

Milan / Paylink

Applications Interface

User Manual

This document is the property of Aardvark Embedded Solutions Ltd and may not be reproduced in part or in total by any means, electronic or otherwise, without the written permission of Aardvark Embedded Solutions Ltd. Aardvark Embedded Solutions Ltd does not accept liability for any errors or omissions contained within this document. Aardvark Embedded Solutions Ltd shall not incur any penalties arising out of the adherence to, interpretation of, or reliance on, this standard. Aardvark Embedded Solutions Ltd will provide full support for this product when used as described within this document. Use in applications not covered or outside the scope of this document may not be supported. Aardvark Embedded Solutions Ltd. reserves the right to amend, improve or change the product referred to within this document or the document itself at any time.

Table of Contents

Table of Contents	2
Revision History	4
Introduction	5
Purpose of Document.....	5
Intended Audience.....	5
Document Layout	5
Getting Started	7
Installation	7
Operation.....	8
OpenMHE.....	9
OpenSpecificMHE	10
EnableInterface	11
DisableInterface.....	11
CurrentValue	12
PayOut.....	13
PayStatus	13
CurrentPaid	14
IndicatorOn / IndicatorOff	14
SwitchOpens / SwitchCloses.....	15
Getting Started Code Examples.....	16
Currency Accept	16
Currency Payout	17
Indicator Example	18
Switch Example	18
Full Game System	19
Background	19
'C' Program Structures and Constants.....	19
AcceptorBlock.....	20
DispenserBlock.....	21
Device Identity Constants	22
CurrentUpdates (1.10.4).....	23
ReadAcceptorDetails.....	24
WriteAcceptorDetails	24
ReadDispenserDetails.....	25
WriteDispenserDetails.....	25
Dispenser Value Reassignment (1.10.7)	26
Token Handling (Coin Ids) (1.11.x)	26
Dual Currency Handling (Coin Ids) (1.11.x)	26
Read out of Acceptor Details(1.11.x)	27
Read out of Dispenser Details(1.11.x)	27
Coin (Note) Routing.....	28
Route coins to a general cash box	28
Route specific coins to a specific cash box.	28
Route coins to a dispenser until it is full then route it to a coin cash box.	28
Paylink Routing - Flow Diagram	29
MDB changer support. (1.10.x)	30
MDB tube level monitoring.....	31
Dispenser Power Fail support. (1.10.x).....	32
Combi Hopper Support. (1.10.x)	32
Multiple Paylink Unit Support.	33
Overview	33
Unit Identification	33
Utility Functions.....	34
CheckOperation (1.11.x)	34

CONFIDENTIAL

Not to be disclosed without prior written permission from Aardvark Embedded Solutions Ltd

NextEvent	35
AvailableValue	36
ValueNeeded	36
SetDeviceKey	37
SerialNumber	37
FirmwareVersion	38
USBDriverStatus	38
USBDriverExit	39
IMHEIConsistencyError	40
Auditing / Event Processing	41
Structure for EventDetailBlock	41
Event Codes for NextEvent / EventDetailBlock	41
cctalk coin processing	43
cctalk note processing	44
ID-003 note processing	45
Note Reader Escrow	46
EscrowEnable	46
EscrowDisable	46
EscrowThroughput	47
EscrowAccept	47
EscrowReturn	48
Escrow system usage	48
Meters / Counters	49
CounterIncrement	49
CounterCaption	49
CounterRead	50
ReadCounterCaption	50
CounterDisplay	51
MeterStatus	51
MeterSerialNo	52
E ² Prom	53
E2PromReset	53
E2PromWrite	54
E2PromRead	54
Bar Codes	55
Barcode Reading	55
BarcodeEnable	55
BarcodeDisable	56
BarcodeInEscrow	57
BarcodeStacked	57
BarcodeAccept	58
BarcodeReturn	58
Barcode Printing	59
BarcodePrint	59
BarcodePrintStatus	60
Engineering Support	61
WriteInterfaceBlock	61
ReadInterfaceBlock	62
Disclaimer	63

Revision History

Version	Date	Author	Description
0.0 - Draft	5 th Feb 03	D. Bush A. Graham	Initial description document.
0.1 - Draft	16 th Feb 03	D. Bush	Detail corrections (Bug Fixes)
0.2 - Draft	28 th Feb 03	D. Bush	Changes to Coin Path handling
0.3 - Draft	10 th Apr 03	D. Bush	Minor change to SystemStatus
0.4 - Draft	30 th Apr 03	D Bush	Further Changes to Coin Path Handling
1.0	14 th Oct 03	D Bush	Addition of Meters Various clarifications
1.1	24 th Nov 03	D Bush	New Meter Functions Changes to details on dispensers
1.2	3 rd Dec 03	D Bush	New E2Prom Functions
1.3	2 nd Apr 04	D. Bush	Various Bug Fixes - new constants
1.4	9 th Aug 05	D Bush.	Sections added on: <ul style="list-style-type: none">• Escrow functions.• Event Queue• Barcode functions
1.5	8 th Mar 06	D Bush	Document structure revised Added a number of Usage Details sections 1.10.x Functions detailed
1.6	13 th Nov 06	A Tainsh	Rewritten the Coin Routing description
1.7	11 th Sep 07	D Bush	Added description of multiple unit support.

Introduction

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.

Document Layout

The document itself is split into a number of sections. Within each section, there are three sections.

- **Operational Overview.**
Where the way in which this area is intended to work is explained.
- **Function Definitions.**
Where you will find exact details on each function call.
- **Usage Details.**
This gives details on exactly how the IMHEI system operates.

The first two sections are intended to reflect different levels of complexity at which an initial 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.

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 an engineer or a cash collector.

3. Utility Functions

These miscellaneous functions are concerned with the administration of the game system.

4. Note Reader Escrow

Here you will find functions that enable the escrow feature provided by note acceptors to be easily used.

5. Meters / Counters

This section is concerned with the support of the SEC meter, a small external unit that allow audit numbers to be maintained

6. E2Prom

The IMHEI units incorporate E²Prom storage for internal configuration storage. Some of this is made available to the PC programmer.

7. Barcode Reading

Here you will find functions that enable the barcoded ticket features provided by some note acceptors to be easily used.

8. Barcode Printing

These functions are used by the IMHEI units to support a "Ticket Printer" which will produce barcoded tickets.

9. Engineering Support

These functions provide full-blown diagnostics and reconfiguration facilities.

Getting Started

Installation

The Genoa / Paylink unit is a standard USB 1.1 peripheral. Installation of the OS driver is as with any USB peripheral, when the unit is detected the user is prompted to insert the installation CD. This CD will install the low level drivers, which may be removed using the standard "Add or Remove Programs" facility for the "AES Genoa USB Drivers" entry.

In addition, two other steps need to be undertaken, at present manually:

- The interface AESIMHEI.DLL needs to be copied from the installation CD to Windows\System32
- The High Level driver program needs to be copied from the installation CD to a convenient folder, and an entry made in the Startup folder to run this at system boot.

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.

Operation

The Milan unit contains an embedded processor that is responsible for all communication with the peripherals.

It handles the *event* based protocols, and uses the results to update a set of *state* tables.

The underlying concept behind the state tables is that the all activity causes counters to be incremented. The application programmer reads out the totals at the time the application starts, and then compares these with the current totals. Peripheral activity will cause these totals to increment, subtracting the old, saved value from the current value enables the application to determine the value inserted by the customer.

Using state tables on the PC in this way allows the programmer to be unconcerned with hardware response times. Although the *state* tables have to be periodically examined to see if anything has changed, there is never any requirement that this is done quickly, and the programmer does not have to be concerned that the OS may suspend his program for significant periods. Regardless of how long the program spends between examinations, the system will function perfectly and no money insertion or payout will be missed.

The following function calls 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.

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	Suggested string for English decoding	Microsoft Mnemonic	Retry
13	The DLL, application or device are at incompatible revision levels.	ERROR_INVALID_DATA	No
20	The system cannot find the device specified.	ERROR_BAD_UNIT	No
21	The device is not ready.	ERROR_NOT_READY	Yes
31	Driver program not running. or No PCI card in system.	ERROR_GEN_FAILURE	Yes
170	The USB link is in use.	ERROR_BUSY	Yes
1167	The device is not connected.	ERROR_DEVICE_NOT_CONNECTED	Yes

Remarks

1. With a USB system, there is a noticeable time for the USB communications to start. This may cause error returns labelled “Yes” under Retry in the above table. This indicates that the call to **OpenMHE** should be retried periodically until it has failed for at least 5 consecutive seconds before deciding that the interface is actually inoperative.
2. 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.
3. 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**.

