

Milan / Paylink

Application Program Interface

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	5
Introduction	6
Purpose of Document.....	6
Intended Audience.....	6
Naming	6
Supported Facilities	6
Version Numbering.....	6
Associated Document(s)	7
Programming Language.....	7
Java / Dot Net / Managed storage.....	8
Low level definition of 'C' style.....	8
Document Layout	9
Getting Started.....	11
Installation	11
USB Device (Genoa)	11
USB Connectivity.....	12
PCI Card	13
Operation.....	13
OpenMHE.....	14
OpenSpecificMHE	15
EnableInterface	16
DisableInterface.....	16
CurrentValue	17
PayOut.....	17
LastPayStatus	18
CurrentPaid	19
IndicatorOn / IndicatorOff	19
SwitchOpens / SwitchCloses.....	20
Getting Started Code Examples.....	21
Currency Accept	21
Currency Payout	22
Indicator Example	23
Switch Example	23
Full Application.....	24
Background	24
Controlling an Acceptor	25
Acceptor Enable / Disable	25
AcceptorBlock structure / object	25
Controlling a Dispenser	27
DispenserBlock structure / object	27
Other Constants	28
Device Identity Constants	28
CurrentUpdates (1.10.4).....	29
ReadAcceptorDetails.....	31
WriteAcceptorDetails	31
ReadDispenserDetails.....	32
WriteDispenserDetails.....	32
Dispenser Value Reassignment (1.10.7)	32
Token Handling (Coin Ids) (1.11.x)	33
Dual Currency Handling (Coin Ids) (1.11.x)	33
Read out of Acceptor Details (1.11.x)	34
Read out of Dispenser Details (1.11.x)	34
Coin Routing.....	35

Route coins to a general cash box	35
Route specific coins to a specific cash box.	35
Route coins to a dispenser until it is full then route it to a coin cash box.	35
Paylink Routing - Flow Diagram	36
MDB changer support.	37
MDB tube level monitoring.....	38
Dispenser Power Fail support.	38
Combi Hopper Support.....	38
Multiple Paylink Unit Support.	39
Overview	39
Unit Identification	39
Detailed Device Support.....	40
Abandoning a payout in progress (1.11.3)	40
Control of a cctalk bulk coin acceptor (1.11.3)	40
Control of unwanted bill payout (1.11.3).....	40
Note Acceptor Escrow.....	41
Escrow Overview.....	41
Escrow system usage	41
EscrowEnable.....	42
EscrowDisable.....	42
EscrowThroughput	43
EscrowAccept.....	43
EscrowReturn	44
Bar Codes	45
BarcodeEnable	46
BarcodeDisable	46
BarcodeInEscrow / BarcodeInEscrowExt.....	47
BarcodeStacked / BarcodeStackedExt	48
BarcodeAccept	49
BarcodeReturn	49
Barcode Printing.....	50
BarcodePrint.....	50
BarcodePrintStatus.....	51
Bill Recycler Operation (1.12.3).....	52
Introduction.....	52
Security.....	52
Component Identity	52
Routing	53
Dispenser Destination.....	53
Routing Control.....	53
Dispenser Emptying	53
Full Dump.....	54
Partial Dump	54
Payout Progress.....	54
Cancelling Payout.....	54
Notification of progress	54
Power Fail.....	55
Temporary power interruption.....	55
Full power Failure	55
Unpaid Bills.....	56
Device Specific Functionality.....	56
Cashcode B2B-300.....	56
Cashcode B2B-60.....	56
Merkur 100.....	56
Innovative NV11 Recycler (DES).....	56
Innovative NV200 Recycler (DES).....	57
JCM UBA Recycler	57

CONFIDENTIAL

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

JCM Vega (DES) & JCM UBA Recycler	57
F56 / F53 Bill Dispenser	57
Meters / Counters.....	59
Mechanical Meters (1.12.4)	59
CounterIncrement.....	59
CounterCaption	60
CounterRead	61
ReadCounterCaption.....	61
CounterDisplay	62
MeterStatus	62
MeterSerialNo.....	63
E ² Prom	64
E2PromReset	64
E2PromWrite	64
E2PromRead	65
Utility Functions.....	66
CheckOperation (1.11.x)	66
NextEvent	67
NextAcceptorEvent NextDispenserEvent NextSystemEvent	68
SetDeviceKey	69
SerialNumber.....	69
FirmwareVersion	70
USBDriverStatus	71
USBDriverExit.....	71
IMHEIConsistencyError.....	72
Auditing / Event Processing	73
Event Codes used by NextEvent / EventDetailBlock	73
cctalk coin processing.....	74
cctalk note processing	75
cctalk hopper processing	76
ID-003 note processing.....	78
DES security (25-12-1)	79
Background	79
Key Exchange	79
Des Lock.....	80
DESSetKey.....	80
DESLockSet	81
DESLockClear	81
DESStatus	82
Engineering Support	83
WriteInterfaceBlock	83
ReadInterfaceBlock	84
Disclaimer	85

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.
1.8	11 th Mar 09	D Bush	Added 1.11.x Functions.
1.9	7 th Oct 10	D Bush	Added Future DES Facilities
1.10	2 nd Mar 11	D Bush	Updated to reflect other environments
1.11	24 th Dec 11	D Bush	Added 1.12.3 Recycler details
1.12	24 th Apr 12	D Bush	Added more recyclers Added NextxxxEvent calls Added mechanical Meters Added Barcode details to Acceptor block

Introduction

Purpose of Document

This document describes the software interface to the AES Intelligent Money Handling Equipment Interface (Paylink or PCI Card) 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 Paylink unit itself or will read the monetary information or diagnostic information provided by the card.

Naming

The system described here has a few names. This section attempts to explain them.

AES	Aardvark Embedded Solutions - us.
IMHEI	Intelligent Money Handling Interface Equipment. This was the original name for the project,. This was however difficult to say, and so was replaced in common use by Milan. It remains in the names in of the header files etc.
Milan	This was originally the name of the first hardware build. It has however become the name of the overall project. Most documents from AES talk about Milan to cover the whole family of products that are used with this API
Paylink	This is the name of the USB module made and sold by Money Control under licence from AES. There are at present three versions of Paylink: Paylink The original, metal cased version. Paylink Lite A much smaller, plastic cased version with a reduced function set. uPaylink (Micro Paylink) a PC software only version, for use with Money controls USB peripherals.
PCI Card	This is the original obsolescent hardware unit. It was known as Milan a long time ago, but this is its current name,

Supported Facilities

It should be noted that this document cover all versions of the Milan / Paylink system, even those versions that are not yet generally available.

Where a facility may not be available with the version that you are running, the topic titles are suffixed with a version indication in brackets

Version Numbering

All AES software releases have a 4 part version number. This is made up from 4 separate fields, coded as:

L L-P P-V V-M M

where:

- M M Is a minor release, representing an upgrade in facilities or bug clearance, but where the application code will remain the same (both source and executable).
- V V is a significant release, where the application will at a minimum need to be re-compiled, and where facilities may have changed to the point where the application code needs to change.
- P P Is a product code. This is 1 for Paylink and is 25 for DES Paylink
- L L Is the release level. This is a code, rather than a level, and has meaning as follows:
- 4 is a full release, and should never contain any errors or omissions. These release happen relatively rarely and a full history of the code is maintained. A code starting 4 uniquely identifies a particular build of the software.
 - 3 is a beta release. This may contain errors as it has not been fully regression tested, but it is intended to be sufficiently stable that development, or even live running is possible. Again, a code starting 3 uniquely identifies a particular build of the software.
Normally the full release of a version will be almost identical to the beta release.
 - 2 is an alpha release. These are only usually issued at the start of a major version. They *should* be stable and bug free, but are not fully tested, especially they will only have had minor regression testing. This release is to enable developers to “get started” with a new set of facilities. Again, a code starting 2 uniquely identifies a particular build of the software.
 - 1 is an engineering release. These are generated during our internal development process, and are occasionally released to customers in response to specific requests. A build code of 1 can only be distinguished by the date / time stamp embedded in the code, and no internal record is kept of the items / changes that have gone into such a build.

Where facilities described in this manual were added after the initial release or the Paylink project, the first version codes (P P-V V-L L) in which they were first implemented is given.

