ZFx86™

System-on-a-Chip

Integrated Development System

Quick Start Guide

November 27, 2001

THIS DOCUMENT AND THE INFORMATION CONTAINED THEREIN IS PROVIDED “AS-IS” AND WITHOUT A WARRANTY OF ANY KIND. YOU, THE USER, ACCEPT FULL RESPONSIBILITY FOR PROPER USE OF THE MATERIAL. ZF LINUX DEVICES, INC. MAKES NO REPRESENTATIONS OR WARRANTIES THAT THIS USER’S MANUAL OR THE INFORMATION CONTAINED THERE-IN IS ERROR FREE OR THAT THE USE THEREOF WILL NOT INFRINGE ANY PATENTS, COPYRIGHT OR TRADEMARKS OF THIRD PARTIES. ZF MICRO DEVICES, INC. EXPLICITLY ASSUMES NO LIABILITY FOR ANY DAMAGES WHATSOEVER RELATING TO ITS USE.

LIFE SUPPORT POLICY:

ZF MICRO DEVICES' PRODUCTS ARE NOT AUTHORIZED FOR USE AS CRITICAL COMPONENTS IN LIFE SUPPORT DEVICES OR SYSTEMS WITHOUT THE EXPRESS WRITTEN APPROVAL OF THE PRESIDENT AND GENERAL COUNSEL OF ZF MICRO DEVICES, INC.

As used herein:

1. Life support devices or systems are devices or systems which, (a) are intended for surgical implant into the body, or (b) support or sustain life, and whose failure to perform when properly used in accordance with instructions for use provided in the labeling, can be reasonably expected to result in a significant injury to the user.

2. A critical component is any component of a life support device or system whose failure to perform can be reasonably expected to cause the failure of the life support device or system, or to affect its safety or effectiveness.

(c)2001 ZF Micro Devices, Inc. All rights reserved.

ZFx86, FailSafe FailSafe Boot ROM, Z-tag ZF-Logic, InternetSafe, OEMmodule SCC, ZF SystemCard, ZF FlashDisk-SC, netDisplay, ZF 104Card, ZF SlotCard, and ZF Micro Devices logo are trademarks of ZF Micro Devices, Inc. Other brands and product names are trademarks of their respective owners.

1. Overview

The ZFx86 Integrated Development System is intended to provide a complete development environment that can be used to demonstrate how a target system based on the ZFx86 chip will perform. It is extremely flexible. The standard PC type interfaces are built into the chip (serial and parallel ports, etc.). Outboard features (video, networking, etc.) can be implemented by installing ISA or PCI interface cards. We supply a PCI Video Adapter, and a PCI Network Adapter, as part of the system. If peripheral adapters are selected that use the same "chips" that the intended product will eventually use, much of the final product's behavior can be characterized before you manufacture the PCB.

The ZFx86 Development System is essentially a standard personal computer, that uses the ZFx86 chip as its "chipset". It has a standard "ATX" form factor motherboard, so adding peripheral cards is an easy matter.

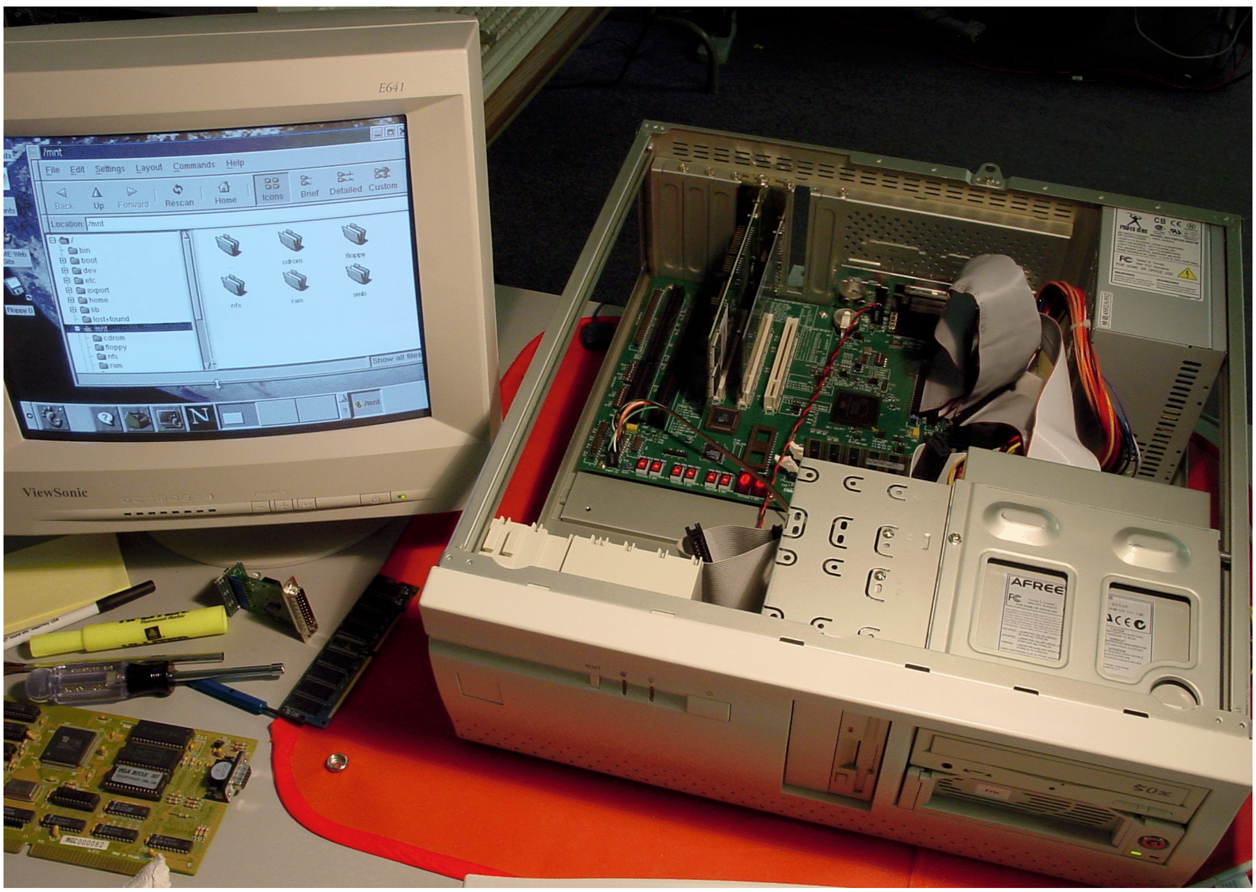


Figure 1-1 ZFx86 Integrated Development System

1.1. Rear Panel Connections

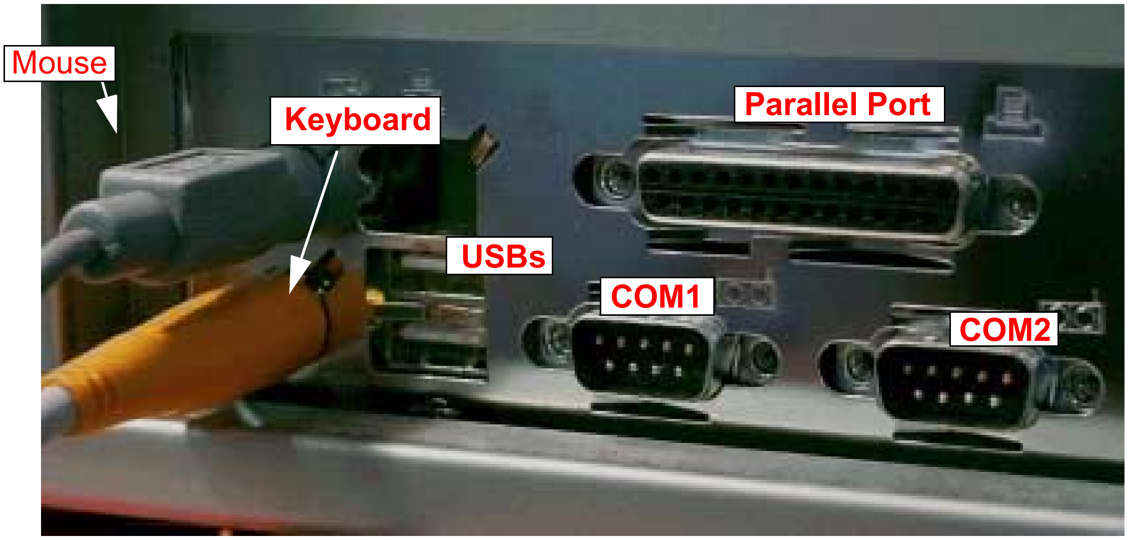


Figure 1-2 Rear Panel Connections

1.2. Supplied Peripheral Cards

The Integrated Development System comes with a Video Card, and a Network Card installed.

The Video Card is installed in PCI Slot # 1, and the Network Card is in PCI Slot #2. There is a jumper associated with full operation of PCI Slot #3 -- see 2.2.7. "JP6 - DMA or PCI Req/Grant" on page 14.

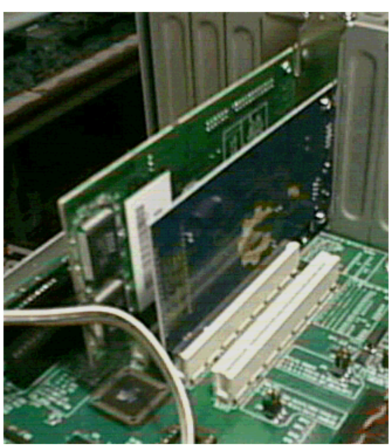


Figure 1-3 Video and Network Cards

1.3. Documents

You may view this document with the Acrobat 4.0 Reader under 32 Bit windows. That reader may be installed from the accompanying CD. If you also put the ZFx86 Data Book and the ZFx86 Training Book PDF files in the same directory, you may use the Acrobat 4 viewer to link to them.

When viewing the set of documents you may find it convenient to open them all at once (select all the PDF files and hit Enter).

1.3.1. The ZFx86 Training Book

The ZFx86 Training Book is a set of foils designed for a stand-up presentation on selected chapters of the Data Book. In some cases, the drawings are more elaborate than those in the Data Book, or there is a longer sequence of examples than might be practical to place in the data book.

1.3.2. Annotated Eval I Schematic

The ZFx86 Integrated Development System contains an Evaluation 1 Board. An annotated schematic of the Evaluation 1 Board is provided as a PDF file. There are pop-up notes on the schematic which open when you double-click on the yellow text icons.

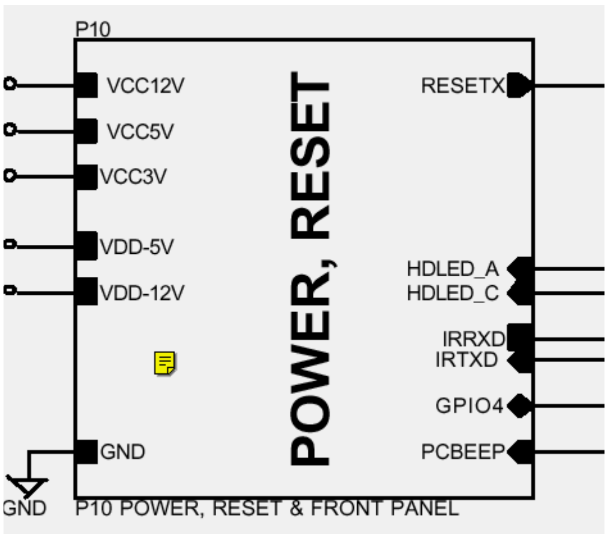


Figure 1-4 Part of Annotated Schematic

The schematic of the Evaluation 1 Board is provided as a design guide. The first page is an overview of the following pages. For example, the Power, Reset circuits are on P10 (page 10) of the schematic, and a top level Power and Reset block appears on P1. The blocks on P1 are also hypertext links to the correct page of the schematic.

You may do text searches for pins (by their Orcad names) on this PDF, in that it is machine generated out of the design program. For the Orcad pin names, see the Pinout Summary in the ZFx86 Data Book.

1.3.3. Annotated Eval I Silk Screen

There is also a PDF file which is a top view silk-screen of the Evaluation Board. That file has annotations which will help you to locate and understand the jumper settings. See 'Jumper Settings Details' on page 16.

1.3.4. Design Orcad and Pads Files

To facilitate your design, we have provided the design files the for Evaluation 1 Board. ZFx86 and IDS Support

Call ZF Micro Devices at 800-683-5943 toll free, or 650-940-4793. You may also send inquiries to info@zfmicro.com. Check our website for support, at www.zfmicro.com. The e-mail address for support is support@zfmicro.com.

You may check for the latest version of the data book at: www.zfmicro.com/download.html.

If the version there is later than the copy you have, you may download a new copy (skipping the registration) by going to www.zfmicro.com/databook.zip.

2. Hardware Setup

2.1. Power and Cabling

The ZFx86 Development System comes with all internal cables. The power supply will handle 220V at 50Hz, or 117 VAC at 60Hz. The system ships set for 117 VAC and ships with a standard USA cable.

2.2. Default Jumper Settings

Several switches and jumpers allow configuration of many of the features the ZFx86 chip provides. We have pre-configured the system to enable as many features as possible, and to configure them in the typical manner.

2.2.1. Clocking

Many different clocking schemes are achievable. We have shipped the system with the settings optimized for general purpose computing. The ZFx86 Development System is pre-configured with the following settings:

2.2.1.1. System Clock - 64MHz

This is configured by JP3 and S1. For details, see 'JP3 SYSCLK Source Jumper' on page 16. and 'S1Clock Multiplier DIP Switches' on page 16.

Figure 2-5 JP3 SYSCLK Source

2.2.1.2. CPU Speed - 128 MHz

This is configured by S3-switches 2 & 3 (bootstrap bits 16 & 17). See 'CPU Speed' on page 17.

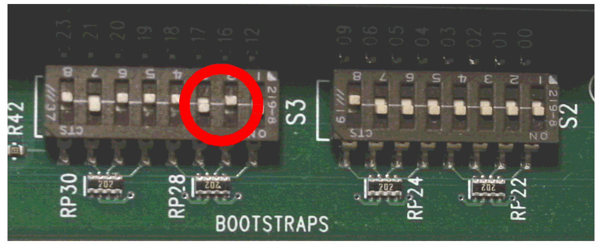


Figure 2-6 BootStrap Bits 16-17

2.2.1.3. PCI Clocking

We set the Front Side PCI BUS (on-board IDE) and USB to 32 MHz. We set the Back Side PCI BUS (peripheral slots) to 32 MHz, and the Back Side PCI Clock Source to Internal.

These are configured by S3-switches 4, 5, and 6 (bootstrap bits 18, 19, and 20).

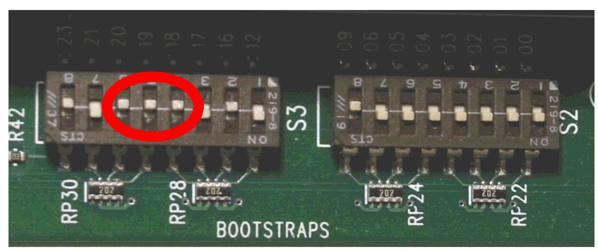


Figure 2-7 BootStrap Bits 18-20

2.2.1.4. 8254 PIT Clock (14MHz)

This is derived from separate oscillator. It is enabled using JP1. See 'JP1: 8254 PIT Clock (14MHz)' on page 17. You can see JP1 as the left-most jumper in the diagram below. The jumper is labeled CLK14M, which is logical signal mhz14_c and ORCAD signal CLK14MHz [AF16].

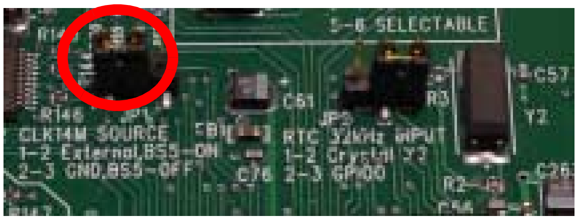


Figure 2-8 JP1 CLK14M Source

2.2.1.5. Real Time Clock (32KHz)

This is derived from separate oscillator and configured with JP9

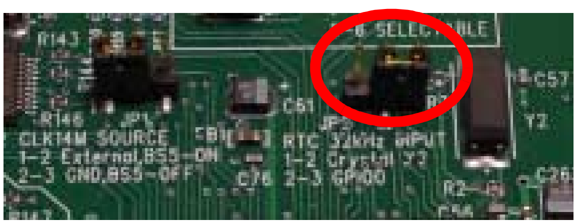


Figure 2-9 JP9 RTC 32KHz INPUT

2.2.2. Clock Source for PCI Bus

JP4 and S3-switch 6 (bootstrap bit 20) allow selecting the Clock Source for the PCI BUS. The system is shipped with this set to "Out". This has the ZFx86 providing the clock to the PCI slots.

