MDINK32/DINK32 User's Guide

Interactive Debugger for PowerPC Microprocessors

Motorola
RISC Applications

Release Date: November 30, 1999
Updated: December 6, 1999
Version 12.0
Revision 0.0

Altivec Enabled

MOTOROLA MDINK32/DINK32 Version 12.0
User's Guide

Chapter 1 DINK32 User’s Guide Index

Chapter 2
Introduction

DINK is an acronym for Demonstrative Interactive Nano Kernel.

DINK32 is a flexible software tool enabling evaluation and debugging of the PowerPC 32-bit microprocessors. The introduction of the PowerPC microprocessor architecture provided an opportunity to create an interactive debugger independent from previous debug monitors. Since the family of PowerPC microprocessors spans a wide market range, DINK32 has to be extensible and portable, as well as being specific enough to be useful for a wide variety of applications. It is designed to be both a hardware and software debugging tool. DINK32 was written in ANSI C and built with modular routines around a central core. Only a few necessary functions were written in PowerPC assembly. This document describes the DINK32 software, the DINK32 command set, utilities, user program execution, errors and exceptions, and restrictions.

MDINK32 (Minimal DINK32) is a limited version of DINK32. It's major purpose is to download versions of DINK32 to the board. Currently, MDINK32 is only available on Excimer and Maximer boards. MDINK32 is supplied with the board. It is burned into sector A15, which is protected. The user can obtain new executable versions of DINK32 from the web site and download them onto the Excimer and Maximer board via MDINK32. New versions of MDINK32 are only available by returning the board to Motorola for an MDINK32 upgrade or building it from the source code.

Chapter 3 MDINK32/DINK32 Features

The MDINK32/DINK32 software package provides:

- Supports the MPC601, MPC603, MPC603e, MPC604, MPC604e, MPC740, MPC750, and the MPC7400.
- Modification and display of general purpose, floating point, altivec, and special purpose registers.
- Assembly and disassembly of PowerPC instructions for modification and display of code.
- Modification, display, and movement of system memory.
- A simplified breakpoint command, allowing setting, displaying, and removing breakpoints.
- Single-step trace and continued execution from a specified address.
- Automatic decompression of compressed s-record files while downloading
- Extensive on-line help.
- Ability to execute user-assembled and/or downloaded software in a controlled environment.
- Logging function for generating a transcript of a debugging session.
- Register set includes all of the PowerPC implementation specific registers.
- Modification of memory at byte, half-word, word and double-word lengths.
- Extensive support for the MPC 60x, MPC 740, MPC750, MPC7400 simplified or extended mnemonics during assembly and disassembly of PowerPC instructions.
- Ability to input immediate values to the assembler as binary, decimal, or hexadecimal.
- Command line download functionality that allows the user to select the download port and then send the data.
- An assembler and disassembler that understands branch labels and the ability to see and clear the branch table that DINK32 is using while assembling and disassembling PowerPC instructions.
- Ability to read and write MPC106 configuration registers. (Not supported on Excimer and Maximer).
- Support for PCI with new “pci-” commands. (Not supported in minimal builds, i.e. Excimer and Maximer).
- Support for Excimer and Maximer flash, fl –dsi and –se, and automatically detect flash on Revision 2 versus 3 of the board. fl -dsi has been expanded to display the memory range for each sector.

- Support for Excimer and Maximer flash, fl -sp and -su.
- Support for Max chip and altivec registers and instructions.
- Support for Kalua chip.
- Support for MPC107 Memory bridge.
- Support for dynamically assigned dink function addresses for downloaded programs, see Appendix G, “Dynamic functions such as printf".

3.1 MDINK32 Overview

The following sections describe the MDINK32 methodology and limited command set., the minimum required hardware configuration, and the memory model. MDINK32 is only available with the Excimer and Maximer platform. The current release of MDINK32 is Version 12.0.

3.2 New features for MDINK32 V12.0

No new functionality.

3.3 MDINK32 Design Methodology

The MDINK32 program’s only purpose is to download DINK32 programs. MDINK32 is loaded at 0xfff00000 and begins execution at 0xfff00100. It’s limited command set is designed to allow easy loading of DINK32 or other programs into FLASH or ROM memory and starting those programs.

3.4 Hardware Configuration Requirements

This MDINK32 software package can be executed on the same microprocessor boards that support DINK32, which include the following devices and minimum memory configuration:

- PowerPC™ 601, 603(e), 604(e), 740/750, MPC7400 microprocessors
- National Semiconductor PC87308 DUART (Yellowknife and Sandpoint Reference Design).or National Semiconductor 16552 DUART (Excimer and Maximer Minimal Evaluation Board)
- 512 K-byte EPROM or Flash
- 512 K-byte RAM

3.5 MDINK32 Software Build Process

MDINK32 can be built from the dink source base. Information for building MDINK32 is given in the DINK32 build section. There is only one version of mdink32 for all Excimer

and Maximer boards. Flash memory is automatically detected.

3.6 MDINK32 Memory Model

See Figure 3-3., “MDINK32/DINK32 Memory Model - Excimer and Maximer".

The following sections describe the DINK32 design methodology, the minimum required hardware configuration, and the memory model. The current release of DINK32 is Version 12.0.

3.7 New features for DINK32 V12.0

1. Detects MPC107.
2. Added makefiles for the GNU gcc compiler in every directory.
3. New commands: env, tau.
4. Support for dynamically assigned dink function addresses for downloaded programs, see Appendix G, “Dynamic functions such as printf".
5. Improved printf formats including floating point displays.
6. Quiet mode on many register displays.
7. Shared memory between host and agent targets using the Address Translation Unit (ATU).

3.8 DINK32 Design Methodology

The modular design of the DINK32 program, its extensive commenting, and its design methodology enable efficient user modification of the code. Thus, DINK32 provides a flexible and powerful framework for users who desire additional functionality.

Hardware Configuration Requirements

This DINK32 software package can be executed on microprocessor boards that include the following devices and minimum memory configuration:

- PowerPC™ 601, 603(e), 604(e), 740/750, 7400 microprocessors
- National Semiconductor PC87308 DUART (Yellowknife and Sandpoint Reference Design). or National Semiconductor 16552 DUART (Excimer and Maximer Minimal Evaluation Board)
- 512 K-byte EPROM or Flash
- 32 M-byte RAM

3.9 DINK Software Build Process

There are two types of platforms.

1. YellowKnife and Sandpoint. DINK32 is loaded at 0xfff00000. The config.h file must set the RESET_BASE macro to RESET_BASE_OTHERS as shown in Table 3-1., “RESET_BASE value"

Table 3-1. RESET_BASE value

Macro Name	Value
RESET_BASE_OTHERS	0xFFF0 (default)
RESET_BASE_EXCIMER	0xFFC0

2. Excimer and Maximer. The config.h file must set the RESET_BASE macro to RESET_BASE_EXCIMER as shown in Table 3-1., “RESET_BASE value"

DINK32 is a sophisticated debug ROM program. Most hardware specific features such as the specific PowerPC processor, the memory map, the target platforms, etc. are automatically detected at run time. This flexibility allows a single version of DINK32 to run on different platforms with different processors; for example the same version of DINK32 will boot the Yellowknife X2 platform with memory map A, the Yellowknife X4 platform with memory Map B, the Sandpoint, as well as the Excimer and Maximer platforms with all the supported PowerPC processors.

The ROM device on the Yellowknife and Sandpoint system is the Plastic Leaded Chip Carrier (PLCC) device. Upgrading the firmware on such system could be as easy as removing and replacing the old ROM with the new one. The ROM devices on the Excimer and Maximer platform however are the thin small surface mount packages (TSSOP). It is not easy to remove such devices on the target hardware for upgrading. To solve this problem, Motorola provides a smaller version of DINK32 called MDINK. The main purpose of mdink is to download DINK32 or other boot program to ROM, thus it provides a robust way for upgrading the firmware.

There are two different versions of DINK:

1. DINK32 provides the capability to download and debug application programs,
2. MDINK32 provides the capability to download and upgrade firmware.

Only DINK32 is available in executable form. It is delivered in the following eight file formats as shown in Table 3-2., “DINK32 File Formats"

Table 3-2. DINK32 File Formats

Board	S record	S Record (-g)	elf	elf/dwarf (-g)
Yellowknife and Sandpoint	dinkyk.src	dinkyk_g.src	dinkyk	dinkyk_g
Excimer	dinkex.src	dinkex_g.src	dinkex	dinkex_g

The source files can be used to build DINK32 or MDINK32.

The source files are *.c, *.s, and *.h.

Other files are makefile and READ_ME

Motorola uses the Metaware tool set to build MDINK32 and DINK32 in a UNIX environment. The syntax of the makefile, therefore, complies with the make program available on UNIX machines. The command to build DINK32 on a UNIX command line is "make dink", and the command to build MDINK32 is "make mdink".

MDINK32 is a subset of DINK32. Both versions share many source files. Of all the files that contribute to the making of MDINK32, the files that MDINK32 does not share with DINK32 is mpar_tb.c and mhelp.c. DINK32's version of mpar_tb.c is par_tb.c and mhelp.c is help.c.

Both can also be build on UNIX with the GNU gcc tool set using makefile_gcc, and on a PC/NT with the Metaware tool set using makefile_pc.

The source files and the makefile of DINK32 and MDINK32 reside in the same directory structure. However, the object files (*.o), the ELF file and S-record file of each version reside on a different directory. When the "make dink" command is executed, the "dink_dir" directory is created, and the output files produced by "make" are put in "dink_dir". Likewise, when the "make mdink" command is executed, the "mdink_dir" directory is created, and the output files are put in "mdink_dir" (see Figure 3-1).

In addition, the makefile, makefile_pc, is used to build on the PC (windowns) platform, and the makefile_gcc is used to build on UNIX with a GNU gcc compiler.

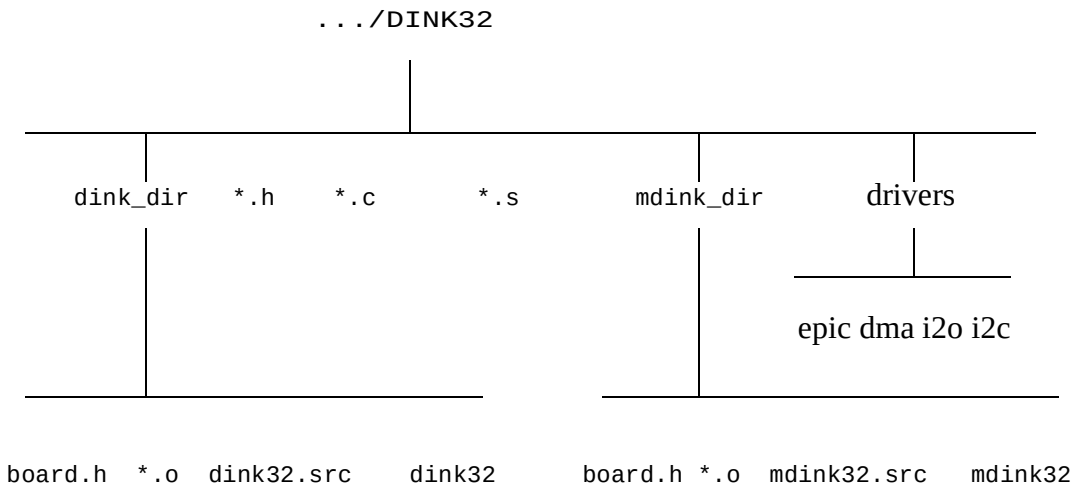


Figure 3-1. DINK32/MDINK32 Directory Organization

When compiling a version of DINK32 to upgrade an Excimer and Maximer board it is important to realize that this module, while relocatable, has a dependency that must be accounted for during compilation. Since, MDINK32 and DINK32 both copy themselves to RAM (and then execute from RAM) it is important to know which address range to copy from FLASH to RAM. If you are building an image which will be located at the reset vector (0xFFF00100) then the #define RESET_BASE (which is located in the config.h file) must be set to 0xFFF0. If, however, you are upgrading a version of DINK32 on an Excimer or Maximer board RESET_BASE should be changed to 0xFFC0 before building your new image. This S-record would then be loaded at address 0xFFC00000. This is the original configuration that came with the Excimer and Maximer board. The command to download a new version of DINK32 on an Excimer and Maximer board would be "dl -fl -o ffc00000" if there is nothing at location 0xffc00000. If replacing an older version then “fw -e” would be used to erase the version (and everything else that was not sector protected) in Flash. See Table 3-1., “RESET_BASE value".

3.10 DINK32 Memory Model

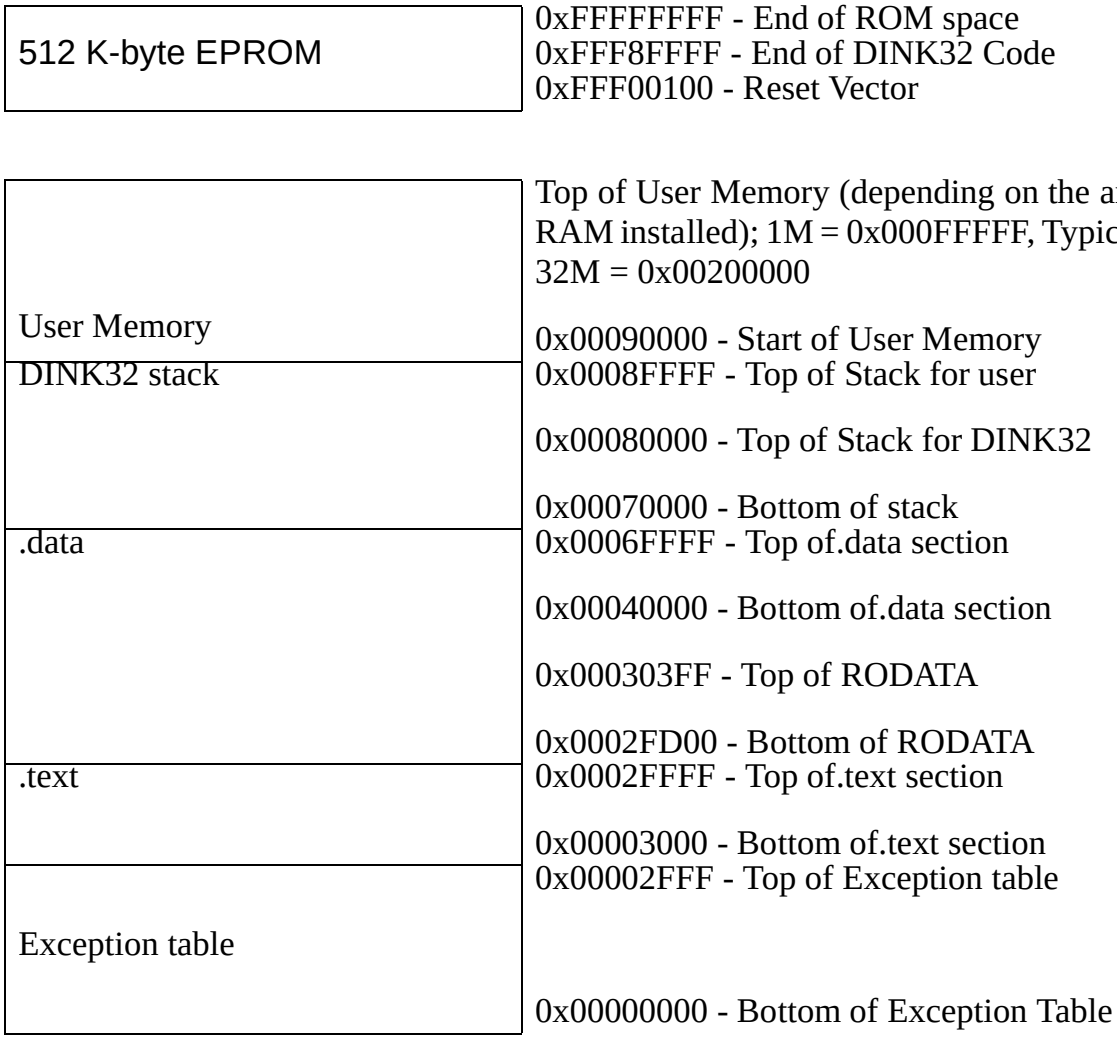
The memory model for DINK32 is shown in Figure 3-2., “DINK32 Memory Model -

Yellowknife and Sandpoint" or Figure 3-3., “MDINK32/DINK32 Memory Model - Excimer and Maximer". The exception vectors and exception code are located within address offsets 0x0000 - 0x2100. The DINK32 code through 0x80000 is copied from the EPROM to RAM so that the data structures can be modified at run time. For example, the data structures for the chip registers need to be modified when the “register modify” command is executed.

The EPROM must be located at address 0xFFF00000 because this is the beginning of the exception address space at system reset. The RAM must be located at address 0x00000000 since that is the low-memory exception address space, where the DINK32 code will be copied. Available user memory space begins at address 0x90000 and ends at the RAM’s upper boundary; address space below 0x90000 is reserved for DINK32.

DINK32 sets the stack pointer, r1, to 0x80000 for the C portion of the DINK32 code. DINK32 sets the user’s stack pointer, r1, to 0x8fff0. As long as the user, once started with a go or trace command, does not use more than 0xfff0 bytes for it’s stack there is no conflict with the stack used by DINK32.

Please reference Figure 3-2 and Figure 3-3 on the following page.



Note: The .text and .data sections are approximates depending on each build version. Actual locations can be ascertained from the xref.txt file in the dink_dir directory.

Figure 3-2. DINK32 Memory Model - Yellowknife and Sandpoint

System ROM

4 Meg Flash ROM	0xFFFFFFFF - End of ROM space
	0xFFF60000 - End of MDINK32 Code
MDINK32	0xFFF00100 - Reset Vector (MDINK32)
User Flash Space	0xFFEFFFFF - Top of User Flash Space
	0xFFC90000 - Bottom of User Flash Space
DINK32	0xFFC8FFFF - End of DINK32 Code
	0xFFC00100 - Start of DINK32 Code
	0xFFC00000 - Beginning of Flash space

System RAM

	Top of User Memory - 0x000FFFFF (1 Meg)
User Memory	0x00090000 - Start of User Memory
DINK32 stack	0x0008FFFF - Top of Stack for user
	0x00080000 - Top of Stack for DINK32
	0x00070000 - Bottom of stack
.data	0x0006FFFF - Top of .data section
	0x00040000 - Bottom of .data section
	0x00030000 - Top of RODATA
	0x0002FD00 - Bottom of RODATA
.text	0x0002FFFF - Top of .text section
	0x00003000 - Bottom of .text section
	0x00002FFF - Top of Exception table
Exception table	0x00000000 - Bottom of Exception Table

Note: The .text and .data sections are approximates depending on each build version.

Figure 3-3. MDINK32/DINK32 Memory Model - Excimer and Maximer

Chapter 4 MDINK32/DINK32 Commands

This chapter describes the DINK32 user commands. The full command mnemonic is listed in the upper left-hand corner and the short command (abbreviation) is listed next in smaller type. All commands listed (except fw -e) are available to DINK32, those commands available to MDINK32 are marked as MDINK32 Compatible.

Commands appear in boldface throughout this chapter.

Note: All addresses entered must be in hexadecimal ***but not*** preceded by "**0x**".

Leading zeros will be added as needed.

Definitions

"MDINK32 Compatible"

This command is also available in MDINK32. Where commands are different between MDINK32 and DINK32, the DINK32 format will be shown first.

"plus"

Usually implies that the command form includes "+". This allows the command to continue to the next stopping place appropriate for its functionality.

"range"

Indicates a two-address form, and usually signifies an inclusive area of code or memory that will be operated on by the command.

"entire family"

Refers to a family of registers. The general purpose registers are a family of thirty two 32-bit registers, numbered 0 to 31. The floating point registers are a family of thirty-two 64-bit registers, numbered 0 to 31. The altivec registers are a family of thirty-two 128-bit registers, numbered 0 to 31.The special purpose registers are not classified as a family due to their architectural design.

"x"

Typing "x" will exit a command if DINK32 is in an interactive mode when a particular command form is used.

4.1 Commands

4.1.1 .(period) .

repeat last command MDINK32 Compatible

Typing a period will repeat the last command entered.

Example:

```
DINK32_750 >> trace 2100
A Run Mode or Trace exception has occurred.

Current instruction Pointer: 0x00002104 stw r13, 0xfff8(r01)

DINK32_750 >> trace +
A Run Mode or Trace exception has occurred.
Current instruction Pointer: 0x00002108 add r03, r00, r01

DINK32_750 >> .
A Run Mode or Trace exception has occurred.
Current instruction Pointer: 0x0000210c mfspr r04, s0274

DINK32_750 >>
```

4.1.2 about about

(*M)DINK32 version information* *MDINK32 Compatible*

The version information for the current implementation of the DINK32 monitor will be displayed on the terminal.

DINK32 Example:

```
DINK32_MPC603ev >>about

A Reset Exception '0x100' initiated this restart
Caches Enabled: [ L1-ICache  L1-DCache ]

DDD   III  N   N  K  K   333    222
D  D   I   NN  N  K K   3   3  2   2
D   D  I   N N N  KK      33     22
D  D   I   N  NN  K K   3   3  22
DDD   III  N   N  K  K   333   22222  for MPC603ev

Metaware Build


Version 12, Revision 0

Written by : Motorola's RISC Applications, Austin, TX
  Released : November 30, 1999:
    System : Welcome to Excimer. A Minimum System PowerPC Design!
 Processor : MPC603ev V12.1 @ 133 MHz, Memory @ 66 MHz

Copyright Motorola, Inc. 1993, 1994, 1995, 1996, 1997, 1998, 1999

Changes for each release, Errata for dink, Future Enhancements
and bug fixes are documented in the file history.c


DINK32_MPC603ev >>
```

MDINK32 Example:

```
MDINK32_603e >>about

Data Cache has been enabled...
Instruction Cache has been enabled...

M   M  DDD   III  N   N  K  K   333    222
MM MM  D  D   I   NN  N  K K   3   3  2   2
M M M  D   D  I   N N N  KK      33     22
M   M  D  D   I   N  NN  K K   3   3  22
```

```
M   M  DDD   III  N   N  K  K   333   22222   for the MPC603

Version 10, Revision 7

Written by : Motorola's RISC Applications, Austin, TX
Released :  March 1, 1999
Welcome to Excimer.  A Minimum System PowerPC Design!

Copyright Motorola, Inc. 1993, 1994, 1995, 1996, 1997, 1998
```

4.1.3 assemble as

DINK32 mini-assembler

- **assemble** address
- **assemble** start **+**
- **assemble** start **-** end

The mini-assembler for the DINK32 system will display the contents of memory at the given location and enter interactive mode. The user will be queried for a valid mnemonics and operands which will be assembled into a valid opcode and stored at that memory location. A location can be left unmodified by typing <return> to pass over it.

The “plus” form of the command will allow the user to start assembling code at a given start location and will be terminated at the end of memory. The “range” version will start at the first address location and automatically terminate at the given end address.

At any point “x” can be entered as a mnemonic and assemble will terminate and return the user to the DINK32 prompt.

Branch labels are recognized by the assembler as a word followed by a colon (:) at the address currently being displayed by the assembler. The assembler tracks the current branch labels and automatically calculates the address to be entered into future instructions. The *symtab,st* instruction is available for manipulating the branch table in DINK32. Branch labels within PowerPC assembly instructions will not be recognized by the assembler if the branch label has not yet been entered into the table. The user may display the branch table list with the st instruction.

The DINK32 assembler ignores any comments preceded by a ‘#’ and any “.org” and “.dc” commands. The assembler does not interpret these lines as anything. It only ignores them. The simplified mnemonics that DINK32 Version 10.5 understands is quite extensive. In general, immediate values, including condition register bit offsets, are assumed to be hexadecimal unless preceded by 0b (binary) or 0d (decimal). Floating point and general purpose registers are recognized just like previous versions of DINK32 where the register number may be preceded by an “r” (general purpose) or an “f” (floating point) but is not necessary. Simplified branch mnemonics involving the condition registers may have the condition register number preceded by “cr” but isn’t necessary. The assembler always expects a “cr” field for compare and branch instructions where, according to the architecture, cr0 is implied if a “cr” field is not given. DINK32 does not implement the implied cr0 functionality of the simplified mnemonics.

Examples:

```
DINK32_603e >>as 60100+
0x00060100 0x85ffffc4 lwzu         r15, 0xffc4( r31 )         rlmi
r00,r02,r05,0,0
0x00060104 0x00ffffa0 WORD       0x00ffffa0       lfd f0,0x0ec5(r1)
0x00060108 0xff0040ef fsel.        f24, f00, f08, f03         rlwnm
r0,r13,r23,0x1,0xa
0x0006010c 0xfe4004ff fnmadd.     f18, f00, f19, f00
0x00060110 0x00ffff01 WORD      0x00ffff01      loop: #branch label
0x00060110 0x00ffff01 BRANCH LABEL loop:
0x00060110 0x00ffff01 WORD       0x00ffff01       ori r26,r2,0xfff
0x00060114 0x00ffff00 WORD      0x00ffff00      lfd f00,0x0503(r0)
0x00060118 0xef0040fd fnmsubs.     f24, f00, f03, f08          cmpw
cr3,r26,r0
0x0006011c 0x7f0000ff WORD        0x7f0000ff        bne cr3,loop
0x00060120 0x22ffbf80 subfic      r23, r31, 0xbf80        x

VERIFYING BRANCH LABELS.....


DONE VERIFYING BRANCH LABELS!
DINK32_603e >>st
Current list of DINK branch labels:
         KEYBOARD:    0x0
         get_char:    0x1e5e4
       write_char:    0x5fac
        TBaseInit:    0x39c4
   TBaseReadLower:    0x39e8
   TBaseReadUpper:    0x3a04
     CacheInhibit:    0x3a20
    InvEnL1Dcache:    0x3a40
      DisL1Dcache:    0x3a88
    InvEnL1Icache:    0x3aac
      DisL1Icache:    0x3b00
        BurstMode:    0x3bfc
         RamInCBk:    0x3c3c
       RamInWThru:    0x3c7c
        dink_loop:    0x5660
      dink_printf:    0x6368

Current list of USER branch labels:
             loop:    0x60110

DINK32_603e >>assemble 60300-60310
0x00060300 0x82ffff00 lwz        r23, 0xff00( r31 )       fadd 1 2 3
0x00060304 0x00ffff00 WORD        0x00ffff00        stw 1 2
0x00060308 0xef0080ff fnmadds.    f24, f00, f03, f16        sc
0x0006030c 0xff0000ff fnmadd.      f24, f00, f03, f00          bdnz
0x60010
0x00060310 0x04ffff00 WORD        0x04ffff00        #Comment
0x00060310 0x04ffff00 WORD        0x04ffff00        nop
DINK32_603e >>

DINK32_MAX >>as 70010
```

```
0x00070010 0xff8000ff fnmadd.     f28, f00, f03, f00        mfvscr v3
DINK32_MAX >>as 70014+
0x00070014 0xff0000ff fnmadd.    f24, f00, f03, f00       mtvscr v12
0x00070018  0x00fbff00  WORD           0x00fbff00           vmhaddshs
v3,v19,v3,v31
0x0007001c  0x00ffff00  WORD             0x00ffff00             vsldoi
v30,v16,v17,7
0x00070020 0xff0000ff fnmadd.     f24, f00, f03, f00         x
DINK32_MAX >>ds 70010+
0x00070010 0x10600604 mfvscr    V3
0x00070014 0x10006644 mtvscr    V12
0x00070018 0x10731fe0 vmhaddshs V3,V19,V3,V31
0x0007001c 0x13d089ec vsldoi     V30,V16,V17,0x7
0x00070020 0xff0000ff fnmadd.     f24, f00, f03, f00
```

4.1.4 bkpt bp

set, delete, list breakpoints

bkpt

- **bkpt** *address*
- **bkpt -d** *index*

The **bkpt** command allows the user to set a breakpoint at a given address, delete a breakpoint at a given index in the breakpoint list, and list the current breakpoints by index and address.

Breakpoints allow the user to run an application program and stop execution when code at the specified address is encountered. This command will set or delete only one breakpoint at a time, and must be repeated for each breakpoint.

Setting a breakpoint will not remove a breakpoint from an address if a breakpoint already exists there. Deleting a breakpoint from an invalid index has no effect. Breakpoints can be set or deleted one at a time and all are displayed during a breakpoint list. A maximum of 20 breakpoints can be set in the system.

Examples:

```
DINK32_750 >> bkpt 60100
Breakpoint set at 0x00060100

DINK32_750 >> bkpt
Current breakpoint list:
1. 0x00060100

DINK32_750 >> bkpt -d 1
Breakpoint deleted

DINK32_750 >> bkpt
Current Breakpoint List:
```

4.1.5 defalias da

define alias

The runalias, ra, command is the companion to this command. While these commands, da and ra, are still available, the **env** command is more flexible.

- **defalias**

This command will allow the user to define an alias to a list of commands (separated by a semicolon). Once the alias has been defined, **runalias** can be used instead of retyping the list of commands. Only one alias may be set at a time, and using **defalias** a second time will overwrite the previously aliased command list. Below is an example of using an alias to single step and display registers.