Associated Document(s)

As from release 1-12-1 (25-12-1 for DES Paylink), an essential part of the Paylink system is the configuration file. This is described in the associated document, “Milan / Paylink Configurable Driver User Manual”.

Programming Language

The majority of the users of Paylink use the C++ language, and the primary DLL interface uses C / C++ interface styles. This document therefore defaults to describing all the functions and data structures using C++ terminology.

The DLL entry point definitions are provided in a Visual Studio compatible library, **Aesimhei.lib**. A translation of this suitable for Borland Code Builder is available as **BorlandAESImhei.lib**.

However as Paylink is entirely independent of the language / environment used and the Paylink package also includes additional interfaces for these, where relevant, terminology for those is also included.

32 bit Visual Basic (VBA and VB6) and Delphi are capable of accessing these C++ structures and functions directly and so header files for these are provided for inclusion into the project.

Java, C# and VB.NET cannot easily access these and so interface libraries are included in the distribution that implement very similar classes in a manner compatible with the environments.

Java / Dot Net / Managed storage

Java, C# and VB.Net are all object oriented languages that use managed storage.

Paylink provides an interface library (AESImhei.java / AESImhei.net.dll) for these languages. The interface provided has the following features in all these languages:

- The interface provides a single object, AESImhei, has no properties and a set of static methods. These methods map directly onto the functions described in this document. As static functions, they are called by prefixing the function name with AESImhei, rather than creating an object of this type.
- The AESImhei object defines in turn a set of embedded objects, which have properties, but no methods. These objects map directly onto the data structures described in this document.
- Where this document refers to a 'C' style array (see below), the AESImhei object creates a managed array of the object so that access to these elements can be carried out as normal.
- Where the base functions in the C++ interface use "char*" strings (see below), these strings are automatically converted by the interface library into native string types.
- All objects in these languages are always "call by reference", in 'C' this has to be indicated by the inclusion of a "*" in the function definition - this "*" can be ignored by programmers in these languages.

Low level definition of 'C' style.

For people working in other languages wishing to understand the function descriptions and data structure excerpts in this document, and the original descriptions in the definitive C / C++ header file, the following type are used in both:

<code>int</code>	A 32 bit signed quantity (in other languages <i>Int32</i> or <i>Integer</i> .) Previous Paylink release used <i>long</i> but this is tending to denoting a 64 bit quantity.
<code>char</code>	An 8 bit quantity. This will be used to hold numbers in the range 0 to 127. This is sometimes <i>Byte</i> or <i>SByte</i> .
<code>char*</code>	A 'C' style string. In detail, this is the memory address of an area containing successive <i>chars</i> (bytes) with the end denoted by a <i>char</i> with a value of zero.

Arrays declarations in 'C' are of the form:

`Type Variable[Length]`

where *Type* is one of the above, and *Length* is the number of items in the array.

Where an array is used as a part of a structure, C++, and hence the underlying interface DLL, has the storage described actually within the structure at the point of the declaration.

When functions in this document use structures, they always suffix the structure name with a *. This accomplishes "Call by Reference" where the value passed is the memory address of the 1st byte of the structure.

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 Milan / Paylink system operates.

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

2. DES security (25-12-1 onwards)

These functions are those that enable an application to apply and check a security lock to a DES Paylink.

Using these calls, the software engineer can ensure that a DES Paylink and set of DES peripherals cannot be tampered with, or used by any other program / PC.

3. Full Application

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 application programmer could determine (say) the exact reason that an attempted payout failed and then notify either an engineer or a cash collector.

4. Utility Functions

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

5. Note Reader Escrow

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

6. Meters / Counters

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

7. E2Prom

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

8. Barcode Reading

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

9. Barcode Printing

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

10. Engineering Support

These functions provide full-blown diagnostics and reconfiguration facilities.

Getting Started

Installation

USB Device (Genoa)

The Milan / Paylink unit Genoa variant is a standard USB 1.1 peripheral.

Installation of the **Windows** driver is as with any USB peripheral, when the unit is detected the user is prompted to insert the installation CD or the access the Internet. As the drivers are signed and released to Microsoft they can be downloaded from the Internet, or they can be installed from the CD (image).

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

- The interface AESIMHEI.DLL needs to be copied from the installation CD to the hard disc. Normally this is to Windows\System32, but any directory on the system PATH can be used.
- The High Level driver program Paylink or AESCDriver 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 system design of Paylink and its support software expects this program to be running continually, if it is not then the latest Paylink units will continually turn their USB interface off and on in attempt to recover communications.
- For Java and dot Net, the appropriate interface specific DLL has to be copied to an appropriate location.

For **Linux** systems, the release is provided as an archive, containing the source of all the required PC software. This consists of three main elements:

- The low level USB interface library "libusb". This is unchanged by Aardvark, but provided so as to ease the installation process.
- FTDI specific drivers "libftdi" that allow the Aardvark code to easily access the libusb library. Again none of the code has been modified by Aardvark.
- The shared library interfaces, the USB driver programs and the examples.

Having unpacked the supplied archive into a convenient folder, the **Install.sh** script should be run.

This will check for the existence of libusb and libftdi. If a version is not already installed, then the supplied version are built and installed automatically. If installed versions are found, then the script allows the user to decide whether or not to change then.

Finally the script will build all the Paylink components and install the programs (in /usr/local/bin) and shared libraries (in /usr/local/lib).

USB Connectivity.***USB Lead***

The Paylink unit is not designed to be added to and removed from a PC, it is designed to be permanently connected. Any disconnection and reconnect of the lead will at least cause exception processing, may cause the Paylink unit to be reset and may cause the interface presented to the PC to change drastically.

USB Driver Program.

On the target system, the supplied USB driver program Paylink or AESCDriver should be regarded as a system service and unconditionally started at system boot. It is possible to stop and start this program, but that causes exception processing to be undertaken as above and is not recommended.

The system should not be regarded as usable for 20 seconds after a Paylink reset, or for 10 seconds after a USB driver program (re-)start.

PCI Card

The obsolescent PCI 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 Windows 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.

Note that PCI Cards do not support version 1.11.1 and later, and do not have any interface to a Linux system.

Operation

The Milan / Paylink 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 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 section describes the minimum set of function calls needed to implement a useful 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 application to open the “Money Handling Equipment” Interface.

int 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. <i>or</i> 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) normally there is only one IMHE Interface unit in a system. Hence any “Open” call refers 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 application 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.

C++

```
int OpenSpecificMHE (char SerialNo[8]);
```

C#, VB, Java

```
int OpenSpecificMHE (string SerialNo);
```

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")**, used with a driver program specified to communicate with Serial Number 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 “turn on” the Milan system. This would be called when the system is initialised and ready.

Until this call is made, none of the functions concerning the operation of the Paylink unit will work. This includes such thing **CheckOperation** and items such as switches and meters function.

If you wish to start running with acceptance disabled, this call should still be made and money acceptance disabled as in the section *Acceptor Enable / Disable* on page 25 below in the part headed *Controlling an Acceptor*.

```
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 provided for completeness, and “closes down” the Milan system, prior to a system close down. Note that items such as switches and meters will also cease functioning at this point. To just disable money acceptance see the section *Acceptor Enable / Disable* on page 25 below in the part headed *Controlling an Acceptor*.

```
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.

int 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 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 unit.
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 application to instruct the interface to pay out coins (or notes).

void PayOut (int 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.

LastPayStatus

Synopsis

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

```
int 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
-9	The hopper reset during a payout	PAY_HOPPER_RESET
-10	The hopper cannot payout the exact amount	PAY_NOT_EXACT
-11	This hopper does not really exist.	PAY_GHOST
-12	Waiting on a valid key exchange	PAY_NO_KEY

Remarks

1. Following a call to **PayOut**, the programmer should poll this to check the progress of the operation.
2. If one or more of multiple hoppers has a problem, the Paylink 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.

int 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. The 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 application to control LED's and indicator lamps connected to the interface.

void IndicatorOn (int IndicatorNumber) ;
void IndicatorOff (int 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 application to read the state of switches connected to the interface card.

