DYNAMIC ENGINEERING
150 DuBois St., Suite C Santa Cruz, CA 95060
(831) 457-8891 **Fax** (831) 457-4793
http://www.dyneng.com
sales@dyneng.com
Est. 1988

IP-ReflectiveMemory

Reflective Memory Interface
256Kx16
RJ45 or IO connector

Driver Documentation

Win32 Driver Model

Manual Revision B5
Corresponding Hardware: Revision A
10-2009-0201
FLASH revision B4 & VR1A
10-2009-0201

IpReflectiveMemory
Multi-User Shared Memory Module
IndustryPack® Module

Windows® Driver Manual

Dynamic Engineering has made every effort to ensure that this manual is accurate and complete. Still, the company reserves the right to make improvements or changes in the product described in this document at any time and without notice. Furthermore, Dynamic Engineering assumes no liability arising out of the application or use of the device described herein.

The electronic equipment described herein generates, uses, and can radiate radio frequency energy. Operation of this equipment in a residential area is likely to cause radio interference, in which case the user, at his own expense, will be required to take whatever measures may be required to correct the interference.

Dynamic Engineering's products are not authorized for use as critical components in life support devices or systems without the express written approval of the president of Dynamic Engineering.

This product has been designed to operate with IP Module carriers and compatible user-provided equipment. Connection of incompatible hardware is likely to cause serious damage.

Revised 7/23/2010

Table of Contents

Introduction

The IpReflectiveMemory driver is a Win32 driver model (WDM) device driver for the IP-ReflectiveMemory IP Module from Dynamic Engineering.

The IpReflectiveMemory driver package has three parts. The driver is installed into the Windows® OS, the test executable and the User Application “Userap” executable.

The driver and test are delivered as installed or executable items to be used directly or indirectly by the user. The Userap code is delivered in source form [C] and is for the purpose of providing a reference to using the driver.

The “test” executable allows the user to use the driver in script form from a DOS window. Each driver call can be accessed, parameters set and returned. Normally not needed or used by the integrator, but a very handy tool in certain circumstances. The test executable has a “help” menu to explain the calls, parameters and returned information.

UserAp is a stand-alone code set with a simple, and powerful menu plus a series of “tests” that can be run on the installed hardware. Each of the tests execute calls to the driver, pass parameters and structures, and get results back. With the sequence of calls demonstrated, the functions of the hardware are utilized for loop-back testing. The software is used for manufacturing test at Dynamic Engineering. For example most Dynamic Engineering PCI based designs support DMA. DMA is demonstrated with the memory based loop-back tests. The tests can be ported and modified to fit your requirements.

The test software can be ported to your application to provide a running start. It is recommended to port the switch and status tests to your application to get started. The tests are simple and will quickly demonstrate the end-to-end operation of your application making calls to the driver and interacting with the hardware.

The menu allows the user to add tests, to run sequences of tests, to run until a failure occurs and stop or to continue, to program a set number of loops to execute and more. The user can add tests to the provided test suite to try out application ideas before committing to your system configuration. In many cases the test configuration will allow faster debugging in a more controlled environment before integrating with the rest of the system.

The hardware manual defines the pinout, the bitmaps and detailed configurations for each feature of the design. The driver handles all aspects of interacting with the hardware. For added explanations about what some of the driver functions do, please refer to the hardware manual.

We strive to make a useable product, and while we can guarantee operation we can't foresee all concepts for client implementation. If you have suggestions for extended features, special calls for particular set-ups or whatever please share them with us, [engineering@dyneng.com] and we will consider and in many cases add them.

IpReflectiveMemory has a Spartan2 Xilinx FPGA to implement the IP Interface, FIFO's protocol control and status for the IO. The main feature of the design is the memory array. The Hardware automatically clears the RAM and establishes the network. The main feature of the driver is to communicate with the RAM array. Writing to the RAM will automatically update the rest of the networked memory. Reading will retrieve the current value stored into memory. The driver also provides the ability to change the hardware operation to use features other than the defaults.

When the IpReflectiveMemory board is recognized by the IP Carrier Driver, the carrier driver will start the IpReflectiveMemory driver which will create a device object for the board. If more than one is found additional copies of the driver are loaded. The carrier driver will load the info storage register on the IP-ReflectiveMemory with the carrier switch setting and the slot number of the IP-ReflectiveMemory device. From within the IpReflectiveMemory driver the user can access the switch and slot information to determine the specific device being accessed when more than one are installed.

