MCL PCI Card User Manual

Money Controls

Table of Contents

Revision History

Version	Date	Author	Description
0.0 - Draft	5th Feb 03	D. Bush A. Graham	Initial description document.
0.1 - Draft	16th Feb 03	D. Bush	Detail corrections (Bug Fixes)
0.2 - Draft	28th Feb 03	D. Bush	Changes to Coin Path handling
0.3 - Draft	10th Apr 03	D. Bush	Minor change to SystemStatus
0.4 - Draft	30th Apr 03	D Bush	Further Changes to Coin Path Handling
1.0	14th Oct 03	D Bush	Addition of Meters Various clarifications
1.1	24th Nov 03	D Bush	New Meter Functions Changes to details on dispensers
1.2	30th Jun 04	D Bush	Changed footer

Purpose of Document

This document describes the software interface to the AES Intelligent Money Handling Equipment Interface (IMHEI) as seen by a software engineer writing in either the C or C++ programming languages on the PC.

Intended Audience

The intended audience of this document is the software engineers who will be writing software on the PC that will communicate with the IMHEI card itself or will read the monetary information or diagnostic information provided by the card.

The functions provided are split into three sections, intended to reflect different levels of complexity at which the game programmer may wish to use the interface.

1. Getting Started

   These are the minimum set of “vanilla” functions that may be used to get a working *demonstration* program running.

   Using these calls alone; the software engineer can write a working program and get a feel for the ease with which he can now communicate with the Money Handling Equipment attached to his game.

   Apart from the money handling equipment, the card also supports a number of Indicators and Switches. Simple calls are provided to allow the software engineer to drive indicators and to interrogate switches.

   The switches are fully de-bounced and allow the games programmer to easily determine either the current *state* of a switch or to determine how many times the game player has *operated* the switch.

2. Full Game System

   These build on the set of functions provided within the “Getting Started” section.

   They add functionality that can determine the *status* of the peripherals attached to the interface card.

   By these status analysis functions, the game programmer could determine (say) the exact reason that an attempted payout failed and then notify either and engineer or a cash collector.

3. Engineering Support

   These functions provide full-blown diagnostics and reconfiguration facilities.

   They allow total reconfiguration of the card and its supported peripheral set, including to totally re-flash the microcontroller within the interface.

   It is envisaged that the *game software* will not use the facilities described here, but *engineering tools* may be written by the customer to allow aspects of the interface board to be changed.

Installation

The IMHEI card is a standard PCI interface card which has the normal Windows Plug 'n' Play automatic installation facilities.

When an interface card is detected in a PC the user is prompted to insert the installation CD. This CD will configure the system to use the card and copy into the system directories the two elements of the interface:

- The device driver: AESIMHEI.SYS
- The interface: AESIMHEI.DLL.

These provide all the software necessary to allow the user's program to access the money handling equipment.

Getting Started

This section describes those function calls that are provided to implement a minimum system. Using the functions described within this section, one can provide a fully working system, with credit and payout capability, as well as a number of indicators and switches.

What is not covered in this section is any error monitoring of the money handling equipment.

OpenMHE

Synopsis
This call is made by the PC application software to open the "Money Handling Equipment" Interface.

```
long OpenMHE (void);
```

Parameters
None

Return Value
If the Open call succeeds then the value zero is returned.

In the event of a failure one of the following standard windows error codes will be returned, either as a direct echo of a Windows API call failure, or to indicate internally detected failures that closely correspond to the quoted meanings.

Error Number	System message string for English decoding	Microsoft Mnemonic
20	The system cannot find the device specified.	ERROR_BAD_UNIT
21	The device is not ready.	ERROR_NOT_READY
87	The parameter is incorrect.	ERROR_INVALID_PARAMETER
170	The requested resource is in use.	ERROR_BUSY
1167	The device is not connected.	ERROR_DEVICE_NOT_CONNECTED
1200	The specified device name is invalid.	ERROR_BAD_DEVICE
1247	An attempt was made to perform an initialisation operation when initialisation has already been completed.	ERROR_ALREADY_INITIALIZED
1056	An instance of the service is already running.	ERROR_SERVICE_ALREADY_RUNNING

Remarks
1. Whereas an Open service normally requires a description of the item to be opened (and returns a reference to that Item) there is only one IMHE Interface unit in a system. Hence any "Open" call must refer to that single item.

