Ethernet Interface Specification
Version 3.0

Last Update: 4-October-2008

Introduction

Win32 Version of our Ethernet SDK

The interface code as a package is written with the “less is more” philosophy. It is intended to be the minimum layer between the client application and the I/O interface module. As such, the client application has the responsibility of initialization, building up I/O requests, parsing responses, handling error conditions and cleanup when the client application closes. The overview section will detail the basics of establishing a connection with one of our Ethernet devices and exchanging data.

Being that our other software products are Win32 applications, the examples included in the Ethernet SDK are for Microsoft Visual C/C++ and for Microsoft Visual Basic. However, they should build under any 32-bit compiler with a few tweaks. As more people convert the SDK to other platforms, we'll make them available here. The demonstration programs are included to help you understand the process of communication with the EBCs, ECOMs and EDRVs.

We will be adding documentation files, example code, and other goodies to the SDK as we learn more about what is required to use our SDK efficiently. As developers of the products, we sometimes over-simplify some aspects of communicating with our modules. As we get questions from developers using our SDK, we will publish FAQs, tip sheets, tables of information, etc. that address these issues and try to head off some of the common problems.

Source Code for our Ethernet SDK

For professional software developers who, for legitimate reasons, cannot use the Ethernet SDK as provided, we will make the source code to the Ethernet SDK available. Some legitimate reasons are: you’re developing for a non-Windows environment (UNIX, Linux, etc.), you’re not using Microsoft’s Visual C/C++ development environment, or you want to include the HEI APIs directly into your application and not use the provided DLLs. We will provide this source code free of charge and will provide assistance in its use via email. Along with the Win32 version of the source code, we do have available two customer provided ports of the source code, one for Linux (RH v8.1) and one for VB.Net 2003. To request the source code for our Ethernet SDK, go to www.Hosteng.com and follow the “SDK Source Code” link on the main page.

This document and the products it represents are a work in progress. We desire your feedback, so if you have some constructive criticism or enhancement ideas, please don’t hesitate to send them in. The preferred method of communicating with us about the Ethernet SDK is via an email sent to SDKSupport@hosteng.com.

Overview

The basic components of an Ethernet connection to any of our Ethernet devices are the **protocol**, the **transport**, and the **device**. The transport is the mechanism the interface APIs use to connect to the underlying protocol. At this time the only supported Transport is WinSock. The protocol is an actual Ethernet protocol such as IPX or TCP/IP. For simplicity, the protocol and transport have been bundled together in the HEITransport structure. The device is the EBC, EBC100, ECOM, ECOM100 or EDRV itself and is represented by an HEIDevice structure.

Before you can interact with the I/O of an Ethernet device, you must create a connection to that device (well duh!). So how does one go about doing that? Generally speaking, it goes something like this:

1) In the client application's initialization routine, call HEIOpen () to initialize the Ethernet driver.

2) For each transport/protocol combination you wish to use, allocate an HEITransport structure, initialize the protocol and transport members, and call HEIOpenTransport ().

3) Initialize a variable pNumDevices to the length (in devices) of the HEIDevice structure array, then using the HEITransport you just opened, call HEIQueryDevices (). The array should be large enough to accommodate the largest number of device you wish to support. The function uses a broadcast to query for Ethernet devices. If any devices are found, pNumDevices is set to the number of devices found and the array is populated with the corresponding number of HEIDevice structures.

   A variation on the HEIQueryDevices is to use HEIQueryDeviceData and specify the value of one of the device's data fields (Module ID, Name, Description, or TCP/IP Address). This allows you to do runtime address resolution and eliminates the need to hardcode a physical device address into your client application.

4) Using the HEITransport and HEIDevice for the desired device call HEIOpenDevice () to open a connection with that EBC, ECOM or EDRV.

5) If the Ethernet device is an EBC or an EDRV, then you'll call HEIReadBaseDef to get the modules and I/O counts for the selected device. You'll then use HEIReadIO to get the I/O state of the base and HEIWriteIO to set the output state of the base (HEIWriteIO also returns the input state).

6) If the Ethernet device is an ECOM, you'll use the HEICCMRequest () and/or HEIKSeqRequest () APIs to read and write the I/O data in the PLC.

7) When you're done call HEICloseDevice () for each device your client application opened.

8) Call HEICloseTransport () to close the transport.

9) Call HEIClose () allow the Ethernet driver to clean up.

Initialization & Shutdown

System

The client application must call HEIOpen () once to initialize the Ethernet interface, and must call HEIClose () upon completion of Ethernet activity to allow the Ethernet system to shutdown and cleanup after itself.