The reference software application has a loop to check for devices. The number of devices found, the locations, and PLL information are printed out at the top of the menu.

IO Control calls (IOCTLs) are used to configure the board and read status. Read and Write calls are used to move data in and out of the device.

Note

This documentation will provide information about all calls made to the drivers, and how the drivers interact with the device for each of these calls. For more detailed information on the hardware implementation, refer to the IpReflectiveMemory user manual (also referred to as the hardware manual).

Driver Installation

There are several files provided in each driver package. These files include driver: IpReflectiveMemory.sys, DDIpReflectiveMemory.h, IpReflectiveMemoryGUID.h, IpCarrier.inf , IpDevice.inf : Part of Carrier Driver Package
Driver Test: IpReflectiveMemoryTest.exe
Userap: User Application source files.

IpReflectiveMemoryGUID.h is a C header file that defines the device interface identifiers for the drivers. DDIpReflectiveMemory.h is a C header file that defines the Application Program Interface (API) to the drivers. These files are required at compile time by any application that wishes to interface with the drivers, but they are not needed for driver installation. The files are included with the Userap fileset.

IpReflectiveMemoryTest.exe is a sample Win32 console applications that makes calls into the driver to test each driver call without actually writing any application code. Not required during driver installation either. Please refer to the User Application software package as a reference for using the driver.

To run IpReflectiveMemoryTest, open a command prompt console window and type ***IpReflectiveMemoryTest -d0 -?*** to display a list of commands (the IpReflectiveMemoryTest.exe file must be in the directory that the window is referencing). The commands are all of the form ***IpReflectiveMemoryTest -dn -im*** where ***n*** and ***m*** are the device number and IpReflectiveMemory driver ioctl number respectively.

This test application is intended to test the proper functioning of each driver call, **not** for normal operation. Many integration efforts will never need the debugger capability that the test menu represents. The test capability will allow the designer to access the card without any other software in the way to make sure that the system can “see” the card and to do basic card manipulations.

Windows 2000 Installation

IP Modules are updated more frequently than the IP Module Carriers are. The Carrier uses two INF files which come with the Carrier driver to install the carrier driver and determine what IP modules are present. Install the carrier driver as described in the carrier installation documentation. Replace the INF Files with the updated ones included with the IpReflectiveMemory driver package. Use the device manager to uninstall and reinstall the carrier driver if it was previously installed.

Please note that Windows uses the INF file after copying to a working file within the INF directory. The INF files are renamed OEMxx.inf and the copy with the same name but a new extension. Please search for and delete both files for both the carrier (IpCarrier.inf) and the Ip Modules (IpDevice.inf) then reinstall to make sure the new INF data is being used.

Copy IpReflectiveMemory.sys and IpDevice.inf to a floppy disk, or CD if preferred. In some cases the files can be accessed over a network or from local HDD. Substitute the network address for the floppy instructions to proceed with an over the network installation.

With the hardware installed, power-on the PCI host computer and wait for the ***Found New Hardware Wizard*** dialogue window to appear. Please see steps listed above if new HW is not found or if unknown IP device is found.
_ Select ***Next***.
_ Select ***Search for a suitable driver for my device.***
_ Select ***Next***.
_ Insert the disk prepared above in the desired drive.
_ Select the appropriate drive e.g. ***Floppy disk drives***.
_ Select ***Next***.
_ The wizard should find the IpDevice.inf file.
_ Select ***Next***.
_ Select ***Finish*** to close the ***Found New Hardware Wizard***.

Windows XP Installation

Copy IpReflectiveMemory.sys and IpDevice.inf to a floppy disk, or CD if preferred. In some cases the files can be accessed over a network or from local HDD. Substitute the network address for the floppy instructions to proceed with an over the network installation.

With the hardware installed, power-on the PCI host computer and wait for the ***Found New Hardware Wizard*** dialogue window to appear.
_ Insert the disk prepared above in the desired drive.
_ Select ***No when asked to connect to Windows Update***.

_ Select ***Next***.
_ Select ***Install the software automatically.***
_ Select ***Next***.
_ Select ***Finish*** to close the ***Found New Hardware Wizard***.

Driver Startup

Once the drivers have been installed they will start automatically when the system recognizes the hardware.

Handles can be opened to a specific board by using the CreateFile() function call and passing in the device names obtained from the system.

The interfaces to the devices are identified using globally unique identifiers (GUIDs), which are defined in IpReflectiveMemoryGUID.h.

