Ethernet Interface Specification 2.0 (Preliminary)
10/30/2003 4:21 PM

Ethernet Interface Specification 2.0 (Preliminary)
10/30/2003 4:21 PM

INTRO

The interface DLL is written with the "less is more" philosophy. It is intended to be the minimum layer between the application and the I/O base. As such, packing requests and parsing responses are left to the client app. At some point in the future we may add some utility routines to assist with such things. We may also add an ActiveX and/or Java Beans interface to the DLL for easier development under VB, Java, etc. At this point (and hopefully for some time) this document and the product it represents are living. We desire your feedback. Best way is email to ebc@hosteng.com. If you don't like the doc, save it. Consider what you paid for it. If you have **constructive** criticism/enhancement ideas, bring them on.

Ethernet Interface Specification 2.0 (Preliminary)
10/30/2003 4:21 PM

OVERVIEW

The basic components of an I/O connection are **protocol**, **transport**, and **device**. The device is the EBC itself. It is represented by an HEIDevice structure. The protocol is an actual ethernet protocol such as IPX, TCP/IP, or NetBios. The transport is the mechanism by which the DLL connects to the underlying protocol, and includes WinSock, IPX interrupt API and NetBios interrupt API. What make this all totally confusing is that you can mix and match the protocols and transports to create a real mess. For instance, while WinSock can access all of the protocols, IPX API can only access IPX, but NetBios can be run over IPX, etc. etc. For simplicity, the protocol and transport are bundled together in an HEITransport structure. Once the connection is magically established, the client may interact with the EBC.

So how does that happen. In general (and subject to change without notice):

1) At the application's initialization, call HEIOpen() to allow the driver to init.
2) For each transport/protocol combination you wish to use, allocate an HEITransport structure, initialize the protocol and transport members, and call HEIOpenTransport().
3) Using the HEITransport you just opened, call HEIQueryDevices(). Initialize the variable pNumDevices points to, to the length (in devices) of the HEIDevice structure array. The array should be large enough to accommodate the largest number of device you wish to support. The function will query for devices, setting *pNumDevices to the number found and initializing the appropriate number of HEIDevice structures.
   A variation on the HEIQueryDevices is to use HEIQueryDeviceData and specify a value of one of the EBC's data fields. This allows you to do runtime address resolution and eliminates the need to hardcode a physical device address into the client application.
4) Using the HEITransport and HEIDevice for the desired device call HEIOpenDevice() to start the connection with the EBC. Once the device has been opened the HEITransport is bound to it.
5) Call HEIReadBaseDef to get the modules and I/O counts for the selected device.
6) Call HEIReadIO to get the I/O state of the base.
7) Call HEIWriteIO to set the output state of the base. HEIWriteIO also returns the input state.
8) When you are done, call HEICloseDevice() to close the device.
9) Call HEICloseTransport() to close the transport.
10) And call HEIClose() in the client shutdown routine to allow the DLL to tidy up.

There are other functions to setup, learn about, and generally abuse the EBC. When we figure out what they do we'll document them.

Just a note: If this document is wrong, ignore it. The header file is always right.

Another note: If you actually want to connect the EBC, the pinout for the connector is as follows…

TD+	pin 1
TD-	pin 2
RD+	pin 3
RD-	pin 6

This is actually the standard 10BaseT card pin out. The hub has TD and RD reversed. If you want to create point to point you will need to cross the cable, serial NULL modem style.

Happy I/Oing!!

INITIALIZATION / SHUTDOWN

SYSTEM

Application should call HEIOpen() once to initialize the ethernet interface, and should call HEIClose() upon completion of ethernet activity to allow the ethernet system to shutdown.

Functions:

```
int HEIOpen
      (
      WORD Ver    // Version number from HEI.H HEIAPIVERSION
      );

int HEIClose();
```

Errors:

PROTOCOL

```
typedef struct
      {
      WORD Transport;         /* HEIT_HOST */
                              /* HEIT_IPX */
                              /* HEIT_NETBIOS */
                              /* HEIT_WINSOCK */

      WORD Protocol;          /* HEIP_HOST*/
                              /* HEIP_IPX */
                              /* HEIP_IP */
                              /* HEIP_NETBIOS */
      /* Encryption stuff. */
      Encryption Encrypt;  /* Set this up before calling HEIOpenTransport  */
                           /* .Algorithm == HEIEN_NONE OR HEIEN_A1 */
                           /* .Key == Encryption key */

      DWORD NetworkAddress;
      } HEITransport;
```

Functions:

```
int HEIOpenTransport
      (
      HEITransport *pTransport,           // Transport to open
      HEIAPIVERSION,                      // Version from HEI.H
      DWORD NetworkAddress                // Network address
      );

int HEICloseTransport
      (
      HEITransport *pTransport            // Transport to close
      );
```

Errors:

DEVICE

```
typedef struct
{
union
      {
      /* Use this for HEIP_HOST protocol addressing. */
      struct
            {
            short Family;           /* AF_UNSPEC == 0 */
            char Nodenum[6];        /* Ethernet network address */
            unsigned short LANNum;  /* Lana number */
            } AddressHost;

      /* Use this for HEIP_IPX protocol addressing. */
      struct
            {
            short Family;           /* AF_IPX == 6 */
            char Netnum[4];         /* Network number */
            char Nodenum[6];        /* Ethernet network address */
            unsigned short Socket;  /* Socket number == 0x7070 */
            } AddressIPX;

      /* Use this for HEIP_IP protocol addressing. */
      struct
            {
            short Family;           /* AF_INET == 2 */
            unsigned short Port;    /* Port number == 0x7070 */
            union                   /* Internet address */
                  {
                  /* Byte addressing */
                  struct { unsigned char b1,b2,b3,b4; } bAddr;

                  /* Word addressing */
                  struct { unsigned short w1,w2; } wAddr;

                  /* DWord addressing */
                  unsigned long lAddr;
                  } AddressingType;
            char  Zero[8];          /* Initialize to zeros */
            } AddressIP;

      /* This is the generic address buffer. */
      BYTE Raw[20];
      } Address;

WORD wParam;      /* Application can use this. */
DWORD dwParam;    /* Application can use this. */

WORD Timeout;     /* Timeout value in ms (can be changed without */
                  /* closing the device). */
WORD Retrys;      /* Number of times to retry (can be changed */
                  /* without closing the device). */

BYTE ENetAddress[6];    /* The ethernet address is placed here in */
                        /* the HEIQueryDevices call. */

WORD RetryCount;        /* Number of retrys which have occured. */
WORD BadCRCCount;       /* Number of packets received with bad CRC */
WORD LatePacketCount;   /* Number of packets received late, (after */
                        /* a timeout) */
```

```
            BOOL ParallelPackets;   /* Setting this to TRUE (after HEIOpenDevice)*/
                                    /* will enable an application                */
                                    /* to send multiple HEIReadIO, HEIWriteIO,   */
                                    /* HEICCMRequest or  HEIKSEQRequest requests */
                                    /* (to different) modules before waiting for */
                                    /* any responses. The application will then  */
                                    /* need to implement its own retry / timeout */
                                    /* mechanism while waiting for the responses.*/
                                    /* The application uses HEIGetResponse to see */
                                    /* if a response for a module has arrived.   */
                                    /* NOTE: The application should not send */
                                    /* multiple requests to a single module */
                                    /* without waiting for the response in  */
                                    /* between. */

            /* Internal - Do not touch!! */
            BOOL UseAddressedBroadcast;   /* Need to close the device and  */
                                          /* reopen it to change this! */
            BOOL UseBroadcast;
            DWORD _dwParam;
            WORD DataOffset;
            HEITransport *_pTransport;    /* Need to close the device */
                                          /* and reopen it to change this! */
            int SizeOfData;
            BYTE *pData;
            void *pBuffer;
            unsigned short LastAppVal;
            } HEIDevice;
```