Functions:

```
    int HEIOpen
    (
        WORD VersionNumber        // HEIAPIVERSION from Hei.h
    );

    int HEIClose
    (
    );
```

Errors:

Protocol

HEITransport Data Structure

```
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;
```

Transport 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:

Error codes from HEIOpenTransport are specific to the underlying transport. Winsock errors are in the 10000+ range. Their definitions and explanation are found in Winsock2.h (look for WSABASEERR). One of the more common errors seen is “10047 – WSAEAFNOSUPPORT Protocol not supported”, which typically means that the HEITransport structure is incorrectly configured or has become corrupt.

Device

HEIDevice Data Structure

```
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 MAC address is placed here in
                                        // the HEIQueryDevices call

        WORD RetryCount;                // Number of retrys which have occurred
        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;
```

HEIDevice Functions:

```
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 ms) for a transaction
        WORD iRetrys,                    // Number of times to retry a transaction
                                         // after a timeout
        BOOL UseAddressedBroadcast       // use addressed broadcast indicator
                                         // to talk to this device or not.
);

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

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,                     // Buffer containing value to query for
        WORD SizeofData                  // Size of data in buffer
);

DWORD HEISetQueryTimeout
(
        DWORD NewTimeout                 // New timeout value in milliseconds
);

DWORD HEIGetQueryTimeout
(
);
```

Errors:

Support Information

SupportDef Data Structure

```
typedef struct
{
        BYTE Version;
        BYTE Bytes2Follow;

#if defined (ANSI_C)
        BYTE UnusedBytess[16];                        // Support Info
#else
        BYTE SUP_FUN_POLLING                   : 1;   // Polling one base
        BYTE SUP_FUN_READ_VER_INFO             : 1;   // Version Info
        BYTE SUP_FUN_READ_SUPPORT_INFO         : 1;   // Support Info
        BYTE SUP_FUN_READ_DEVICE_INFO          : 1;   // Device Info
        BYTE SUP_FUN_POLLING_ALL               : 1;   // Polling all bases
                                                      // (returns Ethernet address)
        BYTE SUP_FUN_WRITE_IO                  : 1;   // Write IO base
        BYTE SUP_FUN_READ_IO                   : 1;   // Read IO Base
        BYTE SUP_FUN_READ_BASE_DEF             : 1;   // ReadBaseDef
        BYTE SUP_FUN_ENUM_SETUP_DATA           : 1;   // Enumerate setup data
        BYTE SUP_FUN_READ_SETUP_DATA           : 1;   // Read setup data
        BYTE SUP_FUN_WRITE_SETUP_DATA          : 1;   // Write setup data
        BYTE SUP_FUN_DELETE_SETUP_DATA         : 1;   // Delete setup data
        BYTE SUP_FUN_READ_ETHERNET_STATS       : 1;   // Read Ethernet statistics
        BYTE SUP_FUN_PET_LINK                  : 1;   // Used to keep the link sense
                                                      // timer from firing in the
                                                      // absence of HEIReadIO or
                                                      // HEIWriteIO messages
        BYTE SUP_FUN_ADDRESSED_BROADCAST: 1;          // Used to broadcast to a
                                                      // particular Ethernet address
        BYTE SUP_FUN_READ_MODULE_STATUS        : 1;   // Read module status
        BYTE SUP_FUN_EXTENDED                  : 1;   // Extended function
        BYTE SUP_FUN_QUERY_SETUP_DATA          : 1;   // Query for particular data
                                                      // type/value
        BYTE SUP_FUN_INIT_BASE_DEF             : 1;   // Initialize base def from
                                                      // backplane
        BYTE SUP_FUN_DATA_BROADCAST            : 1;   // Broadcast to a particular
                                                      // data type
        BYTE SUP_FUN_CCM_REQUEST               : 1;   // Perform CCM Request
        BYTE SUP_FUN_KSEQ_REQUEST              : 1;   // Perform KSEQ Request
        BYTE SUP_FUN_BACKPLANE_REQUEST         : 1;   // Perform backplane request
        BYTE SUP_FUN_WRITE_BASE_DEF            : 1;   // Write Base Def

        BYTE SUP_FUN_EXTEND_RESPONSE           : 1;   // Extends the response pack.
        BYTE SUP_FUN_ACK                       : 1;   // Acknowledge
        BYTE SUP_FUN_NAK                       : 1;   // NOT Acknowledge
        BYTE SUP_FUN_RESPONSE                  : 1;   // Response
        BYTE SUP_FUN_SERIAL_PORT               : 1;   // Execute serial port function
        BYTE SUP_FUN_WRITE_MEMORY              : 1;   // Write a particular mem type
        BYTE SUP_FUN_READ_MEMORY                : 1;    // Read a particular mem type
        BYTE SUP_FUN_ENUM_MEMORY                : 1;    // Get list of all memory types
        BYTE SUP_FUN_READ_SHARED_RAM            : 1;    // Read shared ram
        BYTE SUP_FUN_WRITE_SHARED_RAM           : 1;    // Write shared ram
        BYTE SUP_FUN_ACCESS_MEMORY              : 1;    // Access (read/write) multiple
                                                        // memory types
        BYTE SUP_FUN_COMM_RESPONSE              : 1;    // Response to PLC generated
                                                        // COMM request
        BYTE SUP_FUN_COMM_REQ_ACK               : 1;    // Function from PLC generated
                                                        // COMM request
        BYTE SUP_FUN_WRITE_IO_NO_READ           : 1;    // Write IO with no return read
        BYTE SUP_FUN_COMM_NO_REQ_ACK            : 1;    // Function from PLC generated
                                                        // COMM request
        BYTE SUP_FUN_RUN_PROGRAM                : 1;    // Function to run a program
        BYTE SUP_FUN_REMOTE_API                 : 1;    // Function to execute a
                                                        // function on remote device
        BYTE SUP_FUN_NOTIFY                     : 1;    // Indicates a notification
        BYTE SUP_FUN_COMPLETION                 : 1;    // Indicates completion of some
                                                        // activity
        BYTE SUP_FUN_SET_OS_LOAD                : 1;    // Set Load OS Parameter
        BYTE SUP_FUN_REBOOT                     : 1;    // Reboot OS
        BYTE SUP_FUN_EXTEND_RESPONSE_FIX        : 1;    // Fixed version that extends
                                                        // the response packet.
        BYTE SUP_NEW_STYLE_IO                   : 1;    // This device supports new
                                                        // style I/O requests
        BYTE SUP_HOT_SWAP                       : 1;    // True if this device supports
                                                        // hot swap
        BYTE SUP_TCP_IP                         : 1;    // TRUE if this device supports
                                                        // TCP/IP protocol
        BYTE SUP_HTTP                           : 1;    // TRUE if this device supports
                                                        // HTTP protocol
        BYTE Reserved                           : 6;

        BYTE UnusedBytes[9];                            // Unused
#endif

} SupportDef;
```