Example:

```
DINK32_750 >> trace 2100
A Run Mode or Trace exception has occurred.
Current Instruction Pointer: 0x00002104 lwz r03, 0x0000(r02)

DINK32_750 >> defalias
Current alias definition:
New alias : tr +; rd r
Alias defined as : tr +; rd r

DINK32 will now single step and display the register set each time
runalias is entered.

DINK32_750 >> runalias
A Run Mode or Trace exception has occurred.
Current Instruction Pointer: 0x00002108 add r03, r00, r01
gpr00: 0x00000000 gpr01: 0x00060000
gpr02: 0x00000000 gpr03: 0x0002bc00
gpr04: 0x00000000 gpr05: 0x00000000
gpr06: 0x00000000 gpr07: 0x00000000
gpr08: 0x00000000 gpr09: 0x00000000
gpr10: 0x00000000 gpr11: 0x00000000
gpr12: 0x00000000 gpr13: 0x00000000
gpr14: 0x00000000 gpr15: 0x00000000
gpr16: 0x00000000 gpr17: 0x00000000
gpr18: 0x00000000 gpr19: 0x00000000
gpr20: 0x00000000 gpr21: 0x00000000
gpr22: 0x00000000 gpr23: 0x00000000
gpr24: 0x00000000 gpr25: 0x00000000
gpr26: 0x00000000 gpr27: 0x00000000
gpr28: 0x00000000 gpr29: 0x00000000
gpr30: 0x00000000 gpr31: 0x00000000
```

4.1.6 devdisp dd

DINK32 Peripheral device display

dd,devdisp

- **dd** [device [-b|-h|-w] addr1-addr2]

The devdisp command displays the contents of device registers in a manner similar to that of the memory display command.

- **device** Is the name of the device. If not entered display all known devices
- **-b, -h, -w** Set size of device accesses. If not specified, the default size is bytes for devices.
- **addr1** Is the starting address to display.
- **addr2** Is the optional ending address.
- The dd command with no parameters will display a list of all the known devices.

Example:

```
DINK32_ARTHUR >> dd
         Device    Start     End       Sizes
         ========  ========  ========  =====
         mem       00000000  FFFFFFFF  [BHW]
         nvram     00000000  00000FFF  [B]
         i2c       00000000  0000007F  [B]
         rtc       00000000  0000000D  [B]
         rtcram    0000000E  000000FF  [B]
         apc       00000040  00000048  [B]
        DINK32_ARTHUR >> dd nvram 40
          0x0040 14 3E 27 9C EE FA E9 C0 04 6B 2A 87 08 9C 66 7E
.................
        0x0050 ...
    ...
        dd>x
DINK32_ARTHUR >>
```

4.1.7 devmod dm

DINK32 Peripheral device modify

devmod,dm

dm [device [-b|-h|-w] addr1-addr2]

The device modify command allows interactive modification of device data in registers and/or indirect memory. The **dd** command operates similar to the mm command, with some additional flexibility.

- **device** Is the name of the device. If not entered display all known devices
- **-b, -h, -w** Set size of device accesses. If not specified, the default size is bytes for devices.
- **addr1** Is the starting address to display.
- **addr2** Is the optional ending address or if not specified then display/modify until user types x or ESC.

While examining data, the contents may be modified by entering a hexadecimal value. The value entered is truncated to the specified size and is then written to the device or memory.

When prompted for location, any of the following may be entered:

- **<enter>** go to the next location using the current selected direction (defaults to forward)
- **'v'** set the direction to forward.
- **'^'** set the direction to reverse.
- **'='** set the direction to 0. dm will keep examining and modifying the same location until 'v' or '^' is entered.
- **hex** a value to write.
- **'?'** help

```
DINK32_ARTHUR >> dm nvram 40
     0x0040 : 14 ? <enter>        -- skip
     0x0041 : 3E ? 47             -- new value
     0x0042 : 27 ? ^              -- go back
     0x0041 : 47 ? 48             -- right value
     0x0040 : 14 ? v              -- go forward
     0x0041 : 48 ? =<enter>
     0x0041 : 48 ? <enter>
     0x0041 : 48 ? <enter>
     0x0041 : 4A ? <enter>        -- erratic bit?
```

4.1.8 devtest dev

DINK32 Peripheral device test <Kahlua only>

dev,devtest

- **dev** [+|-] epic
- **dev** [+] [-r] i2c <addr> <-n> [<timeout>]
- **dev** [+] -w i2c <addr> <-n> <str> [<timeout>]
- **dev** [+] DMA p<type>] <src> <dest> [<chn>] [<n>]]

Perform a given I/O test on Kahlua.

```
DINK32_KAHLUA>> devtest -r i2c

0x40:   FE  FE  FE  FE  47  4A  4E  4F  FE  FE  FE  FE  47  4A  4E  4F
....GJMN....GJMN
```

4.1.9 disassem ds

DINK32 mini-disassembler

- **disassem** address
- **disassem** start **+**
- **disassem** start **-** end

The mini-disassembler for the DINK32 system displays the contents of memory at the given address. The contents are shown in hexadecimal opcode format as well as in PowerPC assembly instruction format.

If the “plus” form is used, the command goes into interactive mode and will continue reading and disassembling until the end of memory is reached or until the user types “x”.

If the “range” form is used, the command will continue reading and disassembling for each inclusive address in the range specified.

Note that the above parameter forms can be combined by separating the forms with a comma or white space. This will display multiple disassembled portions of the memory space with one command.

Branch labels entered during an assemble session are displayed during disassembly. In order for branch labels to be calculated correctly, branch labels must be entered before instructions refer to that label.

Examples:

```
DINK32_750 >> ds 60100
0x00060100 0x58402800 rlmi r00, r02, 0x05, 0x00, 0x00

DINK32_750 >> ds 60118-60120
0x00060118 0xc8000503 lfd f00, 0x0503( r00 )
0x0006011c 0x243f002c dozi r01, r31, 0x002c
0x00060120 0x00000000 WORD 0x00000000

DINK32_750 >> ds 60100+
0x00060100 0x58402800 rlmi r00, r02, 0x05, 0x00, 0x00
0x00060104 0xc8010ec5 lfd f00, 0x0ec5( r01 )
0x00060108 0x5da0b854 rlwnm r00, r13, r23, 0x01, 0x0a
0x0006010c 0x00000000 WORD 0x00000000
0x00060110 0x00000000 WORD 0x00000000
0x00060114 0x605affff ori r26, r02, 0xffff
0x00060118 0xc8000503 lfd f00, 0x0503( r00 )
0x0006011c 0x243f002c dozi r01, r31, 0x002c
0x00060120 0x00000000 WORD 0x00000000
0x00060124 0x00000000 WORD 0x00000000
x to quit, anything else to continue >
```

4.1.10 download dl

download data from the host MDINK32 Compatible

RAM download Syntax:

- **download -k** *(keyboard port - duart channel A)*
- **download -h** *(host port - duart channel B)*
- **download** {-k|-h} [-q] [-fx] [-v] [-o offset]

FLASH download Syntax:

- **download -fl [-e] -o** *address (download directly to flash memory)*

This instruction provides the ability to receive data from the host keyboard via the serial port. The data received can be in two formats: S-Records or compressed S-Records, which are automatically decompressed. The data which is downloaded will be placed in memory locations specified by the input file for RAM or as specified for FLASH. There are two separate forms, one for RAM and one for FLASH downloads. Information on S-Records can be found in Appendix D.

RAM download options:

- The "-k" option copies the data stream from the keyboard serial port into memory, while "-h" option copies data from the host serial port. One of these two options must be supplied.
- The "-q" option is quiet mode, no indication of download progress is supplied.
- The "-fx" option enables XON/XOFF (software) flow control for downloading at higher speeds.
- The "-v" option verifies a previous download, printing an error message for each difference found.
- The "-o offset" option adds a hexadecimal offset to the address of the S-Record lines to relocate code.

FLASH download options:

- The “-fl” option indicates a load to FLASH memory.
- The “-e” option indicates to erase all of flash memory before the load.
- The -o address specifies the offset address, default is 0xfff00000.

Default download baud rate is 9600. Maximum baud rate on Excimer and Maximer is 57600 and Yellowknife and Sandpoint is 38400.

See Section 4.1.34, “setbaud sb".

Examples:

```
DINK32_750 >> dl  -k

Set Input Port: set to Keyboard Port
Download Complete.

...

Use the following example when upgrading DINK on Excimer
with a s-record from the PowerPC website:

MDINK32_603e >> dl -fl -o ffc00000
Offset:     0xffc00000
Writing new data to flash.
Line: 50

NOTE:  The complete sequence for upgrading DINK on Excimer would be:
MDINK32_603e >> fw -e
Reboot the Excimer board
MDINK32_603e >> sb -k 57600
MDINK32_603e >> dl -fl -o ffc00000


MDINK32_603e >>
```

4.1.11 env env

Syntax: env [-c][-d][-s][var[=value]]

Description: This command displays or sets environment variables stored in the NVRAM (if available). If no argument is given, the current settings are displayed. Note: quotes (") are usually required.

The ENV command manipulates environment variables, which are of the form VAR=DEF or VAR="def def def". Quotes are needed if non-alphanumeric characters are included.

- For YK/SP, NVRAM is used and preserved, and 4K is available.
- For Excimer and Maximer, the uppermost 1K of SRAM is used. Currently, Excimer and Maximer don't save/restore SRAM->Flash. Since Excimer and Maximer don't wipe the SRAM it can be somewhat useful since it will be preserved between resets.

Using ENV, the system can be configured on startup. The following variables are checked:

- IO -- sets I/O type and modes
  - IO=COM1 Use standard COM port
  - IO="COM1:[9600|19200|..." Use standard COM port and optionally set serial port.
  - IO="PMC:[9600|19200|..." Use serial port on PMC8240/etc.
  - IO=XIO Use VGA card in first slot with a VGA-class code.
  - IO=XIO:USE=nn Use VGA card on slot #nn even if it doesn't appear to be a video card (old cards w/out CLASS codes).
- MEMOPT -- if defined, the equivalent of "meminfo -c -c" is run,which tunes memory using SDRAM I2C info and bus speed.
- ALIAS -- stores last defined alias (da/ra).
- MDMODE -- if set to 1, use the dm/dd commands in place of the mm/md commands. If set to 3, do that and also enable denser output for 'md'.
- RDMODE -- if set to 'q', 'quieten' the register display for SPR's. If set to 'e', 'explain' the fields of SPRs.
- TAUCAL -- saves/restores the TAU calibration field (32-bit ULONG).
- L2CACHE -- sets L2 cache parameters. Options are:
  - L2CACHE={256K|512K|1M|2M} ',' {/1|/1.5|/2|/2.5|/3|/3.5} ',' [late] ',' [do] ',' {0.5ns|1.0ns|1.5ns|2.0ns} ',' [wt] ',' [diff]

If any key is pressed on startup (recommendation is Backspace), the ENV is ignored.

ENV allows for multiple command aliases

Example:

```
ENV R="rd"
ENV X="tr; rd msr; md 90000-90100"
```

You can enter 'r' to do 'rd' (or 'r r3' to do 'rd r3') or 'x' to do all the above def's. Aliases cannot be nested. Note that the ENV does not distinguish between ENV vars and ALIAS vars -- they're lumped together.

ENV allows changing the prompt dynamically. If the string PROMPT is defined in the ENV, it is expanded and displayed using the following rules:

- $d -- dink name, either DINK or MDINK
- $P -- formal processor name, e.g. "MPC7400"
- $p -- informal processor name, e.g. "MAX"
- $T -- current time, "12:34:56PM"
- $t -- TAU temperature, e.g. "26" if 26 deg. C or "26u" if not calibrated yet.
- $! -- history index
- $_ -- CRLF
- All other characters are copied as-is.

Flags:

- -c Clear/Initialize the NVRAM.
- -d Delete named variable.
- -s Saves environment to permanent storage, used for excimer and maximer only.

Most of the SPR’s can suppress the verbose mode, see Section 4.1.30, “regdisp rd".

Example:

This example sets the non verbose mode for certain commands.

```
DINK32_ARTHUR >>env -c
DINK32_ARTHUR >>env rdmode=e
```

After the non verbose mode is set, the following command gives non verbose results. Contrast this with the verbose display in Section 4.1.30, “regdisp rd".

```
DINK32_ARTHUR >>rd msr
MSR : 0x00003930
  POW=0  EE=0  PR=0  FP=1  ME=1  FE0=1  SE=0
  BE=0  FE1=1  IP=0  IR=1  DR=1  RI=0  LE=0
  TLB/GPR=0    VMX=0  PM=0
```

4.1.12 flash fl

flash memory commands; mdink32 limited compatibility

flash

This command will perform a variety of flash memory operations.

Syntax: fl -flags -o value -s sector number

Description: This command performs actions to the flash memory

- -dsi display sector information (dink32/mdink32)
- -e erase all of flash (dink32/mdink32)
- -cp copy MDINK from RAM to Flash (dink32 only)
  Required Flags: -o <value> copy address in flash
  Optional Flags: -e erase flash first
- -sp protect indicated sector (dink32 only)
  Required Flags: -n <value> sector number 0-18
- -su unprotect indicated sector (dink32 only)
  Required Flags: -n <value> sector number 0-18
- -se erase indicated sector (mdink32/dink32)
  Required Flags: -n <value> sector number 0-18

For Version 12.0: -cp is not implemented.

Sector Protect/Unprotect commands require a 12V power supply. See AMD Bulletin, NVD Flash, Sector Protection, available on the www.amd.com web site.

Example:

```
DINK32_603e >>fl -se -n 6
Erasing sector 6

DINK32_603e >>fl -dsi
Display Sector Information 0.7  Excimer Rev 2 and prior
Description      value
Manufacturer ID is 0x1, Device ID is 0x225b
Sector SA0   UNPROTECTED
Sector SA1   UNPROTECTED
Sector SA2   UNPROTECTED
Sector SA3   UNPROTECTED
Sector SA4   UNPROTECTED
Sector SA5   UNPROTECTED
Sector SA6   UNPROTECTED
Sector SA7   UNPROTECTED
Sector SA8   UNPROTECTED
Sector SA9   UNPROTECTED
```

```
Sector SA10  UNPROTECTED
Sector SA11  UNPROTECTED
Sector SA12  UNPROTECTED
Sector SA13  UNPROTECTED
Sector SA14  UNPROTECTED
Sector SA15  UNPROTECTED
Sector SA16  UNPROTECTED
Sector SA17  UNPROTECTED
Sector SA18  UNPROTECTED
```

4.1.13 fupdate fu

FLASH update

fupdate, fu

- fupdate -h [-o offset]
- fudpate -i

The flash update command is used to initialize the contents of the flash devices on a Sandpoint or Yellowknife system. There are two separate functions:

- PPMC ROM Initialization

When used with the '-i' option, the host ROM (the 32-pin PLCC socket on Sandpoint or Yellowknife motherboards) can be copied to the local flash devices on PPMC cards. To use this feature, the system must be set to boot from the host ROM on PCI (usually true), and the PROGMODE switch must be set on the PPMC card (refer to PPMC documentation for details).

- Motherboard Flash updates

When used with the '-h' option the host ROM can be updated with new versions of DINK or with the boot code of an RTOS. Usually the memory contents will be downloaded DINK upgrade or an RTOS boot image. See Section 4.1.10, “download dl" for details on loading the memory image.

> NOTE: The entire flash is erased and replaced with the supplied contents.

If the programming mode fails or is interrupted the flash may be unusable. If the DINK32 code is replaced with another program DINK will be lost unless the new program has similar facilities to download and program DINK into the flash ROM.

Examples

Use the following example store a program in the PCI-based ROM of a Sandpoint or Yellowknife (for example, a DINK upgrade).

```
DINK32_750 >> dl -k -o 100000
Download from Keyboard Port
Offset Srecords by 0x00100000
      ...
Download Complete.
DINK32_750 >> fu -h 100000
YK/SP PCI Flash Programmer
Are you sure? Y
Check flash type: AMD Am29F040
Erasing flash   : OK
```

```
Program flash   : OK
Verifying flash : OK
DINK32_750 >>
```

Use the following example to copy DINK32 into a local-bus Flash on a PPMCcard:

```
DINK32_750 >> fu -i
PPMC Local Flash Programmer\
Are you sure? Y
Check flash type: AMD Am29LV800BB
Erasing flash   : OK
Program flash   : OK
Verifying flash : OK
DINK32_750 >>
```

4.1.14 fw fw -e

Specific FLASH download MDINK32 Only

fw –e [-o <flash address>]

This command copies the contents of the entire 512K of RAM to FLASH starting at flash address 0xFFF00000. The parameter -e is required. The optional parameter –o <flash address> can be used to specify a specific address to copy from ram to rom address. (I.e. replacing flash address 0xfff00000 with the flash address of the user's choosing.

Examples:

```
MDINK32_603e >>fw -e
Chip erase set.
Erasing entire flash memory...
Entering verify erase loop ...
Flash erased!!!
Done erasing flash memory.
Copying 512K ram to flash address fff00000...
```

4.1.15 go go

execute user code MDINK32 Compatible

go address

go +

This command allows the user to execute user code starting at the given address. The “plus” form will allow execution at the address in the SRR0 (Machine Status Save / Restore) register - bits 0-29. This is useful for continuing where a breakpoint or a user break (<ctrl>-c) had previously stopped execution.

A program exception occurs when a breakpoint or illegal opcode is encountered. The breakpoint address will be displayed and the instruction at that address will be disassembled. Note: If a breakpoint is encountered, the user must clear the breakpoint in order for execution to continue.

When the user program begins execution, the stack pointer, r1, is set to 0x8fff0. Hence the user stack begins at 0x8fff0.

Examples:

```
DINK32_750 >> ds 181dc-181f8
0x000181dc 0x3c600000 addis r03, r00, 0x0000
0x000181e0 0x60631234 ori r03, r03, 0x1234
0x000181e4 0x3c800000 addis r04, r00, 0x0000
0x000181e8 0x60845678 ori r04, r04, 0x5678
0x000181ec 0x7c632214 add r03, r03, r04
0x000181f0 0x38841234 addi r04, r04, 0x1234
0x000181f4 0x7c032000 cmp 0, 0, r03, r04
0x000181f8 0x4182ffe4 bc 0x0c, 0x02, 0xffe4

DINK32_750 >> bkpt 181f4
breakpoint set at 0x000181f4

DINK32_750 >> go 181dc
A Program exception has occurred.
Breakpoint Encountered:
Current Instruction Pointer: 0x000181f4 cmp 0, 0, r03, r04

DINK32_750 >> go +
A Run Mode or Trace exception has occurred.
A Program exception has occurred.
Breakpoint Encountered:
Current Instruction Pointer: 0x000181f4 cmp 0, 0, r03, r04
```

4.1.16 help he

help on DINK32 commands MDINK32 Compatible

help <command>

This provides information on the commands implemented by DINK32. Since MDINK32 only has a subset of commands, the help command displays different information.

Examples:

```
DINK32_KAHLUA >>help
            Sandpoint/MPC8240 DINK COMMAND LIST
Command       Mnemonic        Command       Mnemonic
=======       ========        =======       ========
About...           about, ab          Assemble          assemble, as
Benchmark      benchmark, bm   Breakpoint ops bkpt, bp
Define Alias       defalias, da        Device Display    devdisp, dd
Device Modify      devmod, dm          Device Tests      devtest, dev
Disassemble        disassem, ds        Download          download, dl
Flash commands flash, fl       Flash update   fu -s
Go              go             Help           help, he
Info            info, in       Log session    log
Memory Display memdisp, md     Memory Modify  memod, mm
Memory Fill    memfill, mf     Memory Move    memove, mv
Memory Search      memsrch, ms         Memory Test       memtest, mt
Menu             menu, me            PCI Bus Probe     pciprobe, ppr
PCI Slot Display  pcidisp, pd          PCI Reg Modify     pcimod, pm
PCI Config Regs   pciconf, pcf        Register Display   regdisp, rd
Register Modify regmod,   rm   Real-Time Clock rtc
Run Alias         runalias, ra        Set Baud Rate      setbaud, sb
Set Input         setinput, si        Show SPRs          spr_name, sx
Symbol table      symtab, st          Transparent Mode   transpar, tm
Trace          trace, tr         . (repeat last command)


DINK32_MPC603ev >>help
            Excimer DINK COMMAND LIST
Command             Mnemonic          Command           Mnemonic
=======             ========          =======           ========
About...            about, ab         Assemble          assemble, as
Benchmark           benchmark, bm     Breakpoint ops    bkpt, bp
Define Alias        defalias, da      Disassemble       disassem, ds
Download            download, dl      Flash commands    flash, fl
Go                  go                Help              help, he
History             history,hist      Info              info, in
Log session         log               Memory Display    memdisp, md
Memory Modify       memod, mm         Memory Fill       memfill, mf
Memory Info         meminfo, mi       Memory Move       memove, mv
Memory Search       memsrch, ms       Memory Test       memtest, mt
Menu                menu, me          Register Display  regdisp, rd
Register Modify     regmod, rm        Reset             reset, rst
Run Alias           runalias, ra      Set Baud Rate     setbaud, sb
Set Input           setinput, si      Show SPRs         spr_name, sx
```

```
Symbol table      symtab, st        Tau               tau
Transparent Mode  transpar, tm      Trace             trace, tr
. (repeat last command)

For additional details about a command, please type "help <mnemonic>"
DINK32_MPC603ev >>

MDINK
            MINIMUM DINK COMMAND LIST
Command           Mnemonic
=======           ========
About...                  about, ab
Download                  download, dl
Help                      help,he
Go                  go
Menu                      menu, me

DINK32_750 >> help go
```

Individual Commands

```
DINK32_MPC603ev >>help go
GO
==
Mnemonic: go
Syntax: go [<address>|+]
Description: This command allows the user to execute user code
starting at
       the specified address.  Execution will continue until a
breakpoint or
     an exception occurs.
    If the "+" form is used, then execution will start at the address
     defined by the contents of bits 0-29 of SRR0.

     The user should terminate their code with an illegal opcode or
with a
      breakpoint.  The value of dink_loop() is initially placed in
the User
     Programming Model link register.  If you terminate your code
      with a blr to that location you will re-enter DINK.  In the
process,
     however, you will perform the prolog of the dink_loop function
which
      will save registers (ex. lr) off onto the currently defined
stack (ie.
     the value in r1). This may be an unexpected side-effect.

     Note: If a breakpoint is encountered, the user must clear the
     breakpoint in order for execution to continue.
DINK32_MPC603ev >>
```

4.1.17 log log

Toggles logging

Only available on yellowknife and sandpoint.

- log

This command provides the capability to log a debug session. The command toggles the logging function. When logging is enabled, all characters sent to the terminal will be echoed to the host port, the second com port, com2 (duart channel B) in the system. On Yellowknife, this will be the alternate com port to the terminal port. See Section 4.1.34, “setbaud sb".

Example:

```
DINK32_750 >> log

You are enabling logging! After this message all input and output to
your terminal will be mirrored out to the host port. Now would be a
time to open an editor on the host and get into insert mode

DINK32_750 >> log
Logging disabled!
```

4.1.18 memdisp md

display memory

- **memdisp** address
- **memdisp** start +
- **memdisp** start - end

This command displays data stored in the specified memory locations. The display will always be aligned on a 16-byte boundary in which the address given will be included. In order to keep from saturating the screen, a maximum of four lines of data are displayed on the screen, followed by a prompt. To continue viewing data, the user enters <return> at the prompt. Multiple parameters may be entered.

If the \"+\" form is used, the command will continue to display blocks of memory if the user enters <return> at the prompts, until the end of memory is reached or until the user enters an \"x\". If the two-address version is used, the command will display the contents of memory between and including each address specified in the range. If more than four lines of data are requested, the user can then enter an \"x\" at the prompt to quit before the end of the display range.

The start address is normalized to the previous quad-word boundary. Likewise, the ending address is normalized to the next quad-word boundary. For example, if the start address was 0x00000104 then the first memory address to be displayed would be 0x00000100. If the end address was 0x00000104 then the last memory location to be displayed would be 0x0000010C.

Examples:

```
DINK32_750 >> memdisp 60100,60200
0x00060100 00000041 00000042 00000043 00000044
0x00060200 00000000 00000000 00000000 00000000

DINK32_750 >> memdisp 60100-60130
0x00060100 00000041 00000042 00000043 00000044
0x00060110 00000045 00000046 00000047 00000048
0x00060120 00000000 00000000 00000000 00000000
0x00060130 00000000 00000000 00000000 00000000

DINK32_750 >> memdisp 60260+
0x00060260 00000000 00000000 00000000 00000000
0x00060270 00000000 00000000 00000000 00000000
0x00060280 00000000 00000000 00000000 24002400
```

4.1.19 memfill mf

memory fill

memfill start end data

The range of memory spanning from the starting address to the ending address is filled in with the given 32-bit data pattern. The fill is inclusive of the end point.

Examples:

```
DINK32_750 >> memfill 60100 60200 89898989
DINK32_750 >> memfill 60140 6015c 00000000
DINK32_750 >> memdisp 60120-60160
0x00060120 89898989 89898989 89898989 89898989
0x00060130 89898989 89898989 89898989 89898989
0x00060140 00000000 00000000 00000000 00000000
0x00060150 00000000 00000000 00000000 00000000
0x00060160 89898989 89898989 89898989 89898989

DINK32_750 >> memfill 60144 60144 44444444
DINK32_750 >> memdisp 60120-60160
0x00060120 89898989 89898989 89898989 89898989
0x00060130 89898989 89898989 89898989 89898989
0x00060140 00000000 44444444 00000000 00000000
0x00060150 00000000 00000000 00000000 00000000
0x00060160 89898989 89898989 89898989 89898989
```

4.1.20 meminfo mi

mi [-s][-c][-c]

mi displays information about the memory settings. If no option is selected, the current memory controller settings are decoded.

Options (for SODIMM/DIMM-based systems only):

- -s -- show I2C ROM info.
- -c -- compare I2C info to memory controller settings for errors. If -c is entered a second time, the settings will be corrected. Setting the MEMOPT ENV variable is equivalent to entering `mi -c -c` at startup.

Example:

```
DINK32_ARTHUR >>mi
Memory settings:
  ROM Speed: 30 ns (2 clocks)
  SDRAM Bank 0:  Disabled
  SDRAM Bank 1:  Disabled
  SDRAM Bank 2:  Enabled
    Range: [00000000 -> 000fffff]  1 MBytes
    Speed: 0/1/1/1
  SDRAM Bank 3:  Enabled
    Range: [08000000 -> 080fffff]  1 MBytes
    Speed: 0/1/1/1
  SDRAM Bank 4:  Enabled
    Range: [08400000 -> 094fffff]  17 MBytes
    Speed: 0/1/1/1
  SDRAM Bank 5:  Enabled
    Range: [00000000 -> 000fffff]  1 MBytes
    Speed: 0/1/1/1
  SDRAM Bank 6:  Enabled
    Range: [00000000 -> 000fffff]  1 MBytes
    Speed: 0/1/1/1
  SDRAM Bank 7:  Disabled
```

4.1.21 memod mm

memory modify

- **memod** address
- **memod** start **+**
- **memod** start **-** end

Memory modify is an interactive command. It will display the contents of the given memory address and allow the user to change the value stored there. Memory is considered to be a contiguous set of 32-bit integers.

The “plus” form causes the command to start at a given address and continue until the end of memory or until the user types “x” to exit the memory modify loop.

The “range” form allows modifications for the inclusive range from start to end. When the end address is reached the memory modify loop is automatically exited. The user can type “x” at any time to exit the memory modify loop.

- -b for byte
- - h for halfword
- -w for word (default))

Examples:

```
DINK32_750 >> memod 60100
0x00060100 : 0x89898989 : ? 44444444

DINK32_750 >> memod -b 60100
0x00060100 : 0x44444444 : ? 66

DINK32_750 >> memod -h 60100
0x00060100 : 0x66444444 : ? 3333

DINK32_750 >> memod -w 60100
0x00060100 : 0x33334444 : ? 22222222

DINK32_750 >> memod 60110-60118
0x00060110 : 0x89898989 : ? 11111111
0x00060114 : 0x89898989 : ? 22222222
0x00060118 : 0x89898989 : ? 33333333

DINK32_750 >> memod 60200+
0x00060200 : 0x89898989 : ? 12341234
0x00060204 : 0x00000000 : ? 12341234
0x00060208 : 0x00000000 : ? x
```

4.1.22 memove mv

memory move

- **memove** <start addr> <end addrs> <dest addr>

This command copies data from a block of memory, bounded inclusively by the first two addresses, to a block of memory starting at the third address. The result of this command will be two identical blocks of memory. If the third address falls between the first two addresses, an error message is returned and memory will not be modified.

Examples:

```
DINK32_750 >> memfill 60100 60110 ffffffff
DINK32_750 >> memdisp 60100-60150
0x00060100 ffffffff ffffffff ffffffff ffffffff
0x00060110 ffffffff 00000000 00000000 00000000
0x00060120 00000000 00000000 00000000 00000000
0x00060130 00000000 00000000 00000000 00000000
0x00060140 00000000 00000000 00000000 00000000
0x00060150 00000000 00000000 00000000 00000000

DINK32_750 >> memove 60100 60110 60140
DINK32_750 >> memdisp 60100-60150
0x00060100 ffffffff ffffffff ffffffff ffffffff
0x00060110 ffffffff 00000000 00000000 00000000
0x00060120 00000000 00000000 00000000 00000000
0x00060130 00000000 00000000 00000000 00000000
0x00060140 ffffffff ffffffff ffffffff ffffffff
0x00060150 ffffffff 00000000 00000000 00000000
```

4.1.23 memsrch ms

memory search

ms <address> <address> <data>

This command searches for a 32-bit data pattern in the inclusive block specified by the range of the two addresses. If the second address is less than the first address, an error message is returned and no search is performed. If the pattern is found, the addresses of matching data are printed to the screen. The command,

`ms 50100 50200 fff01234`
searches for the data pattern "fff01234" in memory locations 0x50100 to 0x50200 inclusive, and prints the matching addresses.