Functions:

```
int HEIQueryDevices
      (
      HEITransport *pTransport,     // Transport to use
      HEIDevice *pDevices,          // Array of devices to be filled in
      WORD *pNumDevices,            // Number of devices in the array
      HEIAPIVERSION                 // Version number from HEI.H
      );

int HEIQueryDeviceData
      (
      HEITransport *pTransport,     // Transport to use
      HEIDevice *pDevices,          // Array of devices to be filled in
      WORD *pNumDevices,            // Number of devices in the array
      HEIAPIVERSION,                // Version number from HEI.H
      WORD DataType,                // Data type to query for
      BYTE *pData,                  // Data buffer containing value to query for
      WORD SizeofData               // Size of data in buffer
      );

int HEIOpenDevice
      (
      HEITransport *pTransport,     // Transport to associate with the device
      HEIDevice *pDevice,           // Device to open
      HEIAPIVERSION,                // Version number from HEI.H
      WORD iTimeout,                // Timeout in milliseconds for a transaction
      WORD iRetrys,                 // Number of times to retry a transaction
                                    // after a timeout
      BOOL UseAddressedBroadcast    // Indicates whether to use addressed broadcast
                                    // to talk to this device or not.
      );
```

```
int HEICloseDevice
      (
      HEIDevice *pDevice            // Device to close.
      );
```

Errors:

SUPPORT INFORMATION

Offset	Bit Number	Description
0	76543210	Version of Support Info
1	76543210	Length of Data (in Bytes)
2	0	Poll Single Device
	1	Read Version Info
	2	Read Support Info
	3	Read Device Info
	4	Poll All Devices
	5	Write I/O
	6	Read I/O
	7	Read Base Def
3	0	Enum Setup Data
	1	Read Setup Data
	2	Write Setup Data
	3	Delete Setup Data
	4	Read Ethernet Stats
	5	Pet Link
	6	Addressed Broadcast
	7	Read Module Status
4	0	Extended Function
	1	Query Setup Data
	2	Init Base Def
	3	Data Broadcast
	4	CCM Request
	5	Kseq Request
	6	Backplane Request
	7	Write Base Def
5	0	Extend Response
	1	Ack
	2	Nak
	3	Response
	4	Serial Port Operation
	5	Write Memory
	6	Read Memory
	7	ENUM Memory
6	0	Read Shared Ram
	1	Write Shared Ram
	2	Unused
	3	Unused
	4	Unused
	5	Unused
	6	Unused
	7	Unused

Function for Reading:

```
int HEIReadSupportInfo
      (
      HEIDevice *pDevice,           // Device to read support info for
      BYTE *pSupportInfo,           // Pointer to data buffer to be filled
      WORD SizeOfSupportInfo        // Size of data buffer
      );
```

Function for Writing:

N/A

Errors:

VERSION INFORMATION

4 Byte Version Info Structure:

Offset	76543210
0	Major Version
1	Minor Version
2	Build Version (LSB)
3	Build Version (MSB)

Version Info:

Offset	76543210
0	Sizeof Version Info
1	(Version Info) Boot Version
5	(Version Info) OS Version
9	# OS Extensions
10	(Version Info) OS Ext 1 Ver
14	(Version Info) OS Ext 2 Ver
…	…
	(Version Info) OS Ext N Ver

Function for Reading:

```
int HEIReadVersionInfo
      (
      HEIDevice *pDevice,     // Device to read version info for
      BYTE *pVerInfo,         // Buffer to be filled with version info data
      WORD SizeVerInfo        // Size of buffer.
      );
```

Function for Writing:

N/A

Errors:

DEFINITION

BASE

See Appendix A for exciting details about the data format for this function.

Function for Reading:

```
int HEIReadBaseDef
      (
      HEIDevice *pDevice,           // Device to read base def info for
      BYTE *pBaseDefInfo,           // Buffer to hold base def info
                                    // (see Appendix A)
      WORD *pSizeOfBaseDefInfo      // Pointer to the size of the buffer, will
                                    // contain the length of the data in the
                                    // buffer on return.
      );
```

Function for Writing:

```
int HEIWriteBaseDef
      (
      HEIDevice *pDevice,           // Device to write base def  info for
      BYTE *pInputBaseDef,          // Buffer to hold base def info (see Appendix A)
      WORD SizeOfInputBaseDef,      // Size of the buffer
      BYTE *pOutputBaseDef,         // Buffer to hold base def info (same as HEIReadBaseDef)
      WORD *pSizeOfOutputBaseDef    // Pointer to size of the buffer, will contain the length of the
                                    // data in the buffer on return.
      );
```

Errors:

DEVICE DEF

Offset	76543210
0	PLC Family
1	PLC Type
2	Module Type
3	Status Code
4-9	6 Byte Ethernet Address
10-11	k-Bytes of Ram (WORD)
12-13	k-Bytes of Flash (WORD)
14	Dip switch setting
15	Media Type
16-19	Early Power Fail Count
20	Bit 0: Run Relay Status
20	Bit 1: Battery Low Status

PLC Family:	2 - 205 Family
	3 - 305 Family
	4 - 405 Family
PLC Type:	25 - (234)25 PLC
	30 - (234)30 PLC
	35 - (234)35 PLC
	40 - (234)40 PLC
	45 - (234)45 PLC
	50 - (234)50 PLC
Module Type	0 - Ethernet Base Controller module
	1 - Ethernet co-processor module
Status Code:	0 - No Errors
	1 -
Media Type	0 – 10 Base T
	1 – 10 Base F

Function for Reading:

```
int HEIReadDeviceDef
      (
      HEIDevice *pDevice,           // Device to get device info for
      BYTE *pModuleDefInfo,         // Buffer to store device info in
      WORD SizeOfModuleDefInfo      // Size of device info buffer
      );
```

Function for Writing:

N/A

Read / Write

I/O

See Appendix A for details about the data format for this function.

Functions for Reading:

```
// Function for reading I/O from a single device.
int HEIReadIO
      (
      HEIDevice *pDevice,     // Device to read I/O from
      BYTE *pBuffer,          // Buffer to hold I/O info (see Appendix A)
      WORD BuffSize           // Size of I/O info buffer.
      );

// Function for reading I/O from multiple devices.
int HEIReadIOEx
      (
      HEIDevice *apDevice[],          //  Array of pointers to devices to read I/O from
      BYTE *apData[],                 //  Array of  pointers to data buffers to hold I/O data (see Appendix A)
      WORD aSizeofData[],             //  Array of buffer sizes.
      int aErrorCode[],         //  Array of returned error codes.
      int DeviceCount                 //  Number of devices in array.
      );
```

Functions for Writing:

```
// Function for writing I/O to a single device.
int HEIWriteIO
      (
      HEIDevice *pDevice,     // Device to write I/O to
      BYTE *pData,            // Buffer containing I/O data (see Appendix A)
      WORD SizeofData,        // Size of Buffer containing I/O data
      BYTE *pReturnData,      // Buffer to hold returned I/O info
                              // (same as ReadIO) NULL to ignore
      WORD *pSizeofReturnData // Pointer to size of return buffer
                              // (same as ReadIO) NULL to ignore
      );

// Function for writing I/O to multiple devices.
int HEIWriteIOEx
      (
      HEIDevice *apDevice[],          // Array of pointers to the devices
      BYTE *apData[],                 // Array of pointers to data buffers with I/O data to write  (see
                                      //  Appendix A)
      WORD aSizeofData[],             // Array of data buffer sizes (for write data)
      BYTE *apReturnData[],           // Array of pointers to data buffers for incoming (read) data.
      WORD aSizeofReturnData[],       // Array of data buffer sizes (for read data)
      int aErrorCode[],         // Array of error codes.
      int DeviceCount                 // Number of devices in array.
      );
```

Error values:

ReturnValue == 0 → No error, warning, or info
ReturnValue < 0 → Error or Warning or Info
    ReturnValue & 0x1000 → Error exists in a module (call HEIReadModuleStatus)
    ReturnValue & 0x2000 → Warning exists in a module (call HEIReadModuleStatus)
    ReturnValue & 0x4000 → Info exists in a module (call HEIReadModuleStatus)
ReturnValue > 0 → Undefined