OpenSpecificMHE

Synopsis

This call is made by the PC application software to open or to switch to one of the multiple “Money Handling Equipment” Interfaces installed on the PC.

Details on how a system works with multiple Paylinks are given in a later section.

long OpenSpecificMHE (char SerialNo[8]);

Parameters

None

Return Value

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

In the event of a failure the same standard windows error codes are returned as for **OpenMHE**.

Remarks

1. Every Paylink requires a unique instance of the USB driver program to be running. If there is no driver for the Paylink whose Serial Number is quoted, then the function returns 31 (ERROR_GEN_FAILURE).
2. As the default serial number for Paylink unit is “AE000001”, the **OpenMHE** call is equivalent to the call **OpenSpecificMHE("AE000001")**
3. This call may be issued repeatedly with no ill effects. Each call will serve to swap all the other calls in this document to the specified unit.

EnableInterface

Synopsis

The **EnableInterface** call is used to allow “turn on” the IMHE. This would be called when a game is initialised and ready. Until this call is made, no acceptors will accept credit.

void EnableInterface (void) ;

Parameters

None

Return Value

None

Remarks

1. Normally the application will initialise the saved values of all the information it is monitoring before this call.
2. 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
-8	The internal self checks failed	PAY_SECURITY_FAIL

Remarks

1. Following a call to **PayOut**, the programmer should poll this to check the progress of the operation.
2. If one out of multiple hoppers has a problem, the PCI card will do the best it can. If it can not pay out the entire amount, the status will reflect the last attempt.

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.

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 API can support 64 switches, though note that not all of these may be support by any particular hardware unit.

Return Value

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

Remarks

1. The convention is that at initialisation time all switches are open, a switch that starts off closed will therefore return a value of 1 to a SwitchCloses call immediately following the OpenMHE call.
2. The expression (SwitchCloses(n) == SwitchOpens(n)) will always return 0 if the switch is currently closed and 1 if the switch is currently open.
3. Repeat pressing / tapping of a switch by a user will be detected by an increment in the value returned by SwitchCloses or SwitchOpens.
4. 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)
5. The inputs are debounced. The unit reads all 16 inputs every 2 milliseconds. If we detect a change, we then require the next two reads to give exactly the same pattern before reporting the change. This means that a simple "electronic" input change will be reported between 4 and 6 milliseconds of it occurring.

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:

Each function will provide the central processing for a small command line demonstration program.

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

Background

When implementing a full game implementation, tighter control over the behaviour and response of the individual acceptors and hoppers is frequently necessary, for such purposes as routing coins to hoppers and cashboxes and emptying hoppers. Some more details on these operations are given at the end of this section.

The data retrieval functionality is achieved by reading the control blocks for the acceptors (with **ReadAcceptorDetails**) and possibly hoppers (with **ReadDispenserDetails**) at initialisation time and then continually checking the current contents of these against saved copies. To aid in this process the **CurrentUpdates** function guarantees that; if it returns an unchanged value then none of the control blocks will have changed.

Most of the control functionality is achieved by reading a data structure from the API, modifying it as appropriate or necessary and writing it back. Four functions are involved: **ReadAcceptorDetails**, **ReadDispenserDetails**, **WriteAcceptorDetails** & **WriteDispenserDetails**.

All these functions identify the individual units by a serial number, in the range 0...N-1. The programmer should not assume that any particular unit is present at any particular number, the numbers are assigned dynamically and are liable to change from run to run.

To find the particular unit of interest, the programmer should scan number from 0 up, looking for a match on the structure members.

For an acceptor, this will usually involve the **unit** field. Although this is defined as single 32 bit number, it is created by concatenating four 8 bit values. The program will usually only be interested in distinguishing the coin and note acceptors, which are distinguished by values in the top 8 bits. For this purpose two 'C' macros are defined, **IS_COIN_ACCEPTOR(unit)** and **IS_NOTE_ACCEPTOR(unit)**, see below, which can easily be translated into other languages.

For a dispenser, this will normally involve the **value** as that shows the coin value assumed by Milan interface, which is the most important distinguishing feature of a dispenser.

'C' Program Structures and Constants

This is currently an extract of the 'C' header file, the comments should serve to define the various fields, we hope to enhance the format of this section of the document in the near future.

For people unfamiliar with 'C', a data item prefixed with **long** is a 32 bit integer, a data item prefixed with **char** is an 8 bit integer.

AcceptorBlock

Constants for AcceptorBlock

```
enum AcceptorConstants
{
    ACCEPTOR_DEAD      = 0x00000001,    /* No response to communications for this device */
    ACCEPTOR_DISABLED  = 0x00000004,    /* Disabled by Interface */
    ACCEPTOR_INHIBIT    = 0x00000008,    /* Specific by Application */
    ACCEPTOR_FRAUD      = 0x00000010,    /* Reported from device */
    ACCEPTOR_BUSY       = 0x00000020,    /* Reported from device */
    ACCEPTOR_FAULT      = 0x00000040,    /* 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 is inhibited
    long      Count ;               /* Total number read "ever"
    long      Path ;                /* Set by PC: this coin's chosen output path
    long      PathCount ;           /* Number "ever" sent down the chosen Path
    long      PathSwitchLevel ;     /* Set by PC: PathCount level to switch coin to
                                   default path
    char      DefaultPath ;         /* Set by PC: Default path for this specific coin
    char      FutureExpansion ;     /* Set by PC: for future use
    char      HeldInEscrow ;        /* count of this note / coin in escrow (usually max 1)
    char      FutureExpansion2 ;    /* for future use
    char*     CoinName ;            /* A string, usually as returned from the acceptor,
                                   describing this 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      EventCount ;          /* Count of events (e.g. rejects) for this acceptor
    char      Currency[4] ;         /* Main currency code reported by an intelligent
                                   acceptor
    AcceptorCoin Coin[MAX_ACCEPTOR_COINS] ; // (only NoOfCoins are set up)
    long      SerialNumber ;        /* Reported serial number (0 if N/A)
    char*     Description ;         /* Device specific string for type / revision / coin
                                   set
} AcceptorBlock ;
```

DispenserBlock

Constants for DispenserBlock

```
enum DispenserConstants
{
    MAX_DISPENSERS          = 16           // Maximum handled

    // Coin Count Status Values
    DISPENSER_COIN_NONE     = 0,           // No dispenser coin reporting
    DISPENSER_COIN_LOW      = 1,           // Less than the low sensor level
    DISPENSER_COIN_MID      = 2,           // Above low sensor but below high
    DISPENSER_COIN_HIGH     = 3,           // High sensor level reported

    DISPENSER_ACCURATE       = -1,         // Coin Count reported by Dispenser
    DISPENSER_ACCURATE_FULL = -2,         // The Dispenser is full

    DISPENSER_REASSIGN_VALUE = 100,        // The Value has just been updated by the
                                           application
    DISPENSER_VALUE_REASSIGNED = 101      // The updated Value has just been accepted by the
                                           IMHEI
} ;
```

