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

Ip1553,
BC, MT, RT & RTM

WDF Driver Documentation

Developed with Windows Driver Foundation Ver1.19

Manual Revision 1P1
Corresponding PCB: Revision 03
10-2006-1403
Corresponding Firmware: Revision 0401

lp1553, BC, MT, RT & RTM
WDF Device Drivers for the IP-1553
2-Channel MIL-STD 1553A/B
Interface IndustryPack® Module

Dynamic Engineering
150 DuBois, Suite C
Santa Cruz, CA 95060
(831) 457-8891
FAX: (831) 457-4793

Manual Revised March 23, 2020

Table of Contents

Introduction

The Ip1553, BC, MT, RT and RTM drivers are Windows 10 device driver for the IP-1553 Industry-pack (IP) module from Dynamic Engineering. This driver was developed with the Windows Driver Foundation version 1.19 (WDF) from Microsoft, specifically the Kernel-Mode Driver Framework (KMDF).

The IP-1553 board has a Spartan6 Xilinx FPGA to implement the Industry Pack interface and protocol control and status for two DDC channels. The 1553 devices can operate in one of four modes: Bus Controller, Monitor, Remote Terminal or Remote Terminal/Monitor mode. Respectively the BC, MT, RT or RTM drivers control the devices when they are operating in these modes. The appropriate driver is loaded automatically for the operating mode selected.

The IP-1553 driver package has two parts. The drivers and the User Application "UserApp" executable. The drivers are delivered as installed and executable items to be used directly or indirectly by the user. The UserApp code is delivered in source form [C] and is for the purpose of providing a reference to using the driver.

The UserApp 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.

The test software can be ported to your application to provide a running start. It is recommended to port the Register 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 test suite is designed to accommodate up to 5 boards. The number of boards can be expanded. See Main.c to increase the number of handles.

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.

When the IP-1553 is recognized by the system configuration utility it will start the Ip1553 driver. The Ip1553 driver enumerates the channels and creates one or two separate BC, MT, RT or RTM device objects. If two devices are installed, this allows the I/O channels to be totally independent while the base driver controls the device items that are common. 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 I/O channel device memory. When the 1553 devices are first powered-on, or after a hardware reset has been issued, the channel devices will be in Idle mode. If channel drivers have never been installed onto machine, then an IOCTL call to the Ip1553 driver followed by installation of corresponding driver in the device manager, is necessary. After the driver has been installed once, simply calling the appropriate IOCTL will change the operating mode of the channel.

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 IP-1553 user manual (also referred to as the hardware manual) and HoltIC 62203 User Guide available from – http://www.holtic.com

Driver Installation

There are several files provided in each driver package. These files include Ip1553.sys, BusCntrl.sys, Montor.sys, RemTerm.sys, RmTrmMon.sys, Ip1553Public.h, IpPublic.h, BusCntrlPublic.h, MontorPublic.h, RemTermPublic.h, RemTrmMonPublic.h, Ip1553.inf, BusCntrl.inf, Montor.inf, RemTerm.inf, RmTrmMon.inf, ip1553.cat, buscntrl.cat, montor.cat, remterm.cat, and rmtrmmon.cat.

Ip1553Public.h, BusCntrlPublic.h, MontorPublic.h, RemTermPublic.h, RemTrmMonPublic.h, and IpPublic.h are C header files that define the Application Program Interface (API) to the driver. These files are required at compile time by any application that wishes to interface with the driver, but are not needed for driver installation.

Warning: The appropriate IP carrier driver must be installed before any IP modules can be detected by the system.

Windows 10 Installation

Copy the inf files, the cat files, the sys files, and the other IP module drivers to a removable memory device or other accessible location as preferred.

With the IP hardware installed, power-on the host computer.

- Open the ***Device Manager*** from the control panel.
- Under ***Other devices*** there should be an item for each IP module installed on the IP carrier. The label for a module installed in the first slot of the first PCIe3IP carrier would read ***PcieCar0 IP Slot A****.
- Right-click on the first device and select ***Update Driver Software***.
- Insert the removable memory device prepared above if necessary.
- Select ***Browse my computer for driver software***.
- Select ***Browse*** and navigate to the memory device or other location prepared above.
- Select ***Next***. The IP1553 device driver should now be installed.
- Select ***Close*** to close the update window.
- Right-click on the remaining IP slot icons and repeat the above procedure as necessary.

***** If the [***Carrier] IP Slot [x]*** devices are not displayed, click on the ***Scan for hardware changes*** icon on the Device Manager tool-bar.

Note: When the 1553 Base driver is installed the channel(s) will be in Idle mode. In order to install the channel driver(s) the SetChanMode IOCTL must be called, followed by a manual installation in the Device Manager. After the channel driver has been installed using the Device Manage once, then a call to the IOCTL is sufficient for installation.

Driver Startup
Once the driver has been installed it will start automatically when the system recognizes the hardware.

A handle can be opened to a specific module by using the CreateFile() function call and passing in the device name obtained from the system.

See the *main.c* file provided with the user test software for an example of obtaining a device handle to a specific module. Use IOCTL_IP_1553_MODE described in the IO controls section in order to set a channel to the desired mode. See the Set1553Dev.c file for an example how this is done.