2. Even following this call, all the money handling equipment will be *disabled* and rejecting all currency inserted until the successful execution of a call to **EnableInterface**.

EnableInterface

Synopsis
The **EnableInterface** call is used to allow users to enter coins or notes into the system. This would be called when a game is initialised and ready to accept credit.

```
void EnableInterface (void) ;
```

Parameters
None

Return Value
None

Remarks
1. This must be called following the call to **OpenMHE** before any coins / notes will be registered.

DisableInterface

Synopsis
The **DisableInterface** call is used to prevent users from entering any more coins or notes.

```
void DisableInterface (void) ;
```

Parameters
None

Return Value
None

Remarks
1. There is no guarantee that a coin or note can not be successfully read after this call has been made, a successful read may be in progress.

CurrentValue

Synopsis
Determine the current monetary value that has been accepted

The **CurrentValue** call is used to determine the total value of all coins and notes read by the money handling equipment connected to the interface.

```
long CurrentValue (void) ;
```

Parameters
None

Return Value
The current value, in the lowest denomination of the currency (i.e. cents / pence etc.) of all coins and notes read.

Remarks
1. The value returned by this call is never reset, but increments for the life of the interface card. Since this is a long (32 bit) integer, the card can accept £21,474,836.47 of credit before it runs into any rollover problems. This value is expected to exceed the life of the game.

2. It is the responsibility of the application to keep track of value that has been used up and to monitor for new coin / note insertions by increases in the returned value.

3. Note that this value should be read following the call to **OpenMHE** and before the call to **EnableInterface** to establish a starting point before any coins or notes are read.

PayOut

Synopsis
The **PayOut** call is used by the PC application to instruct the interface to pay out coins (or notes).

```
void PayOut (long Value) ;
```

Parameters
Value

This is the value, in the lowest denomination of the currency (i.e. cents / pence etc.) of the coins and notes to be paid out.

Return Value
None

Remarks
1. This function operates in value, not coins. It is the responsibility of the interface to decode this and to choose how many coins (or notes) to pay out, and from which device to pay them.

PayStatus

Synopsis
The PayStatus call provides the current status of the payout process.

```
long LastPayStatus (void) ;
```

Parameters
None

Return Values.

Value	Meaning	Mnemonic
0	The interface is in the process of paying out	PAY_ONGOING
1	The payout process is up to date	PAY_FINISHED
-1	The dispenser is empty	PAY_EMPTY
-2	The dispenser is jammed	PAY_JAMMED
-3	Dispenser non functional	PAY_US
-4	Dispenser shut down due to fraud attempt	PAY_FRAUD
-5	The dispenser is blocked	PAY_FAILED_BLOCKED
-6	No Dispenser matches amount to be paid	PAY_NO_HOPPER
-7	The dispenser is inhibited	PAY_INHIBITED

Remarks
Following a call to **PayOut**, the programmer should poll this to check the progress of the operation.

IndicatorOn / IndicatorOff

Synopsis

The IndicatorOn / IndicatorOff calls are used by the PC application to control LED's and indicator lamps connected to the interface.

```
void IndicatorOn  (long IndicatorNumber) ;
void IndicatorOff (long IndicatorNumber) ;
```

Parameters

IndicatorNumber

This is the number of the Lamp that is being controlled.

Return Value

None

Remarks

1. Although the interface is described in terms of lamps, any equipment at all may in fact be controlled by these calls, depending only on what is physically connected to the interface card.

SwitchOpens / SwitchCloses

Synopsis
The calls to **SwitchOpens** and **SwitchCloses** are made by the PC application to read the state of switches connected to the interface card.

```
long SwitchOpens  (long SwitchNumber) ;
long SwitchCloses (long SwitchNumber) ;
```

Parameters
SwitchNumber

This is the number of the switch that is being controlled.