Structure for DispenserBlock

```
typedef struct
{
    long      Unit ;                     // Specification of this unit
    long      Status ;                   // Individual Dispenser status
                                           // This takes the same values as PayStatus()

    long      InterfaceNumber ;           // The bus / connection
    long      UnitAddress ;               // For addressable units
    long      Value ;                     // The value of the coins in this dispenser
    long      Count ;                     // Number dispensed according to the hopper records
    long      Inhibit ;                   // Set to 1 to inhibit Dispenser
    long      Currency ;                  // The currency code reported by
                                           // an intelligent dispenser

    long      CoinCount ;                 // The number of coins in the dispenser
    long      CoinCountStatus ;           // Flags Relating to Coin Count (See above)
    long      SerialNumber ;              // Reported serial number (0 if N/A)
    char*     Description ;               // Device specific string for type / revision
} 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 specific code ORed with
- DP_COIN_PAYOUT_DEVICE ORed with
- DP_CCTALK_INTERFACE ORed with
- DP_MANU_MONEY_CONTROLS

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
} ;

#define IS_ACCEPTOR(code)          (code & 0x02000000)
#define IS_COIN_ACCEPTOR(code)    ((code & DP_GENERIC_MASK) == DP_COIN_ACCEPT_DEVICE)
#define IS_NOTE_ACCEPTOR(code)    ((code & DP_GENERIC_MASK) == DP_NOTE_ACCEPT_DEVICE)
#define IS_PAYOUT(code)           (code & 0x01000000)

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,
    DP_ARDAC_INTERFACE             = 0x00050000,
    DP_JCM_INTERFACE               = 0x00060000,
    DP_GPT_INTERFACE               = 0x00070000,
    DP_MDB_INTERFACE               = 0x00080000,
    DP_MDB_LEVEL_3_INTERFACE       = 0x00080000,
    DP_MDB_LEVEL_2_INTERFACE       = 0x00090000,
    // Some Generic Identities
    DP_ID003_NOTE                  = 0 | DP_JCM_INTERFACE
                                | DP_NOTE_ACCEPT_DEVICE,
    DP_MDB_LEVEL_2                 = 0 | DP_MDB_LEVEL_2_INTERFACE
                                | DP_COIN_ACCEPT_DEVICE,
    DP_MDB_LEVEL_3                 = 0 | DP_MDB_LEVEL_3_INTERFACE
                                | DP_COIN_ACCEPT_DEVICE,
    DP_MDB_LEVEL_2_TUBE            = 0 | DP_MDB_LEVEL_2_INTERFACE
                                | DP_COIN_PAYOUT_DEVICE,
    DP_MDB_TYPE_3_PAYOUT           = 0 | DP_MDB_LEVEL_3_INTERFACE
                                | DP_COIN_PAYOUT_DEVICE,
    DP_MDB_BILL                    = 0 | DP_MDB_INTERFACE
                                | DP_NOTE_ACCEPT_DEVICE,
    DP_CC_GHOST_HOPPER             = 255 | DP_CCTALK_INTERFACE // Used by Value hopperz
                                | DP_COIN_PAYOUT_DEVICE,
} ;

#define GET_INTERFACE(code) ((code >> 16) & 0xff)
```

```

enum ManufacturerIdentities
{
    DP_MANUFACTURER_MASK           = 0x0000ff00,
    DP_MANU_UNKNOWN                = 0x00000000,
    DP_MANU_MONEY_CONTROLS        = 0x00000100,
    DP_MANU_INNOVATIVE_TECH       = 0x00000200,
    DP_MANU_MARS_ELECTRONICS      = 0x00000300,
    DP_MANU_AZKOYEN               = 0x00000400,
    DP_MANU_NRI                   = 0x00000500,
    DP_MANU_ICT                   = 0x00000600,
    DP_MANU_JCM                   = 0x00000700,
    DP_MANU_GPT                   = 0x00000800,
    DP_MANU_COINCO                = 0x00000900,
    DP_MANU_ASAHI_SEIKO           = 0x00000A00,
    DP_MANU_ASTROSYSTEMS          = 0x00000B00,
} ;

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

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

    // Asahi Seiko Devices
    DP_AS_WH2                     = 0 | DP_MANU_ASAHI_SEIKO
                                     | DP_CCTALK_INTERFACE
                                     | DP_COIN_PAYOUT_DEVICE,
} ;

```

Please see the latest AESIMHEI.H file in the SDK for an up to date list of these.

CurrentUpdates (1.10.4)

Synopsis

Detect updates to the data presented to the API by the firmware.

The fact that the value returned by **CurrentUpdates** has changed, prompts the application to re-examine all the variable data in which it is interested.

long CurrentUpdates (void) ;

Parameters

None

Return Value

Technically **CurrentUpdates** returns the number of times that the API data has been updated since the PC system initialised. In practice, only *changes* in this value are significant.

Remarks

1. It is possible that the value could change without any visible data changing.
2. *This is only available with the DLL associated with firmware versions 1.10.8 and higher.*

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.

Dispenser Value Reassignment (1.10.7)

Releases of Paylink after 1.10.7 allow the value of the coin associated with a Dispenser to be re-assigned.

To do this:

- the dispenser to be updated should be found using **ReadDispenserDetails()**,
- the **Dispenser.Value** updated to the new value,
- the **Dispenser.Status** field changed to DISPENSER_REASSIGN_VALUE
- and **WriteDispenserDetails()** used to update the record to Paylink.
-

Paylink will acknowledge that the update has been processed by setting the **Dispenser.Status** field to DISPENSER_VALUE_REASSIGNED. *If this value is not seen in the **Dispenser.Status** field, then the value change has not be processed by Paylink.*

Token Handling (Coin Ids) (1.11.x)

As tokens do not have a known value, they appear as coins with value zero. The only way for a game to detect tokens is to use the **CurrentUpdates()** function to detect activity, and then to check for increases in the count of the token(s) to be accepted(**Coin.Count**).

The index for the coin that holds the count for a particular token can be obtained by searching the coin array belonging to the acceptor and comparing the coin name (**Coin.CoinName**) with that of the token.

Dual Currency Handling (Coin Ids) (1.11.x)

If an acceptor is being used to accept coins of more than one currency, the application can determine the currency of a specific coin by examining the first two characters of the name of the coin (**Coin.CoinName**). For supported acceptors, the firmware guarantees that a coin name will always contain a currency code as the first two characters of a coin name.

ccTalk	This contains up to eight characters as returned by the Request Coin Id (184) command.
ID-003	This contains a representation of the three bytes as return by the Get Currency Assignment (0x8A) command. The first two bytes are the hex value for country code, then a '/', then the base value as a decimal number, followed by a '^', then the count of extra zeros as a decimal number.
MDB	TBD
GPT	TBD
ARDAC	The Ardac protocol does not return any information about notes.

Read out of Acceptor Details (1.11.x)

Different protocols / manufacturers provide different details on acceptors. The **Acceptor.Description** field is generated as follows:

ccTalk	The replies to: <ul style="list-style-type: none"> Request Currency Specification ID (91), Request Currency Revision (145), Request Software Revision (241) & Request Product Code (244) commands, separated by '~' characters. Each individual field is truncated to 15 characters, and is omitted if there is no response to the command, although the '~' character is still inserted.
ID-003	The entire reply to the "Get Version Request" (0x88) command
MDB	TBD
GPT	TBD
ARDAC	TBD.

The **Acceptor.SerialNumber** field is generated as follows:

ccTalk	The binary reply to the ID Serial No (242) command.
ID-003	The "standard" ID-003 protocol does not allow for a serial number. A non-standard 0x8F query is issued and any response will be stored here.
MDB	TBD
GPT	TBD
ARDAC	TBD.

Read out of Dispenser Details (1.11.x)

Different protocols / manufacturers provide different details on acceptors. The **Description (Dispenser.Description)** field is generated as follows:

ccTalk	The replies to: <ul style="list-style-type: none"> Request Software Revision (241) & Request Product Code (244) commands, separated by '~' characters. Each individual field is truncated to 15 characters, and is omitted if there is no response to the command, although the '~' character is still inserted.
MDB	TBD

The **Dispenser.SerialNumber** field is generated as follows:

ccTalk	The binary reply to the ID Serial No (242) command.
MDB	TBD

Coin (Note) Routing.

Coins can be easily routed to fill a coin dispenser and one or more cash boxes.

There are 3 routing techniques:

- Route coins to a general cash box.
- Route specific coins to a specific cash box.
- Route specific coins to a dispenser until it is full then route it to a coin specific cash box.

There are 3 settings for each coin that are important:

- Coin.Path The path to the coin specific hopper.
- Coin.DefaultPath The path to the coin specific cash box.
- Coin.PathSwitchLevel When Coin.PathCount reaches Coin.PathSwitchLevel coins are routed to the coin cash box.

Route coins to a general cash box

- Set all coin paths to the desired route.

e.g. General Cash box on route 4.

- Path 4 for all coins
- DefaultPath 0 for all coins
- PathSwitchLevel 0 for all coins

Route specific coins to a specific cash box.

- Set Coin.Path for each coin that is routed to a specific cash box.
- The other 2 coin settings are zero.

e.g. General Cash box on route 4, coins 1 and 2 have separate cash boxes on routes 5 and 6.

- Path 5 for coin 1, 6 for coin 2 and 4 for all other coins
- DefaultPath 0 for all coins
- PathSwitchLevel 0 for all coins

Route coins to a dispenser until it is full then route it to a coin cash box.