Setup

Setup Data Types:

```
#define DT_IP_ADDRESS          0x0010  // 4 Byte IP address (Used for IP communication) - Must be initialized before
                                       // IP communication can be used with the device.
#define DT_NODE_NUMBER         0x0020  // 4 Byte Node Number (Used by PC application software)
#define DT_NODE_NAME           0x0016  // 256 Byte Node Name (Used by PC application software)
#define DT_DESCRIPTION         0x0026  // 256 Byte Node Description (Used by PC application software)
#define DT_LINK_MONITOR        0x8006  // 256 Byte Link monitor setup (Used to configure link watchdog)
                                       // Link monitor is used to define how long the module should wait without
                                       // a ReadIO, WriteIO, or PetDevice call before determining that the control
                                       // program has gone away.  It also defines what to do with the outputs in
                                       // this event.
                                       // Format:
        typedef struct
                {
                DWORD Timeout;         /* Timeout 0 == Don't use link monitor */
                BYTE Mode;             /* Mode: */
                                       /*              0 == Clear outputs */
                                       /*              1 == Set outputs to given I/O data pattern */
                BYTE Data[251];        /* Pattern:  Used with set outputs,  same format */
                                       /*                      as data for HEIWriteIO call. */
                } LinkMonitor;


#define DT_ENCRYPT_KEY_FLASH   0x0014
#define DT_ENCRYPT_KEY_RAM     0x8014

        typedef struct
                {
                BYTE Algorithm;        /* Algorithm to use for encryption */
                                       /*              0 == No encryption */
                                       /*              1 == Private key encryption */
                BYTE Unused[3];        /* Reserved for later */
                BYTE Key[60];          /* Encryption key (null terminated) */
                }  Encryption;


#define DT_RXWX_SETTINGS       0x0015
#define DT_SETTINGS            0x0015

        typedef struct
                {
                WORD SizeofSettings;   /* sizeof(HEISettings) */

                /* Action items. */
                DWORD Flags;           /* Flags used to control things. */
                                       /* Bit:    Function:  */
                                       /*              0-31            Unused          */
```

```
        /* RXWX Config items. */
        WORD RXWXACKTimeout;       /* Timeout for receiving ACK / NAK */
        WORD RXWXResponseTimeout;  /* Timeout for receiving response. */
        WORD RXWXMaxRetrys         ;          /* Number of times to retry a transaction. */

        /* RXWX Stat Items. */
        WORD RXWXMaxACKTime;       /* STAT: Max number of ms we have waited for an ack. */
        WORD RXWXMaxRSPTime;       /* STAT: Max number of ms we have waited for a response. */
        DWORD RXWXACKRetrys;       /* STAT: Number of retrys for an ack. */
        DWORD RXWXRSPRetrys;       /* STAT: Number of retrys for a response. */
        DWORD RXWXCompleted;       /* STAT: Number of successfully completed transactions */
        DWORD RXWXTimeouts;        /* STAT: Number of timeouts on transactions (after retrys) */
        DWORD RXWXOverruns;        /* STAT: Number of times the PLC requested a transaction while */
                                   /* one was being processed. */
        DWORD RXWXErrors;          /* STAT: Number of times an invalid code was found or a  */
                                   /*transaction   was NAKed. */

        /* Other stuff */
        BYTE Version;              /* Version of this structure.  Currently 0 */

        /* K-Sequence Retrys */
        WORD KSeqMaxRetrys;        /* Max number of times to retry a K-Sequence request          */
        WORD KSeqRetrys;           /* STAT: Number of K-Sequence retrys.                         */
        WORD KSeqTimeouts;         /* STAT: Number of K-Sequence timeouts. */

        BYTE Unused[81];           /* Reserved for future use.   */
        } HEISettings;


#define DT_SERIAL_SETUP       0x0011

        /* Serial port defines */
        #define SERIAL_1_STOP_BIT     0
        #define SERIAL_2_STOP_BITS    1

        #define SERIAL_7_DATA_BITS    0
        #define SERIAL_8_DATA_BITS    1

        #define SERIAL_NO_PARITY      0
        #define SERIAL_ODD_PARITY     2
        #define SERIAL_EVEN_PARITY    3

        #define SERIAL_SLAVE          0
        #define SERIAL_MASTER         1
        #define SERIAL_PROXY          1

        #define SERIAL_NO_RTS         0
        #define SERIAL_USE_RTS        1



        typedef struct
                {
```

```
        DWORD BaudRate;                                 /* Baud rate to use i.e. 9600 */

    #if defined (ANSI_C)
        BYTE    ConfigData;
    #else
        BYTE    StopBits        : 1;    /* 0 == 1 Stop bit;  1 == 2 Stop bits */
        BYTE    DataBits : 1;   /* 0 == 7 Data bits;  1 == 8 Data bits */
        BYTE    Parity          : 2;    /* 0 == 1 == None; 2 == Odd;  3 == Even */
        BYTE    Mode            : 1;    /* 0 == Slave;  1 == Master/Proxy */
        BYTE    UseRTS          : 1;        /* 0 == Don't use RTS line;  1 == Use RTS line */
        BYTE    Reserved        : 2;    /* Reserved locations */
    #endif
        BYTE    PreTransmitDelay; /* If UseRTS == 1 delay this many ms (times 2) before starting transmit */
        BYTE    PostTransmitDelay;  /* If UseRTS == 1 delay this many ms (times 2) after ending transmit */
        BYTE    Unused[1];
        } SerialSetup;

#define DT_BASE_DEF             0x0017  /* 512 Byte Base Def (405 HEIWriteBaseDef) */

#define DT_MODULE_SETUP     0x0024  /* 64 Byte data from FLASH. See ModuleSetup structure */

    typedef struct
        {
        BYTE RunRelayMode;      /* 405 EBC Run Relay Mode */
                                /* See Run relay modes below. */
        BYTE Unused[63];
        } ModuleSetup;

    /* Run relay Modes */
    #define RRM_LINK_GOOD               0       /* Run relay on when link is good (default) */
    #define RRM_LINK_NOT_GOOD           1       /* Run relay on when link is not good */
    #define RRM_POWERUP_ON              2       /* EBC turns run relay mode on on powerup */
    #define RRM_MANUAL_ON               3       /* Run relay mode is controlled by control software */



Function for Reading:
    int HEIReadSetupData
        (
        HEIDevice *pDevice,     // Device to read from
        WORD SetupType,         // Data type to read (see list of data types)
        BYTE *pData,            // Buffer to store setup data in
        WORD *pSizeofData       // Size of setup data buffer, returns number of bytes placed in buffer.
        );

Function for Writing:

    int HEIWriteSetupData
        (
        HEIDevice *pDevice,     // Device to setup
        WORD SetupType,         // Data type to setup (see list of data types)
        BYTE *pData,            // Setup Data to store
        WORD SizeofData         // Size of setup data
```

```
);
```

Function for Removing:

```
int HEIDeleteSetupData
    (
    HEIDevice *pDevice,     // Device to remove data from
    WORD SetupType          // Data type to remove (see list of data types)
    );
```

Function for enumerating:

```
int HEIEnumSetupData
    (
    HEIDevice *pDevice,             // Device to enumerate
    WORD *pData,                    // WORD Buffer to hold data types
    WORD *pSizeofDataInWords        // Size of WORD Buffer, returns number of data types found
    );
```

Config

The config functions are used to write configuration data to specific modules. This data is currently only used with our Terminator I/O family of modules. Certain of our Terminator Analog Input and Output modules have configuration bytes which are used to configure the modules behavior. Please refer to the documentation that comes with your modules to see if there are configuration options.

See Appendix A for details about the data format for these functions.

Function for reading:

```
int HEIReadConfigData
    (
    HEIDevice *pDevice,             // Device to read config data from
    BYTE *pData,                    // Buffer to hold config data
    WORD *DataSize                  // INPUT: Size of pData buffer,
                                    // OUTPUT: Num bytes placed in pData buffer
    );
```

Function for writing:

```
int HEIWriteConfigData
    (
    HEIDevice *pDevice,             // Device to read config data from
    BYTE *pData,                    // Config data to write (See Appendix A for format)
    WORD SizeofData,                // Bytes of config data to write.
    BYTE *pReturnData,              // Buffer to hold return config data
    WORD *pSizeofReturnData         // INPUT: Size of pReturnData buffer
```

```
                                        // OUTPUT: Num bytes placed in pReturnData buffer
    );
```

NOTE: You can also Use HEIWriteIO to write config data using type DF_CONFIG

ECOM Functions

Function to perform a CCM / DirectNet request on an ECOM module

```
int HEICCMRequest
    (
    HEIDevice *pDevice,     // Device to perform request on
    BOOL bWrite,            // if TRUE, we are writing data other we are reading data
    BYTE DataType,          // Type of data to read / write (see DirectNet manual)
    WORD Address,           // Address of data to read / write
    WORD DataLen,           // Length of data to read / write
    BYTE *pData             // Buffer for read / write data
    );
```

Function to perform a K-Sequence request on an ECOM Module

```
int HEIKSEQRequest
    (
    HEIDevice *pDevice,     // Device to perform request on
    WORD DataLenIn,         // Length of K-Sequence request
    BYTE *pData,            // Buffer for input / output data
    WORD *pDataLen          // Length of data returned
    );
```