In principle the interface card can support 64 switches, though note that not all of these may be physically present within a game cabinet.

Return Value
The number of times that the specified switch has been observed to open or to close, respectively.

Remarks
1. The value returned by this call is only (and always) reset by the OpenMHE call.

2. The convention is that at initialisation time all switches are open.

3. A switch that starts off closed will therefore return a value of 1 to a SwitchCloses call immediately following the OpenMHE call.

4. The expression (SwitchCloses(n) == SwitchOpens(n)) will always return 0 if the switch is currently closed and 1 if the switch is currently open.

5. The pressing / tapping of a switch by a user will be detected by an increment in the value returned by SwitchCloses or SwtichOpens.

6. The user only needs to monitor changes in one of the two functions (in the same way as most windowing interfaces only need to provide functions for button up or button down events).

Getting Started Code Examples

The following code fragments are intended to provide clear examples of how the calls to the IMHEI are designed to be used:

Currency Accept

```
void AcceptCurrencyExample(int NoOfChanges)
    {
    long LastCurrencyValue ;
    long NewCurrencyValue  ;

    long OpenStatus = OpenMHE() ;

    if (OpenStatus != 0)
        {
        printf("IMHEI open failed - %ld\n", OpenStatus) ;
        }
    else
        {
        // Then the open call was successful
        // Currency acceptance is currently disabled
        LastCurrencyValue = CurrentValue() ;

        printf("Initial currency accepted = %ld pence\n",
                                              LastCurrencyValue) ;
        EnableInterface() ;

        printf("Processing %d change events\n", NoOfChanges) ;
        while (NoOfChanges > 0)
            {
            Sleep(100) ;

            NewCurrencyValue = CurrentValue() ;
            if (NewCurrencyValue != LastCurrencyValue)
                {
                // More money has arrived (we do not care where from)
                printf("The user has just inserted %ld pence\n",
                            NewCurrencyValue - LastCurrencyValue) ;
                LastCurrencyValue = NewCurrencyValue ;
                --NoOfChanges ;
                }
            }
        }
    }
```

Currency Payout

```
void PayCoins(int NoOfCoins)
    {
    long OpenStatus = OpenMHE() ;

    if (OpenStatus != 0)
        {
        printf("IMHEI open failed - %ld\n", OpenStatus) ;
        }
    else
        {
        // Then the open call was successful
        // The interface is currently disabled
        EnableInterface() ;

        PayOut(NoOfCoins * 100) ;
        while (LastPayStatus() == 0)
           {}
        if (LastPayStatus() < 0)
           {
           printf("Error %d when paying %d coins\n",
                                    LastPayStatus(), NoOfCoins) ;
           }
        else
           {
           printf("%d coins paid out\n", NoOfCoins) ;
           }
        }
    }
```

Indicator Example

```
void LEDs(void)
   {
   long OpenStatus = OpenMHE() ;
   char Loop ;

   if (!OpenStatus)
      {
      EnableInterface() ;

      for (Loop = 0 ; Loop < 8 ; ++Loop)
         {
         IndicatorOn(Loop) ;
         Sleep(1000) ;
         }

      for (Loop = 0 ; Loop < 8 ; ++Loop)
         {
         IndicatorOff(Loop) ;
         Sleep(1000) ;
         }

      DisableInterface() ;
      }
   }
```

Switch Example

```
void LEDs(void)
   {
   long OpenStatus = OpenMHE() ;
   char Loop ;

   if (!OpenStatus)
      {
      EnableInterface() ;

      for (Loop = 0 ; Loop < 8 ; ++Loop)
         {
         printf("Switch %d is currently %s\n", Loop,
                  SwitchCloses(Loop) == SwitchOpens(Loop) ?
                  "Open" : "Closed") ;

         printf("It has closed %d times!\n", SwitchCloses(Loop)) ;
         }

      DisableInterface() ;
      }
   }
```

Full Game System

CurrentPaid

Synopsis
The CurrentPaid call is available to keep track of the total money paid out because of calls to the PayOut function.

```
long CurrentPaid (void) ;
```

Parameters
None

Return Value
The current value, in the lowest denomination of the currency (i.e. cents / pence etc.) of all coins and notes ever paid out.