IO Controls

The driver uses 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 Ip1553 driver are described below:

IOCTL_IP_1553_GET_INFO

Function: Returns the driver and firmware revisions, module instance number and location and other information.
Input: None
Output: DRIVER_IP_DEVICE_INFO structure
Notes: This call does not access the hardware, only stored driver parameters. NewIpCntl indicates that the module's carrier has expanded slot control capabilities. See the definition of DRIVER_IP_DEVICE_INFO below.

```
typedef struct _DRIVER_IP_DEVICE_INFO {
   UCHAR    DriverRev;         // Driver revision
   UCHAR    FirmwareRev;       // Firmware major revision
   UCHAR    FirmwareRevMin;    // Firmware minor revision
   UCHAR    InstanceNum;       // Zero-based device number
   UCHAR    CarrierSwitch;     // 0..0xFF
   UCHAR    CarrierSlotNum;    // 0..7 -> IP slots A, B, C, D, E, F, G or H
   UCHAR    CarDriverRev;      // Carrier driver revision
   UCHAR    CarFirmwareRev;    // Carrier firmware major revision
   UCHAR    CarFirmwareRevMin;// Carrier firmware minor revision
   UCHAR    CarCPLDRev;        //*Used for PCIe carriers only**0xFF for others
   UCHAR    CarCPLDRevMin;     //*Used for PCIe carriers only**0xFF for others
   BOOLEAN  Ip32MCapable;      // IP capable of both 8MHz and 32MHz operation
   BOOLEAN  NewIpCntl;         // New IP slot control interface
   WCHAR    LocationString[IP_LOC_STRING_SIZE];
} DRIVER_IP_DEVICE_INFO, *PDRIVER_IP_DEVICE_INFO;
```

IOCTL_IP_1553_SET_IP_CONTROL

Function: Sets various control parameters for the IP slot the module is installed in.
Input: IP_SLOT_CONTROL structure
Output: None
Notes: Controls the IP clock speed, interrupt enables and data manipulation options for the IP slot that the board occupies. See the definition of IP_SLOT_CONTROL below. For more information refer to the IP carrier hardware manual.

```
typedef struct _IP_SLOT_CONTROL {
   BOOLEAN  Clock32Sel;
   BOOLEAN  ClockDis;
   BOOLEAN  ByteSwap;
   BOOLEAN  WordSwap;
   BOOLEAN  WrIncDis;
   BOOLEAN  RdIncDis;
   UCHAR    WrWordSel;
   UCHAR    RdWordSel;
   BOOLEAN  BsErrTmOutSel;
   BOOLEAN  ActCountEn;
} IP_SLOT_CONTROL, *PIP_SLOT_CONTROL;
```

IOCTL_IP_1553_GET_IP_STATE
Function: Returns control/status information for the IP slot the module is installed in.
Input: None
Output: IP_SLOT_STATE structure
Notes: Returns the slot control parameters set in the previous call as well as status information for the IP slot that the board occupies. See the definition of IP_SLOT_STATE below.

```
typedef struct _IP_SLOT_STATE {
   BOOLEAN  Clock32Sel;
   BOOLEAN  ClockDis;
   BOOLEAN  ByteSwap;
   BOOLEAN  WordSwap;
   BOOLEAN  WrIncDis;
   BOOLEAN  RdIncDis;
   UCHAR    WrWordSel;
   UCHAR    RdWordSel;
   BOOLEAN  BsErrTmOutSel;
   BOOLEAN  ActCountEn;
 // Slot Status
   BOOLEAN  IpInt0En;
   BOOLEAN  IpInt1En;
   BOOLEAN  IpBusErrIntEn;
   BOOLEAN  IpInt0Actv;
   BOOLEAN  IpInt1Actv;
   BOOLEAN  IpBusError;
   BOOLEAN  IpForceInt;
   BOOLEAN  WrBusError;
   BOOLEAN  RdBusError;
} IP_SLOT_STATE, *PIP_SLOT_STATE;.
```

IOCTL_IP_1553_GET_STATUS
Function: Returns the status bits in the IP_1553_STATUS register.
Input: None
Output: Register configuration (unsigned short integer)
Notes: Returns the interrupt status for the two 1553 devices.

IOCTL_IP_1553_RESET_DEV
Function: Does a hardware reset of one of the 1553 devices.
Input: 1553 channel to reset (unsigned character)
Output: None
Notes: The selected 1553 device will revert to Bus Controller mode after a hardware reset.

IOCTL_IP_1553_SET_MODE
Function: Selects the operating mode for a 1553 channel device.
Input: device channel number and mode (IP_1553_CHAN_MODE structure)
Output: None
Notes: All handles referencing the channel device must be closed before issuing this command or the device object will not be removed from the system. See the definition of IP_1553_CHAN_MODE below.

```
typedef enum _IP_1553_MODE_SEL {
   BUS_CONTROL,
   MONITOR,
   REMOTE_TERM,
   REM_TERM_MON,
   IDLE
} IP_1553_MODE_SEL, *PIP_1553_MODE_SEL;

 // Channel configuration
typedef struct _IP_1553_CHAN_MODE {
   UCHAR             Channel;
   IP_1553_MODE_SEL  Mode;
} IP_1553_CHAN_MODE, *PIP_1553_CHAN_MODE;
```

IOCTL_IP_1553_GET_MODE
Function: Returns the operating mode for the selected 1553 channel device.
Input: device channel number (unsigned character)
Output: Operating mode (IP_1553_MODE_SEL enumeration type)
Notes: See the definition of IP_1553_SEL above.

IOCTL_IP_1553_REGISTER_EVENT
Function: Registers an Event object to be signalled 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_IP_1553_FORCE_INTERRUPT
Function: Causes an IP interrupt to occur.
Input: None
Output: None
Notes: Causes an interrupt to be asserted on the IP bus. This IOCTL is used for test and development, to test interrupt processing.