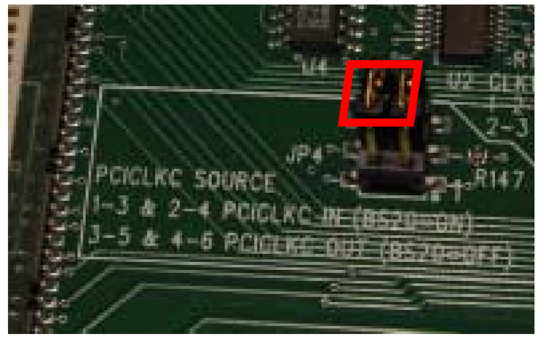


Figure 2-10 JP4 PCICLK Source

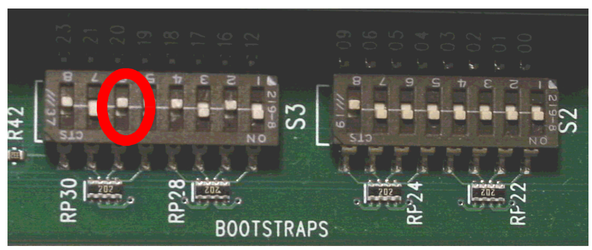


Figure 2-11 BootStrap Bit 20

2.2.3. Watchdog Timer Oscillator Source

This is set to Internal using JP2.

Figure 2-12 JP2 Watchdog OSC

2.2.4. JTAG Chain

Only the Xilinx chip is connected to the JTAG chain by default. This is configured with JP8

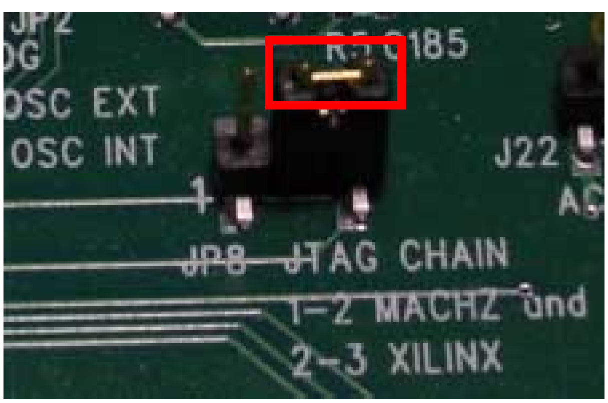


Figure 2-13 JP8 JTAG Chain

2.2.5. CMOS Battery

CMOS Battery - Normal. (moving JP5 allows clearing the CMOS contents)

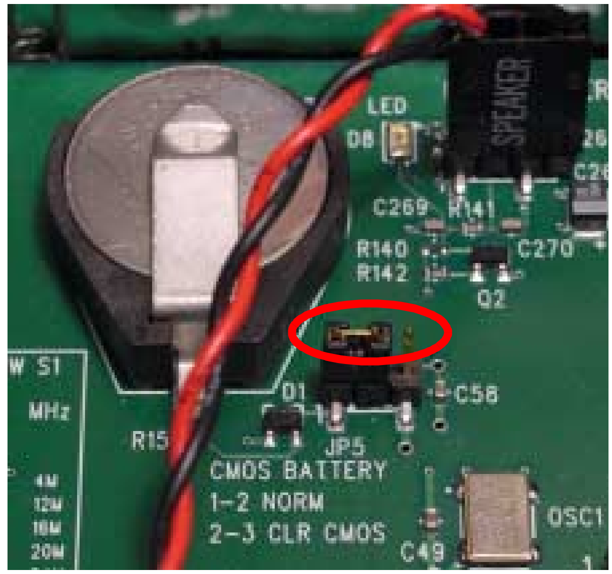


Figure 2-14 JP5 CMOS Battery

2.2.6. PCI Request/Grant #3

Enabled. This is configured with S2-switch 8 (bootstrap bit 9). If you elect to change the setting on this switch, please read 2.3.7. "PCI Request/Grant Bootstrap 9" on page 18

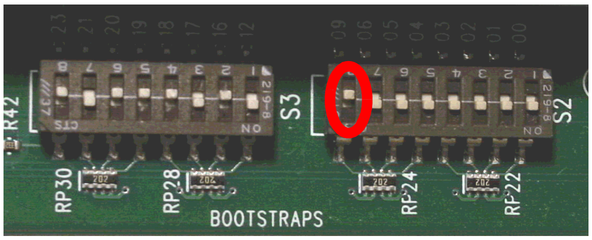


Figure 2-15 Bootstrap Bit 9

2.2.7. JP6 - DMA or PCI Req/Grant

JP6 allows choosing between activating DMA Req/Ack #1, or PCI Req/Gnt #2. The system is shipped with PCI Req/Gnt #2 enabled. This jumper works in conjunction with PCI Request/Grant #3. If you elect to change the setting on these jumpers, please read 2.3.7. "PCI Request/Grant Bootstrap 9" on page 18

Figure 2-16 JP6 - DMA or PCI Req/Grant

2.2.8. External Boot

The External Boot is set to use a 16-bit wide device. The BUR is disabled (so the system boots from the BIOS in FLASH). These are configured with S3-switches 1 and 8 (bootstrap bits 12 and 23).

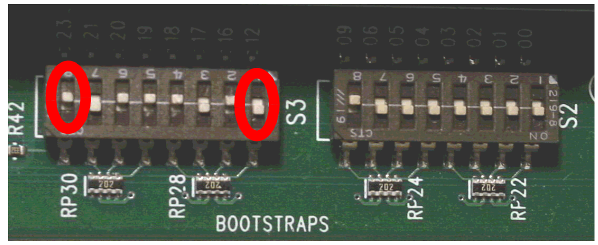


Figure 2-17 Bootstrap Bits 12, 23

2.2.9. USB Test Mode - disabled

This is configured with S3-switch 7 (bootstrap bit 21)

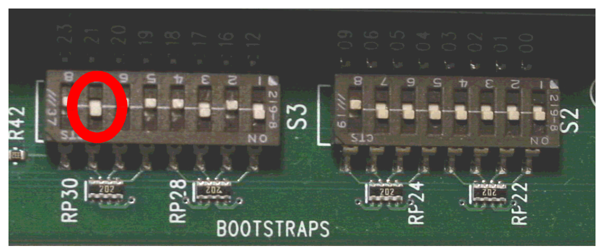


Figure 2-18 Bootstrap Bit 21

2.2.10. Flash Chip Selection

Table 2.1: Flash Chip Selection

Socket	JP7	S3
Bytewide socket U5 and U6	JP7 00 - 02 (Pins 1-3)	S3 #12 Off
ATMEL flash	JP7 02 - 04 (Pins 3-5)	S3 #12 On
AMD flash	JP7 03 - 05 (Pins 4-6)	S3 #12 Off

2.2.11. System Clocking Tables

Table 2.2: SYS and PCI Clock Speed

SYS and PCI Clock	Speed
1 - 3 & 2 - 4	33 MHz
3 - 5 & 4 - 6	66 MHz
1 - 3 only	80 MHz
2 - 4 only	66 MHz

Table 2.3: SYSCLK Source JP3

JP3	Source
1 - 2	CLK2SYS Table 2.2
3 - 4	CLK = 48 MHz
5 - 6	Selectable Table 3

Table 2.4: Clock Mode Bootstrap Registers 16-17

16	17	Multiplier
0	0	x1
0	1	x2
1	1	x3
1	0	x4

Example for 100 MHz:
Set JP3 to 1-2 (see Table 2.3)
Set SYSCLK speed jumper to 1-3 and 2-4
Set S3 registers 16 and 17 to ON

Table 2.5: Clock Multiplier Switch S1

1	2	3	4	MHz
1	1	1	1	
0	1	1	1	
1	0	1	1	4 MHz
0	0	1	1	12 MHz
1	1	0	1	16 MHz
0	1	0	1	20 MHz
1	0	0	1	24 MHz
0	0	0	1	32 MHz
1	1	1	0	
0	1	1	0	
1	0	1	0	8 MHz
0	0	1	0	
1	1	0	0	32 MHz
0	1	0	0	40 MHz
1	0	0	0	48 MHz
0	0	0	0	64 MHz

2.3. Jumper Settings Details

2.3.1. JP3 SYSCLK Source Jumper

Jumper JP3 selects the system clock source. Since the system clock can be any frequency in the range of 4 MHz to 66 MHz, we can select the fixed 33 MHz clock, the fixed 48 MHz clock, or we can use the generated frequencies in the range of 4 to 66 MHz.

In the default condition, in the IDS, we set JP3 to 5-6 selectable and then use the CLK Multiplier DIP Switch S1 to select 64 MHz.

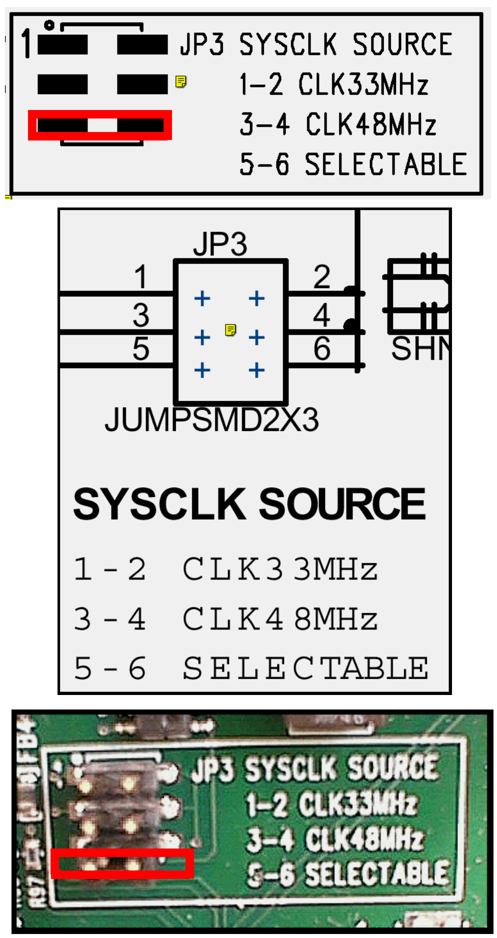


Figure 2-19 JP3 SYSCLK Source Detail

2.3.2. S1Clock Multiplier DIP Switches

If you select 5-6 on JP3, then the clock comes from the S1 selection. In the IDS Board, we use 0000 which gives us a 64 MHz SYSCLK.

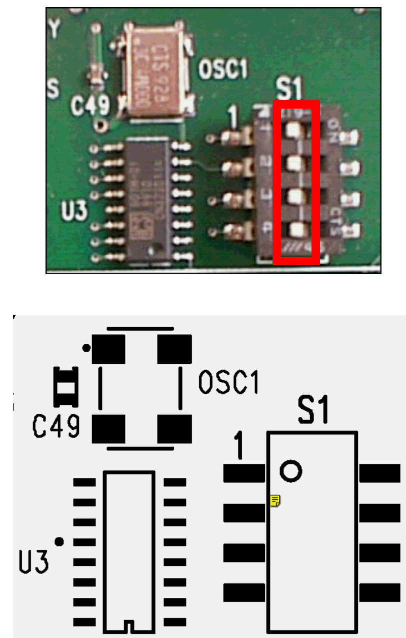


Figure 2-20 S1 Clock Multiplier Select

Table 2.6: Multiplier Select Table

S0	S1	S2	S3	MHz
1	0	1	1	4M
0	0	1	1	12M
1	1	0	1	16M
0	1	0	1	20M
1	0	0	1	24M
0	0	0	1	32M
1	0	1	0	8M
0	0	1	0	N/A
1	1	0	0	32M
0	1	0	0	40M
1	0	0	0	48M
0	0	0	0	64M

2.3.3. CPU Speed

SYSCLK, normally 64 MHz, is routed into the CPU where it is multiplied by 1, 2, 3 or 4 by a Delay Locked Loop. The multiplier, normally 2, comes from bootstrap bits 16 and 17.

2.3.4. PCI Clocking

There is a FrontSide PCI Bus associated with the North Bridge, and a BackSide PCI bus associated with the SouthBridge. In general, you will run them both at 33 MHz. If SYSCLK is 64 MHz, then bootstrap bits 17 and 18 should be set to 1, which specifies SYSCLK/2 for both the Frontside and BackSide PCI Bus.

2.3.5. JP1: 8254 PIT Clock (14MHz)

Generally you would like to drive the 8254 PIT using the proper 14 MHz crystal frequency. . JP1 allows you to route the crystal to the input pin, or *ground* the input pin. This is described in the Annotated Evaluation 1 Board Schematic.PDF. The annotated schematic is ORCAD generated, and you may search for JP1 using a text search. The annotation (on the top of page 2) says:

> "Using JP1 you can select the source of the 14 MHz PIT (8254) clock. This can be generated internally or input into the check. Connect 1-2 to select the generated clock (14 MHz) and 2-3 grounds the clock input to the ZFx86. Do not leave the clock input floating".

Note that if you do not route the crystal to the input pin, you can provide a substitute 14 MHz clock by dividing the SYSCLK by 1484 (internally).

2.3.6. JP9 - Real Time Clock (32KHz)

The RTC has two input pins to connect the crystal to it.

These pins contain a amplifier. In case the frequency is provided from external source the input to amplifier must be used It can be seen from page two of the Annotated Evaluation 1 Board Schematic.PDF.

AE01 is the output of amplifier; AF01 is input. If you use the crystal it is connected with caps and some external resistors between these two pins.

In case we use the divided clock it is routed to the input AF01 with jumper JP9 in position 2-3. The GPIO will be set to output the 32 KHz divided down from 48 MHZ. It goes through the amplifier and into the RTC. As the crystal is disconnected from feedback loop it only adds a small load to the output what does not interfere with operation.

If the crystal is connected to feedback loop of amplifier (JP9 to 1-2) it starts to oscillate. These oscillations are amplified and then used by RTC.

The backup battery provides the voltage to this amplifier to keep the generator running when main supply is disconnected.

The annotated schematic describes JP9: position

- 1-2 activates the 32-khz crystal generator.
- 2-3 Selects the internally divided GPIO0 output to feed the RTC.

This connection is only necessary to provide the timer in a mode where 32 Khz crystal is not used. Power management, Watchdog and SDRAM refresh run off the divided clock.

2.3.7. PCI Request/Grant Bootstrap 9

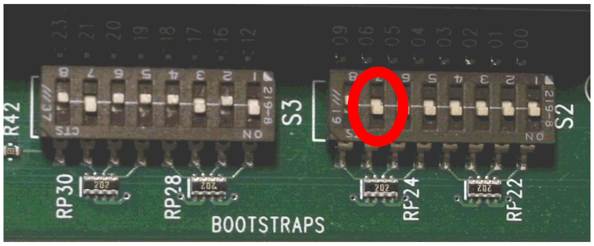


Figure 2-21 BS9 PCI Request/Grant

JP6 allows choosing between activating DMA Req/Ack #1, or PCI Req/Gnt #2. The system is shipped with PCI Req/Gnt #2 enabled.

Figure 2-22 JP6 DMA/PCI

This jumper works in conjunction with PCI Request/Grant #3.

These two options must ALWAYS be used in conjunction with each other.

i) The DIP Switch (S2-8, Bootstrap Bit #9) tells the ZFx86 Chip which type of signals are to be associated with the shared I/O pins on the ZFx86 Chip itself. Like all of the Bootstrap Register Bits, the state of this DIP Switch is read only during Hardware Reset. Once read, changing its state will have no effect until the system is reset.

ii) The two jumpers at JP6 route the signals to the appropriate places on the rest of the board. These must be set to use the same type of signals as previously defined by the DIP Switch.

The nomenclature is inconsistent (thus confusing) when looking at these two controls. The DIP Switch refers to PCI Request/Grant #3, while the JP6 Jumpers refer to the same signals as PCI Request/Grant #2. These do in fact REALLY control the same signals on the board. If the numbers are ignored, and the choice is made simply between DMA or PCI signals, this will be easier to understand.

A few more details on this:

a) While designing the ZFx86 “System on a Chip”, every attempt was made to include all of the standard features and services used by current PCs, plus as many extras as we could squeeze in. In order to accomplish this, a few trade-offs had to be made. The number of available I/O pins on the ZFx86 do not allow all functions to operate simultaneously. We assumed that designers would tend to user either PCI or ISA devices, but not a large number of both types on the same board. The decision was made to share one set of I/O pins between the following functions:

i) DMA Request/Acknowledge #1

- These signals are used (primarily) by ISA cards.
- Specifically, ISA Sound Boards tend to use these as one of the default DMA channels.
- It is often possible to select a different DMA channel, and avoid using these signals.

ii) PCI Request/Grant on Slot #3

(1) These signals are needed for any card installed into the third PCI slot.