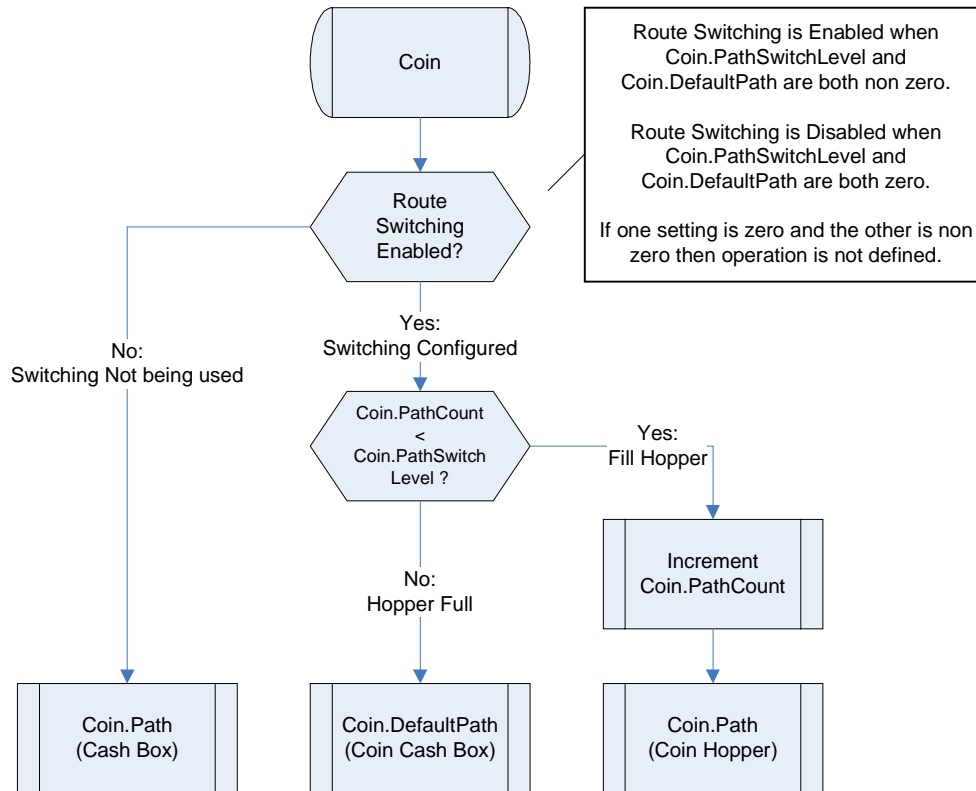
- Set Coin.Path to the dispenser route for each coin that is routed to a hopper.
- Set Coin.DefaultPath to the cash box route for each coin that is routed to a hopper. **This must be non zero.**
- Set Coin.PathSwitchLevel to the Coin.PathCount value at which the dispenser becomes full. **This must be non zero.**

e.g. General Cash box on route 4, coin 1 goes to a dispenser on route 1 and a cash box on route 2. Coin.PathCount is 100 and there is space for 300 more coins in the hopper.

- Path 1 for coin 1, 4 for all other coins
- DefaultPath 2 for coin 1, 0 for all other coins
- PathSwitchLevel 400 for coin 1, 0 for all other coins

When coins are routed to the dispenser (via the Coin.Path route) the variable Coin.PathCount is incremented. When PathCount reaches PathSwitchLevel, further coins are routed to the coin cash box. As the dispenser pays out coins the PathSwitchLevel should be increased by the corresponding amount. Further coins will then be routed to the dispenser again until the new switch level is reached.

Paylink Routing - Flow Diagram



Notes:

- **Setting route 0 should be avoided as it does not exist on an SR5 coin acceptor.**
- **The settings for PathSwitchLevel and PathCount are restored automatically by Paylink after a reset.**

MDB changer support. (1.10.x)

If an MDB changer is used, it will appear as an acceptor in very much the same way as any other acceptor. The coins that are routed to tubes can be distinguished as having a non zero Routed Path, although, obviously, any changes made to the routing will be ignored.

With the payout, the situation is slightly more complicated. The MDB changer protocol supports two different payout mechanisms, a basic one that is always present and an extended one, which is supported on some level 3 changers. The basic provides control over the individual payout tubes, but has no feedback as to whether the payout works. The extended one provides feedback as to the success of the payout, but does not allow any control over which tubes the payout is from.

The solution adopted is to always provide one dispenser for each tube, which is run using the basic mechanism and, if the extended mechanism is present, to provide an additional dispenser which is run using the extended mechanism. Where an extended mechanism dispenser is available, the individual tubes are pre-set to inhibited.

To perform a “normal” payout, you just issue a **PayOut()** request and call **PayStatus()** and **CurrentPaid()** to monitor the results. If you have a level 2 changer, **CurrentPaid()** will update almost instantaneously rather than at the end and will always show that all coins have been paid. If you have a level 3 changer, **CurrentPaid()** will update during the process, and you may get a PAY_EMPTY status from **PayStatus()**, with **CurrentPaid()** then reflecting the actual payout achieved.

The current levels of MDB tubes, *as reported by the coin-changer*, are returned in the field **CoinCount**. In addition, the field **CoinCountStatus** will contain the value DISPENSER_ACCURATE for a normal tube, and DISPENSER_ACCURATE_FULL if the changer is reporting the tube as full. Note that the levels reported by the changer do not necessarily update in a “sensible” fashion after a payout.

Should you wish to perform an operation on a specific tube (e.g. emptying it), you should inhibit the extended mechanism dispenser and enable the specific tube you wish to control.

As the manufacturer is shown in the acceptor detail block for the changer, the extended mechanism dispenser has the constant type **DP_MDB_TYPE_3_PAYOUT** while the individual tubes have the type **DP_MDB_LEVEL_2_TUBE**.

MDB tube level monitoring.**Monitoring:**

The main method for determining tube levels is via the Tube Status (0x02) MDB command.

This is issued during startup and then every 25 seconds. The response to this is copied directly into the tube coin level, and one of the DISPENSER_ACCURATE or DISPENSER_ACCURATE_FULL level statuses set.

Coin Insertion:

When a coin insertion (event code 0x40) is reported as going to a tube, the changer also includes an updated value for the tube level. If this is non-zero then this is used to overwrite the coin level for the tube. When a coin insertion is reported as going to the cashbox for a coin that has an associated tube, Paylink immediately issues a Tube Status (0x02) MDB command to obtain an accurate on the levels. (Note that after a delay of up to 25 seconds this will be replaced by the value from a Tube Status command)

Manual Dispense:

When a manual dispense (event code 0x80) is reported then the reported tube level copied directly into the tube coin level. . (Note that after a delay of up to 25 seconds this will be replaced by the value from a Tube Status command if that is different)

Payout:

While a payout is in progress, no updates are made to the coin level. As soon as the payout completes, Paylink immediately issues a Tube Status (0x02) MDB command to obtain the changer's opinion of the new levels.

Dispenser Power Fail support. (1.10.x)

Some dispensers, especially some hoppers produced by MCL, are guaranteed to correctly count coins even if power is removed during a payout sequence. This facility is explicitly supported in the Paylink software. The **Count** field for these hoppers is set during initialisation to correspond to the “total coins paid since manufacture” value retrieved from the hopper, and is then updated as payouts occur. It is this field that allows for the correct counting of coins over a power failure.

At the end of every payout sequence, the Paylink stores, internally, the **Count** for each hopper. At initialisation as well as reporting the retrieved count, it is also compared with the saved value. This enables the **CurrentPaid()** function to continue to report the correct value, and also generates an **IMHEI_COIN_DISPENSER_UPDATE Event** (see below) to register this update.

Combi Hopper Support. (1.10.x)

This single unit dispenses two different coin values. It is therefore handled in a similar way to the MDB system. There is a primary dispenser, which is set up as a normal unit with a **Unit** field of DP_MCL_SCH3, and a **Value** field with the lower coin value in it. The **Count** in this dispenser is the count of the lower value coins dispensed. In addition, another dispenser is set up, with a matching **Address** field, a **Unit** field of DP_CC_GHOST_HOPPER, the **Value** of the higher coin and the **Count** of the higher value coins dispensed.

Note that, due to limitations of the unit, during a payout operation the **Count** of the main dispenser *only* is updated, as though all coins dispensed were of this value. At the end of the sequence, while **LastPayStatus()** is still returning PAY_ONGOING, the accurate count of both coins is retrieved and the two separate **Count** fields updates. The result of this is that, as the operation finishes, the **Count** for the lower value dispenser decrements.

Multiple Paylink Unit Support.

Overview

Although the Paylink system was designed on the basis of a single Paylink unit being connected to a PC, facilities are provided to support multiple Paylink units.

The only change that is visible to a programmer when multiple units are in use is that the **OpenSpecificMHE** is used to associate a specific one of the multiple Paylink unit interface areas with the program.