Example:

```
DINK32_603e >>md 60100-60120
0x00060100       10ff7f00 00ffff00 ff2023ff ff0402ff        .........
#.....
0x00060110                  00ffff00  00ffff00  ff5008ff  ff1002ff
.........P......
0x00060120                  00efef00  00ffff00  ff0100ff  ff0030ff
..............0.


DINK32_603e >>ms 60100 60120 ff5008ff
        0x00060118
```

4.1.24 memtest mt

memory test

- mt [-d dev][-b|-h|-w][-l loop][-t][-h][-a][-q] addr1-addr2

The memtest command performs various memory tests on local memory or device registers. The basic format is:

mt [-d dev][-b|-h|-w][-l loop][-t][-h][-a][-q] addr1-addr2

- -d device Test the indicated device instead of memory. Use the "dm" command to get a list of devices. NOTE: testing non-volatile I2C EEPROM devices can destroy valuable information as well as reduce the life expectancy of those devices.
- -b, -h, -w Test memory or device using byte, half-word or word accesses. Memory can be tested in any size, while devices may be limited to bytes. If not specified, the default size is word for memory and bytes for devices.
- -l loop-cnt Specifies the number of times the memory test should perform all tests. If not specified, each test is performed once, while if '0' is specified, the test is run forever.
- -x If specified, the testing halts immediately when any error is found. This is useful for extended passes to trap on any error.
- -q Perform only a quick test.
- -a Perform all defined memory tests (can be slow).
- -n list Perform only specified memory tests. Tests are selected by adding one or more of the following letters to "list":
  - -0 : walking 0's test (non-destructive, slow)
  - -1 : walking 1's test (non-destructive, slow)
  - -A : address=data test (destructive)
  - -Q : quick pattern test (non-destructive)
  - -R : random pattern test (non-destructive)
  - -S : write sensitivity test (destructive, slow)
- -t Show elapsed time (only on systems with a real-time clock).

- addr1-addr2 Specifies the starting and ending address, respectively. The addresses must be aligned to the size of the access (as specified by the-b/-h/-w option) Note: be careful not to test memory regions used by DINK. 0x90000 is a safe starting point for DINK 11.0.2 or earlier.

Examples:

```
DINK32_ARTHUR >>mt -q 90000-1fffffc
                                        This quickly tests the default
32MB SDRAM DIMM
                                    on Yellowknife/Sandpoint systems.

DINK32_ARTHUR >>mt -q 90000-1fffffc
  PASS 1:
   Quick Test.....................................................PASS
Completed tests: No errors.

DINK32_ARTHUR >> mt -b -a -l 0 -x 90000-1ffffff
Use all defined test to test 32MB of memory, using only byte
accesses.  Repeat the test forever unless an error occurs.

DINK32_ARTHUR >>mt -b -a -l 0 -x 90000-1fffff
  PASS 1:
   Quick Test.....................................................PASS
   Random Pattern Test............................................PASS
   Walking 1's Test...............................................PASS
   Walking 0's Test...............................................PASS
   Address March Test.............................................PASS
   Write Sensitivity Test.........................................PASS


DINK32_ARTHUR >>mt -n S -t 90000-1fffff
Test 32MB using only the write sensitivity test,  and report the
elapsed time.

DINK32_ARTHUR >>mt -t -n S 90000-A0000
  PASS 1:
   Write Sensitivity Test.........................................PASS
Completed tests: No errors.
Elapsed time: 0:00:16
DINK32_ARTHUR >>
```

4.1.25 menu me

show list of DINK32 commands MDINK32 Compatible

menu (same as “help”)

This command will list all of the commands that are available in the current implementation of DINK32.

Examples:

```
DINK32_ARTHUR >>menu
            Excimer DINK COMMAND LIST
Command             Mnemonic          Command             Mnemonic
=======             ========          =======             ========
About...            about, ab         Assemble            assemble, as
Benchmark           benchmark, bm     Breakpoint ops      bkpt, bp
Define Alias        defalias, da      Disassemble         disassem, ds
Download            download, dl      Flash commands      flash, fl
Go                  go                Help                help, he
History             history,hist      Info                info, in
Log session         log               Memory Display      memdisp, md
Memory Modify       memod, mm         Memory Fill         memfill, mf
Memory Info         meminfo, mi       Memory Move         memove, mv
Memory Search       memsrch, ms       Memory Test         memtest, mt
Menu                menu, me          Register Display    regdisp, rd
Register Modify     regmod, rm        Reset               reset, rst
Run Alias           runalias, ra      Set Baud Rate       setbaud, sb
Set Input           setinput, si      Show SPRs           spr_name, sx
Symbol table        symtab, st        Tau                 tau
Transparent Mode    transpar, tm      Trace               trace, tr
. (repeat last command)

For additional details about a command, please type "help <mnemonic>"


MDINK32_ARTHUR >>menu
            MINIMUM DINK COMMAND LIST
Command             Mnemonic
=======             ========
About...            about, ab
Download            download, dl
Flash ram to rom    fw  -e
Flash display       fl -dsi
Help                help,he
Go                  go
Menu                menu, me

For additional details about a command, please type "help <mnemonic>"
```

4.1.26 pciconf pcf

pci probe command (on systems with a PCI bus)

pciconf <devNum>

This command displays 26 common PCI configuration registers, and 16 additional device specific registers of a PCI device. The devNum depends on which PCI slot the device is attached to, and it can be found by executing the ppr (PCI Device Probe) command.

Example:

```
DINK32_750 >> ppr
devNo          PCI ADR.             DEVICE ID     VENDOR ID
=====          ========             ========= =========
11             0x80005800           0x0565        0x10ad

DINK32_750 >> pcf 11
ADDR.          VALUE                DESCRIPTION
=====          =====                ===========
0x00           0x10ad               Vendor ID
0x02           0x0565               Device ID
0x04           0x0007               PCI command
0x06           0x0200               PCI status
0x08           0x04                 Revision ID
0x09           0x00                 Standard Programming Interface
0x0a           0x01                 Subclass code
0x0b           0x06                 Class code
0x0c           0x00                 Cache line size
0x0d           0x00                 Latency timer
0x0e           0x80                 Header type
0x0f           0x00                 BIST control
0x10           0x00000000           Base Address Register 0
0x14           0x00000000           Base Address Register 1
0x18           0x00000000           Base Address Register 2
0x1c           0x00000000           Base Address Register 3
0x20           0x00000000           Base Address Register 4
0x24           0x00000000           Base Address Register 5
0x28           0x00000000           Cardbus CIS Pointer
0x2c           0x0000               Subsystem Vendor ID
0x2e           0x0000               Subsystem ID
0x30           0x00000000           Expansion ROM Base Address
0x3c           0x00 Interrupt line
0x3d           0x00 Interrupt pin
0x3e           0x00 MIN_GNT
Type <return> to continue or "x" to quit >>
```

4.1.27 pcidisp pd

pci display (on systems with a PCI bus)

pcidisp <devNum> <regNum>

This command reads a configuration register (regNum) of a PCI device (devNum). The devNum depends on the PCI slot the device)is attached, and it can be found by executing the ppr (PCI Device Probe) command..

Example:

```
DINK32_750 >> pcidisp 11 10

0x10 0x12345678 Base Address Register 0
```

4.1.28 pcimod pm

pci modify (on systems with a PCI bus)

pcimod <devNum> <regNum>

This command is used to modify the content of a configuration register (regNum) of a PCI device (devNum). The DevNum depends on the PCI slot the device is attached to, and it can be found by executing the ppr (PCI Device Probe) command. This command first displays the current value of the desired register, then asks the user to enter the new value.

This command does not return an error if the register requested is a read-only register.

Example:

```
DINK32_750 >> pcimod 11 10
0x10 0x00000000 Base Address Register 0
New Value? 12345678

DINK32_750 >> pcidisp 11 10
0x10 0x12345678 Base Address Register 0
```

4.1.29 pciprobe ppr

pci probe command (on systems with a PCI bus; non-excimer build)

pciprobe

This command scans all legal PCI device numbers (from 10 to 31) and detects whether any device is attached to them. If a PCI device is found, the following information is displayed:

Device number, PCI address, Device Id and Vendor Id.

Example:

```
DINK32_750 >> pciprobe

Dev #  PCI ADDR      DEVICE ID     VENDOR ID     CLASS
===== ========== ========== ========== ==================
 11    0x80005800    0x0565        0x10ad        Bridge Interface
 12    0x80006000   (cannot probe self)
 15    0x80007800    0x2000        0x1022        Network Interface
```

4.1.30 regdisp rd

display registers

Syntax:
rd[-v][-e][r|rx|rx+|rx-ry|f|fx|fx+|fx-fy|sx|spr_name|northbridge|nb|mpc106|mpc107|mpc8240]

- **regdisp r** - entire general register family
- **regdisp r**x - one general purpose register
- **regdisp r**x+ - from rx to r31
- **regdisp r**x-**r**y - from rx to ry
- **regdisp f** - entire floating point family
- **regdisp f**x - one floating point register
- **regdisp f**x+ - from fx to f31
- **regdisp f**x-**f**y - from fx to fy
- **regdisp** SPR by name- view spr by name, such as hid0, contents.
- **regdisp s**x - one special purpose register
- **regdisp v**x - one altivec vector register
- **regdisp v+** - all altivec vector registers
- **regdisp -v** - verbose display, only valid if `env -c, env rdmode=e` is set.

This will display the contents of the specified registers. This command offers the user several options for viewing the registers. The whole family of general purpose registers or floating point registers can be viewed by typing "**regdisp** r" or "**regdisp** f" respectively. A single register can be viewed by specifying rx, fx, or sx, where the first character denotes the register family and the second character denotes the register number. Special purpose registers may be selected by their standard abbreviations as well as their register number.

The "plus" form displays the contents of the register family starting with the given register up to and including the last register in that family. The "range" form displays the contents of the registers from rx to ry or from fx to fy.

Note that the "entire family", "plus", and "range" forms are not available in the special purpose register family. This is due to the architectural design feature in which the special purpose registers all have unique register numbers and are not numbered sequentially.

The above parameter forms can be combined by separating them with a comma or white space. This will display multiple registers in different register families with one command. Note that the register display is aligned on an even-numbered register boundary, so if an even numbered register needs to be displayed, the odd-numbered register following it is

also displayed.

Most of the SPRs can suppress the verbose mode. This is still the default for compatibility purposes. If suppressed you can get verbose mode by with the following commands, `rd -v` and you can do `rd -e` to get the fields explained (where possible). Not all SPRs are quietened, just the most interesting ones. The 601 registers are not suppressed. To enable quiet mode use these commands: `env -c, env rdmode=e`, see Section 4.1.11, “env env".

Verbosity is suppressed for:

- XER SDR1 CR IABR PMC4 LR SRR0 FPSCR MMCR0 SIA
- CTR SRR1 MSR PMC1 THRM2 DSISR SPRGx SRx PMC2 THRM3
- DAR EAR HID1 MMCR1 THRM1 DEC PVR PMC3 ICTC
- L2CR USIA HASH1 HID1 DBATxU
- UPMC2 UMMCR1 HASH2 SDA DBATxL
- UPMC3 UMMCR0 IMISS DABR TBU
- UPMC4 DMISS ICMP IBATxU TBL
- UPMC1 DCMP RPA IBATxL MSSCR0
- MSSCR1 UBAMR PIR UMMCR2
- VRSAVE VSCR MMCR2 BAMR

Field descriptions are interpreted for:

- DBATxU DBATxL SRx PVR IBATxU IBATxL HID1 L2CR
- CR FPSCR MSR IABR THRM1 THRM2 THRM3 L2CR DABR MPC10x: PICR1/A8 and PICR2/AC

Examples:

```
DINK32_750 >> regdisp r1-r2,f4-f6,hid0
gpr00: 0x00000000 gpr01: 0x00060000
gpr02: 0x00000000 gpr03: 0x000068ac
fpr04: 0x0000000000000000 fpr05: 0x0000000000000000
fpr06: 0x0000000000000000 fpr07: 0x0000000000000000

DINK32_750> regdisp hid0
Hardware Implementation Dependent 0
------------------------------------------------------------
hid0 : 0x80010080
10000000000000010000000010000000
```

```
+............................... === master checkstop enable
+.............................. === microcode selftest checkstop
latch
+............................. === checkstop following a machine
check
+............................ === multi-side hit in the tlb
+........................... === multi-side hit in cache directory
+.......................... === sequencer hang
+......................... === dispatch time-out
+........................ === bus address parity error
+....................... === bus data parity error
+...................... === cache parity error
+..................... === invalid microcode instruction
+.................... === pio bus protocol error
+++................. === reserved
++++++++++++++.. === checkstop enables
+. === error in main cache (in array init)
+ === reserved

DINK32_750 >> regdisp r1 f2 r3 f4 r8 s5
gpr00: 0x00000000 gpr01: 0x00060000
fpr02: 0x0000000000000000 fpr03: 0x0200feed010cab00
gpr02: 0x00000000 gpr03: 0x000068ac
fpr04: 0x0000000000000000 fpr05: 0x0000000000000000
gpr08: 0x00000000 gpr09: 0x00000000

DINK32_750 >> regdisp r23+
gpr22: 0x2cab4dad gpr23: 0x00000000
gpr24: 0x00000000 gpr25: 0x00000000
gpr26: 0x00000000 gpr27: 0x00700007
gpr28: 0x00000000 gpr29: 0x00000000
gpr30: 0x00face00 gpr31: 0x00000000

DINK32_MAX >>rd v2
vr  2:  0x00000000   00000000   00000000   00abcdef

DINK32_MAX >>rm v2
vr  2:  0x00000000   00000000   00000000   00abcdef   : ? 12345678

DINK32_MAX >>rd v2
vr  2:  0x00000000   00000000   00000000   12345678

DINK32_MAX >>rd v
vr  0:  0xffffffff   ffffffff   ffffffff   ffffffff
vr  1:  0xffffffff   ffffffff   ffffffff   ffffffff
vr  2:  0x00000000   00000000   00123456   78abcdef
...
vr 29:  0xffffffff   ffffffff   ffffffff   ffffffff
vr 30:  0xffffffff   ffffffff   ffffffff   ffffffff
vr 31:  0x00000000   12345678   abcdef00   87654321
```

This example contrasts the verbose mode versus the non verbose mode of display. See the command env.

```
DINK32_ARTHUR >>rd msr
MSR : 0x00003930
  POW=0  EE=0  PR=0  FP=1  ME=1  FE0=1  SE=0
  BE=0  FE1=1  IP=0  IR=1  DR=1  RI=0  LE=0
  TLB/GPR=0    VMX=0  PM=0

DINK32_ARTHUR >>rd -v msr
Machine State Register
------------------------------------------------------------------------
------
MSR : 0x00003930
00000000000000000011100100110000
++++++++++++...................  === reserved
            +...................  === activates power management
             +..................  === tlb gpr overlay enable
              +.................  === reserved
               +................  === external interrupt enable
                +...............  === privilege level
                 +..............  === floating-point available
                  +.............  === machine check enable
                  +............  === floating point exception point 0
                     +..........  === single-step trace enable
                      +.........  === reserved
                     +........  === floating point exception point 1
                        +.......  === reserved
                         +......  === exception prefix
                         +.....  === instruction address translation
                           +....  === data address translation
                            +...  === reserved
                            +..  === performance monitor marked mode
                           +.  === RESET or MC exception recoverable
                               +  === little endian mode enable

DINK32_ARTHUR >>
```

4.1.31 regmod rm

modify registers

Syntax:
rm[-v][-e][r|rx|rx+|rx-ry|f|fx|fx+|fx-fy|sx|spr_name|northbridge|nb|mpc106|mpc107|mpc8240]

- **regmod r** - entire general register family
- **regmod r**x - one general purpose register
- **regmod r**x**+** - from rx to r31
- **regmod r**x-**r**y - from rx to ry
- **regmod f** - entire floating point family
- **regmod f**x - one floating point register
- **regmod f**x**+** - from fx to f31
- **regmod f**x-**f**y - from fx to fy
- **regmod** SPR by name- view spr by name, such as hid0, contents.
- **regmod s**x - one special purpose register
- **regmod v**x - one altivec vector register
- **regmod v+** - all altivec vector registers
- **regmod -v** - verbose display, only valid if `env -c, env rdmode=e` is set.

This command modifies the contents of the specified registers. r, f will access the entire general purpose or floating point family; rx, fx, sx, or spr_name will access the specified register. Multiple parameters may be entered. The user can enter <return> to leave data unmodified, or an \"x\" to quit. If the \"+\" form is used, the command will display one register at a time and prompt the user for a new value. It will continue to do this for the entire family starting with the specified register. If the two-address version is used, the command will display one register at a time and prompt the user for a new value. It will do this for all the registers specified in the range.

Note that special purpose, and mpc106 registers can only be accessed individually and not as a family or with the \"+\" or range forms. mpc106 supports -b, -h, -w options for byte, halfword, and word access.

Most of the SPR's can suppress the verbose mode, see Section 4.1.30, "regdisp rd".

Examples:

```
DINK32_603e >>rm r6
gpr06 = 0x00000000 : ? 12345678
```

```
DINK32_603e >>rd r6
gpr06: 0x12345678 gpr07: 0x00000000

DINK32_603e >>rm mpc106 70
ADDR. VALUE DESCRIPTION
===== ===== ===========
0x70 0x0000 Power management config. 1
new value ? 1234

DINK32_603e >>rd mpc106 70
ADDR. VALUE DESCRIPTION
===== ===== ===========
0x70 0x1234 Power management config. 1

DINK32_603e >> rm f4-f7, s8
"displays the contents of floating point register 4 and prompts the
user for new data, then increments through registers 5-7. Then the
contents of s8 are displayed and can be modified.”

DINK32_603e >> rm mpc106 -h 0xaa
" sets the contents of the mpc106 register in halfword starting at
offset 0xaa.”
```

4.1.32 rtc rtc

modify/display real time clock <yellowknife and sandpoint only>

rtc [-s][-w]

The rtc command allows setting or displaying the real-time clock available on the Yellowknife or Sandpoint systems.

- **-s** Sets the clock; you are prompted for the date and time.
- **-w** Watches the clock. The date and time are repeated until a key is pressed on the keyboard.

If no option is given, the current date and time are displayed.

Example:

```
DINK32_KAHLUA >>rtc
2000/00/14 03:38:14
DINK32_KAHLUA >>rtc -s
Year   : 99
Month  : 06
Day    : 21
Hour   : 11
Minute : 48
Second : 00
Set to: 1999/06/21 11:48:00
1999/06/21 11:48:00
DINK32_KAHLUA >>
```

4.1.33 runalias ra

run alias

runalias

This instruction will read in the string which the user has defined as an alias. Then, the commands in this string will be executed sequentially. Also see the da and env commands.

Example:

```
DINK32_750 >> runalias

The runalias command can also be embedded within a command line. For
example, if the alias string has previously been defined as
tr +; rd r

Typing the command:

DINK32_750 >> log; trace 2100; runalias; log
is identical to typing

DINK32_750 >> log; trace 2100; tr +; rd r; log
See defalias for a complete example.
```

4.1.34 setbaud sb

displays or changes the speed of the serial port <mdink32 compatible>

- setbaud [-h | -k]
- setbaud [-h | –k] rate

This command sets the baud rate for the host serial port (-h) or the keyboard serial port (-k) by specifying the appropriate flag followed by a valid rate (2400,4800,9600,19200,38400,57600). If only a flag is specified, the current baud rate for that serial port is returned.

- Example: "`sb -h`" would return the current baud rate for the host serial port.
- Example: "`sb -k 9600`" would set the host serial port baud rate to 9600.

4.1.34.1 Host versus Keyboard.

Used by `log`, `sb`,`dl`, and `tr` commands. See Section 4.1.10, “download dl", Section 4.1.17, “log log", and Section 4.1.37, “transpar tm".

- The keyboard serial port (-k) indicates serial port com1, which is used for normal communication between the terminal emulator and the evaluation board. Thus `sb -k` and `dl -k` indicate to use the current serial port. Thus for, `dl -k`, use the terminal emulator, `transfer send text file,` feature on the terminal emulator connected to com1.
- The host serial port (-h) indicates serial port com2, which is not normally used. One can connect another terminal emulator to this serial port and with the dl -h command, download a file. This port is only available with the Sandpoint and Yellowknife platforms.

NOTES:

- The maximum baud rate on the Yellowknife and Sandpoint platform is 38400.
- The Excimer and Maximer platform will not return the current baud rate correctly.
- The default baud rate on all platforms is 9600.

Examples:

```
MDINK32_603e >>setbaud -k 57600

 Baud rate changing to 57600...BØ

<NOTE: user must then change the baud rate on the terminal to
correspond to 57600>
```

```
MDINK32_603e >>
```

4.1.35 symtab st

displays DINK32 symbol table information

- symtab -c
- symtab -d

This command shows selected DINK symbols and user defined symbols and their associated addresses. User symbols can be defined by the as command. The -c option is to clear all user symbols. The -d option is to delete a single user symbol. The user cannot delete or clear DINK's symbols. The symbols in the table can be used as the address (@symbol) of the branch instruction while executing the as command.

Examples:

```
DINK32_603e >>as 60000+
0x00060000 0xff0000ef fsel.       f24, f00, f00, f03        br1:xor
r1,r2,r3
0x00060000 0xff0000ef BRANCH LABEL br1:
0x00060000 0xff0000ef fsel.     f24, f00, f00, f03      xor r3,r4,r5
0x00060004 0xffc037fc fnmsub      f30, f00, f31, f06        br2:xor
r1,r5,r6
0x00060004 0xffc037fc BRANCH LABEL br2:
0x00060004 0xffc037fc fnmsub      f30, f00, f31, f06        x

VERIFYING BRANCH LABELS.....

DONE VERIFYING BRANCH LABELS!
DINK32_603e >>ds 60000
0x00060000 0x7c832a78 BRANCH LABEL br1:
0x00060000 0x7c832a78 xor         r03, r04, r05
DINK32_603e >>as 60100
0x00060100 0x85ffffc4 lwzu        r15, 0xffc4( r31 )        br3:xor
r5,r6,r7
0x00060100 0x85ffffc4 BRANCH LABEL br3:
0x00060100 0x85ffffc4 lwzu        r15, 0xffc4( r31 )        x

VERIFYING BRANCH LABELS.....

DONE VERIFYING BRANCH LABELS!
DINK32_603e >>st
Current list of DINK branch labels:
         KEYBOARD:    0x0
         get_char:    0x1e5e4
       write_char:    0x5fac
        TBaseInit:    0x39c4
   TBaseReadLower:    0x39e8
   TBaseReadUpper:    0x3a04
     CacheInhibit:    0x3a20
    InvEnL1Dcache:    0x3a40
      DisL1Dcache:    0x3a88
```

```
    InvEnL1Icache:    0x3aac
      DisL1Icache:    0x3b00
        BurstMode:    0x3bfc
         RamInCBk:    0x3c3c
       RamInWThru:    0x3c7c
        dink_loop:    0x5660
      dink_printf:    0x6368

Current list of USER branch labels:
              br1:    0x60000
              br2:    0x60004
              br3:    0x60100
DINK32_603e >>st -d br2
DINK32_603e >>st
Current list of DINK branch labels:
         KEYBOARD:    0x0
         get_char:    0x1e5e4
       write_char:    0x5fac
        TBaseInit:    0x39c4
   TBaseReadLower:    0x39e8
   TBaseReadUpper:    0x3a04
     CacheInhibit:    0x3a20
    InvEnL1Dcache:    0x3a40
      DisL1Dcache:    0x3a88
    InvEnL1Icache:    0x3aac
      DisL1Icache:    0x3b00
        BurstMode:    0x3bfc
         RamInCBk:    0x3c3c
       RamInWThru:    0x3c7c
        dink_loop:    0x5660
      dink_printf:    0x6368

Current list of USER branch labels:
              br1:    0x60000
              br3:    0x60100
DINK32_603e >>st -c
DINK32_603e >>st
Current list of DINK branch labels:
         KEYBOARD:    0x0
         get_char:    0x1e5e4
       write_char:    0x5fac
        TBaseInit:    0x39c4
   TBaseReadLower:    0x39e8
   TBaseReadUpper:    0x3a04
     CacheInhibit:    0x3a20
    InvEnL1Dcache:    0x3a40
      DisL1Dcache:    0x3a88
    InvEnL1Icache:    0x3aac
      DisL1Icache:    0x3b00
        BurstMode:    0x3bfc
         RamInCBk:    0x3c3c
       RamInWThru:    0x3c7c
        dink_loop:    0x5660
      dink_printf:    0x6368
```

```
Current list of USER branch labels:
DINK32_603e >>
```

4.1.36 tau tau

TAU Thermal Assist Unit CONTROL

tau [-c cal][-w][-fh]

Description: This command displays or calibrates the TAU (Thermal Assist Unit). If no option is entered, the current temperature is displayed (with or without calibration). TAU calibration values are always saved in the environment variable TAUCAL (if ENV storage is available).

Flags:

- -c Calibrate the TAU to the actual temperature (in ^C).
- -w Watch the TAU (until a key is pressed)
- -fh Show results in Fahrenheit.

TAU calibration values are always saved in the environment variable TAUCAL, if ENV storage is available.

Example:

```
DINK32_ARTHUR >>tau
Tjc = 58 ^C (uncalibrated)
DINK32_ARTHUR >>tau -c 18
Tjc = 18 ^C
DINK32_ARTHUR >>tau
Tjc = 18 ^C
DINK32_ARTHUR >>tau -fh
Tjc = 32 ^F
```

4.1.37 transpar tm

(transparent mode for com2; non-excimer build)

- transpar

This command will put DINK32 into a transparent mode, giving the user direct access to the host. In other words, as the user types data into the keyboard, that data is sent directly to the host serial port. In addition, data that comes in from the host serial port will be forwarded to the keyboard serial port. The user can exit from transparent mode by typing <ctrl>-a.

See Section 4.1.10, “download dl", and Section 4.1.37, “transpar tm"

Example:

```
DINK32_750 >> tm

<cntr-a>
```

4.1.38 trace tr

single step trace

- **trace** address
- trace +

This allows the user to single-step through a user program. The microprocessor will execute a single instruction, and then return control back to the firmware. If a specific address is given, then a single instruction is executed from that address. However, if the “plus” form is used, then the address of the instruction to execute is derived from bits 0-29 of the SRR0 (Machine Status Save / Restore) register. After the instruction has been executed, control is returned to the firmware (DINK32) and the user can examine the programming model or continue to trace through instructions.

Example:

```
DINK32_750 >> ds 2100
0x00002100 0x7c0802a6 mfspr r00, s0008

DINK32_750 >> trace 2100
A Run Mode or Trace exception has occurred.
Current instruction Pointer: 0x00002104 stw r13, 0xfff8(r01)

DINK32_750 >> trace +
A Run Mode or Trace exception has occurred.
Current instruction Pointer: 0x00002108 add r03, r00, r01

DINK32_750 >> .
A Run Mode or Trace exception has occurred.
Current instruction Pointer: 0x0000210c mfspr r04, s0274
```

Chapter 5 DINK32 Command Form Summary

1. **.(period) .** - repeat last command
2. **about about** - displays version information
3. **assemble as -** address- assemble at one address
4. **bkpt bp** - set, delete, list breakpoints
5. **defalias da -** command list - define alias for listed commands
6. **devdisp dd** list - display contents of device registers
7. **devmod dm** list - modify device data in device registers.
8. **devtest dev** list - perform an I/O test on Kahlua
9. **disassem ds -** address - disassemble at one address
10. **download dl** - download S-Record file to board RAM or flash
11. **env env** - Environment controls
12. **flash fl -** flash commands
13. **fupdate fu** - copy PCI boot rom to local PPMC
14. **fw fw -e** - erase all of Flash memory and load RAM to ROM (mdink32)
15. **go go -** address - execute from given address
16. **help he -** command - show more information on command
17. **log log** - record debug session to host
18. **memdisp md -** address - display memory at one address
19. **memfill mf -** start, end, data - fill memory block with data pattern
20. **meminfo mi -** displays information about the memory settings
21. **memod mm** address - modify memory at one address
22. **memove mv -** start, end, dest - move memory block to destination
23. **memsrch ms -** start, end, data - search memory block for data
24. **memtest mt -** perform various memory tests on local memory or device registers.
25. **menu me -** show list of available commands
26. **pciconf pcf -** display all config registers of a PCI device
27. **pcidisp pd** - display contents of a PCI config register
28. **pcimod pm -** modifies PCI device config register data
29. **pciprobe ppr -** scans for PCI devices
30. **regdisp rd -** display entire general register family

31. **regmod rm -** modify entire general register family
32. **rtc rtc -** set and/or display the real time clock
33. **runalias ra -** execute the commands in the alias
34. **setbaud sb -** display or change the serial port baud rate
35. **symtab st -** displays DINK32 symbol table
36. **tau tau** display temperature from the Thermal Assist Unix
37. **transpar tm -** transparent mode Yellowknife only
38. **trace tr -**address trace from given address

Chapter 6 Utilities

6.1 S-Record Compression/Decompression

6.1.1 Overview

To assist in the compression of S-Record files, a conversion utility is included with the source code for DINK32. The dcomp utility is written in portable ANSI-compliant C, which is easily compiled under UNIX or a PC. The dcomp utility performs both compression and decompression of S-records. It is provided so that the user may compress their S-record before downloading them to the board. They will automatically detected as compressed S-records by DINK and decompressed before being written to the proper memory locations.