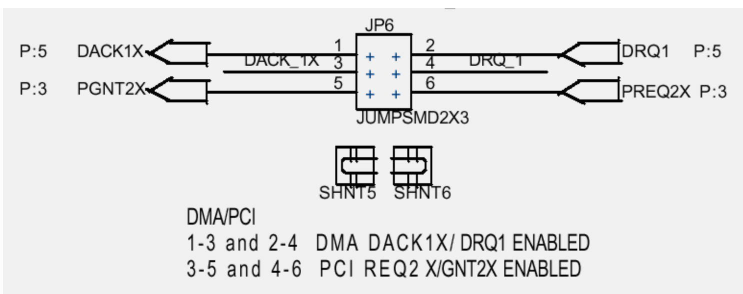
How this relates to the Development System

The ZFx86 Development System, and any other design built around the ZFx86 Chip, can use either of these sets of signals, but only ONE of the two functions can be used on a given board.

- The selection of which set of signals the shared pins will actually provide, is made during the Hardware Reset pulse. The ONLY way to select the other set of signals, is to perform a Hardware Reset.
- The Development System is intended to demonstrate as many features as possible. It has both PCI and ISA slots. Since it has both types of slots, and needs to provide as many features as possible, we need a means to select the desired functionality. The Development System uses one DIP Switch, and two jumpers to select which set of functions will be used. These controls are described further in the next section of this manual.
- In most designs, one of these two functions will be hardwired, and no selection mechanism would be required.

An ISA Sound Card

If you install an ISA Sound Board, you may need to change the settings for these controls. In most cases, the default configuration (all three PCI slots active) will give you more flexibility.



9	63H	1	3^{rd} PCI Request	Third PCI Request/Grant 1 = drq1 = req2_n and dack1_n = gnt2_n

B14	DRQ1	ISA DMA (Optional PCI Master req2_n)
A14	DACK1_N	ISA DMA (optional PCI Master gnt2_n)

Figure 2-23 Analysis of JP6 DMA/PCI

Notes from the Schematic

The Annotated IDS Schematic illustrates JP6 as shown in Figure 2-23. The pop-up note says:

> “The 3rd request-grant pair is shared with DMA request/grant on the ISA bus. The lines must go to the single selected location. Depending on the Bootstrap setting, the jumpers must be in either ISA DMA or PCI GRANT/REQ position.”

The appropriate bootstrap bit is bit 9, which controls the functions of ZFx86 pins A14 and pin B14. When bootstrap 9 is set high (the default) then the signals take their ISA function. The signals then need to be routed to the ISA slots using JP6.

3. Software, Documentation and Design Aids

3.1. Powering Up

The following text describes the necessary items for powering up your IDS system.

Initial Screen - DOS or Linux

When you first power up the system, you will be prompted to type "Dos" or "Linux" to boot into either environment. To run the VxWorks or Windows CE demos, boot to DOS.

Using DOS

There is a limited function Caldera DR-DOS partition. One limitation is that there is no support for the CD-ROM drive. To fix this, see 'Using CD ROM Drive from DOS' on page 23).

Using Linux

If you select Linux, you will eventually get to the log in screen. You have two choices:

Table 3-4: Login to Linux

Prompt	Root (superuser)	User
ZFx86 login:	root	user
Password:	machzroot	machz

Once you are in Linux, you may exit by typing lowercase "halt" or by hitting CTRL-ALT-DEL. For more information on Linux shutdown, see 'Shutting Down Linux' on page 23.

3.2. System BIOS

The BIOS loaded on this integrated development system is a ZF Micro Devices customization based on Phoenix BIOS 4.0 Revision 6.

You may update the BIOS using the Dongle (shown in Figure 3-1).

Figure 3-1 The Dongle

Insert the dongle so that the LEDs are facing the back of the system.

3.3. Software on the Hard Drive

There are three partitions on the provided Hard Disk:

- 1 GB Caldera DR-DOS Partition
- 128 MB Linux Swap Partition
- 4 GB Linux Partition

See "Usage Tips" on page 23.

3.4. Software on the ZF CD

3.4.1. Z-Tag Manager Win 95/98

The primary use of the Z-Tag Manager is to program the dongle. A good example appears in 'Demonstration Program' on page 25.

3.4.2. Acrobat Readers Win/Linux

You may download the latest Acrobat Reader for Linux or WIN 95/98/NT/2000 from the Adobe web site http://www.adobe.com/products/acrobat/readstep.html. A set of recent readers appears on the ZF CD enclosed with the Development System. Linux comes with the Acrobat 4 Reader as well, so this may be pre-installed on the Integrated Development System hard disk.

3.5. Documentation on the ZF CD

Various .pdf documents appear on the ZF Development System CD under the \documents directory. In order for the hypertext links between documents and the Acrobat index to work, all .pdf documents must be in the same directory. So either access them directly from the CD, or place the Document directory and its content in a single directory in your host system. Some pdf files are listed in Table 3-5.

Table 3-5: Documentation on the ZF CD

Filename	Description
ZFx86 Data Book.pdf	Complete Data Book.
ZFx86 Training Book.pdf	Contains many details of the unique features of the ZFx86 device, including the Dongle (shown in Figure 3-1).
ZFx86 Integrated Development System Quick Start Guide.pdf	This manual.
Annotated Evaluation 1 Board Schematic.pdf	Development System board schematic with comments to clarify different aspects of the board.
Ztag Manager Manual.pdf	Z-tag Manager Manual
ZFx86_Eval_1_pcb.zip	PADS PCB Design file for the Development System Board[a]
ZFx86_Eval_1.DSN	Orcad Design file for Development System board[a]

a. PADs files use a *.pcb extension. Orcad files use a *.dsn extension. Orcad is the schematic capture tool. The board layout tool uses the PADs files. The process runs Orcad --> PADs --> Gerber file --> fabricated board.

3.6. Usage Tips

This section includes hints and notes which may provide helpful to you as you use the system.

3.6.1. Shutting Down Linux

You should never turn the power off to a system running Linux without first performing the proper “Shutdown Procedure”. Linux, like many other Operating Systems, runs with several files left “Open” to speed up the response time during normal operation. These files and services must be shut down before the power is removed to prevent corruption of the Hard Disk. The “Shutdown Procedure” can usually only be performed by the System Administrator (root).

If you are logged in as a normal User, enter the “su” (Switch User) command, and the “root” password to temporarily gain “root” privileges. Once you have “root” privileges, issue the following command to shut down the system:

Shutdown –h now

This will start the “Shutdown Procedure”.

Many messages will be displayed on the screen as the various programs and services are shut down. Wait until you see a line that reads “power down”. It is now safe to turn the power off.

The Shutdown command allows you to schedule a shutdown at a later time. You may also shutdown using the Halt command, or by doing CTRL+ALT+DEL. These latter methods are not as graceful, but they work.

3.6.2. Using CD ROM Drive from DOS

The Caldera DR-DOS does not ship with MSCDEX.EXE, so the CD will not work. The DOS driver for the CD is provided on a floppy shipped with the Integrated Development System.

You have two methods to "fix" this problem if you wish to use the CD under DOS:

(1) if you have licensed Caldera DR-DOS, add the MSCDEX file and change autoexec.bat and config.sys as follows:

autoexec.bat:

```
mscdex.exe /d:cd001 /m:10
```

config.sys:

```
device=himem.sys
DEVICE=cddrv.sys /D:CD001
dos=high,umb
```

(2) If you have licensed another version of DOS, say MS-DOS or PC-DOS: boot the hard disk into DOS, power down and boot your own DOS version. Use SYS to transfer the System to Drive C. Reboot into DOS and replace the rest of the DOS files.

3.6.3. Set the Boot Default to DOS

Your IDS will boot to Linux by default, unless you type DOS soon after bootup. To change this default, if you are going to use DOS for a while, log into linux as root/machzroot.

Edit file /etc/lilo.conf (with your favorite Linux text editor). Change "default=linux" to "default=dos".

You may also change the on-screen prompt by editing file /boot/message. You may change the line "Linux will start" to "DOS will start",.

Once you are out of the editor, type "lilo" and press enter. That will recreate the lilo configuration files. (Lilo stands for Linux Loader). You may then type "reboot" in Linux.

3.6.4. VxWorks Setup

See the VxWorks Demo Programs in Chapter 4 following.

3.6.5. Using the Flash

GPIO0 can be connected to 2 flash sockets:

FLASH_1 and FLASH_2 using jumpers at
FLASH 0-4 CS CONF header.

4. Demonstration Program

Thedemonstration programs demonstrate software or software techniques using the ZFx86 integrated development system. The development system provides a general-purpose platform for developing with the ZFx86. The IDS's PCI in ISA slots are available to test peripherals, and because the development system arrives up and running (out of the box) with software installed on hard disk.

Wherever possible, these demonstration programs are already installed on the hard disk. In addition, the source and binary files appear on the accompanying CD. In the same manner that we annotated the IDS board schematic, we provide some annotation and notes on the software source code, and we provide instructions on how to build the software.

There are many different ways to build and test software. Some software is built using a command line interface and some using a batch or makefile. Other software is built using some sort of IDE (integrated development environment). Testing may be done on the target system or perhaps with some type of cable using a debugger on the host.

In the case of VxWorks, many developers do virtually everything on the host using Tornado 2. The Tornado 2 environment allows individual modules to be rebuilt and downloaded to the target while the target remains running.

4.1. Dongle Flash Programmer

The BUR accepts a sequence of records via the Z-tag port (*not* the COM1 port). These record sequences form a command structure. The following text discusses the event sequence when you power up the ZFx86 chip:

4.1.1. Analysis: ZFx86 Power On

When you power on or reset the ZFx86 chip, while the address bus is tri-state, the chip samples the 24 bits on the ISA address bus. The hardware designer may use DIP switches or jumpers to override the 24 default settings on these bits. These 24 bits are read into an internal 24-bit register called the boot parameters register. One of the bits (or dip switches) tells the ZFx86 whether to boot from the external flash or from the internal BUR.

In normal operation the ZFx86 boots from external flash (that is, whatever is selected by the memcs_0 pin). If bootstrap bit 23 is asserted, then the ZFx86 boots from the internal Boot Up ROM (BUR). When the BUR comes up its first operation is to read from the Z-tag port. The Z-tag port is essentially a two wire port designed to read from a dongle. If no dongle is found on the Z-tag port, then BUR issues a command prompt to the serial port. Thus, on a power-on-reset (or reset) the ZFx86 boots either from the external flash or from the BUR. If it boots from the BUR, you may provide a series of commands through the Z-tag port. These commands are actually data structures which may contain downloadable code and data. The BUR itself contains:

- Basic chip initialization code (a kind of mini-POST)
- A simple console driven debugger
- Code to process Z-tag command and data structures
- Ymodem transfer and download function
- A set of callable subroutines (sort of a mini-BIOS)
- Download and execute test program capability (BUR executables)

You use the Z-tag port to download small test programs into the ZFx86 internal static RAM. These programs assist with the bring up of a newly designed ZFx86 prototype, or they may be associated with a FailSafe recovery of system code.

To flash a new BIOS, use the Z-tag port to download a flash programmer followed by a data packet containing a BIOS image. The flash programmer uses the callable BUR subroutines to read data packets from the Z-tag port and copy those packets into the flash.

Place the command and data structures into the IDS using a Windows-based program called the Z-tag manager. This program formats the command and data structures and then writes them out to the printer port.

ZF provides a device called a "Dongle" to facilitate this download. Connect one end the Dongle to the printer port cable, and on the other end to a 14 pin connector designed to mate to a similar connector on a target board.

4.1.2. Dongle Types

ZF offers two dongle types:

- Memory Dongle containing 1 or 2 SEEPROMs for upto 256K program storage
- Fast-PassThrough Dongle

4.1.2.1. Using the Memory Dongle

Connect the Memory Dongle to the printer port of a host system running the Z-tag manager, and use the host system to load commands and data into the Dongle's serial EEPROMs. Then disconnect the Dongle from the host and connect the Dongle to the 14 pin connector on the target board. Thus you carry the code to the target. Once you plug the dongle into the target board and reset the board, the BUR reads and executes the commands it downloads from the Memory Dongle. Typically, you place small test programs or an image of a BIOS to be flashed into the Dongle.

4.1.2.2. Using the Fast-PassThrough Dongle

Connect the Fast Pass-through Dongle using a cable from the host system's printer port to the Dongle, then plug the Dongle into the target board. The Z-tag manager provides the data structures directly to the BUR. Thus pass-through mode may be used to transfer large data blocks very quickly into your target system.

In either case, when you plug the Dongle into the target board, there's an electrical connection in the 14 pin connector which automatically asserts bootstrap bit 23 and therefore causes a BUR boot on reset.

4.1.2.3. Using An On-Board Dongle

It is also possible to place a SEEPROM on the target board and manually assert boot strap 23 (via a dip switch or jumper). This effectively gives you an onboard Dongle. With appropriate switching your target board may support both an onboard and off board Dongle. An onboard Dongle may be used for fail-safe recovery of a target system.

4.1.3. Demo of the Flash Programmer

We need to make a data structure in the Dongle as shown in the Z-tag Manager User's Guide The data structure shown in the manual is:

<02 "Select Serial Device" on page 27. >

<01 "Upload and Execute Command (Basket Contains Our Program)" on page 28. >

<FE optional - "Starting Address Parameter (Basket Contains A Parameter for Our Program)" on page 30. >

<F0 "RLE Compressed Basket (the ROM Image)" on page 31. >

<05 "Stop Processing" on page 31. >

The select serial device allows the flash programmer (the BUR Extension program which runs in the ZFx86 on Chip SRAM) to write notes to you, the user, on the ZFx86 serial port.

This text section demonstrates the Flash Programmer and the "write" fuctionality. The flash programmer reads the BIOS Image from the "FF" data basket, a block at a time, and programs it into the FLASH.

When the Flash Programmer is done, we use "05" Stop Processing to finish the job.

Install the Z-Tag Manager Software

There is a video presentation in the root of the IDS CD -- which shows you how to install and use the Z-tag manager. You may run this in a windows environment (use the Autorun or click on file Zfdefsys.hlp).

Install the Z-tag manager by unzipping the package (found on your CD under \Z-tag Manager). Unzip into a temporary directory as you do not need the files once the installation is done. When you finish copying the files, run SETUP.EXE from the temporary directory.

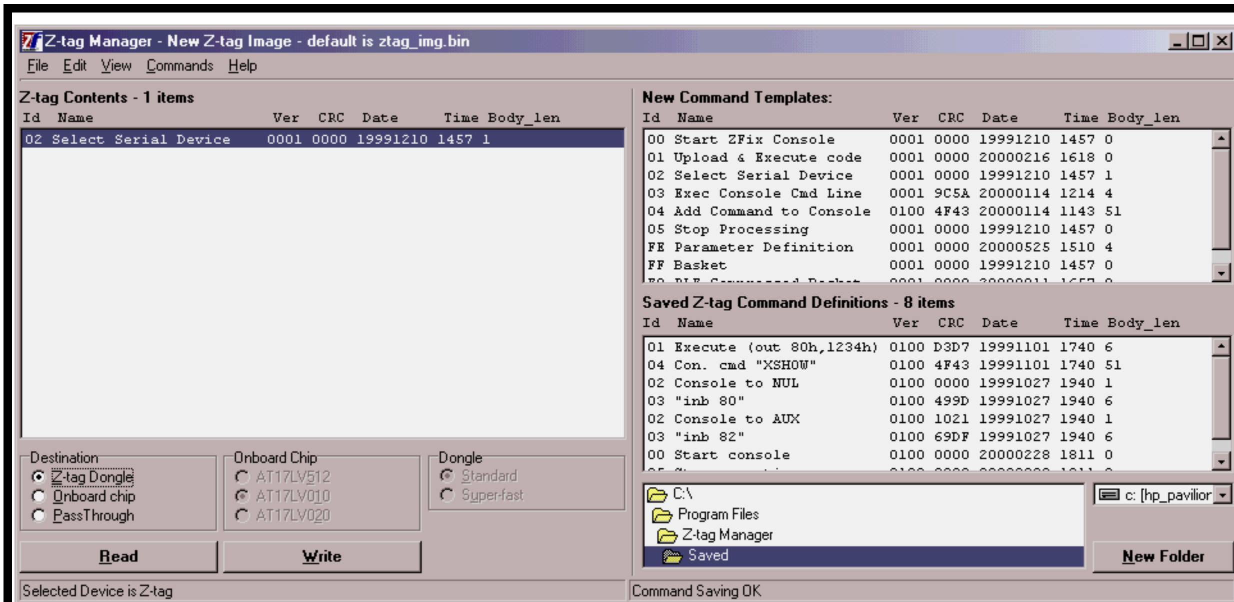


Figure 4-2 Z-tag Manager

4.1.4. Build Command Set In the Z-tag Manager

Select Serial Device

First enable the serial port. This instructs the flash programmer to write any comments to the serial port during the flashing operation. Double-click on the **Select Serial Device** command, and select the "Serial Port" button as the device.