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

VersionDef and VersionInfoDef Data Structures

```
typedef struct
{
        BYTE MajorVersion;
        BYTE MinorVersion;
        WORD BuildVersion;

} VersionDef;

typedef struct
{
        BYTE SizeofVersionInfo;
        VersionDef BootVersion;
        VersionDef OSVersion;
        BYTE NumOSExtensions;
        VersionDef OSExt[10];

} VersionInfoDef;
```

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:

Functions for EBCs, EBC100s and EDRVs

Base Definition

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
);
```

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
);
```

Functions for handling Hot-Swap in Terminator I/O:

```
int HEIRescanBase
(
        HEIDevice *pDevice,                  // Device to Rescan the base
        DWORD    Flags,                      // Defines what to do with the RAM
                                             // copy of the I/O data
        BYTE    *pBaseDefInfo,               // Buffer to hold base def info
                                             // (see Appendix A)
        WORD      *pSizeOfBaseDefInfo        // Pointer to size of the buffer
);


int HEIInitBaseDef
(
        HEIDevice *pDevice,                  // Device to Rescan the base
        BYTE        *pBaseDefInfo,           // Buffer to hold base def info
                                             // (see Appendix A)
        WORD       *pSizeOfBaseDefInfo       // Pointer to size of the buffer
);
```

Errors:

Device Definition

Device Family and Type Defines

```
// Module type defines
#define MT_EBC          0           // Ethernet base controller
#define MT_ECOM         1           // Ethernet communications module
#define MT_WPLC         2           // WinPLC
#define MT_DRIVE        3           // Drive card
#define MT_ERMA         4           // Ethernet remote master
#define MT_CTRIO        5           // Counter I/O card
#define MT_AVG_DISP     6           // AVG Display Adapter card
#define MT_PBC          7           // Profibus controller
#define MT_PBCC         8           // Profibus IO coprocessor
#define MT_UNK          0xFF

// Module Family defines for MT_EBC, MT_ECOM, MT_WPLC, MT_ERMA
#define MF_005          0
#define MF_205          2
#define MF_305          3
#define MF_405          4
#define MF_TERM         10

// Module Family defines for MT_DRIVE
#define MF_100_SERIES   1           // Hitachi L100 and SJ100 drives
#define MF_J300         2           // Hitachi J300 drive
#define MF_300_SERIES   3           // Hitachi SJ300 drive
#define MF_GS           4           // GS Series drives  GS-EDRV

// Module Family defines for MT_AVG_DISP
#define MF_EZ_TOUCH     1           // AVG EZ-Touch Ethernet adapter
```