IOCTL_IP_1553_SET_VECTOR
Function: Sets the value of the interrupt vector.
Input: Interrupt vector value (unsigned character)
Output: None
Notes: This value will be driven onto the low byte of the data bus in response to an INT_SEL strobe, which is used in vectored interrupt cycles. This value will also be read in the interrupt service routine and stored for future reference.

IOCTL_IP_1553_GET_VECTOR
Function: Returns the current interrupt vector value.
Input: None
Output: Interrupt vector value (unsigned character)
Notes:

IOCTL_IP_1553_ISR_STATUS
Function: Returns the interrupt status and vector read in the last ISR.
Input: None
Output: IP_1553_ISR_STAT structure
Notes: The status contains the interrupt vector and the contents of the status register read in the last ISR execution. Also, if bit 12 is set in the interrupt status, it indicates that a bus error occurred for this IP slot. See below for the definition of IP_1553_ISR_STAT.

```
 // Interrupt status and vector
typedef struct _IP_1553_ISR_STAT {
   USHORT   IntSlotStatus;
   USHORT   IntDevStatus;
   USHORT   InterruptVector;
} IP_1553_ISR_STAT, *PIP_1553_ISR_STAT;
```

IOCTL_IP_1553_LOOP_TEST
Function: Tests the external loopback path between bus A and bus B.
Input: None
Output: None
Notes: Must call the IOCTL_IP_1553_STOP_LOOP_TEST after running this IOCTLs to return device back to normal operation. It is recommended that IOCTL_IP_1553_GET_MODE is run before and IOCTOL_IP_1553_SET_MODE is run after this loop test is run.

IOCTL_IP_1553_STOP_LOOP_TEST
Function: Takes the 1553 out of test mode.
Input: None
Output: None
Notes: Should be run after IOCTL_IP_1553_LOOP_TEST to take it out of test mode.

IOCTL_IP_1553_WRITE_MEM_WORD
Function: Write a single word to the IP1553 RAM.
Input: IP_1553_MEM_WRITE
Output: None
Notes:

```
typedef struct _IP_1553_MEM_WRITE {
   UCHAR    Channel;
   USHORT   Offset;
   USHORT   Data;
} IP_1553_MEM_WRITE, *PIP_1553_MEM_WRITE;
```

IOCTL_IP_1553_READ_MEM_WORD
Function: Read a single word from the IP1553 RAM.
Input: IP_1553_MEM_READ_ADDR
Output: USHORT
Notes:

```
typedef struct _IP_1553_READ_ADDR {
   UCHAR    Channel;
   USHORT   Offset;
} IP_1553_READ_ADDR, *PIP_1553_READ_ADDR;
```

IOCTL_IP_1553_GET_NUM_CHANS
Function: Returns the number of 1553 devices (channels) installed.
Input: None
Output: UCHAR
Notes: The number of channels will be either 1 or 2.

The IOCTLs defined for the Bus Controller driver are described below:

IOCTL_BUS_CNTRL_GET_INFO
Function: Returns the bus controller driver revision, the IP-1553 device instance number and the IP-1553 device channel number.
Input: None
Output: BUS_CNTRL_DRIVER_DEVICE_INFO structure
Notes: The device number is passed to the channel devices so that the base device and channel device handles can be coordinated to reference the same physical module in the application software. See the definition of BUS_CNTRL_DRIVER_DEVICE_INFO.

```
typedef struct _BUS_CNTRL_DRIVER_DEVICE_INFO {
   ULONG    DriverRev;  // Bus controller driver rev.
   USHORT   DeviceNum;  // IP-1553 device instance number from Ip1553 driver
   UCHAR    Channel;    // IP-1553 device channel number (0 or 1)
} BUS_CNTRL_DRIVER_DEVICE_INFO, *PBUS_CNTRL_DRIVER_DEVICE_INFO;
```

IOCTL_BUS_CNTRL_SET_CONFIG
Function: Sets the channel and bus controller internal device register configuration.
Input: BUS_CNTRL_DEV_CONFIG structure
Output: None
Notes: This call controls the channel enable, power-on BIT disable and Sleep mode as well as numerous internal device register controls. See BusCntrlPublic.h for the definition of BUS_CNTRL_DEV_CONFIG.

IOCTL_BUS_CNTRL_GET_CONFIG
Function: Returns the channel and bus controller internal device configuration.
Input: None
Output: BUS_CNTRL_DEV_CONFIG structure
Notes: Returns the values set by the previous call. See BusCntrlPublic.h for the definition of BUS_CNTRL_DEV_CONFIG.

IOCTL_BUS_CNTRL_GET_STATUS
Function: Returns the channel status and the values read from the two bus controller device interrupt status registers.
Input: None
Output: BUS_CNTRL_INT_STATUS structure
Notes: See the definition of BUS_CNTRL_INT_STATUS below.

```
typedef struct _BUS_CNTRL_INT_STATUS {
   USHORT   IntStatReg;
   USHORT   DevIntStat1;
   USHORT   DevIntStat2;
} BUS_CNTRL_INT_STATUS, *PBUS_CNTRL_INT_STATUS;
```

IOCTL_BUS_CNTRL_SET_MEM_WRITE_PTR
Function: Sets the value of the bus controller internal RAM write pointer.
Input: Memory offset (unsigned short integer)
Output: None
Notes: The memory write pointer is stored by the driver to be referenced in future WriteFile() calls.