If you select the "Z-tag" as the serial device, it implies that you want to output to the LEDs on the Dongle.

This command enables data output to the serial port. The console setting remains selected until the next execution of this command, so only execute this command when changing/disabling the output device.

A data structure is (of course) created by this command. The structure is exactly like that of basket's, only the command ID is different. There is a 1-byte body file which contains 0, 1 or 2 depending on your radio button selection.

When the flash programmer runs, it sets a variable in source file BURAPI.ASM. This variable is actually an alias to a BUR internal data structure.

SerialMode db 0; 0=none 1=Serial 2=ZTAG_LEDS

.

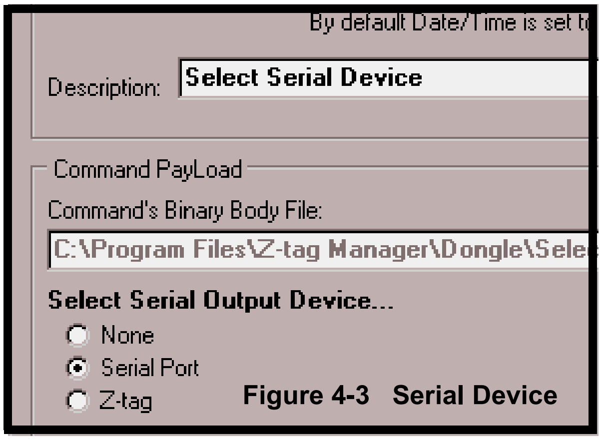
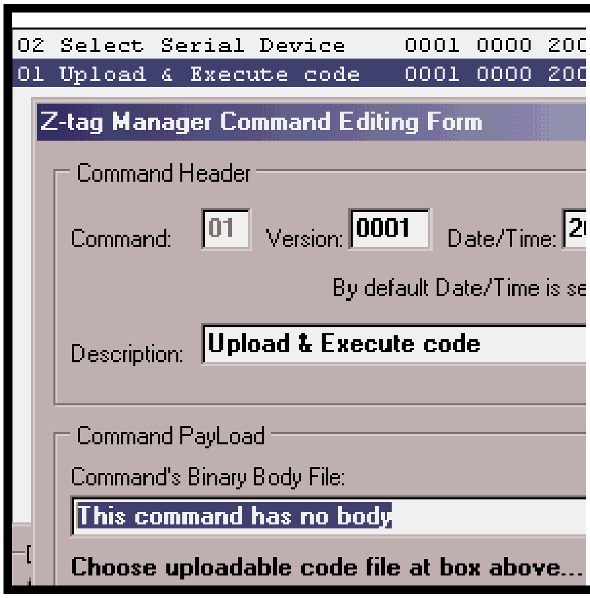


Figure 4-3 Serial Device

Upload and Execute Command (Basket Contains Our Program)

To this point command(s) in the dongle, which may be viewed as a set of data structures, have been interpreted solely by the BUR (there *is* no other code to interpret them). The upload and execute command contains a “data basket” which is our flash programmer code: the code which reads the ROM image from the Dongle and Writes the ROM Image into flash.



At this point we are editing an 01 Upload and Execute command which, when executed, causes the BUR to transfer our program into the ZFx86 on-chip SRAM.

Type the name of the .COM or .ROM file for your flash programmer, or use the “Browse” button to navigate to the program.

There is an important reason for this browse step: if you use the Browse button to locate the file, then a subsequent “refresh bodies” command automatically identifies the original file location and updates the body (payload) if you update your flash programmer.

Figure 4-4 Edit 01 Upload/Execute Command

Preparing the Payload for the Upload and Execute Command (Our Program)

You must load a file that executes a code image at memory space “0”. In general, a .COM file image starts executing at 0x100 by default. To solve this problem, we put an ORG 0 into the source file. (See the text below.) The .COM file thus generated by the assembler and linker is “different”, so consider renaming it using a .ROM or a .BUR file extension. The extension used is irrelevant to the Z-tag manager.

Use the resulting .COM file as a payload for the BUR “upload and execute” function, renaming it using a .BUR extension. Compile using MASM 6.11 with following command line:

`ml /nologo /Fl /Zm /Fm AM29Fxxx.ASM`

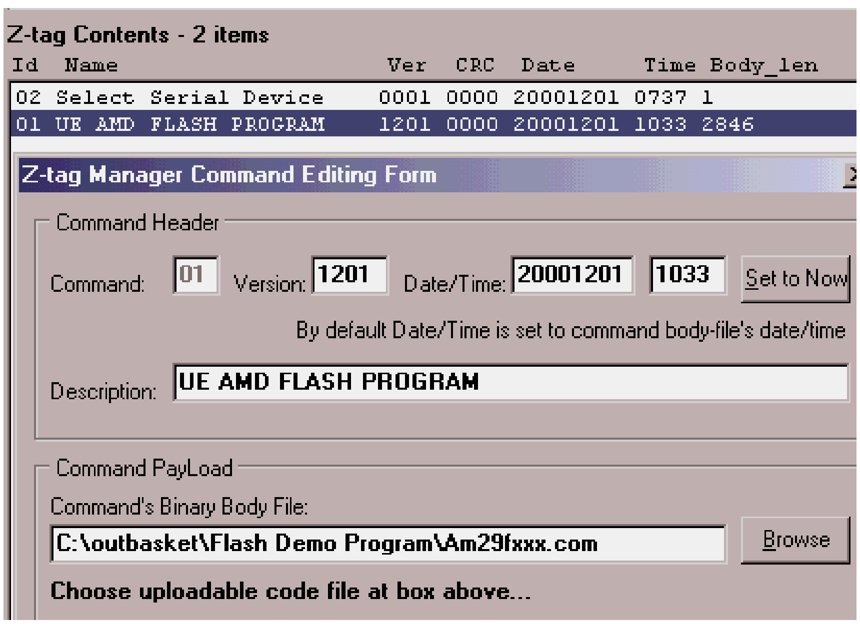
The ORG 0 is placed into the code to ensure that we get the desired output file:

```
assume cs:code; es:code; ds:nothing          ; see page 37
code                segment USE16 public
                    org     0

START:
                    push    cs
                    pop     es
                    call    PRINT_AREGS         ; demonstration uses debugsub.asm
                    jmp     start1

                     include debugsub.asm      ; not needed
```

Editing Our Program Name into the Command’s Payload



In this example we set up our own personal version code, and changed the description field associated with this 01 command instance, and browsed to access the .COM file we want to upload and execute. The browse operation filled in the full pathname.

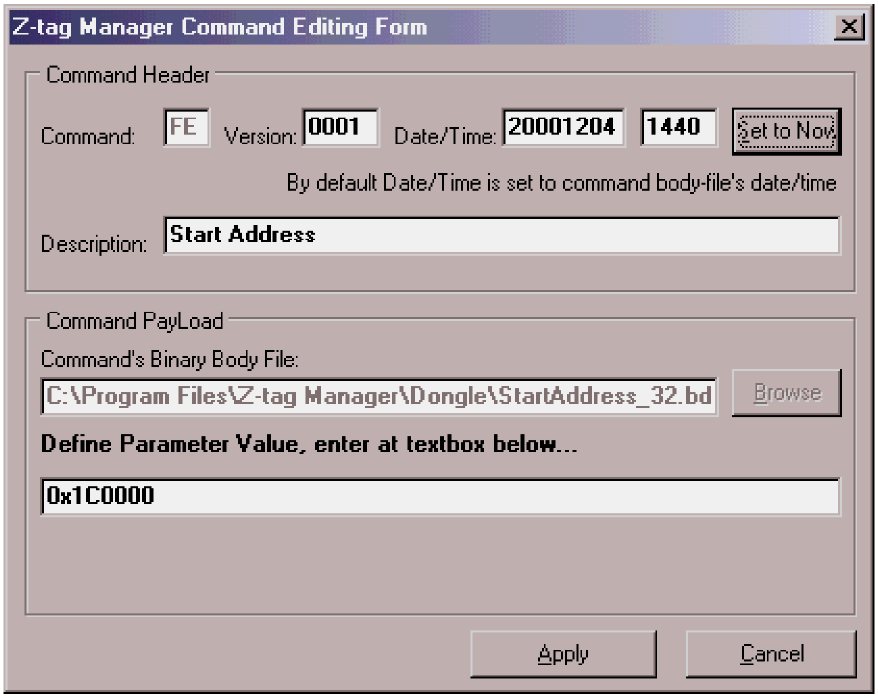
Eventually our program loads and starts executing. The program then looks for a parameter and then for the ROM image data in the dongle data structure.

Figure 4-5 Command 02 - Upload and Execute

Note that the 01 Upload & Execute command name has been edited (in the Description field) to "UE AMD FLASH PROGRAM" which means upload and execute the AMD Flash Programmer.

Starting Address Parameter (Basket Contains A Parameter for Our Program)

The FE Parameter Definition command provides a 32-bit number to the Flash Programmer. You can see the execution of this section of code in Figure 4-9 Monitoring COM1.



If the particular flash programmer code you use takes a parameter, an FE Parameter Definition Command would appear next. In the body file put the starting address (or whatever is required by the specific flash programmer). Here we have picked up the FE command and named it the "Starting Address"

Figure 4-6 The Parameter Definition Command

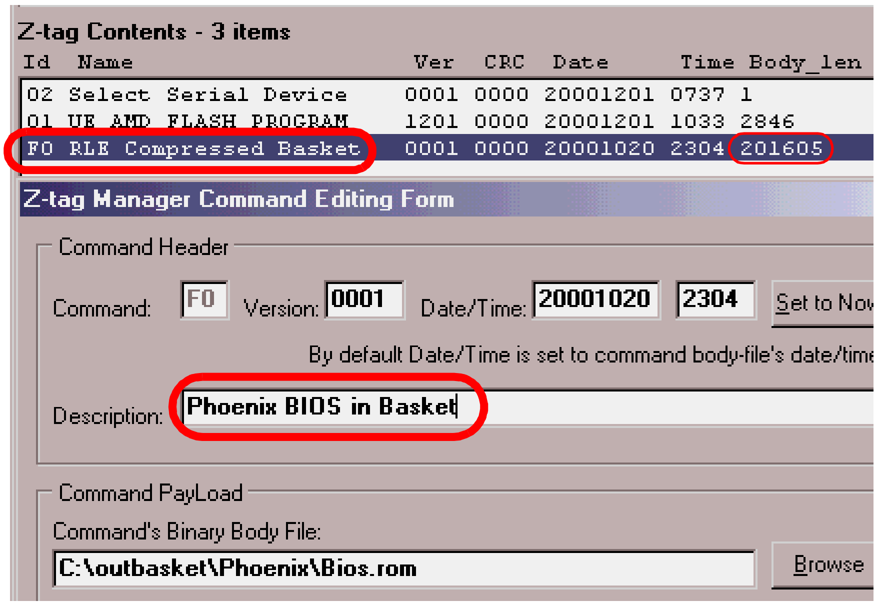
```
start1:
        LEA     DI, message1    ; 'ZFx86 IDS AMD Demo BUR Flash Programmer'
        call    print
        LEA     DI, message2    ; -------------------------------------
',CR,LF,0
        call    print

        call    ZTPrepareRead   ; BUR function use before ZTRead

        call    ParmRecord2EAX  ; fetch parameter record to EAX
        jnc     ParmOK          ; procedure in flashpgm.asm
```

RLE Compressed Basket (the ROM Image)

The FO – RLE Compressed "Basket" command is a structure that contains data that we call the the payload. Use the FO Compressed Basket command to capture the Phoenix BIOS image.



Note that in this picture we selected the F0 command into the list, and then edited it to put the 210605 byte file BIOS.ROM in the command payload. Now we *could* edit it again to change the description field. We *could* update the date/time to "Now".

If you put the file name in using the Browse button, then "Commands → refresh bodies" will update the command payload if it finds a newer file in the same directory path.

Figure 4-7 Editing the F0 Command

Stop Processing

The Stop command lights the GREEN LED on Z-tag dongle and freezes the BUR. Use this command last to notify the operator that the program is complete. It prevents an infinite execution of data fetch/exec procedures.

4.1.5. Using the Command Sequence

Transfer Image to Dongle

Set the radio button to Destination →Z-tag Dongle and push the Write Push button. Once this is done, place the Dongle on the IDS socket and press the reset button. When the red LED stays on continuously, the flashing is complete.

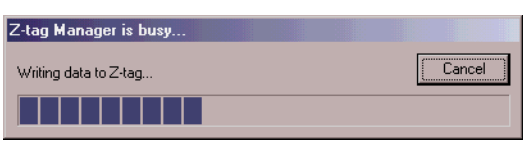


Figure 4-8 Writing to the Dongle

Monitoring the Flash Programmer

The ZFx86 outputs a prompt on the ZFix Console when you apply power and bootstrap BS23 is in the on positon and there is no Dongle plugged into the board.

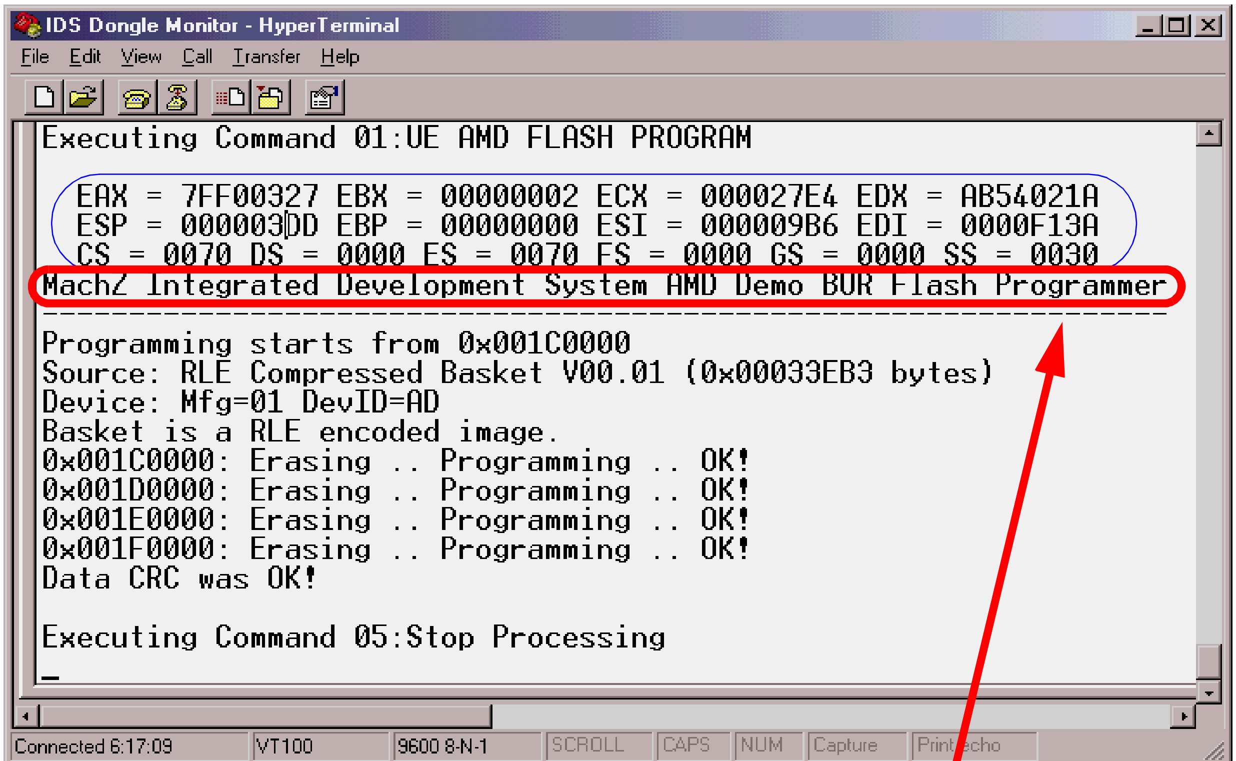
1. Set up HyperTerminal to 9600-N-1 and no handshake.
2. Type "help" to see the available command list.
3. Once you know that you have the serial port working, plug the Memory Dongle in and press the Reset button.

You see the output messages from the BUR and the Flash Programmer as they are written to the serial port. The “01 Executing Command" comes from the BUR, but it uses the title we changed to “UE AMD FLASH PROGRAMMER”.

The register display comes from a call

PRINT_AREGS on line 72 (see the listing following on page 36).