```
int SwitchOpens (int SwitchNumber) ;  
int SwitchCloses (int 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 'C' code fragments are intended to provide clear examples of how the calls to the Paylink 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)
{
    int LastCurrencyValue ;
    int NewCurrencyValue ;

    int 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)
{
    int 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)
{
    int 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)
{
    int 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 Application

Background

When implementing a full application 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 *nothing* in *any* control block 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 sequence 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.

Controlling an Acceptor

For an acceptor, the relevant control block usually involves matching on 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.

The **AcceptorBlock** contains various fields that are relevant to the acceptor as a whole, and an embedded array of **AcceptorCoins**, which contain fields relevant to individual coins (or notes.)

To control an acceptor, the correct method is to find the acceptor is question with **ReadAcceptorDetails()** (see below), modify one or more field and then write the modified data back using **WriteAcceptorDetails()** (see below)

Most fields within the **AcceptorBlock** are self-explanatory, or are adequately described by the comment included in the header file included below.

A few fields are worthy of a more detailed explanation are:

- Status:** Each bit of this field has a specific use. The bits are itemised below and are named `ACCEPTOR_XXX` where xxx is the usage.. Note that `ACCEPTOR_INHIBIT` is, uniquely, set by the application and read by Paylink.
- NoOfCoins** This specifies the number of entries in the **Coin** array that are used. Later entries are neither written to nor read.

Fields with names involving "Path" are concerned with coin routing, a detailed description of which is given in the section "**Coin Routing.**" below.

Acceptor Enable / Disable

Enabling / disabling money acceptance is performed as with any other acceptor control. The specific operation is to ensure that the *bit* `ACCEPTOR_DISABLED` is set in the **Status** field (usually by using the OR operation) before writing back the modified data.

When reading the AcceptorBlock for an acceptor that is disabled, this bit will already be set. To enable an acceptor it will be necessary to clear this bit ((usually by using an AND operation) before writing back the modified data.

It is acceptable practice to disable or enable all acceptors in the system, without paying any attention to any other information about them.

AcceptorBlock structure / object

Constants for Acceptors

Name	Meaning
ACCEPTOR_DEAD	No response to communications for this device
ACCEPTOR_DISABLED	Disabled by Interface
ACCEPTOR_INHIBIT	Specific by Application
ACCEPTOR_FRAUD	Fraud attempt has been detected
ACCEPTOR_BUSY	the Device is reporting itself as busy
ACCEPTOR_FAULT	The Milan unit cannot communicate with the device
ACCEPTOR_NO_KEY	The Milan unit needs to acquire a DES key from the unit
MAX_ACCEPTOR_COINS	The absolute Maximum coins or notes handled by any device

Fields / properties for the AcceptorBlock Object

Field Type	Name	Meaning
int	Unit	Specification of this unit
int	Status	AcceptorStatuses - zero if device OK
int	NoOfCoins	The number of different coins handled
int	InterfaceNumber	The bus / connection
int	UnitAddress	For addressable units
int	DefaultPath	The default routing for coins
int	BarcodesStacked	The total number of barcode tickets stacked by this acceptor
char[4] string	Currency	Main currency code reported by an acceptor
AcceptorCoin	Coin[MAX_ACCEPTOR_COINS]	The coins (only NoOfCoins are set up)
int	SerialNumber	Reported serial number (0 if N/A)
char* string	Description	Device specific string for type / revision / coin set
int	EscrowBarcodeHere	If this is non zero, then the barcode reported by BarcodeInEscrow is from this acceptor

Fields / properties for the AcceptorCoin sub object

Field Type	Name	Meaning
int	Value	Value of this coin
int	Inhibit	If set non zero by the PC: this coin is inhibited
int	Count	Total number read "ever"
int	Path	Set by PC: this coin's chosen output path
int	PathCount	Number "ever" sent down the chosen Path
int	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* string	CoinName	A string, usually as returned from the acceptor, describing this coin

Note that although the routing fields, Path and DefaultPath, are defined as "set by PC", they are, where possible, initialised to the values read from the acceptor.

Controlling a Dispenser

For a dispenser, the relevant control block usually involves matching on the `value` field as that shows the coin value being used by the Paylink unit, which is the most important distinguishing feature of a dispenser.

The `Inhibit` field of a dispenser can be set a value other than zero to prevent Paylink from trying to use that dispenser to satisfy a Payout request. Note that setting this field has actual effect on the peripheral itself.

The `status` field of a dispenser is only updated when an attempt is made to pay out from the dispenser. In particular a `status` of `PAY_EMPTY` will only be cleared when a successful pay out is performed. In addition, certain “exotic” operations on a Dispenser can be triggered by the application writing codes into the `status` field.

DispenserBlock structure / object

Constants for Dispensers

Name	Meaning
MAX_DISPENSERS	Maximum handled
Coin Count Status Values	
DISPENSER_COIN_NONE	No dispenser coin reporting
DISPENSER_COIN_LOW	Less than the low sensor level
DISPENSER_COIN_MID	Above low sensor but below high
DISPENSER_COIN_HIGH	High sensor level reported
DISPENSER_ACCURATE	Coin Count reported by Dispenser
DISPENSER_ACCURATE_FULL	The Dispenser is full
Special Status Values	
DISPENSER_REASSIGN_VALUE	The Value has just been updated by the application
DISPENSER_VALUE_REASSIGNED	The updated Value has just been accepted by Paylink
DISPENSER_CASHBOX_DUMP	Dump the hopper if you can (for note recyclers only)
DISPENSER_PARTIAL_DUMP	Dump some of the hopper if you can
DISPENSER_DUMP_FINISHED	A Recycler dump has just complete

Fields / properties for the DispenserBlock

Field Type	Name	Meaning
int	Unit	Specification of this unit
int	Status	Individual Dispenser status This takes the same values as <code>PayStatus()</code>
int	InterfaceNumber	The bus / connection
int	UnitAddress	For addressable units
int	Value	The value of the coins in this dispenser
int	Count	Number dispensed according to the hopper records
int	Inhibit	Set to 1 to inhibit Dispenser
int	NotesToDump	Used in conjunction with <code>DISPENSER_PARTIAL_DUMP</code>
int	CoinCount	The number of coins in the dispenser
int	CoinCountStatus	Flags Relating to Coin Count (See above)
int	SerialNumber	Reported serial number (0 if N/A)
char* string	Description	Device specific string for type / revision

Other Constants

The following is the description of the structures / object used by this interface library. For full details of the structures the reader is referred to the various header files / interface libraries distributed in the SDK folder of the Milan software releases.

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**

From this, you can create some simple classification functions, e.g.

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

For the complete list of device specific constants the reader is referred to the AESImhei.h 'C' header file, or its language specific equivalent.

The general components of these identities are:

Name	Value	
DP_GENERIC_MASK	0xff000000	The mask to extract the following parts
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	
These describe the interface via which this device is connected:		
DP_INTERFACE_MASK	0x00ff0000	The mask to extract the following parts
DP_INTERFACE_UNIT	0x00000000	
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	
DP_F56_INTERFACE	0x000A0000	
DP_CCNET_INTERFACE	0x000B0000	
These describe the manufacturer of the device.		
DP_MANUFACTURER_MASK	0x0000ff00	The mask to extract the following parts
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	
DP_MANU_MERKUR	0x00000C00	
DP_MANU_FUJITSU	0x00000D00	
DP_MANU_CASHCODE	0x00000E00	
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 DP_COIN_PAYOUT_DEVICE Used by Value hopperz

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.

int CurrentUpdates (void) ;

Parameters

None

CONFIDENTIAL

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.

```
int ReadAcceptorDetails ( int          Number ,  
                          AcceptorBlock* Snapshot ) ;
```

Parameters

1. Number
The sequence 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

Non zero if the specified input device exists, Zero if the end of the list is reached.

Remarks

The sequence 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 ( int          Number ,  
                            AcceptorBlock* Snapshot ) ;
```

Parameters

1. Number
The sequence 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 sequence 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.

```
int ReadDispenserDetails(    int                Number,  
                             DispenserBlock* Snapshot) ;
```

Parameters

1. Number
The sequence 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

Non zero if the specified input device exists, Zero if the end of the list is reached.

Remarks

The sequence 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( int                Number,  
                             DispenserBlock* Snapshot) ;
```

Parameters

1. Number
The sequence 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 sequence 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 an application 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) 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 or notes of more than one currency, the application can determine the currency of a specific coin or note by examining the first characters of the name of the coin (**Coin.CoinName**).