IOCTL_BUS_CNTRL_SET_MEM_READ_PTR
Function: Sets the value of the bus controller internal RAM read pointer.
Input: Memory offset (unsigned short integer)
Output: None
Notes: The memory read pointer is stored by the driver to be referenced in future ReadFile() calls.

IOCTL_BUS_CNTRL_WRITE_MEM_WORD
Function: Writes a single word to the bus controller internal RAM.
Input: Memory offset and data to write (BUS_CNTRL_MEM_WRITE structure)
Output: None
Notes: This call is used to write small amounts of data or to non-contiguous areas of memory.

IOCTL_BUS_CNTRL_READ_MEM_WORD
Function: Reads a single word from the bus controller internal RAM.
Input: Memory offset (unsigned short integer)
Output: Memory word (unsigned short integer)
Notes: This call is used to read small amounts of data or from non-contiguous areas of memory.

IOCTL_BUS_CNTRL_SET_IINST_PTR
Function: Sets the value of the bus controller initial instruction pointer.
Input: Memory offset (unsigned short integer)
Output: None
Notes: This is the memory address of the first instruction of the message processing program.

IOCTL_BUS_CNTRL_GET_IINST_PTR
Function: Returns the value of the bus controller initial instruction pointer.
Input: None
Output: Memory offset (unsigned short integer)
Notes: Returns the value written in the previous call.

IOCTL_BUS_CNTRL_SET_GPQ_PTR
Function: Sets the value of the bus controller general-purpose queue pointer.
Input: Memory offset (unsigned short integer)
Output: None
Notes: The general purpose queue can be used for any purpose the user wishes. The queue is written by specific instructions in the message processing code.

IOCTL_BUS_CNTRL_GET_GPQ_PTR
Function: Returns the value of the bus controller general-purpose queue pointer.
Input: None
Output: Memory offset (unsigned short integer)
Notes: Returns the value written in the previous call.

IOCTL_BUS_CNTRL_SET_INTERRUPT_CONFIG
Function: Sets the bus controller device interrupt condition enables.
Input: BUS_CNTRL_INT_CONFIG structure
Output: None
Notes: Determines which conditions in the bus controller device will cause an interrupt. See the 1553 device data sheet for descriptions of the possible bus controller interrupt conditions. See BusCntrlPublic.h for the definition of BUS_CNTRL_INT_CONFIG.

IOCTL_BUS_CNTRL_GET_INTERRUPT_CONFIG
Function: Returns the values set in the previous call.
Input: None
Output: BUS_CNTRL_INT_CONFIG structure
Notes: See the 1553 device data sheet for bus controller interrupt condition descriptions. See BusCntrlPublic.h for the definition of BUS_CNTRL_INT_CONFIG.

IOCTL_BUS_CNTRL_START_PROCESSING
Function: Starts message processing at the address specified in the initial instruction pointer register.
Input: None
Output: None
Notes: The start bit is written to the start/reset register which initiates message processing. This instruction will also resume message processing if the message processor has halted due to an error or a halt instruction.

IOCTL_BUS_CNTRL_STOP_PROCESSING
Function: Stops the bus controller message processing.
Input: None
Output: None
Notes: This call writes a reset to the start/reset register and then does basic initialization of the bus controller.

IOCTL_BUS_CNTRL_SET_GP_FLAGS
Function: Sets, clears, toggles or leaves unchanged each of the eight general-purpose flags of the bus controller device.
Input: BUS_CNTRL_GPFLAG_CNTRL structure
Output: None
Notes: BUS_CNTRL_GPFLAG_CNTRL contains an array of eight values that specify the operation to be performed on each of the eight general-purpose flags. These flags are used to control the flow of the message processing program. See BusCntrlPublic.h for the definition of BUS_CNTRL_GPFLAG_CNTRL.

IOCTL_BUS_CNTRL_GET_GP_FLAGS
Function: Returns the state of the general-purpose flags and the status results from the last message processed by the bus controller.
Input: None
Output: BUS_CNTRL_GPFLAG_STATE structure
Notes: See BusCntrlPublic.h for the definition of BUS_CNTRL_GPFLAG_STATE.

IOCTL_BUS_CNTRL_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_BUS_CNTRL_ENABLE_INTERRUPT
Function: Enables the bus controller device master interrupt.
Input: None
Output: None
Notes: This command must be run to allow the 1553 bus controller device to generate interrupts. The master interrupt is disabled in the driver interrupt service routine. Therefore this command must be run after each interrupt is processed to re-enable the interrupts.

IOCTL_BUS_CNTRL_DISABLE_INTERRUPT
Function: Disables the master interrupt for the 1553 bus controller device.
Input: None
Output: None
Notes: This call is used when interrupt processing is no longer desired.

IOCTL_BUS_CNTRL_FORCE_INTERRUPT
Function: Causes a 1553 channel interrupt to occur.
Input: None
Output: None
Notes: Causes an interrupt to be asserted on the IP bus as if it were caused by the 1553 bus controller device. This IOCTL is used for test and development, to test interrupt processing.

IOCTL_BUS_CNTRL_GET_ISR_STATUS
Function: Returns the interrupt status and associated information from the last ISR.
Input: None
Output: Interrupt status values (BUS_CNTRL_INT_STAT)
Notes: Returns the status and 1553 bus controller device interrupt status (ISR1 and ISR2) that was read in the interrupt service routine for the last 1553 bus controller device interrupt serviced. The FPGA status register interrupt bit is shifted down three or five positions depending on the 1553 channel number to make it consistent for both channels.