It is envisaged that in a system with multiple Paylink units a separate instance of the program will be running for each Paylink unit interface area and a supervisory level will start the different programs. This is not compulsory as **OpenSpecificMHE** can be called repeatedly with different parameters so as to switch between Paylink unit interface areas.

Unit Identification

The USB interface chip on a Paylink unit provides a "Serial Number". This is pre-set during manufacture to "AE000001" - but is not used or checked in a system that does not have multiple units.

When the **AESWDriver** program is run, the default is for it to search all USB devices that may be a Paylink, and connect to the first one it finds. When the /S=<SerialNo> switch is provided on the command line, this has two effects:

Firstly, it causes the driver program to create a named Paylink unit interface area, which can then be connected to by an **OpenSpecificMHE** call with a matching parameter.

Secondly it causes the driver program to search all USB devices that may be a Paylink until it finds one with a matching programmed serial number.

The serial number is *not* associated with the Paylink firmware, and any release of Paylink firmware may be used in a multiple Paylink system. The **PaylinkSerial** utility is available as a part of the released SDK, which takes as a parameter a serial number and programs it into the single Paylink unit currently connected to the system.

Utility Functions

CheckOperation (1.11.x)

Synopsis

This call allows an application to check that the Paylink and its connection to the PC are operational. It also allow the application to automatically close down currency acceptance in the event of any PC malfunction.

**long CheckOperation(long Sequence,
 long Timeout)**

Parameters

1. Sequence
A unique number for this call, freely chosen by the application.
2. Timeout
A time in milliseconds before which another **CheckOperation()** call must be made, *with a different value in **Sequence***, in order to continue the normal operation of Paylink. If zero, then this functionality is inactive from then on.

Return Value

The last **Sequence** value of which the Paylink unit has been notified, or -1 if the Paylink does not support this facility.

Remarks

1. In normal operation, Paylink can be expected to have updated the value to be returned by this within 100 milliseconds of the previous call. It is suggested that this call is made every 500 milliseconds or longer to allow for transient delays.
2. If the **Timeout** expires, Paylink will “silently” disable all the acceptors that are connected to it. The next call to **CheckOperation()** will “silently” re-enable them. This facility is not operation until the first call of **CheckOperation()**.

NextEvent

Synopsis

This call provides access to all the detailed workings of the peripherals connected to the system. All Acceptor / Dispenser events such as errors, frauds and rejects (including pass / fail of internal self test) that are received will be queued (in a short queue) and can be retrieved with **NextEvent** calls.

```
int NextEvent(EventDetailBlock* EventDetail);
```

Parameters

3. EventDetail
NULL, or the address of the single structure at which to store more details of the event given by the return value.

Return Value

The return code is 0 (IMHEI_NULL) if no event is available, otherwise it is the next event.

Remarks

3. In the case where one or more events are missed, the code IMHEI_OVERFLOW will replace the missed events.
4. If only basic information is required, then (as note, coin & Dispenser event codes do not overlap) the **EventDetail** parameter can often be set to NULL, as the device is implicit in the event.
5. The values for the **EventCodes** returned are in the separate header file **ImheiEvent.h** (see Appendix 1)
6. The **RawEvent** field for various drivers is as follows:

Driver Software	Raw Code for Event	Raw Code for Fault
cctalk coin	Byte from "Read Buffered Credit" response.	1 st byte of "Perform self test" response.
cctalk note	Byte from "Read Buffered Bill Events" response.	1 st byte of "Perform self test" response.
Ardac II note		
ID-003	Response to "Status Poll"	The byte following a FAILURE response
GPT note		
MDB Bill Acceptor		
MDB Changer		

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.

At present, this facility is not implemented!

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.

SetDeviceKey

Synopsis

The **SetDeviceKey** call is made by the PC application software to set such things as an encryption key.

```
void SetDeviceKey (long InterfaceNo,  
                  long Address,  
                  long Key);
```

Parameters

1. InterfaceNo
The Interface on which the device is located
2. Address
The address of the device whose key is being updated
3. Key
The 32 bit key to be remembered for the device.

Return Value

None

Remarks

1. At present, this can only be used for a Lumina acceptor at address 40 on interface 2, the cctalk interface. The key (as 6 hex digits) is used as the encryption key.
2. An example application for this is available within the SDK folder structure.

SerialNumber

Synopsis

The **SerialNumber** call provides access to the electronic serial number stored on the device.

```
long SerialNumber (void) ;
```

Parameters

None

Return Value

32 bit serial number.

Remarks

1. A serial number of -1 indicates that a serial number has not been set in the device.
2. A serial number of 0 indicates that the device firmware does not support serial numbers

FirmwareVersion

Synopsis

The **FirmwareVersion** call allows a control application to discover the exact description of the firmware running on the unit.

```
long FirmwareVersion (char* CompileDate,  
                     char* CompileTime);
```

Parameters

1. CompileDate
This is a pointer to a 16 byte area that receives a null terminated printable version of the date on which the firmware was installed.
2. CompileTime
This is a pointer to a 16 byte area that receives a null terminated printable version of the time at which the firmware was installed.

Return Values

The firmware version, as a 32 bit integer. This is normally shown as 4 x 8 bit numbers separated by dots.

Remarks

Either or both of the character pointers may be null.

USBDriverStatus

Synopsis

The USBDriverStatus call allows an interested application to retrieve the status of the USBDriver program for Paylink system.

```
USBStatus DLL USBDriverStatus (void) ;
```

Parameters

None

Return Values

Mnemonic	Value	Meaning
NOT_USB	-1	Interface is to a PCI card
USB_IDLE	0	No driver or other program running
STANDARD_DRIVER	1	The driver program is running normally
FLASH_LOADER	2	The flash re-programming tool is using the link
MANUFACTURING_TEST	3	The manufacturing test tool is using the link
DRIVER_RESTART	4	The standard driver is in the process of exiting / restarting
USB_ERROR	5	The driver has received an error from the low level driver

Remarks

1. For PCI systems this is obviously meaningless and the system returns NOT_USB
2. Be aware that further error statuses may be added. Any response other than STANDARD_DRIVER should be regarded as indicating that the system is not currently functional.

USBDriverExit

Synopsis

The **USBDriverExit** call allows a control application to request that the USB driver program exits in a controlled manner.

void USBDriverExit (void) ;

Parameters

None

Return Values

None

Remarks

This sets the **USBDriverStatus** to DRIVER_RESTART. Driver programs with version 1.0.3.1 or greater will report their exit by changing the **USBDriverStatus** to USB_IDLE.

For PCI systems this is obviously meaningless and has no effect.

IMHEIConsistencyError

Synopsis

The **IMHEIConsistencyError** call allows an application to check that a transient (hardware) error has not caused corruption of the underlying data structures used to hold the current monetary situation. Although the use of state tables removes the vulnerability of the system to time problems, it increases its vulnerability to *expensive* hardware errors (which could falsely indicate very large money increments.)

```
char* DLL IMHEIConsistencyError(int CoinTime,  
                                int NoteTime) ;
```

Parameters

None

1. CoinTime
Default STANDARD_COIN_TIME = 500 msec.
This is the minimum time in milliseconds that will elapse between successive coin insertions. It should be overridden by the application where a fast coin acceptor is in use.
2. NoteTime
Default STANDARD_NOTE_TIME = 5000 msec.
This is the minimum time in milliseconds that will elapse between successive note insertions. It should be overridden by the application where a fast note acceptor is in use.

Return Value

If all the data structures are both consistent and reasonable, the function returns NULL.

If there is any problem an English text message is returned describing the problem.

Remarks

1. A non-NULL return is a totally unrecoverable situation.
It is expected that application will report the error, and then stop operation.
2. As well as calling this function periodically, it is recommended that it is called after the detection of a credit increase.

Auditing / Event Processing

This section elaborates further on the processing behind the events returned by the NextEvent() function.

There is no intention that these events would be used for the normal operation of the game. Rather, the intention is that they can be captured and presented in “management” reports.

(Obviously, the application can respond automatically to events such as fraud, by disabling everything for a while, but this doesn’t form part of the algorithms by which the application manages the peripherals.)