Note: The exact values returned are dependent upon the acceptor manufacturers and hence can not be given here.

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 returned by the Get Currency Assignment (0x8A) command. The first three characters are the decimal 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	All MDB coins are the same currency. The coin name contains the Value as a decimal number, followed by a * followed by the (constant) Scaling as a decimal number
CCNet	This is set from the Get Bill Table (41H) command. The string is the 3 chars from the 3 byte "Country Code" followed by the decoded value as a decimal number.

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 Revision / Issue (145 / 96+243), Request Currency Specification ID / Code (91 / 96+244), Request Software Revision (241) & Request Product Code (244) commands, separated by '~' characters. Each individual field 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	From the Status and Extended Identification Commands <ul style="list-style-type: none"> Country Currency Code (4 BCD characters) Decimal Places (1 Character) Manufacturer (3 Characters) Model Number (12 Characters) Software Version (4 characters) separated by '~' characters.
CCNet	This is the 15 character "Part Number" from the "Identification" (37H) command.

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	Bytes Z4-Z15 from the Extended Identification Command, converted from decimal characters to a number.
CCNet	The "Chassis Serial Number" from the Module Identification Request (53H0 command, converted from decimal characters to a number.

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	N/A (Integral part of Acceptor)
F56	The 12 character firmware revision, followed a '~' followed by the 32 character device information.

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

ccTalk	The binary reply to the ID Serial No (242) command.
MDB	N/A (Integral part of Acceptor)
F56	Not Available

Coin Routing.

Paylink provides facilities to partially automate the routing of coins to fill a coin dispenser and one or more cash boxes. This enables Paylink to accurately change the routing in the potentially very small delay between one coin and the next.

These facilities only apply to coins. The routing for notes is completely left to the application, as there are no time constraints.

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.

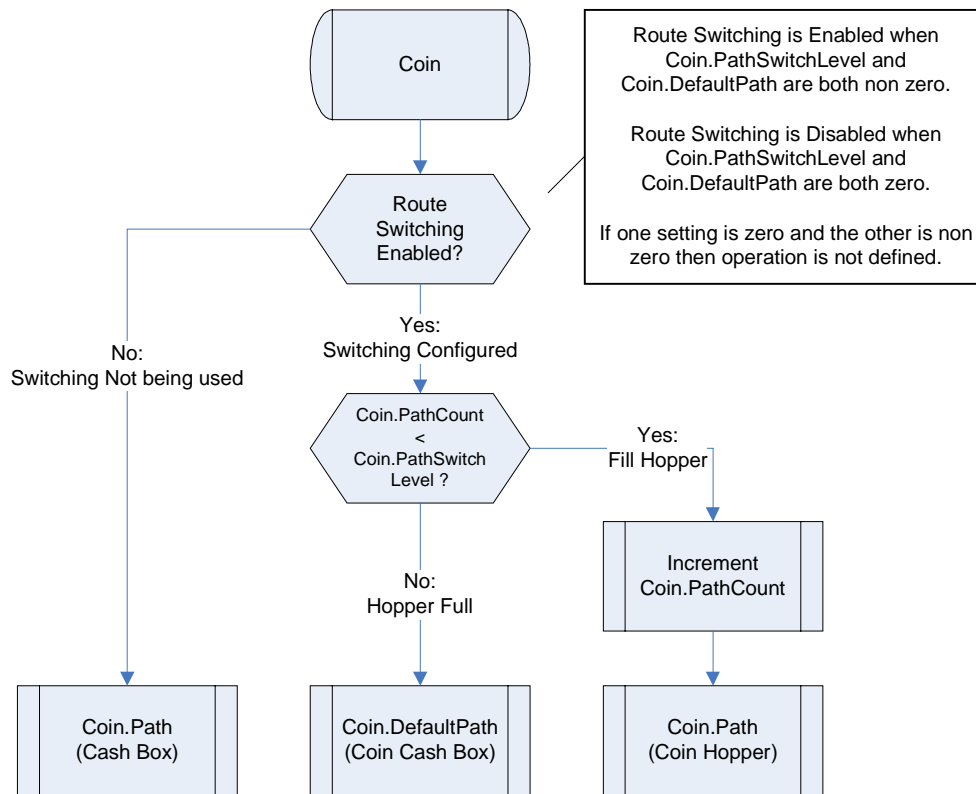
- Set Coin.Path to the dispenser route for each coin that is routed to a hopper.
- Set Coin.DefaultPath to the cash box route for each coin that is routed to a hopper. **This must be non zero.**
- Set Coin.PathSwitchLevel to the Coin.PathCount value at which the dispenser becomes full. **This must be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

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 already shown in the acceptor detail block for the changer, the extended mechanism dispenser has a **Unit** field with the constant value of **DP_MDB_TYPE_3_PAYOUT** while the individual tubes have **Unit** fields with the constant value of **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. . (Note that after a delay of up to 25 seconds this will then be replaced by the value from a Tube Status command)

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 value for the levels

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 then 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.

Some dispensers, especially hoppers produced by MCL and some bill recyclers, 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 Paylink start-up initialisation to correspond to the "total coins paid since manufacture" value retrieved from the hopper, and is then updated as payouts occur. 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.

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_SCH3A, 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 around the idea 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 the program with a specific one of the multiple Paylink unit interface areas.

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 (Windows) **PaylinkSerial** utility is available as a part of the released SDK, which takes as a parameter a serial number and programs it into the **only** Paylink unit currently connected to the system.

Detailed Device Support.

Abandoning a payout in progress (1.11.3)

As well as preventing a payout operation from starting, the Inhibit field in a dispenser is now used during an actual payout. If the application set a dispenser inhibit while a payout it is in progress, Paylink will attempt to abandon the payout in progress on that device.

Note that the overall payout will still continue on all the other dispensers that are not inhibited. To cancel an entire payout the application should inhibit all dispensers.

Control of a cctalk bulk coin acceptor (1.11.3)

Paylink will automatically send the required command to operate the motor on a bulk coin acceptor, but the application may want to trigger the special "reject clearance mode". This is requested by inhibiting all the coins for the acceptor, and inhibiting the acceptor itself. (To just stop accepting coins, it is only necessary to inhibit the acceptor)

Control of unwanted bill payout (1.11.3)

Under failure conditions a number of bill handling systems can enter a state where bills are not accessible to the end user, but cannot be returned to a cash / reject location. When Paylink detects these circumstances, it will pause its operation, queue a **IMHEI_NOTE_DISPENSER_PENDING** event with the number of bills as the RawEvent field and automatically set an inhibit on.

The bills can be delivered by clearing the inhibit for all the dispensers that form part of the unit.

Note Acceptor Escrow

Escrow Overview

Almost all note acceptors provide a facility known as “escrow” , whereby after the note has been identified, it is held within the acceptor and can then be either returned to the user or fully accepted and stacked (if a stacker is in use).

Paylink *always* uses this facility (unless it is not available) as it provides enhanced security during the note acceptance process. (As Paylink issues an accept for each note from escrow it therefore *has* to have accurately accounted for how many it has read.)

By *default*, the Paylink system automatically issues an accept command as soon as note is reported in the escrow position.

If the application wishes to have more control over the acceptance process, Paylink provides “note in escrow” facilities that allow the Application full control over whether to issue an accept command or to issue a reject command.

Escrow system usage

The Paylink unit fully supports escrow system usage. It reports the note that is currently held in escrow by an acceptor, and allows the application to either return or accept the escrow holding of the acceptor.

In most system only one escrow capable acceptor will be present, the Paylink unit will however support escrow on an unlimited number of acceptors. In order to allow for accurate information and control to pass between the application and the Paylink 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. For the application 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 Paylink unit 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 application. Following either call, the **EscrowThroughput** value may increase immediately due to another acceptor having an escrow holding.

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

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.

int 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.

Bar Codes

Where an acceptor provides barcode facilities, the Paylink unit can 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, in a similar way to that already described. The barcode is held in the acceptor pending a call from the application that will either stack or return it.

In most systems, only one barcode capable acceptor will be present, the Paylink unit will however support barcodes on an unlimited number of acceptors. In order to allow for accurate information and control to pass between the application and the Paylink 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.