SERIAL PORT Functions

Functions to write to a communications port:

```
int HEIWriteComm
        (
        HEIDevice *pDevice,     // Device to use
        WORD Num2Write,         // Number of bytes to write to modules serial port
        BYTE *pData             // Data to write
        );

int HEIWriteCommEx
        (
        HEIDevice *pDevice,     // Device to use
        BYTE Port ,             // Port to write to
        WORD Num2Write,         // Number of bytes to write to modules serial port
        BYTE *pData             // Data to write
        );
```

Functions to read from a communications port:

```
int HEIReadComm
        (
        HEIDevice *pDevice,     // Device to use
        WORD *pNum2Read,        // Number of bytes to read from modules serial port
        BYTE *pData             // Buffer to hold data
        );

int HEIReadCommEx
        (
        HEIDevice *pDevice,     // Device to use
        BYTE Port,              // Port to read from
        WORD *pNum2Read,        // Number of bytes to read from modules serial port
        BYTE *pData             // Buffer to hold data
        );
```

Functions to get number of read chars available for a communications port:

```
int HEIGetRXAvailable
        (
        HEIDevice *pDevice,     // Device to use
        WORD *pAvailable        // Return value for number of bytes available to read
        );

int HEIGetRXAvailableEx
        (
        HEIDevice *pDevice,     // Device to use
        BYTE Port,              // Port to get RX available for
        WORD *pAvailable        // Return value for number of bytes available to read
        );
```

Functions to get number of TX chars left in a communications port:

```
int HEIGetTXLeft
        (
        HEIDevice *pDevice,     // Device to use
        WORD *pLeft             // Pointer to WORD to hold number of TX chars
        );

int HEIGetTXLeftEx
        (
        HEIDevice *pDevice,     // Device to use
        BYTE Port,              // Port to use
        WORD *pLeft             // Pointer to WORD to hold number of TX chars
        );
```

Functions to get flush characters from a communications port:

```
int HEIFlushRXQueue
        (
        HEIDevice *pDevice      // Device to use
        );

int HEIFlushRXQueueEx
        (
        HEIDevice *pDevice,     // Device to use
        BYTE Port               // Port to use
        );

int HEIFlushTXQueueEx
        (
        HEIDevice *pDevice,     // Device to use
        BYTE Port               // Port to use
        );
```

Functions to configure a communications port:

```
int HEISetupSerialPort
        (
        HEIDevice *pDevice,     // Device to use
        SerialSetup *pSetup,    // Pointer to SerialSetup structure (see DT_SERIAL_SETUP above)
        BOOL WriteToFlash       // If TRUE, will write setup to flash, if FALSE, will not update FLASH
        );

int HEIReadSerialPortSetup
        (
        HEIDevice *pDevice,     // Device to use
        SerialSetup *pSetup     // Pointer to SerailSetup structure (see DT_SERIAL_SETUP above)
        );




int HEISetupSerialPortEx
```

```
        (
        HEIDevice *pDevice,     // Device to use
        BYTE Port,              // Port to configure
        SerialSetup *pSetup,    // Pointer to SerialSetup structure (see DT_SERIAL_SETUP above)
        BOOL WriteToFlash
        );

int HEIReadSerialPortSetupEx
        (
        HEIDevice *pDevice,     // Device to use
        BYTE Port,              // Port to read configuration from
        SerialSetup *pSetup     // Pointer to SerialSetup structure (see DT_SERIAL_SETUP above)
        );
```

Function to perform multiple operations on a serial port:

```
int HEIAccessComm
        (
        HEIDevice *pDevice,             // Device to use
        WORD SendDataSize,              // Size of data in pSendData
        BYTE *pSendData,                // Data to send (see below)
        WORD *pReturnDataSize,          // Returns number of bytes in pReturnData
        BYTE *pReturnData               // Data returned from port
        );
```

The format of the data is as follows:

```
Command
Port
Optional data byte(s)
Command
Port
Optional data byte(s)
…..
Command
Port
Optional data byte(s)
SPC_DONE
```

The following commands can be used:

SPC_WRITE_PORT – Writes one or more bytes to the given port
Format: SPC_WRITE_PORT PortNum NumBytes Byte1 Byte2 … ByteN
SPC_READ_PORT – Reads one or more bytes from the given port
Format: SPC_READ_PORT PortNum NumBytes
SPC_RX_FLUSH – Flush the RX buffer for the given port
Format: SPC_RX_FLUSH PortNum
SPC_TX_FLUSH – Flush the TX buffer for the given port
Format: SPC_TX_FLUSH PortNum
SPC_DONE – Indicates the end of the chain of commands
Format: SPC_DONE

The following additional items may be returned from the call to HEIAccessComm

SPC_ERROR - Reports the last error for the given port
Format: SPC_ERROR PortNum ErrorNum (See HEIE_XXX in HEI.H)

SPC_READ_RESPONSE - Response to an SPC_READ _PORT Function
Format: SPC_READ_RESPONSE PortNum Byte1 Byte2 … ByteN

MEMORY Functions

Memory functions are used to read and write and enumerate PLC type memory in an EBC. The currently supported memory types are as follows:

```
#define MT_KOYO_V     0x0200          /* V-Memory */
#define MT_KOYO_C     0x0000          /* C-Memory */
#define MT_KOYO_Z     0x0120          /* Scratch Pad Memory */
```