Remarks

1. This value that is returned by this function is updated in real time, as the money handling equipment succeeds in dispensing coins.

2. The value that is returned by this call is never reset, but increments for the life of the interface card. It is the responsibility of the application to keep track of starting values and to monitor for new coin / note successful payments by increases in the returned value.

3. Note that this value can be read following the call to **OpenMHE** and before the call to **EnableInterface** to establish a starting point before any coins or notes are paid out.

SystemStatus

Synopsis
The **SystemStatus** call provides a single summary of the status all the money handling equipment connected to the interface. It is a logical OR of the status of all of the individual device statuses, together with the overall system.

```
long SystemStatus (void) ;
```

Parameters
None

Return Value
Zero if all devices are completely normal.

If anything is non normal bits from the three enumerations: SystemConstants, AcceptorConstants and DispenserConstants will be set.

Remarks
This returns a logical OR of the status of all of the individual device statuses.

AvailableValue

Synopsis
The **AvailableValue** call is available to keep track of how much money is available in the coin (or note) dispensers.

```
long AvailableValue (void) ;
```

Parameters
None

Return Value
The approximate minimum value, in the lowest denomination of the currency (i.e. cents / pence etc.) of all coins and notes that could be paid out.

Remarks
The accuracy of the value returned by this call is entirely dependent upon the accuracy of the information returned by the money dispensers.

If no information is obtainable, this returns 10,000 if at least one dispenser is working normally, and zero if all dispensers are failing to pay out.

ValueNeeded

Synopsis
The **ValueNeeded** call provides an interface to an optional credit card acceptor unit.

It is not envisaged that this would be used within many systems, but may be used, for example, in vending applications.

```
void ValueNeeded (long Amount) ;
```

Parameters
Amount

The figure that **CurrentValue** is required to reach before the next event can happen.

Return Value
None

Remarks
1. This function does not necessarily have any affect on the system. If the MHE includes a credit card acceptor, or similar, then the MHE interface unit will arrange for the next use of that unit to bring **CurrentValue** up to latest figure supplied by this routine.

2. If **CurrentValue** is greater or equal to the last supplied figure then any such acceptors are disabled.

ReadAcceptorDetails

Synopsis
The ReadAcceptorDetails call provides a snapshot of all the information possessed by the interface on a single unit of money handling equipment.

```
bool ReadAcceptorDetails (long           Number,
                          AcceptorBlock* Snapshot) ;
```

Parameters
1. Number
   The serial number of the coin or note acceptor about which information is required.
2. Snapshot
   A pointer to a program buffer into which all the information about the specified acceptor will be copied.

Return Value
True if the specified input device exists, False if the end of the list is reached.

Remarks
The serial numbers of the acceptors are contiguous and run from zero upwards.

WriteAcceptorDetails

Synopsis
The **WriteAcceptorDetails** call updates all the changeable information to the interface for a single unit of money accepting equipment.

```
void WriteAcceptorDetails (long           Number,
                           AcceptorBlock* Snapshot) ;
```

Parameters
1. Number
   The serial number of the coin or note acceptor being configured.
2. Snapshot
   A pointer to a program buffer containing the configuration data for the specified acceptor. See below for details.

Return Value
None.

Remarks
The serial numbers of the acceptors are contiguous and run from zero upwards.
A call to **ReadAcceptorDetails** followed by call to **WriteAcceptorDetails** for the same data will have no effect on the system.

ReadDispenserDetails

Synopsis
The **ReadDispenserDetails** call provides a snapshot of all the information possessed by the interface on a single unit of money dispensing equipment.

```
bool ReadDispenserDetails (long      Number,
                           DispenserBlock* Snapshot) ;
```

Parameters
1. Number
   The serial number of the coin or note dispenser about which information is required.
2. Snapshot
   A pointer to a program buffer into which all the information about the specified dispenser will be copied.