The IOCTLs defined for the Monitor driver are described below:

IOCTL_MONITOR_GET_INFO
Function: Returns the monitor driver version, the IP-1553 device instance number and the IP-1553 device channel number.
Input: None
Output: MONITOR_DRIVER_DEVICE_INFO structure
Notes: The device number is passed to the channel devices so that the base device and channel device handles can be coordinated to reference the same physical module in the application software. See MonitorPublic.h for the definition of MONITOR_DRIVER_DEVICE_INFO.

IOCTL_MONITOR_SET_CONFIG
Function: Sets the channel and monitor internal device register configuration.
Input: MONITOR_DEV_CONFIG structure
Output: None
Notes: This call controls the channel enable, power-on BIT disable and Sleep mode as well as numerous internal device register controls. See MonitorPublic.h for the definition of MONITOR_DEV_CONFIG.

IOCTL_MONITOR_GET_CONFIG
Function: Returns the channel and monitor internal device configuration.
Input: None
Output: MONITOR_DEV_CONFIG structure
Notes: Returns the values set by the previous call. See MonitorPublic.h for the definition of MONITOR_DEV_CONFIG.

IOCTL_MONITOR_GET_STATUS
Function: Returns the channel status and the values read from the two monitor device interrupt status registers.
Input: None
Output: MONITOR_INT_STATUS structure
Notes: See MonitorPublic.h for the definition of MONITOR_INT_STATUS and the relevant status register bits.

IOCTL_MONITOR_SET_MEM_WRITE_PTR
Function: Sets the value of the monitor internal RAM write pointer.
Input: Memory offset (unsigned short integer)
Output: None
Notes: The memory write pointer is stored by the driver to be referenced in future WriteFile() calls.

IOCTL_MONITOR_SET_MEM_READ_PTR
Function: Sets the value of the monitor internal RAM read pointer.
Input: Memory offset (unsigned short integer)
Output: None
Notes: The memory read pointer is stored by the driver to be referenced in future ReadFile() calls.

IOCTL_MONITOR_WRITE_MEM_WORD
Function: Writes a single word to the bus controller internal RAM.
Input: Memory offset and data to write (MONITOR_MEM_WRITE structure)
Output: None
Notes: This call is used to write small amounts of data or to non-contiguous areas of memory.

IOCTL_MONITOR_READ_MEM_WORD
Function: Reads a single word from the bus controller internal RAM.
Input: Memory offset (unsigned short integer)
Output: Memory word (unsigned short integer)
Notes: This call is used to read small amounts of data or from non-contiguous areas of memory.

IOCTL_MONITOR_SET_ISQ_PTR
Function: Sets the initial value of the monitor interrupt status queue pointer.
Input: Memory offset (unsigned short integer)
Output: None
Notes: Bits 15 to 6 define the base address of the interrupt status queue. The value of these bits will not change in the course of processing. Bits 5 to 0 should initially be set to zero. These bits will increment as data is inserted into the queue. The ISQ pointer is always referencing the next location to be written. Use of the 64-word interrupt status queue is optional. If used, two words will be written to the queue for each interrupt event. The first word is the interrupt vector which indicates the cause of the interrupt. If bit zero in the interrupt vector is a one, it indicates that the interrupt was a result of a message. If this is the case, the second word will be a pointer to the first word in the command stack entry for that message. If the interrupt was caused by a RAM parity error, the second word will be the RAM address where the error occurred. If the interrupt was not caused by a message or a RAM parity error, the second word is not used and will be all zeros.

IOCTL_MONITOR_GET_ISQ_PTR
Function: Returns the current value of the monitor status queue pointer.
Input: None
Output: Memory offset (unsigned short integer)
Notes: The value of the interrupt status pointer is modulo 64. When the pointer reaches an address that is divisible by 64 it will roll-over to the initial pointer value.

IOCTL_MONITOR_SET_INTERRUPT_CONFIG
Function: Sets the monitor device interrupt condition enables.
Input: MONITOR_INT_CONFIG structure
Output: None
Notes: Determines which conditions in the monitor device will cause an interrupt. See the 1553 device data sheet for descriptions of the possible monitor interrupt conditions. See MonitorPublic.h for the definition of MONITOR_INT_CONFIG.

IOCTL_MONITOR_GET_INTERRUPT_CONFIG
Function: Returns the values set in the previous call.
Input: None
Output: MONITOR_INT_CONFIG structure
Notes: See the 1553 device data sheet for monitor interrupt condition descriptions. See MonitorPublic.h for the definition of MONITOR_INT_CONFIG.

IOCTL_MONITOR_START_PROCESSING
Function: Starts the message monitoring process.
Input: None
Output: None
Notes: The start bit is written to the start/reset register which initiates the message monitoring process.

IOCTL_MONITOR_STOP_PROCESSING
Function: Stops the message monitoring process
Input: None
Output: None
Notes: This call writes a reset to the start/reset register and then does basic initialization of the monitor device.

IOCTL_MONITOR_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_MONITOR_ENABLE_INTERRUPT
Function: Enables the monitor device master interrupt.
Input: None
Output: None
Notes: This command must be run to allow the 1553 monitor device to generate interrupts. The master interrupt is disabled in the driver interrupt service routine. Therefore this command must be run after each interrupt is processed to re-enable the interrupts.