6.1.2 Building

6.1.2.1 Files

The dcomp package consists of two c files, dc_tb.c, dc_unix.c and three header files, dink.h, errors.h, and sublib.h. However, these header files call other header files, so dcomp must be built in the dink32 source directory.

6.1.2.2 Modification of header file

The dink.h file uses the #define macro `ON_BOARD`, which is set by config.h. Since dcomp must be built with ON_BOARD undefined, it is necessary to modify the config.h file. Ensure that you return config.h to it's released form before trying to build dink32 or mdink32. At about line 84 of the config.h file, you will find the line, `#define ON_BOARD`. Comment out this line. After the change this code will be:

/* For trying to build a version that runs under Unix,

comment out the #define for ON_BOARD. */

/* #define ON_BOARD */

6.1.2.3 Build command

Use any available c compiler, such as UNIX cc, or gcc, or Metaware, or PC compiler, cl (Microsoft c compiler). This description uses the generic `CC` for the compiler invocation.

CC dc_unix.c dc_tb.c -o dcomp

This command will build the executable `dcomp`. Dcomp will run on the machine on which it is built. It does not run on the Excimer and Maximer or Yellowknife and Sandpoint board.

6.1.3 Command syntax

```
Usage:
        dcomp -options <input_file >output_file.
Options:
        -c Compress an SRecord file.
  -e Expand a previously compressed file into an       SRecord file.
Examples:
        dcomp -c <a.out.mx >a.out.cmp
        dcomp -e <a.out.cmp >a.out.mx

Note that this program uses stdin and stdout, so the < symbol and >
symbol are required.

example:

Unix $  dcomp -c <dink32.src >c_dink32.src
This command will compress the file dink32.src and create the
compressed file c_dink32.src.

Unix $  dcomp -e <c_dink32.src >e_dink32.src
This command will decompress (expand)  the file c_dink32.src and
create the decompressed (i.e. expanded) file e_dink32.src.

e_dink32.src is equivalent to the original dink32.src file.

UNIX $    ls -l c_dink32.src e_dink32.src dink32.src
-rw-r-----  1 maurie     361189 Jan 22 09:43 c_dink32.src
-rw-r-----  1 maurie     597181 Jan 22 09:41 dink32.src
-rw-r-----  1 maurie     597181 Jan 22 10:41 e_dink32.src
```

6.2 bat_decode

6.2.1 Overview

The bat_decode program will decode BATU and BATL hex values supplied in hex. The value of the bats will be displayed and described.

6.2.2 Building

To compile and link the program use this command. This description uses the generic CC for the compiler invocation.:

CC bat_decoder.c -o bat_decoder.out

6.2.3 Command syntax

```
Usage:
```

bat_decoder.out < inputfile > outputfile

```
Examples:

bat_decoder.out < bat.in > bat.out


Note that this program uses stdin and stdout, so the < symbol and >
symbol are required.

example:
```

Input description:

<an integer> How many bat pairs per line are supplied?

<some_description>: <batlower_value> <batupper_value>

where:

<some_description> has no spaces or tabs (use underscore to connect names), must be 19 characters or less. The character array has only 20 characters. For bigger descriptions this line can be changed.

<batlower_value> is a hex value

<batupper_value> is a hex value

As an example, if you wanted to decode two pairs of bats:

```
2
ibat0: 10000001 10000fff
dbat0: 1000001a 10000fff
```

If you want a description line you can use batlower=batupper=0 as in:

```
3
This_is_a_test 0 0
ibat0: 10000001 10000fff
dbat0: 1000001a 10000fff
```

The output is:

```
Bat Decoder - enter the bat values and display the meaning
    IBAT and DBAT have same meaning
    Format:  description: upperbat_value lowerbat_value
    How many bat entry pairs, one pair per line
Please enter the Lower  and Upper bat value in hex
This_is_a_test: Decoding the bat
Both bats are zero, Disabled

Please enter the Lower  and Upper bat value in hex
ibat0: Decoding the bat
  For batu = 0x10000fff
```

```
     BEPI Logical address is     = 0x10000000
     BL Block Length is  = 0x3ff        128 MB
         Range is   = 0x10000000 - 0x17ffffff
     VS is  = 0x1 Supervisor mode access
     VP is  = 0x1 User mode access
  For batl = 0x10000001
     BRPN Physical address is     = 0x10000000
     WIMG     = 0x0
        W off Not Write Through i.e. Write back
        I off Not Cache Inhibited, i.e. use cache
        M off Not Memory Coherent, i.e. non-coherent
        G off Not Guarded, i.e. unguarded
     PP Block Access Protection Control    = 0x1
         Read Only
Please enter the Lower  and Upper bat value in hex
dbat0: Decoding the bat
  For batu = 0x10000fff
     BEPI Logical address is     = 0x10000000
     BL Block Length is  = 0x3ff        128 MB
         Range is   = 0x10000000 - 0x17ffffff
    VS is  = 0x1 Supervisor mode access VP is  = 0x1 User mode access
  For batl = 0x1000001a
     BRPN Physical address is     = 0x10000000
     WIMG     = 0x3
        W off Not Write Through i.e. Write back
        I off Not Cache Inhibited, i.e. use cache
        M on  Memory Coherent
        G on  Guarded
     PP Block Access Protection Control    = 0x2
         Read and Write
```

6.3 Memory Test

A simple memory test is included in DINK as an option. It is enabled via a #define in config.h. If MEMORY_TEST is defined, then, before DINK is copied from ROM to RAM a memory test will be performed from address 0x0 to the MEMORY_END || 0x0000 location. If MEMORY_END is defined as 0x7 then the test is performed between 0x0 and 0x70000. The address of the memory location is written into the memory location and then read back. If an error is detected then the verify loop will go to an infinite loop located at error_memory_test. The location of this loop can be found in the map file and can easily b

The following listing will show up on the flash screen:

Memory test performed from 0x00000000 - 0x70000

The user may feel free to enhance the memory test algorithm by adding additional test into the memory_test function located in except2.s

Note: The user must ensure that the ending address (MEMORY_END) is valid or the debug

monitor may not boot.

There is also a memory test command, mt.

Chapter 7 User Program Execution

The DINK32 firmware includes a file transfer utility that allows the user to download S-Record files from the host to the target board.

This download function stores the S-Records into memory starting at the address given in the S-Record file. The user can then use the **go** or **trace** command to execute the user program. Listed below are the steps to take to execute a user program.

7.1 Execution Steps

Download the user program to run on DINK32.

1. Create an executable S-Record file of the user program to be run on DINK32. Most modern compiler vendors supply a facility for converting an executable or generating an S-Record file directly. E.g. Gnu supplies an elfhex tool, Metaware supplies an elf2hex tool. Ensure that the S-Record is a Motorola type S-Record file.
2. Download the s-record file into memory on the target board using the DINK32 download command. The same command is used for compressed s-Record files. Using a terminal program, receive an S-Record file into the target board. The recommeded settings are databits = 8, parity = none, stopbits = 1, flowcontrol = hardware (although none will work), and baud rate = 57600 on excimer, 38400 for yellowknife.
3. This optional step may be desired. The default baud rate is 9600, however, DINK32 is capable of downloading at 57600 on Excimer and Maximer and 38400 on Yellowknife and Sandpoint. For large programs, we suggest changing the baud rate to 57600 before the download. One can start and debug the downloaded program in any baud rate. However before pressing the reset button restore the baud rate to 9600.
4. go 90000. One needs to build the executable program so that it starts at address 0x90000. Upon invocation, the program will use r1 as the stack pointer, which will have been set to 0x8fff0 by DINK32.

Note: Hardware flow control is implemented on the Excimer and Maximer platform and is required for file downloading.

Example:

```
DINK32_750 >> sb -k 57600
Change the baud rate to 57600. Also change the setting on your
terminal emulator.

DINK32_750 >> dl -k
Downloading in s-record format.
```

```
Download Complete.

DINK32_750 >>
```

Set breakpoints, if necessary, and execute the user program at the location to which it was downloaded using **go** or **trace**.

```
DINK32_750 >> go <address>
DINK32_750 >> trace <address>
```

Chapter 8 Errors and Exceptions

8.1 Error Codes

8.1.1 Parser Errors

- 0xFB00 UNKNOWN_COMMAND unknown command
- 0xFB01 UNKNOWN_REGISTER unknown register
- 0xFB02 ILLEGAL_RD_STAGE cannot specify whole register family in range
- 0xFB03 ILLEGAL_REG_FAMILY cannot specify a range of special registers
- 0xFB04 RANGE_CROSS_FAMILY cannot specify a range across register families
- 0xFB05 UNIMPLEMENTED_STAGE invalid rd or rmm parameter format
- 0xFB06 UNKNOWN_OPERATOR unknown operator in arguments
- 0xFB07 INVALID_FILENAME invalid download filename

8.1.2 Errors from Error Checking Toolbox

- 0xFD00 INVALID NOT valid
- 0xFD01 VALID valid
- 0xFD02 INVALID_SIZE the input was not 8 characters long
- 0xFD03 OUT_OF_BOUNDS_ADDRESS the address given falls outside of valid memory defined by MEM_START to MEM_END
- 0xFD04 INVALID_HEX_INPUT one of more of the characters entered are not valid hex
- characters. Valid hex characters are 0-9, A-F, a-f
- 0xFD05 INVALID_REGISTER a given register does not exist
- 0xFD07 NOT_WORD_ALIGNED the given address is not word-aligned. A word-aligned address ends in 0x0,0x4,0x8,0xc
- 0xFD08 REVERSED_ADDRESS the starting address is greater than the ending address.
- 0xFD09 RANGE_OVERLAP the address specified as the destination is within the source

8.1.3 addresses

- 0xFD0A ERROR an error occurred
- 0xFD0B INVALID_PARAM invalid input parameter

8.1.4 Get Argument Errors

- 0xFE00 INVALID_NUMBER_ARGS invalid number of command arguments
- 0xFE01 UNKNOWN_PARAMETER unknown type of parameter

8.1.5 Tokenizer Toolbox Errors

- 0xFF00 ILLEGAL_CHARACTER unrecognized character in input stream
- 0xFF01 TTL_NOT_SORTED token translation list not sorted
- 0xFF02 TTL_NOT_DEFINED token translation list not assigned
- 0xFF03 INVALID_STRING unable to extract string from input stream
- 0xFF04 BUFFER_EMPTY input buffer is empty
- 0xFF05 INVALID_MODE input buffer is in an unrecognized mode
- 0xFF06 TOK_INTERNAL_ERROR internal tokenizer error
- 0xFF07 TOO_MANY_IBS too many open input buffers
- 0xFF08 NO_OPEN_IBS no open input buffers

8.1.6 Screen Toolbox Errors

- 0xFC00 RESERVED_WORD used a reserved word as an argument

8.1.7 Breakpoint Errors

- 0xFA00 FULL_BPDS breakpoint data structure is full

8.1.8 Download Errors

- 0xF900 NOT_IN_S_RECORD_FORMAT not in S-Record Format
- 0xF901 UNREC_RECORD_TYPE unrecognized record type
- 0xF902 CONVERSION_ERROR ascii to int conversion error
- 0xF903 INVALID_MEMORY bad S-Record memory address

8.1.9 Compression and Decompression Errors

- 0xF800 COMP_UNK_CHARACTER unknown compressed character
- 0xF801 COMP_UNKNOWN_STATE unknown binary state
- 0xF802 NOT_IN_COMPRESSED_FORMAT not in compressed S-Record format

8.1.10 DUART Handling Errors

- 0xF700 UNKNOWN_PORT_STATE unrecognized serial port configuration

- 0xF600 TM_NEEDS_BOTH_PORTS transparent mode needs access to two serial ports

8.1.11 Register Errors

- 0xF600 SPR_NOT_FOUND cannot find register in special purpose register file

8.1.12 Flash Errors

- 0xF100 FLASH_ERROR error in flash command activity

8.2 Exceptions

There are twenty one exceptions in this version of DINK32. A message indicating which exception has occurred is displayed for all of them except System Reset.

- 0x0100 **System Reset**
- 0x0200 **Machine Check**
- 0x0300 **Data Access**
- 0x0400 **Instruction Access**
- 0x0500 **External Interrupt**
- 0x0600 **Alignment**
- 0x0700 **Program**
- 0x0800 **Floating-Point Unavailable**
- 0x0900 **Decrementer**
- 0x0A00 **I/O Controller Interface Error**
- 0x0C00 **System Call**
- 0x0D00 **Trace**
- 0x0E00 **Floating Point Assist**
- 0x0F00 **Performance Monitor**
- 0x1000 **Instruction Translation Miss**
- 0x1100 **Data Load Translation Miss**
- 0x1200 **Data Store Translation Miss**
- 0x1300 **Instruction Address Breakpoint**
- 0x1400 **System Management Interrupt**
- 0x1600 **Java Mode denorm detection**
- 0x2000 **Run Mode or Trace**

System Reset occurs when the software is booted up or the evaluation board is reset. The other exceptions occur due to interrupts or errors in the execution of the code.

When using DINK, the user is notified of exceptions by a message that appears on the terminal. Control is returned to the firmware. If the exception was caused by the completion of a trace or by arriving at a breakpoint during execution of the user's code, the user can continue testing. Otherwise the user may need to modify the code to correct a problem and download the program again to resume testing.

For details on what causes each exception, see the Programming Environments Manuals (PEM) and the appropriate PowerPC User's Manual for the part in question.

Chapter 9 Restrictions

9.1 Special Purpose Registers

There are four Special Purpose General Registers (SPRGs), numbered 0 through 3.
DINK32 makes use of SPRG2 and SPRG3, so any user values placed into these two registers will be destroyed whenever control is returned to DINK32. The user is encouraged to place any values that are of interest or necessity into only SPRG0 and SPRG1, although the user can use the other two SPRGs for calculations or temporary storage.

Chapter 10 Known Bugs

10.1 Known Bugs

- **setbaud** On Excimer and Maximer platform the sb –h or –k without a baud rate will always return 0.
- All of the user caches may not be flushed on exceptions and breakpoints.
- The assembler will silently ignore any register it doesn't recognize, inserting 0 in it's place. For example: mfspr r3,1010 will substitute mfsrp r3,0.
- **env** is not in the help menu, however, help env is available.
- The gcc built version of DINK32 srecord and elf file
  - is 50% larger than the Metaware build

Appendix A Adding Commands and Arguments

A.1 Help

All help information is displayed by the help.c file. The help file has two types of help, the main summary menu and the specific help information for a specific command.

A.1.1 Help Menus

There are two summary help menus, one for dink32 and the other for mdink32. They are discriminated by the "dink_type" variable. dink_type = 0 for dink32 and dink_type = 1 for mdink32. Simply add the summary command to the appropriate menu. The menus are simply PRINT statements in the function menu().

There is no distinction between dink32 and mdink32 for the specific command help file. Simply build a function called `help_<command>` such as `help_info().` This function consists entirely of PRINT commands describing the new command.

To make the specific help commands available, specify the help function with the command function in the command_tb.h file. There are two steps.

1. add an extern for the command and help functions. Such as extern STATUS par_bm() and extern void help_bm() for the benchmark command.
2. Add the command name, tag, function and help function name to the structure cmd_struct dink_cmds.
   - struct cmd_struct dink_cmds[NUM_CMD] = {
   - {"ab", "about", NO_TAG, par_about, help_about},
   - {"as", "assemble", MODIFY_TAG, par_asdm, help_asm},
   - {"ds", "disassem", DISPLAY_TAG, par_asdm, help_disasm},

The entry in this table will "register" your command and your help file. The members of each entry are: short_name, long_name, tag, function_name, and help_function_name. The tag is used to specify the argument list for your function and is invoked in the par_head_parser function in par_tb.c. NO_TAB indicates that no command pointer is sent to your function, i.e. define your function with a null argument list, as STATUS newcommand(); CMD_TAG will send you a pointer to a string with the invocation command from the command line, but not the argument list. I.e. define your function with a string pointer, such as STATUS newcommand(char *dink_cmd), dink_cmd will be a null terminated string containing only the invocation command. Such as dink_cmd - "new_command\0".

Example (existing about command)

```
help.c

void help_about()
{
PRINT("ABOUT: \n");
PRINT("====== \n");
PRINT("Mnemonic: about, ab \n");
PRINT("Syntax: ab \n");
PRINT("Description: This command displays the general information
");
PRINT("on DINK32.\n");
PRINT("Example: \"ab\" would display the opening screen of DINK32.
\n");
}
```

Example (fl command)

```
help.c


void help_flash()
{
PRINT("FLASH COMMANDS: \n");
PRINT("====== \n");
PRINT("Mnemonic: flash, fl \n");
PRINT("Syntax: fl -flags -o value -s sector number\n");
PRINT("Description:  This  command  performs  actions  to  the  flash
memory\n");
PRINT("Flags:  -e   erase     erase all of flash\n");
PRINT("Flags:  -cp  copy      copy MDINK from RAM to Flash\n");
PRINT("                Required Flags:  -o <value> copy address in
flash\n");
PRINT("          Optional Flags: -e      erase flash first\n");
PRINT("Flags:  -sp  protect   indicated sector\n");
PRINT("           Required Flags:  -n <value> sector number 0-18\n");
PRINT("Flags:  -su  unprotect indicated sector\n");
PRINT("           Required Flags:  -n <value> sector number 0-18\n");
PRINT("Flags:  -se  erase     indicated sector\n");
PRINT("           Required Flags:  -n <value> sector number 0-18\n");
PRINT("  Example:  fl -sp -n 5 - sector protect sector 5 \n");
}
```

A.2 Input Arguments

Now we are ready to specify input arguments. Arguments are effected by entries in two tables, one is toks.h and the other is toks.c. The toks.h table is a set of lines of #define macros. Each argument is treated as a member of a symbol table called SYMBOL_BASE_TOK. The base of the table is defined as some value. There are several

such bases for various other symbols, such as the REG_GEN_BASE_TOK. By reading the comments at the beginning of the file, we ascertain that this is a scheme to guarantee that all tokens (command arguments, register names, etc.) have a unique integer value that can be used by the tokenizer to uniquely identify any symbol desired by the dink32 code.

A.2.1 Input Token Facility

Specify the name of your token with a #define macro, and give it the value of one more than the previous values.

> Note: either do not exceed the MAX_SYMBOLS_TOKENS size defined in toks.h, currently set at 32 or increase the value.

example:

```
       toks.h
#define  DASH_TO SYMBOL_BASE_TOK + 2 /* symbol2 - the dash(-) symbol
*/
...
#define   BOTH_TOK SYMBOL_BASE_TOK + 8 /* symbol8 to select both
serial ports */
#define  HOST_TOK SYMBOL_BASE_TOK + 9 /* symbol9 select only the host
port */
#define  KEY_TOK SYMBOL_BASE_TOK + 10 /* symbol10 select only the
keyboard */
#define  QUEST_TOK SYMBOL_BASE_TOK + 11 /*  symbol11  the  question
mark (?) */
```

This example is for the si (setinput command). It defines the dash token and the k,h,and ? command arguments, which are invoked as:

si [-k | -h | -?].

The ADD_TOKEN macro in toks.c adds these symbols to tokenizer so that the function can search the argument list.

example:

```
       toks.c
ADD_TOKEN("both",BOTH_TOK, &i); /* symbol8 - to select both serial
ports */
ADD_TOKEN("host",HOST_TOK, &i); /* symbol9 - to select only the host
port */
ADD_TOKEN("key",KEY_TOK,  &i);  /*  symbol10  -  to  select  only  the
keyboard port */
ADD_TOKEN("k",KEY_TOK, &i); /* same as above */
ADD_TOKEN("?\0",QUEST_TOK, &i); /* symbol11 - the question mark (?)
symbol */
```

Note that the token is a null terminated string, not a single character. In this example, we

are looking for the strings "both", "host", "key", "k", and "?" and the comment tells us which symbol it refers to in the toks.h file.

There are at least two ways to get these tokens. par_si uses the getarg_tok function as this code fragment shows:

```
if( (status = getarg_tok(&state))!=SUCCESS) return status;

              PRINT("Set Input Port : ");
              switch(state)
              {
              case BOTH_TOK : duart_configuration = BOTH_PORTS;
```

A more extensive method is to use the functions tok_is_next_token and tok_get_next_token.

These examples are from the new flash_commands that will be in the next release.

The code shown below extracts the arguments from the command line.

This code will parse the line:

```
fl -sp -n 5
however, it will give an error for these lines:
fl -sp -n f1   hex value
fl -xp -n 1    -xp instead of valid  -sp | -su | -se  etc
fl -sp  1      missing -n
fl -sp -n      missing a decimal value
```

toks.h:

```
#define  SECTOR_PROTECT_TOK  SYMBOL_BASE_TOK + 15/* symbol15 - 'sp'
for sector protect */
#define  SECTOR_UNPROTECT_TOK  SYMBOL_BASE_TOK + 16/*   symbol16   -
'su' for sector unprotect */
#define  SECTOR_ERASE_TOK SYMBOL_BASE_TOK + 17/* symbol17 - 'se' for
sector erase */
#define  FLASH_COPY_TOK SYMBOL_BASE_TOK + 18/* symbol18 - 'cp' for
flash copy */
#define  SECTOR_NUMBER_TOK  SYMBOL_BASE_TOK + 19/*  symbol19  -  'n'
for sector number */
```

toks.c

```
ADD_TOKEN("sp",SECTOR_PROTECT_TOK, &i);/* symbol15 - Sector Protect
*/
ADD_TOKEN("su",SECTOR_UNPROTECT_TOK,   &i);/*   symbol16   -   Sector
Unprotect */
ADD_TOKEN("se",SECTOR_ERASE_TOK, &i);/* symbol17 - Sector Erase */
ADD_TOKEN("cp",FLASH_COPY_TOK, &i);/* symbol18 - Sector Erase */
ADD_TOKEN("n",SECTOR_NUMBER_TOK,  &i);/*  symbol19  -  Sector  Number
value */
```

fl.c

This code checks the first token for a dash, then the second token for one of sp, su, se, e, cp. The function get_sector_number gets the sector number specified.

```
       if (!(tok_is_next_token(DASH_TOK)))
       {
       PRINT("Must specify [-sp | -su | -se | -e | -cp]\n");
              return FAILURE;
       }

if ((status = tok_get_next_token(&token, temp)) != SUCCESS)
          {
       PRINT("Must specify [-sp | -su | -se | -e | -cp]\n");
              return status;
          }

              switch (token)
              {
                case SECTOR_PROTECT_TOK:
                  get_sector_number(&sector_number);
                  PRINT("Got -sp, -n is %d\n",sector_number);
                  break;
                case SECTOR_UNPROTECT_TOK:
                  get_sector_number(&sector_number);
                  PRINT("Got -su, -n is %d\n",sector_number);
                  break;
```

This code gets the next token, which must be a -n and then gets the next token which must be an ascii string containing one valid decimal number, which will be converted to int by the ascii_to_int_dec function.

```
if (!(tok_is_next_token(DASH_TOK)))
  {
     PRINT("Must specify [-n ]\n");
     return FAILURE;
  }

   if ( (status = tok_get_next_token(&token, temp))
                          == SUCCESS)
   {
   if (token != SECTOR_NUMBER_TOK)
   {
     PRINT("Must specify [-n ]\n");
     return FAILURE;
   }

   if ( (status = tok_get_next_token(&token, temp)) != SUCCESS)
   {
     return FAILURE;
   }
            if  ((status  =  ascii_to_int_dec(temp,  sector_number,
```

```
strlen(temp)))
                        != SUCCESS)
                {
                     PRINT("Error getting decimal value.\n");
                     return (status);
                }
```

Appendix B Adding ERROR Groups to MDINK/DINK32

B.1 Error Group Files

The two files used for adding an ERROR grouping to dink32 and mdink32 are err_tb.h and errors.h.

Both files contain the defined macro, NUM_ERRORS, and both must be changed whenever a new error group is added.

B.1.1 err_tb.h

About line 30, increment NUM_ERRORS by the number of error groups you are adding. In this case, change it from 46 to 47.

#define NUM_ERRORS 47

Now add the new entry to the structure err_element. This structure has two parts, the code and a string constant for the error message. Add the message

{FLASH_ERROR, "FLASH error") /* 46 */

It is a good idea to add a comment to the end of any added lines for the struct entries with the error number.

B.1.2 errors.h

About line 51 increment the defined macro NUM_ERRORS as in err_tb.h. It is important to do this as err_tb.h includes this file. However, it then defines NUM_ERRORS again as we saw above. In effect, overwriting the NUM_ERRORS value in this file, errors.h.

This file is used to define the code for each error message. This code is printed out along with the string for the error. About line 215, add the value for the FLASH_ERROR code.

#define FLASH_ERROR 0xf100.

0xF100 was chosen, because it appears that the grouping is determined by the first two hex characters and the last two hex characters are just sequential increments for errors in that category. So codes 0xf5xx through 0xffxx were already in use. So chose 0xf1xx randomly from the available ones of 0xf0xx through 0xf4xx.

These are the only files that need to be changed. The actual work is performed by err_tb.c. When a dink32 function returns to the main dink32 loop it can return one of these error messages. As in return(FLASH_ERROR);. Then the function err_print_error (about line 35) searches this structure, err_list, comparing the error number with the err_list[i].code. When

it finds the code, it prints the code value and the error message. If it can't find the code, then it prints the message, UNKNOWN ERROR.

Appendix C History of MDINK32/DINK32 changes

C.1 Version 12.0 November 30, 1999.

1. Implement a dink transfer table to dynamically assign dink functions such as printf, dinkloop, getchar, in a table so that it is no longer necessary to statically determine the function address and change them in demo or dhrystones or any user program.
2. Configuration (environment variables) are saved in NVRAM for yk/sp, saved in RAM for Excimer and Maximer. New command, env, manipulates these configurations. Also implements multiple command aliases, however, da and ra are still available.
3. New command, tau, display and/or calibrate the Thermal Assist Unit.
4. Faster download and no need to set character delays on the serial line, implemented by turning on the duart FIFO.
5. Turn on both banks of memory in the YellowKnife and Sandpoint, now 32Megabytes is available on dink32 startup.
6. Improved printf format facilities, including floating point.
7. Most commands can now be placed into quiet mode, and verbose mode can be used with the -v command. Default is verbose on both, same as always, with or without ENV. The '-e' mode expands fields and can be made default with env RDMODE=e. Only Excimer and Maximer require the setup, and RDMODE can be 'Q' (quiet), 'E' (expand fields), or anything else. On Excimer and Maximer it can be set up with these commands:
   ```
   env -c, env rdmode=0
   ```
8. The dl command can be placed in silent mode with the "-q".
9. rd or rm can use these aliases for the memory register, northbridge, nb, mpc106, mpc107, or mpc8240.
10. Fixed command termination character, 'x', so it will not restart if unexpected.
11. Fixed problems with double prompts printed on startup with DCACHE.
12. Implement a new makefile, makefile_gcc, and conform the dink code to build with the gcc PowerPC eabi compatible compiler. Build and load works, all memory features are broken. This will be fixed in the next release.
13. Implemented flash programming for PCI-hosted boot ROM on YK/SP platforms. The command 'fl -h' transfers 512k from a specified memory location to the flash.
14. Added share memory between host and agent targets using the Address Translation Unit (ATU).

C.2 Version 11.0.2 June 1, 1999

1. Fixed invalid cacheing on 603. 603 does not reset the cache invalidate bits in hardware, so added the facility in software.
2. Detects MPC107.
3. About command now reports board and processor identification.
4. Improved the help facility.
5. Added makefiles for the PC, makefile_pc in every directory.

C.3 Version 11.0.1 May 1, 1999 Not Released

1. Change the location of Stack pointer load/save. DINK code now occupies through 0x0080000. USER CODE MUST NOT START EARLIER THAN 0x0090000!
2. Fixed vector alignment.
3. Fixed VSCR register implementation issue.
4. Fixed access issue for registers VRSAVE,RSCR,FPSCR,RTCU, RTCL & RPA.
5. Fixed HID1 display for 603e, 604e.
6. Fixed breakpoint/exception problem broken in rev10.7 for 603e.
7. Fixed location of exception vectors after EH1200, they were wrong.
8. Fixed flushhead in except2.s to work correctly.

C.4 Version 11.0 March 29, 1999

1. Add AltiVec support for the MAX processor.
2. Added vector registers to register list.
3. Add assembler disassembler code for altivec mnemonics.
4. fl -dsi has been expanded to display the flash memory range for each sector.

C.5 Version 10.7 February 25, 1999

1. Add 1999 to copyright dates.
2. Add timeout to flash_write_to_memory, so an unfinished write to flash won't last for ever, it will timeout and issue an error message.
3. Add test all flash write for protected sector and if protected issue an error and refuse the write.
4. Disable transpar,tm from excimer.
5. Set DCFA bit from 0 to 1 for MAX chips only

C.6 Version 10.6 January 25, 1999

1. Implement the history.c file and allow the about command to use constants for Version, Revision, and Release.
2. Implement the fl –dsi and fl -se commands.
3. Automatically detect flash between Board Rev 2 and 3.
4. Remove the fw -e command from DINK32, it is only available in MDINK32.