The basic barcodes processed by the IMHEI system are in the format "Interleaved 2 of 5" and are 18 characters long. The basic functions therefore return a 19 character NULL terminated string.

Later barcode systems now return up to 40 characters, so API functions with the suffix Ext have been added, which will handle any length of barcode.

Basic Barcodes can also be printed if a dedicated barcode printer is connected.

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 / BarcodeInEscrowExt

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.

Originally, barcodes from note acceptors were always 18 characters plus a trailing NULL, recently the size of the barcode has increase, so a new function is provided to handle these larger barcodes.

C++

```
int BarcodeInEscrow (char BarcodeString[19]) ;  
int BarcodeInEscrowExt (char* BarcodeString,  
                        int    BufferLength) ;
```

C#, VB

```
int BarcodeInEscrow (ref string BarcodeString) ;  
int BarcodeInEscrowExt (ref string BarcodeString,  
                        int    BufferLength) ;
```

Java

```
int BarcodeInEscrow (string BarcodeString[1]) ;  
int BarcodeInEscrowExt (string BarcodeString[1],  
                        int    BufferLength) ;
```

Parameters

1. BarcodeString
A pointer to a string / buffer of at least 19 (or BufferLength) 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.
2. BufferLength
The length of the provided buffer.

Return Value

The return value is non zero if there is a barcode ticket currently held in an Acceptor, zero 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.
2. Whilst this function is returning no zero, then *exactly one* of the Acceptors accessed by ReadAcceptorDetails will have an EscrowBarcodeHere field with a non-zero value.

BarcodeStacked / BarcodeStackedExt

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**.

Originally, barcodes from note acceptors were always 18 characters plus a trailing NULL, recently the size of the barcode has increase, so a new function is provided to handle these larger barcodes.

C++

```
int BarcodeStacked (char BarcodeString[19]) ;  
int BarcodeStackedExt (char BarcodeString[19],  
                      int BufferLength) ;
```

C#, VB

```
int BarcodeStacked (ref string BarcodeString) ;  
int BarcodeStackedExt (ref string BarcodeString,  
                      int BufferLength) ;
```

Java

```
int BarcodeStacked (string BarcodeString[1]) ;  
int BarcodeStackedExt (string BarcodeString[1],  
                      int BufferLength) ;
```

Parameters

1. BarcodeString
A pointer to a string / buffer of at least 19 (or BufferLength) 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.
2. BufferLength
The length of the provided buffer.

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

1. 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.
2. 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.
3. Whenever the value returned by this is incremented, then *exactly one* of the Acceptors accessed by ReadAcceptorDetails will have a BarcodesStacked field that also increments.

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 ticket.

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 ticket.

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 Paylink 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).

Fields / properties for the TicketDescription Object / Structure

Field Type	Name	Comment
int	TicketType	The "template" for the ticket
char* / string	BarcodeData	
char* / string	AmountInWords	
char* / string	AmountAsNumber	But still a string
char* / string	MachineIdentity	
char* / string	DatePrinted	
char* / string	TimePrinted	