The User Application software contains a file called “main.c”. Main has the initialization needed to get the handle to the assets of the installed Ip-ReflectiveMemory device.

The main file provided is designed to work with our test menu and includes user interaction steps to allow the user to select which board is being tested in a multiple board environment. The integrator can hardcode for single board systems or use an automatic loop to operate in multiple board systems without using user interaction. For multiple user systems it is suggested that the board number is associated with a switch setting so the calls can be associated with a particular board from a physical point of view.

IO Controls

The drivers use IO Control calls (IOCTLs) to configure the device. IOCTLs refer to a single Device Object, which controls a single board or I/O channel. IOCTLs are called using the Win32 function DeviceIoControl() (see below), and passing in the handle to the device opened with CreateFile() (see above). IOCTLs generally have input parameters, output parameters, or both. Often a custom structure is used.

```
BOOL DeviceIoControl(
  HANDLE        hDevice,          // Handle opened with
CreateFile()
  DWORD         dwIoControlCode, // Control code defined in API
header file
  LPVOID        lpInBuffer,       // Pointer to input parameter
  DWORD         nInBufferSize,    // Size of input parameter
  LPVOID        lpOutBuffer,      // Pointer to output parameter
  DWORD         nOutBufferSize,   // Size of output parameter
  LPDWORD       lpBytesReturned, // Pointer to return length
parameter
  LPOVERLAPPED lpOverlapped,     // Optional pointer to
overlapped structure
);                                 // used for asynchronous I/O
```

The IOCTLs defined for the IpreflectiveMemory driver are described below:

IOCTL_IPREFMEM_GET_INFO

Function: Return the Instance Number, Switch value, PLL device ID, Xilinx rev and Current Driver Version

Input: None

Output: DRIVER_DEVICE_INFO : Structure

Notes: CarrierSwitch is the configuration of the carrier dip-switch that has been set by the User (see the board silk screen for bit position and polarity). CarrierSlotNum is the numerical equivalent to the carrier slot IpReflectiveMemory is installed into. The PLL ID is the device address of the PLL device. This value, which is set at the factory, is usually 0x69 but may also be 0x6A. See DDIpReflectiveMemory.h for the definition of DRIVER_DEVICE_INFO.

IOCTL_IPREFMEM_LOAD_PLL_DATA

Function: Loads the internal registers of the PLL.

Input: IPREFMEM_PLL_DATA structure

Output: None

Notes:

IOCTL_IPREFMEM_READ_PLL_DATA
Function: Returns the contents of the PLL's internal registers
Input: None
Output: IPREFMEM_PLL_DATA structure
Notes: The register data is output in the IPREFMEM_PLL_DATA structure in an array of 40 bytes.

IOCTL_IPREFMEM_SET_IP_CONTROL
Function: Write to channelized Slot control register on carrier for IP.
Input: ULONG
Output: none
Notes: See bit map names in DDIpReflectiveMemory.h

IOCTL_IPREFMEM_GET_IP_CONTROL
Function: Read Slot control register on carrier
Input: none
Output: ULONG
Notes: See bit map definitions in DDIpReflectiveMemory.h

IOCTL_IPREFMEM_SET_BASE_CONFIG
Function: Write to Base Control Register - general access to base control register of card, use with bit definitions Structure used to pass data.
Input: IPREFMEM_BASE_CONFIG
Output: none
Notes: Use for general purpose – bit mapped access to the base control register. The bits associated with the PLL are read only for this call. Use the PLL specific IOCTL's to control the PLL. This it to avoid putting the PLL into an unknown state when accessing other parts of the register.

IOCTL_IPREFMEM_GET_BASE_CONFIG
Function: Read from Base Control Register - general access from base control register of card, use with bit definitions
Input: none
Output: IPREFMEM_BASE_CONFIG
Notes: Use for general purpose – bit mapped access to the base control register.

IOCTL_IPREFMEM_GET_NODE_SWITCH
Function: Read DIPSWITCH's located on IP. Both Net Address and Option Control
Input: none
Output: USHORT
Notes: 7-0 = NodeAddress Switch 15-8 = Option Switch

IOCTL_IPREFMEM_GET_MESSAGE_COUNT
Function: Read Roll Over Count of messages received
Input: none
Output: USHORT
Notes: 16 bit counter advances when valid messages are received. Rolls over at end count. Can be used for message traffic indication and network operation.