C.7 Version 10.5 November 24, 1998

1. Changed default reset address to be -xfff0 for standalone dink
2. Fix bugs in trace command

C.8 Version 10.4 November 11, 1998

1. Recapture 10.3 LED post routine in MDINK
2. Add BMC_BASE_HIGH for kahlua to reach the high config registers
3. Added memory test feature during POR.
4. Corrected ending address for kahlua X4 configuartion
5. Added basic Kahlua support

C.9 Version 10.3 no date

1. This was never released

C.10 Version 10.2 September 11, 1998

1. This release is the same as Version 10 Revision 1

C.11 Version 10.1 September 10, 1999

1. Enable ICACHE and DCACHE

C.12 Version 9.5 August 5, 1998

1. Implement flash commands, fw -e and basic flash erase and write support.
2. Split dink into two types, mdink - minimal dink and dink.
3. Implement support for excimer.

C.13 Version 9.4 May 22, 1998

1. Implement L2 Backside Code.
2. Turned on DCACHE and ICACHE as default at boot time.
3. Added Yellowknife X4 boot code (Map A & B)

C.14 Prior to Version 9.4 Approximately October 10, 1997

1. Merged CHRP and PREP
2. Added W_ACCESS (Word access) H_ACCESS, and B_ACCESS
3. One version of dink works with all processors, 601, 603, 604, and ARTHUR.

Appendix D S-Record Format Description

D.1 General Format

An S-record is a file that consists of a sequence of specially formatted ASCII character strings. Each line of the S-record file adheres to the same general format (with some variation of the specific fields) and must be 78 bytes or fewer in length. A typical S-record file might look like this:

```
S0100000777269746 56D656D2E73726563AA
S219070000740000007000000003D20DEAD6129BEEF3C60000060E0
S2190700156300003CC0004060C600007D20192E7CE0182E7C07FC
S21907002A4800408200143863040 07C0330004180FFE848000059
S20907003F004800000068
S804070000F4
```

This information is an encoding of data to be loaded into memory by a S-record loader. The address at which the data is loaded is determined by the information in the S-record. The data is verified through the use of a checksum located at the end of each record. Each record in a file should be followed by a linefeed.

The general format of an S-record is as follows:

```
Type          char[2]
Count         char[2]
Address     char[4,6, or 8]
Data          char[0-64]
Checksum  char[2]
```

Note that the fields are composed of characters. Depending on the field, these characters may be interpreted as hexadecimal values or as ASCII characters. Typically, the values in the Type field are interpreted as characters, while the values in all other fields are interpreted as hex digits.

Type: Describes the type of S-record entry. There are S0, S1, S2, S3, S5, S7, S8, and S9 types. This information is used to determine the format of the remainder of the characters in the entry. The specific format for each S-record type is discussed in the next section.

Count: When the two characters comprising this field are interpreted as a hex value, indicates the number of remaining character pairs in the record.

Address: These characters are interpreted as a hex address. They indicate the address where the data is to be loaded into memory. The address may be interpreted as a 2, 3, or 4 bytes address, depending on the type of record. 2-byte addresses require 4 characters, 3-byte addresses require 6 characters, and 4-byte addresses require 8 characters.

Data: This field can have anywhere from 0 to 64 characters, representing 0-32 hexadecimal bytes. These values will be loaded into memory at the address specified in the address field.

Checksum: These 2 characters are interpreted as a hexadecimal byte. This number is determined as follows: Sum the byte values of each pair of hex digits in the count, address, and data fields of the record. Take the one's complement. The least significant byte of the result is used as the checksum.

D.2 Specific Formats

Each of the record types has a slightly different format. These are all derived from the general format specified above and are summarized in the following table.

TypeDescription

S0

Contains header information for the S-record. This data isn't actually loaded into memory. The address field of an S0 record is unused and will contain 0x0000. The data field contains the header information, which is divided into several sub-fields:

```
char[20] module name
char[2] version number
char[2] revision number
char[0-36] text comment
```

Each subfield is composed of ASCII characters. These are paired and interpreted as one byte hex values in the case of the revision number and version number fields. For the module name and text comment fields these values should be interpreted as hexadecimal values of ASCII characters.

S1

The address field is interpreted as a 2-byte address. The data in the record is loaded into memory at the address specified.

S2

The address field is interpreted as a 3-byte address. The data in the record is loaded into memory at the address specified.

S3

The address field is interpreted as a 4-byte address. The data in the record is loaded into memory at the address specified.

S5

The address field is interpreted as a 2-byte value which represents a count of the number of

S1, S2, and S3 records previously transmitted. The data field is unused.

S7

The address field is interpreted as a 4-byte address and contains the execution start address. The data field is unused.

S8

The address field is interpreted as a 3-byte address and contains the execution start address. The data field is unused.

S9

The address field is interpreted as a 2-byte address and contains the execution start address. The data field is unused.

D.3 Examples

Following are some sample S-record entries broken into their parts with a short explanation:

```
Example 1: S01000077726974656D656D2E73726563AA
Separated: S0-10-0000-7772697465 6D656D2E73726563-AA
```

•Type: S0 - this is a header record •Count: 10 - interpreted as 0x10; indicates that 16 character pairs follow •Address: 0000 - interpreted as 0x0000. The address field for S0 is always 0x0000. •Data: Since this is a header record, the information can be interpreted in a number of ways. It doesn't really matter since you usually don't use this field for anything interesting. •Checksum: AA - the checksum

```
Example 2: S21907000074000000700000003D20DEAD6129BEEF3C60000060E0
Separated:
S2-19-070000-74000000700000003D20DEAD6129BEEF3C60000060-E0
```

•Type: S2 - the record consists of memory-loadable data and the address should be interpreted as 3 bytes •Count: 19 - interpreted as 0x19; indicates that 25 character pairs follow •Address: 070000 - data will be loaded at address 0x00070000 •Data: Memory loadable data representing executable code •Checksum: E0 - checksum

```
Example 2: S804070000F4
Separated: S8-04-070000-F4
```

•Type: S8 - this is the record with the execution start address; also indicates we have reached the end of our s-record •Count: 04 - interpreted as 0x04; indicates that 4 character

pairs follow •Address: 070000 - execution will begin at 0x00070000 •Data: None - this field is unused for S8 records. •Checksum: F4 - checksum

D.4 Summary of Formats

The following table summarizes the length (in characters, bytes) of each field for the different S-record types. It is useful as a reference when parsing records manually during debug.

Table 10-1. Summary of Formats in Bytes

Type	Count	Address	Data	Checksum
S0	2	n/a	0-60	2
S1	2	2 byte address	0-64	2
S2	2	3 byte address	0-64	2
S3	2	4 byte address	0-64	2
S5	2	2 byte count	0	2
S7	2	4 byte execution address	0	2
S8	2	3 byte execution address	0	2
S9	2	4 byte execution address	0	2

Appendix E Example Code

E.1 General Information

Four example directories are included in the DINK32 distribution. These directories include all the source files, a makefile, and a README. All these directories contain examples of using the new dynamic dink addresses as described in Appendix G.

E.2 Demo

The demo directory contains source files that can be built to build an application that can then be downloaded into dink at address 0x90000 and run.

E.2.1 Building

The demo can be built with the UNIX or PC command, `make -f makedemo`. The demo.src file can be downloaded with the DINK32 command `dl -k`. It can be executed with the DINK32 command, `go 90000`. Demo will run continuously. It can be stopped by a reset, or by setting the flow control to none before the go 90000.

E.2.2 Function Addresses

All dink function addresses are determined dynamically, see Appendix G for more information.

E.3 Dhrystone

The dhrystone directory contains source files that can be built to build an application that can then be downloaded into dink at address 0x90000 and run. The dhrystone directory has two subdirectories ties, MWnosc and watch. The makefile is contained in the MWnosc directory. This directory contains all the code necessary to build and run a Dhrystone benchmark program. Before starting execution, change the value of hid0 and dbat1l. DINK32 by default starts the downloaded program with caches off and cache inabled in the dbats. Change hid0 to 0000cc00 and dbat1l to 12. Use these commands:
`rm hid0 | 0000cc00, rm dbat1l | 12.`

E.3.1 Building

The demo can be built with the UNIX or PC command, `make`. After making the dhrystone src, download the file, dhry.src with the DINK32 command `dl -k`. Then change the hid0 register to 8000C000 and change the dbat1L to 12.

There are two makefiles:

- makefile - use the UNIX PowerPC cross tools.
- makefile_pc - use the PC PowerPC cross tools.

It can be executed with the DINK32 command, `go 90000`.

E.3.2 Function Addresses

All dink function addresses are determined dynamically, see Appendix G for more information.

E.4 L2test

The directory contains source files that can be built to build an application that can then be downloaded into dink at address 0x90000 and run. This application will test the L2 cache and exercise the performance monitor. Read the l2test.readme for more information.

E.4.1 Building

The demo can be built with the UNIX or PC command, `make`. There are seven targets, composed of a UNIX PowerPC target, a UNIX native target, and a PC target. The l2test.src file can be downloaded with the DINK32 command `dl -k`. It can be executed with the DINK32 command, `go 90000`. There are two makefiles:

- makefile - used for this release of DINK32 R12 and beyond.
- makefile_dink11 - used for previous releases of dDINK32.

E.4.2 Function Addresses

All dink function addresses are determined dynamically, see Appendix G for more information.

E.5 printtest

The directory contains source files that can be built to build an application that can then be downloaded into dink at address 0x90000 and run. This application will test the various printf features.

E.5.1 Building

The demo can be built with the UNIX or PC command, `make`. There are seven targets, composed of a UNIX PowerPC target, a UNIX native target, and a PC target. The l2test.src file can be downloaded with the DINK32 command `dl -k`.

There are two makefiles:

- makefile - use the UNIX PowerPC cross tools.

- makefile_pc - use the PC PowerPC cross tools.

It can be executed with the DINK32 command, `go 90000`.

E.5.2 Function Addresses

All dink function addresses are determined dynamically, see Appendix G for more information.

Appendix F Updating DINK32 from the Web

F.1 General Information

The DINK32 web site is part of the motorola non-confidential web site. The URL is:

http://www.mot.com/SPS/PowerPC/tecsupport/tools/DINK32/index.html

The format in general includes elf and sfiles for DINK32 both debug and non-debug on.

F.1.1 For YellowKnife and Sandpoint:

Using a ROM burner or in line ROM emulator load the dink32 sfile.

See Section 4.1.13, “fupdate fu".

F.1.2 For Excimer and Maximer:

Using the mdink32 facility running on an Excimer and Maximer board, download the new dink32 with the command dl -fl -o ffc00000, then using your terminals ascii download facility, download the dink32 sfile. See Section 4.1.14, “fw fw -e" and Section 4.1.10, “download dl".

MDINK32 is not supplied as elf or sfiles on this site. However, all the code (some code is purposefully removed and the object files are substituted) is available to build mdink32. Loading MDINK32 requires unprotecting sector 15 on the Excimer and Maximer and using some type of emulator to download the code.

Selected DINK32 code is available at this site. Some files are not released in source form, however, the object code for the removed files are supplied so that DINK32 can be built.

All the source, including the removed code, is available from the Motorola confidential site and can be obtained from you Motorola Salesperson.

F.2 Makeing a DINK32 or MDINK32 from the Release

This release does not include several source files. These source files are included here as empty files. None of the dink_dir or mdink_dir directories are included in this distribution. In order to modify any of the source files and remake a dink or mdink, it is necessary to copy the appropriate directory from the "objects" directory to this source directory and name it dink_dir or mdink_dir.

The objects directories are:

- dink_excimer_met/
- dink_yk_met/
- mdink_excimer_met/
- dink_excimer_met_g/
- dink_yk_met_g/
- mdink_excimer_met_g/
- dink_excimer_pc/
- dink_yk_pc/
- mdink_excimer_pc/
- dink_excimer_pc_g/
- dink_yk_pc_g/
- mdink_excimer_pc_g/:
- dink_excimer_gcc/
- dink_yk_gcc/
- mdink_excimer_gcc/

The naming convention is:

- dink - dink
- mdink - mdink
- excimer - excimer or maximer
- met - metaware compiler on unix
- gcc - gnu gcc compiler on unix
- pc - metaware compiler on an NT/PC.

The steps to make a succesful compile are:

1. copy one of the sfile directories to the source directory and call it dink_dir or mdink_dir
2. make tch This will touch all the object files in the dink_dir or mdink_dir directories, so that none of the empty *.c files will replace the associated object file.
3. make your source file changes.
4. make dink or make mdink.

> If you forget the "make tch", then remove the dink_dir or mdink_dir directory, and recopy it.

example:

- unzip the dink32_12_0.zip file, it will unzip to readable.
- unzip the dink32_12_0_objects.file it will unzip to objects.
- copy one of the objects to the unzipped readable file.
  - e.g.
    cp -r objects/dink_yk_met readable
    make tch
    make dink

Appendix G Dynamic functions such as printf

G.1 General Information

Many library functions such as printf are available via the DINK32 debugger. In the past, it has been necessary to ascertain the address of these functions, which change with each compile, from the cross reference listing, and statically set these addresses in the programs that used these features. The demo and dhrystone directories included with the DINK32 distribution contained examples of how to set these static function addresses. With the release of DINK32 V11.1 and V12.0, these addresses are now dynamically ascertained and the user only need call a few functions and set up some #defines. This technique is described in this appendix. Users with access to the entire DINK32 source base can modify or add DINK32 functions.

G.2 Methodology and implementation.

This method is implemented with a static structure that is filled with the current functions address during link time. The table is allocated in the file par_tb.c. Only users with access to this file can change the contents of the table, thereby, determining which DINK32 functions are available. par_tb.c is only available via the motorola sales office, it is not included on the web site. However, all users can use the technique for linking their code with the these DINK32 functions.

The structure is defined in dink.h as dink_exports

```
typedef struct  {
    int version;                                        /*  0 */
    unsigned long *keyboard;                            /*  4 */
    int (*printf)(const char*,...);                     /*  8 */
    unsigned int (*dink_loop)();                        /* 12 */
    int (*is_char_in_duart)();                          /* 16 */
    unsigned int (*menu)();                             /* 20 */
    unsigned int (*par_about)();                        /* 24 */
    unsigned int (*disassemble)(/*long, long*/);        /* 28 */
    char (*get_char)(unsigned long);                    /* 32 */
    char (*write_char)(char);                           /* 36 */
    } dink_exports;
```

and populated in par_tb.c as dink_transfer_table.

```
dink_exports dink_transfer_table = {
        1,
        &KEYBOARD,
        (int (*)(const char*,...))dink_printf,
        dink_loop,
```

```
is_char_in_duart,
menu,
par_about,
disassemble,
get_char,
write_char
};
```

As you can see, at this time, these are the only functions that are supported. Additional or replacement DINK32 functions can be added to the table.

This table is allocated and linked into the DINK32 binaries. The user typically downloads his/her program into the starting location of free memory, at this release, address 0x90000. Unfortunately, the user program has no way of determining where the dink_transfer_table is located. Therefore when DINK32 transfers control to the user program, it sets the address of the dink_transfer_table in general purpose register 21 in go_tr2.s. This register appears to be immune from being used by the compiler prior to the invocation of the user programs start address, usually, main(). Therefore the user must call the supplied function, set_up_transfer_base, or equivalent, which is described below in G.4. After this call the address of the dink_transfer_table is available to the user program.

G.3 Setting up the static locations.

The table below shows all the functions that are currently supported.

Table 1: DINK32 dynamic names

DINK32 name	Common name
Version of table	1
&KEYBOARD	com port for Keyboard support
dink_printf	printf
dink_loop	DINK32 idle function
is_char_in_duart	has DINK32 detected a character
menu	entry point for DINK32 menu function
par_about	entry point for DINK32 about function
disassemble	entry point for DINK32 disassemble function
get_char	get_char - get next character from com port
write_char	put_char - send character to com port

To change or add any new DINK32 functions, one must change the dink_transfer_table.

To use any of these functions in user code, define the user code function name to be the dink function name. For example, to link the user code printf to the DINK32 printf function, `#define printf dink_printf`, to link the user code put_char to DINK32 write_char, `#define put_char writechar`. See the directories demo and dhrystone for examples of setting up these #define statements.

G.4 Using the Dynamic Functions.

Using these functions is implemented via the assembly language file, `dinkusr.s`, and the include file `dinkusr.h`. The user #includes dinkusr.h and links in dinkusr.s during compilation/link time. All of the functions in this table except `set_up_transfer_base`, transfer control to the DINK32 function while leaving the link register, lr, unchanged. This effectively transfers control to the DINK32 function and the DINK32 function on completion returns directly to the caller in the user's code. The functions supplied in dinkusr.s are shown in the table below.

Table 2: dinkusr.s Functions

Function name	Function definition
set_up_transfer_base	Capture the dink_transfer_table address from r21 and store it into a local memory cell for future use. You must call this function before using any of the functions below, and it should be called immediately after entry, such as the first statement in main().
dink_printf	DINK32 entry into printf.
dink_loop	DINK32 idle loop
is_char_in_duart	DINK32 function to determine if a character has been received.
menu	DINK32 display menu function.
par_about	DINK32 display about function.
disassemble	DINK32 disassemble instruction
get_KEYBOARD	Return address of keyboard com port
get_char	DINK32 get next character from the duart buffer, essentially the keyboard for the user. This function requires the KEYBOARD value, obtained from get_KEYBOARD, as an argument. See G.6 example program _getcannon for an example of the correct way to obtain this value.

Table 2: dinkusr.s Functions

Function name	Function definition
write_char	DINK32 put character to the output buffer.

The simple steps for using these dynamic addresses are:

1. Use DINK32 V11.1 or later.
2. Use #define for local functions that you wish to connect to the DINK32 functions example: #define printf dink_printf
3. The first executable statement in your C code must be: set_up_transfer_base();
4. Now whenever your program calls one of these functions, such as printf, it will transfer control to the equivalent DINK32 function.

G.5 Error Conditions.

The only error condition is a trapword exception, which indicates that the dink_transfer_table address is zero. This is caused by one of the following conditions:

1. The user has not called `set_up_transfer_base()`
2. R12 is getting trashed before set_up_transfer_base() is called.
3. The DINK32 version does not support dynamic functions. DINK32 V11.0.2 was the last version that DID NOT support this feature. Ensure that you are using DINK32 V12.0 or greater.

G.6 Alternative method for Metaware only.

While printf is fairly straightforward, scanf is more complex. In the drystone directory, a local copy of scanf is supplied in the file, `support.c`. Scanf and printf can also be emulated in a simpler program when using the metaware compiler. Two metaware functions are supplied to the user to give control to characters that are scanned into and out of the program buffers. Refer to the metaware documentation for more information than is given here.

When the user compiles and links with the -Hsds flag, two functions, `int _putcanon(int a)`, and `int _getcanon()` are called whenever the user gets or receives a character. Thus, the user can write the simple functions shown below, and scanf and printf will use the DINK32 functions for printf and scanf. In this case, it is not necessary to use #define to change the name of the printf or scanf functions or write your own printf or scanf function. It is still necessary to call `set_up_transfer_base()` as the first statement in your program.

```
/*****************************************************************
**
 *  Functions to capture characters from printf and scanf using
 *  the -Hsds hooks in the metaware compiler
 *  mlo  7/22/99
*****************************************************************
*/

#include "dinkusr.h"

int _putcanon(int a)
{
/*  grab the character sent by printf in -Hsds and
 *  use it in dink putchar
*/
char c;
        c=a;
        write_char(c);
        return 1;
}

int _getcanon()
{
/*  extract the character received by scanf in -Hsds and use
 *  it in dink putchar
 */
unsigned long key;
        key = get_KEYBOARD();
        return (get_char(key ));
}
```

Appendix H MPC8240 (Kahlua) Drivers

H.1 Drivers directory.

There are four drivers for the MPC8240 integrated peripheral devices.

- DMA - memory controller
- I2C - serial controller
- I2O - doorbell controller
- EPIC - interrupt controller

Sample code for each of these drivers are in the directory, drivers, under dink32. Under the drivers directory are four directories, one for each controller see Figure 3-1. The following sections describe the driver and the sample code. Each driver is discussed in one of the following four appendices.

- Appendix I, “MPC8240 DMA Memory Controller."
- Appendix J, “MPC8240 I2C Driver Library."
- Appendix K, “MPC8240 I2O Doorbell Driver"
- Appendix L, “MPC8240 EPIC Interrupt Driver"

Appendix I MPC8240 DMA Memory Controller.

This section provides information about the generic Application Program Interface (API) to the DMA Driver Library as well as information about the implementation of the Kahlua-specific DMA Driver Library Internals (DLI).

I.1 Background

The intended audience for this document is assumed to be familiar with the DMA protocol. It is a companion document to the Kahlua specification and other documentation which collectively give details of the DMA protocol and the Kahlua implementation. This document provides information about the software written to access the Kahlua DMA interface. This software is intended to assist in the development of higher level applications software that uses the DMA interface.

> Note: The DMA driver software is currently under development. The only mode that is functional is a direct transfer (chaining is not yet implemented). Only transfers to and from local memory has been tested. Controlling a remote agent processor is not yet implemented. Of the various DMA transfer control options implemented in Kahlua, the only ones currently available in this release of the DMA library are source address, destination address, length, channel, interrupt steering and snoop enable.

I.2 Overview

This document consists of these parts:

- An Application Program Interface (API) which provides a very simple, "generic", application level programmatic interface to the DMA driver library that hides all details of the Kahlua-specific implementation of the interface (i.e., control register, status register, embedded utilities memory block, etc.). Features provided by the Kahlua implementation that may or may not be common with other implementations (i.e., not "generic" DMA operations) are made available to the application; however, the interface is controlled by passing parameters defined in the API rather than the application having to have any knowledge of the Kahlua implementation (i.e., registers, embedded utilities memory block, etc.) The API will be expanded to include chaining mode and additional DMA transfer control features in future releases.
- DMA API functions showing the following:

  - — how the function is called (i.e., function prototype) parameter definition possible return values brief description of what the function does
  - — an explanation of how the functions are used by an application program (DINK32 usage employed as examples)
- A DMA Driver Library Internals (DLI) which provides information about the lower level software that is accessing the Kahlua-specific implementation of the DMA interface.
- DMA DLI functions showing the following:
  - — how the function is called (i.e., function prototype)
  - — parameter definition possible
  - — return values
  - — brief description of what the function does

I.3 DMA Application Program Interface (API)

API functions description

The DMA API function prototypes, defined return values, and enumerated input parameter values are declared in drivers/dma/dma_export.h.

The functions are defined in the source file drivers/dma/dma1.c.

```
DMA_Status
DMA_Initialize(int(*app_print_function)(char*,...));
```

- `app_print_function` is the address of the optional application's print function, otherwise NULL if not available
- Return: DMA_Status return value is either DMA_SUCCESS or DMA_ERROR.

Description:

Configure the DMA driver prior to use, as follows:

The optional print function, if supplied by the application, must be similar to the C standard library printf library function: accepts a format string and a variable number (zero or more) of additional arguments. This optional function may be used by the library functions to report error and status condition information. If no print function is supplied by the application, the application must provide a NULL value for this parameter, in which case the library will not attempt to access a print function.

> NOTE: Each DMA transfer will be configured individually by the function call that initiates the transfer. If it is desirable to establish a default configuration, these could be added as

parameters. Alternately, the first (or most recent) transfer configuration values could also be used to establish defaults.

NOTE: This function call triggers the DMA library to read the eumbbar so that it is available to the driver, so it is a requirement that the application first call DMA_Initialize before starting any DMA transfers. This could be eliminated if the other functions read the eumbbar if it has not already been done.

```
DMA_Status DMA_direct_transfer( DMA_INTERRUPT_STEER int_steer,
DMA_TRANSFER_TYPE type,
unsigned int source,
unsigned int dest,
unsigned int len,
DMA_CHANNEL channel,
DMA_SNOOP_MODE snoop);
```

- int_steer controls interrupt steering, use defined constants as follows:
  DMA_INT_STEER_LOCAL to steer to local processor
  DMA_INT_STEER_PCI to steer to PCI bus through INTA_
- type is the type of transfer, use defined constants as follows:
  DMA_M2M local memory to local memory (note, this is currently the only one tested)
  DMA_M2P local memory to PCI
  DMA_P2M PCI to local memory
  DMA_P2P PCI to PCI
- source is the source address of the data to transfer
- dest is the destination address, the target of the transfer
- len is the length in bytes of the data
- channel is the DMA channel to use for the transfer, use defined constants as follows:
  DMA_CHN_0 Kahlua has two channels, zero and one
  DMA_CHN_1
- snoop controls processor snooping of the DMA channel buffer, use defined constants a follows:
  DMA_SNOOP_DISABLE
  DMA_SNOOP_ENABLE
- Return: DMA_Status return value is either DMA_SUCCESS or DMA_ERROR.

Description:

Initiate the DMA transfer.

This function does not implement any validation of the transfer. It does check the status of the DMA channel to determine if it is OK to initiate a transfer, but the application must

handle verification and error conditions via the interrupt mechanisms.

I.3.1 API Example Usage

The ROM monitor program DINK32 currently uses the DMA API to initiate a direct data transfer in local memory only. The DINK32 program runs interactively to allow the user to transfer a block of data in local memory. DINK32 obtains information from the user as follows: interrupt steering, transfer type, source address of the data, destination (target) address, length of the data to transfer, DMA channel, and snoop control.

Note that the initialization call to configure the DMA interface is made once: the first time the user requests a DMA transfer operation. Each transmit or receive operation is initiated by a single call to a DMA API function. The DINK32 program is an interactive application, so it gives the DMA library access to its own print output function. DINK32 does not currently implement any handling of interrupts for error handling or completion of transfer verification.

These are the steps DINK32 takes to perform a DMA transfer:

1. Call DMA_Initialize (if first transfer) to identify the optional print function.
2. Call DMA_direct_transfer to transmit the buffer of data.

The following code samples have been excerpted from the DINK32 application to illustrate the use of the DMA API:

```
#define PRINT dink_printf
int dink_printf( unsigned char *fmt, ... )
{
/* body of application print output function, */
}
/* In the function par_devtest, for testing the DMA device interface
*/
{
/* initialize the DMA handler, if needed */
if ( DMAInited == 0 )
{
DMA_Status status;
if ((status = DMA_Initialize( PRINT ) ) != DMA_SUCCESS )
{
PRINT( "devtest DMA: error in initiation\n" );
return ERROR;
} else {
DMAInited = 1;
}
}
return test_dma( en_int ); /* en_int is the steering control option
*/
}
/*****************************************************
* function: test_dma
```

```
*
* description: run dma test
*
* note:
* test local dma channel
*****************************************************/
static STATUS test_dma( int en_int )
{
int len = 0, chn = 0;
long src = 0, dest = 0;
int mode = 0;
DMA_SNOOP_MODE snoop = DMA_SNOOP_DISABLE;
DMA_CHANNEL channel;
DMA_INTERRUPT_STEER steer;
/* The default for en_int is 0, the default for steering DMA
interrupts is
* to route them to the PCI bus through INTA_. At least, that is the
DINK
* default behavior. If the DINK user puts a '+' on the command line,
that
* means route to local processor because '+' means "enable
interrupts".
*/
steer = ( en_int == 0 ? DMA_INT_STEER_PCI : DMA_INT_STEER_LOCAL );

/* read source and destination addresses, length, type, snoop and
channel */
...

/* validate and translate to API defined parameter values */
...

/* call the DMA library to initiate the transfer */
if ( DMA_direct_transfer ( steer, type, (unsigned int)src,
(unsigned int)dest, (unsigned int)len, channel, snoop) !=
DMA_SUCCESS )
{
PRINT( "dev DMA: error in DMA transfer test\n" );
return ERROR;
}
return SUCCESS;
}
```

I.4 DMA Driver Library Internals (DLI)

This information is provided to assist in further development of the DMA library.

All of these functions are defined as static in the source file drivers/dma/dma1.c.

