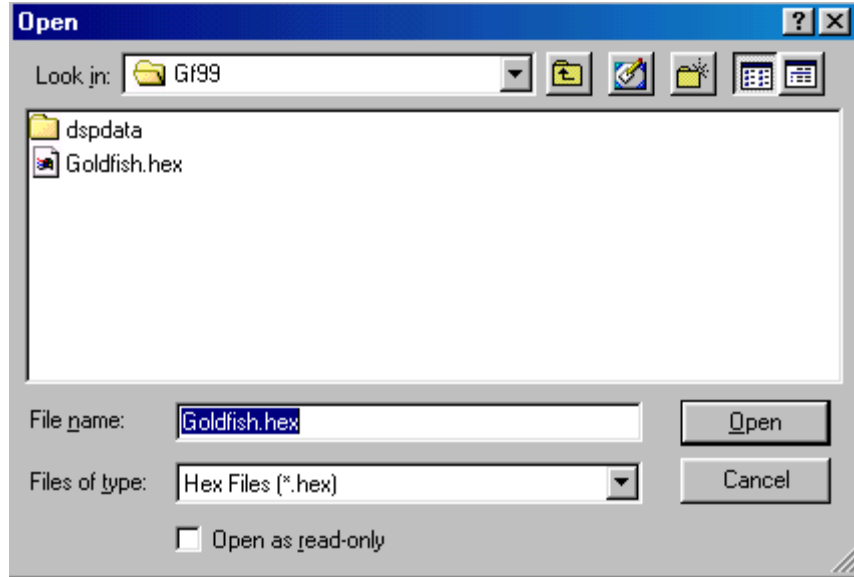


Figure 10: Select Hex File for Merging to one Binary File

The next step prompts the user to select more hex files to be merged together. Since we only have one more file, loaddsp.hex, to merge, and it (starting from program flash address of 0x40000) is 3 64K-byte pages behind the beginning of goldfish program (starting from program flash address of 0x10000), the user needs to locate the loaddsp.hex, input 0x30000 at the “starting address” edit box, then press “Done” button.

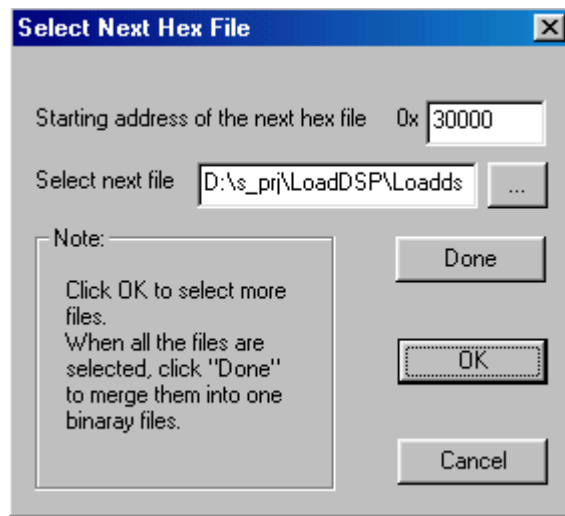


Figure 11: Specify Next Hex File and Its Position

3.5 Binary to Header Files Conversion

The fourth button in the toolbox, as its name suggests, converts a binary file to a c language header file.

4 Debugger

4.1 Introduction to Debugger Package

For ease of development, we also provide a debugging platform using serial port. The debugger package is different with the goldfish software package introduced in previous chapters. Unlike the goldfish software package uses usb cable to connect the device and the host PC, debugger uses the RS232 serial port instead.

Debugger software package is also divided between PC host software and device firmware. PC software includes a debugger application PalmDbg.exe and a dynamic link library GF_Api.dll. Although the dll name is the same with the one used in goldfish software packet, they are in fact not the same and not exchangeable. The reason that they share a same name is that the one used for debugger is in fact the earlier version of the one used in goldfish software package. Earlier goldfish software was developed with serial communication support. However, as it evolved to usb communication, the old GF_Api.dll are not suitable any more and is only kept for debugger use.

The firmware in the debugger package is a proview32 project, which is developed with debugger supporting feature. Generally speaking, it includes the serial communication protocol, the monitor module, and the code to be debugged. As the debugger is developed for debugging small projects, larger projects that have to use bank-switching feature cannot use the debugger function. The common method to use the debugger is to write your own code with main() function and add them to the debugger enabled project.

The currently release of the debugger packet include such a demo proview32 project named uda. In this project, a file named camera.c is the code for debug. Developers should replace this file with their own code to debug.

4.2 Usage of Debugger Package

Follow the following steps when to use the debugger package to debug your code.