Return Value
True if the specified input device exists, False if the end of the list is reached.

Remarks
The serial numbers of the dispensers are contiguous and run from zero upwards.

WriteDispenserDetails

Synopsis
The **WriteDispenserDetails** call updates all the changeable information to the interface for a single unit of money handling equipment.

```
void WriteDispenserDetails (long           Number,
                            DispenserBlock* Snapshot) ;
```

Parameters
1. Number
   The serial number of the coin or note dispenser being configured.
2. Snapshot
   A pointer to a program buffer containing the configuration data for the specified dispenser. See below for details.

Return Value
None.

Remarks
The serial numbers of the dispensers are contiguous and run from zero upwards. A call to **ReadDispenserDetails** followed by call to WriteDispenserDetails for the same data will have no effect on the system.

‘C’ Program Structures and Constants

These definitions are not required for the simplest “Getting Started” level of use.
However, when implementing a full game implementation, these definitions will be used.
As with the prototypes and library files these will be provided as the SDK for the system.

System

```
enum SystemConstants
    {
    SYSTEM_MASK           = 0xf0000000,

    INTERFACE_FAILED      = 0x80000000
  } ;
```

AcceptorBlock

Constants for AcceptorBlock

```
enum AcceptorConstants
    {
    ACCEPTOR_MASK      = 0x0000ffff,
                           // No response to communications for this device

    ACCEPTOR_DEAD      = 0x00000001,
                           // No response from any device on this connection

    ACCEPTOR_ALL_DEAD  = 0x00000002,

    ACCEPTOR_DISABLED  = 0X00000004,    // Disabled by Interface
    ACCEPTOR_INHIBIT   = 0X00000008,    // Specific by Application
    ACCEPTOR_FRAUD     = 0X00000010,    // Reported from device

    MAX_ACCEPTOR_COINS = 256            // Maximum coins or notes
                                        // handled by any device
    } ;
```

Structures for AcceptorBlocks

```
typedef struct
    {
    long        Value ;               // Value of this coin
    long        Inhibit ;             // Set by PC: "this coin inhibited"
    long        Count ;               // Total number read "ever"
    long        Path ;                // Set to specify this coin chosen output path
    long        PathCount ;           // Number "ever" sent down the chosen Path
    long        PathSwitchLevel ;     // PathCount level to switch coin to default path
    long        DefaultPath ;         // Default path for this specific coin
    } AcceptorCoin ;

typedef struct
    {
    long           Unit ;             // Specification of this unit
    long           Status ;           // AcceptorStatuses - zero if device OK
    long           NoOfCoins ;        // The number of different coins handled
    long           InterfaceNumber ;  // The bus / connection
    long           UnitAddress ;      // For addressable units
    long           DefaultPath ;
    long           RejectCount ;      // Count of coins / notes rejected
    long           Currency ;         // Currency code reported
                                      // by an intelligent acceptor
    AcceptorCoin   Coin[MAX_ACCEPTOR_COINS] ;  // (only NoOfCoins are set up)
    } AcceptorBlock ;
```

DispenserBlock

Constants for DispenserBlock

```
enum DispenserConstants
    {
    DISPENSER_MASK         = 0x0fff0000,
    MAX_DISPENSERS         = 16           // Maximum handled
  } ;
```

Structures for DispenserBlock

```
typedef struct
    {
    long        Unit ;                  // Specification of this unit
    long        Status ;                // Dispenser Status - see PayStatus call
    long        InterfaceNumber ;       // The bus / connection
    long        UnitAddress ;           // For addressable units
    long        Value ;                 // The value of the coins in this dispensor
    long        Count ;                 // Number dispensed since interface commissioned
    long        Inhibit ;
    long        Currency ;              // The currency code reported by
                                        // an intelligent dispenser
    } DispenserBlock ;
```

Device Identity Constants

These constants are ORed together to form the coded device identity that can be extracted from the interface.