I.4.1 Common Data Structures and Values

The following data structures, tables and status values are defined (see drivers/dma/dma.h

unless otherwise noted) for the Kahlua DMA driver library functions.

```
These are the register offsets in a table of the Embedded Utilities
Memory Block addresses for the DMA registers.
#define NUM_DMA_REG 7
#define DMA_MR_REG 0
#define DMA_SR_REG 1
#define DMA_CDAR_REG 2
#define DMA_SAR_REG 3
#define DMA_DAR_REG 4
#define DMA_BCR_REG 5
#define DMA_NDAR_REG 6
```

The table that contains the addresses of the local and remote registers for both DMA channels (defined in drivers/dma/dma1.c):

```
unsigned int dma_reg_tb[][14] = {
/* local DMA registers */
{
/* DMA_0_MR */ 0x00001100,
/* DMA_0_SR */ 0x00001104,
/* DMA_0_CDAR */ 0x00001108,
/* DMA_0_SAR */ 0x00001110,
/* DMA_0_DAR */ 0x00001118,
/* DMA_0_BCR */ 0x00001120,
/* DMA_0_NDAR */ 0x00001124,
/* DMA_1_MR */ 0x00001200,
/* DMA_1_SR */ 0x00001204,
/* DMA_1_CDAR */ 0x00001208,
/* DMA_1_SAR */ 0x00001210,
/* DMA_1_DAR */ 0x00001218,
/* DMA_1_BCR */ 0x00001220,
/* DMA_1_NDAR */ 0x00001224,
},
/* remote DMA registers */
{
/* DMA_0_MR */ 0x00000100,
/* DMA_0_SR */ 0x00000104,
/* DMA_0_CDAR */ 0x00000108,
/* DMA_0_SAR */ 0x00000110,
/* DMA_0_DAR */ 0x00000118,
/* DMA_0_BCR */ 0x00000120,
/* DMA_0_NDAR */ 0x00000124,
/* DMA_1_MR */ 0x00000200,
/* DMA_1_SR */ 0x00000204,
/* DMA_1_CDAR */ 0x00000208,
/* DMA_1_SAR */ 0x00000210,
/* DMA_1_DAR */ 0x00000218,
/* DMA_1_BCR */ 0x00000220,
/* DMA_1_NDAR */ 0x00000224,
},
};
```

```
These values are the function status return values:
```

```
typedef enum _dmastatus
{
DMASUCCESS = 0x1000,
DMALMERROR,
DMAPERROR,
DMACHNBUSY,
DMAEOSINT,
DMAEOCAINT,
DMAINVALID,
DMANOEVENT,
} DMAStatus;
```

These structures reflect the bit assignments of the DMA registers.

```
typedef enum dma_mr_bit
{
IRQS = 0x00080000,
PDE = 0x00040000,
DAHTS = 0x00030000,
SAHTS = 0x0000c000,
DAHE = 0x00002000,
SAHE = 0x00001000,
PRC = 0x00000c00,
EIE = 0x00000080,
EOTIE = 0x00000040,
DL = 0x00000008,
CTM = 0x00000004,
CC = 0x00000002,
CS = 0x00000001,
} DMA_MR_BIT;
typedef enum dma_sr_bit
{
LME = 0x00000080,
PE = 0x00000010,
CB = 0x00000004,
EOSI = 0x00000002,
EOCAI = 0x00000001,
} DMA_SR_BIT;
/* structure for DMA Mode Register */
typedef struct _dma_mr
{
unsigned int reserved0 : 12;
unsigned int irqs : 1;
unsigned int pde : 1;
unsigned int dahts : 2;
unsigned int sahts : 2;
unsigned int dahe : 1;
unsigned int sahe : 1;
unsigned int prc : 2;
unsigned int reserved1 : 1;
unsigned int eie : 1;
unsigned int eotie : 1;
unsigned int reserved2 : 3;
unsigned int dl : 1;
unsigned int ctm : 1;
```

```
/* if chaining mode is enabled, any time, user can modify the
* descriptor and does not need to halt the current DMA transaction.
* Set CC bit, enable DMA to process the modified descriptors
* Hardware will clear this bit each time, DMA starts.
*/
unsigned int cc : 1;
/* cs bit has dua role, halt the current DMA transaction and
* (re)start DMA transaction. In chaining mode, if the descriptor
* needs modification, cs bit shall be used not the cc bit.
* Hardware will not set/clear this bit each time DMA transaction
* stops or starts. Software shall do it.
*
* cs bit shall not be used to halt chaining DMA transaction for
* modifying the descriptor. That is the role of CC bit.
*/
unsigned int cs : 1;
} DMA_MR;
/* structure for DMA Status register */
typedef struct _dma_sr
{
unsigned int reserved0 : 24;
unsigned int lme : 1;
unsigned int reserved1 : 2;
unsigned int pe : 1;
unsigned int reserved2 : 1;
unsigned int cb : 1;
unsigned int eosi : 1;
unsigned int eocai : 1;
} DMA_SR;
/* structure for DMA current descriptor address register */
typedef struct _dma_cdar
{
unsigned int cda : 27;
unsigned int snen : 1;
unsigned int eosie : 1;
unsigned int ctt : 2;
unsigned int eotd : 1;
} DMA_CDAR;
/* structure for DMA byte count register */
typedef struct _dma_bcr
{
unsigned int reserved : 6;
unsigned int bcr : 26;
} DMA_BCR;
/* structure for DMA Next Descriptor Address register */
typedef struct _dma_ndar
{
unsigned int nda : 27;
unsigned int ndsnen : 1;
unsigned int ndeosie: 1;
unsigned int ndctt : 2;
unsigned int eotd : 1;
} DMA_NDAR;
/* structure for DMA current transaction info */
```

```
typedef struct _dma_curr
{
unsigned int src_addr;
unsigned int dest_addr;
unsigned int byte_cnt;
} DMA_CURR;
```

I.5 Kahlua DMA Driver Library Internals: function descriptions

The API function DMA_direct_transfer (described above) accepts predefined parameter values to initialize a DMA transfer. These parameters are used by the DMA driver library functions to set up the Kahlua DMA status and mode registers so that the application does not have to interface to the Kahlua processor on such a low level. A description of the processing performed in the DMA_direct_transfer function and descriptions of the lower level DMA driver library functions follow.

This is a description of the DMA_direct_transfer processing, which initiates a simple direct transfer:

1. Read the mode register (MR) by calling DMA_Get_Mode
2. Set the values in the mode register as follows:
   IRSQ is set from the int_steer parameter
   if steering DMA interrupts to PCI, set EIE and EOTIE
   the other mode controls are currently hard coded:
   PDE cleared
   DAHS = 3; however this is ignored because DAHE is cleared
   SAHS = 3; however this is ignored because SAHE is cleared
   PRC is cleared
   DL is cleared
   CTM is set (direct mode)
   CC is cleared
3. Validate the length of transfer value, report error and return if too large
4. Read the current descriptor address register by calling DMA_Poke_Desp
5. Set the values in the CDAR as follows:
   SNEN is set from the snoop parameter
   CTT is set from the type parameter
6. Write the CDAR by calling DMA_Bld_Desp, which checks the channel status to ensure it is free
7. Write the source and destination address registers (SAR and DAR) and the byte count register (BCR) by calling DMA_Bld_Curr, which maps them according to channel and host and ensure the channel is free
8. Write the mode register by calling DMA_Set_Mode

9. Begin the DMA transfer by calling DMA_Start, which ensures the channel is free and then clears and sets the mode register channel start (CS) bit
10. The proceeding steps 6 through 9 are done in a sequence so that each call must return a successful status prior to executing the following step. The status is checked and error conditions are reported at this point if all did not execute successfully.
11. If this point is reached, the DMA transfer was initiated successfully, return success status

These are descriptions of the DMA library functions reference above in the DMA_direct_transfer processing steps.

```
DMAStatus DMA_Get_Mode( LOCATION host,
unsigned eumbbar,
unsigned int channel,
DMA_MR *mode);
```

- host is LOCAL or REMOTE, only LOCAL is currently tested
- eumbbar is EUMBBAR for LOCAL or PCSRBAR for REMOTE
- channel is DMA_CHN_0 or DMA_CHN_1
- mode is a pointer to the structure (DMA_MR) to receive the mode register contents
- Return value is DMASUCCESS or DMAINVALID

Description:

Read the DMA mode register.

```
DMAStatus DMA_Poke_Desp( LOCATION host,
unsigned eumbbar,
unsigned int channel,
DMA_CDAR *desp);
```

- host is LOCAL or REMOTE, only LOCAL is currently tested
- eumbbar is EUMBBAR for LOCAL or PCSRBAR for REMOTE
- channel is DMA_CHN_0 or DMA_CHN_1
- desp is a pointer to the structure (DMA_CDAR) to receive the CDAR contents
- Return value is DMASUCCESS or DMAINVALID

Description:

Read the current descriptor address register (CDAR) specified by host and channel.

```
DMAStatus DMA_Bld_Desp( LOCATION host,
unsigned eumbbar,
unsigned int channel,
DMA_CDAR *mode);
```

- host is LOCAL or REMOTE, only LOCAL is currently tested

- eumbbar is EUMBBAR for LOCAL or PCSRBAR for REMOTE
- channel is DMA_CHN_0 or DMA_CHN_1
- desp is a pointer to the structure (DMA_CDAR) holding the CDAR control bits
- Return value is DMASUCCESS, DMACHNBUSY or DMAINVALID

Description:

Set the current descriptor address register (CDAR) specified by host and channel to the given values.

```
DMAStatus DMA_Bld_Curr( LOCATION host,
unsigned eumbbar,
unsigned int channel,
DMA_CURR *desp);
```

- host is LOCAL or REMOTE, only LOCAL is currently tested
- eumbbar is EUMBBAR for LOCAL or PCSRBAR for REMOTE
- channel is DMA_CHN_0 or DMA_CHN_1
- desp is a pointer to the structure (DMA_CURR) holding the source, destination and byte count
- Return value is DMASUCCESS, DMACHNBUSY or DMAINVALID

Description:

Set the source address register (SAR), destination address register (DAR) and byte count register (BCR) specified by host and channel to the given values.

```
DMAStatus DMA_Start( LOCATION host,
unsigned eumbbar,
unsigned int channel);
```

- host is LOCAL or REMOTE, only LOCAL is currently tested
- eumbbar is EUMBBAR for LOCAL or PCSRBAR for REMOTE
- channel is DMA_CHN_0 or DMA_CHN_1
- Return value is DMASUCCESS, DMACHNBUSY or DMAINVALID

Description:

Start the DMA transfer on the specified host and channel. Ensure the channel is free, then clear and set the CS bit in the mode register. That 0 to 1 transition triggers the DMA transfer.

Appendix J MPC8240 I2C Driver Library.

This section provides information about the generic Application Program Interface (API) to the I2C Driver Library as well as information about the implementation of the Kahlua-specific I2C Driver Library Internals (DLI).

J.1 Background

The intended audience for this document is assumed to be familiar with the I2C bus protocol. It is a companion document to the Kahlua specification and other documentation which collectively give details of the I2C protocol and the Kahlua implementation. This document provides information about the software written to access the Kahlua I2C interface. This software is intended to assist in the development of higher level applications software that uses the I2C interface.

> Note: The I2C driver software is currently under development. The only modes that are functional are the master-transmit and master-receive in polling mode.

J.2 Overview

This document consists of these parts:

- An Application Program Interface (API) which provides a very simple, generic, application level programmatic interface to the I2C driver library that hides all details of the Kahlua-specific implementation of the I2C interface (i.e., control register, status register, embedded utilities memory block, etc.).
- I2C API functions showing the following:
  - — how the function is called (i.e., function prototype)
  - — parameter definition
  - — possible return values
  - — brief description of what the function does
  - — an explanation of how the functions are used by an application program (DINK32 usage employed as examples)
- An I2C Driver Library Internals (DLI) which provides information about the lower level software that is accessing the Kahlua-specific implementation of the I2C interface.
  - — I2C DLI functions showing the following:
  - — how the function is called (i.e., function prototype)
  - — parameter definition

— possible return values

— brief description of what the function does

J.3 I2C Application Program Interface (API)

J.3.1 API functions description

The I2C API function prototypes, defined return values, and enumerated input parameter values are declared in drivers/i2c/i2c_export.h.

The functions are defined in the source file drivers/i2c/i2c1.c.

```
I2C_Status I2C_Initialize( unsigned char addr,
I2C_INTERRUPT_MODE en_int,
int (*app_print_function)(char *,...));
```

- addr is the Kahlua chip's I2C slave device address
- en_int controls the I2C interrupt enable status: I2C_INT_ENABLE = enable, I2C_INT_DISABLE = disable
- app_print_function is the address of the optional application's print function, otherwise NULL if not available
- Return: I2C_Status return value is either I2C_SUCCESS or I2C_ERROR.

Description:

Configure the I2C library prior to use, as follows:

The interrupt enable should be set to I2C_INT_DISABLE, the I2C library currently only supports polling mode.

The slave address can be set to the I2C listening address of the device running the application program, but the DLI does not yet support the application's device responding as an I2C slave to another I2C master device.

The optional print function, if supplied by the application, must be similar to the C standard library printf library function: accepts a format string and a variable number (zero or more) of additional arguments. This optional function may be used by the I2C library functions to report error and status condition information. If no print function is supplied by the application, the call to I2C_Initialize must provide a NULL value for this parameter, in which case the I2C library will not attempt to access a print function.

```
I2C_Status I2C_do_transaction( I2C_INTERRUPT_MODE en_int,
I2C_TRANSACTION_MODE act,
unsigned char i2c_addr,
unsigned char data_addr,
int len,
char *buffer,
I2C_STOP_MODE stop,
```

```
int retry,
I2C_RESTART_MODE rsta);
```

- en_int controls the I2C interrupt enable status (currently use I2C_INT_DISABLE only, polling mode)
- act is the type of transaction: I2C_MASTER_RCV or I2C_MASTER_XMIT
- i2c_addr is the I2C address of the slave device
- data_addr is the address of the data on the slave device
- len is the length in bytes of the data
- buffer is a pointer to the buffer that contains the data (xmit mode) or receives the data (rcv mode)
- stop controls sending an I2C STOP signal after completion (curently use I2C_STOP only)
- retry is the timeout retry value (currently ignored)
  rsta controls I2C restart (currently use I2C_NO_RESTART only)
- Return: I2C_Status return value is either I2C_SUCCESS or I2C_ERROR.

Description:

Act as the I2C master to transmit (or receive) a buffer of data to (or from) an I2C slave device.

This function currently only implements a simple master-transmit or a master-receive transaction. It does not yet support the application retaining I2C bus ownership between transactions, operating in interrupt mode, or acting as an I2C slave device.

J.3.2 API Example Usage

The ROM monitor program DINK32 uses the I2C API in both currently implemented modes: master-transmit and master-receive. The DINK32 program runs interactively to allow the user to transmit or receive a buffer of data from an I2C device at address 0x50 on the Kahlua PMC card. DINK32 obtains information from the user as follows: read/write mode, I2C device address for the data (this is the address of the data on the I2C device, not the I2C bus address of the device itself, which is hard-coded in DINK32), the raw data (if in write mode), and the length of the data to transfer to or from the device. Note that the initialization call to configure the I2C interface is actually made only once, the first time the user requests an I2C transmit or receive operation. Each transmit or receive operation is performed by a single call to an I2C API function. The DINK32 program is an interactive application, so it gives the I2C library access to its own print output function.

These are the steps DINK32 takes to perform a master-transmit transaction:

1. Call I2C_Initialize (if needed) to set the Kahlua I2C address, polling mode, and identify the optional print function.

2. Call I2C_do_transaction to transmit the buffer of data.

These are the steps DINK32 takes to perform a master-receive transaction in polling mode:

1. Call I2C_Initialize (if needed) to set the Kahlua I2C address, polling mode, and identify the optional print function.
2. Call I2C_do_transaction to receive the buffer of data.

The following code samples have been excerpted from the DINK32 application to illustrate the use of the I2C API:

```
#define PRINT dink_printf
int dink_printf( unsigned char *fmt, ... )
{
/* body of application print output function, see Appendix ??? */
}
/* In the function par_devtest, for testing the I2C device interface
*/
{
/* initialize the I2C handler to I2C address 48, if needed */
if ( I2CInited == 0 )
{
I2C_Status status;
if ((status = I2C_Initialize( 48, en_int, PRINT ) ) != I2C_SUCCESS )
{
PRINT( "devtest I2C: error in initiation\n" );
return ERROR;
} else {
I2CInited = 1;
}
}
return test_i2c( action, en_int );
}
static unsigned char rcv_buffer[BUFFER_LENGTH] = { 0 };
static unsigned char xmit_buffer[BUFFER_LENGTH] = { 0 };
/*******************************************************
* function: test_i2c
*
* description: run i2c test by polling the device
*
* note:
* Test i2c device on PMC card, 0x50 serial EPROM.
* The device test data is currently only printable characters.
*
* This function gets some data from the command line, validates it,
* and calls the I2C library function to perform the task.
*******************************************************/
static STATUS test_i2c( int act, int en_int )
{
int retry = 800, len = 0, rsta = 0, addr = 0;
unsigned char eprom_addr = 0x50;
/* read transaction address */
... addr ...
/* read # of bytes to transfer */
```

```
... len ...

/* validate the data address, length, etc. */
...
/* If transmitting, get the raw data into the transmit buffer */
... xmit_buffer[] ...
/* read built-in I2C device on Kahlua PMC card */
if ( act == DISPLAY_TAG )
{
if ( I2C_do_transaction ( en_int, I2C_MASTER_RCV, eprom_addr, addr,
len, rcv_buffer, I2C_STOP, retry, I2C_NO_RESTART ) != I2C_SUCCESS )
{
PRINT( "dev I2C: error in master receive test\n" );
return ERROR;
} else {
rcv_buffer[len] = 0; /* ensure NULL terminated string */
PRINT( "%s",rcv_buffer); /* expecting only printable data */
PRINT( "\n" );
}
}
/* write to built-in I2C device on Kahlua PMC card */
if ( act == MODIFY_TAG )
{
if ( I2C_do_transaction ( en_int, I2C_MASTER_XMIT, eprom_addr, addr,
len, xmit_buffer, I2C_STOP, retry, I2C_NO_RESTART ) != I2C_SUCCESS )
{
PRINT( "dev I2C: error in master transmit test\n" );
return ERROR;
}
}
return SUCCESS;
}
```

J.4 I2C Driver Library Internals (DLI)

This information is provided to assist in further development of the I2C library to enable the application to operate as an I2C slave device, interrupt enabled mode, bus retention between consecutive transactions, correct handling of device time out, no slave device response, no acknowledgment, I2C bus arbitration loss, etc.

All of these functions are defined as static in the source file drivers/i2c/i2c1.c.

J.4.1 Common Data Structures and Values

These data structures and status values are defined (see drivers/i2c/i2c.h) for the Kahlua I2C driver library functions:

These are the offsets in the Embedded Utilities Memory Block for the I2C registers.

```
#define I2CADR 0x00003000
#define I2CFDR 0x00003004
#define I2CCR 0x00003008
```

```
#define I2CSR 0x0000300C
#define I2CDR 0x00003010
typedef enum _i2cstatus
{
I2CSUCCESS = 0x3000,
I2CADDRESS,
I2CERROR,
I2CBUFFFULL,
I2CBUFFEMPTY,
I2CXMITERROR,
I2CRCVERROR,
I2CBUSBUSY,
I2CALOSS,
I2CNOEVENT,
} I2CStatus;
```

These structures reflect the bit assignments of the I2C registers.

```
typedef struct _i2c_ctrl
{
unsigned int reserved0 : 24;
unsigned int men : 1;
unsigned int mien : 1;
unsigned int msta : 1;
unsigned int mtx : 1;
unsigned int txak : 1;
unsigned int rsta : 1;
unsigned int reserved1 : 2;
} I2C_CTRL;
typedef struct _i2c_stat
{
unsigned int rsrv0 : 24;
unsigned int mcf : 1;
unsigned int maas : 1;
unsigned int mbb : 1;
unsigned int mal : 1;
unsigned int rsrv1 : 1;
unsigned int srw : 1;
unsigned int mif : 1;
unsigned int rxak : 1;
} I2C_STAT;
Values to indicate receive or transmit mode.
typedef enum _i2c_mode
{
RCV = 0,
XMIT = 1,
} I2C_MODE;
```

J.5 Kahlua I2C Driver Library Internals: function descriptions

```
I2CStatus I2C_Init( unsigned int eumbbar,
unsigned char fdr,
unsigned char addr,
```

```
unsigned int en_int);
```

- eumbbar is the address of the Embedded Utilities Memory Block
- fdr is the frequency divider value used to set the I2C clock rate
- addr is the Kahlua chip's I2C slave device address
- en_int controls the I2C interrupt enable status: 1 = enable, 0 = disable
- Return: I2CStatus return value is always I2CSUCCESS.

Description:

Set the frequency divider (I2CFDR:FDR), listening address (I2CADR:[7:1]), and interrupt enable mode (I2CCR:MIEN).

```
I2C_CTRL I2C_Get_Ctrl( unsigned int eumbbar );
```

- eumbbar is the address of the Embedded Utilities Memory Block
- Return: I2C_CTRL is the contents of the I2C control register (I2CCR)

Description:

Read the I2C control register.

```
void I2C_Set_Ctrl( unsigned int eumbbar, I2C_CTRL ctrl);
```

- eumbbar is the address of the Embedded Utilities Memory Block
- ctrl is the contents of the I2C control register (I2CCR)
- Return: none

Description:

Set the I2C control register.

```
I2CStatus I2C_put( unsigned int eumbbar,
unsigned char rcv_addr,
unsigned char *buffer_ptr,
unsigned int length,
unsigned int stop_flag,
unsigned int is_cnt );
```

- eumbbar is the address of the Embedded Utilities Memory Block
- rcv_addr is the receiver's I2C device address
- buffer_ptr is pointer to the data buffer to transmit
- length is the number of bytes in the buffer
- stop_flag: 1 - signal STOP when buffer is empty
- 0 - don't signal STOP when buffer is empty
- is_cnt: 1 - this is a restart, don't check MBB
- 0 - this is a not restart, check MBB
- Returns: Any defined status indicator

Description:

Set up to send a buffer of data to the intended rcv_addr. If stop_flag is set, after the whole buffer is sent, generate a STOP signal provided that the receiver doesn't signal the STOP in the middle. Caller is the master performing transmitting. If no STOP signal is generated at the end of current transaction, the master can generate a START signal to another slave address.

> The function does not actually perform the data buffer transmit, it just sets up the DLI global variables to control the transaction and calls I2C_Start to send the slave address out on the I2C bus in transmit mode. The application must check the return status to find out if the bus was obtained, then enter a loop of calling I2C_Timer_Event to poll the I2C handler to actually perform the transaction one byte at a time, while checking the return status to determine if there were any errors and if the transaction has completed.

```
I2CStatus I2C_get( unsigned int eumbbar,
unsigned char sender_addr,
unsigned char *buffer_ptr,
unsigned int length,
unsigned int stop_flag,
unsigned int is_cnt );
```

- eumbbar is the address of the Embedded Utilities Memory Block
- sender_addr is the sender's I2C device address
- buffer_ptr is pointer to the data buffer to transmit
- length is the number of bytes in the buffer
- stop_flag: 1 - signal STOP when buffer is empty
- 0 - don't signal STOP when buffer is empty
- is_cnt: 1 - this is a restart, don't check MBB
- 0 - this is a not restart, check MBB
- Returns: Any defined status indicator

Description:

Set up to receive a buffer of data from the desired sender_addr. If stop_flag is set, when the buffer is full and the sender does not signal STOP, generate a STOP signal. Caller is the master performing receiving. If no STOP signal is generated, the master can generate a START signal to another slave address.

> The function does not actually perform the data buffer receive,

it just sets up the DLI global variables to control the transaction and calls I2C_Start to send the slave address out on the I2C bus in receive mode. The application must check the return status to find out if the bus was obtained, then enter a loop of calling I2C_Timer_Event to poll the I2C handler to actually perform the transaction one byte at a time, while checking the return status to determine if there were any errors and if the transaction has completed.

```
I2CStatus I2C_Timer_Event( unsigned int eumbbar, I2CStatus
(*handler)( unsigned int ) );
```

- eumbbar is the address of the Embedded Utilities Memory Block
- handler is a pointer to the function to call to handle any existing status event,
- Returns: I2CNOEVENT if there is no completed event, the I2CSR MIF bit is not set results from call to the handler function if there was a pending event completed

Description:

In polling mode, I2C_Timer_Event can be called to check the I2C status and call the given (or the default: I2C_ISR) handler function if the I2CSR MIF bit is set.

```
I2CStatus I2C_Start( unsigned int eumbbar,
unsigned char slave_addr,
I2C_MODE mode,
unsigned int is_cnt );
```

- eumbbar is the address of the Embedded Utilities Memory Block
- slave_addr is the I2C address of the receiver
- mode: XMIT(1) - put (write)
- RCV(0) - get (read)
- is_cnt: 1 - this is a restart, don't check MBB
- 0 - this is a not restart, check MBB
- Returns: Any defined status indicator

Description:

Generate a START signal in the desired mode. Caller is the master. The slave_addr is written to bits 7:1 of the I2CDR and bit 0 of the I2CDR is set to 0 for mode = XMIT or 1 for mode = RCV. A DLI-global variable MasterRcvAddress is set if mode = RCV (used by I2C_ISR function).

```
I2CStatus I2C_Stop( unsigned int eumbbar );
```

- eumbbar is the address of the Embedded Utilities Memory Block
- Returns: Any defined status indicator

Description:

Generate a STOP signal to terminate the master transaction.

```
I2CStatus I2C_Master_Xmit( unsigned int eumbbar );
```

- eumbbar is the address of the Embedded Utilities Memory Block
- Returns: Any defined status indicator

Description:

Master sends one byte of data to slave receiver. The DLI global variables ByteToXmit, XmitByte, and XmitBufEmptyStop are used to determine which data byte, or STOP, to transmit. If a data byte is sent, it is written to the I2CDR. This function may only be called when the following conditions are met: I2CSR.MIF = 1 I2CSR.MCF = 1 I2CSR.RXAK = 0 I2CCR.MSTA = 1 I2CCR.MTX = 1

```
I2CStatus I2C_Master_Rcv( unsigned int eumbbar );
```

- eumbbar is the address of the Embedded Utilities Memory Block
- Returns: Any defined status indicator

Description:

Master receives one byte of data from slave transmitter. The DLI global variables ByteToRcv, RcvByte, and RcvBufFulStop are used to control the accepting of the data byte or sending of a STOP if the buffer is full. This function may only be called when the following conditions are met: I2CSR.MIF = 1 I2CSR.MCF = 1 I2CCR.MSTA = 1 I2CCR.MTX = 0

```
I2CStatus I2C_Slave_Xmit( unsigned int eumbbar );
```
[NOTE untested]

- eumbbar is the address of the Embedded Utilities Memory Block
- Returns: I2CSUCCESS if data byte sent
  I2CBUFFEMPTY if no data in sending buffer

Description:

Slave sends one byte of data to requesting master. The DLI global variables ByteToXmit, XmitByte, and XmitBuf are used to determine which byte, if any, to send. This function may only be called when the following conditions are met: I2CSR.MIF = 1 I2CSR.MCF = 1 I2CSR.RXAK = 0 I2CCR.MSTA = 0 I2CCR.MTX = 1

```
I2CStatus I2C_Slave_Rcv(unsigned int eumbbar );
```
[NOTE untested]

- eumbbar is the address of the Embedded Utilities Memory Block
- Returns: I2CSUCCESS if data byte received
  I2CBUFFFULL if buffer is full or no more data expected

Description:

Slave receives one byte of data from master transmitter. The DLI global variables ByteToRcv, RcvByte, and RcvBufFulStop are used to control the accepting of the data byte or setting the acknowledge bit (I2CCR.TXAK) if the expected number of bytes have been received. This function may only be called when the following conditions are met: I2CSR.MIF = 1 I2CSR.MCF = 1 I2CCR.MSTA = 0 I2CCR.MTX = 0

```
I2CStatus I2C_Slave_Addr( unsigned int eumbbar );
```
[NOTE untested]

- eumbbar is the address of the Embedded Utilities Memory Block
- Returns: I2CADDRESS if asked to receive data
  results from call to I2C_Slave_Xmit if asked to transmit data

Description:

Process slave address phase. Called from I2C_ISR. This function may only be called when the following conditions are met: I2CSR.MIF = 1 I2CSR.MAAS = 1

```
I2CStatus I2C_ISR(unsigned int eumbbar );
```
- eumbbar is the address of the Embedded Utilities Memory Block
- Returns:
  - I2CADDRESS if address phase for master receive
  - results from call to I2C_Slave_Addr if being addressed as slave (untested)
  - results from call to I2C_Master_Xmit if master transmit data mode
  - results from call to I2C_Master_Rcv if master receive data mode
  - results from call to I2C_Slave_Xmit if slave transmit data mode (untested)
  - results from call to I2C_Slave_Rcv if slave receive data mode (untested)
  - I2CSUCCESS if slave has not acknowledged, generated STOP (untested)
  - I2CSUCCESS if master has not acknowledged, wait for STOP (untested)
  - I2CSUCCESS if bus arbitration lost (untested)

Description:

Read the I2CCR and I2CSR to determine why the I2CSR.MIF bit was set which caused this function to be called. Handle condition, see above in possible return values. This function is called in polling mode as the handler function when an I2C event has occurred. It is intended to be a model for an interrupt service routine for polling mode, but this is untested and the design has not been reviewed or confirmed. This function may only be called when the following condition is met: I2CSR.MIF = 1

> This function is tested only for the master-transmit and master-receive in polling mode. I don't think it is tested even in those modes for situations when the slave does not acknowledge or bus arbitration is lost or buffers overflow, etc.

J.5.1 DLI Functions Written but not Used and not Tested:

```
I2CStatus I2C_write( unsigned int eumbbar,
unsigned char *buffer_ptr,
unsigned int length,
unsigned int stop_flag );
```

- eumbbar is the address of the Embedded Utilities Memory Block
- buffer_ptr is pointer to the data buffer to transmit
- length is the number of bytes in the buffer
- stop_flag: 1 - signal STOP when buffer is empty
- 0 - don't signal STOP when buffer is empty
- Returns: Any defined status indicator

Description:

Send a buffer of data to the requiring master. If stop_flag is set, after the whole buffer is sent, generate a STOP signal provided that the requiring receiver doesn't signal the STOP in the middle. Caller is the slave performing transmitting.

```
I2CStatus I2C_read( unsigned int eumbbar,
unsigned char *buffer_ptr,
unsigned int length,
unsigned int stop_flag );
```

- eumbbar is the address of the Embedded Utilities Memory Block
- buffer_ptr is pointer to the data buffer to transmit
- length is the number of bytes in the buffer
- stop_flag: 1 - signal STOP when buffer is empty
- 0 - don't signal STOP when buffer is empty
- Returns: Any defined status indicator

Description:

Receive a buffer of data from the sending master. If stop_flag is set, when the buffer is full and the sender does not signal STOP, generate a STOP signal. Caller is the slave performing receiving.