This memory may also be accessed by a serial device (such as a DV1000 or Optimate panel) through the modules serial port when the serial port is setup in slave mode.

```
int HEIReadMemory
        (
        HEIDevice *pDevice,     // Device to access
        WORD Type,              // Type of memory to read
        DWORD Offset,           // Offset to read from
        WORD NumBytes,          // Number of bytes to read
        BYTE *pBuffer           // Buffer to hold memory read  from device
        );

int HEIWriteMemory
        (
        HEIDevice *pDevice,     // Device to access
        WORD Type,              // Type of memory to write
        DWORD Offset,           // Offset to write to
        WORD NumBytes,          // Number of bytes to write
        BYTE *pBuffer           // Data to write
        );

int HEIENumMemory
        (
        HEIDevice *pDevice,     // Device to access
        WORD *pNumTypes,        // Input: number of  MemoryTypeDefs in pBuffer
                                // Output: number of MemoryTypeDefs used
        MemoryTypeDef *pBuffer          // Pointer to array of MemoryTypeDef structures
        );

typedef struct
        {
        WORD Type;              /* Type of memory */
        DWORD Size;             /* Size of memory */
        DWORD Unused[4];        /* Unused */
        } MemoryTypeDef;

int HEIAccessMemory
```

```
        (
        HEIDevice *pDevice,     // Device to access
        MemRef  MemRefs[],      // Array of Memory reference structures (See below)
        WORD NumRefs            // Number of memory references in structure.
        );

#define ACCESS_READ             0
#define ACCESS_WRITE1

typedef struct sMemRefDetail
        {
        BYTE Direction;         // ACCESS_READ == Read, ACCESS_WRITE == Write
        WORD Type;              // Memory type
        DWORD Offset;           // Memory Offset
        WORD NumBytes;          // Number of bytes
        } MemRefDetail;

typedef struct
        {
        MemRefDetail Detail;    // Memory type, offset, numbytes, and direction
        BYTE *pBuffer;          // Data buffer for read/write operation
        } MemRef;
```

Miscellaneous

This function is used to keep the given Device from firing the Link watchdog in the absence of a ReadIO or WriteIO call.

```
int HEIPetDevice
        (
        HEIDevice *pDevice      // Device to pet.
        );
```

This function is used to obtain ethernet statistics.

```
int HEIReadEthernetStats
        (
        HEIDevice *pDevice,     // Device to get statistics from
        BYTE *pData,            // Buffer to hold statistics data  (see below for format)
        WORD *DataSize,         // Pointer to size of buffer, returns number of bytes placed in buffer.
        BOOL Clear              // If TRUE, the ethernet statistics will be cleared after they are read.
                                // If FALSE, the ethernet statistics will not be cleared.
        );
```

Format of data from HEIReadEthernetStats

```
#pragma pack(1)
typedef struct
        {
        WORD SizeofEthernetStats;       // Size of this structure
        DWORD MissedFrameCount;         // Number of  frames missed by the ethernet chip
        DWORD TransmitCollisionCount; // Number of transmit collisions
        DWORD DiscardedPackets;         // Number of packets received, but discarded.
        } EthernetStats;
```

This function is used to read the status (error, warning, info, etc.) for each module in the base.

```
int HEIReadModuleStatus
        (
        HEIDevice *pDevice,     // Device to read the status from
        BYTE *pData,            // Buffer to hold the status data  (see appendix A for format)
        WORD *DataSize,         // Size of buffer, returns number of bytes placed in buffer.
        BOOL Reset              // If TRUE, module status will be reset after being read
                                // If FALSE, module status will not be reset
        );
```

NEW FORMAT VARIABLE LENGTH BASEDEF AND I/O STATE DATA SPECIFICATION

New format buffers begin with the following two bytes:

0xBn 0x00

n is the revision number from 0x00-0x0F (Currently 0x01)

To use HEIReadIO to request new I/O format on a device that supports both formats, set the first byte of the return buffer to: 0xBn (where 'n' is as above).

Example:

```
ReturnDataSize = sizeof(ReturnData);
ReturnData[0] = 0xB1;            // Request new I/O format.
int Error = HEIReadIO(pDevice, ReturnData, &ReturnDataSize);
```

```
Function:   Slot - Begins any slot specific data.
Code:       00
Format:     00 ss [ll mm]
            ss: Slot number 0 – 255
                  if (ss==255)
                        ll mm is a two-byte slot number
                  else
                        ll mm not included

Function:   Module definition
Code:       01
Format:     01 nn [ll mm] tt ii [Type specific data]
            nn:   Length of type data in bytes
                  if (nn==255)
                        ll mm is a two-byte length
                  else
                        ll mm not included
            tt:   generic module type.
                        0 - No module
                        1 - Generic I/O
                        2 – Intelligent Module (Type 1)
                        3 – Intelligent Module (Type 2)
                        4 – Special I/O
                        5 – Special I/o
                        6 - Unassigned
                        7 – List of types.
                        8 - FF Unassigned
            ii:   Module ID.

            Type = 0
            Format: 01 00 00 FF
                  No additional data

            Type = 1
            Format: 01 04 01 ii xx yy kk vv
```

```
      xx:   Discrete input count.
      yy:   Discrete output count.
      kk:   Word in count.
      vv:   Word in/out count.
      (see Type 1 reference table below)

Type = 2
Format: Unassigned

Type = 3 && ID = 0x18
Format: 01 04 03 18 xx yy kk vv
      xx:   Discrete input count.
      yy:   Discrete output count.
      kk:   DWord input count.
      vv:   DWord output count.


Type = 3 && ID != 0x18
Format: Unassigned

Type = 4
Format: 01 06 04 ii xx yy kk vv cc dd
      ii:   Unused (currently zero)
      xx:   Discrete input count.
      yy:   Discrete output count.
      kk:   Word input count.
      vv:   Word output count.
      cc:   DWord input count.
      cd:   Dword output count.



Type = 5
Format: 01 0C 05 ii di do bi bo wi wo dwi dwo fi fo dbli dblo
      ii:   Module ID
      di:   Discrete input count.
      do:   Discrete output count.
      Wi:   Word input count
      Wo:   Word output count
      DWi:  Dword input count
      DWo:  DWord output count.
      bi:   Byte input count
      bo:   Byte output count
      Fi:   Float input count
      Fo:   Float output count
      Dbli: Double input count
      Dblo: Double output count

Type = 6
Format: Unassigned




Type = 7
Foramt: 01 nn [ll mm] 07 ii ## T1 N1 T2 N2 .. T# N# [oT oF o1 .. oX]
      nn:   Length of type data in bytes
            if (nn==255)
                  ll mm is a two-byte length
            else
```

```
                    ll mm not included
              ii:   Module ID
              ##:   Number of type/num pairs following
              T1-#: Data format:
                    /* Defines for Data formats */
                    #define DF_BIT_IN             0x03
                    #define DF_BIT_OUT            0x04
                    #define DF_BYTE_IN            0x10
                    #define DF_BYTE_OUT           0x11
                    #define DF_WORD_IN            0x05
                    #define DF_WORD_OUT           0x06
                    #define DF_DWORD_IN           0x08
                    #define DF_DWORD_OUT          0x09
                    #define DF_DOUBLE_IN          0x12
                    #define DF_DOUBLE_OUT         0x13
                    #define DF_FLOAT_IN           0x14
                    #define DF_FLOAT_OUT          0x15

              N1-#:       Number of given data format elements.
              OT:   Optional type
                    #define MT_EBC    0
                    #define MT_ECOM   1
                    #define MT_WPLC   2
                    #define MT_DRIVE  3
                    #define MT_ERMA   4
                    #define MT_UNK    0xFF
              OF:   Optional Family
                    /* 2 == 205 */
                    /* 3 == 305 */
                    /* 4 == 405 */
                    /* 10 == Terminator */
              O1-X: Optional data
```