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.

```
int 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.

Bill Recycler Operation (1.12.3)

Introduction

The original Paylink model was based around the concept of independent acceptors and dispensers, with the main specification for a dispenser being the MCL / cctalk hopper.

The advent of bill / note recyclers has meant that the Paylink model has had to be enhanced to include these. The approach adopted has been based around the idea that bill / note recycler (and the coin changer MDB device) is a combination in a single unit of an acceptor and number of dispensers.

This section describes the evolved handling of recyclers (and bill dispensers) that is in place in versions 1.12.3 and later.

Security

The connections used by Paylink fall into three categories, cctalk, MDB and RS232. These communications systems tend to match up with three different markets:

RS232 is used with a number of protocols to connect expensive bill acceptors / recyclers for

applications that are typically in expensive, secure enclosures. Any interference with this connection will generally tend to be visible to the Paylink application.

MDB is used in very cheap systems, typically vending machines. The connection is vulnerable to interference, but the amounts of money involved tend to be low.

cctalk is a very versatile system, which can be used in high value applications. The connection is relatively vulnerable to interference and so encryption is used to provide security. The latest security option for cctalk is **DES** encryption, which involves the exchange of local, random keys.

Paylink has, as a philosophy, the idea that it will connect to anything. Using **DES** encryption for a bill and a coin acceptor do not therefore make sense and people wishing to create a fully secure system using these are referred to the DES Paylink system (which is closed down and secure)

A **DES** bill recycler however makes sense, as the locked down aspect here is in the peripheral - Paylink can connect to any recycler, but the recycler will only communicate with Paylink. Given the vulnerability to high value fraud of a cctalk recycler, Paylink therefore *insists* that all new cctalk recyclers supported must use **DES** encryption.

As a special case the Merkur recycler is supported - this is not a general purpose protocol and Merkur recyclers are not expected to be used in vulnerable systems.

Component Identity

The general approach to identifying these devices is that the acceptor part usually contains the overall description of the unit, and the dispensers are (relatively arbitrarily) identified by a sequence number (from 1 upwards) and the value that they dispense.

Where necessary a dispenser can be tied to acceptor by type and by having the same serial number. The `DispenserBlock.m_UnitAddress` field(s) will contain a "sequence" number that identifies the dispenser. If the unit only has one dispenser, the field will contain 1.

On multi-drop system, such as cctalk and MDB, the address of the parent acceptor will be OR'ed with this sequence number so that multiple units can be distinguished.

Routing

Dispenser Destination

Paylink, during its initialisation of the unit, determines the value of the coin / bill in the dispenser(s) and which coins / bills are routed into which dispensers. The “sequence” number(s) are stored into the `AcceptorCoin.Path` field(s). All the other `AcceptorCoin.Path` fields will be zero.

When a coin / bill is accepted and routed into a dispenser this fact is always identified by Paylink and the `AcceptorCoin.PathCount` is accurately incremented to show this.

Paylink then updates the `DispenserBlock.CoinCount` field by actually querying the unit. Depending upon the actual unit this will be either accurate or an approximation. With a bill recycler the result is usually an accurate figure, with an MDB changer the result is often approximate.

The value returned will however *always* be that reported by the device, any systematic corrections will have to be handled by the application.

Routing Control.

For some bill recycler units, such as the Merkur MD100 and MDB Changers, the routing is fixed and it is not possible for Paylink, and hence the application, to change this.

For other units, the routing can be changed by Paylink. The application notifies Paylink of the desired routing by changing the Path fields of the incoming Coin (Bill) array item to contain the associated dispenser serial number.

Options available are:

- If the Path field for a currently recycled bill is set to zero, the unit will stop diverting bills into the recycler. If there are no bills stored, then the Dispenser value will go to 999999999, (this is irrelevant to payouts, as the dispenser will return “empty” if it is attempted to be used), if bills are currently stored they may remain available to be paid out (depending upon the device capabilities)
- The application should not set two or more separate bills to contain the same value in the Path field, if this is done, then this situation is undefined
- For all coin / bills with a non-zero path number, if the path field for a bill corresponds to the sequence number for a dispenser, then Paylink will update the recycler to direct that bill into the corresponding dispenser. As a part of this process, many units will cause any bills already in the dispenser will be stored into the cashbox.

Note that as the specification is “where to send the bill” it is not possible to have two dispensers regarded as having the same value bill.

The options for automatic re-routing using `DefaultPath` and `PathSwitchLevel` are only available with coin acceptors. For note recyclers, these fields are not used.

Dispenser Emptying

The high value represented by the bills in recyclers means that the dispensing of bills requires the interaction of the recipient. The high value of the bills stored in the recycler also means that users are liable to want to empty them at the end of the day.

These two factors mean that bill recycler manufacturers implement a “dump to cash box” facility so that the bills can easily be retrieved.

Full Dump

A full dump is where the recycler takes every bill from a dispenser into the cash box until the dispenser registers as empty.

Triggering this is implemented on Paylink by the user setting a `DispenserBlock.Status` value of `DISPENSER_CASHBOX_DUMP`.

On recyclers that maintain guaranteed accurate counts of bill, the application can monitor the dump process by observing the `DispenserBlock.CoinCount` going to zero.

On both these and other recyclers, the application can check for the `DISPENSER_CASHBOX_DUMP` value being replaced by another status. Where the dump process completes normally, the status will take value of `DISPENSER_DUMP_FINISHED`.

Partial Dump

As well as the above facility to cycle every bill into the cashbox, many recyclers provide the ability to perform a partial cashbox dump processes, which can be used to leave a “float” of bills in the recycler.

Triggering this is implemented on Paylink by the user setting the count of bills that are to be dumped in the new `DispenserBlock.NotesToDump` field and a value of `DISPENSER_PARTIAL_DUMP` in the `DispenserBlock.Status` field.

The application can usually monitor the dump process by observing the `DispenserBlock.CoinCount` field reducing by the requested amount.

The application can check for the `DISPENSER_PARTIAL_DUMP` value being replaced by another status. Where the dump process completes normally, the status will take value of `DISPENSER_DUMP_FINISHED`.

Payout Progress

Cancelling Payout

Bill recyclers / dispensers in general hold the bills awaiting collection by the user, and Paylink does not regard the Payout as complete until the bill has actually been taken. If the application program decided that the bill has been forgotten, it can abandon the payout by setting the inhibit flag on the dispenser. This will cause Paylink to request that the recycler abandons the payout and returns the bill to the cash box.

Note that following the abandonment of the bill payout Paylink will automatically proceed to attempt payout in coins, so in the usual case all the coin dispensers should be disabled at the same time as the bill recycler.

Notification of progress

While a bill recycler / dispenser is holding a bill awaiting collection by the user, Paylink does not regard the Payout as complete. The fact that the bill is available to be taken is however possibly of significance to the application, and therefore Paylink will update the `DispenserBlock.Count` and `DispenserBlock.CoinCount` fields for the relevant dispenser as soon as the bill is accessible. They will not then change when it is taken.

Power Fail

Temporary power interruption

Should the power / communications to the recycler fail during a payout while Paylink continues to run, Paylink will initially just wait for the communications to restart, and will then continue as though there has been no interruptions.

Where an interruption lasts “a long time” then Paylink will abandon the payout attempt. This will result in a dispenser / payout status of PAY_US. If at the time the payout is abandoned, Paylink is aware of a bill awaiting collection by the user, it will be regarded as having been paid out. It will not therefore be substituted by coins and can result in a normal payout completion status.

After the timeout, if / when normal communication with the recycler is resumed, Paylink will check the current status of the unit.

- A bill that was paid and collected during the interruption will cause the `DispenserBlock.Count` field to be incremented by the appropriate amount and a `IMHEI_NOTE_DISPENSER_UPDATE` event entered in the `NextEvent()` queue.
- A bill that has been sent from a dispenser to the cash box (as a part of the start up recovery process) will merely result in the `DispenserBlock.CoinCount` being updated.
- A bill that was awaiting collection, and has still not been taken, will cause the dispenser status to change to `PAYOUT_ONGOING` until it is eventually collected. This will have no effect on the `DispenserBlock.Count` field or Payout system.
- A bill that was awaiting collection but is **now** known to have been automatically recycled to the cash box, will cause the `DispenserBlock.Count` field to be decremented (to “undo” the payout) and a `IMHEI_NOTE_DISPENSER_UPDATE` event entered in the `NextEvent()` queue.

Full power Failure

Should the power to PC and recycler fail during a payout the application may wish to reconstruct the partial results of the last payout attempt. To facilitate this, Paylink will attempt to handle the interrupted payout according to the above rules

To do this requires that the `DispenserBlock.Count` field be maintained over power cycles - thus applications that so desire can record the `Count` fields before a payout is started, and then react accordingly if on startup they discover that a payout was in progress.

With coin hoppers, the speed of the payout means that the only place where an accurate record of interrupted payouts can possibly be obtained is in the hopper itself. Note dispensers typically do not provide such facilities, and Paylink therefore maintains the record itself.

The net result is that lifetime totals are maintained and reported in the `Count` fields, which are retrieved and updated by Paylink depending on the data read from the device during startup.

If this startup processing causes Paylink to suspected an uncompleted payout actually completed, an `IMHEI_COIN_DISPENSER_UPDATE` event is entered in the `NextEvent()` queue.

Unpaid Bills

Some devices (e.g. the B2B-300 bill recycler and F56 bill dispenser) have a delivery stage where bills are accumulated for eventual payout.

Following a power failure, it can happen that bills are in this output stage and are inaccessible to the user. The only thing that Paylink can do at this point is to complete the delivery of these bills, but as there is a potentially long time since the application requested the dispense, it is inappropriate to just deliver them at power up.

(Some F56 / F53 models also have a problem that following a power failure during a dispense there can be an unknown number of bills awaiting delivery.)

In these cases, for each dispenser device that is believed to have notes ready for delivery, Paylink marks the device inhibited, and generates an **IMHEI_NOTE_DISPENSER_PENDING** event with the number of bills as the RawEvent field. If the number of bills is unknown then 99 is used.

To complete the delivery process, the application should clear the inhibit on **all** the relevant dispensers.

Device Specific Functionality

The above description is the ideal that Paylink strives to achieve. The actual functionality provided by specific devices can however interfere with this, so all supported models of note / bill recycler are itemised here:

Cashcode B2B-300

The Cashcode B2B-300 accumulates bills to be dispensed in separate unit before presenting them to the user. When bills are “found” in the dispenser during startup, the only thing the unit *can* do is to dispense them.

When Paylink discovers this situation, during startup, or following a “long” power fail, it will undertake **Unpaid Bill** processing as above.

Cashcode B2B-60

The Cashcode B2B-60 operates by paying bills one at time for retrieval through the acceptor. To allow for full control in the event of a power failure / connection problem, Paylink runs the acceptor so that each note is a separate transaction.

Merkur 100

This recycler automatically restarts a payout on power up, unless a software reset is issued before the acceptor reaches the point at which the delivery is under way.

During the startup process, Paylink issues such a reset, so if the two units power up approximately at the same time, no spurious bill is paid. If this succeeds, then any bills “in progress” will be returned to the stacker.

Innovative NV11 Recycler (DES)

This is a standard, single bill recycler, with no special features.

Innovative NV200 Recycler (DES)

This recycler by design stores all bills into a single storage space. To allow for control over the payout operations, Paylink treats each denomination as stored into a separate “dispenser”; so each denomination is set up as routing into a matching dispenser.

To stop storing a particular denomination, the routing can be zeroed - and to empty all bills of a particular denomination into the cash box, the corresponding dispenser can be dumped.

JCM UBA Recycler

This unit can have bills “manually” loaded into the storage stackers. When this occurs, the UBA unit doesn’t know about the event and so does not update its internal counts, these are only updated when bills that have been accepted are stacked.

Similarly, if bills that have been stacked automatically are manually taken, the counts are not reduced.

The counts returned by Paylink are those from the UBA unit, and so under these circumstances will be incorrect - it is up to the application to compensate for those bills it knows have been manually inserted.

When the counter of the number of bills in a stacker reaches zero Paylink will still attempt to pay bills - if this succeeds the UBA counter will remain at zero - it will not go negative.

Similarly, if the stacker runs out of bills when the UBA counter is non-zero, the UBA will zeroize the counter.

JCM Vega (DES) & JCM UBA Recycler

These recyclers can retrieve a bill waiting for collection, and send it to the cash box.

If the dispenser is inhibited during a payout then, as well as preventing further payouts, Paylink will actually retrieve the bill waiting for collection. As this occurs after the bill has been accounted for, both the `DispenserBlock.Count` field and the `CurrentPaid()` return value will decrement.

This is command also used in those recovery situations where the application has not been informed that the bill has been dispensed.

This can help avoid the situation where payouts that were uncompleted actually occur.

F56 / F53 Bill Dispenser

The F56 device comes with a number of different options. Although the F53 is a different device number, Paylink just regards it as another option of an F56. All descriptions are therefore of an F56.

These F56 have a number of unique characteristics:

- The F56 only reports cassettes that are present, the existence of a location for a cassette is not discoverable. Paylink therefore only reports the status of cassette locations in which it has seen a cassette.
- A cassette can have a pattern of magnets set into it to indicate the type of bills with which it is loaded. The F56 configuration can include bill descriptions corresponding to these magnet patterns, which can specify value, bill length and bill thickness. If a newly discovered cassette matches such a pattern specification, then the bill value and sizes are set from the specification.
- If no magnet specification is given, or if there is no match, then the sizes default to a generic accept all size and the value is set as 999999999. This can be overridden to its correct value using the standard Paylink facilities.
- A pool area / note delivery option is possible, with delivery to the front or to the rear. Part of the configuration specification of an F56 has to include whether or not a delivery option is fitted.

- The F56 records in non-volatile memory the number bills delivered from a payout position. This value is reported to the application in the `DispenserBlock.Count` field.
- Some F56 models allow for the recovery of a failed dispense operation - on others this information is not available. Where this information is not available and the unit has a final dispense stage, then the notes are left in the pool area and **Unpaid Bill** processing performed.
- An F56 can be fitted with a shutter at the bill delivery stage. Paylink will automatically send a close shutter command when bills have been taken from the delivery stage by the user.

Meters / Counters

The Paylink units support the concept of external meters that are accessible from the outside of the PC system.

In keeping with the Paylink concept, an interface is defined to an idealised meter. This will be implemented transparently by the card using the available hardware. Currently the Paylink unit will support either a **Starpoint Electronic Counter**, or from 1 to 8 mechanical meters.

Mechanical Meters (1.12.4)

From 1.12.4 onwards, Paylink supports mechanical meters, driven using pulses through the general-purpose high power outputs. Suitable meters are required to operate on DC at 20 pulses per second or faster.

Configuration file entries are used to map Counter Numbers 1 to 8 onto the Paylink outputs.

Paylink records how many pulses have been sent, and how many are currently required. It attempts to handle the fact that while the pulses are being output the power may be cycled. Paylink updates its non-volatile memory as it turns on the transistor at the start of the pulse. This means that during a power cycle at most one pulse may be lost (as it is not driven for long enough) but no spurious pulses can be generated.

CounterIncrement

Synopsis

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