J.6 I2C support functions

```
unsigned int get_eumbbar( );
```

- Returns: base address of the Embedded Utilities Memory Block

Description:

See Embedded Utilities Memory Block and Configuration Register Summary for information about the Embedded Utilities Memory Block Base Address Register. This function is defined in kahlua.s.

```
unsigned int load_runtime_reg( unsigned int eumbbar,
& nbsp; unsigned int reg);
```

- eumbbar is the address of the Embedded Utilities Memory Block
- reg specifies the register: I2CDR, I2CFDR, I2CADR, I2CSR, I2CCR
- Returns: register content

Description:

The content of the specified register is returned. This function is defined in drivers/i2c/i2c2.s.

```
unsigned int store_runtime_reg( unsigned int eumbbar,
& nbsp; unsigned int reg,
& nbsp; unsigned int val);
```

- eumbbar is the address of the Embedded Utilities Memory Block
- offset specifies the register: I2CDR, I2CFDR, I2CADR, I2CSR, I2CCR
- val is the value to be written to the register
- Return: No return value used, it should be declared void.

Description:

The value is written to the specified register. This function is defined in drivers/i2c/i2c2.s

Appendix K MPC8240 I2O Doorbell Driver

K.1 I2O Description of Doorbell Communication between Agent and Host

The sequence of events that transpire during communication via the I2O doorbell registers between host and agent applications running on Kahlua are described. This implementation enables basic doorbell communication. It can be expanded to include other Kahlua message unit activity via the message registers and the I2O message queue.

K.1.1 System startup and memory map initialization

An understanding of the agent's Embedded Utilities Memory Block Base Address Register (EUMBBAR) and Peripheral Control and Status Registers Base Address Register (PCSRBAR) is important for I2O doorbell communication because both host and agent use the agent's inbound and outbound doorbell registers and message unit status and control registers. The host accesses the agent's registers via the agent's PCSR and the agent accesses its own registers via its own EUMB. It is worth noting that some registers, such as the doorbell registers, can be accessed via either the PCSR or the EUMB. Other registers, such as the message unit's status and interrupt mask registers, can only be accessed via one or the other of the PCSR or EUMB, but not both. The I2O library functions require the caller to provide the base address (which will be either the PCSR or the EUMB) and a parameter indicating which is used. In the DINK32 environment, functions are provided to obtain both of these base addresses: get_kahlua_pcsrbar() and get_eumbbar(). The methods of setting and obtaining the PCSR and EUMB base addresses are application-specific, but the register offsets and bit definitions of the registers are specified for the Kahlua chip memory map B and will be the same for all applications. Details about the register offsets within the EUMB and PCSR as well as bit definitions within registers are found in the Kahlua or Kahlua User's Manual.

When the Kahlua host and agent come up running the DINK32 application, the host application assigns the agent's PCI address for the PCSR and writes it in the agent's PCSRBAR by calling config_kahlua_agent(). The agent application initializes its own EUMBBAR [this actually happens in the KahluaInit() function, defined in .../kahlua.s] and inbound and outbound address translation windows. This is done during initialization in the main() function, main.c.

```
/*
              ** Try to enable a Kahlua slave device. This is only
enabled for Map B.
             */
             if (address_map[0] == 'B')
```

```
                if (target_mode == 0)


                     /* probe PCI to see if we have a kahlua */
                    if (pciKahluaProbe( KAHLUA_ID_LO, VENDOR_ID_HI,
&target_add
r)==1)


                        PRINT("Host ....\n");
                        config_kahlua_agent( );
                        }
                    }
                 else if (target_type == (( KAHLUA_ID_LO << 16 ) |
VENDOR_ID_HI ))
PRINT("Agent ....\n");
                        /* Inbound address translation */
                  sysEUMBBARWrite(L_ATU_ITWR, ATU_BASE|ATU_IW_64K);
                             pciRegSet(PCI_REG_BASE, PCI_LMBAR_REG,
PCI_MEM_ADR);
                        /* Outbound address translation */
                  sysEUMBBARWrite(L_ATU_OTWR, 0x100000|ATU_IW_64K);
                        sysEUMBBARWrite(L_ATU_OMBAR, 0x81000000);
                        }
                }
```

K.1.2 Interrupt Service Routines: I2O_ISR_host() and I2O_ISR_agent()

There is a fundamental difference in the interrupt service routine (ISR) for the host and agent: the I2O_ISR_agent function only has to handle inbound message unit interrupts, but the I2O_ISR_host must handle any possible interrupt from a Kahlua agent, not limited to the agent's outbound message unit. The ISRs implemented at present just check for doorbell activity. If a doorbell event occurred, the ISR prints out a simple message including the doorbell register content and the doorbell register is cleared. Otherwise, the ISR prints a message that it was unable to determine the cause of the interrupt. The I2O_ISR_agent function checks the Inbound Message Interrupt Status Register (IMISR) to determine the cause of the message unit interrupt. The Message Unit interrupt can occur because of doorbell, message register, or message queue activity. The ISR will distinguish and handle the interrupt accordingly; but at first stage implementation, only doorbell interrupts will be handled.

The I2O library function I2OInMsgStatGet() is used to read the IMISR. It returns the content of the IMISR after applying the mask value in the Inbound Message Interrupt Mask Register (IMIMR) and clears the status register. The I2O library function I2ODBGet() is used to read the IDBR. It returns the content and clears the register. Similarly, the I2O_ISR_host function checks the agent's Outbound Message Interrupt Status Register

(OMISR) to determine if the cause of the interrupt was due to the agent's outbound doorbell. It is important to note that the I2O_ISR_host must be expanded to check for any kind of expected interrupt from the agent, not just message unit interrupts. The I2O library function I2OOutMsgStatGet() is used to read the OMISR. It returns the content of the OMISR after applying the mask value in the Outbound Message Interrupt Mask Register (OMIMR) and clears the status register. The I2O library function I2ODBGet() is used to read the ODBR. It returns the content and clears the register.

The two functions I2O_ISR_host() and I2O_ISR_agent() are defined in the source file .../drivers/i2o/i2o1.c and are linked into the libdriver.a library. For testing, they are currently manually called when requested by the user in the function test_i2o(). Eventually, the host and agent will register an interrupt service routine (ISR) with their respective Embedded Programmable Interrupt Controller (EPIC) systems. Details about how to register the ISRs with EPIC are not yet specified. It may take the form of a function call to an EPIC-provided function that accepts a pointer to the ISR function. Alternately, it may be integrated by the linker by placing a reference to the ISR functions in some configuration table. When the integration takes place, this document will be updated to reflect the details. The code for the entire I2O_ISR_host function follows. Note that the only type of interrupt that is currently handled is doorbell interrupt from the message unit, but there are comments in the code indicating where to check for other causes of interrupts. The code can be found in i2o1.c.

K.1.3 Enable Doorbell Interrupts:

Since the agent is servicing the inbound doorbell, the agent enables it by calling the I2O library function I2ODBEnable(), which clears the Inbound Doorbell Interrupt Mask (IDIM) bit in the Inbound Doorbell Interrupt Mask Register (IMIMR). The IMIMR is at offset 0x104 in the agent's Embedded Utilities Memory Block (EUMB), whose address is in the agent's EUMBBAR. Similarly, since the host is servicing the agent's outbound doorbell, the host enables it by calling the I2O library function I2ODBDisable(), which clears the Outbound Doorbell Interrupt Mask (ODIM) bit in the agent's Outbound Message Interrupt Mask Register (OMIMR). The OMIMR is at offset 0x34 in the agent's PCSR block, whose address is in the agent's PCSRBAR at offset 0x14 in the agent's Configuration Registers.

The address of the agent's Configuration Registers are known by the host and are accessible from the PCI bus. At present, the user interface in DINK32 allows the user to set or clear the ODIM or IDIM bit. The functions I2ODBEnable() and I2ODBDisable() are defined in .../drivers/i2o/i2o1.c to perform this task. See the code in test_i2o() for a usage example. It is interesting to note that the observed behavior of the Kahlua chip with regard to message unit registers is not dependant on the ODIM and IDIM bit settings Even if the ODIM or IDIM mask bits are set, writes to the affected doorbell are not blocked and the appropriate bit is set in the message unit's status register. It is up to software to apply the mask to the status register to determine whether or not to take any action. The interrupt should not occur

if the mask bit is set, but this has not yet been tested.

K.1.4 Writing and Reading Doorbell Registers:

The functions I2ODBPost() and I2ODBGet() are defined in .../drivers/i2o/i2o1.c to write a bit pattern to or return the contents of the agent's inbound and outbound doorbell registers. Note that the agent application accesses both inbound and outbound doorbell registers via its own EUMB and the host application accesses these same doorbell registers via the agent's PCSR. See the code in test_i2o() for usage examples.

K.1.4.1 Host Rings an Agent via Agent's Inbound Doorbell

The host application calls the I2O library function I2ODBPost() to write the bit pattern to the agent's Inbound Door Bell Register (IDBR). If the inbound doorbell is enabled, this generates a Message Unit interrupt to the agent processor and the agent's EPIC unit will execute the I2O_ISR_agent function to determine the cause of the message unit interrupt and handle it appropriately. If the inbound doorbell is not enabled, no interrupt is generated; but the doorbell and the status register bit are still set. The agent application reads the IDBR by calling the I2O library function I2ODBGet(). This clears the IDBR.

K.1.4.2 Agent Rings a Host via Agent's Outbound Doorbell

The agent application calls the I2O library function I2ODBPost() to write the bit pattern to the agent's Outbound Door Bell Register (ODBR). If the outbound doorbell is enabled, this causes the outbound interrupt signal INTA_ to go active which interrupts the host processor. After the ISR is integrated into the EPIC unit, this mechanism will be documented here. If the outbound doorbell is not enabled, no interrupt is generated; but the doorbell and the status register bit are still set. The host application reads the ODBR by calling the I2O library function I2ODBGet(). This clears the ODBR.

Sample application code. Here is some sample code from the DINK32 function test_i2o() in device.c that provides examples of how the I2O library functions can be used by an application. When this section of code is entered, the DINK32 user interface has already set the local variables "mode" and "bit". Mode reflects the user request. Bit is the doorbell bit number to set. Mode = 4 to manually run the ISR's for testing prior to integration with EPIC.

```
/* different depending on if DINK = is running on host or agent */
   if (target_mode  0)
  {
    /*  running on host */
     unsigned int kahlua_pcsrbar  get_kahlua_pcsrbar();
    /* PRINT("kahlua's pcsrbar   0x%x\n",kahlua_pcsrbar); */
    switch (mode)
    {
      case 0:
      /* read agent's  outbound DB register and print it out */
      db_reg_content  I2ODBGet(REMOTE,kahlua_pcsrbar);
```

```
                PRINT("Agent's     outbound   doorbell   register:
0x%x\n",db_reg_content);
      break;

      case = 1:
      /* set  agent's inbound doorbell register */
      db_reg_content  1 << bit;
      I2ODBPost(REMOTE,kahlua_pcsrbar,db_reg_content);
      break;

      case = 2:
      /* enable  agent's outbound DB register interrupts */
      if (I2ODBEnable(REMOTE,kahlua_pcsrbar,0) ! = I2OSUCCESS)
PRINT("Cannot enable agent's outbound doorbell interrupt.\n");
      else
         PRINT("Enabled agent's outbound doorbell interrupt.\n");
       break;

      case = 3:
      /*  disable agent's outbound DB register interrupts */
      if (I2ODBDisable(REMOTE,kahlua_pcsrbar,0) ! = I2OSUCCESS)
              PRINT("Cannot  disable  agent's  outbound  doorbell
interrupt.\n");
      else
         PRINT("Disabled agent's outbound doorbell interrupt.\n");
       break;

      #ifdef DBG_I2O
      case 4:
      I2O_ISR_host();
      break;
      #endif
    }
  }
  else
  {
/* running on agent */
    /*  PRINT("kahlua's eumbbar  0x%x\n",eumbbar); */
    switch (mode)
    {
      case 0:
      /* read agent's inbound  DB register and print it out */
      db_reg_content  I2ODBGet(LOCAL,eumbbar);
               PRINT("Agent's     inbound   doorbell   register:
0x%x\n",db_reg_content);
      break;

      case = 1:
      /* set  agent's outbound doorbell register */
      db_reg_content  1  << bit;
      I2ODBPost(LOCAL,eumbbar,db_reg_content);
       break;

      case = 2:
```

```
      /* enable  agent's inbound DB register interrupts */
      if  (I2ODBEnable(LOCAL,eumbbar,3) ! I2OSUCCESS)
       PRINT("Cannot enable agent's inbound doorbell interrupt.\n");
        else
PRINT("Enabled agent's inbound doorbell interrupt.\n");
       break;

      case = 3:
      /*  disable agent's inbound DB register interrupts */
      if  (I2ODBDisable(LOCAL,eumbbar,3) ! I2OSUCCESS)
                PRINT("Cannot  disable  agent's  inbound  doorbell
interrupt.\n");
       else
        PRINT("Disabled agent's inbound doorbell interrupt.\n");
       break;

      #ifdef DBG_I2O
      case 4:
      I2O_ISR_agent();
      break;
      #endif
    }
  }
```

K.1.4.3 Descriptions of the I2O library functions

```
I2OSTATUS I2ODBEnable ( LOCATION loc,unsigned int base,unsigned int
in_db )
```

- loc = LOCAL or REMOTE: Use LOCAL if called from agent, REMOTE if called from host. This controls the use of the base parameter as PCSR (ifREMOTE) or EUMB (if LOCAL) and selection of outbound (if REMOTE) or inbound(if LOCAL) mask registers.
- base is the base address of PCSR or EUMB.
- in_db is used for LOCAL to control enabling of doorbell and/or machine check
- Returns: I2OSUCCESS

Description:

Enable the specified doorbell interrupt by clearing the appropriate mask bits.

```
I2OSTATUS I2ODBDisable( LOCATION loc,unsigned int base,unsigned int
in_db )
```

- Same as I2ODBEnable, but it disables the specified interrupts bysetting the mask bits.

```
unsigned int I2ODBGet( LOCATION loc,unsigned int base)
```

- loc = LOCAL or REMOTE: Use LOCAL if called from agent, REMOTE ifcalled from host. This controls the use of the base parameter as PCSR (ifREMOTE) or EUMB (ifLOCAL) and selection of outbound (if REMOTE) or inbound(if LOCAL) doorbell registers.
- base is the base address of PCSR or EUMB.

- Returns:Contents of agent's inbound (if loc = LOCAL) or outbound (if loc REMOTE) doorbell register.

Description:

Returns content of specified doorbell register and clears the doorbell register.

```
void I2ODBPost( LOCATION loc,unsigned int base,unsigned int msg )
```

- loc = LOCAL or REMOTE: Use LOCAL if called from agent, REMOTE ifcalled from host. This controls the use of the base parameter as PCSR (ifREMOTE) or EUMB (if LOCAL) and selection of outbound (if REMOTE) or inbound(if LOCAL) doorbell registers.
- base is the base address of PCSR or EUMB
- msg is the 32 bit value written to the specified doorbell register

Description:

The 32 bit value is written to the specified doorbell register.

```
I2OSTATUS I2OInMsgStatGet(unsigned int eumbbarI2OIMSTAT *val)
```

- eumbbar is the base address of the agent's EUMB
- *val receives the agent's inbound message interrupt statusregister
- Returns: I2OSUCCESS

Description:

The agent's Inbound Message Interrupt Status Register (IMISR)content is masked by the agent's Inbound Message Interrupt Mask Register(IMIMR) and placed in the address given in the val parameter. The IMISRregister is cleared.

```
I2OSTATUS I2OOutMsgStatGet(unsigned int pcsrbar,I2OOMSTAT *val)
```

- pcsrbar is the base address of the agent's PCSR
- *val receives the agent's outbound message interrupt statusregister
- Returns: I2OSUCCESS

Description:
The agent's Outbound Message Interrupt Status Register (OMISR)content is masked by the agent's Outbound Message Interrupt Mask Register(OMIMR) and placed in the address given in the val parameter. The OMISRregister is cleared.

K.2 I2C Driver Library

This section provides information about the generic Application Program Interface (API) to the I2C Driver Library as well as information about the implementation of the Kahlua-specific I2C Driver Library Internals (DLI).

K.2.1 Background

The intended audience for this document is assumed to be familiar with the I2C bus protocol. It is a companion document to the Kahlua specification and other documentation which collectively give details of the I2C protocol and the Kahlua implementation. This document provides information about the software written to access the Kahlua I2C interface. This software is intended to assist in the development of higher level applications software that uses the I2C interface.

> Note: The I2C driver software is currently under development. The only modes that are functional are the master-transmit and master-receive in polling mode.

K.2.2 Overview

This document consists of these parts:

- An Application Program Interface (API) which provides a very simple, generic, application level programmatic interface to the I2C driver library that hides all details of the Kahlua-specific implementation of the I2C interface (i.e., control register, status register, embedded utilities memory block, etc.).
- I2C API functions showing the following:
  - how the function is called (i.e., function prototype)
  - parameter definition possible
  - return values
  - brief description of what the function does
  - an explanation of how the functions are used by an application program (DINK32 usage employed as examples)
- An I2C Driver Library Internals (DLI) which provides information about the lower level software that is accessing the Kahlua-specific implementation of the I2C interface.
- I2C DLI functions showing the following:
  - how the function is called (i.e., function prototype)
  - parameter definition
  - possible return values
  - brief description of what the function does

K.2.3 I2C Application Program Interface (API)

K.2.3.1 API functions description

The I2C API function prototypes, defined return values, and enumerated input parameter

values are declared in drivers/i2c/i2c_export.h. The functions are defined in the source file drivers/i2c/i2c1.c.

```
I2C_Status I2C_Initialize( unsigned char addr, I2C_INTERRUPT_MODE
en_int, int (*app_print_function)(char *,...));
```

- addr is the Kahlua chip's I2C slave device address
- en_int controls the I2C interrupt enable status: I2C_INT_ENABLE = enable, I2C_INT_DISABLE = disable
- app_print_function is the address of the optional application's print function, otherwise NULL if not available
- Return: I2C_Status return value is either I2C_SUCCESS or I2C_ERROR.

Description:

Configure the I2C library prior to use, as follows:

The interrupt enable should be set to I2C_INT_DISABLE, the I2C library currently only supports polling mode.

The slave address can be set to the I2C listening address of the device running the application program, but the DLI does not yet support the application's device responding as an I2C slave to another I2C master device.

The optional print function, if supplied by the application, must be similar to the C standard library printf library function: accepts a format string and a variable number (zero or more) of additional arguments. This optional function may be used by the I2C library functions to report error and status condition information. If no print function is supplied by the application, the call to I2C_Initialize must provide a NULL value for this parameter, in which case the I2C library will not attempt to access a print function.

```
I2C_Status I2C_do_transaction( I2C_INTERRUPT_MODE en_int,
I2C_TRANSACTION_MODE act,
unsigned char i2c_addr,
unsigned char data_addr,
int len,
char *buffer,
I2C_STOP_MODE stop,
int retry,
I2C_RESTART_MODE rsta);
Where:
```

- en_int controls the I2C interrupt enable status (currently use I2C_INT_DISABLE only, polling mode)
- act is the type of transaction: I2C_MASTER_RCV or I2C_MASTER_XMIT
- i2c_addr is the I2C address of the slave device
- data_addr is the address of the data on the slave device

- len is the length in bytes of the data
- buffer is a pointer to the buffer that contains the data (xmit mode) or receives the data (rcv mode)
- stop controls sending an I2C STOP signal after completion (curently use I2C_STOP only)
- retry is the timeout retry value (currently ignored)
- rsta controls I2C restart (currently use I2C_NO_RESTART only)
- Return: I2C_Status return value is either I2C_SUCCESS or I2C_ERROR.

Description:

Act as the I2C master to transmit (or receive) a buffer of data to (or from) an I2C slave device.

This function currently only implements a simple master-transmit or a master-receive transaction. It does not yet support the application retaining I2C bus ownership between transactions, operating in interrupt mode, or acting as an I2C slave device.

K.2.3.2 API Example Usage

The ROM monitor program DINK32 uses the I2C API in both currently implemented modes: master-transmit and master-receive. The DINK32 program runs interactively to allow the user to transmit or receive a buffer of data from an I2C device at address 0x50 on the Kahlua PMC card. DINK32 obtains information from the user as follows: read/write mode, I2C device address for the data (this is the address of the data on the I2C device, not the I2C bus address of the device itself, which is hard-coded in DINK32), the raw data (if in write mode), and the length of the data to transfer to or from the device. Note that the initialization call to configure the I2C interface is actually made only once, the first time the user requests an I2C transmit or receive operation. Each transmit or receive operation is performed by a single call to an I2C API function. The DINK32 program is an interactive application, so it gives the I2C library access to its own print output function.

These are the steps DINK32 takes to perform a master-transmit transaction:

1. Call I2C_Initialize (if needed) to set the Kahlua I2C address, polling mode, and identify the optional print function.
2. Call I2C_do_transaction to transmit the buffer of data.

These are the steps DINK32 takes to perform a master-receive transaction in polling mode:

1. Call I2C_Initialize (if needed) to set the Kahlua I2C address, polling mode, and identify the optional print function.
2. Call I2C_do_transaction to receive the buffer of data.

The following code samples have been excerpted from the DINK32 application to illustrate

the use of the I2C API from par_devtest in device.c:

```
#define PRINT dink_printf
int dink_printf( unsigned char *fmt, ... )
{
/* body of application print output function, */
}
/* In the function par_devtest, for testing the I2C device interface
*/
{
/* initialize the I2C handler to I2C address 48, if needed */
if ( I2CInited == 0 )
{
I2C_Status status;
if ((status = I2C_Initialize( 48, en_int, PRINT ) ) != I2C_SUCCESS )
{
PRINT( "devtest I2C: error in initiation\n" );
return ERROR;
} else {
I2CInited = 1;
}
}
return test_i2c( action, en_int );
}
static unsigned char rcv_buffer[BUFFER_LENGTH] = { 0 };
static unsigned char xmit_buffer[BUFFER_LENGTH] = { 0 };
/*******************************************************
* function: test_i2c
*
* description: run i2c test by polling the device
*
* note:
* Test i2c device on PMC card, 0x50 serial EPROM.
* The device test data is currently only printable characters.
*
* This function gets some data from the command line, validates it,
* and calls the I2C library function to perform the task.
*******************************************************/
static STATUS test_i2c( int act, int en_int )
{
int retry = 800, len = 0, rsta = 0, addr = 0;
unsigned char eprom_addr = 0x50;
/* read transaction address */
... addr ...
/* read # of bytes to transfer */
... len ...

/* validate the data address, length, etc. */
...
/* If transmitting, get the raw data into the transmit buffer */
... xmit_buffer[] ...
/* read built-in I2C device on Kahlua PMC card */
if ( act == DISPLAY_TAG )
{
```

```
if ( I2C_do_transaction ( en_int, I2C_MASTER_RCV, eprom_addr, addr,
len, rcv_buffer, I2C_STOP, retry, I2C_NO_RESTART ) != I2C_SUCCESS )
{
PRINT( "dev I2C: error in master receive test\n" );
return ERROR;
} else {
rcv_buffer[len] = 0; /* ensure NULL terminated string */
PRINT( "%s",rcv_buffer); /* expecting only printable data */
PRINT( "\n" );
}
}
/* write to built-in I2C device on Kahlua PMC card */
if ( act == MODIFY_TAG )
{
if ( I2C_do_transaction ( en_int, I2C_MASTER_XMIT, eprom_addr, addr,
len, xmit_buffer, I2C_STOP, retry, I2C_NO_RESTART ) != I2C_SUCCESS )
{
PRINT( "dev I2C: error in master transmit test\n" );
return ERROR;
}
}
return SUCCESS;
}
```

K.2.4 I2C Driver Library Internals (DLI)

This information is provided to assist in further development of the I2C library to enable the application to operate as an I2C slave device, interrupt enabled mode, bus retention between consecutive transactions, correct handling of device time out, no slave device response, no acknowledgment, I2C bus arbitration loss, etc.

All of these functions are defined as static in the source file drivers/i2c/i2c1.c.

K.2.4.1 Common Data Structures and Values

These data structures and status values are defined (see drivers/i2c/i2c.h) for the Kahlua I2C driver library functions:

These are the offsets in the Embedded Utilities Memory Block for the I2C registers.

```
#define I2CADR 0x00003000
#define I2CFDR 0x00003004
#define I2CCR 0x00003008
#define I2CSR 0x0000300C
#define I2CDR 0x00003010
typedef enum _i2cstatus
{
I2CSUCCESS = 0x3000,
I2CADDRESS,
I2CERROR,
I2CBUFFFULL,
I2CBUFFEMPTY,
I2CXMITERROR,
```

```
I2CRCVERROR,
I2CBUSBUSY,
I2CALOSS,
I2CNOEVENT,
} I2CStatus;
These structures reflect the bit assignments of the I2C registers.
typedef struct _i2c_ctrl
{
unsigned int reserved0 : 24;
unsigned int men : 1;
unsigned int mien : 1;
unsigned int msta : 1;
unsigned int mtx : 1;
unsigned int txak : 1;
unsigned int rsta : 1;
unsigned int reserved1 : 2;
} I2C_CTRL;
typedef struct _i2c_stat
{
unsigned int rsrv0 : 24;
unsigned int mcf : 1;
unsigned int maas : 1;
unsigned int mbb : 1;
unsigned int mal : 1;
unsigned int rsrv1 : 1;
unsigned int srw : 1;
unsigned int mif : 1;
unsigned int rxak : 1;
} I2C_STAT;
Values to indicate receive or transmit mode.
typedef enum _i2c_mode
{
RCV = 0,
XMIT = 1,
} I2C_MODE;
```

K.2.4.2 Kahlua I2C Driver Library Internals: function descriptions

```
I2CStatus I2C_Init( unsigned int eumbbar,
unsigned char fdr,
unsigned char addr,
unsigned int en_int);
```

- eumbbar is the address of the Embedded Utilities Memory Block
- fdr is the frequency divider value used to set the I2C clock rate
- addr is the Kahlua chip's I2C slave device address
- en_int controls the I2C interrupt enable status: 1 = enable, 0 = disable
- Return: I2CStatus return value is always I2CSUCCESS.

Description:

Set the frequency divider (I2CFDR:FDR), listening address (I2CADR:[7:1]), and interrupt

enable mode (I2CCR:MIEN).

```
I2C_CTRL I2C_Get_Ctrl( unsigned int eumbbar );
```

: eumbbar is the address of the Embedded Utilities Memory Block

- Return: I2C_CTRL is the contents of the I2C control register (I2CCR)

Description:

Read the I2C control register.

```
void I2C_Set_Ctrl( unsigned int eumbbar, I2C_CTRL ctrl);
```

- eumbbar is the address of the Embedded Utilities Memory Block
- ctrl is the contents of the I2C control register (I2CCR)
- Return: none

Description:

Set the I2C control register.

```
I2CStatus I2C_put( unsigned int eumbbar,
unsigned char rcv_addr,
unsigned char *buffer_ptr,
unsigned int length,
unsigned int stop_flag,
unsigned int is_cnt );
```

- eumbbar is the address of the Embedded Utilities Memory Block
  rcv_addr is the receiver's I2C device address
- buffer_ptr is pointer to the data buffer to transmit
- length is the number of bytes in the buffer
- stop_flag: 1 - signal STOP when buffer is empty
- 0 - don't signal STOP when buffer is empty
- is_cnt: 1 - this is a restart, don't check MBB
- 0 - this is a not restart, check MBB
- Returns: Any defined status indicator

Description:

Set up to send a buffer of data to the intended rcv_addr. If stop_flag is set, after the whole buffer is sent, generate a STOP signal provided that the receiver doesn't signal the STOP in the middle. Caller is the master performing transmitting. If no STOP signal is generated at the end of current transaction, the master can generate a START signal to another slave address.

The function does not actually perform the data buffer transmit,

it just sets up the DLI global variables to control the transaction and calls I2C_Start to send the slave address out on the I2C bus in transmit mode. The application must check the return status to find out if the bus was obtained, then enter a loop of calling I2C_Timer_Event to poll the I2C handler to actually perform the transaction one byte at a time, while checking the return status to determine if there were any errors and if the transaction has completed.

```
I2CStatus I2C_get( unsigned int eumbbar,
unsigned char sender_addr,
unsigned char *buffer_ptr,
unsigned int length,
unsigned int stop_flag,
unsigned int is_cnt );
```

- eumbbar is the address of the Embedded Utilities Memory Block
- sender_addr is the sender's I2C device address
  buffer_ptr is pointer to the data buffer to transmit
- length is the number of bytes in the buffer
- stop_flag: 1 - signal STOP when buffer is empty
- 0 - don't signal STOP when buffer is empty
- is_cnt: 1 - this is a restart, don't check MBB
- 0 - this is a not restart, check MBB
- Returns: Any defined status indicator

Description:

Set up to receive a buffer of data from the desired sender_addr. If stop_flag is set, when the buffer is full and the sender does not signal STOP, generate a STOP signal. Caller is the master performing receiving. If no STOP signal is generated, the master can generate a START signal to another slave address.