Example

As an example, a Money Controls Serial Compact Hopper 2 will have the following device code DP_MCL_SCH2, made up from:

- A device specifc code ORed with
- DP_COIN_PAYOUT_DEVICE ORed with
- DP_CCTALK_INTERFACE ORed with
- DP_MANU_MONEY_CONTROLS ORed with

This is a device code of **0x01020101**

```
enum GenericDevices
    {
    DP_GENERIC_MASK               =  0xff000000,

    DP_COIN_ACCEPT_DEVICE         =  0x02000000,
    DP_NOTE_ACCEPT_DEVICE         =  0x12000000,
    DP_CARD_ACCEPT_DEVICE         =  0x22000000,

    DP_COIN_PAYOUT_DEVICE         =  0x01000000,
    DP_NOTE_PAYOUT_DEVICE         =  0x11000000,
    DP_CARD_PAYOUT_DEVICE         =  0x21000000
    } ;

enum InterfaceNumbers
    {     // These describe the interface via which this device is connected:
    DP_INTERFACE_MASK             =  0x00ff0000,
    DP_INTERFACE_UNIT             =  0x00000000,
    DP_ONBOARD_PARALLEL_INTERFACE =  0x00010000,
    DP_CCTALK_INTERFACE           =  0x00020000,
    DP_SSP_INTERFACE              =  0x00030000,
    DP_HII_INTERFACE              =  0x00040000
    } ;

enum ManufacturerIdentities
    {                           // These describe the manufacturer of the device.
    DP_MANUFACTURER_MASK          =  0x0000ff00,
    DP_MANU_UNKNOWN               =  0x00000000,
    DP_MANU_MONEY_CONTROLS        =  0x00000100,
    DP_MANU_INNOVATIVE_TECH       =  0x00000200,
    DP_MANU_MARS_ELECTRONICS      =  0x00000300
    } ;
```

```
enum ManufacturerSpecificDeviceTypes
    {                           // These device types are manufacturer-dependent,
           //       so that each manufacturer can have up to 255 known devices.
    DP_SPECIFIC_DEVICE_MASK       =  0x000000ff,

    DP_PARALLEL_COIN               = 1 | DP_MANU_UNKNOWN
                                       | DP_ONBOARD_PARALLEL_INTERFACE
                                       | DP_COIN_ACCEPT_DEVICE,

    DP_PARALLEL_NOTE               = 1 | DP_MANU_UNKNOWN
                                       | DP_ONBOARD_PARALLEL_INTERFACE
                                       | DP_NOTE_ACCEPT_DEVICE,

                                                  // Money Controls Devices
    DP_MCL_SCH2                   =  1 | DP_MANU_MONEY_CONTROLS
                                       | DP_CCTALK_INTERFACE
                                       | DP_COIN_PAYOUT_DEVICE,

    DP_MCL_SR3                    =  2 | DP_MANU_MONEY_CONTROLS
                                       | DP_ONBOARD_PARALLEL_INTERFACE
                                       | DP_COIN_ACCEPT_DEVICE,

    DP_MCL_SR5                    =  3 | DP_MANU_MONEY_CONTROLS
                                       | DP_CCTALK_INTERFACE
                                       | DP_COIN_ACCEPT_DEVICE,

    DP_MCL_SR5R                   =  4 | DP_MANU_MONEY_CONTROLS
                                       | DP_CCTALK_INTERFACE
                                       | DP_COIN_ACCEPT_DEVICE,

    DP_MCL_LUMINA                 =  5 | DP_MANU_MONEY_CONTROLS
                                       | DP_CCTALK_INTERFACE
                                       | DP_NOTE_ACCEPT_DEVICE,

                                                 // Innovative Technology Devices
     DP_ITEK_NV7                  =  1 | DP_MANU_INNOVATIVE_TECH
                                       | DP_SSP_INTERFACE
                                       | DP_NOTE_ACCEPT_DEVICE,

                                                  // Mars Electronics Devices
    DP_MARS_CASHFLOW_126          =  1 | DP_MANU_MARS_ELECTRONICS
                                       | DP_HII_INTERFACE
                                       | DP_COIN_ACCEPT_DEVICE,

    DP_MARS_CASHFLOW_9500         =  2 | DP_MANU_MARS_ELECTRONICS
                                       | DP_HII_INTERFACE
                                       | DP_COIN_ACCEPT_DEVICE
    } ;
```

Coin (Note) Routing.