The print to the serial port (also shown using a Hyper Terminal monitor set to 9600-8-N-1) of the message “ZFx86 Integrated...” comes from the code in line 115-116 (below and on page 38.).




```
115     LEA      DI, message1     ; 'ZFx86 Integrated Development System AMD Demo BUR
116             call     print
117             LEA      DI, message2     ; ----------------------------------------
                      ',CR,LF,0
118             call     print
```

Figure 4-9 Monitoring COM1

Saving Your Work

Save your work as a binary image or a command set. Once you have made up a command list, save it as a binary image file (using **File →Save Device Image As**). When you execute this save and later restore the Z-tag contents, the panel loads from the previously saved file. You can later recover the binary image using **File →Open Device Image**.

Alternatively, you can save the command list. To do this, use the **Saved Z-tag Command Definitions** window (right center) which is a window into the highlighted Folder (lower right). Anything you drop into the saved command list remains there for your future use.

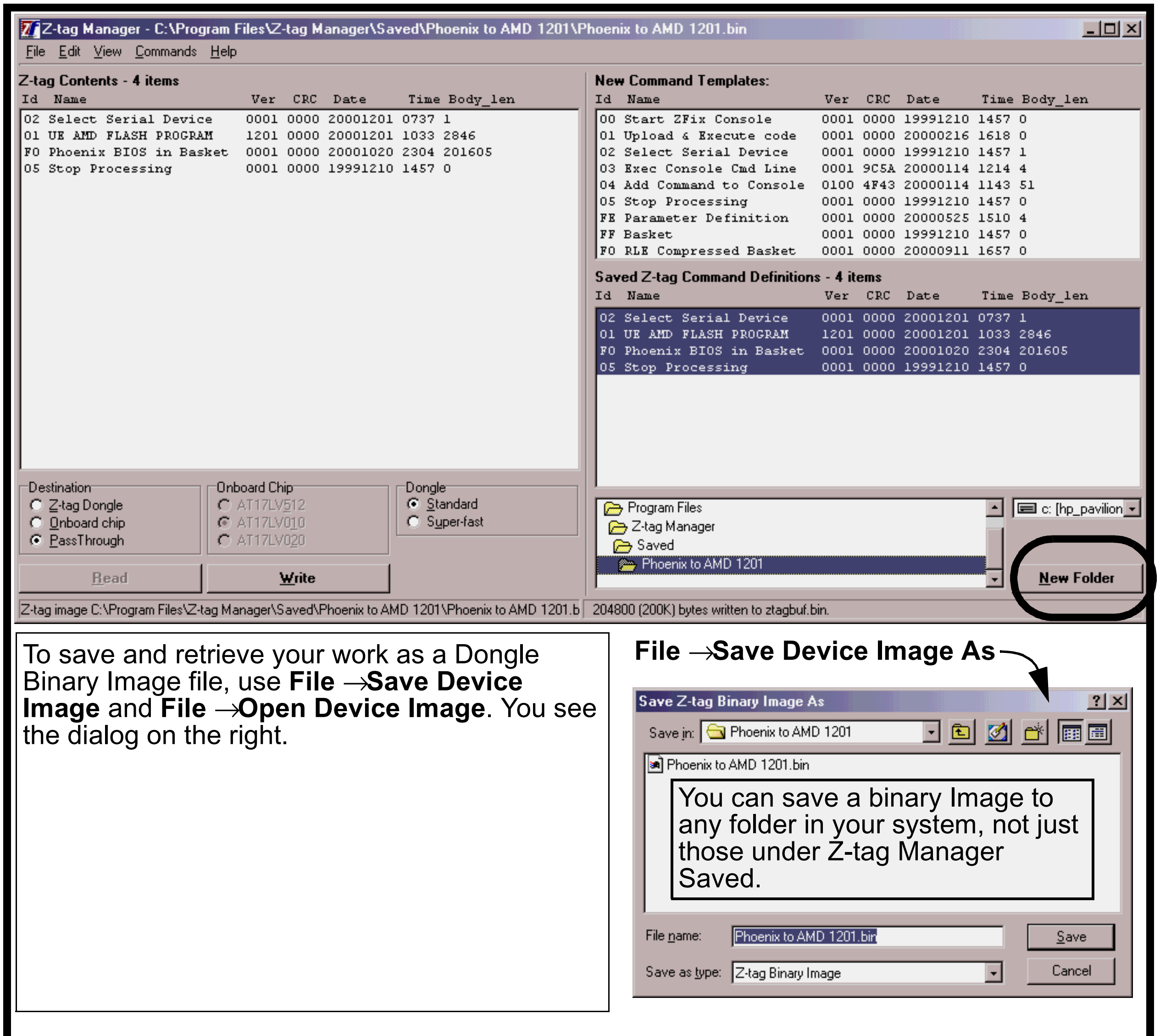


Figure 4-9 Saving Your Work

The **Command →Refresh Bodies** only updates those selected bodies in the main (upper left) window. For example, if you change the flash program's object file, select that line and use the refresh bodies command to update the program image which is inside of the command list.

4.1.6. Analysis: Source Code

The source code is made up of five files:

Table 4-6: BUR Flash Demo File Set

File	Function
AMDFLASH.ASM	Main Line Program
DEBUGSUB.ASM	Demo to Dump Registers
RLE_MEMW.ASM	Interface to Dongle Structures and Utilities for Memory Window Creation
BURAPI.ASM	Interface to BUR
FLASHPGM.ASM	Procedures

The BUR is in ROM code on the ZFx86 chip. It is operational when you boot from BUR. It is not operational when you boot from an external ROM. To access the procedures in the BUR, see the BUR API documentation in the ZFx86 Data Book. You will need to include file BURTAPI.ASM as we do in the example following (see line 442).

The source files for this program are on the ZFx86 Integrated Development System CD in directory \BUR Programs\AMD Flash Demonstration Program. The listing of AMDFLASH.ASM following is fully commented and has some additional footnotes for clarity.

The flash programmer accepts a parameter which specifies where to put the basket data in the Flash Chip. In the case of the Phoenix BIOS and a 256K Dongle, we need to compress the BIOS into an RLE basket. However, other things can be put into the flash as well with the Dongle: thus the parameter can be useful. To illustrate this, the demonstration program picks up the parameter in line 123 following (see page 38).

4.1.7. Program Preparation Steps

The program was prepared using Microsoft MASM 6.11d. The “d” update is a good thing to get. Editing is done in a DOS window, and to make sure the environment is set up correctly you may run:

```
path %path%;c:\masm611\bin;c:\masm611\binr;
```

Then to build the .COM file use:

ml /Fl /Zm /Fm /AT AMDFLASH.ASM > errors
more errors

This allows you to bring up the errors file in your text editor.

4.1.7.1. Using the "Fast Dongle" in PassThrough Mode

Speed up the download process by using the "Fast Dongle[1]" in PassThrough mode. Run the cable from the printer port of the host system to the dongle. Leave the dongle plugged into the target ZFx86 IDS.

Remember to select the “PassThrough” button on the Z-tag Manager Console.

The RESET push button on the front of the IDS causes the write process to commence. That is, the bar graph on “Writing passthrough data” will not start to move until you push the reset button. The program actually runs as it is downloading. The BUR reads the Select Serial Device and the Upload and Execute Command, and then the program itself reads the Start Address Parameter Block and the RLE Compressed Image.

4.1.7.2. Using the "Memory Dongle" in PassThrough Mode

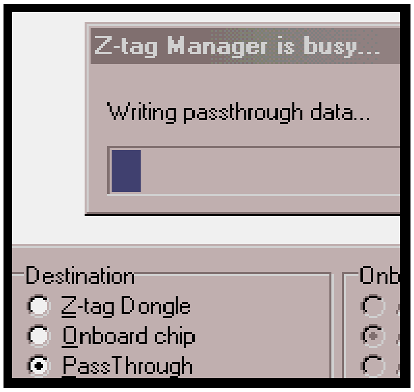
Speed up the download process by using the "Memory Dongle"[2] in PassThrough mode.

1. Set the jumpers as marked on the Memory Dongle.
2. Connect the printer cable from the printer port of the host to the dongle. Leave the

1. The Fast Dongle contains no on-board SEEPROM chips.
2. The Memory Dongle contains one or two SEEPROM chips, and two jumpers that require configuring. See the Z-tag Manager User’s Guide for more information.

dongle plugged into the target ZFx86 Integrated Development System. Remember to enable the “PassThrough” button on the Z-tag Manager Console.

Figure 4-10 Enable PassThrough Mode



3. Push the RESET button on the front of the IDS. This causes the Write to begin.

The bar graph on “Writing passthrough data” moves after you push the reset button. The program actually runs as it is downloading. The BUR reads the Select Serial Device and the Upload and Execute Command, and then the program itself reads the Start Address Parameter Block and the RLE Compressed Image.

The passthrough mode is slower in that the Z-tag interfaced is paced by the ACK signal in passthrough mode – allowing the ZFx86 to wait for data from the printer port.

4.1.7.3. Placing the BIOS or Image in the Memory Dongle

You may program the BIOS (or other flash item) into the Memory Dongle with the Z-Tag Manager. Use the “refresh bodies” command to simplify editing process of the BIOS file. See Figure 4-11.

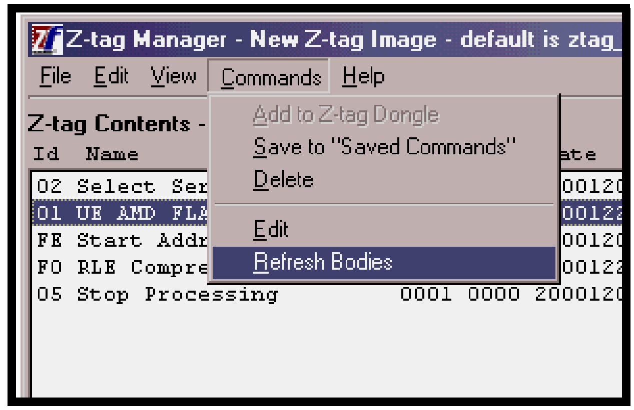


Figure 4-11 Refresh Bodies

Select (highlight) those items which have baskets which you wish to refresh. The baskets should have been originally identified using the Browse button. See Figure 4-5, or Figure 4-7. To select more than one item, hold down the CTRL key. Then select “Refresh Bodies”. The file references (.COM file for the program, or the .ROM file for the ROM image) updates. Note that the file names do not require these specific extensions. When the refresh occurs, the time/date stamp then agrees with the time/date stamp on the .COM or .ROM file.

Once you write the image to the Dongle using the Write button, put the dongle into the ZFx86 Integrated Development System with the LEDs facing the back. The Dongle is properly seated when both LEDs are lit. Reset the computer and set up a HyperTerminal Monitor for the COM1 port, and your flash operation will complete in a matter of seconds. See Figure 4-9.

Note: When programming the dongle, use a printer port extension cable so that you need not to to the back of the host system.

4.2. VxWorks Shell Demo

The shell demo is the simplest possible demo we have for VxWorks. Refer to page 36 through page 45. It is generated from one directory, and it brings up a VxWorks shell to which you can add your own C program as a task. A trivial “hello world” program is in there as a starting point.

```
1   Source Listing - AMDFLASH.ASM
2   comment *
3
4       This program is a ZFx86 BUR Extension which will execute out of the ZFx86
5       on-chip SRAM under control of the BUR.  The program is a flash programmer
6       for AMD 8-bit chips This programmer reads a variable-length payload from
7       the Z-tag input port and writes the data into the flash device.  The
8       programmer requires one parameter record which is the start address within
9       the flash.  This programmer will write the payload to flash chip, starting
10       from the address defined in the parameter.
11
12       There is no checking for overruns.  If the chip is smaller than needed,
13       wrap-around will occur.
14
15       (c)2000 ZF Micro Devices, Inc.
16
17
18       revision history
19       ----------------
20       CRC 12-26-00        changed structure, removed macros, added comments
21
22
23     Target Chip:          AMD AM29F0xx in ZFx86 Integrated Development System
24     Size:                 2 Megabyte
25     Chipselect:           ms_cs0
26     Mode:                 8-bit
27     Chip Address:         defined by Parameter record (use 0x000C0000)
28     Window size:          10000H (64K)
29
30       This code executes as BUR "Load and execute" function.  It will fetch
31       parameter record and payload code following to the executable code in
32       dongle.  Compile using MASM 6.11 with following command line:
33
34           ml /Fl /Zm /Fm AMDFLASH.ASM
35
36       The resulting .COM file can be directly used as payload for BUR "Upload and
37       Execute" function
38
39     endcomment *
40
41   .486p
42
43
44   ZFINDEX         EQU     218H    ; ZF-logic Index
45   ZFRW8           EQU     219H
46   ZFRW1632        EQU     21AH
```

Notes: line 2 uses a * to start a comment, and thus all test is a comment until the end of line 39. You could also say comment x .... x where then x becomes the comment delimiter. This illustrates how the "comment" directive in ML 6.11 allows multi-line comments. In this example, the term endcomment is not necessary -- just the word comment and a start and stop delimiter.

On line 41 we identify the source code as 486 level, so that 486 instructions may be used. The p says protected mode, so we could use those instructions too (at leasst, ML the assembler would allow it).

```
47
48
49      MEMCS         EQU   0                       ; use chip select 0
50      PRGBASE       EQU   0E0000h                 ; where to create memory window
51      CS_PAGE       EQU   ((26h+(MEMCS*12))+8)    ; ZF Logic Page Register for MEMCS
52      CS_BASE       EQU   (26h+(MEMCS*12))        ; base register for specified
chipselect
53      CS_SIZE       EQU   ((26h+(MEMCS*12))+4)    ; ZF Logic Size Register for MEMCS
54      MEMCONTROLL   EQU   5AH                     ; ZF Logic Memory Control Low
55      ZT_WRITE      EQU   5EH                     ; ZF Logic ZT Data Write Register
56
57
58      _16BIT_RW     EQU     00000000b
59      _8BIT_RW      EQU     00000001b
60      WINDOWMODE    EQU     _8BIT_RW                ; "global" parameter
61
62      CR            EQU     0DH
63      LF            EQU     0AH
64      TRUE          EQU     1
65      FALSE         EQU     0
66
67                    assume cs:code; es:code; ds:nothing
68    code            segment USE16 public
69                    org     0
70
71    START:
72                    push    cs
73                    pop     es
74                    call    PRINT_AREGS     ; demonstration uses debugsub.asm
75                    jmp     start1
76
77                    include debugsub.asm    ; not needed
78
79
80    message1        db      CR,LF, 'ZFx86 Integrated Development System AMD Demo BUR
Flash Programmer',CR,LF,0
81    message2        db             '------------------------------------------------
----------------',CR,LF,0
82    message3        db      '0x',0
83    message4        db      ': Erasing .. ',0
84    message5        db      'Flash programming start address missing!',CR,LF,0
85    message6        db      'Programming starts from 0x',0
86    message7        db      'Device: Mfg=',0
87    message8        db      ' DevID=',0
88    message9        db      'Detected Flash Device: ',0
89    message10       db      'Programming .. ',0
90    message11       db      'RLE basket checksum error ',0
```

Notes: We need to send up a memory window (sort of a portal) to provide access the the flash chip. It works sort of like the LIM logic of the past: the flash chip may be quite large and cannot live in the memory space -- the one on the IDS is 2 MB. Also, we want to access the chip with 0 glue logic (no extra components). The ZFx86 built-in memory windows solve both problems. But that means that in order to access the chip, we need to create a window. We are running in BUR so we own the machine -- so we have decided to create the window from E0000 to F0000. Line 50 sets the PRGBASE, and the size is hardwired into other procedures as 64K. That means we will program the chip 64K at a time, and then move the target portion of the window using the ZFx86 PAGE register.

```
91   message12       db      'Data CRC failure: ',0
92   message13       db      'Chip Not Supported',CR,LF,0
93   message14       db      'FAILED!',CR,LF,0
94   message15       db      'OK!',CR,LF,0
95   message16       db      'Data CRC was OK!',CR,LF,0
96   message17       db      'Programming Failure ...',CR,LF,0
97
98   fIDError        db      FALSE
99   FlashBase       dd      ?    ; variable to store starting address in chip
100
101   ;   ----------------------------------------------------------------
102   ;   subroutine print - print ASCIIZ string. ES:DI -> string on entry
103   ;   ----------------------------------------------------------------
104
105   print   proc
106           pusha
107           xor     cx, cx
108           call    SerSend
109           popa
110           ret
111   print   endp
112
113
114
115   start1:
116           LEA     DI, message1    ; 'ZFx86 Integrated Development System AMD Demo
BUR Flash Programmer'
117           call    print
118           LEA     DI, message2    ; ----------------------------------------------
--------------------------',CR,LF,0
119           call    print
120
121           call    ZTPrepareRead   ; BUR function use before ZTRead
122
123           call    ParmRecord2EAX  ; fetch parameter record to EAX
124           jnc     ParmOK          ; procedure in flashpgm.asm
125
126       comment * in this generalized program, you can specify the target address
127       within the flash chip.  If you are "burning" in the Phoenix BIOS in the
128       IDS, using the 2 MB IDS AMD Flash Chip, you use starting address 0x1C0000.
129       We could hard code this into the program, but leave it here as an
130       instructional example.  *
131
```

Notes: The best way to understand the AMD Flash Chip is to read the document on their web site -- AMD data sheet for the Am29F016D, publication 21444 Rev E. The source code of program AMDFLASH.EXE (a DOS program used to flash the ZFx86 Integrated Development System BIOS via a Floppy or HDU) also contains some good C examples on how to deal with the AMD Flash Chip. In line 98 we create a boolean (flag ID Error) which we will set TRUE if either the manufacturer ID or chip ID is not what we are looking for.