IOCTL_IPREFMEM_REGISTER_EVENT

Function: Registers an event to be signaled when an interrupt occurs.
Input: Handle to the Event object
Output: None
Notes: The caller creates an event with CreateEvent() and supplies the handle returned from that call as the input to this IOCTL. The driver then obtains a system pointer to the event and signals the event when a user interrupt is serviced. The user interrupt service routine waits on this event, allowing it to respond to the interrupt.

IOCTL_IPREFMEM_ENABLE_INTERRUPT

Function: Enables the channel Master Interrupt.
Input: None
Output: None
Notes: This command must be run to allow the board to respond to user interrupts. The master interrupt enable is disabled in the driver interrupt service routine when a user interrupt is serviced. Therefore this command must be run after each interrupt occurs to re-enable it.

IOCTL_IPREFMEM_DISABLE_INTERRUPT

Function: Disables the channel Master Interrupt.
Input: None
Output: None
Notes: This call is used when user interrupt processing is no longer desired.

IOCTL_IPREFMEM_FORCE_INTERRUPT

Function: Causes a system interrupt to occur.
Input: None
Output: None
Notes: Causes an interrupt to be asserted on the PCI bus as long as the channel master interrupt is enabled. This IOCTL is used for development, to test interrupt processing. Board level master interrupt also needs to be set.

IOCTL_IPREFMEM_SET_VECTOR
Function: Writes an 8 bit value to the interrupt vector register
Input: UCHAR
Output: none
Notes: Required when used in non auto-vectored systems

IOCTL_IPREFMEM_GET_VECTOR
Function: Returns a stored vector value
Input: None
Output: UCHAR
Notes:

IOCTL_IPREFMEM_GET_ISR_STATUS
Function: Returns the interrupt status, and vector read in the ISR from the last user interrupt.
Input: None
Output: IPREFMEM_INT_STAT
Notes: Returns the interrupt status that was read in the interrupt service routine of the last interrupt caused by one of the enabled channel interrupts.

IOCTL_IPREFMEM_SET_MEM_DATA
Function: write to Memory array
Input: IP_MEMORY_WRITE
Output: None
Notes: See DDIpReflectiveMemory.h for structure. Pass Address and Data in

IOCTL_IPREFMEM_GET_MEM_DATA
Function: Read from Memory Array
Input: Address ULONG
Output: USHORT Data
Notes:

IOCTL_IPREFMEM_SET_NET_ADD_MATCH
Function: Set the address to match for incoming messages
Input: USHORT
Output: None
Notes: 7-0 = node address to compare against 15-8 spare. If enabled an interrupt can be generated when a received Memory update message has a matching node identifier.

IOCTL_IPREFMEM_GET_NET_ADD_MATCH
Function: Read from Address Match register
Input: None
Output: USHORT
Notes: full register is returned including spare bits

IOCTL_IPREFMEM_SET_STATUS
Function: write to status register to clear sticky bits
Input: USHORT
Output: None
Notes: See DDIpReflectiveMemory.h for bit definitions. Some status bits are held until explicitly cleared by writing back with those positions set. Error status in particular.

IOCTL_IPREFMEM_GET_STATUS
Function: Read from Status Register
Input: None
Output: USHORT
Notes: See DDIpReflectiveMemory.h for bit definitions. No side affects – status can be read multiple times. Clearing bits take explict write with Set Status IOCTL.

IOCTL_IPREFMEM_SET_LED_CONFIG
Function: Write to LED Control Register - Structure used to pass data.
Input: IPREFMEM_LED_CONFIG
Output: none
Notes: Select Forced Flash or activity based LED action per LED. Enable and Disable spare LED's (0,1)

IOCTL_IPREFMEM_GET_LED_CONFIG
Function: Read from LED Control Register – Structure returned with status LED control options.
Input: none
Output: IPREFMEM_LED_CONFIG
Notes: Determine current configuration of Select Forced Flash or activity based LED action per LED. Enable and Disable spare LED's (0,1)

IOCTL_IPREFMEM_CLR_BAD_MESS_CNT
Function: Write to clearing function for bad message counter
Input: USHORT
Output: none
Notes: Bad Message Counter is incremented whenever a bad message is detected – the error LED will flash and the counter will advance. 16 bit counter reset to "0000". Counts 0->FFFF->0.

IOCTL_IPREFMEM_GET_BAD_MESS_CNT
Function: Read from bad message counter
Input: none
Output: USHORT
Notes: Bad Message Counter is incremented whenever a bad message is detected – the error LED will flash and the counter will advance. 16 bit counter reset to “0000”. Counts 0->FFFF->0. Reading from this port will return the current count of bad messages.