IOCTL_MONITOR_DISABLE_INTERRUPT
Function: Disables the master interrupt for the 1553 monitor device.
Input: None
Output: None
Notes: This call is used when interrupt processing is no longer desired.

IOCTL_MONITOR_FORCE_INTERRUPT
Function: Causes a 1553 channel interrupt to occur.
Input: None
Output: None
Notes: Causes an interrupt to be asserted on the IP bus as if it were caused by the 1553 monitor device. This IOCTL is used for test and development, to test interrupt processing.

IOCTL_MONITOR_GET_ISR_STATUS
Function: Returns the interrupt status and associated information from the last ISR.
Input: None
Output: Interrupt status values (MONITOR_INT_STAT)
Notes: Returns the status and 1553 monitor device interrupt status (ISR1 and ISR2) that was read in the interrupt service routine for the last 1553 monitor device interrupt serviced. The FPGA status register interrupt bit is shifted down three or five positions depending on the 1553 channel number to make it consistent for both channels.

The IOCTLs defined for the Remote Terminal driver are described below:

IOCTL_REM_TERM_GET_INFO
Function: Returns the remote terminal driver revision, the IP-1553 device instance number and the IP-1553 device channel number
Input: None
Output: REM_TERM_DRIVER_DEVICE_INFO structure
Notes: The device number is passed to the channel devices so that the base device and channel device handles can be coordinated to reference the same physical module in the application software. See RemTermPublic.h for the definition of REM_TERM_DRIVER_DEVICE_INFO.

IOCTL_REM_TERM_SET_CONFIG
Function: Sets the channel and remote terminal internal device register configuration.
Input: REM_TERM_DEV_CONFIG structure
Output: None
Notes: This call controls the channel enable, power-on BIT disable and Sleep mode as well as numerous internal device register controls. See RemTermPublic.h for the definition of REM_TERM_DEV_CONFIG.

IOCTL_REM_TERM_GET_CONFIG
Function: Returns the channel and remote terminal internal device configuration.
Input: None
Output: REM_TERM_DEV_CONFIG structure
Notes: Returns the values set by the previous call. See RemTermPublic.h for the definition of REM_TERM_DEV_CONFIG.

IOCTL_REM_TERM_GET_STATUS
Function: Returns the channel status and the values read from the two remote terminal device interrupt status registers.
Input: None
Output: REM_TERM_INT_STATUS structure
Notes: See RemTermPublic.h for the definition of REM_TERM_INT_STATUS and the relevant status register bits.

IOCTL_REM_TERM_SET_MEM_WRITE_PTR
Function: Sets the value of the remote terminal internal RAM write pointer.
Input: Memory offset (unsigned short integer)
Output: None
Notes: The memory write pointer is stored by the driver to be referenced in future WriteFile() calls.

IOCTL_REM_TERM_SET_MEM_READ_PTR
Function: Sets the value of the remote terminal internal RAM read pointer.
Input: Memory offset (unsigned short integer)
Output: None
Notes: The memory read pointer is stored by the driver to be referenced in future ReadFile() calls.

IOCTL_REM_TERM_WRITE_MEM_WORD
Function: Writes a single word to the bus controller internal RAM.
Input: Memory offset and data to write (REM_TERM_MEM_WRITE structure)
Output: None
Notes: This call is used to write small amounts of data to non-contiguous areas of memory.

IOCTL_REM_TERM_READ_MEM_WORD
Function: Reads a single word from the bus controller internal RAM.
Input: Memory offset (unsigned short integer)
Output: Memory word (unsigned short integer)
Notes: This call is used to read small amounts of data from non-contiguous areas of memory.

IOCTL_REM_TERM_SET_ISQ_PTR
Function: Sets the initial value of the remote terminal interrupt status queue pointer.
Input: Memory offset (unsigned short integer)
Output: None
Notes: Bits 15 to 6 define the base address of the interrupt status queue. The value of these bits will not change in the course of processing. Bits 5 to 0 should initially be set to zero. These bits will increment as data is inserted into the queue. The ISQ pointer is always referencing the next location to be written. Use of the 64-word interrupt status queue is optional. If used, two words will be written to the queue for each interrupt event. The first word is the interrupt vector which indicates the cause of the interrupt. If bit zero in the interrupt vector is a one, it indicates that the interrupt was a result of a message. If this is the case, the second word will be a pointer to the first word in the command stack entry for that message. If the interrupt was caused by a RAM parity error, the second word will be the RAM address where the error occurred. If the interrupt was not caused by a message or a RAM parity error, the second word is not used and will be all zeros.

IOCTL_REM_TERM_GET_ISQ_PTR
Function: Returns the current value of the interrupt status queue pointer.
Input: None
Output: Memory offset (unsigned short integer)
Notes: The value of the interrupt status pointer is modulo 64. When the pointer reaches an address that is divisible by 64 it will roll-over to the initial pointer value.

IOCTL_REM_TERM_SET_INTERRUPT_CONFIG
Function: Sets the remote terminal device interrupt condition enables.
Input: REM_TERM_INT_CONFIG structure
Output: None
Notes: Determines which conditions in the 1553 device will cause an interrupt. See the 1553 device data sheet for descriptions of the possible remote terminal interrupt conditions. See RemTermPublic.h for the definition of REM_TERM_INT_CONFIG.

IOCTL_REM_TERM_GET_INTERRUPT_CONFIG
Function: Returns the values set in the previous call.
Input: None
Output: REM_TERM_INT_CONFIG structure
Notes: See the 1553 device data sheet for remote terminal interrupt condition descriptions. See RemTermPublic.h for the definition of REM_TERM_INT_CONFIG.