DeviceDef Data Structure

```
typedef struct
{
        BYTE PLCFamily;                         // See MF_XXX defines above
        BYTE Unused1;
        BYTE ModuleType;                        // See MT_XXX defines above
        BYTE StatusCode;
        BYTE EthernetAddress[6];                // Hardware Ethernet address
        WORD RamSize;                           // In K-Byte increments
        WORD FlashSize;                         // In K-Byte increments
        BYTE DIPSettings;                       // Settings of the 8 dip switches
        BYTE MediaType;                         // 0 = 10Base-T; 1 = 10Base-F
        DWORD EPFCount;                         // Early power fail count (405 EBC)

#if defined (ANSI_C)
        BYTE   Status;
#else
        BYTE RunRelay: 1;                       // 405 EBC Run Relay Status
        BYTE BattLow : 1;                       // 405 EBC Battery Low indicator
        BYTE UnusedBits: 6;                     // Unused status bits
#endif

        WORD BattRamSize;                       // Size in K-Bytes of battery-backed ram
        BYTE ExtraDIPS;                         // Extra Dip switches on Terminator EBC's
        BYTE ModelNumber;
        BYTE EtherSpeed;                        // 0=10MBit; 1=100MBit
        BYTE PLDRev[2];

        BYTE Unused[14];

} DeviceDef;
```

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

Errors:

EBC & EBC100 Configuration

Setup Name & Address

```
#define DT_IP_ADDRESS     0x0010          // 4 Byte IP address
                                          // (Used by PC application software)

#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)
```

Setup I/O Watchdog Timer

```
#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
                                          // HEIReadIO, HEIWriteIO, or HEIPetDevice call
                                          // before determining that the control program
                                          // has gone walk-about. It also defines what to
                                          // do with the outputs in this event.
typedef struct
{
        DWORD Timeout;                    // Timeout:
                                          //      Value in ms, 0 = disable 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;
```

Setup Encrypted Communications

```
#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;
```

H4-EBC Analog I/O configuration and Run Relay configuration

```
#define DT_BASE_DEF            0x0017  // 512 Byte Base Def (405 HEIWriteBaseDef)
#define DT_MODULE_SETUP        0x0024  // 64 Byte data from FLASH.
#define RRM_LINK_GOOD          0       // H4-EBC Run Relay ON when link is good
#define RRM_LINK_NOT_GOOD      1       // H4-EBC Run Relay ON when link is bad
#define RRM_POWERUP_ON         2       // H4-EBC Run Relay ON at power up
#define RRM_MANUAL_ON          3       // H4-EBC Run relay mode under user control

typedef struct
{
        BYTE RunRelayMode;                      // 405 EBC Run Relay Mode
        BYTE Unused[63];
} ModuleSetup;
```

Function for Reading:

```
        int HEIReadSetupData
        (
                HEIDevice *pDevice,             // Device to read from
                WORD SetupType,                 // Data type to read
                BYTE *pData,                    // Buffer to store setup data in
                WORD *pSizeofData               // Size of setup data buffer
        );
```

Function for Writing:

```
        int HEIWriteSetupData
        (
                HEIDevice *pDevice,             // Device to setup
                WORD SetupType,                 // Data type to setup
                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
        );
```

Function for enumerating:

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

I/O Module Configuration

The configuration 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 the configuration data using type DF_CONFIG

Read & Write I/O

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 from
        BYTE       *apData[],            // Array of pointers to data buffers
                                         // to hold the I/O data
                                         // (see Appendix A for I/O data format)
        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 for I/O Data format)
        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
);
```

```
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 (write data)
        BYTE      *apReturnData[],       // Array of pointers to data buffers for
                                         // incoming (read) data
        WORD      aSizeofReturnData[],   // Array of data buffer sizes (read data)
        int       aErrorCode[],          // Array of error codes
        Int       DeviceCount            // Number of devices in array
);

// Function for writing I/O to a single device but do not return the current I/O status

int HEIWriteIONoRead
(
        HEIDevice *pDevice,              // Device to write I/O to
        BYTE      *pData,                // Buffer for I/O data (see Appendix A)
        WORD      SizeofData             // Size of Buffer for I/O data
);
```

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