Type = 8 - FF
Format: Unassigned

Type 1 Reference table:

PLC Type	Len (n)	Type (tt)	ID (ii)	Discrete In (xx)	Discrete Out (yy)	Word In (kk)	Word Out (vv)	Description
205	0	1	0xFF	0	0	0	0	Empty Slot
205	4	1	0xFE	8	0	0	0	8 In Discrete
205	4	1	0xFD	0	8	0	0	8 Out Discrete
205	4	1	0xF7	4	4	0	0	4 In/4 Out Disc.
205	4	1	0xEF	4	0	0	0	4 In Discrete
205	4	1	0xDF	0	4	0	0	4 Out Discrete
205	4	1	0xBF	16	0	0	0	16 In Discrete
205	4	1	0x7F	0	16	0	0	16 Out Discrete
205	4	1	0xFC	8	8	0	0	8 In/8 Out Disc.
205	4	1	0x7E	32	0	0	0	32 In Discrete
205	4	1	0xF9	0	32	0	0	32 Out Discrete
205	4	1	0xFA	0	0	2	0	2 In Analog
205	4	1	0xF6	0	0	0	2	2 Out Analog
205	4	1	0x3F	0	0	0	2	2 Out Analog
205	4	1	0x3E	0	0	4	0	4 In Analog
205	4	1	0x3D	0	0	4	2	4 In/2 Out Anlg.
205	4	1	0x3B	0	0	8	0	8 In Analog
205	0	0xFF	0xFB	0	0	0	0	Counter I/F Mod.
205	0	0xFF	0xEE	0	0	0	0	Z-01DM
205	0	0xFF	0xDE	0	0	0	0	Z-02RM
205	0	0xFF	0xBE	0	0	0	0	Z-13RM

205	0	0xFF	0xEE	0	0	0	0	H2-ECOM & H2-ERM
205	4	5	Ox51	64	64	8	12	H2-CTRIO (v1.x) 8 In DWORD 12 Out DWORD
205	4	5	0x51	96	96	0	12	H2-CTRIO (v2.x) 8 In DWORD 12 Out DWORD
205	4	0xFF	0x50	0	0	0	0	H2-SERIO

TIO	4	7	0x11	8	0	0	0	8 In Discrete
TIO	4	7	0x11	16	0	0	0	16 In Discrete
TIO	4	7	0x12	0	8	0	0	8 Out Discrete
TIO	4	7	0x12	0	16	0	0	16 Out Discrete
TIO	4	7	0x25	0	0	0	0	8 In DWORD
TIO	4	7	0x26	0	8	0	0	8 Out DWORD
TIO	4	7	0x25	0	0	0	0	16 In DWORD
TIO	4	7	0x26	0	8	0	0	16 Out DWORD
TIO	4	7	0x38	64	64	8	12	T1H-CTRIO (V1.x) 8 In DWORD 12 Out DWORD
TIO	4	7	0x38	96	96	0	12	T1H-CTRIO (V2.x) 8 In DWORD 4 Out DWORD

405	4	1	0x81	8	0	0	0	8 In Discrete
405	4	1	0x82	16	0	0	0	16 In Discrete
405	4	1	0x84	32	0	0	0	32 In Discrete
405	4	1	0x87	64	0	0	0	64 In Discrete
405	4	1	0x90	0	8	0	0	8 Out Discrete
405	4	1	0xA0	0	16	0	0	16 Out Discrete
405	4	1	0xC0	0	32	0	0	32 Out Discrete
405	4	1	0xF0	0	64	0	0	64 Out Discrete
405	4	3	0x18	16	32	0	0	HSC Module Note: Also has 7 DWORDs (see HSC spec)
*405	4	1	0x89	0	0	4	0	D4-04AD
*405	4	1	0xA9	0	0	4	0	F4-04AD (16-Bit 2’s complement mode)
*405	4	1	0xB9	0	0	4	0	F4-04AD (16-Bit Mode)
*405	4	1	0x99	0	0	4	0	F4-04ADS
*405	4	1	0x8A	0	0	8	0	F4-08AD & F4-08THM-n
*405	4	1	0xC8	0	0	0	2	D4-02DA
*405	4	1	0xC9	0	0	0	4	F4-04DA
*405	4	1	0xD9	0	0	0	4	F4-04DA-1 & F4-04DA-2
*405	4	1	0xCA	0	0	0	8	F4-08DA-1
*405	4	1	0xCB	0	0	0	16	F4-16DA-1
405	4	3	0x1A	64	64	8	12	H4-CTRIO (v1.x) 8 In DWORD 12 Out DWORD
405	4	3	0x1A	96	96	0	12	H4-CTRIO (v2.x) 8 In DWORD 12 Out DWORD

* - Indicates cannot be detected by EBC (405 Analog modules cannot be detected, and must be configured with a call to HEIWriteBaseDef)

Function: Module status
Code: 2
Format: 02 nn [ll mm] ff ee ww ii nn

```
nn:   Length in bytes of status data (beginning with ff)
            if (nn==255)
                  ll mm is a two-byte length
            else
                  ll mm not included
ff:   Flags:
           76543210
              ||||
              |||+- Module error (ee)
              ||+-- Module warning (ww)
              |+--- Module info (ii)
              +---- Module internal (nn)
ee:   Module error value
ww:   Module warning value
ii:   Module info value
nn:   Module internal value
```

See HEI.H for a complete list of error values (HEIE_XXX).