IOCTL_REM_TERM_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_REM_TERM_ENABLE_INTERRUPT
Function: Enables the remote terminal device master interrupt.
Input: None
Output: None
Notes: This command must be run to allow the 1553 remote terminal device to generate interrupts. The master interrupt is disabled in the driver interrupt service routine. Therefore this command must be run after each interrupt is processed to re-enable the interrupts.

IOCTL_REM_TERM_DISABLE_INTERRUPT
Function: Disables the master interrupt for the 1553 remote terminal device.
Input: None
Output: None
Notes: This call is used when interrupt processing is no longer desired.

IOCTL_REM_TERM_FORCE_INTERRUPT
Function: Causes a 1553 channel interrupt to occur.
Input: None
Output: None
Notes: Causes an interrupt to be asserted on the IP bus as if it were caused by the 1553 remote terminal device. This IOCTL is used for test and development, to test interrupt processing.

IOCTL_REM_TERM_GET_ISR_STATUS
Function: Returns the interrupt status and associated information from the last ISR.
Input: None
Output: Interrupt status values (REM_TERM_INT_STAT)
Notes: Returns the status and 1553 remote terminal device interrupt status (ISR1 and ISR2) that was read in the interrupt service routine for the last 1553 remote terminal device interrupt serviced. The FPGA status register interrupt bit is shifted down three or five positions depending on the 1553 channel number to make it consistent for both channels.

The IOCTLs defined for the Remote Terminal Monitor driver are described below:

IOCTL_RM_TRM_MON_GET_INFO
Function: Returns the remote terminal/monitor driver revision, the IP-1553 device instance number and the IP-1553 device channel number
Input: None
Output: RM_TRM_MON_DRIVER_DEVICE_INFO structure
Notes: The device number is passed to the channel devices so that the base device and channel device handles can be coordinated to reference the same physical module in the application software. See the definition of RM_TRM_MON_DRIVER_DEVICE_INFO below.

```
typedef struct _RM_TRM_MON_DRIVER_DEVICE_INFO {
   ULONG    DriverRev;
   USHORT   DeviceNum;
   UCHAR    Channel;
} RM_TRM_MON_DRIVER_DEVICE_INFO, *PRM_TRM_MON_DRIVER_DEVICE_INFO;
```

IOCTL_RM_TRM_MON_SET_CONFIG
Function: Sets the channel and remote terminal/monitor internal device register configuration.
Input: RM_TRM_MON_DEV_CONFIG structure
Output: None
Notes: This call controls the channel enable, power-on BIT disable and Sleep mode as well as numerous internal device register controls. See the definition of RM_TRM_MON_DEV_CONFIG in the RmTrmMonPublic.h file.

IOCTL_RM_TRM_MON_GET_CONFIG
Function: Returns the channel and remote terminal/monitor internal device configuration.
Input: None
Output: RM_TRM_MON_DEV_CONFIG structure
Notes: Returns the values set by the previous call. See the definition of RM_TRM_MON_DEV_CONFIG in the RmTrmMonPublic.h file.

IOCTL_RM_TRM_MON_GET_STATUS
Function: Returns the channel status and the values read from the two remote terminal/monitor device interrupt status registers.
Input: None
Output: RM_TRM_MON_INT_STATUS structure
Notes: See RmTrmMonPublic.h for the definition of RM_TRM_MON_INT_STATUS.

IOCTL_RM_TRM_MON_SET_MEM_WRITE_PTR
Function: Sets the value of the remote terminal/monitor internal RAM write pointer.
Input: Memory offset (unsigned short integer)
Output: None
Notes: The memory write pointer is stored by the driver to be referenced in future WriteFile() calls.

IOCTL_RM_TRM_MON_SET_MEM_READ_PTR
Function: Sets the value of the remote terminal/monitor internal RAM read pointer.
Input: Memory offset (unsigned short integer)
Output: None
Notes: The memory read pointer is stored by the driver to be referenced in future ReadFile() calls.

IOCTL_RM_TRM_MON_WRITE_MEM_WORD
Function: Writes a single word to the bus controller internal RAM.
Input: Memory offset and data to write (RM_TRM_MON_MEM_WRITE structure)
Output: None
Notes: This call is used to write small amounts of data to non-contiguous areas of memory.

IOCTL_RM_TRM_MON_READ_MEM_WORD
Function: Reads a single word from the bus controller internal RAM.
Input: Memory offset (unsigned short integer)
Output: Memory word (unsigned short integer)
Notes: This call is used to read small amounts of data from non-contiguous areas of memory.

IOCTL_RM_TRM_MON_SET_ISQ_PTR
Function: Sets the initial value of the remote terminal/monitor interrupt status queue pointer.
Input: Memory offset (unsigned short integer)
Output: None
Notes: Bits 15 to 6 define the base address of the interrupt status queue. The value of these bits will not change in the course of processing. Bits 5 to 0 should initially be set to zero. These bits will increment as data is inserted into the queue. The ISQ pointer is always referencing the next location to be written. Use of the 64-word interrupt status queue is optional. If used, two words will be written to the queue for each interrupt event. The first word is the interrupt vector which indicates the cause of the interrupt. If bit zero in the interrupt vector is a one, it indicates that the interrupt was a result of a message. If this is the case, the second word will be a pointer to the first word in the command stack entry for that message. If the interrupt was caused by a RAM parity error, the second word will be the RAM address where the error occurred. If the interrupt was not caused by a message or a RAM parity error, the second word is not used and will be all zeros.