```
// Functions to read and write to devices that have shared RAM (e.g. CTRIO)

int HEIReadSharedRAM
(
        HEIDevice *pDevice,              // Device to read from
        WORD      Base,                  // Base the device is in
        WORD      Slot,                  // Slot number the device is in
        WORD      Address,               // Shared RAM address
        WORD      Bytes2Read,            // Number of BYTES to read
        BYTE      *pBuffer               // Buffer to place the data read
);

int HEIWriteSharedRAM
(
        HEIDevice *pDevice,              // Device to write to
        WORD      Base,                  // Base number the device in
        WORD      Slot,                  // Slot number the device is in
        WORD      Address,               // Shared RAM address
        WORD      Bytes2Write,           // Number of BYTEs to write
        BYTE      *pBuffer               // the data to write
);
```

EBC Serial Port Functionality

The Serial port on EBCs work in one of two modes (the mode is set via NetEdit v3):

- Slave Mode (Default) - the port expects a serial master device to send it commands that it will respond to - for example, an operator panel.
- Proxy Mode - the EBC expects to get commands over the Ethernet ports that it will relay out the serial port, it essentially make the EBCs serial port a remote serial port for the PC connecting over Ethernet.

Serial Port Functions

```
#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
```

Note: You can use either HEIWriteSetupData(pDevice, DT_SERIAL_SETUP, pData, SizeofData) or use HEIWriteSerialSetup / HEIWriteSerialSetupEx to configure the serial port on the EBC and EBC100.

SerialSetup Data Structure

```
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;    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;
```

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 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 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             // Number of bytes available to read
);

int HEIGetRXAvailableEx
(
        HEIDevice *pDevice,          // Device to use
        BYTE Port,                   // Port to get RX available for
        WORD *pAvailable             // 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
);


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

The EBC and EBC100 set aside some of their RAM to emulate some PLC memory. This memory is intended to be accessed by operator panels (such as a DV1000 or an Optimate panel). These memory functions are used to enumerate, read and write from the PC to theses blocks of PLC type memory in an EBC and EBC100. 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
#define MT_KOYO_SP       0x0120       // SP Memory

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

        } MemoryTypeDef;


        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
        );
```

```
#define ACCESS_READ       0
#define ACCESS_WRITE      1

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;


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

Miscellaneous

This function is used to keep the given device from firing the Link Sense watchdog in the absence of HEIReadIO or HEIWriteIO calls.

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

This function is used to obtain Ethernet statistics.

```
        typedef struct
        {
                WORD SizeofEthernetStats;               // Size of this structure
                DWORD MissedFrameCount;                 // Number of frames missed
                DWORD TransmitCollisionCount;           // Number of transmit collisions
                DWORD DiscardedPackets;                 // Number of packets received, but
                                                        // discarded.
        } EthernetStats;

        int HEIReadEthernetStats
        (
                HEIDevice *pDevice,                     // Device to get statistics from
                BYTE      *pData,                       // Buffer to hold statistics data
                WORD      *DataSize,                    // Pointer to size of buffer,
                                                        // returns number of bytes in buffer
                BOOL Clear                              // If TRUE, Ethernet statistics will be
                                                        // cleared after they are read.
        );
```

If a HEIReadIO or HEIWriteIO call returns an error, use this function to read the I/O module status data (broken transmitter alarms, blown fuses, missing 24V, etc.). This function call returns the I/O module status data for every 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 the size of the
                                                        // module status data
                BOOL Reset                              // If TRUE, module status will be reset
                                                        // after being read
        );
```

Functions for ECOMs and ECOM100s

Configuration

```
#define DT_RXWX_SETTINGS                 0x0015
#define DT_SETTINGS                      0x0015
```

HEISettings Data Structure

```
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;             // Max number of ms we’ve waited for an ACK
        WORD RXWXMaxRSPTime;             // Max number of ms we’ve waited for a
                                         // response
        DWORD RXWXACKRetrys;             // Number of retrys for an ACK
        DWORD RXWXRSPRetrys;             // Number of retrys for a response
        DWORD RXWXCompleted;             // Number of successfully completed
                                         // transactions
        DWORD RXWXTimeouts;              // Number of timeouts on transactions
                                         // (after retrys)
        DWORD RXWXOverruns;              // Number of times the PLC requested a
                                         // transaction while one was being processed
        DWORD RXWXErrors;                // Number of times an invalid code was
                                         // found or a transaction was NAK’d
        // 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;                 // Number of K-Sequence retrys
        WORD KSeqTimeouts;               // Number of K-Sequence timeouts

        BYTE Unused[81];                 // Reserved for future use

} HEISettings;
```

Note: Use HEIWriteSetupData(pDevice, DT_RXWX_SETTINGS, pData, SizeofData) to configure the ECOM and ECOM100.