Error/Warning/Info/Internal Values:	
Value:	Description:
117	Write attempted to an invalid analog channel.
121	Analog Input Channel failure; nn contains channel number that failed.
122	Unused analog input channels exist
139	Broken transmitter; nn contains channel number that failed.
142	Channel fail multiple; nn contains channel BITS from module. Ex: If bit 0 is set then channel 0 has failed If bit 1 and 3 are set then channels 1 and 3 have failed.
200-216	XX unused analog input channels exist where: XX = Value – 200.
> 32 (0x20) and < 64 (0x40) for 405 Family	BIT Type of Error 0 Terminal block off 1 External P/S voltage low 2 Fuse blown 3 Bus Error 4 Module init error (intelligent module) 5 Faults exist in module (this bit is set if any of the above bits are set) Example: 0x22: External P/S Voltage low

Function: Discrete input state data (block)
Code: 3
Format: 03 nn [ll mm] ss [2nd byte] [3rd byte] [nnth byte]

```
nn:   Length of data in bytes
            if (nn==255)
                  ll mm is a two-byte length
            else
                  ll mm not included

ss:   Data bytes
```

Function: Discrete output state data (block)
Code: 4
Format: 04 nn [ll mm] ss [2nd byte] [3rd byte] [nth byte]

```
nn:   Length of data in bytes
            if (nn==255)
                  ll mm is a two-byte length
            else
                  ll mm not included
ss:   Data bytes
```

Function: Word input state data (block)
Code: 5
Format: 05 nn [ll mm] wl wm [2nd word] [3rd word] [nth word]

```
nn:   Length of data in bytes
            if (nn==255)
                  ll mm is a two-byte length
            else
                  ll mm not included
wl:   Least significant byte
wm:   Most significant byte
```

Function: Word output state data (block)
Code: 6
Format: 06 nn [ll mm] wl wm [2nd word] [3rd word] [nth word]

```
nn:   Length of data in bytes
            if (nn==255)
                  ll mm is a two-byte length
            else
                  ll mm not included
wl:   Least significant byte
wm:   Most significant byte
```

Function: Base - Begins any base specific data.
Code: 7
Format: 07 bb [ll mm]

```
bb: Base number
            if (bb==255)
                  ll mm is a two-byte base number
            else
                  ll mm not included
```

Function: DWord input state data (block)
Code: 8
Format: 08 nn [ll mm] b0 b1 b2 b3 [2nd DWord] [3rd DWord] [nth DWord]

```
            nn:   Length of data in bytes
                        if (nn==255)
                              ll mm is a two-byte length
                        else
                              ll mm not included
            b0: Least significant byte of least significant word
            b1: Most significant byte of least significant word
            b2: Least significant byte of most significant word
            b3: Most significant byte of most significant word
```

Function: DWord output state data (block)
Code: 9
Format: 09 nn [ll mm] b0 b1 b2 b3 [2nd DWord] [3rd DWord] [nth DWord]

```
            nn:   Length of data in bytes
                        if (nn==255)
                              ll mm is a two-byte length
                        else
                              ll mm not included
            b0: Least significant byte of least significant word
            b1: Most significant byte of least significant word
            b2: Least significant byte of most significant word
            b3: Most significant byte of most significant word
```

Function: Offset given number of elements.
Currently only supported for 405 HSC Module and Hitachi Drive controller to offset to Specific DWord. Offset is given as 2 bytes and is an offset Of the given number of elements (i.e. DWord's)
Code: A
Format: 0A nn ll mm

```
            n: Number of bytes following code byte (2)
            ll: Least significant byte of offset
            mm: Most significant byte of offset
```

Function: New – style I/O write (When used as first byte of data packet)
Code: B
Format: BV nn

```
            V: Version of new style write (currently 1)
            nn: Number of bytes following (currently 0)
```

Function: Delay for the given number of 50 microsecond periods.
Code: D
Format: 0D 04 ll lm ml mm

```
            4:    Number of bytes following code byte (DWORD == 4 Bytes)
            ll:   Least significant word least significant byte
            lm:   Least significant word most significant byte
            ml:   Most significant word least significant byte
            mm:   Most significant word most significant byte
```

Function: Double input state data (block)
Code: 0x12
Format: 12 nn [ll mm] b0 b1 b2 b3 [2nd Float] [3rd Float] [nth Float]

```
            nn:   Length of data in bytes
                        if (nn==255)
                              ll mm is a two-byte length
                        else
```

```
                        ll mm not included
            b0: Least significant byte of least significant word
            b1: Most significant byte of least significant word
            b2: Least significant byte of most significant word
            b3: Most significant byte of most significant word
```

Function: Double output state data (block)
Code: 0x13
Format: 13 nn [ll mm] b0 b1 b2 b3 [2nd Float] [3rd Float] [nth Float]

```
            nn:   Length of data in bytes
                        if (nn==255)
                              ll mm is a two-byte length
                        else
                              ll mm not included
            b0: Least significant byte of least significant word
            b1: Most significant byte of least significant word
            b2: Least significant byte of most significant word
            b3: Most significant byte of most significant word
```

Function: Float input state data (block)
Code: 0x14
Format: 14 nn [ll mm] b0 b1 b2 b3 [2nd Float] [3rd Float] [nth Float]

```
            nn:   Length of data in bytes
                        if (nn==255)
                              ll mm is a two-byte length
                        else
                              ll mm not included
            b0: Least significant byte of least significant word
            b1: Most significant byte of least significant word
            b2: Least significant byte of most significant word
            b3: Most significant byte of most significant word
```

Function: Float output state data (block)
Code: 0x15
Format: 15 nn [ll mm] b0 b1 b2 b3 [2nd Float] [3rd Float] [nth Float]

```
            nn:   Length of data in bytes
                        if (nn==255)
                              ll mm is a two-byte length
                        else
                              ll mm not included
            b0: Least significant byte of least significant word
            b1: Most significant byte of least significant word
            b2: Least significant byte of most significant word
            b3: Most significant byte of most significant word
```

Function: Config data (block)
Code: 0x16
Format: 16 nn [ll mm] c0 .. cn

```
            nn:   Length of config data in bytes
                        if (nn==255)
                              ll mm is a two-byte length
                        else
                              ll mm not included
            c0-cn: Config Data Bytes
```

Function: End block
Code: 0xFF
Format: FF

NOTES:

Once a base has been selected with the base function code all subsequent codes apply to that base.

Base zero is assumed until a base function code has be issued.

Once a slot has been selected with the slot function code all subsequent codes apply to that slot.

Slot selection or codes within a slot are not order dependent.

Slots may be selected more than once.