The function does not actually perform the data buffer receive, it just sets up the DLI global variables to control the transaction and calls I2C_Start to send the slave address out on the I2C bus in receive mode. The application must check the return status to find out if the bus was obtained, then enter a loop of calling I2C_Timer_Event to poll the I2C handler to actually perform the transaction one byte at a time, while checking the return status to determine if there were any errors and if the transaction has completed.

```
I2CStatus I2C_Timer_Event( unsigned int eumbbar, I2CStatus
(*handler)( unsigned int ) );
```

- eumbbar is the address of the Embedded Utilities Memory Block

- handler is a pointer to the function to call to handle any existing status event,
- Returns: I2CNOEVENT if there is no completed event, the I2CSR MIF bit is not set results from call to the handler function if there was a pending event completed

Description:

In polling mode, I2C_Timer_Event can be called to check the I2C status and call the given (or the default: I2C_ISR) handler function if the I2CSR MIF bit is set.

```
I2CStatus I2C_Start( unsigned int eumbbar,
unsigned char slave_addr,
I2C_MODE mode,
unsigned int is_cnt );
```

- eumbbar is the address of the Embedded Utilities Memory Block
- slave_addr is the I2C address of the receiver
- mode: XMIT(1) - put (write)
- RCV(0) - get (read)
- is_cnt: 1 - this is a restart, don't check MBB
- 0 - this is a not restart, check MBB
- Returns: Any defined status indicator

Description:

Generate a START signal in the desired mode. Caller is the master. The slave_addr is written to bits 7:1 of the I2CDR and bit 0 of the I2CDR is set to 0 for mode = XMIT or 1 for mode = RCV. A DLI-global variable MasterRcvAddress is set if mode = RCV (used by I2C_ISR function).

```
I2CStatus I2C_Stop( unsigned int eumbbar );
```

- eumbbar is the address of the Embedded Utilities Memory Block
- Returns: Any defined status indicator

Description:

Generate a STOP signal to terminate the master transaction.

```
I2CStatus I2C_Master_Xmit( unsigned int eumbbar );
```

- eumbbar is the address of the Embedded Utilities Memory Block
- Returns: Any defined status indicator

Description:

Master sends one byte of data to slave receiver. The DLI global variables ByteToXmit, XmitByte, and XmitBufEmptyStop are used to determine which data byte, or STOP, to transmit. If a data byte is sent, it is written to the I2CDR. This function may only be called when the following conditions are met: I2CSR.MIF = 1 I2CSR.MCF = 1 I2CSR.RXAK =

0 I2CCR.MSTA = 1 I2CCR.MTX = 1

```
I2CStatus I2C_Master_Rcv( unsigned int eumbbar );
```

- eumbbar is the address of the Embedded Utilities Memory Block
- Returns: Any defined status indicator

Description:

Master receives one byte of data from slave transmitter. The DLI global variables ByteToRcv, RcvByte, and RcvBufFulStop are used to control the accepting of the data byte or sending of a STOP if the buffer is full. This function may only be called when the following conditions are met: I2CSR.MIF = 1 I2CSR.MCF = 1 I2CCR.MSTA = 1 I2CCR.MTX = 0

```
I2CStatus I2C_Slave_Xmit( unsigned int eumbbar );
```
[NOTE untested]

- eumbbar is the address of the Embedded Utilities Memory Block
- Returns: I2CSUCCESS if data byte sent I2CBUFFEMPTY if no data in sending buffer

Description:

Slave sends one byte of data to requesting master. The DLI global variables ByteToXmit, XmitByte, and XmitBuf are used to determine which byte, if any, to send. This function may only be called when the following conditions are met: I2CSR.MIF = 1 I2CSR.MCF = 1 I2CSR.RXAK = 0 I2CCR.MSTA = 0 I2CCR.MTX = 1

```
I2CStatus I2C_Slave_Rcv(unsigned int eumbbar );
```
[NOTE untested]

- eumbbar is the address of the Embedded Utilities Memory Block
- Returns: I2CSUCCESS if data byte received I2CBUFFFULL if buffer is full or no more data expected

Description:

Slave receives one byte of data from master transmitter. The DLI global variables ByteToRcv, RcvByte, and RcvBufFulStop are used to control the accepting of the data byte or setting the acknowledge bit (I2CCR.TXAK) if the expected number of bytes have been received. This function may only be called when the following conditions are met: I2CSR.MIF = 1 I2CSR.MCF = 1 I2CCR.MSTA = 0 I2CCR.MTX = 0

```
I2CStatus I2C_Slave_Addr( unsigned int eumbbar );
```
[NOTE untested]

- eumbbar is the address of the Embedded Utilities Memory Block

- Returns: I2CADDRESS if asked to receive data
  results from call to I2C_Slave_Xmit if asked to transmit data

Description:

Process slave address phase. Called from I2C_ISR. This function may only be called when the following conditions are met: I2CSR.MIF = 1 I2CSR.MAAS = 1

```
I2CStatus I2C_ISR(unsigned int eumbbar );
```

- eumbbar is the address of the Embedded Utilities Memory Block
- Returns:
  - — I2CADDRESS if address phase for master receive results from call to I2C_Slave_Addr if being addressed as slave (untested)
  - — results from call to I2C_Master_Xmit if master transmit data mode
  - — results from call to I2C_Master_Rcv if master receive data mode
  - — results from call to I2C_Slave_Xmit if slave transmit data mode (untested)
  - — results from call to I2C_Slave_Rcv if slave receive data mode (untested)
  - — I2CSUCCESS if slave has not acknowledged, generated STOP (untested)
  - — I2CSUCCESS if master has not acknowledged, wait for STOP (untested)
  - — I2CSUCCESS if bus arbitration lost (untested)

Description:

Read the I2CCR and I2CSR to determine why the I2CSR.MIF bit was set which caused this function to be called. Handle condition, see above in possible return values. This function is called in polling mode as the handler function when an I2C event has occurred. It is intended to be a model for an interrupt service routine for polling mode, but this is untested and the design has not been reviewed or confirmed. This function may only be called when the following condition is met: I2CSR.MIF = 1

> [NOTE: This function is tested only for the master-transmit and master-receive in polling mode. I don't think it is tested even in those modes for situations when the slave does not acknowledge or bus arbitration is lost or buffers overflow, etc.]

K.2.4.3 The following DLI functions were written but not used and not tested:

```
I2CStatus I2C_write( unsigned int eumbbar,
unsigned char *buffer_ptr,
unsigned int length,
unsigned int stop_flag );
```

- eumbbar is the address of the Embedded Utilities Memory Block
- buffer_ptr is pointer to the data buffer to transmit
- length is the number of bytes in the buffer

- stop_flag: 1 - signal STOP when buffer is empty
- 0 - don't signal STOP when buffer is empty
- Returns: Any defined status indicator

Description:

Send a buffer of data to the requiring master. If stop_flag is set, after the whole buffer is sent, generate a STOP signal provided that the requiring receiver doesn't signal the STOP in the middle. Caller is the slave performing transmitting.

```
I2CStatus I2C_read( unsigned int eumbbar,
unsigned char *buffer_ptr,
unsigned int length,
unsigned int stop_flag );
```

- eumbbar is the address of the Embedded Utilities Memory Block
- buffer_ptr is pointer to the data buffer to transmit
- length is the number of bytes in the buffer
- stop_flag: 1 - signal STOP when buffer is empty
- 0 - don't signal STOP when buffer is empty
- Returns: Any defined status indicator

Description:

Receive a buffer of data from the sending master. If stop_flag is set, when the buffer is full and the sender does not signal STOP, generate a STOP signal. Caller is the slave performing receiving.

K.2.4.4 I2C support functions

```
unsigned int get_eumbbar( );
```

- Returns: base address of the Embedded Utilities Memory Block

Description:

See Embedded Utilities Memory Block and Configuration Register Summary for information about the Embedded Utilities Memory Block Base Address Register. This function is defined in kahlua.s.

> [NOTE: I don't understand the initialization sequences for establishing the config_addr and config_data well enough at this point to be able to explain them; however, I think it is essential to offer the user a complete explanation of the initialization process.]

```
unsigned int load_runtime_reg( unsigned int eumbbar,
unsigned int reg);
```

- eumbbar is the address of the Embedded Utilities Memory Block

- reg specifies the register: I2CDR, I2CFDR, I2CADR, I2CSR, I2CCR
- Returns: register content

Description:

The content of the specified register is returned. This function is defined in drivers/i2c/i2c2.s.

```
unsigned int store_runtime_reg( unsigned int eumbbar,
unsigned int reg,
unsigned int val);
```

- eumbbar is the address of the Embedded Utilities Memory Block
- offset specifies the register: I2CDR, I2CFDR, I2CADR, I2CSR, I2CCR
- val is the value to be written to the register
- Return: No return value used, it should be declared void.

Description:

The value is written to the specified register. This function is defined in drivers/i2c/i2c2.s

Appendix L MPC8240 EPIC Interrupt Driver

This appendix describes the sample EPIC driver source code provided in this DINK32 release and its usage on the Sandpoint Reference Platform running DINK32.

L.1 General Description

EPIC is the embedded programmable interrupt controller feature implemented on Motorola's MPC8240 integrated processor. It is derived from the Open Programmable Interrupt Controller (PIC) Register Interface Specification R1.2 developed by AMD and Cyrix. EPIC on the MPC8240 provides support for up to five external interrupts or one serial-style interrupt line (supporting 16 interrupts), four internal logic-driven interrupts (DMA0, DMA1, I^2C, I_2O), four global timers, and it supports a pass through mode. Please refer to Chapter 12 of the MPC8240 User's Manual for a more in depth description of EPIC on the MPC8240.

L.2 EPIC Specifics

Unlike other embedded features of the MPC8240 such as DMA and I_2O, the EPIC unit is accessible from the local processor only. The control and status registers of this unit cannot be accessed by external PCI devices. The EPIC registers are accessed as an offset from the Embedded Utilities Memory Block (EUMB). The EPIC unit supports two modes: Mixed and Pass-through.

> The DINK32 EPIC driver sample code demonstrates EPIC in direct mode and also error checks for Pass-through mode, but Serial mode is not implemented due to the current inability to test this mode on the Sandpoint reference platform.
>
> The EPIC registers are in little-endian format. If the system is in big-endian mode, the bytes must be appropriately swapped by software. DINK32 is written for big-endian mode and the sample code referred to in this appendix performs the appropriate byte swapping.

L.2.1 Embedded Utilities Memory Block (EUMB)

The EUMB is a block of local and PCI memory space allocated to the control and status registers of the embedded utilities. The embedded utilities of the MPC8240 are the Messaging Unit (I_2O), DMA controller, EPIC, I^2C, and ATU. The local memory map location of the EUMB is controlled by the embedded utilities memory block base address register (EUMBBAR). The PCI bus memory map location of the EUMB is controlled by

the peripheral control and status registers base address register (PCSRBAR). Since EPIC is only accessible from local memory, only the EUMBBAR is of concern for this appendix.

Please refer to the following sections in the MPC8420 User's Manual:
Section 4.4 Embedded Utilities Memory Block
Section 5.5 Embedded Utilities Memory Block Base Address Register
Section 5.1 Configuration Register Access

L.2.2 EPIC Register Summary

The EPIC register map occupies a 256 Kilobyte range of the EUMB. All EPIC registers are 32 bits wide and reside on 128 bit address boundaries. The EPIC registers are divided into four distinct areas whose address offsets are based on the EUMB location in local memory controlled by the value in the EUMBBAR configuration register.

The EPIC address offset map areas:

- 0x4_1000 - 0x4_10F0: Global EPIC register map
- 0x4_1100 - 0x4_FFF0: Global timer register map
- 0x5_0000 - 0x5_FFF0: Interrupt source configuration register map
- 0x6_0000 - 0x6_0FF0: Processor-related register map

Please refer to Section 12.2 in the MPC8420 User's Manual for the complete EPIC register address map table and Section 12.9 for all register definitions.

L.2.3 EPIC Modes

- Pass-through Mode

This mode provides a mechanism to support alternate interrupt controllers such as the 8259 interrupt controller architecture. Pass-through is the default mode of the EPIC unit.

- Mixed Mode

This mode supports two subsequent interrupt modes, either a serial interrupt mode (up to 16 serial interrupt sources) or a direct interrupt mode (up to 5 interrupt sources).

Refer to Sections 12.4 -12.6 in the MPC8240 User's Manual for more on EPIC modes.

L.3 Directory Structure

DINK32/drivers/epic

- epic.h: contains all EPIC register address macros and all function declarations
- epic1.c: contains all C language routines

- epic2.s: contains all Assembly language routines
- epicUtil.s: contains assembly routines to load and store to registers in the EUMB
- makefile: used by the DINK32 makefile to build this directory into a driver library
- Readme.txt: a text version of this appendix

L.4 EPIC Cross-Reference Table Structure

The following table is defined in epic1.c in order to cross reference interrupt vector numbers with the corresponding interrupt vector/priority register address and interrupt service routine address:

```
/*  Register  Address  Offset/  Vector  Description  /ISR  Addr
cross-reference table */
struct SrcVecTable SrcVecTable[MAXVEC] =
{
  { EPIC_EX_INT0_VEC_REG, "External Direct/Serial Source 0",  0x0},
  { EPIC_EX_INT1_VEC_REG, "External Direct/Serial Source 1",  0x0},
  { EPIC_EX_INT2_VEC_REG, External Direct/Serial Source 2",  0x0},
  { EPIC_EX_INT3_VEC_REG, "External Direct/Serial Source 3",  0x0},
  { EPIC_EX_INT4_VEC_REG, "External Direct/Serial Source 4",  0x0},
  { EPIC_SR_INT5_VEC_REG,  "External Serial Source 5",       0x0},
  { EPIC_SR_INT6_VEC_REG,  "External Serial Source 6",       0x0},
  { EPIC_SR_INT7_VEC_REG,  "External Serial Source 7",       0x0},
  { EPIC_SR_INT8_VEC_REG,  "External Serial Source 8",       0x0},
  { EPIC_SR_INT9_VEC_REG,  "External Serial Source 9",       0x0},
  { EPIC_SR_INT10_VEC_REG, "External Serial Source 10",      0x0},
  { EPIC_SR_INT11_VEC_REG, "External Serial Source 11",      0x0},
  { EPIC_SR_INT12_VEC_REG, "External Serial Source 12",      0x0},
  { EPIC_SR_INT13_VEC_REG, "External Serial Source 13",      0x0},
  { EPIC_SR_INT14_VEC_REG, "External Serial Source 14",      0x0},
  { EPIC_SR_INT15_VEC_REG, "External Serial Source 15",      0x0},
  { EPIC_TM0_VEC_REG,         "Global Timer Source 0",       0x0},
  { EPIC_TM1_VEC_REG,         "Global Timer Source 1",       0x0},
  { EPIC_TM2_VEC_REG,         "Global Timer Source 2",       0x0},
  { EPIC_TM3_VEC_REG,         "Global Timer Source 3",       0x0},
  { EPIC_I2C_INT_VEC_REG,  "Internal I2C Source",            0x0},
  { EPIC_DMA0_INT_VEC_REG, "Internal DMA0 Source",           0x0},
  { EPIC_DMA1_INT_VEC_REG, "Internal DMA1 Source",           0x0},
  { EPIC_MSG_INT_VEC_REG,  "Internal Message Source",        0x0}
};

Each of the 24 entries conforms to the following:

{      "vector/priority register address offset",
"text description",
"Interrupt Service Routine address"                    }.
```

The first column of the structure contains the macro for each of the 24 interrupt

vector/priority register address offsets in EPIC. The middle column is the text description of the interrupt vector, and the last column is the address of the registered interrupt service routine (ISR) for each interrupt vector. Currently the structure is initialized such that each vector ISR address is 0x0. This can be modified such that each defaults to a "catch all ISR" address instead of 0x0. As each interrupt vector is set up, an ISR must be registered with EPIC via the epicISRConnect() routine in the epic1.c source file. This routine takes the ISR function name and stores the address of that function in the ISR Address structure location corresponding to the interrupt vector number. Although each interrupt's vector/priority register allows the vector number to range from 0-255, this structure limits the vector number range from 0-23. So as each interrupt's vector/priority register is set up, the 8-bit vector field value must match the vector number location in the structure.

L.5 EPIC Sample Routines

The EPIC sample routines are contained in the epic1.c and epic2.s files. All C language routines are in epic1.c and all assembly language routines are in epic2.s. These routines, along with the structure described in L.4, “EPIC Cross-Reference Table Structure", can be used as sample code for systems using the MPC8240 EPIC Unit. L.6, “EPIC Commands in DINK32" describes how these routines are used by DINK32.

L.5.1 Low Level Routines

The following routines are in the epic2.s source file:

- External Interrupt Control Routines:
  - CoreExtIntEnable(): enables external interrupts by setting the MSR[EE] bit
  - CoreExtIntDisable(): disables external interrupts by clearing the MSR[EE] bit
- Low Level Exception Handler:
  - epic_exception():
    Save the current (interrupted) programming model/state
    Calls epicISR() to service the interrupt
    Restore the programming model/state and
    RFI back to interrupted process

L.5.2 High Level Routines

The following routines are in the epic1.c source file:

L.5.2.1 EPIC Initialization Routines:

epicInit(): initialize the EPIC Unit by:

- Setting the reset bit in the Global Configuration Register which will:
  - Disables all interrupts

  - — Clears all pending and in-service interrupts
  - — Sets EPIC timers to base count
  - — Sets the value of the Processor Current Task Priority to the highest priority (0xF) thus disabling interrupt delivery to the processor
  - — Reset spurious vector to 0xFF
  - — Default to pass-through mode
- Sets the EPIC operation mode to Mixed Mode (vs. Pass Through or 8259 compatible mode)
  - — If IRQType (input) is Direct IRQs:
  - — IRQType is written to the SIE bit of the EPIC Interrupt Configuration Register (ICR)
  - — clkRatio is ignored
  - — If IRQType is Serial IRQs: (Note: not supported in DINK32)
  - — both IRQType and clkRatio will be written to the ICR register

epicCurTaskPrioSet(): Change the current task priority value

epicIntISRConnect(): Register an ISR with the EPIC unit cross-reference table

L.5.2.2 High Level Exception Handler:

epicISR(): this routine is a catch all for all EPIC related interrupts:

- perform IACK (interrupt acknowledge) to get the vector number
- check if the vector number is a spurious vector
- cross-reference vector ISR (interrupt service routine) from table
- call the vector ISR
- perform EOI (end of interrupt) for the interrupt vector

L.5.2.3 Direct/Serial Register Control Routines:

epicIntEnable(): enable an interrupt source

epicIntDisable(): disable and interrupt source

epicIntSourceConfig(): configure and interrupt source

L.5.2.4 Global Timer Register Control Routines:

epicTmBaseSet(): set the base count value for a timer

epicTmBaseGet(): get the base count value for a timer

epicTmCountGet(): get the current counter value for a timer

epicTmInhibit(): inhibit counting for a timer

epicTmEnable(): enable counting for a timer

L.6 EPIC Commands in DINK32

The following commands are typed from the DINK32 command line to control the EPIC unit.

- help dev epic - Display usage of EPIC commands
- dev epic - Display content and addresses of EPIC registers, and current task priority
- dev epic in - Initialize the EPIC unit (this calls the epicInit() routine)
- dev epic ta [Number]- Change the Processor Task priority register
- dev epic en [Vector(0-23)] - Enable a particular interrupt vector
- dev epic dis [Vector(0-23)] - Disable a particular interrupt vector
- dev epic con [Vector(0-23)] - Print content of a Source Vector/Priority register
- dev epic con [Vector(0-23) Polarity(0|1) Sense(0|1) Priority (0-15)]
  - Program the Source Vector/Priority register

Example:

dev epic in - Initialize EPIC unit

dev epic en 1 - Enable interrupt vector 1

dev epic ta 10 - Set the Processor Task priority register to 10

dev epic dis 5 - Disable interrupt vector 5

dev epic con 2- Print the configuration of Interrupt vector 2

dev epic con 7 1 0 5- Configure the source Vector/Priority

register of vector 7 to have the following properties:

Polarity = 1

Sense = 0

Priority = 5

L.7 EPIC Unit Startup

When the MPC8240 comes up running DINK32, the EUMBBAR is configured such that

the EUMB is located at an offset of 0xFC00_0000 from local memory. The EPIC unit is untouched by the DINK32 initialization routines and is left in its default state of Pass-Through mode. External interrupts are also left untouched and left in the default state of disabled. The following list shows the necessary initialization steps and routine calls needed to utilize the EPIC unit:

- epicInit()
- For each interrupt vector to be used:
  - epicSourceConfig()
  - epicISRConnect()
- For each interrupt vector to be used:
  - epicIntEnable()
- epicCurTaskPrioSet()
- CoreExtIntEnable()

L.8 External Interrupt Exception Path in DINK32

The path of an external interrupt exception in DINK32 begins at the 0x500 interrupt exception vector. All DINK32 exception vector locations are set up in the same manner which is to save the exception type and pass the exception handling to a catch all exception handler. This handler is called `handle_ex` and is located in the except2.s DINK32 source file.

In the `handle_ex` handler a check is performed to see if the exception was a 0x500 and if DINK32 is running on an MPC8240. If the two conditions are true, the exception handling is passed to the EPIC low level interrupt handler, `epic_exception` located in the epic2.s source file. `Epic_exception:` handles any necessary context switching and saving of state before calling the EPIC high level interrupt handler, epicISR() located in the epic1.c source file.

Note: Currently, `epic_exception` first checks the mode of the EPIC unit. If in pass-through mode, an error message is printed stating that the EPIC unit is in pass-through mode and must be initialized.

`EpicISR()` acknowledges the interrupt by calling the `epicIACK()` which returns the vector number of the interrupting vector source. This vector number is then compared to the spurious vector value located in the EPIC Spurious Vector Register. If the interrupting vector is a spurious vector the interrupt is ignored and state is restored to the interrupted process. If the interrupting vector is a valid interrupt, then the vector number is used to reference the vector ISR from the cross-reference table. The vector ISR is then called to service the particular interrupt. Once the ISR completes and returns, an end-of-interrupt is issued by calling `epicEOI()`. Control then returns to `epic_exception`.

`Epic_exception` finishes by restoring state and performs an RFI (return from interrupt) back to the interrupted process.

L.9 Example Usage on Sandpoint Reference Platform

The EPIC driver source code currently defaults to a debug state. This state is controlled by the `-DEPICDBG` compiler directive in the makefile located in the EPIC source directory. The compiler directive allows the driver code to be much more verbose and informative when exercising the EPIC units features in the debug state. Demonstration code is also inserted in this debug state. The demo code is inserted into the `epicInit()` routine and allows for an interactive demonstration of external interrupts. The external interrupts demonstrated are IRQ lines 1 and 2, and Global Timers 0 and 1.

L.9.1 L.9.1 Sandpoint Reference Platform

The Sandpoint Reference Platform provides a means to test external interrupts via two slide switches (S5 and S6) located on the mother board. Although these switches can be manipulated to demo the EPIC unit, this is not the intended function of the switches. The intended usage of these switches is described in the document titled, " Sandpoint PPMC Processor PCI Mezzanine Card Host Board Technical Summary".

Switch S5 manipulates a 5V signal that originates from the interrupt output line of the Winbond southbridge chip in the center of mother board. With S5 slid to the left, a 5V signal is passed on, with S5 slid right, a 0V signal is passed on. The EPIC IRQ0-4 interrupt lines can be configured to be active-low or active-high triggered.

Switch S6 specifies to which IRQ line (IRQ1 or IRQ 2) the interrupt signal from S5 is passed. With the S6 slid right, IRQ1 is selected. With S6 slid left, IRQ2 is selected.

L.9.2 Demo Code Snippet

The following code is included in the epicInit() routine when compiled with the `-DEPICDBG` compiler directive.

```
/* EPIC test code */
#ifdef EPICDBG

/* The following test code enables is specific for the Sandpoint
System.
Steps:
Configure interrupts for IRQ1 and IRQ2.
Set base counts for timer0 and timer1. Timer0 will interrupt twice
for every one timer1 interrupt.
Enable interrupts for timer0 and timer1.
Enable interrupts for IRQ1 and IRQ2.
Lower the current task priority.
Enables external interrupts. */
```

```
/* set int 1 to active low, edge-sensitive, priority 10 */
       printf("Configure Int 1\n");
       tmp = epicIntSourceConfig(1,0,0,10);
       epicISRConnect(1,IRQ1ISR);
/* set int 2 to active low, edge-sensitive, priority 10 */
       printf("Configure Int 2\n");
       tmp = epicIntSourceConfig(2,0,0,10);
       epicISRConnect(2,IRQ2ISR);
/* set timer 0 */
       printf("setting timer 0 base count to 0x1000000\n");
       sysEUMBBARWrite(EPIC_TM0_BASE_COUNT_REG, 0x01000000);
       epicISRConnect(16,Timer0ISR);
/* set timer 1 */
       printf("setting timer 1 base count to 0x2000000\n");
       sysEUMBBARWrite(EPIC_TM1_BASE_COUNT_REG, 0x02000000);
       epicISRConnect(17,Timer1ISR);
/* set priority and vector# and clear mask for timer 0 */
       printf("configuring timer 0\n");
       sysEUMBBARWrite(EPIC_TM0_VEC_REG, 0x000a0010);
/* set priority and vector# and clear mask for timer 1 */
       printf("configuring timer 1\n");
       sysEUMBBARWrite(EPIC_TM1_VEC_REG, 0x000a0011);
/* enable interrupt vector 1 */
       printf("Enable Int 1\n");
       epicIntEnable(1);
/* enable interrupt vector 2 */
       printf("Enable Int 2\n");
       epicIntEnable(2);
/* lower current task priority */
       printf("Lower Current Task Priority\n");
       epicCurTaskPrioSet(5);
/* enable external interrupts */
       printf("Enable External Interrupts in MSR\n");
       CoreExtIntEnable();
#endif /* EPICDBG */
```

L.9.3 Running the Interactive Demo

The interactive demo requires that DINK32 is running on a Sandpoint system with an MPC8240 PMC module. From the DINK32 command line, initialize the EPIC unit by typing the EPIC initialization command. DINK32 will respond with initialization messages and then immediately start taking the timer interrupts. The user may now also manipulate the S5 and S6 switches to trigger interrupts on the IRQ1 and IRQ2 lines. Of course while all these external interrupts are being handled, DINK32 continues to run and will accept user input at the command line, while simultaneously writing status to the terminal.

```
DINK32_KAHLUA >>dev epic in
Initialize epic
Configure Int 1
In epicISRConnect(): Vector: 1 -> ISRAddr: 32d54
Configure Int 2
```

```
In epicISRConnect(): Vector: 2 -> ISRAddr: 32d94
setting timer 0 base count to 0x1000000
In epicISRConnect(): Vector: 16 -> ISRAddr: 32dd4
setting timer 1 base count to 0x2000000
In epicISRConnect(): Vector: 17 -> ISRAddr: 32e14
configuring timer 0
configuring timer 1
Enable Int 1
Enable Int 2
Lower Current Task Priority
Enable External Interrupts in MSR
DINK32_KAHLUA >>In epicISR() for vector#: 16
Interrupt Service Routine for Timer 0
In epicISR() for vector#: 17
Interrupt Service Routine for Timer 1
In epicISR() for vector#: 16
Interrupt Service Routine for Timer 0
In epicISR() for vector#: 16
Interrupt Service Routine for Timer 0
In epicISR() for vector#: 17
Interrupt Service Routine for Timer 1
In epicISR() for vector#: 16
Interrupt Service Routine for Timer 0
In epicISR() for vector#: 16
In epicISR() for vector#: 1
Interrupt Service Routine for interrupt 1
In epicISR() for vector#: 1
Interrupt Service Routine for interrupt 1
In epicISR() for vector#: 16
Interrupt Service Routine for Timer 0
In epicISR() for vector#: 17
Interrupt Service Routine for Timer 1
In epicISR() for vector#: 1
In epicISR() for vector#: 2
Interrupt Service Routine for interrupt 2
In epicISR() for vector#: 2
Interrupt Service Routine for interrupt 2
In epicISR() for vector#: 16
Interrupt Service Routine for Timer 0
In epicISR() for vector#: 16
```

The user can show that DINK32 can still respond to user is input by manually disabling the timer interrupts. To disable the timers, access their vector/priority registers located in local memory by typing the following:

/*this modifies the memory location oftimer0 vector/priority register */

```
mm ffc041120 <enter>
```

/*this sets the interrupt mask bit */

```
00000080 <enter>
```

/*this modifies the memory location of timer1 vector/priority register */

```
mm ffc041160 <enter>
```

/*this sets the interrupt mask bit */

```
00000080 <enter>
```

Once the two registers are modified, neither timers should interrupt again. DINK32 should still be accepting user commands, and switches S5 and S6 can still be manipulated to generated interrupts.

L.10 Code and Documentation Updates

For the most up-to-date versions of the EPIC sample driver code and this appendix/document please visit the following URL:

http://www.mot.com/SPS/PowerPC/teksupport/faqsolutions/code/index.html