Structure for EventDetailBlock

```
typedef struct
{
    long    EventCode ;           // The code (the same as returned by NextEvent)
    long    RawEvent ;           // The actual code returned by the peripheral
    long    DispenserEvent ;     // True if the device was a dispenser
                                   // False for an acceptor
    long    Index ;              // The ReadxxxBlock index of the generating device
} EventDetailBlock ;
```

Event Codes for NextEvent / EventDetailBlock

Event codes have an internal structure, allowing cateogizations. The bottom 6 bits are the unique code for the event, serious fault related codes have bit 5 set. Above this are bits describing the type of unit affected.

```
// enums to allow this cateogisation to be acheived
enum
{
    EVENT_CODE_MASK           = 0x03f,
    UNIT_TYPE_MASK           = ~0x03f,
    FAULT_BIT                 = 0x020,
    COIN_DISPENSER_EVENT      = 0x040,
    NOTE_DISPENSER_EVENT      = 0x080,
    COIN_EVENT                = 0x0C0,
    NOTE_EVENT                = 0x100,
} ;

// The common base codes
enum
{
    EVENT_OK,                 // Internal use only
    EVENT_BUSY,               // Internal use only

    REJECTED,
    INHIBITED,
    MISREAD,
    FRAUD,
    JAM,
    JAM_FIXED,
    RETURN,
    OUTPUT_PROBLEM,
    OUTPUT_FIXED,
    INTERNAL_PROBLEM,
    UNKNOWN,
    DISPENSE_UPDATE,

    // Fault codes
    NOW_OK = 0,
    REPORTED_FAULT,
    UNIT_TIMEOUT,
    UNIT_RESET,
    SELF_TEST_REFUSED,
```

CONFIDENTIAL

```

    } ;

// The actual Full Event Codes
enum
{
    // General
    IMHEI_NULL = 0,
    IMHEI_INTERFACE_START = 1,
    IMHEI_APPLICATION_START = 2,
    IMHEI_APPLICATION_EXIT = 3,

    IMHEI_OVERFLOW = 0x1f,

    // Coin Acceptors
    IMHEI_COIN_NOW_OK = COIN_EVENT | FAULT_BIT | NOW_OK,
    IMHEI_COIN_UNIT_REPORTED_FAULT = COIN_EVENT | FAULT_BIT | REPORTED_FAULT,
    IMHEI_COIN_UNIT_TIMEOUT = COIN_EVENT | FAULT_BIT | UNIT_TIMEOUT,
    IMHEI_COIN_UNIT_RESET = COIN_EVENT | FAULT_BIT | UNIT_RESET,
    IMHEI_COIN_SELF_TEST_REFUSED = COIN_EVENT | FAULT_BIT | SELF_TEST_REFUSED,

    IMHEI_COIN_REJECT_COIN = COIN_EVENT | REJECTED,
    IMHEI_COIN_INHIBITED_COIN = COIN_EVENT | INHIBITED,
    IMHEI_COIN_FRAUD_ATTEMPT = COIN_EVENT | FRAUD,
    IMHEI_COIN_ACCEPTOR_JAM = COIN_EVENT | JAM,
    IMHEI_COIN_COIN_RETURN = COIN_EVENT | RETURN,
    IMHEI_COIN_SORTER_JAM = COIN_EVENT | OUTPUT_PROBLEM,
    IMHEI_COIN_INTERNAL_PROBLEM = COIN_EVENT | INTERNAL_PROBLEM,
    IMHEI_COIN_UNCLASSIFIED_EVENT = COIN_EVENT | UNKNOWN,

    // Note Acceptors
    IMHEI_NOTE_NOW_OK = NOTE_EVENT | FAULT_BIT | NOW_OK,
    IMHEI_NOTE_UNIT_REPORTED_FAULT = NOTE_EVENT | FAULT_BIT | REPORTED_FAULT,
    IMHEI_NOTE_UNIT_TIMEOUT = NOTE_EVENT | FAULT_BIT | UNIT_TIMEOUT,
    IMHEI_NOTE_UNIT_RESET = NOTE_EVENT | FAULT_BIT | UNIT_RESET,
    IMHEI_NOTE_SELF_TEST_REFUSED = NOTE_EVENT | FAULT_BIT | SELF_TEST_REFUSED,

    IMHEI_NOTE_REJECT_NOTE = NOTE_EVENT | REJECTED,
    IMHEI_NOTE_INHIBITED_NOTE = NOTE_EVENT | INHIBITED,
    IMHEI_NOTE_NOTE_MISREAD = NOTE_EVENT | MISREAD,
    IMHEI_NOTE_FRAUD_ATTEMPT = NOTE_EVENT | FRAUD,
    IMHEI_NOTE_ACCEPTOR_JAM = NOTE_EVENT | JAM,
    IMHEI_NOTE_ACCEPTOR_JAM_FIXED = NOTE_EVENT | JAM_FIXED,
    IMHEI_NOTE_NOTE_RETURNED = NOTE_EVENT | RETURN,
    IMHEI_NOTE_STACKER_PROBLEM = NOTE_EVENT | OUTPUT_PROBLEM,
    IMHEI_NOTE_STACKER_FIXED = NOTE_EVENT | OUTPUT_FIXED,
    IMHEI_NOTE_INTERNAL_ERROR = NOTE_EVENT | INTERNAL_PROBLEM,
    IMHEI_NOTE_UNCLASSIFIED_EVENT = NOTE_EVENT | UNKNOWN,

    // Coin Dispenser
    IMHEI_COIN_DISPENSER_NOW_OK = COIN_DISPENSER_EVENT | FAULT_BIT | NOW_OK,
    IMHEI_COIN_DISPENSER_REPORTED_FAULT = COIN_DISPENSER_EVENT | FAULT_BIT | REPORTED_FAULT,
    IMHEI_COIN_DISPENSER_TIMEOUT = COIN_DISPENSER_EVENT | FAULT_BIT | UNIT_TIMEOUT,
    IMHEI_COIN_DISPENSER_RESET = COIN_DISPENSER_EVENT | FAULT_BIT | UNIT_RESET,
    IMHEI_COIN_DISPENSER_SELF_TEST_REFUSED = COIN_DISPENSER_EVENT | FAULT_BIT | SELF_TEST_REFUSED,

    IMHEI_COIN_DISPENSER_FRAUD_ATTEMPT = COIN_DISPENSER_EVENT | FRAUD,
    IMHEI_COIN_DISPENSER_UPDATE = COIN_DISPENSER_EVENT | DISPENSE_UPDATE,
} ;

```

cctalk coin processing

Fault Events

During start-up the cctalk command “Do self Test” is sent to the acceptor. The response is queued as an event with the first byte of the response in **RawEvent** and an **EventCode** type of **IMHEI_COIN_NOW_OK** or **IMHEI_COIN_UNIT_REPORTED_FAULT**.

If the unit is reset (the sequence number is found to be zero) or repeated messages are ignored **IMHEI_COIN_UNIT_RESET** or **IMHEI_COIN_UNIT_TIMEOUT** event is queued. Whenever any of these faults have been reported, the handler will continually “poll” the acceptor with “Do self Test” commands until a “non-faulty” response is returned.

Coin Events

When the acceptor reports an event other than an accepted coin, this is queued as a **COIN_DISPENSER_EVENT** event, with the actual event byte reported in **RawEvent**.

The handler classifies cctalk events as:

Event Number	Meaning	Event Classification
1	Coin Rejected	REJECTED
2	Coin Inhibited	INHIBITED
3	Multiple window	REJECTED
4	Wake-up timeout	JAM
5	Validation timeout	JAM
6	Credit sensor timeout	JAM
7	Sorter opto timeout	OUTPUT_PROBLEM
8	2nd close coin error	REJECTED
9	Accept gate not ready	REJECTED
10	Credit sensor not ready	REJECTED
11	Sorter not ready	REJECTED
12	Reject coin not cleared	REJECTED
13	Validation sensor not ready	REJECTED
14	Credit sensor blocked	JAM
15	Sorter opto blocked	OUTPUT_PROBLEM
16	Credit sequence error	FRAUD
17	Coin going backwards	FRAUD
18	Coin too fast (over credit sensor)	FRAUD
19	Coin too slow (over credit sensor)	FRAUD
20	C.O.S. mechanism activated (coin-on-string)	FRAUD
21	DCE opto timeout	FRAUD
22	DCE opto not seen	FRAUD
23	Credit sensor reached too early	FRAUD
24	Reject coin (repeated sequential trip)	FRAUD
25	Reject slug	FRAUD
26	Reject sensor blocked	JAM
27	Games overload	INTERNAL_PROBLEM
28	Max. coin meter pulses exceeded	INTERNAL_PROBLEM
128-159	Inhibited Coin	INHIBITED
254	Flight Deck Open	RETURN

cctalk note processing

Fault Events

Shortly after start-up the cctalk command "Do self Test" is sent to the acceptor. The response is queued as an event with the first byte of the response in **RawEvent** and an **EventCode** type of **IMHEI_NOTE_NOW_OK** or **IMHEI_NOTE_UNIT_REPORTED_FAULT**.