```
void CounterIncrement(int CounterNo,  
                      int 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 by the current hardware, 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.

C++

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

C#, VB, Java

```
void CounterCaption(int CounterNo,  
                   string Caption);
```

```
void CounterCaption(long CounterNo,
```

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.
4. This is not relevant for mechanical meters.

CounterRead

Synopsis

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

int CounterRead(int 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 error conditions have prevented 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.)
3. For a mechanical meter, the total of all increment calls made to Paylink is stored and returned by this call.

ReadCounterCaption

Synopsis

The **ReadCounterCaption** call is used to determine the caption for the specified counter

C++

char* CounterCaption(int CounterNo);

C#, VB, Java

string CounterCaption(int 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. If available, 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 (int 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 1 second followed by the value for 2 seconds, for the counter corresponding to the positive form of the code

Return Value

None

Remarks

1. This result of this call with a negative parameter is undefined if the counter has an associated caption.
2. Whenever the meter displayed is changed, the caption (if set) is unconditionally displayed for one second.
3. This is not relevant for mechanical meters.

MeterStatus

Synopsis

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

```
int 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

1. For Mechanical Meters, METER_OK is returned for correctly defined outputs. Paylink has no way of detecting whether anything is actually connected to the outputs.

MeterSerialNo

Synopsis

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

```
int 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 Paylink unit 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 application to clear all the *internal* E²PROM memory on the card. This is the area where the Paylink system keeps the value in / value out counters, the configuration information, etc.

```
void E2PromReset(int LockE2Prom);
```

Parameters

1. LockE2Prom
This is a 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 application to write to all or part of the user E²PROM on the card.

C++

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

C#, VB, Java


```
void E2PromWrite (Byte[] UserBuffer,  
                 int    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 application to obtain all or part of the user E²PROM from the card.

C++

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

C#, VB, Java

```
void E2PromRead (Byte[] UserBuffer,  
                 int    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.

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 allows the application to automatically close down currency acceptance in the event of any PC malfunction.

```
int CheckOperation(int Sequence,  
                  int 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()**.
3. From **1.11.3** onwards, the configuration file can specify an output related to this function. If ther outp is specfied then it is driven *only* when the **Timeout** is being checked, and has not expired.

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 in conjunction with an EventDetailBlock object.

Fields / properties for the EventDetailBlock Object

Field Type	Name	Meaning
int	EventCode	The code (the same as returned by NextEvent)
int	RawEvent	The actual code returned by the peripheral
int	DispenserEvent	Non zero if the device was a dispenser, zero for an acceptor
int	Index	The ReadxxxDetails index of the generating device

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

Parameters

1. 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

1. In the case where one or more events are missed, the code IMHEI_OVERFLOW will replace the missed events.
2. 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.
3. The values for the **EventCodes** returned are in the separate header file **ImheiEvent.h** (see Appendix 1)
4. 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.
ID-003	Response to "Status Poll"	The byte following a FAILURE response
CCNet	For Rejected , the reason byte else the response to "Status Poll"	The Code1 byte following a 47H response
MDB Bill Acceptor	Event byte from Poll response	See Table Below
MDB Changer		See Table Below

MDB Fault / Self Test Codes - These come from a changer device as two bytes and are packed into a single byte by the Milan code. The following interprets this byte:

Reported Byte Range	MDB Error Type	MDB Second byte
0x00->0x6F	0x11 - Discriminator	As Reported Byte
0x70->0x7F	0x10 - General Changer	Reported Byte - 0x70
0x80->0xCF	0x12 - Accept Gate	Reported Byte - 0x80
0xE0->0xEE	0x13 - Separator	(Reported Byte - 0xE0) * 2
0xEF	0x14	N/A
0xF0->0xFF	0x15	Reported Byte - 0xF0

NextAcceptorEvent

NextDispenserEvent

NextSystemEvent

Synopsis

These calls provide controlled access to exactly the same set of events as the **NextEvent** call described above.

The difference is that, rather than providing access to one single queue with all events, these provide access to a number of queues. One independent queue is provided for *each* acceptor in the system, one for each dispenser in the system, and one, final queue for all system oriented events.

```
int NextAcceptorEvent(int Number,
                     EventDetailBlock* EventDetail);
int NextDispenserEvent(int Number,
                      EventDetailBlock* EventDetail);

int NextSystemEvent(EventDetailBlock* EventDetail);
```

Parameters

1. Number
The same value as that used in a call to ReadxxxDetails. All events returned will have an Index value equal to this.
2. 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

Remarks

1. If these calls are used in a system that also calls **NextEvent**, the result is undefined.
2. Systems with more than 32 acceptors or dispensers should not use these calls.
3. Un-accessed queues will silently discard events.

SetDeviceKey

Synopsis

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

Note that this does **not** store the encryption key into the peripheral, it merely notifies Paylink of the key to use for communication.

```
void SetDeviceKey (int InterfaceNo,  
                  int Address,  
                  int 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 cctalk BNV acceptor at address 40 on 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 Paylink device.

```
int 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.

C++

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

C#, VB

```
int FirmwareVersion (ref string CompileDate,  
                    ref string CompileTime);
```

Java

```
Int FirmwareVersion (string[] CompileDate,  
                    string[] CompileTime);
```

Parameters

1. CompileDate
This is a 16 byte string that receives a printable version of the date on which the firmware was created.
2. CompileTime
This is a 16 byte string that receives a printable version of the time at which the firmware was created.

Return Values

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

Remarks

In 'C', 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.

int 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.)

C++

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

C#, VB, Java

```
string 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 usually a totally unrecoverable situation.
It is expected that a production 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 application. 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.)

Events fall into two categories, notifications and faults. Notifications are just that, the incoming information is passed along to the application.

The fact of a fault on the other hand is remembered by Paylink, and when the fault clears, a NOW_OK “fault” event will be generated.

A specific bit in the event code is reserved for indicating fault events.

Event Codes used by NextEvent / EventDetailBlock

Event codes have an internal structure, allowing categorizations. The bottom 6 bits are the unique event classification code, fault related codes have bit 5 set and otherwise overlap these events code.

More significant bits describe the type of unit affected.

For details of the exact makeup of the values of these codes, users are referred to the ImheiEvent.h header file.

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 events categorised as **OUTPUT_PROBLEM**, **JAM** & **INTERNAL_PROBLEM**, are also reported as self test faults on some acceptors. They are therefore automatically latched as faults (without sending the self test fault) and hence a **NOW_OK** “fault” is generated when they clear.