- Copy PalmDbg.exe and GF_Api.dll to the debugger PC.
- Replace camera.c in the firmware project with your code.
- Build the firmware project, program the firmware code to the program flash.
- Launch PalmDbg.exe and open the project to be debugged from the File->Open Project menu.
- Config the serial port settings of PalmDbg. Select menu Debug->Start. The “Config Start Option” dialog with 3 pages will show up. Choose the debugger option in the “Mode” Page, then switch to “Debugger” page, select the COM port you are using and set the baud rate to 14400. After configuration, press the “OK” button, PalmDbg now enters debugging state, waiting debugging information from the serial port. Once the serial port is configured after the first time using PalmDbg, user can directly enter the debugging state by clicking the “Start” button on the toolbar.
- Power up the connected device. The device will automatically enter the monitored state and exchange debugging information with the debugger PC. User will see now that the PalmDbg enters to the halted state, with the source line where current program pointer points becomes green.
- Use the debug functions to debug your code. These functions include: set/remove breakpoints, view/modify memory, step/run programs, etc.

4.3 Debugging Functions

4.3.1 Set/Remove Breakpoints

In fact, the first line of the main() function is the default breakpoint in the program. When the device is power up or just been reset, PalmDbg will command the device to stop at this breakpoint. When the program is stopped at a breakpoint, we say that the device is *halted*. Only when halted, device is then able to exchange debugging information with PalmDbg. The following picture shows when the device is just powered up and halted at the first line of main() function.

The user can Set/Remove breakpoints at desired place using the menu command Edit->Toggle Breakpoint or by a shortcut key F9. Move the cursor to the source line and press F9 will toggle the breakpoint in the line. When breakpoint is set at a line, the line will be turned into red to indicate that it has a breakpoint set there.

As the debugging function is provide both by software and hardware debugger board, some limitations apply to set/remove breakpoint.

- Only 12 breakpoints are allowed to be set at one time. When reaching this limit, the user should remove some unused breakpoints and then allowed to set more breakpoint.
- Do not set/remove breakpoint at the current halted program pointer. This action will cause the debugger hardware out of logic. For example, when the device is halted at one line, either you stepped there or by setting a breakpoint and run there, do not try to add/remove a breakpoint there.
- Try to avoid setting a breakpoint in a loop. This will cause the device to be halted at the place each time the loop continues.

4.3.2 Step in Program

Select the step command, either by selecting menu command Debug->Step Into or by pressing short-cut key F11 to step in program

4.3.3 View/Modify memory and variables

PalmDbg allows the user to view/modify different parts of hardware memory, including: controller special function registers, controller internal data memory, PALM-I controller buffer, PALM-I dsp memory, and SDRAMs connected to PALM-I.

4.3.3.1 View/Modify Controller Special Functions Registers

Use the register dialog to view controller registers. This dialog is initially prompted when starting a debugging session. User could close the dialog and reopen it by selecting menu command View->Debug Windows->Registers. Some of the registers are modifiable by user and some are not. Those that can be modified have an editable edit box. To modify the values, edit the value and press the “Modify” button. To reload the values from the device, press the “Refresh” button.

4.3.3.2 View/Modify Controller Internal Memory

Controller internal memory includes *data* and *idata* memory part. The first 128 bytes of the internal memory is direct addressable (it is also could be indirect addressed), and is called data memory, while the following 128 bytes can only be indirect addressed, therefore called idata memory. Select the menu command View->Debug Windows->Memory->Data and View->Debug Windows->Memory->Idata to view them respectively. To modify them, edit in the memory view window, then click the right mouse button, select the “Write Block to Device” command from the pop-up menu.

4.3.3.3 View/Modify PALM-I Controller Buffer

PALM-I has 4.5K on-chip SRAM which can be used as controller external memory. And this 4.5K SRAM are divided into 2 parts as controller buffer 1 and controller buffer 2. Controller buffer 1 has 4K bytes and is mapped into controller external buffer 0x6000 ~ 0x7000. Controller buffer 2 has only 0.5K bytes and is mapped into controller external buffer 0x7000 ~ 0x7200. Contrller buffer 2 is able to do dma transfers with the SDRAM, controller buffer 2 cannot.

To view those 2 controller buffers, select the menu command View->Debug Windows->Memory->Controller Buffer1 and View->Debug Windows->Memory->Controller Buffer 2. Unlike controller internal buffers, the content of these 2 controller buffers are not refreshed after the device is halted. This is because the time to reload the contents of these 4.5K bytes buffer from device is much longer than that of the 256 bytes internal buffer. Therefore, if the user wants to see the updated controller buffer content, he has to press the “Refresh Controller Buffer” button on the toolbar.

4.3.3.4 View/Modify PALM-I Dsp Memory

The method to view/modify PALM-I dsp memories is similar to view/modify PALM-I controller buffers. Select the menu commands like View->Debug Windows->Memory->DMC to view/modify each part of the dsp memory.

4.3.3.5 View/Modify a Variable

To view/Modify a variable, select the menu command View->Debug Windows->Variables, a docking windows will be docked to the bottom of the application. Single click on the “Click here to add new watch” cell, the cell becomes editable, input the variables name, then the address and the contents (four bytes after the address) of the variable will then be displayed.

To remove the watch, single click on the variables name, after the cell becomes editable, clear the name and press enter, the watch is then removed.