Some acceptors reply to this command with a NAK, these are reported as **IMHEI_NOTE_SELF_TEST_REFUSED**.

If the unit is reset (the sequence number is found to be zero) or repeated messages are ignored **IMHEI_NOTE_UNIT_RESET** or **IMHEI_NOTE_UNIT_TIMEOUT** event is queued.

Whenever any of these faults have been reported, the handler will continually "poll" the acceptor with "Do self Test" commands until a "non-faulty" response is returned.

Note Events

When the acceptor reports an event other than an accepted coin, this is queued as an **NOTE_DISPENSER_EVENT** event, with the actual event byte reported in **RawEvent**.

The handler classifies cctalk events as:

Event Number	Meaning	Event Classification
0	Master inhibit active	INHIBITED
1	Bill returned from escrow	RETURN
2	Invalid bill (due to validation fail)	REJECTED
3	Invalid bill (due to transport problem)	REJECTED
4	Inhibited bill (on serial)	INHIBITED
5	Inhibited bill (on DIP switches)	INHIBITED
6	Bill jammed in transport (unsafe mode)	MISREAD
7	Bill jammed in stacker	OUTPUT_PROBLEM
8	Bill pulled backwards	FRAUD
9	Bill tamper	FRAUD
10	Stacker OK	OUTPUT_FIXED
11	Stacker removed	OUTPUT_PROBLEM
12	Stacker inserted	OUTPUT_FIXED
13	Stacker faulty	OUTPUT_PROBLEM
14	Stacker full	OUTPUT_PROBLEM
15	Stacker jammed	OUTPUT_PROBLEM
16	Bill jammed in transport (safe mode)	JAM
17	Opto fraud detected	FRAUD
18	String fraud detected	FRAUD
19	Anti-string mechanism faulty	INTERNAL_PROBLEM

ID-003 note processing

Fault Events

There is no specific self test command with ID-003, the acceptor reports faults in response to a poll. When the protocol handler completes its initialisation, the first idle response is reported as **IMHEI_NOTE_NOW_OK**.

When a **FAILURE** response to a status poll is received, this is reported as an **IMHEI_NOTE_UNIT_REPORTED_FAULT** event. A failure status is expected to be continually reported by the acceptor until it is cleared. When the acceptor again reports **IDLING**, then an **IMHEI_NOTE_NOW_OK** event is reported.

Other “non normal” responses to a status poll are reported as events as they are receive according to the table below.

In a similar way to the action for faults, **OUTPUT_FIXED** is reported when events that translate to **OUTPUT_PROBLEM** are cleared.

Status Value	Name	Event Classification
0x17	REJECTING	REJECTED
0x41	POWER_UP_WITH_BILL_IN_ACCEPTOR	REJECTED
0x42	POWER_UP_WITH_BILL_IN_STACKER	REJECTED
0x43	STACKER_FULL	OUTPUT_PROBLEM
0x44	STACKER_OPEN	OUTPUT_PROBLEM
0x45	JAM_IN_ACCEPTOR	JAM
0x46	JAM_IN_STACKER	OUTPUT_PROBLEM
0x47	PAUSE	UNKNOWN
0x48	CHEATED	FRAUD
0x49	FAILURE	- Fault Report
0x4A	COMMUNICATION_ERROR	INTERNAL_PROBLEM

Note Reader Escrow

EscrowEnable

Synopsis

Change the mode of operation of all escrow capable acceptors to hold inserted currency in escrow until a call of **EscrowAccept**.

The **EscrowEnable** call is used to start using the escrow system

```
void EscrowEnable (void) ;
```

Parameters

None

Return Value

None

EscrowDisable

Synopsis

Change the mode of operation of all escrow capable acceptors back to the default mode in which all currency is fully accepted on insertion

```
void EscrowDisable (void) ;
```

Parameters

None

Return Value

None

Remarks

1. If any currency is currently held in escrow when this call is made, it will be accepted without comment.

EscrowThroughput

Synopsis

Determine the cumulative monetary value that has been held in escrow since the system was reset.

The **EscrowThroughput** call is used to determine the cumulative total value of all coins and notes read by the money handling equipment that have ever been held in escrow.

long EscrowThroughput (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 held in Escrow.

Remarks

1. It is the responsibility of the application to keep track of value that has been accepted and to monitor for new coin / note insertions by increases in the returned value.
2. Note that this value should be read following the call to **OpenMHE** and before the call to **EnableInterface / EscrowEnable** to establish a starting point before any coins or notes are read.
3. If the acceptor auto-returns the coin / note then this will fall to its previous value. This can (potentially) occur *after* a call to **EscrowAccept()** or **EscrowReturn()** if the acceptor has already started its return sequence.

EscrowAccept

Synopsis

If the acceptor that was last reported as holding currency in escrow is still in that state, this call will cause it to accept that currency.

void EscrowAccept (void) ;

Parameters

None

Return Value

None

Remarks

1. If a second acceptor has (unreported) currency in escrow at the time this call is made, it will immediately cause the **EscrowThroughput** to be updated.
2. If no currency is currently held in escrow when this call is made, it will be silently ignored.

EscrowReturn

Synopsis

If the acceptor that was last reported as holding currency in escrow is still in that state, this call will cause it to return that currency.

void EscrowReturn (void) ;

Parameters

None

Return Value

None

Remarks

1. If a second acceptor has (unreported) currency in escrow at the time this call is made, it will immediately cause the **EscrowThroughput** to be updated.
2. If no currency is currently held in escrow when this call is made, it will be silently ignored.

Escrow system usage

Where an acceptor provides escrow facilities, the IMHEI card fully supports these: by enabling escrow mode. It reports the note that is currently held in escrow by an acceptor, and allows the game to either return or accept the escrow holding of the acceptor.

In most system only one escrow capable acceptor will be present, the IMHEI card will however support escrow on an unlimited number of acceptors. In order to allow for accurate information and control to pass between the game and the IMHEI firmware, the escrow holding reported is limited to a single acceptor at time. If two acceptors are holding escrow at the same time, the second will not be reported until the first has completed.

At start-up, the system does not report escrow details and all acceptors are run in "normal" mode where all currency is accepted. To use escrow the call **EscrowEnable** is issued. Following this the call **EscrowThroughput** will return the *total* value of all currency that has ever been held in escrow (in the same way as for **CurrentValue** except that the value is not preserved over resets). An increase in the value returned indicates that a note is now in escrow. The **HeldInEscrow** field within the **AcceptorCoin** structure will indicate the number of each note / coin that is currently being held.

The **EscrowAccept** call will cause the IMHEI card to complete the acceptance of the currency in question. When complete, this will be indicated by an increase in **CurrentValue**. An **EscrowReturn** call will cause the currency to be returned with no further indication to the game. Following either call, the **EscrowThroughput** value may increase immediately due to another acceptor having an escrow holding.

If the game wishes to stop using the escrow facilities, it may issue the **EscrowDisable** call. This will have the side effect of accepting any outstanding escrow holdings.

Meters / Counters

The IMHEI units 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(long CounterNo,  
                      long 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(long 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(long 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 **ReadCounterCaption** call is used to determine the caption for the specified counter

char* CounterCaption(long 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.

E²Prom

Included in the IMHEI card is E²PROM memory, which is used by the embedded process to maintain counters etc. 256 bytes of this E²PROM is available to users to store essential information if they wish to run their system with no other writeable storage.

In this section, routines are described to access this user storage and to allow for a user application to clear all the E²PROM memory on the card, after testing and before delivery to an end user.

E2PromReset

Synopsis

The **E2PromReset** call is made by the PC application software to clear all the *internal* E²PROM memory on the card. This is the area where the IMHEI system keeps the value in / value out counters, the configuration information, etc.

```
void E2PromReset(long LockE2Prom);
```

Parameters

1. LockE2Prom
This is a Boolean flag. If zero, then the E2PROM may be reset again later.
If non zero, then **all** future calls to this function will have no effect on the card.

Return Value

None

Remarks

An example application for this is available within the SDK folder structure.

E2PromWrite

Synopsis

The **E2PromWrite** call is made by the PC application software to write to all or part of the user E²PROM on the card.

```
void E2PromWrite (void* UserBuffer,  
                  long  BufferLength);
```

Parameters

1. UserBuffer
This is the address of the user's buffer, from which **BufferLength** bytes of data are copied to the start of the user area.
2. BufferLength
This is the count of the number bytes to be transferred. If this is greater than 256 the extra will be silently ignored.