Back on line 72 we set ES to point to CS so that all of our messages would be ES relative. That's because the SerSend procedure in the BUR (see Appendix B of the ZFx86 Data Book) requires ES:DI to point to the ASCIIZ string to print to the serial port COM1. The print procedure uses **SerSend** on line 108. Line 443 has the include for BURAPI.ASM which is the interface layer to the BUR, and which defines SerSend.

Line 121, 123: Procedures **ZTPrepareRead** is documented in Appendix B of the ZFx86 Data Book. The program now wants to read the parameter (starting target address in the flash) from the Z-tag Parameter block. It calls a procedure (grep the 4 ASM source files) which does this.

```
132
133         LEA     DI, message5    ; 'Flash programming start address missing!'
134         call    print
135         jmp     ExitPgmrFail
136   ParmOK:
137         and     eax, 0FFFFF000h ;  ZF Logic BASE reg: rightmost = 0
138         mov     FlashBase, eax
139         push    eax             ; save for a moment
140
141         LEA     DI, message6    ; 'Programming starts from 0x'
142         call    print
143
144         pop     eax             ; restore eax
145         call    SerOut32        ; BUR API proc writes EAX to Serial
146         call    CRLF            ; BUR API proc writes CRLF to Serial
147
148         ; Get the Basket Command
149
150         MOV     BX, OFFSET ExitPgmrFail ; pass in fail address
151         Call    process_basket_header   ; our procedure in flashsub.asm
152
153         ; Create Memory window for accessing the flash device
154
155         Call    Create_Memory_Window
156
157         ; Time to program some flash.
158
159     comment * Check, whenever or not we have supported part on-board.  A good
160     reference here is the AMD data sheet for the Am29F016D, the publication
161     21444 Rev E on the AMD web site. The chip used on the ZFx86 Integrated
162     Development system has a Manufacturer ID of 01 and a device ID of 0xAD. *
163
164
165         mov     al, 90h                 ; send out the 1-2-3rd bus
166         call    JEDEC_Cmd_8             ;  cycles as per data sheet
167
168         push    (PRGBASE/10h)
169         pop     ds
170         xor     si, si                  ; read from 0-1 and get
171         mov     ax, ds:[si]             ;  Manu and Device ID
172         mov     bx, ax                  ; save
173
174         MOV     DI, OFFSET Message7     ; 'Device: Mfg='
175         call    print
176         call    SerOut8
177
```

Line 137: we mask out the rightmost 3 hex digits of the BASE, as the ZF Logic hardware ignores these bits. That way our FlashBase variable agrees with the hardware.

Lines 150-151: Subroutine process_basket_header is in our source file RLE_MEMW.ASM (not printed but on the CD). It expects a parameter which is the 16-bit (OFFSET) address of where to go if it fails. The source file provides some top-level documentation of the header structure. If the header checks out, this procedure prints the message 'Source: ' and then the basket name. You can see the printout in Figure 4-9 Monitoring COM1. The name printed is the one we put into the Basket Header using the Z-Tag Manager. Variable "basketsize" is set by this routine. The ZTRead pointer is left at the start of the "payload" of the basket.

```
178
179  ;       if (MfgID != 1) fID_error = TRUE;
180
181          CMP     AL, 1                   ; hardwired for AMD
182          JE      endMANUtest
183          MOV     fIDError, TRUE
184  endMANUtest:
185
186          MOV     DI, OFFSET Message8     ; ' DevID='
187          call    print
188          shr     ax, 8
189
190  ;       if (DeviceID != 0xAD) fID_error = TRUE;
191
192          CMP     AL, 0ADH                ; hardwired for AMD 29F016D
193          JE      endIDtest
194          MOV     fIDError, TRUE
195  endIDtest:
196
197          call    SerOut8
198          call    CRLF
199
200          mov     al, 0F0h                ; reset command to
201          mov     ds:[si], al             ; end identification mode
202
203
204  ;       if (fIDerror)
205
206          CMP     fIDerror, TRUE          ; hardwired for AMD 29F016D
207          JNE     ChipSupported
208          LEA     DI, Message13           ; chip not supported
209          CALL    Print
210          JMP     ExitPgmrFail
211
212
213  ChipSupported:
214
215          ; ====== Basket handling begins here ======
216
217      COMMENT * Note, that in order to maintain correct checksum for basket, we
218      must calculate data CRC for RLE header as well!  Thus we resetCRC prior to
219      calling process_RLE_header but do not verify the checksum until after all
220      the basket data is read.  *
221
222          call    ResetCRC                ; BUR API routine prior to INT 17H
223          MOV     AX, OFFSET ExitPgmrFail
```

Line 179, 204: To verify that we are programming the correct chip, we check and printout out the mfg and device ID. If we don't find the specific chip we are looking for, we set fID_error TRUE and terminate.

Line 200: To check the chip/mfg ID, we executed a command sequence to put us in what AMD calls autoselect mode. You terminate that with a reset. Note that ds was set to point to our memory window in line 169

Line 222: The header of the RLE Compressed Basket (and the header of an uncompressed basket) contains a checksum for the basket data. We piggyback on the INT 17H call which was written into the BUR for YMODEM protocol checksum, and checksum the payload data as we read it in. First we reset it.

```
224          call    process_RLE_header      ; process RLE header if any
225          mov     ebp, BasketSize         ; total bytes to program
226
227  PrgLoop:                                ; Erase sector now
228          LEA     DI, message3            ; '0x'
229          CALL    Print
230          mov     eax, FlashBase
231          call    SerOut32                ; write AX as xxxx in hex
232
233          LEA     DI, message4            ; ': Erasing .. '
234          CALL    Print
235
236   ;  There are two 3-byte sequences to do a sector erase.
237
238          push    (PRGBASE/10h)
239          pop     fs              ; fs:0 points to our memory window
240          xor     si, si
241
242          mov     al, 80h         ; sector erase command part 1
243          call    JEDEC_Cmd_8
244          mov     al, 30h         ; sector erase command part 2
245          call    JEDEC_Cmd_8
246
247      comment * Erase is now in progress.  Check, when we are ready.  There is a
248      bur variable which is incremented 18.2 times/second.  182 times is a 10
249      second timeout.  *
250
251          call    DSBX2Var                ; DS:BX -> BUR data area (RAM)
252          mov     ax, 182
253          mov     ds:[bx.CountDown], ax   ; gives 10sec. timeout
254
255          mov     dx, ZFINDEX
256          mov     al, ZT_WRITE            ; ZT_SIG_OUT
257          out     dx, al                  ;
258          inc     dx                      ;
259          in      al, dx                  ;
260          and     al, 11110101b           ; turn off LED's
261          out     dx, al
262
263          call    ZTPrepareRead           ; ZFLogic back to track
264
265      comment * to test for erase, you only need to look at D7 which is forced to
266      a 0 until the erase is done. The FFFFFFF test below will accomplish the
```

Lines 224-225: Subroutine process_RLE_header will set variable BasketSize to the real (expanded value) if this is a compressed basket. We then store BasketSize in EBP as we will use that as a count-down register. This is a rather unconventional use of EPB, but we are not using stack frame parameter passing...

Lines 241-245: There are 6 bytes which need to be output to the AMDFLASH chip to cause an erase. Bytes 1-2 and 4-5 are the same. JEDEC_Cmd_8 will output two bytes and then take the third from the value passed in AL. See the AMD Data Sheet. The code is in flashpgm.asm. It uses PRGBASE to access the memory segment.

Line 251: **DSBS2Var** sets DS:BX to point to a common data area in the BUR. Included in this is a word variable which is decremented ever time tick, bx.CountDown. The ".CountDown" is a structure reference and provides in the assembler the proper OFFSET and TYPE for the variable.

```
267       same thing. *
268
269
270   @@:
271          cmp     word ptr ds:[bx.CountDown], 0   ; did we timeout
272          jz      @f
273          cmp     dword ptr fs:[si], 0FFFFFFFFh   ; D7 = 1 is a
274          jnz     @b                              ; loop until erase completes
275   @@:
276          mov     word ptr ds:[bx.CountDown], 0   ; reset timer, so we will not
lose our blinking LED's
277          cmp     dword ptr fs:[si], 0FFFFFFFFh   ; test again for branch
278          jz      @f
279
280          LEA     DI, message14   ; 'FAILED!'
281          call    print
282
283          jmp     ExitPgmrFail
284   @@:
285
286          ; All set. Programming ...
287
288          LEA     DI, message10           ; 'Programming .. '
289          call    print
290
291          mov     ecx, 64*1024
292          cmp     ecx, ebp                ; EBP is # of bytes left to program
293          jbe     @f
294          mov     ecx, ebp                ; if less than 64K left, program it
295   @@:
296          cmp     ecx, 0
297          jz      PgmDone                 ; don't do anything if basket size is 0
298
299          push    (PRGBASE/10h)
300          pop     ds
301          xor     si, si
302
303   BytePrgLoop:
304          mov     al, 0A0h                ; "program" command
305          call    JEDEC_Cmd_8
306
307          call    ZTRead_RLE              ; read and do INT 17 to maintain checksum
308
309          mov     [si], al                ; write byte
310          mov     bl, al                  ; save byte for future compare
311
312           ; on the extreme case we can have PCI backside clock 80Mhz. ISA
```

Lines 270-275: The @f and @b constructs save labels in that you can jump to the previous or next @@. In this case we go forward if we timeout and we go forward if we find a 32-bit FFFFFFFFH. When the flash chip erases, it sets all the bits to 1. Also, during the erase bit 7 of each byte is held at 0 to indicate that it is "working". So checking for FFFFFFFF is an OK way to test for erase completion. On line 277 we test again to see if we got there due to timeout or to completion.

Line 292: We carry the remaining number of bytes to program in EBP. That was initialized in line 225.

Lines 303-310: There is a sequence of four bytes you need to send out to program. The standard JEDEC two byte sequence, an A0H, and then the data. We have DS set in line 300 to our window, and SI is the OFFSET.

```
313           ; divider is 8, so we have 10M ISA bus clock. Programming cycle can be max.
314           ; 50us, so we need to wait here about 500 ISA cycles to kill that time.
315
316
317           mov     di, 500
318     WaitWriting:
319           cmp     ds:[si], bl
320           jz      ByteOk
321           in      al, 80h                 ; create one ISA cycle for delay
322           dec     di
323           jnz     WaitWriting
324
325           LEA     DI, Message14   ; 'FAILED!'
326           CALL    print
327           jmp     ExitPgmrFail
328
329     ByteOk:
330           inc     si                      ; advance address to next
331           loopd   BytePrgLoop
332     PgmDone:
333           LEA     DI, message15   ; 'OK!'
334           call    print
335
336           ; Anything left?
337
338           sub     ebp, 64*1024        ; if ( (ebp-64K) <= 0 ) go to @f
339           jz      @f
340           jc      @f
341
342       ; move to next page - note that CS0 and page size are fixed, but that
343       ; flashbase came in from the parameter record
344
345           add     FlashBase, 64*1024      ; update page to next
346           Call    Set_Flash_Offset        ; uses CS0, FlashBase
347
348           jmp     PrgLoop
349
350           ; Now when programming is done, go check the data checksum if it was RLE image
351
352           cmp     BasketType, 0FFh
```

Lines 317-331: Since we need a short timeout, we are counting ISA bus cycles. Frankly, you never get a timeout -- it always works. But we do have an upper limit here. To ascertain that we are done, we compare the data we are writing with the data we read. AMD suggests an alternative way -- to wait until a busy bit in the data field is done -- but this way seems to work just fine (and we do verify the data).

Lines 338-340: We may have programmed a partial sector, or we may have programmed a full 64K. If there was more than 64K in EBP we still have something left to go. So we subtract 64K and if EBP is 0 we are done. If EBP is negative we are done.

Lines 345-346: Every time we move forward 64K in the AMD, we need to move our ZF Logic Window to point to that block. Note that although erase is flash sector specific, programming is byte specific. So the fact that FlashBase did not start on a 64K boundary is not important. Thus FlashBase (and the ZF Logic Memory Window) need not be 64K aligned.

```
        jz      @f

        mov     eax, RLE_CheckSum
        cmp     eax, RLE_Chk            ; compare against calculated checksum
        jz      @f

        LEA     DI, message11   ; 'RLE basket checksum error '
        call    print

        call    SerOut32
        mov     al, ' '
        call    SerSend2
        mov     eax, RLE_Chk
        call    SerOut32
        call    CRLF

        jmp     ExitPgmrFail
@@:
        call    ZTPrepareRead

        ; Get original checksum

        call    ZTRead
        shl     ax, 8
        call    ZTRead
        xchg    al, ah

        call    DSBX2Var                ; get variables block to DS:BX
        cmp     word ptr ds:[bx.YModemCRChi_C], ax
        jz      CRCOK

        LEA     DI, message12   ; 'Data CRC failure: '
        call    print

        call    SerOut16
        mov     al, ' '
        call    SerSend2
        mov     ax, word ptr ds:[bx.YModemCRChi_C]
        call    SerOut16
        call    CRLF
        jmp     ExitPgmrFail
CRCOK:

        LEA     DI, message16   ; 'Data CRC was OK!'
        CALL    print
        jmp     ExitPgmrOk

ExitPgmrFail: ; Light up RED LED and do not do anything else!

        mov     dx, ZFINDEX
        mov     al, ZT_WRITE            ;ZT_SIG_OUT
        out     dx, al
        inc     dx
        in      al, dx
        and     al, 11110101b or      al, ZT_LED_RED          ;00001000b
```

```
        out     dx, al

        LEA     DI, message17   ; 'Programming Failure ...'
        CALL    print
        jmp     $

ExitPgmrOk:
        ; Light up GREEN LED and continue with BUR
        mov     dx, ZFINDEX
        mov     al, ZT_WRITE            ;ZT_SIG_OUT - ZTAGWRITE
        out     dx, al
        inc     dx
        in      al, dx
        and     al, 11110101b
        or      al, ZT_LED_GREEN        ;00000010b
        out     dx, al
ExitPgmr:
        call    CRLF

        ; Set timer to maximum value. This is useful to prevent the
        ; BUR from blinking with LED's when loading next commands.
        ; This way we maintain our RED or GREEN LED setting

        call    DSBX2Var
        mov     word ptr ds:[bx.CountDown], 0FFFFh

        ; Always exit with ZFL registers prepared for accelerated read!

        call    ZTPrepareRead
        retf                            ; resume with BUR


    ; in-line includes here so no "space" wasted due to segment combination

       include BURAPI.ASM      ; interface to the BUR
        include RLE_MEMW.ASM    ; common routines
        include FLASHPGM.ASM    ; flash programmer common routines
code    ends

END START
ENDS
```

The VxWorks demos currently load off the DOS partition. You may want to upgrade your DOS partition (see 'Using CD ROM Drive from DOS' on page 23), but that is not necessary for the demo to operate. You may also wish to set the default boot of your IDS to DOS rather than Linux. See 'Set the Boot Default to DOS' on page 23.

4.2.1. Running the Shell Demo

The demo software is pre-installed on the ZFx86 Integrated Development System in the DOS partition:

```
\ (root)
   |
   |____vxworks0
          |___vxload.com
          |___vxworks.st
          |___vx.bat
          |___bootrom.sys
```

Figure 4-12 VxWorks Shell Demo

To run the program, change to the vxworks0 directory and type vx. Here is what it does:

```
119     del \vxworks.*
120     copy vxworks.st \
121     vxload bootrom.sys
```

vx.bat in line 1 removes any previous vxworks.* file from the root (as both demos put their own version in the root). In line 2 it copies the vxworks.st image to the root, and in line 3 it invokes VxLoad. VxLoad is a DOS program which will load a VXworks executable from the DOS file system[1]. We use VxLoad to load bootrom.sys. In turn, bootrom.sys loads vxworks.st from the root of the hard disk. How does bootrom.sys know to do that? See the DEFAULT BOOT LINE just below!

If you get the error message the error message "Image memory is occupied - try to reduce system space" -- make sure that there is "nothing" in config.sys or autoexec.bat which will consume memory.