Warranty and Repair

Dynamic Engineering warrants this product to be free from defects under normal use and service and in its original, unmodified condition, for a period of one year from the time of purchase. If the product is found to be defective within the terms of this warranty, Dynamic Engineering's sole responsibility shall be to repair, or at Dynamic Engineering's sole option to replace, the defective product.

Dynamic Engineering's warranty of and liability for defective products is limited to that set forth herein. Dynamic Engineering disclaims and excludes all other product warranties and product liability, expressed or implied, including but not limited to any implied warranties of merchandisability or fitness for a particular purpose or use, liability for negligence in manufacture or shipment of product, liability for injury to persons or property, or for any incidental or consequential damages.

Dynamic Engineering's products are not authorized for use as critical components in life support devices or systems without the express written approval of the president of Dynamic Engineering.

Service Policy

Before returning a product for repair, verify as well as possible that the driver is at fault. The driver has gone through extensive testing and in most cases it will be “cockpit error” rather than an error with the driver. When you are sure or at least willing to pay to have someone help then call the Customer Service Department and arrange to speak with an engineer. We will work with you to determine the cause of the issue. If the issue is one of a defective driver we will correct the problem and provide an updated module(s) to you [no cost]. If the issue is of the customer’s making [anything that is not the driver] the engineering time will be invoiced to the customer. Pre-approval may be required in some cases depending on the customer’s invoicing policy.

Out of Warranty Repairs

Out of warranty support will be billed. The current minimum repair charge is $125. An open PO will be required.

For Service Contact:

Customer Service Department
Dynamic Engineering
150 DuBois Street, Suite C
Santa Cruz, CA 95060
831-457-8891
831-457-4793 Fax