IOCTL_RM_TRM_MON_GET_ISQ_PTR
Function: Returns the current value of the interrupt status queue pointer.
Input: None
Output: Memory offset (unsigned short integer)
Notes: The ISQ pointer is always referencing the next location to be written.

IOCTL_RM_TRM_MON_SET_INTERRUPT_CONFIG
Function: Sets the remote terminal/monitor device interrupt condition enables.
Input: RM_TRM_MON_INT_CONFIG structure
Output: None
Notes: Determines which conditions in the remote terminal/monitor device will cause an interrupt. See the 1553 device data sheet for descriptions of the possible remote terminal/monitor interrupt conditions. See RmTrmMonPublic.h for the definition of RM_TRM_MON_INT_CONFIG.

IOCTL_RM_TRM_MON_GET_INTERRUPT_CONFIG
Function: Returns the values set in the previous call.
Input: None
Output: RM_TRM_MON_INT_CONFIG structure
Notes: See the 1553 device data sheet for remote terminal/monitor interrupt condition descriptions. See RmTrmMonPublic.h for the definition of RM_TRM_MON_INT_CONFIG.

IOCTL_RM_TRM_MON_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_RM_TRM_MON_ENABLE_INTERRUPT
Function: Enables the remote terminal/monitor device master interrupt.
Input: None
Output: None
Notes: This command must be run to allow the 1553 remote terminal/monitor device to generate interrupts. The master interrupt is disabled in the driver interrupt service routine. Therefore this command must be run after each interrupt is processed to re-enable the interrupts.

IOCTL_RM_TRM_MON_DISABLE_INTERRUPT
Function: Disables the master interrupt for the 1553 remote terminal/monitor device.
Input: None
Output: None
Notes: This call is used when interrupt processing is no longer desired.
IOCTL_RM_TRM_MON_FORCE_INTERRUPT
Function: Causes a 1553 channel interrupt to occur.
Input: None
Output: None
Notes: Causes an interrupt to be asserted on the IP bus as if it were caused by the 1553 remote terminal/monitor device. This IOCTL is used for test and development, to test interrupt processing.

IOCTL_RM_TRM_MON_GET_ISR_STATUS
Function: Returns the interrupt status and associated information from the last ISR.
Input: None
Output: Interrupt status values (RM_TRM_MON_INT_STAT)
Notes: Returns the status and 1553 remote terminal/monitor device interrupt status (ISR1 and ISR2) that was read in the interrupt service routine for the last 1553 remote terminal/monitor device interrupt serviced. The FPGA status register interrupt bit is shifted down three or five positions depending on the 1553 channel number to make it consistent for both channels.

Write

Bus Control, Remote Terminal, Monitor or Remote Terminal Monitor data is written to the device using the write command. Writes are executed using the Win32 function WriteFile() (see below) and passing in the handle to the target device, a pointer to a pre-allocated buffer containing the data to be written, an unsigned long integer that represents the number of bytes to be transferred, a pointer to an unsigned long integer to contain the number of bytes actually written, and a pointer to an optional Overlapped structure for performing asynchronous I/O. The initial RAM address to begin the write is stored in the driver and can be updated with the SET_MEM_WRITE_PTR call.

```
BOOL WriteFile(
  HANDLE       hDevice,               // Handle opened with CreateFile()
  LPVOID       lpBuffer,              // Pointer to write buffer
  DWORD        nNumberOfBytesToWrite, // Size of write buffer
  LPDWORD      lpNumberOfBytesWritten,// Pointer to actual length parameter
  LPOVERLAPPED lpOverlapped,          // Optional pointer to overlapped
);                                    //  structure used for asynchronous I/O
```

Read

Bus Control, Remote Terminal, Monitor or Remote Terminal Monitor data is read from the device using the read command. Reads are executed using the Win32 function ReadFile() (see below) and passing in the handle to the target device, a pointer to a pre-allocated buffer that will contain the data read, an unsigned long integer that represents the number of bytes to be transferred, a pointer to an unsigned long integer to contain the number of bytes actually read, and a pointer to an optional Overlapped structure for performing asynchronous I/O. The initial RAM address to begin the read is stored in the driver and can be updated with the SET_MEM_READ_PTR call.

```
BOOL ReadFile(
  HANDLE       hDevice,               // Handle opened with CreateFile()
  LPVOID       lpBuffer,              // Pointer to read buffer
  DWORD        nNumberOfBytesToRead,  // Size of read buffer
  LPDWORD      lpNumberOfBytesRead,   // Pointer to actual length parameter
  LPOVERLAPPED lpOverlapped,          // Optional pointer to overlapped
);                                    //  structure used for asynchronous I/O
```

Warranty and Repair

Please refer to the warranty page on our website for the current warranty offered and options.
http://www.dyneng.com/warranty.html

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 or e-mail and arrange to work with an engineer. We will work with you to determine the cause of the issue.

Support

The software described in this manual is provided at no cost to clients who have purchased the corresponding hardware. Minimal support is included along with the documentation. For help with integration into your project please contact sales@dyneng.com for a support contract. Several options are available. With a contract in place Dynamic Engineers can help with system debugging, special software development, or whatever you need to get going.

For Service Contact:

Customer Service Department
Dynamic Engineering
150 DuBois Street, Suite C
Santa Cruz, CA 95060
831-457-8891
support@dyneng.com