When you run the demo, VxWorks will come up in the shell. Once VxWorks is up, you should be able to ping the "e" address of 192.168.200.144. See the line below from VxWorks CONFIG.H file:

#undef DEFAULT_BOOT_LINE \
"ata=0,0(0,0)host:/ata0/vxWorks.st h=192.168.200.129 e=192.168.200.144 u=target tn=target pw=target o=elPci"

The shell command **hostShow** shows you your target and host IP addresses. They appear as above. You may use the following VxWorks shell control characters and commands. Note that the commands are case sensitive, and that with the help commands you need to finish the scroll list <CR> or quit them Q<CR>.

Table 4-7: VxWorks Shell

Command	Description
CTRL+C	Abort and restart the shell.
CTRL+X	Reboot (trap to the ROM monitor).
help	print list of shell commands
i	list current tasks
debHelp	print debugger help info
netHelp	print network help info

Besides the shell commands, we have provided the classic "hello world" task. In this case it is:

```
#include <stdio.h>
void charlie_task (void);
int sam;
void charlie_task (void) {
printf ("Hello There");
}
```

Figure 4-13 source file charlie.c

You can run this task from the shell by entering **sp charlie_task** -- and you can see the address of the symbol charlie_task by typing **lkup "charlie_task"** or **lkup "sam"**. You will

1. There are many ways of getting a VxWorks image into memory. In a typical target system, VxLoad is *not* the way to go. However, it serves our purpose nicely on the IDS.

note that charlie_task is a text symbol and sam is a bss (block starting with symbol) or data symbol. In the next section you can see how to rebuild (or modify and rebuild) these files.

4.2.2. Rebuilding the Shell Demo

The shell demo is simple in that the target files BOOTROM.SYS and VXWORKS.ST are both built from within the same directory. It is perhaps simplistic in that it does not show off the VxWorks Tornado Integrated Development Environment. That said, here's how to do it:

Install Tornado (we will use the compiler and editor, but not the IDE) on your host development system. Create a directory in the root called ataboot (you may use your own name). Then copy into that directory the contents of the IDS CD folder ataboot: VxWorks Demos DOS Bootable\VxWorks Shell Demo\ataboot.

Rebuilding BOOTROM.SYS

To rebuild bootrom.sys, execute the file makbootunc.bat. This file contains:

1 call \Tornado\host\x86-win32\bin\torvars
2 make clean
3 make bootrom_uncmp
4 copy bootrom_uncmp a:\bootrom.sys

In line 1 we execute torvars.bat which sets up the path for the Tornado tools. If you do this in a DOS window, after a while your environment string gets rather long as you keep calling torvars.bat.

Rebuilding VXWORKS.ST

To rebuild vxworks.st, execute the file makst.bat. This file contains the following items:

1 call \Tornado\host\x86-win32\bin\torvars
2 rem make clean
3 make vxWorks.st
4 copy vxWorks.st a:\

The makefile contains a macro (a define) for MACH_EXTRA as follows:

MACH_EXTRA = charlie.o # crc 08-09-00

This statement will cause charlie.c to be compiled and included in the vxworks image. You can also compile charlie.c by typing **make charlie.o** in the ataboot directory (once torvars has been called). You do not have to open the tornado IDE to do any of this.

4.3. VxWorks Menued Demo

The VxWorks Menued Demo features a “real” application program and also uses the Tornado IDE (project facility) to build the VxWorks image. The VxWorks demo program itself currently uses a text menu, but a future demo will make use of the Zinc Application to provide a graphics interface. When you run text-mode menu can select desired items and perform specific actions. Included are:

- '1. PING Test".
- '2. Net Receiver Test", '3. Net Sender Test", and '4. Net Loopback Test".
- '5. FTP Server Test".
- '6. Hard Disk Performance Test and 7. RAM-Disk Performance Test".
- Information about running tasks
- Stop running test processes
- Exit to VxWorks Shell

The user is able to run multiple demo instances or performance tests concurrently as separate tasks and thus see the performance impact to whole system. Not all items can be run as concurrent tasks, further features are described in the “Using the Demo Software” and “Test menu items in detail” chapters.

Required Target Hardware

The standard ZFx86 Integrated Development system is needed for running this VxWorks demo program. This includes:

- 3Com905TX 10/100 network card (required for network tests)
- Hard disk with 10-100 Mbytes of free space (required for HD performance tests)

Required Host Hardware/SW

You only need a host computer if you decide to modify the VxWorks Menued demo. To do this, you also need to use the Tornado Tools.

4.3.1. Running the Menued Demo

The demo software is pre-installed on the ZFx86 IDS in the DOS partition:

```
\ (root)
  └──vxworks1
       ├──vxload.com
       ├──vxworks
       ├──vx.bat
       └──bootrom.sys
```

Figure 4-14 VxWorks Shell Demo

To run the program, change to the vxworks1 directory and type vx. Here is what it does:

```
1    del \vxworks.*
2    copy vxworks \
3    vxload bootrom.sys
```

Using the Menued Demo

The source code of the demo follows.

When vxworks demo is successfully loaded by bootrom and successfully starts, it first asks user for the desired RAM-disk size. See Line 82 on page 54. Press Enter key to allocate 32 Mbytes for RAM-disk or enter any other amount in kilobytes. When the RAM-disk is created as “/RAMDISK” device, a corresponding message is displayed.

Then the clock watchdog is started which is used for measuring elapsed time during performance tests. In addition the hard disk is attached as “/ATA” device. Finally, the IDS VxWorks Demonstration Program main menu is presented to user. The items in menu are:

1. PING test
2. Net Receiver Test
3. Net Sender Test
4. Net Loopback Test
5. FTP Server Test
6. Hard disk performance test
7. RAM-disk performance test

i Show running tasks info
d Show info about available devices
s Stop running processes
q Exit to Shell

An “Enter option (h for help): “ - prompt is presented to user and user should select corresponding number or letter and press the Enter-key.

When some of the tests are executed, the output results will appear after certain intervals on the bottom of the screen. Other lines on screen will be scrolled up and finally off the screen. When multiple processes are running and user wants to start additional test-tasks or end some, then the user should just type in the appropriate number or letter followed by Enter.

The “**s**” menuitem allows you to stop all currently running network tests and performance tests.

The “**i**” menuitem shows info about currently running tasks and their status.

The “**d**” menuitem shows info about all defined devices (serial ports, RAM-disks, block devices) in the system.

The “**q**” menuitem ends the IDS Demonstration Program and exits to the VxWorks interactive Shell.

The IDS demo program can the be restarted only by rebooting the system. To do this, press CTRL-X at the shell prompt.

Demo Menu Items In Detail

1. PING Test

Ping is just meant for testing network connectivity between different machines. The user is prompted for the target computer's IP address and repetition count. The Ping to different machines can be executed multiple times and thus multiple Ping tasks are spawned in the VxWorks environment. The Ping test is able to run concurrently with all other test items. The Ping test cannot be terminated by pressing “s” in user menu so please be cautious with entering the ping repetition count.

2. Net Receiver Test

This item starts a network listener task for a certain TCP-IP network port on the test machine. The user is prompted for a port number. When the sender task is also executed somewhere in the network for this IDS computer and directed to the same port, then the listener task prints out a network transfer rate every 10 seconds. There is no output in the case when there is no network traffic. The network sender task can be launched using the menu item “3”.

You may run multiple receiver tasks for different port numbers on the same machine.

The receiver tasks can be running concurrently with all other test item tasks.

3. Net Sender Test

This starts a network sender task which sends packets to certain destination machine's certain port. The machine IP address and port number are asked from user. When there is no network receiver task launched on target computer for the same port number then the sender task will also exit with corresponding error message (connect failed).

There can be running multiple sender tasks on the same machine with different target IP-s or even for different ports on the same machine.

The sender tasks can be running concurrently with all other test items except FTP server test.

When sender task is running on the IDS machine for example, then FTP file transfer from remote host to this IDS machine is not possible.

4. Net Loopback Test

This menu item is actually a combination of two previous items. It prompts user for a desired port number and then starts both the network receiver task and sender task on the same machine (actually for IP address 127.0.0.1 which is localhost) for the same port. There can be also multiple concurrent network loopback tests running in the system and they can be running concurrently with all other test items except FTP server. When the sender task is running on this machine, then FTP file transfer from remote host to this IDS machine is not possible.

5. FTP Server Test

The FTP server is actually running on the system as soon as the demo is started. When you select “5” from the main menu, instructions are displayed which describe how to do a FTP file transfer from a remote machine to this test machine's RAMDISK.

During network loopback tests, net sender tests, or disk performance tests, the FTP server tasks do no respond because of lower priority of FTP Server tasks.

Instead of the proposed “/RAMDISK” directory the user can also do a “cd /ATA” on remote computer's FTP client prompt and thus transfer a test file also to IDS test machine's hard disk.

6. Hard Disk Performance Test and
7. RAM-Disk Performance Test

The menu items “6” and “7” use actually the same subroutine for performing disk access transfer rate measuring, only in case of “6” the test files will be created on hard disk and in case “7” the test files are created on RAM-disk.

The user is prompted for a test file size in kilobytes and the test repetition count.

After each read or write cycle to the target device the read or write transfer rate is displayed on screen.

There can be multiple simultaneous disk transfer tests running and they can run simultaneously with all test items except FTP test. During disk tests the FTP server tasks are in a “pending” state because of their lower priority.

4.3.2. Building Menued Demo Software

The demo software itself includes binaries and source code for building needed Board Support Package (bootrom.sys) and for building demo program (vxworks) using the WindRiver Tornado 2.0 IDE. (In part 4.1, we used the compiler and other GNU tools installed with the IDE, but we did not use the IDE.)

Rebuilding BOOTROM.SYS

BOOTROM.SYS and VXWORKS are built from different directories. BOOTROM.SYS is built from the DOS command line (as in the previous demo), but VXWORKS is built using the Tornado Project Facility. That said, here's how to do it:

Install Tornado (we will use the compiler and editor, but not the IDE) on your host development system. Create a directory in the root called ataboot1 (you may use your own name). Then copy into that directory the contents of the IDS CD folder ataboot1: VxWorks Demos DOS Bootable\VxWorks Menued Demo\ataboot1.

To rebuild bootrom.sys, execute the file makboot.bat. This file contains:

1 call \Tornado\host\x86-win32\bin\torvars
2 make clean
3 make bootrom_uncmp
4 copy bootrom_uncmp bootrom.sys

Before building BOOTROM.SYS, you may set appropriate IP addresses for host and target in \config.h. There in config.h are defined in multiple boot lines, edit the one which is not commented (undef) out. If you want to use the over-the-net boot, comment the ata boot line and uncomment in the net boot line.

Copy the new bootrom.sys to the IDS hard disk into directory VxWorks1.

Rebuilding VXWORKS

Build vxworks with Tornado Project facility. The first step is to copy some files from the IDS CD to your host system. Look on the IDS CD for VxWorks Demos DOS Bootable\VxWorks Menued Demo\Tornado\target\proj. Copy the files from the ...Tornado\target\proj folder on the CD to the Tornado\target\proj folder on your host. Then click on the wsp file.[1]

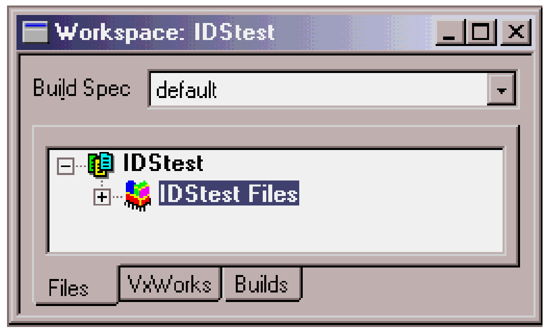


Figure 4-15 IDStest: Files

You can expand the file list with the [+] key, and if you subsequently click right on any file you get a menu of actions you can perform on the file. This is called a "context" menu in that it represents things that you might want to do in the current "context". See Figure 4-16.

1. If you hit F1 and get the Tornado help it says: "The workspace window divides your project information into three categories: Files, VxWorks, and Builds. Move between the three categories by using the tab controls at the bottom of the workspace window.
The Files view displays information about the files associated with the projects in the workspace.
The VxWorks view displays information about the operating system components that may be included in VxWorks or bootable application projects. This view is empty for downloadable application projects.
The Builds view displays information about the builds specifications defined for projects in the workspace."

Tornado Build →Rebuild All

If you click “Build” on the Tornado Menu Bar, you will see a pull-down which also allows you to rebuild the project.

rior to a build, you can go to the VxWorks tab of the Workspace:IDStest panel and expand the list to see which components of VxWorks are included in the build. You will note that the ATA hard drive component is enabled, and that the IDE hard drive is not. If you click right on ATA hard drive, you will see the component properties. note that “macro” or include for this is INCLUDE_ATA. That is the philosophy of the project tool: the project tool sets the necessary includes in the configuration files for you, and it checks for dependencies.

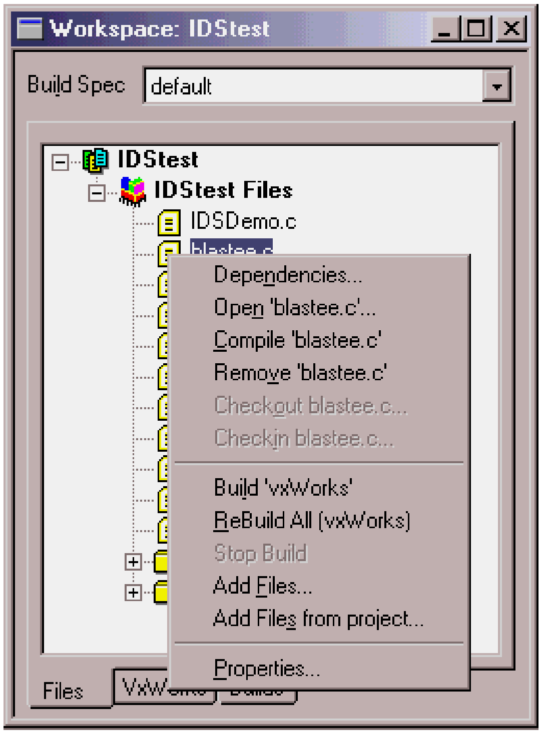


Figure 4-16 File Context Menu

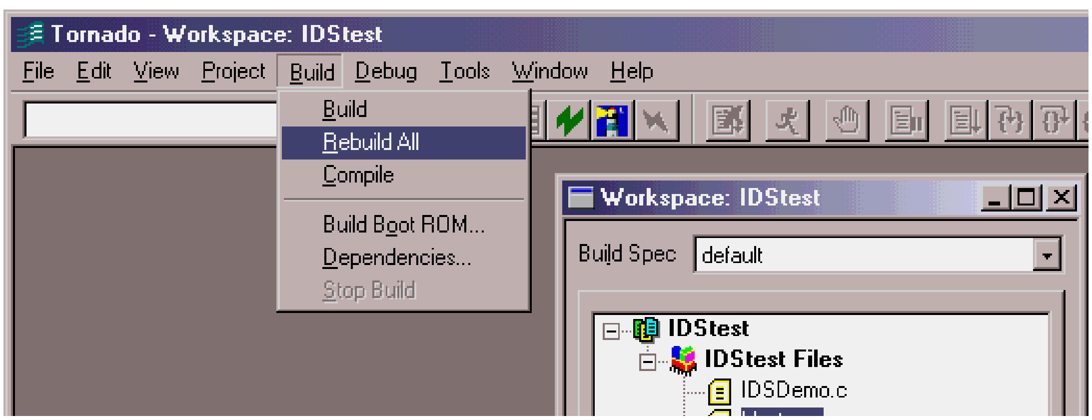
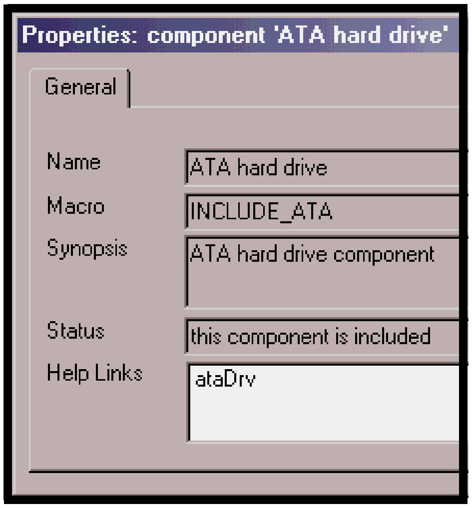


Figure 4-17 Tornado - Build - Rebuild All



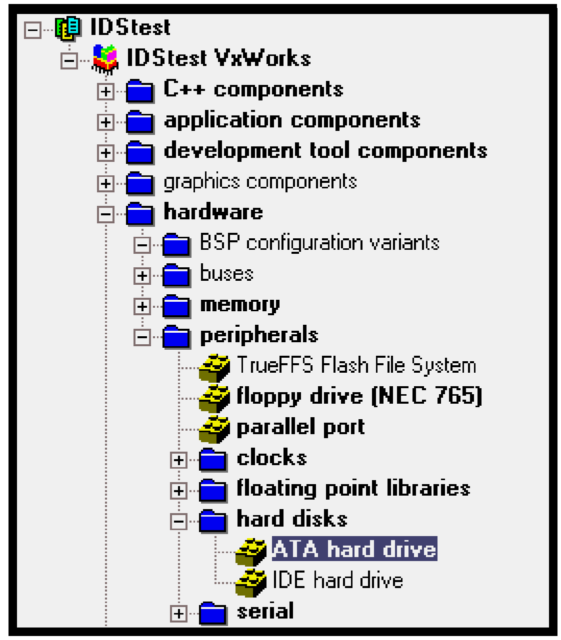


Figure 4-18 IDStest: VxWorks

If you execute Rebuild all, you will see the build output on the screen, and at the end of that you will see:

```
ld386 -X -N -e _sysInit -Ttext 00120000 \
dataSegPad.o partialImage.o ctdt.o symTbl.o -o vxWorks
C:\TORNADO\host\x86-win32\bin\vxsize 386 -v 00020000 00120000 vxWorks
vxWorks: 705920(t) + 85912(d) +  34012(b) = 825844
Done.
```

In this example, the total size of VxWorks is 825,844 bytes comprised of (t) text, (d) data, and (b) bss. Bss represents uninitialized data and stands for block starting with symbol.