Reading & Writing PLC Memory

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
                                    // if FALSE, we are reading data
        BYTE       DataType,        // Type of data to read / write
                                    // (see table below and DirectNET manual)
        WORD       Address,         // Address of data to read / write
                                    // (see table below and DirectNET manual)
        WORD        DataLen,        // Length of data to read / write
                                    // (see table below and DirectNET manual)
        BYTE        *pData          // Buffer for read / write data
                                    // (see table below and DirectNET manual)
);
```

DirectNET (CCM) Protocol Details

	Description	PLC Data Type	Qty (dec)	PLC V-Range (octal)	CCM Data Type (hex)	Length (bytes)	CCM Range (hex)
Input	Global I/O	GX	2048	V0000 – V3777	32	1	001 - 100
	Inputs	X	1024	V0000 – V1777	32	1	101 - 180
	SP Relays	SP	256	V0000 – V0377	32	1	181 - 200
Output	Global I/O	GY	2048	V0000 – V3777	33	1	001 - 100
	Outputs	Y	1024	V0000 – V1777	33	1	101 - 180
	Control Relays	C	2048	V0000 – V3777	33	1	181 - 280
	Stage Status	S	1024	V0000 – V1777	33	1	281 - 300
	Timer Status	T	256	V0000 – V0377	33	1	301 - 320
	Counter Status	CT	256	V0000 – V0377	33	1	321 - 340
Memory	V-memory	V	17056	V00000 – V41237	31	2	0001 - 42A0
	Ladder Program	L	131071	N/A	37	3	00000 - 1FFFF
	Scratchpad	Z	65535	N/A	36	1	0000 - FFFF

The K-Sequence protocol is a proprietary protocol for the Koyo/Automationdirect.com PLC line. We at Host Engineering cannot give you a copy of the K-Sequence protocol specification or publish any of its details. You must request a copy of this specification from AutomationDirect.com and in most cases they will ask you to sign an NDA before releasing it to you. Send an email to techbox@automationdirect.com to start the process.

There is really only one reason to implement a K-Sequence device driver and that is to provide the ability to write to a singe bit. The smallest amount of data the DirectNET (CCM) protocol can access is a BYTE, which means that if you want to access a single output bit with DirectNET, you have to write the full BYTE that contains the bit in question. Most people will employ this scheme with DirectNET like this:

1. Read the BYTE that contains the BIT you want access to.
2. Set the BIT in question to the desired state.
3. Write the BYTE back to the PLC.

There are some potential problems with doing things this way, but for most people it's an acceptable solution.

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 and/or output data
        WORD      *pDataLen        // Length of data returned
);
```

Appendix A

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: 0x00

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
```

NOTES:

- **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.**

Function: Module definition

Code: 0x01

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 = 00

Format: 01 00 00 FF

No additional data

Type = 01

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.

Note:

- **See the appropriate Type 1 reference tables below for the details about the specific I/O modules in your system.**

Type = 02

Format: Unassigned

Type = 03 && 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 = 03 && ID != 0x18
Format: Unassigned

Type = 04
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 = 05
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 = 06
Format: Unassigned

Type = 07
Format: 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
                        // 0 == 05/06
                        // 2 == 205
                        // 3 == 305
                        // 4 == 405
                        // 10 == Terminator
        O1-X:   Optional data
```

Type = 08 - FF
Format: Unassigned

Type 1 Reference table for DL205 I/O modules:

Len	Type	ID	DI	DO	WI	WO	DWI	DWO	I/O Description	AD.com Part #
0	1	0xFF	0	0	0	0	0	0	Empty Slot	<na>
4	1	0xFE	8	0	0	0	0	0	8 In Discrete	D2-08ND3 D2-08NA-1 D2-08NA-2 F2-08SIM
4	1	0xFD	0	8	0	0	0	0	8 Out Discrete	D2-08TD1 D2-08TD2 D2-08TA D2-08TR F2-08TA F2-08TR F2-08TRS
4	1	0xF7	8	8	0	0	0	0	4 In/4 Out Discrete	D2-08CDR
4	4	0xE7	16	16	0	0	0	0	16 In/16 Out Discrete (Prot)	F2-16TD1P F2-16TD2P
4	1	0xEF	8	0	0	0	0	0	4 In Discrete	<na>
4	1	0xDF	0	8	0	0	0	0	4 Out Discrete	D2-04TD1 D2-04TRS
4	1	0xBF	16	0	0	0	0	0	16 In Discrete	D2-16ND3-2 D2-16NA
4	1	0x7F	0	16	0	0	0	0	16 Out Discrete	D2-16TD1-2 D2-16TD2-2 D2-12TA D2-12TR
4	1	0xFC	8	8	0	0	0	0	8 In/8 Out Discrete	<na>
4	1	0x7E	32	0	0	0	0	0	32 In Discrete	D2-32ND3 D2-32ND3-2
4	1	0xF9	0	32	0	0	0	0	32 Out Discrete	D2-32TD1 D2-32TD2
4	1	0xFA	0	0	2	0	0	0	2 In Analog	<na>
4	1	0xF6	0	0	0	2	0	0	2 Out Analog	F2-02DA-1 F2-02DA-1L
4	1	0x3F	0	0	0	2	0	0	2 Out Analog	F2-02DA-2 F2-02DA-2L
4	1	0x4E	0	0	0	2	0	0	2 Out Analog	F2-02DAS-1 F2-02DAS-2
4	1	0x3C	0	0	4	0	0	0	4 In Analog	F2-04RTD F2-04THM
4	1	0x3E	0	0	4	0	0	0	4 In Analog	F2-04AD-1 F2-04AD-1L F2-04AD-2 F2-04AD-2L
4	1	0x3D	0	0	4	2	0	0	4 In/2 Out Analog	F2-4AD2DA

Len	Type	ID	DI	DO	WI	WO	DW I	DW O	I/O Description	AD.com Part #
4	1	0x3B	0	0	8	0	0	0	8 In Analog	F2-08AD-1 F2-08AD-2
4	1	0x4F	0	0	0	8	0	0	8 Out Analog	F2-08DA-1 F2-08DA-2
4	1	0x37	0	0	8	8 **	0	0	8 In/4 Out Analog	F2-8AD4DA-1 F2-8AD4DA-2
0	0xFF	0xFB	0	0	0	0	0	0	<not supported>	<na>
0	0xFF	0xEE	0	0	0	0	0	0	<not supported>	<na>
0	0xFF	0xDE	0	0	0	0	0	0	<not supported>	<na>
0	0xFF	0xBE	0	0	0	0	0	0	<not supported>	<na>
0	0xFF	0xEE	0	0	0	0	0	0	<not supported>	H2-ECOM H2-ECOM-F H2-ECOM100 H2-ERM H2-ERM-F
4	5	0x51	64	64	8	12	8	4	H2-CTRIO (v1.x)	H2-CTRIO
4	7	0x51	96	96	0	12	8	4	H2-CTRIO (v2.x)	H2-CTRIO
4	0xFF	0x50	0	0	0	0	0	0	H2-SERIO	H2-SERIO

** This module has 4 output channels W0-W3, the three configuration DWORDs are mapped into the next three outputs W4 – W6, W7 is unused.

Type 1 Reference table for DL405 I/O modules:

Len	Type	ID	DI	DO	WI	WO	DWI	DWO	I/O Description	AD.com Part #
4	1	0x81	8	0	0	0	0	0	8 In Discrete	D4-08ND3S D4-08NA F4-08NE3S
4	1	0x82	16	0	0	0	0	0	16 In Discrete	D4-16ND2 D4-16ND2F D4-16NA D4-16NA-1 D4-16NE3 D4-16SIM
4	1	0x84	32	0	0	0	0	0	32 In Discrete	D4-32ND3-1 D4-32ND3-2
4	1	0x87	64	0	0	0	0	0	64 In Discrete	D4-64ND2
4	1	0x90	0	8	0	0	0	0	8 Out Discrete	D4-08TD1 F4-08TD1S D4-08TA D4-08TR D4-08TRS-1 D4-08TRS-2
4	1	0xA0	0	16	0	0	0	0	16 Out Discrete	D4-16TD1 D4-16TD2 D4-16TA D4-16TR
4	1	0xC0	0	32	0	0	0	0	32 Out Discrete	D4-32TD1 D4-32TD1-1 D4-32TD2
4	1	0xF0	0	64	0	0	0	0	64 Out Discrete	D4-64TD1
4	3	0x18	16	32	0	0	** 7	** 7	High-Speed Counter	D4-HSC
4	1	0x89	0	0	4	0	0	0	* 4 In Analog	D4-04AD
4	1	0xA9	0	0	0	0	4	0	* 4 In Analog	F4-04AD 32
4	1	0xB9	0	0	4	0	0	0	* 4 In Analog	F4-04AD 16
4	1	0x99	0	0	4	0	0	0	* 4 In Analog	F4-04ADS
4	1	0x9A	0	0	8	0	0	0	* 8 In Analog	F4-08THM F4-08RTD
4	1	0x8A	0	0	8	0	0	0	* 8 In Analog	F4-08AD F4-08THM-n
4	1	0x8B	0	0	16	0	0	0	* 16 In Analog	F4-16AD-1 F4-16AD-2
4	1	0xC8	0	0	0	0	0	2	* 2 Out Analog	D4-02DA
4	1	0xC9	0	0	0	4	0	0	* 4 Out Analog	F4-04DA
4	1	0xD9	0	0	0	4	0	0	* 4 Out Analog	F4-04DA-1 F4-04DA-2
4	1	0xE9	0	0	0	4	0	0	* 4 Out Analog	F4-04DAS-1 F4-04DAS-2