Return Value

None

Remarks

1. This call schedules the write to the E²PROM memory and returns immediately. There is no way of knowing when the E²PROM has actually been updated but, barring hardware errors, it will be complete within one second of the call.

E2PromRead

Synopsis

The **E2PromRead** call is made by the PC application software to obtain all or part of the user E²PROM from the card.

```
void E2PromRead (void* UserBuffer,  
                 long  BufferLength);
```

Parameters

1. UserBuffer
This is the address of the user's buffer, into which the current contents of the user E²PROM area are copied.
2. BufferLength
This is the count of the number bytes to be transferred. If this is greater than 256 the extra will be silently ignored.

Return Value

None

Remarks

1. Unwritten E²Prom memory is initialised all one bits.
2. Writes performed by E2PromWrite will be reflected immediately in the data returned by this function, regardless of whether or not they have been committed to E²Prom memory.

Bar Codes

Where an acceptor provides barcode facilities, the IMHEI card fully support this by enabling bar code acceptance and reporting the barcodes read.

Barcode reading is always handled using the Escrow position on the acceptor. The barcode is held in the acceptor pending a call from the application the either stack or return it.

In most systems, only one barcode capable acceptor will be present, the IMHEI card will however support barcodes on an unlimited number of acceptors. In order to allow for accurate information and control to pass between the game and the IMHEI firmware, the barcode reported is limited to a single acceptor at time. If two acceptors are holding barcoded tickets at the same time, the second will not be reported until the first has completed.

All the barcodes processed by the IMHEI system are in the format "Interleaved 2 of 5" and are 18 characters long. (Functions return a 19 character, NULL terminated, string.)

Barcodes read by the IMHEI can also be printed if a dedicated barcode printer is connected.

Barcode Reading

BarcodeEnable

Synopsis

Change the mode of operation of all Barcode capable acceptors to accept tickets with barcodes on them.

The **BarcodeEnable** call is used to start using the Barcode system

```
void BarcodeEnable (void) ;
```

Parameters

None

Return Value

None

BarcodeDisable

Synopsis

Change the mode of operation of all Barcode capable acceptors back to the default mode in which only currency is accepted.

```
void BarcodeDisable (void) ;
```

Parameters

None

Return Value

None

Remarks

1. If a Barcoded ticket is currently held when this call is made, it will be returned without comment.

BarcodeInEscrow

Synopsis

This is the regular “polling” call that the application should make into the DLL to obtain the current status of the barcode system. If a barcode is read by an acceptor, it will be held in escrow and this call will return true in notification of the fact.

```
bool BarcodeInEscrow (char BarcodeString[19]) ;
```

Parameters

1. BarcodeString
A pointer to a buffer of at least 18 characters into which the last barcode read from any acceptor is placed. This will be all NULL if no barcoded ticket has been read since system start-up.

Return Value

The return value is true if there is a barcode ticket currently held in an Acceptor, false if there is not.

Remarks

1. There is no guarantee that at the time the call is made the acceptor has not irrevocably decided to auto-eject the ticket.

BarcodeStacked

Synopsis

Following a call to **BarcodeAccept** the system *may* complete the reading of a barcoded ticket. If it does, then the count returned by **BarcodeStacked** will increment. There is no guarantee that this will take place, so the application should continue to poll **BarcodeInEscrow**.

```
long BarcodeStacked (char BarcodeString[19]) ;
```

Parameters

2. BarcodeString
A pointer to a buffer of at least 18 characters into which the last barcode read from any acceptor is placed. This will be all NULL if no barcoded ticket has been read since system start-up.

Return Value

The count of all the barcoded tickets that have been stacked since system start-up. An increase in this value indicates that the current ticket has been stacked - its contents will be in the **BarcodeString** buffer.

Remarks

2. It is the responsibility of the application to keep track of the number of tickets that have been accepted and to monitor for new 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 / BarcodeEnable** to establish a starting point before any new tickets are read.

BarcodeAccept

Synopsis

If the acceptor that was last reported as holding a Barcode ticket is still in that state, this call will cause it to accept that currency.

void BarcodeAccept (void) ;

Parameters

None

Return Value

None

Remarks

1. If a second acceptor has (unreported) currency in Barcode at the time this call is made, it will immediately cause the **BarcodeTicket** to be updated.
2. If no ticket is currently held when this call is made, it will be silently ignored.

BarcodeReturn

Synopsis

If the acceptor that was last reported as holding a Barcode ticket is still in that state, this call will cause it to return that currency.

void BarcodeReturn (void) ;

Parameters

None

Return Value

None

Remarks

1. If a second acceptor has (unreported) currency in Barcode at the time this call is made, it will immediately cause the **BarcodeTicket** to be updated.
2. If no ticket is currently held when this call is made, it will be silently ignored.

Barcode Printing

BarcodePrint

Synopsis

This call is used to print a barcoded ticket, if the IMHEI system supports a printer.

```
void BarcodePrint (TicketDescription* TicketContents) ;
```

Parameters

1. TicketContents.
Pointer to a TicketDescription structure that holds pointers to the strings that the application is "filling in". NULL pointers will cause the relevant fields to default (usually to blanks).

typedef struct

```
{  
long TicketType ; // The "template" for the ticket  
char* BarcodeData ;  
char* AmountInWords ;  
char* AmountAsNumber ; // But still a string  
char* MachineIdentity ;  
char* DatePrinted ;  
char* TimePrinted ;  
} TicketDescription ;
```

Return Value

None

Remarks

1. There are a number of fields that can be printed a barcode ticket.
Rather than provide a function with a large number of possibly null parameters, we use a structure, which may have fields added to end.
The user should ensure that all unused pointers are zero.
2. Before issuing this call the application should ensure that **BarcodePrintStatus** has returned a status of **PRINTER_IDLE**
3. The mechanics of the printing mechanism rely on **BarcodePrintStatus** being called regularly after this call, in order to "stage" the data to the interface.

BarcodePrintStatus

Synopsis

This call is used to determine the status of the barcoded ticket printing system.

long BarcodePrintStatus (void) ;

Parameters

None

Return Value

Mnemonic	Value	Meaning
PRINTER_NONE	0	Printer completely non functional / not present
PRINTER_FAULT	0x80000000	There is a fault somewhere
PRINTER_IDLE	0x00000001	The printer is OK / Idle /Finished
PRINTER_BUSY	0x00000002	Printing is currently taking place
PRINTER_PLATEN_UP	0x00000004	
PRINTER_PAPER_OUT	0x00000008	
PRINTER_HEAD_FAULT	0x00000010	
PRINTER_VOLT_FAULT	0x00000040	
PRINTER_TEMP_FAULT	0x00000080	
PRINTER_INTERNAL_ERROR	0x00000100	
PRINTER_PAPER_IN_CHUTE	0x00000200	
PRINTER_OFFLINE	0x00000400	
PRINTER_MISSING_SUPPY_INDEX	0x00000800	
PRINTER_CUTTER_FAULT	0x00001000	
PRINTER_PAPER_JAM	0x00002000	
PRINTER_PAPER_LOW	0x00004000	
PRINTER_NOT_TOP_OF_FORM	0x00008000	
PRINTER_OPEN	0x00010000	
PRINTER_TOP_OF_FORM	0x00020000	
PRINTER_JUST_RESET	0x00040000	

Remarks

1. The mechanics of the printing mechanism rely on this being called regularly after the **BarcodePrint** call, in order to “stage” the data to the interface, until **PRINTER_BUSY** is no longer returned.
2. Any reported fault that requires an operator action will cause the **PRINTER_FAULT** bit to be set.
3. A **PRINTER_NONE** status will be reported if the printer is powered off after having been working.

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,  
                          void*   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, with any addresses and embedded lengths required by the peripheral device included. Overheads such as standard checksums will be added by the IMHEI.

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,
                        void*   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

+ve return values indicate a message has been returned.

Other values are:

-5	INTERFACE_NO_DATA	The handshake has completed, but no data was returned.
-4	INTERFACE_TOO_LONG	Input command is too long
-3	INTERFACE_NON_EXIST	Non command oriented interface (the corresponding WriteInterfaceBlock was ignored)
-2	INTERFACE_OVERFLOW	Command buffer overflow (the corresponding WriteInterfaceBlock was ignored)
-1	INTERFACE_TIMEOUT	Timeout on the interface - no response occurred (The interface will be reset if possible)
0	INTERFACE_BUSY	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.

Disclaimer

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