If you now look into \Tornado\target\proj\IDStest\default, you will find the object files created and the vxWorks file. Copy the vxWorks file, which is a vXworks image, to the vxworks1 directory of your IDS.

Optional Network Boot Feature

Many Wind River developers set up their BOOTROM.SYS so that it will get the vxworks image from the host over the LAN. In that way, you do not have to copy the VxWorks file to the target via a floppy.

If you built your bootrom.sys for network boot, then after the bootrom.sys is executed, it automatically loads the vxworks file from host computer over the network and then launches it on the IDS. In order the loading to succeed, the

Wind River “FTP server” must be running on your host computer. The “FTP server” software is installed along with Tornado 2 installation and it can be found in the Start menu under "Programs\Tornado2". Run the FTP server and check the “User Rights” option under the “Security” menu. Add a new user named “target” with password “target” and set it's home directory to be D:\TORNADO\TARGET\PROJ\IDSTEST\DEFAULT where d:\tornado is your Tornado2 main installation directory.

Make sure that before building a bootrom.sys for network booting also the vxworks-file path corresponds fully to your build path. The example network boot line in config.h is:

#define DEFAULT_BOOT_LINE "elPci(0,0)host:/tornado/target/proj/idstest/default/vxWorks h=192.168.100.34 e=192.168.100.15 u=target tn=target pw=target o=elPci f=8"

In this case the path to vxworks is /tornado/target/proj/idstest/default/vxWorks. The “h” parameter in boot line describes the remote host's (the development host) IP address and “e” parameter defines the demo-machine's IP address.

4.3.3. IDS Menu Demo Main Source File IDS Demo.C

```
/* this IDS Demo software main body file

Version History
0.0.1  14.08.00 RaJ  First Draft

*/
#include "stdio.h"
#include "taskLib.h"
#include "pingLib.h"
#include "shellLib.h"
#include "kernelLib.h"
#include "usrLib.h"
#include "wdLib.h"

int makeRamDisk (int sizeK, char * RDname);
int blastee(int port, int size, int blen);    /* port,2000,16000 */
int blaster(char * destAddr,int port, int size, int blen);    /* port,2000,16000 */

int DiskRWTest(char * diskname,int fsize,int repcnt);

extern int blasteeStop;
extern int blasterStop;

#define BUF_SIZE 1024

int writeNum=0;
int testfilecnt=0;

int exitflag=FALSE;
int stopflag=FALSE;

unsigned long clkticks=0;

```

```
WDOG_ID      clkWd=NULL;
void clkWdFn(int);


/*********************************************************************************/
void demoHelp(void)
   {
   printf("\n*******************************************************************\n");
   printf("IDS VxWorks Demonstration Program v.1.0 by RaJ %s %s\n",__DATE__,__TIME__);
   printf("*******************************************************************\n\n");

   printf("      1. PING test\n");
   printf("      2. Net Receiver Test\n");
   printf("      3. Net Sender Test\n");
   printf("      4. Net Loopback Test\n");
   printf("      5. FTP server test\n");
   printf("      6. Hard disk performance test\n");
   printf("      7. RAM-disk performance test\n\n");
   /*
   printf("      8. Graphics demo\n");
   printf("      9. Internet Browser demo\n\n");
   */

   printf("      i  Show running tasks info\n");
   printf("      d  Show info about available devices\n");
   printf("      s  Stop running processes\n");
   printf("      q  Exits to Shell\n\n");
   }

/*********************************************************************************/
int IDSDemo(void)
   {
   int j,pnum;
   int ret,filesize,rdsize=0,repcnt;
   int taskcnt=0,tid;
   char ch;
   char buf[255];
   char nbuf[80];
   char abuf[80];
   char sourcedisk[80];

   /* enable round-robin scheduling */
   kernelTimeSlice(1); /* 1 tick per task */

   /* create RAM-DISK */
   for(j=0;j<3;j++)
      {
                  printf("Enter desired RAM-disk size in kbytes (press Enter for 32M): ");
      gets(buf);
      if(buf[0]==0)
         rdsize=32768;
      else
         sscanf(buf,"%d",&rdsize);
      ret=makeRamDisk(rdsize*1.2,"RAMDISK");
      if(ret==0)
```

```
            break;
        /* else try again for 2 times, then fail */
                    }

    if (clkWd == NULL && (clkWd = wdCreate ()) == NULL)
                    {
                    printf ("cannot create CLK watchdog\n");
                    exit (1);
                    }

    sysClkRateSet(100);
    wdStart (clkWd, 1, clkWdFn, clkWd);
    printf("Clock watchdog started, rate set to 100 ticks/sec.\n");

    /* connect to ATA disk */
    ret=usrAtaConfig(0,0,"/ATA");
    if(ret!=OK)
        {
        printf("Attaching to ATA disk failed!\n");
        }
    else
        {
        printf("ATA disk attached OK.\n");
        }

    /* kill possibly running shell task */
    /*
    tid=taskNameToId("tShell");
    taskDelete(tid);
    */

    demoHelp();

    /* stay in demo as long as needed */
    while(exitflag==FALSE)
        {
        printf("\nEnter option (h for help): ");
        gets(buf);
        ch=buf[0];

        /* now begin selection processing */
        switch(ch)
            {
            case 'h':
            case 'H':
                                    demoHelp();
                break;

            case '1':
                printf("\n**************** PING TEST ********************\n");
                printf("Enter IP to ping (press Enter for 127.0.0.1): ");
                gets(buf);
                if(buf[0]==0)
                    {
                    sprintf(buf,"127.0.0.1");
```

```
            }

        printf("Enter number of times to ping (press Enter for 10): ");
        gets(abuf);
        if(abuf[0]==0)
            {
            pnum=10;
            }
        else
            {
            sscanf(abuf,"%d",&pnum);
            }

        taskcnt++;
        sprintf(nbuf,"task%d",taskcnt);
        taskSpawn(nbuf,150,0,10000,ping,buf,pnum,0,0,0,0,0,0,0,0);
        printf("Ping task running for %d times...\n",pnum);
        break;

    case '2':
        printf("\n************* Net Receiver Test ***************\n");
        printf("Enter desired PORT nr (press Enter for 2000): ");
        gets(buf);
        if(buf[0]==0)
            {
            pnum=2000;
            }
        else
            {
            sscanf(buf,"%d",&pnum);
            }

        taskcnt++;
        sprintf(nbuf,"task%d",taskcnt);
        taskSpawn(nbuf,200,0,10000,blastee,pnum,2000,16000,0,0,0,0,0,0,0);
        printf("Receiver task running...\n");
        break;

    case '3':
        printf("\n************* Net Sender Test *****************\n");
        printf("Enter desired PORT nr (press Enter for 2000): ");
        gets(buf);
        if(buf[0]==0)
            {
            pnum=2000;
            }
        else
            {
            sscanf(buf,"%d",&pnum);
            }

        printf("Enter receiver machine's IP address (press Enter for 127.0.0.1): ");
        gets(buf);
        if(buf[0]==0)
            {
```

```
            sprintf(buf,"127.0.0.1");
            }

        taskcnt++;
        sprintf(nbuf,"task%d",taskcnt);
        taskSpawn(nbuf,200,0,10000,blaster,buf,pnum,2000,16000,0,0,0,0,0,0);
        printf("Sender task running...\n");
        break;

    case '4':
        printf("\n************* Net Loopback Test ***************\n");
        printf("Enter desired PORT nr (press Enter for 2000): ");
        gets(buf);
        if(buf[0]==0)
            {
            pnum=2000;
            }
        else
            {
            sscanf(buf,"%d",&pnum);
            }

        taskcnt++;
        sprintf(nbuf,"task%d",taskcnt);
        printf("Starting receiver for port %d...\n",pnum);
        taskSpawn(nbuf,200,0,10000,blastee,pnum,2000,16000,0,0,0,0,0,0,0);

        sprintf(buf,"127.0.0.1");

        taskcnt++;
        sprintf(nbuf,"task%d",taskcnt);
        printf("Starting sending to %s:%d\n",buf,pnum);
        taskSpawn(nbuf,200,0,10000,blaster,buf,pnum,2000,16000,0,0,0,0,0,0);
        printf("Loopback test running...\n");
        break;

    case '5':
        printf("\n************* FTP Server Test *****************\n");
        printf(" FTP Server is already running, maximum space in /RAMDISK is %d kbytes.\n\n",rdsize);
        ifAddrGet("elPci0",buf);
        printf(" This machine's IP address is %s\n\n",buf);
        printf(" Establish a FTP connection from remote machine to this \n \
               test machine by issuing 'FTP %s' command from remote machine's \n \
               command line. Enter 'target' as user and 'target' as password.\n \
               If successfully logged in, issue commands 'bin' to set binary \n \
               transfer mode and 'cd /RAMDISK' to set target directory on this \n \
               test machine.\n",buf);
        printf(" Use 'lcd local_needed_dir' command to set remote machine's working \n \
               directory, use 'put file_name' to send files from remote machine to \n \
               this test machine and use 'get filename' to transfer files from this \n \
               test machine to remote one. Use 'bye' to log out, 'dir' to get info \n \
               about files in test machine's RAMDISK.\n");
        break;

    case '6':
```

```
        printf("\n******** Hard-disk Performance Test ***********\n");
        printf("Enter size of test file in kbytes (press Enter for 10M): ");
        gets(buf);
        if(buf[0]==0)
            {
            filesize=10*1024;
            }
        else
            {
            sscanf(buf,"%d",&filesize);
            }

        printf("Enter number of repetitions (0=forever, press Enter for 10): ");
        gets(buf);
        if(buf[0]==0)
            {
            repcnt=10;
            }
        else
            {
            sscanf(buf,"%d",&repcnt);
            }

        testfilecnt++;

        sprintf(nbuf,"tstRW%d",testfilecnt);
        taskSpawn(nbuf,200,0,10000,DiskRWTest,"ATA",filesize,repcnt,0,0,0,0,0,0,0);
        printf("Hard-disk performance test running...\n");
        break;


    case '7':
        printf("\n******** RAM-disk Performance Test ************\n");
        printf("Enter size of test file in kbytes (max %d, press Enter for 32M): ",rdsize);
        gets(buf);
        if(buf[0]==0)
            {
            filesize=32*1024;
            }
        else
            {
            sscanf(buf,"%d",&filesize);
            }


        printf("Enter number of repetitions (0=forever, press Enter for 10): ");
        gets(buf);
        if(buf[0]==0)
            {
            repcnt=10;
            }
        else
            {
            sscanf(buf,"%d",&repcnt);
            }
```

```
            testfilecnt++;

            sprintf(nbuf,"tstRW%d",testfilecnt);
            taskSpawn(nbuf,200,0,10000,DiskRWTest,"RAMDISK",filesize,repcnt,0,0,0,0,0,0,0);
            printf("RAM-disk performance test running...\n");
            break;

                  case 's':
                  case 'S':
            printf("\n******** Stopping Running Processes ***********\n");
                                    blasteeStop=TRUE;
            blasterStop=TRUE;
            stopflag=TRUE;
            break;

                  case 'q':
                  case 'Q':
            printf("\n*********** Exiting to Shell ******************\n");

                exitflag=TRUE;
            stopflag=TRUE;

            blasteeStop=TRUE;
            blasterStop=TRUE;
            shellInit(10000,TRUE);
            break;

          case 'i':
          case 'I':
            printf("\n******** Info about running tasks *************\n");
            i(0);
            break;

          case 'd':
          case 'D':
            printf("\n********** Info about devices ****************\n");
            devs();
            break;

          default:
                                    break;
          }
        } /* while(not exit) */

    wdCancel(clkWd);
    wdDelete(clkWd);

    return(0);
    }

/**********************************************************************/
void clkWdFn (int parm)
    {
    clkticks++;
```

```

   if(clkticks%6000==0)
      logMsg("Uptime %ld minutes.\n",clkticks/6000);

   if(exitflag==FALSE)
      {
      wdStart (clkWd, 1, clkWdFn, 0);
      }
   else
      {
      wdCancel(clkWd);
      wdDelete(clkWd);
      logMsg("Clock WDOG stopped.\n");
      }
   }


/*********************************************************************/
int DiskRWTest(char * diskname,int fsize,int repcnt)
   {
   FILE * fp;
   char fname[80];
   int i,j,ret,kbytes;
   unsigned long startticks;
   float transfer;
   char *buf1;
   char *buf2;

   stopflag=FALSE;

   if(repcnt==0)
      repcnt=2000000000;

   sprintf(fname,"/%s/t%07d",diskname,clkticks);

   buf1=(char *)malloc(2048);
   if(buf1==NULL)
      {
      printf("Memory allocation for buf1 failed!\n");
      return(-1);
      }

   /* initialize 1K buf */
   for(j=0;j<1024;j=j+8)
              {
              sprintf(buf1+j,"%07ld",j);
              }

   buf2=(char *)malloc(2048);
   if(buf2==NULL)
      {
      printf("Memory allocation for buf2 failed!\n");
      free(buf1);
      return(-1);
      }
```

```
writeNum=0;

for(i=0;i<repcnt;i++)
   {
   rm(fname);

   startticks=clkticks;

   if(stopflag==TRUE)
      break;

   fp=fopen(fname,"wb");
   if(fp==NULL)
      {
      printf("\n%s open for write failed!\n",fname);
      free(buf1);
      free(buf2);
      return(-1);
      }

   for (kbytes=0;kbytes<fsize;kbytes++) /* kbytes loop */
      {
      if(stopflag==TRUE)
         break;

                  ret=fwrite(buf1,BUF_SIZE,1,fp);
                  if(ret<1)
                                    {
                                    printf("\nWrite to %s error!\n",fname);
         fclose(fp);
         free(buf1);
         free(buf2);
         return(-1);
                                    }

      writeNum+=BUF_SIZE;

      } /* for fsize of kbytes */

   fclose(fp);

   if((clkticks-startticks)>0)
      transfer=((float)fsize*1024)*(float)sysClkRateGet()/(float)(clkticks-startticks);
   else
      transfer=0.0;
   printf("%s WRITE test %d done: %10.0f bytes/sec\n",fname,i+1,transfer);

   /********** now reading test ****************************/
   startticks=clkticks;

   if(stopflag==TRUE)
      break;

```

```
        fp=fopen(fname,"rb");
        if(fp==NULL)
          {
          printf("\n%s open for read failed!\n",fname);
          free(buf1);
          free(buf2);
          return(-1);
          }

        for (kbytes=0;kbytes<fsize;kbytes++) /* kbytes loop */
          {
          if(stopflag==TRUE)
            break;

                ret=fread(buf2,BUF_SIZE,1,fp);
                if(ret<1)
                        {
                        printf("\n%s reading error at %d kbytes!\n",fname,kbytes);
            fclose(fp);
            free(buf1);
            free(buf2);
            return(-1);
                        }

          } /* for fsize of kbytes */

        fclose(fp);

        if((clkticks-startticks)>0)
          transfer=((float)fsize*1024)*(float)sysClkRateGet()/(float)(clkticks-startticks);
        else
          transfer=0.0;
        printf("%s READ  test %d done: %10.0f bytes/sec\n",fname,i+1,transfer);

        }/* for repcnt */

      free(buf1);
      free(buf2);

      rm(fname);

      printf("DiskRWTest for %s ended.\n",fname);
      return(0);
    }
```

ZF Micro Devices, Inc.

1052 Elwell Court

Palo Alto, California 94303

(650) 965-3800 · Fax 965-4050

www.zfmicro.com