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 note, this is queued as an **NOTE_DISPENSER_EVENT** event, with the actual event byte reported in **RawEvent**.

The events categorised as MISAREAD, JAM & INTERNAL_PROBLEM, are also reported as self test faults on some acceptors. They are therefore automatically latched as faults (without sending the self test fault) and hence a NOW_OK "fault" is generated when they clear.

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

cctalk hopper processing

This is divided into two parts, the processing associate with reporting the ongoing ability of a functioning hopper to pay out coins, and the that associated with checking that the hopper is operational.

Both of these require a "Test Hopper" command to be sent to the unit, but the reporting mechanism is different.

The ongoing ability to pay out is reported as the Status field in the dispenser block, the results of the regular check are reported as "self test" events.

Note: that when a Payout is issued the results of the "self Test" are ignored - the dispense coins command is sent to the hopper regardless.

On a regular basis the "Test Hopper" command is sent to the each hopper and the result evaluated. After start-up, and regularly thereafter, a **IMHEI_COIN_DISPENSER_NOW_OK** is reported if there are no errors.

The defined return from this command is a string of up to 4 bytes (depending upon the exact unit) with one (or theoretically more) bits set to indicate the problem.

The action of Paylink is to regard these bytes as containing 32 bits. The bits are classified by this section of Paylink as an Error, a Fraud attempt, a Payout result or "information only". Paylink scans along these bits looking for the first Error or Fraud bit that is non-zero. Other bits are ignored.

The bit number of this first bit (i.e. a number between 0 to 31) is then returned in **RawEvent** and an **EventCode** of either **IMHEI_COIN_DISPENSER_FRAUD_ATTEMPT** or **IMHEI_COIN_DISPENSER_REPORTED_FAULT**

For reference, the bit numbers, and their classification are:

Bit Number	Meaning	Event Classification	Payout Result
0	Jammed	Information only	PAY_JAMMED
1	Empty	Information only	PAY_EMPTY
2	Reversed	Information only	
3	Idle fraud blocked	Fraud	PAY_FRAUD
4	Idle fraud short	Fraud	PAY_FRAUD
5	Payout blocked	Information only	PAY_FAILED_BLOCKED
6	Power up	Information only	
7	Disabled	Fault	
8	Fraud short	Fraud	PAY_FRAUD
9	Single coin mode	Fault	
10	Checksum a	Fault	
11	Checksum b	Fault	
12	Checksum c	Fault	
13	Checksum d	Fault	
14	Pwr fail during write	Fault	
15	Pin locked	Fault	
16	Powerdown during payout	Information only	
17	Unknown coin type paid	Fault	
18	Pin number incorrect	Fault	
19	Incorrect cipher key	Fault	
20	Unused	Information only	
21	Unused	Information only	
22	Unused	Information only	
23	Unused	Information only	
24	Unused	Information only	
25	Unused	Information only	
26	Unused	Information only	
27	Unused	Information only	
28	Unused	Information only	
29	Unused	Information only	
30	Use other hopper	Information only	PAY_NOT_EXACT
31	Opto fraud	Fraud	PAY_FRAUD

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

DES security (25-12-1)

Background

With the increasing sophistication of fraudulent activity, in 2010 Money Controls introduced a published scheme for using DES encryption in conjunction with a system of random keys to provide very high security on ccTalk peripherals.

In order to use these high security peripherals to their full, an enhanced Paylink has been produced, known as DES Paylink. As a part of its enhancements, this includes a special “DES Button” accessible through a small hole in the case. This DES Button provides important functionality in handling the engineer configuration of a DES system.

Key Exchange

A DES Key is a string of 8 bytes. Details on the DES system are given in Wikipedia and elsewhere.

All DES peripherals have special facility for entering PC exchange mode, so that as a part of the engineer configuration of a DES system the ccTalk peripherals can be asked to generate a new, random, DES key and to return this to the Paylink.

By design, Paylink does not normally query peripherals for their DES key. If the key stored by Paylink for a peripheral is wrong, the application is notified and the green LED flashes at double speed. If a peripheral that has been identified as having a wrong key has generated a new key, then pressing the DES Button on the Paylink will cause Paylink to retrieve this new key and to store it for future use.

Application notification of a peripheral awaiting a valid key is by the bit:
ACCEPTOR_NO_KEY in the status field for acceptors and the value:
PAY_NO_KEY is the status field for hoppers.

Des Lock

A DES system involving Paylink can be basically secured in one of two ways:

1. The PC and Paylink are both in the same secure enclosure.
Here there is no need to provide any security control over the USB connection - access to the USB cable is equivalent to access to the hard disc of the PC, and this level of access cannot be countered by electronic means.
2. The Paylink, or more particularly the USB connection, is accessible from the general cabinet area.
Here a fraud attempt is possible by removing existing USB cable and connecting the Paylink USB socket to the fraudster's laptop.

To prevent this security problem, the PC application can use DES lock. The functions associated with DES lock are described in this section.

(Only) While Paylink is DES locked:

- The PC and Paylink cross check that each other are using the same key.
- The Payout call only works if the key has been matched
- New DES peripherals can not be added without Paylink being first unlocked
- Pressing the DES button deletes all DES keys (peripheral and Paylink) and unlocks Paylink

Some points about the DES Lock system are:

- Updates to existing Paylink applications are optional - although there can be a security risk.
- The DES lock key is provided by the PC, and so can be held on a read only disk system.
- A DES lock aware application will spot if a different Paylink is substituted, **or** if the Paylink is unlocked in order to change the peripherals

DESSetKey

Synopsis

Inform the DLL that of the current key that is to be shared between the PC and Paylink

```
void DESSetKey (char Key[8]) ;
```

Parameters

1. Key
The 8 byte DES key previously applied using the **DESLockSet** function.

Return Value

None.

Remarks

1. The Key should be as unpredictable as possible. Ideally, it will be a random number generated by the application and saved for future use. For system with read only file systems, it could be derived from Processor ID or similar.
2. The **DESStatus** function (see below) will enable the application to determine the success of this function.

DESLockSet

Synopsis

Apply the lock using the key quoted in this function call.

```
void DESLockSet (char Key[8]);
```

Parameters

1. Key
The 8 byte DES key chosen by the PC.

Return Value

None

Remarks

If the Paylink is already DES Locked, then this function will not change the key unless **DESSetKey** has already matched the key stored by Paylink.

DESLockClear

Synopsis

Clear any previous applied DES lock.

```
void DESLockSet (void);
```

Parameters

None

Return Value

None

Remarks

3. If the Paylink is already DES Locked, then this function will not unlock Paylink unless **DESSetKey** has already matched the key stored by Paylink.
4. This function differs from pressing the DES button in that keys for the existing DES peripherals are not lost. This can therefore be used by application when an engineer wishes to only update a single peripheral.

DESStatus

Synopsis

The DESStatus call provides the current status of the DES lock system.

```
int DESStatus (void) ;
```

Parameters

None

Return Values.

Value	Meaning	Mnemonic
0	The Paylink is unlocked	DES_UNLOCKED
1	DES Key matched by Paylink and PC	DES_MATCH
-1	Not a DES Paylink	DES_NOT
-2	Paylink wrong key	DES_WRONG
-3	DES Key checking is still being performed.	DES_CHECKING
-4	DES Lock is being applied	DES_APPLYING

Remarks

1. Following a call to **DESLockSet**, or **DESSetKey**, the programmer should poll this to check the operation.
2. The Paylink system is only operation when either DES_UNLOCKED or DES_MATCH has been returned by this function.

Engineering Support

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

They are included here for completeness, but can be ignored if you are just interfacing application 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.

C++

```
void WriteInterfaceBlock (int    Interface,
                          void*  Block,
                          int    Length) ;
```

C#, VB, Java

```
void WriteInterfaceBlock (int    Interface,
                          Byte[] Block,
                          int    Length) ;
```

Parameters

1. Interface

The sequence 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 Paylink.

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**.

C++

```
int ReadInterfaceBlock (int    Interface,
                       void*  Block,
                       int    Length) ;
```

C#, VB, Java

```
int ReadInterfaceBlock (int    Interface,
                       Byte[] Block,
                       int    Length) ;
```

Parameters

1. Interface
The sequence 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