The technique for routing coins is not necessarily obvious. The design is based around the idea of one or more cash boxes, with particular coins being routed to other destinations (probably dispensers) if the dispenser is not full.

For the acceptor as a whole, the default destination (**`Acceptor.DefaultPath`**) is set up to the main cash box; either before installation, or by the application. For each coin, in addition, a separate default destination (**`Coin.DefaultPath`**) can be set up to indicate a separate cash box for that coin. If this is left as / set to zero then the acceptor wide default is used.

For each coin that requires special handling, a specific destination (**`Coin.Path`**) is then set up. (This is the route to use to send the coin to the dispenser)

Associated with each coin is an (interface maintained) count of the total number of instances of the coin that have ever gone down that specific path (**`Coin.PathCount`**). This number is undisturbed over changes in the value of the specific path - i.e. it is related only to the coin, not to the path.

For each coin, a level (**`Coin.PathSwitchLevel`**) is available, at which the coin will be routed to its default path. At interface initialisation this is zero for each coin, i.e. they will all be routed to the default destination.

The basic algorithm for applications is to set the specific path for each “payout” coin to the route that will take it to its dispenser and then detect, by operator input, that the dispenser is full.

At this point, the level (**`Coin.PathSwitchLevel)`** is set to the current path count (**`Coin.PathCount`**). From then on, whenever coin(s) are paid, the application increments the level (**`Coin.PathSwitchLevel)`** by the number of coins paid out. (This number is available in the dispenser detail field **`Dispenser.Count`**) The interface will, consequently, send coins to the dispenser until it is again full and then automatically switch to the cash box, with no further input from the application.

Note that the value(s) for **`Coin.PathSwitchLevel`** has to be preserved by the application.

Meters

The IMHEI card will support the concept of external meters that are accessible from the outside of the PC system.

In keeping with the IMHEI concept, an interface is defined to an idealised meter. This will be implemented transparently by the card using the available hardware. Initially the IMHEI will support a ***Starpoint Electronic Counter***, although other hardware may be supported at a later date.

CounterIncrement

Synopsis

The **CounterIncrement** call is made by the PC application software to increment a specific counter value.

```
void CounterIncrement(unsigned char  CounterNo,
                      unsigned short Increment);
```

Parameters

1. CounterNo
   This is the number of the counter to be incremented.

2. Increment
   This is the value to be added to the specified counter.

Return Value

None

Remarks

1. If the counter specified is higher than the highest supported, then the call is silently ignored.

CounterCaption

Synopsis
The **CounterCaption** call is used to associate a caption with the specified counter. This is related to the **CounterDisplay** call described below.

```
void CounterCaption(unsigned char CounterNo,
                    char*         Caption);
```

Parameters
1. CounterNo
   This is the number of the counter to be associated with the caption.
2. Caption
   This is an ASCII string that will be associated with the counter.

Return Value
None

Remarks
1. The meter hardware may have limited display capability. It is the system designer's responsibility to use captions that are within the meter hardware's capabilities.
2. If the counter specified is higher than the highest supported, then the call is silently ignored.
3. The specified caption is **not** stored in the meter, even if the meter offers this facility.

CounterRead

Synopsis
The **CounterRead** call is made by the PC application software to obtain a specific counter value as stored by the meter interface.

```
long CounterRead(unsigned char  CounterNo);
```

Parameters
1. CounterNo
   This is the number of the counter to be incremented.

Return Value
The Value of the specified meter at system start-up.

Remarks
1. If the counter specified is higher than the highest supported, then the call returns -1

2. If the counter external hardware does not support counter read-out, then this will return the total of all increments since PC start-up.
3. If error conditions prevent the meter updating, this call will show the value it **should** be at, not its actual value. (The value is read only read from the meter at system start-up.)