Len	Type	ID	DI	DO	WI	WO	DWI	DWO	I/O Description	AD.com Part #
4	1	0xCA	0	0	0	8	0	0	* 8 Out Analog	F4-08DA-1 F4-08DA-2
4	1	0xCB	0	0	0	16	0	0	* 16 Out Analog	F4-16DA-1 F4-16DA-2
4	3	0x1A	64	64	8	12	8	4	H4-CTRIO (v1.x)	H4-CTRIO
4	3	0x1A	96	96	0	12	8	4	H4-CTRIO (v2.x)	H4-CTRIO

*** DL405 Analog I/O modules cannot automatically be detected, and must be configured with a call to HEIWriteBaseDef.**

**** 7 total R/W DWORDs (see HSC spec for details)**

Type 1 Reference table for Terminator I / O modules:

Len	Type	ID	DI	DO	WI	WO	DW I	DW O	I/O Description	AD.com Part #
4	7	0x11	8	0	0	0	0	0	8 In Discrete	T1K-08ND3 T1K-08NA-1
4	7	0x11	16	0	0	0	0	0	16 In Discrete	T1K-16ND3 T1K-16NA-1
4	7	0x12	0	8	0	0	0	0	8 Out Discrete	T1K-08TD1 T1K-08TD2-1 T1K-08TA T1K-08TAS T1K-08TR T1K-08TRS
4	7	0x12	0	16	0	0	0	0	16 Out Discrete	T1K-16TD1 T1K-16TD2-1 T1K-16TA T1K-16TR
4	7	0x25	0	0	0	0	8	0	8 In DWORD	T1F-08AD-1 T1F-08AD-1F T1F-08AD-2 T1F-08AD-2F
4	7	0x25	0	0	0	0	16	0	16 In DWORD	T1F-16AD-1 T1F-16AD-2 T1F-14THM T1F-16RTD
4	7	0x26	0	8	0	0	0	8	8 Out DWORD	T1F-08DA-1 T1F-08DA-2
4	7	0x26	0	8	0	0	0	16	16 Out DWORD	T1F-16DA-1 T1F-16DA-2
4	7	0x27	0	8	0	0	2	2	2 In/2 Out DWORD	<na>
4	7	0x27	0	8	0	0	4	2	4 In/2 Out DWORD	<na>
4	7	0x27	0	8	0	0	4	4	4 In/4 Out DWORD	<na>
4	7	0x27	0	8	0	0	8	4	8 In/4 Out DWORD	T1F-8AD4DA-1 T1F-8AD4DA-2
4	7	0x27	0	8	0	0	8	8	8 In/8 Out DWORD	T1F-8AD8DA
4	7	0x32	0	8	0	0	0	0	8 Out Discrete Iso	T1H-08TDS
4	7	0x38	96	96	0	12	8	4	T1H-CTRIO (v2.x)	T1H-CTRIO

Function: Module status

Code: 0x02

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
```

The following is a list of the values you'll most likely see. Please consult HEI.H for a complete list of error, warning, information and internal values (HEIE_XXX).

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. Example: If bit 1 and 3 are set, channels 1 and 3 have failed.
153	Terminator I/O Hot-Swap Error: Module Not Responding – a module that was logged into the system has been removed
154	Terminator I/O Hot-Swap Error: Base Changed – a module has been added to the system
200-216	XX unused analog input channels exist where: XX = Value – 200.
> 32 (0x20) and < 64 (0x40) for DL405 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: 0x03
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: 0x04
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: 0x05
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: 0x06
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: 0x07

Format: 07 bb [ll mm]

bb: Base number

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

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 been issued.**

Function: DWORD input state data (block)

Code: 0x08

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: 0x09

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 DL405 HSC Module (D4-HSC) and Hitachi Drive (HA-EDRV2) controller to offset to a specific DWORD. Offset is given as 2 bytes and is an offset of the given number of elements (i.e. DWORDs)

Code: 0x0A

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: 0xB

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: 0D

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