support@dyneng.com

Appendix

Reference copy of structures for evaluation

The following structures shown are available in the DDIpReflectiveMemory.h files included with the driver. The structures are included here for your evaluation when considering the driver package. The electronic versions included with the driver should be used with your project. The names track the register bit definitions. For details about particular signals please refer to the HW manual.

```
#define PLL_MESSAGE1_SIZE       16
#define PLL_MESSAGE2_SIZE       24
#define PLL_MESSAGE_SIZE        (PLL_MESSAGE1_SIZE + PLL_MESSAGE2_SIZE)

typedef struct _DRIVER_IP_DEVICE_INFO
{
  ULONG    DriverVersion;
  ULONG    InstanceNumber;
  UCHAR    CarrierSwitch;    // 0..0xFF
  UCHAR    CarrierSlotNum;   // 0..4 -> 16-bit slots A, B, C, D or E
  UCHAR    PllDeviceId;
} DRIVER_IP_DEVICE_INFO, *PDRIVER_IP_DEVICE_INFO;



typedef enum _FORCE_INT {SET, CLEAR} FORCE_INT, *PFORCE_INT;
// choices for Force Interrupt bit, HW default is CLEAR

typedef enum _RAM_CLK_SEL {X2OSC, PLLA} RAM_CLK_SEL, *PRAM_CLK_SEL;
// choices for Ram Clock Selection bit, HW default is OSC X2

typedef enum _CLK_OUT_SEL {X1OSC, PLLB} CLK_OUT_SEL, *PCLK_OUT_SEL;
// choices for IO Speed bit, HW default is OSC X1

typedef enum _NODE_MATCH_INT_EN {NODE_MATCH_INT_DISABLED,
NODE_MATCH_INT_ENABLED} NODE_MATCH_INT_EN,
*PNODE_MATCH_INT_EN;
// choices for Node Match Interrupt, default is disabled

typedef enum _NET_RAM_DISABLE {EN_NR, DIS_NR} NET_RAM_DISABLE,
*PNET_RAM_DISABLE;
// choices for NetWork RAM disable bit, HW default is enabled
```

```
typedef enum _NET_IP_DISABLE {EN_IP, DIS_IP} NET_IP_DISABLE,
*PNET_IP_DISABLE;
// choices for NetWork IP disable bit, HW default is enabled

typedef enum _NET_PT_DISABLE {EN_PT, DIS_PT} NET_PT_DISABLE,
*PNET_PT_DISABLE;
// choices for NetWork PT disable bit, HW default is enabled

typedef enum _NET_ID_DISABLE {EN_ID, DIS_ID} NET_ID_DISABLE,
*PNET_ID_DISABLE;
// choices for NetWork ID disable bit, HW default is enabled

typedef enum _PLL_EN {EN_PLL_EN, DIS_PLL_EN} PLL_EN, *PPLL_EN;
// choices for PLL Enable bit, HW default is Disabled

typedef enum _PLL_SCLK {CLK_HIGH, CLK_LOW} PLL_SCLK, *PPLL_SCLK;
// choices for SCLK bit, HW default is Low

typedef enum _PLL_S2 {S2_HIGH, S2_LOW} PLL_S2, *PPLL_S2;
// choices for S2 Suspend, HW default is Low

typedef enum _PLL_SDAT {DATA_ONE, DATA_ZERO} PLL_SDAT, *PPLL_SDAT;
// choices for PLL Command Data, HW default is Zero
```

```
// Base Register Control Structure
typedef struct _IPREFMEM_BASE_CONFIG
{
      FORCE_INT                 ForceInt; // Set to cause an interrupt to the host
      RAM_CLK_SEL               RamClkSel;    // 0 = 2x Osc clock, 1 = PLLA
      CLK_OUT_SEL               ClkOutSel;    // 0 = 1x Osc clock, 1 = PLLB
      NODE_MATCH_INT_EN         NodeMatchIntEn; // 0 = not Enabled, 1 = NodeMatch
                                 Interrupt Enabled, clear by write back to status or
                                 disable this bit
      BOOLEAN                    ForceWrite; // TRUE = All RAM accesses are
                                 forwarded to the network - even if they match local
                                 RAM, FALSE = standard operation
      BOOLEAN                    RamDumpDisable;  // TRUE =  Do not do RAM Dump
                                 - only affects master node, FALSE = perform RAM
                                 Dump when network reacquired
      NET_RAM_DISABLE           NetRamDisable;
// 1 = RAM is not updated from NetWork Messages, Network is still forwarded => for
                                 isolated local RAM
      NET_IP_DISABLE            NetIpDisable;
// 1 = NetWork is not updated for an IP RAM Write operation  => for local RAM carved
                                 out from network
      NET_PT_DISABLE            NetPtDisable;
// 1 = NetWork Pass Through is disabled - test purposes only
      NET_ID_DISABLE            NetIdDisable;
// 1 = Do not remove messages from this node => test only to cause Master to have to
                                 clean up messages with loop-bit set
      PLL_EN                    PllEn;      // Output Enable control for PLL, Read Only
      PLL_SCLK                  PllSClk; // Output to PLL, Command Clock, Read Only
      PLL_S2                    PllS2;      // S2/Suspend signal to PLL, Read Only
      PLL_SDAT                  PllSDat; // Command data to / from PLL, Read Only
} IPREFMEM_BASE_CONFIG, *PIPREFMEM_BASE_CONFIG;


// Interrupt status and vector
typedef struct _IPREFMEM_INT_STAT
{
  USHORT   InterruptStatus;
  USHORT   InterruptVector;
} IPREFMEM_INT_STAT, *PIPREFMEM_INT_STAT;
```

```
// memory write structure
typedef struct _IP_MEMORY_WRITE
{
  ULONG   MemoryOffset;
  USHORT  MemoryData;
} IP_MEMORY_WRITE, *PIP_MEMORY_WRITE;


typedef struct _IPREFMEM_PLL_DATA
{
  UCHAR    Data[PLL_MESSAGE_SIZE];
} IPREFMEM_PLL_DATA, *PIPREFMEM_PLL_DATA;



typedef struct _IPREFMEM_LED_CONFIG
{
        BOOLEAN         LedMasterStFlEn;        //clear for forced flash mode, set
                        for activity based
        BOOLEAN         LedMasterEnFlEn;        //clear for forced flash mode, set
                        for activity based
        BOOLEAN         LedErrorFlEn;           //clear for forced flash mode, set
                        for activity based
        BOOLEAN         LedLocalStFlEn;         //clear for forced flash mode, set
                        for activity based
        BOOLEAN         LedIpMsgSentFlEn;       //clear for forced flash mode, set
                        for activity based
        BOOLEAN         LedNetMsgSentFlEn;      //clear for forced flash mode, set
                        for activity based
        BOOLEAN         LedSpare0;              //Set to illuminate Spare LED 0
        BOOLEAN         LedSpare1;              //Set to illuminate Spare LED 1
}IPREFMEM_LED_CONFIG, *PIPREFMEM_LED_CONFIG;
```