ReadCounterCaption

Synopsis
The Read**CounterCaption** call is used to determine the caption for the specified counter

```
char* CounterCaption(unsigned char CounterNo);
```

Parameters
1. CounterNo
   This is the number of the counter to be incremented.

Return Value
None

Remarks
1. If the counter specified is higher than the highest supported, then the call returns an empty string (“”).
2. All captions stored in the meter are read out at system start-up and used to initialise the captions used by the interface.

CounterDisplay

Synopsis
The **CounterDisplay** call is used to control what is displayed on the meter.

```
void CounterDisplay (long DisplayCode) ;
```

Parameters
1. DisplayCode
   If positive, this specifies the counter that will be continuously display by the meter hardware.

   If negative, then the display will cycle between the caption (if set) for the specified counter for 1 second, followed by its value for 2 seconds.

Return Value
None

Remarks
1. This result of this call with a negative parameter is undefined if no counters have an associated caption.
2. Whenever the meter displayed is changed, the caption (if set) is always displayed for one second.

MeterStatus

Synopsis
The **MeterStatus** call is used determine whether working meter equipment is connected.

```
long MeterStatus (void);
```

Parameters
None

Return Value
One of the following:

Value	Meaning	Mnemonic
0	A Meter is present and working correctly	METER_OK
1	No Meter has ever been found	METER_MISSING
2	The Meter is no longer functioning	METER_DIED
3	The Meter is functioning, but is itself reporting internal problems	METER_FAILED

Remarks
None

MeterSerialNo

Synopsis
The **MeterSerialNo** call is used determine which item meter equipment is connected.

```
long MeterSerialNo ( void );
```

Parameters
None

Return Value
The 32-bit serial number retrieved from the meter equipment.

Remarks
1. Where the meter equipment is not present or does not have serial number capabilities, zero is returned.

Engineering Support

It is not envisaged that games programmers will use these particular functions.

They are included here for completeness, but can be ignored if you are just interfacing game software to a collection of standard peripherals.

WriteInterfaceBlock

Synopsis
The **WriteInterfaceBlock** call sends a "raw" block to the specified interface.

There is no guarantee as to when, in relation to this, regular polling sequences will be sent, except that while the system is *disabled*, the interface card will not put any traffic onto the interface.

```
void WriteInterfaceBlock (long  Interface,
                          char*  Block,
                          long   Length) ;
```

Parameters
1. Interface

   The serial number of the interface that is being accessed.

2. Block

   A pointer to program buffer with a raw message for the interface. This must be a sequence of bytes, and must have any checksums and addresses required by the peripheral device included.

3. Length

   The number of bytes in the message.

Return Value
None

Remarks
Using this function with some interfaces does not make sense, see status returns from **ReadInterfaceBlock**.

ReadInterfaceBlock.

Synopsis

The **ReadInterfaceBlock** call reads the "raw" response to a single **WriteInterfaceBlock**.

```
long ReadInterfaceBlock (long   Interface,
                         char*  Block,
                         long   Length) ;
```

Parameters

1. Interface
   The serial number of the interface being accessed

2. Block
   A pointer to the program buffer into which any response is read.

3. Length
   The space available in the program buffer.

Return Values

- 3	Non command oriented interface (the corresponding **WriteInterfaceBlock** was ignored)
- 2	Command buffer overflow (the corresponding **WriteInterfaceBlock** was ignored)
- 1	Timeout on the interface - no response occurred (The interface will be reset if possible)
0	The response from the **WriteInterfaceBlock** has not yet been received
> 0	Normal successful response - the number of bytes received and placed into the buffer.

Remarks

1. Repeated calls to **WriteInterfaceBlock** without a successful response are not guaranteed not to overflow internal buffers.

2. The program is expected to "poll" the interface for a response, indicated by a non-zero return value.

This manual is intended only to assist the reader in the use of this product and therefore Money Controls shall not be liable for any loss or damage whatsoever arising form the use of any information or particulars in, or any incorrect use of the product. Money Controls reserve the right to change product specifications on any item without prior notice